EDICE

PROGRAMOVÁNÍ

PRO KAŽDÉHO UŽIVATELE

mistrovství v C++

stephen prata

Od základů k pokročilému objektovému programování

Platné pro Windows, Unix i Linux

Výklad podle posledního standardu ANSI/ISO C++

Popisuje knihovnu STL

Srozumitelný výklad, demonstrativní příklady

Kontrolní otázky a odpovědi

computer press

VŠECHNY CESTY K INFORMACÍM

SAMS

Stručný obsah

KAPITOLA 1: Začínáme ... 1
KAPITOLA 2: Vydáváme se do C++ ... 17
KAPITOLA 3: Práce s daty ... 49
KAPITOLA 4: Odvozené typy ... 89
KAPITOLA 5: Cykly a relační výrazy ... 145
KAPITOLA 6: Příkazy větvení a logické operátory ... 193
KAPITOLA 7: Funkce – programové moduly v C++ ... 227
KAPITOLA 8: Příběhy ve funkcích ... 277
KAPITOLA 9: Objekty a třídy ... 351
KAPITOLA 10: Práce se třídami ... 399
KAPITOLA 11: Třídy a dynamické přidělování paměti ... 451
KAPITOLA 12: Dědičnost tříd ... 511
KAPITOLA 13: Znovupoužitelný kód v jazyce C++ ... 567
KAPITOLA 14: Přátelé, výjimky a další ... 637
KAPITOLA 15: Třída STRING a standardní knihovna šablon ... 699
KAPITOLA 16: Vstup, výstup a soubory ... 775
DODATEK A: Základy číselných soustav ... 857
DODATEK B: Klíčová slova jazyka C++ ... 861
DODATEK C: Znaková sada ASCII ... 863
DODATEK D: Priorita operátorů ... 867
DODATEK E: Ostatní operátory ... 871
DODATEK F: Šablonová třída STRING ... 881
DODATEK G: Metody a funkce knihovny STL ... 899
DODATEK H: Vybraná literatura ... 927
DODATEK I: Konverze na ANSI/ISO standard C++ ... 929
DODATEK J: Odpovědi na opakovací otázky ... 937
REJSTŘÍK ... 957

Obsah

KAPITOLA 1:
Začínáme ... 1

Učíme se C++ ... 1
Trochu historie ... 2
Jazyk C ... 3
Filozofie programování v C ... 3
Objektově orientované programování ... 4
Generické programování ... 6
C++ ... 6
Přenositelnost a standardy ... 7
Mechanizmus vytváření programu ... 9
Vytvoření zdrojového kódu ... 10
Kompilace a sestavení ... 11
Kompilace a sestavování pod Unixem ... 11
Turbo C++ 2.0, Borland C++ 3.1 (DOS) ... 12
Kompilátory Windows ... 13
Kompilátory pro Macintosh ... 14
Konvence použité v této knize ... 15
Náš systém ... 16

KAPITOLA 2:
Vydáváme se do C++ ... 17

Zahájení C++ ... 17
Funkce main() ... 19
Hlavička funkce jako rozhraní ... 20
Proč main() s jiným jménem není to samé ... 21
Komentáře v C++ ... 22
Preprocesor C++ a soubor iostream ... 23
Jména hlavičkových souborů ... 24
Prostory jmen ... 24
Výstup C++ pomocí cout ... 25
Znak nového řádku (\n) ... 27
Formátování zdrojového kódu C++ ... 28
Styl zdrojového kódu C++ ... 29
Více o příkazech C++ ... 29

Deklarační příkazy a proměnné 30
Přiřazovací příkaz 32
Nový trik pro cout 32
Další příkazy C++ 33
Použití cin 34
Více o cout 34
Dotek třídy 35
Funkce 36
Použití funkce s návratovou hodnotou 37
Varianty funkcí 40
Uživatelsky definované funkce 41
Tvar funkce 42
Hlavičky funkcí 43
Uživatelsky definované funkce s návratovou hodnotou 44
Shrnutí příkazů 46
Shrnutí 47
Opakovací otázky 47
Programovací cvičení 48

KAPITOLA 3:
Práce s daty 49

Jednoduché proměnné 49
Jména proměnných 50
Celočíselné typy 51
Celočíselné typy short, int a long 51
Poznámky k programu 54
Typy bez znaménka 56
Jaký typ použít 58
Celočíselné konstanty 59
Jak C++ určuje typ konstanty 60
Typ char: Znaky a malá celá čísla 61
Poznámky k programu 63
Členská funkce: cout.put() 63
Konstanty char 64
Char se znaménkem a bez znaménka 66
Když potřebujete větší velikost: wchar_t 67
Nový typ bool 68
Kvalifikátor const 68
Čísla v pohyblivé řádové čárce 69
Zápis čísel v pohyblivé řádové čárce 70
Typy pohyblivé řádové čárky 71

Poznámky k programu 73
Konstanty v pohyblivé řádové čárce 74
Výhody a nevýhody pohyblivé řádové čárky 74
Aritmetické operátory v C++ 75
Jaké pořadí: Priorita operátorů a asociativita 77
Odchylky při dělení 78
Operátor modulo (zbytek po celočíselném dělení) 79
Konverze typů 80
Konverze při přiřazení 81
Konverze ve výrazech 82
Konverze při předávání parametrů 83
Přetypování 84
Shrnutí 85
Opakovací otázky 86
Programovací cvičení 87

KAPITOLA 4:
Odvozené typy 89

Úvod do polí 89
Poznámky k programu 92
Více o inicializaci pole 93
Řetězce 94
Spojování řetězců 95
Použití řetězců v poli 96
Poznámky k programu 97
Dobrodružství na vstupu řetězce 97
Řádkově orientovaný vstup: getline() a get() 99
Prázdné řádky a další problémy 102
Míchání řetězcového a numerického vstupu 103
Úvod do struktur 104
Poznámky k programu 107
Další vlastnosti struktury 108
Pole struktur 110
Bitová pole 110
Uniony 111
Výčtové typy 112
Nastavení hodnot enumerátorů 114
Rozsahy hodnot pro výčty 114
Ukazatele a volná paměť 115
Deklarování a inicializace ukazatelů 118
Ukazatele a čísla 121

Alokace paměti pomocí operátoru new 121
Poznámky k programu 123
Uvolnění paměti pomocí operátoru delete 124
Vytváření dynamických polí pomocí operátoru new 124
Vytvoření dynamického pole pomocí operátoru new 125
Použití dynamického pole 126
Ukazatele, pole a aritmetika ukazatelů 128
Poznámky k programu 129
Shrnutí vlastností ukazatelů 131
Ukazatele a řetězce 132
Poznámky k programu 134
Vytváření dynamických struktur pomocí operátoru new 137
Příklad na použití new a delete 139
Poznámky k programu 140
Automatická, statická a volná paměť 141
Automatické proměnné 141
Statická paměť 141
Volná paměť 141
Shrnutí 142
Opakovací otázky 143
Programovací cvičení 144

KAPITOLA 5:
Cykly a relační výrazy 145

Úvod do cyklu for 145
Části cyklu for 147
Výrazy a příkazy 149
Nevýrazy a příkazy 151
Přizpůsobování pravidel 152
Zpět do cyklu for 153
Poznámky k programu 154
Změna velikosti kroku 155
Procházení řetězců pomocí cyklu for 155
Operátory inkrementování (++) a dekrementování (—) 156
Sdružování přiřazovacích operátorů 158
Složené příkazy neboli bloky 159
Operátor čárka (neboli více syntaktických triků) 160
Poznámky k programu 161
Lahůdky operátoru čárka 162
Relační výrazy 163
Chyba, kterou možná děláte 164
Porovnání řetězců 166

Poznámky k programu 168
Cyklus while 169
Poznámky k programu 170
for oproti while 171
Čekání 173
Cyklus do while 175
Cykly a vstup textu 176
Použití nepřizdobeného cin pro vstup 177
Poznámky k programu 177
cin.get(char) vede k vysvobození 178
Kterou cin.get()? 179
Podmínka konce souboru 180
Konec souboru končí vstup 182
Běžná spojení 182
Ještě další cin.get() 183
Vnořené cykly a dvourozměrná pole 186
Inicializace dvourozměrného pole 188
Shrnutí 189
Opakovací otázky 190
Programovací cvičení 191

KAPITOLA 6:
Příkazy větvení a logické operátory 193

Příkaz if 193
Příkaz if else 195
Formátování vašich příkazů if else 197
Konstrukce if else if else 198
Logické výrazy 199
Logický operátor NEBO: || 199
Logický operátor A: && 201
Poznámky k programu 203
Nastavení rozmezí pomocí && 203
Poznámky k programu 204
Logický operátor NE: ! 205
Poznámky k programu 206
Fakta o logických operátorech 207
Knihovna cctype obsahující funkce pro práci se znaky 208
Operátor ?: 210
Příkaz switch 211
Použití enumerátorů jako návěští 215

Příkazy switch a if else ... 216
Příkazy break a continue ... 217
Poznámky k programu ... 218
Cykly, které čtou čísla ... 219
Poznámky k programu ... 222
Shrnutí ... 222
Opakovací otázky ... 223
Programovací cvičení ... 225

KAPITOLA 7: Funkce – programové moduly v C++ ... 227

Přehled funkcí ... 227
Definování funkce ... 228
Vytváření funkčních prototypů a volání funkcí ... 230
Proč prototypy? ... 232
Syntaxe prototypu ... 232
Co pro vás prototypy dělají ... 233
Parametry funkcí a předávání hodnotou ... 234
Násobné parametry ... 235
Poznámky k programu ... 237
Jiná funkce se dvěma parametry ... 237
Poznámky k programu ... 240
Funkce a pole ... 240
Pole a ukazatele (znovu) ... 241
Důsledky použití polí jako parametrů ... 242
Poznámky k programu ... 244
Více funkcí, které pracují s poli ... 244
Naplnění pole ... 245
Zobrazení pole a jeho ochrana pomocí const ... 246
Modifikace pole ... 247
Sestavení částí dohromady ... 247
Poznámky k programu ... 249
Ukazatele a const ... 249
Funkce a řetězce ve stylu jazyka C ... 252
Poznámky k programu ... 253
Funkce, které navracejí řetězce ... 254
Poznámky k programu ... 255
Funkce a struktury ... 256
Předání a navrácení struktur ... 256
Další příklad ... 258
Poznámky k programu ... 262

Předání adres struktur ... 263
Rekurze ... 265
Poznámky k programu ... 267
Ukazatele na funkce ... 267
Základy ukazatelů na funkce ... 268
Získání adresy funkce ... 268
Deklarování ukazatele na funkci ... 268
Použití ukazatelů na vyvolání funkce ... 269
Shrnutí ... 271
Opakovací otázky ... 272
Programovací cvičení ... 273

KAPITOLA 8:
Příběhy ve funkcích ... 277

Vložené funkce ... 277
Referenční proměnné ... 280
Vytvoření proměnné reference ... 280
Reference jako funkční parametry ... 283
Poznámky k programu ... 286
Vlastnosti a zvláštnosti referencí ... 287
parametrparametr ... 287
Dočasné proměnné, referenční parametry a konstanty ... 288
Použití referencí se strukturou ... 290
parametrparametr ... 290
Poznámky k programu ... 292
Úvahy spojené s navrácením reference nebo ukazatele ... 294
Kdy používat parametry typu odkaz ... 294
Standardní parametry ... 295
Poznámky k programu ... 297
Polymorfismus funkcí (přetěžování funkcí) ... 298
Příklad přetížení ... 300
Kdy používat přetěžování funkcí ... 303
Šablony funkcí ... 303
Přetížené šablony ... 306
Explicitní specializace ... 308
Přístup podle první generace ... 309
Druhá generace ... 309
Třetí generace ... 310
Příklad ... 311
Konkretizace a specializace ... 313
Která funkce? ... 314
Přesné shody a nejlepší shody ... 315

Funkce s více parametry ... 317
Oddělená kompilace ... 317
Paměťové třídy, rozsah platnosti a vazba ... 321
Rozsah platnosti a vazba ... 322
Automatické proměnné ... 322
Automatické proměnné a zásobník ... 325
Proměnné typu register ... 326
Třída paměti static ... 327
Externí proměnné ... 328
Poznámky k programu ... 329
Modifikátor static (lokální proměnné) ... 330
Vazba a externí proměnné ... 333
Kvalifikátory paměťové třídy: const, volatile a mutable ... 336
Více o const ... 337
Paměťové třídy a funkce ... 338
Jazyková vazba ... 339
Paměťové třídy a dynamická alokace ... 339
Prostory jmen ... 340
Tradiční prostory jmen v C++ ... 340
Nové vlastnosti prostoru jmen ... 342
Using-deklarace a using-direktivy ... 343
Další vlastnosti prostoru jmen ... 345
Nepojmenované prostory jmen ... 347
Prostory jmen a budoucnost ... 347
Shrnutí ... 347
Opakovací otázky ... 349
Programovací cvičení ... 350

KAPITOLA 9:
Objekty a třídy ... 351
Procedurální a objektově orientované programování ... 354
Abstrakce a třídy ... 355
Co je typ ... 355
Třída ... 356
Veřejné nebo privátní? ... 359
Implementace členských funkcí třídy ... 360
Poznámky ke členským funkcím ... 362
Vložené metody ... 363
Který objekt? ... 364
Použití třídy ... 365
Dosavadní poznatky ... 367
Konstruktory a destruktory třídy ... 369

Deklarace a definice konstruktorů 370
Použití konstruktoru 370
Implicitní konstruktor 371
Destruktory 373
Vylepšení třídy Stock 373
Hlavičkový soubor 374
Implementační soubor 375
Klientský soubor 377
Poznámky k programu 378
Konstantní členské funkce 379
Opakování konstruktorů a destruktorů 379
Poznáváme objekty: ukazatel this 381
Pole objektů 386
Rozsah platnosti třídy 389
Abstraktní datový typ 391
Shrnutí 395
Opakovací otázky 396
Programovací cvičení 397

KAPITOLA 10:
Práce se třídami 399

Přetěžování operátorů 400
Zpracování času 401
Přidání operátoru sčítání 404
Omezení přetížení 406
Další přetížené operátory 408
Úvod k přátelům 409
Vytváření spřátelených funkcí 410
Obecný druh přítele: přetížení operátoru 412
První verze přetížení operátoru 412
Druhá verze přetížení operátoru 413
Přetížené operátory: členské funkce a nečlenské funkce 417
Další přetěžování – třída Vector 418
Použití stavové položky 425
Další přetěžování 427
Násobení 428
Další vylepšení: přetížení přetíženého operátoru 428
Komentář k implementaci 429
Použití třídy Vector na problém náhodné chůze 430
Poznámky k programu 432
Automatické konverze a přetypování tříd 433

Poznámky k programu 438
Konverzní funkce 439
Použití automatické typové konverze 441
Konverze a přátelé 443
Možnost výběru 445
Shrnutí 446
Opakovací otázky 447
Programovací cvičení 448

KAPITOLA 11: Třídy a dynamické přidělování paměti 451

Dynamická paměť a třídy 452
Příklad na zopakování a statické položky tříd 452
Poznámky k programu 459
Znovu operátory new a delete 461
Potíže s třídou String 462
Implicitní členské funkce 464
Implicitní konstruktor 465
Kopírovací konstruktor 465
Kde jsme udělali chybu 467
Operátor přiřazení 470
Kde jsme udělali chybu 471
Oprava přiřazení 471
Nová, vylepšená třída String 472
Kdy se v konstruktoru používá operátor new 479
Používání ukazatelů na objekty 481
Opakování technik 485
Přetížení operátoru 485
Konverzní funkce 485
Třídy s konstruktory používajícími operátor new 485
Simulace fronty 486
Třída Queue 487
Rozhraní 487
Implementace 488
Metody třídy 490
Ostatní metody třídy 494
Třída Customer 496
Simulace 499
Shrnutí 504
Opakovací otázky 505
Programovací cvičení 507

KAPITOLA 12: Dědičnost tříd 511

Jednoduchá základní třída 512
Dědičnost – vztah je 515
Deklarace odvozené třídy 517
Implementace odvozené třídy 520
Inicializace objektů pomocí objektů 523
Ostatní členské funkce 523
Poznámky k programu 527
Řízení přístupu – chráněný režim 528
Vztah je, reference a ukazatele 529
Virtuální členské funkce 531
Aktivace dynamické vazby 532
Proč dva druhy vazeb 536
Jak virtuální funkce pracují 536
Co je potřeba znát o virtuálních funkcích 538
Konstruktory 538
Destruktory 538
Přátelé 539
Neprovedení předefinování 539
Předefinování skrývá metody 539
Dědičnost a přiřazení 540
Smíšené přiřazení 541
Přiřazení a dynamické přidělení paměti 543
Případ první – odvozená třída nepoužívá operátor new 545
Případ druhý – odvozená třída používá operátor new 545
Abstraktní základní třídy 550
Opakování návrhu tříd 552
Členské funkce generované kompilátorem 553
Implicitní konstruktor 553
Kopírovací konstruktor 553
Operátor přiřazení 554
Další metody třídy 554
Konstruktory 554
Destruktory 554
Konverze 555
Předávání objektu hodnotou a předávání reference 556
Vrácení objektu a vrácení reference 556
Použití modifikátoru const 557
Poznámky k veřejné dědičnosti 557
Vztah je 558
Co není součástí dědictví 558

Operátor přiřazení 558
Privátní metody a metody chráněné 560
Virtuální metody 560
Destruktory 561
Shrnutí funkcí třídy 561
Shrnutí 562
Opakovací otázky 562
Cvičení 564

KAPITOLA 13: Znovupoužitelný kód v jazyce C++ 567

Třídy s objekty jako položkami 567
Třída ArrayDb 569
Vylepšení operátoru [] 571
Alternativa s modifikátorem const 571
Příklad třídy Student 575
Inicializace obsažených objektů 576
Použití rozhraní pro obsažený objekt 577
Použití nové třídy 578
Soukromá dědičnost 579
Příklad třídy Student (nová verze) 580
Inicializace komponent základní třídy 580
Používání metod základních tříd 581
Použití upravené třídy Student 584
Kompozice nebo soukromá dědičnost? 585
Chráněná dědičnost 586
Změna přístupu pomocí klíčového slova using 586
Šablony tříd 588
Definice šablony třídy 588
Použití šablony třídy 591
Bližší pohled na šablonu třídy 594
Nesprávné použití zásobníku ukazatelů 594
Správné použití zásobníku ukazatelů 595
Poznámky k programu 599
Příklad šablony pole a netypové parametry 599
Použití šablony u skupiny tříd 602
Poznámky k programu 608
Univerzálnost šablon 608
Specializace šablon 611
Implicitní instance 611
Explicitní instance 611
Explicitní specializace 611
Částečné specializace 612

Vícenásobná dědičnost 613
Kolik tříd Worker? 614
Nová pravidla pro konstruktory 617
Která metoda? 618
Smíšené virtuální a nevirtuální základní třídy 626
Virtuální základní třídy a dominance 627
Přehled vícenásobné dědičnosti 628
Shrnutí 628
Opakovací otázky 631
Programovací cvičení 632

KAPITOLA 14:
Přátelé, výjimky a další 637

Přátelé 637
Spřátelené třídy 638
Spřátelené členské funkce 642
Jiné přátelské vztahy 646
Sdílení přátelé 646
Šablony a přátelé 647
Vnořené třídy 649
Vnořené třídy a přístup k nim 650
Rozsah platnosti 651
Řízení přístupu 652
Vnořování do šablony 652
Výjimky 655
Poznámky k programu 658
Mechanismus výjimek 658
Poznámky k programu 660
Univerzálnost výjimek 661
Vícenásobné pokusné bloky 664
Uvolnění zásobníku 665
Další možnosti 667
Výjimky a třídy 668
Výjimky a dědičnost 674
Třída exception 679
Výjimka bad_alloc a operátor new 680
Když výjimky bloudí 681
Opatření při používání výjimek 683
RTTI 685
K čemu je RTTI dobré 685
Jak RTTI pracuje 685

Operátor dynamic_cast 686
Operátor typeid a třída type_info 690
Špatné použití RTTI 693
Operátory přetypování 694
Shrnutí 696
Opakovací otázky 697
Programovací cvičení 698

KAPITOLA 15: Třída STRING a standardní knihovna šablon 699

Třída string 699
Vytvoření řetězce 700
Poznámky k programu 702
Vstup třídy string 704
Práce s řetězci 705
Poznámky k programu 709
Co dál 710
Třída auto_ptr 711
Použití třídy auto_ptr 712
Poznámky ke třídě auto_ptr 714
Standardní knihovna šablon 715
Šablonová třída vector 716
Co se dá s vektory dělat 718
Další možnosti práce s vektory 723
Generické programování 727
K čemu jsou potřeba iterátory 728
Druhy iterátorů 731
Vstupní iterátor 732
Výstupní iterátor 733
Dopředný iterátor 733
Obousměrný iterátor 733
Iterátor přímého přístupu 733
Hierarchie iterátorů 734
Koncepty, vylepšení a modely 735
Ukazatel jako iterátor 735
Funkce copy() a šablony ostream_iterator a istream_iterator 736
Ostatní užitečné iterátory 737
Druhy kontejnerů 741
Koncept kontejneru 742
Sekvence 744
Vektor (vector) 745
Obousměrná fronta (deque) 746

Seznam (list) ... 746
Fronta (queue) ... 749
Fronta s prioritou (priority_queue) ... 750
Zásobník (stack) ... 750
Asociativní kontejnery ... 751
Příklad s kontejnerem set ... 752
Příklad s kontejnerem multimap ... 755
Funkční objekty (aka funktory) ... 757
Koncepty funktorů ... 758
Předdefinované funktory ... 760
Přizpůsobivé funktory a funkční adaptéry ... 762
Algoritmy ... 764
Skupiny algoritmů ... 764
Všeobecné vlastnosti ... 765
Použití standardní knihovny šablon ... 766
Ostatní knihovny ... 770
Shrnutí ... 770
Opakovací otázky ... 772
Programovací cvičení ... 773

KAPITOLA 16:
Vstup, výstup a soubory ... 775

Přehled vstupu a výstupu ... 776
Proudy a vyrovnávací paměti ... 776
Proudy, vyrovnávací paměti a soubor iostream ... 778
Přesměrování ... 781
Výstup pomocí objektu cout ... 782
Přetížený operátor << ... 782
Výstup a ukazatele ... 784
Řetězení výstupu ... 784
Ostatní metody třídy ostream ... 785
Vyprázdnění výstupní vyrovnávací paměti ... 788
Formátování pomocí objektu cout ... 789
Změna číselného základu používaného při zobrazení ... 790
Úprava šířky polí ... 792
Výplňové znaky ... 794
Nastavení přesnosti zobrazování hodnoty s pohyblivou řádovou čárkou ... 794
Tisk koncových nul a desetinných oddělovačů ... 795
Další informace o metodě setf() ... 796
Standardní manipulátory ... 802
Hlavičkový soubor iomanip ... 803
Vstup pomocí objektu cin ... 805

Jak pohlíží výraz cin >> na vstupní proud 807
Stavy proudů 808
Nastavení stavů 809
Vstup, výstup a výjimky 810
Účinky proudových stavů 810
Další metody třídy istream 812
Jednoznakový vstup 812
Který tvar použít pro vstup znaku 815
Vstup řetězce: metody getline(), get() a ignore() 816
Neočekávaný vstup řetězce 818
Další metody třídy istream 819
Poznámky k programu 821
Vstup ze souboru a výstup do souboru 823
Jednoduchý vstup ze souboru a výstup do souboru 824
Otevření více souborů 827
Zpracování příkazového řádku 827
Kontrola stavu proudu a metoda is_open() 830
Režimy souboru 831
Přidání dat do souboru 834
Binární soubory 836
Přímý přístup 841
Formátování incore 848
Co dále 851
Shrnutí 851
Opakovací otázky 853
Programovací cvičení 854

DODATEK A:
Základy číselných soustav 857
Čísla v osmičkové soustavě 857
Čísla v šestnáctkové soustavě 858
Čísla ve dvojkové soustavě 858
Dvojková a šestnáctková soustava 859

DODATEK B:
Klíčová slova jazyka C++ 861

DODATEK C:
Znaková sada ASCII 863

DODATEK D:
Priorita operátorů 867

DODATEK E:
Ostatní operátory 871

Bitové operátory 871
Operátory posunu 871
Bitové logické operátory 873
Několik běžných technik s bitovými operátory 875
Zapnutí bitu 876
Přepnutí bitu 876
Vypnutí bitu 876
Test hodnoty bitu 876
Operátory dereferencování členů 877

DODATEK F:
Šablonová třída STRING 881

Třináct typů a jedna konstanta 882
Datové informace, konstruktory a další 883
Implicitní konstruktor 885
Konstruktor používající pole 885
Konstruktor používající část pole 885
Kopírovací konstruktor 886
Konstruktor používající n kopií znaku 887
Konstruktor pracující s rozsahem 887
Různé metody pro práci s pamětí 888
Přístup k řetězci 888
Základní přiřazení 889
Prohledávání řetězce 890
Skupina metod find() 890
Skupina metod rfind() 890
Skupina metod find_first_of() 891
Skupina metod find_last_of() 891
Skupina metod find_first_not_of() 892
Skupina metod find_last_not_of() 892
Metody a funkce pro porovnání řetězců 893
Modifikátory třídy string 894
Připojení a přidání 894
Další přiřazení 895
Metody vkládání 895

Metody odstraňující znaky 896
Metody pro nahrazování 896
Další upravující metody: copy() a swap() 897
Výstup a vstup 897

DODATEK G: Metody a funkce knihovny STL 899

Položky společné všem kontejnerům 899
Další položky kontejnerů vector, list a deque 901
Další položky kontejnerů set a map 903
Funkce knihovny STL 905
Nemodifikující sekvenční operace 905
Změnové sekvenční operace 909
Třídicí a relační operace 915
Třídění 917
Binární hledání 918
Slučování 919
Množinové operace 920
Operace pracující s haldou 922
Funkce Minimum a Maximum 923
Změny pořadí 924
Numerické operace 925

DODATEK H: Vybraná literatura 927

DODATEK I: Konverze na ANSI/ISO standard C++ 929

Direktivy preprocesoru 929
Definujte konstanty pomocí modifikátoru const místo direktivy #define 929
Definujte krátké funkce pomocí klíčového slova inline místo direktivy #define 931
Používání funkčních prototypů 932
Přetypování 932
Seznamte se s vlastnostmi C++ 933
Používejte nové uspořádání hlavičkových souborů 933
Používejte prostory jmen 933
Používejte šablonu autoptr 934
Používejte třídu string 935
Používejte knihovnu STL 935

DODATEK J: Odpovědi na opakovací otázky 937

Kapitola 2 937
Kapitola 3 937
Kapitola 4 938
Kapitola 5 939
Kapitola 6 940
Kapitola 7 941
Kapitola 8 943
Kapitola 9 945
Kapitola 10 947
Kapitola 11 948
Kapitola 12 950
Kapitola 13 951
Kapitola 14 952
Kapitola 15 953
Kapitola 16 955

REJSTŘÍK 957

O autorovi

Stephen Prata vyučuje astronomii, fyziku a počítačovou vědu na univerzitě Marin v kalifornském Kentfieldu. Titul B.S. (Bachelor of Science) získal na California Institute of Technology a titul Ph.D. na University of California, Berkeley. Je autorem nebo spoluautorem více než dvanácti knih pro vydavatelství The Waite Group, včetně knih *Artificial Life Playhouse* a *Certified Course in Visual Basic 4*. Pro vydavatelství napsal také knihu *New C Primer Plus*, která získala v roce 1990 od asociace Computer Press Association cenu za nejlepší učebnici a kniha *Mistrovství v C++* byla na tutéž cenu nominována v roce 1991.

Věnování

Mým kolegů a studentům z univerzity v Marin, se kterými je radost pracovat. – Stephen Prata.

Poděkování

Poděkování ke třetímu vydání

Chci poděkovat redaktorům z vydavatelství Macmillan Publishing a Waite Group za role, které sehráli při sestavování této knihy: Tracymu Dunkelbergerovi, Susan Waltonové a Andree Rosenbergové. Děkuji také Russi Jacobsovi za jeho obsahovou a technickou úpravu. Ze společnosti Metrowerks bych chtěl poděkovat Davu Markovi, Alexi Harperovi a zvláště Ronu Liechtymu za pomoc a spolupráci.

Poděkování ke druhému vydání

Chci poděkovat Mitchellu Waitovi a Scottu Calamarovi za podporu druhého vydání a Joelu Fugazzottovi a Joann Millerové, kteří dovedli projekt do konce. Děkuji Michaelu Marcottymu ze společnosti Metrowerks, který zodpověděl mé otázky týkající se beta verze kompilátoru CodeWarrior. Také chci poděkovat následujícím instruktorům, kteří věnovali svůj čas posouzení prvního vydání. Jsou jimi Jeff Buckwalter, Earl Brynner, Mike Holland, Andy Yao, Larry Sanders, Shahin Momtazi a Don Stephens. Nakonec chci poděkovat Heidi Brumbaughové za pomoc při úpravě obsahu a revizi materiálu.

Poděkování k prvnímu vydání

Zvláště chci poděkovat Mitchi Waitovi za jeho práci při vývoji, formování a úpravě této knihy a za revizi rukopisu. Velmi si cením práce Harryho Hendersona při revizi několika posledních kapitol a při testování programů kompilátorem Zortech C++. Děkuji Davidu Gerroldovi za revizi celého rukopisu a za propagaci potřeb méně zkušených čtenářů. Také děkuji Hanku Shiffmanovi za testování programů pomocí Sun C++ a Kentu Williamsovi za testování programů pomocí AT&T cfront a G++. Děkuji Nan Borresonové ze společnosti Borland International za citlivou a ochotnou spolupráci při testování pomocí Turbo C++ a Borland C++. Děkuji Ruth Myersové a Christine Bushové za neúnavné vyřizování záplavy dokumentů týkajících se tohoto projektu. Nakonec děkuji Scottu Calamarovi, který na vše dohlížel.

Předmluva k prvnímu vydání

Když v roce 1984 vydavatelství Waite Groupe poprvé vydalo knihu *C Primer Plus*, používal se jazyk C již asi deset let, ale byl teprve na začátku rozmachu. Jsme pyšní na důležitou roli, kterou naše kniha sehrála při uvádění programátorů do tohoto jazyka. Dnes dosáhl jazyk C++ odvozený od jazyka C podobného stupně ve svém vývoji. Je v rozkvětu, protože nabízí nové paradigma – objektově orientované programování neboli OOP, které naprosto vyhovuje moderním programovacím potřebám. Společnost AT&T tedy přepisuje operační systém UNIX do C++, protože C++ zlepšuje spolehlivost, udržování a zno-

vupoužitelnost kódu. Ze stejných důvodů vyvíjí společnost Apple pomocí jazyka C++ software pro svou řadu počítačů Macintosh a protože OOP techniky jsou přirozené pro takové programovací prvky, jako jsou okna a dialogová okna. Jednotliví programátoři se obracejí k jazyku C++, protože jeho nové vlastnosti vracejí do programování vzrušení. Je tedy přirozeně na čase vydat knihu *Mistrovství v C++* a k tomuto rozkvětu přispět.

Mezi současností a dobou minulou je jeden rozdíl – o jazyce C++ bylo napsáno mnohem více knih než o jazyce C, když byl nový. Žádná z těchto nových knih o C++ však nehraje takovou roli jako tato kniha od vydavatelství Waite Group. Mnoho knih o C++ předpokládá, že již znáte jazyk C a že ho znáte dobře. To příliš nepomáhá těm, kteří chtějí k C++ přejít například od Pascalu nebo BASICu nebo těm, kteří se jazyku C věnovali rekreačně a nestali se v něm odborníky. Většina dalších knih o C++ předkládá úplný popis jazyka, nejen nové prvky, ale stále předpokládá, že máte dostatečné znalosti jazyka C a programování všeobecně. Některé knihy o C++ jsou výbornými příručkami, ale jako učebnice tohoto jazyka se příliš nehodí. Několik titulů o C++ již vyrazilo na zteč. Pouze se však dotýkají několika nových kapitol ve srovnání s jazykem C, ale nový materiál plně nezačleňují ani nevystihují nové vzrušující rysy související s objektově orientovaným programováním.

Vezměte si knihu od vydavatelství Waite Group. My nepředpokládáme, že znáte jazyk C a základní jazyk C začleňujeme současně při rozebírání rysů jazyka C++. Předpokládáme sice, že již máte určité zkušenosti s programováním, ale základy nevynecháváme. Jazyk C++ jsme se snažili v knize předložit v souladu s tradičními přednostmi našeho vydavatelství:

- Kniha má být snadná na používání a má sloužit jako přívětivý průvodce.
- Kniha nepředpokládá, že jste již seznámeni se všemi relevantními programovacími pojmy.
- Kniha klade důraz na pohodlné učení pomocí stručných, snadno napsatelných příkladů, pomocí kterých máte za určitou dobu pochopit jeden až dva pojmy.
- Kniha objasňuje pojmy pomocí ilustrací.
- Kniha obsahuje cvičení na test vašich znalostí a je vhodná jak pro samostudium, tak i pro učení ve skupině.

Kniha *Mistrovství v C++* předkládá základy jazyka C++ a ilustruje je pomocí krátkých, věcných programů, které lze snadno přepsat a se kterými lze experimentovat. Jejím cílem není encyklopedický popis všech rysů a nuancí jazyka C++, ale předkládá nejdůležitější aspekty a základy ponechává pro další studium. Dozvíte se o vstupu a výstupu, jak napsat programy pro opakované úkoly a pro výběr z nabídky, naučíte se mnoho způsobů zpracování dat a používání funkcí. Dozvíte se o důležitých pojmech objektově orientovaného programování jako je skrývání dat (je to velmi zábavné), polymorfismus (není to tak zlé, jak to vypadá) a dědičnost. Kromě základních programovacích technik se naučíte filozofii objektově orientovaného programování. Uděláme vše pro to, aby tato prezentace byla krátká, jednoduchá a zábavná. Naším cílem je, abyste na konci byli schopni psát solidní, efektivní programy a dobře se přitom bavili.

Poznámka pro instruktory

Jedním z cílů třetího vydání je nabídnout knihu, kterou bude možné používat jako učebnici pro samostudium i jako normální učebnici. Uvádíme několik rysů, které budou užitečné při používání třetího vydání knihy jako učebnice:

- Tato kniha popisuje obecné rysy jazyka C++ a nezávisí tedy na nějaké určité implementaci.
- Obsah se řídí standardem výboru pro ISO/ANSI C++ a zahrnuje rozebírání šablon, standardní knihovny šablon, třídy string, výjimek, RTTI a prostoru jmen.
- Kniha nepředpokládá předchozí znalost jazyka C a mohou ji tedy používat i ti, kteří se s tímto jazykem nesetkali. (Určité programovací zkušenosti jsou však žádoucí.)
- Témata jsou uspořádána tak, aby úvodní kapitoly bylo možné projít rychle jako opakování pro ty, kteří již mají znalosti jazyka C.
- Kapitoly obsahují opakovací otázky a programovací cvičení.
- Kniha uvádí několik témat, která jsou vhodná pro kursy počítačových věd, jako jsou abstraktní datové typy, zásobníky, fronty, jednoduché seznamy, simulace, generické programování a používání rekurze při implementování strategie rozděl a panuj.
- Většina kapitol je dostatečně krátkých a lze je probrat do jednoho týdne.
- Kniha se zabývá tím, *kdy* určité prvky použít a *jak* je použít. Veřejnou dědičnost například spojuje se vztahem *je*, zatímco kompozici a soukromou dědičnost se vztahem *má*. Také rozebírá, kdy je a kdy není vhodné používat virtuální funkce.

Jak je tato kniha uspořádána

Tato kniha je rozdělena do 16 kapitol a 10 příloh, které jsou níže shrnuty.

Kapitola 1: Začínáme

Tato kapitola se zabývá tím, jak Bjarne Stroustrup vytvořil programovací jazyk C++, tím, že do jazyka C přidal podporu objektově orientovaného programování. Dozvíte se o rozdílech mezi procedurálními jazyky, jakým je například jazyk C, a jazyky objektově orientovanými, jakým je C++. Přečtete si o společném úsilí ANSI/ISO při vývoji standardu C++. Kapitola se zabývá mechanismy vytvoření programu v C++ a naznačuje přístup několika běžných kompilátorů C++. Nakonec jsou popsány konvence používané v této knize.

Kapitola 2: Vydáváme se do C++

Kapitola 2 vás provede procesem vytváření jednoduchých programů v C++. Dozvíte se o roli funkce **main()** a o některých typech příkazů používaných v programech napsaných v C++. Budete používat předdefinované objekty **cin** a **cout** pro programy pracující se vstupem a výstupem a naučíte se vytvářet a používat různé proměnné. Nakonec budete uvedeni do funkcí, programovacích modulů C++.

Kapitola 3: Práce s daty

Jazyk C++ obsahuje vestavěné typy pro ukládání dvou druhů dat: celých čísel (čísel bez zlomkové části) a čísel s plovoucí řádovou čárkou (čísel se zlomkovou částí). Pro uspokojení různých potřeb programátorů nabízí jazyk C++ pro každou kategorii několik typů. Tato kapitola uvedené typy rozebírá a také se zabývá vytvářením proměnných a psaní konstant různých typů. Také se dozvíte, jak jazyk C++ postupuje při implicitních a explicitních konverzích z jednoho typu na druhý.

Kapitola 4: Odvozené typy

Jazyk C++ umožňuje vytvořit typy, které jsou propracovanější než typy vestavěné. Nejdokonalejší formou je třída, probíraná v kapitolách 9, 10, 11, 12 a 13. Tato kapitola rozebírá jiné formy včetně polí, které mohou obsahovat několik hodnot jednoho typu, struktur, které mohou obsahovat několik hodnot různých typů, a ukazatelů, které identifikují místo uložení v paměti. Také se naučíte vytvářet a ukládat textové řetězce a zpracovávat vstup a výstup. Nakonec se dozvíte o několika způsobech, jak C++ postupuje při přidělování paměti a rovněž o operátorech **new** a **delete**, kterými se paměť spravuje explicitně.

Kapitola 5: Cykly a relační výrazy

Programy musí často provádět některé činnosti opakovaně a jazyk C++ nabízí za tímto účelem tři cyklické struktury: **for**, **while** a **do while**. Tyto cykly potřebují vědět, kdy mají skončit a relační operátory C++ vám umožňují vytvářet testy cyklů. Také se naučíte vytvářet cykly, které čtou vstup a zpracovávají ho znak po znaku. Nakonec se naučíte vytvářet dvourozměrná pole a zpracovávat je pomocí vnořených cyklů.

Kapitola 6: Příkazy větvení a logické operátory

Programy se dokážou chovat inteligentně, pokud své chování dokážou přizpůsobit okolnostem. V této kapitole se naučíte řídit tok programu pomocí příkazů **if**, **if else** a **switch** a pomocí podmíněného operátoru. Naučíte se, jak pomocí logických operátorů vyjádřit rozhodovací testy. Rovněž se setkáte s knihovnou funkcí **cctype** pro vyhodnocování znakových relací, například testování, zda je znak číslem nebo netisknutelným znakem.

Kapitola 7: Funkce – programové moduly v C++

Funkce jsou základními stavebními kameny programování v jazyce C++. Tato kapitola se soustředí na ty rysy, které mají funkce v C++ společné s funkcemi v jazyce C. Především si zopakujete všeobecný formát definice funkce a prozkoumáte, jak funkční prototypy zvyšují spolehlivost programů. Také se naučíte psát funkce pro zpracování polí, znakových řetězců a struktur. Dále se dozvíte o rekurzi, což je případ, kdy funkce volá sebe samu a zjistíte, jak ji lze implementovat při použití strategie rozděl a panuj. Nakonec se setkáte s ukazateli na funkce, s jejichž pomocí můžete pomocí parametru jedné funkci říct, jakou druhou funkci má použít.

Kapitola 8: Příběhy ve funkcích

Tato kapitola odkrývá nové rysy, které jazyk C++ přidává do funkcí. Dozvíte se o vložených funkcích, které mohou urychlit provádění programu za cenu jeho velikosti. Budete pracovat s referenčními proměnnými, které představují alternativní způsob předávání informací funkcím. Implicitní parametry umožňují funkci automaticky doplnit hodnoty do parametrů, které jsou při volání funkce vynechány. Přetěžování funkcí umožňuje vytvářet funkce se stejným názvem, ale s odlišným seznamem parametrů. Všechny tyto rysy se často využívají při návrhu třídy. Také se dozvíte o funkčních šablonách, které umožňují specifikovat návrh skupiny příbuzných funkcí. Naučíte se sestavovat vícesouborové programy. Nakonec prozkoumáte paměťové třídy, rozsah platnosti, spojování a prostory jmen, které určují, jaké části programu budou o určité proměnné vědět.

Kapitola 9: Objekty a třídy

Třída je uživatelem definovaný typ a objekt je instance třídy jako proměnná. Tato kapitola vás uvede do objektově orientovaného programování a návrhu tříd. Deklarace třídy popisuje informace uložené v objektu třídy a rovněž operace (metody třídy), které jsou pro objekty třídy povolené. Některé části objektu jsou viditelné vnějšímu světu (veřejná část), zatímco některé jsou skryty (soukromá část). Speciální metody třídy (konstruktory a destruktory) vstupují do hry při vytváření a rušení objektů. O všech těchto a jiných podrobnostech třídy se dozvíte v této kapitole a uvidíte, jak lze pomocí tříd implementovat takové abstraktní datové typy (ADT), jako například zásobník.

Kapitola 10: Práce se třídami

V této kapitole se dozvíte více o třídách. Nejdříve se naučíte přetěžovat operátory, což vám umožní definovat operátory jako + pro práci s objekty tříd. Dozvíte se o spřátelených funkcích, které mohou přistupovat k datům, která jsou pro okolní svět nepřístupná. Uvidíte, jak určité konstruktory a přetížené operátory členských funkcí mohou být použity pro konverzi do třídních typů a zpět.

Kapitola 11: Třídy a dynamické přidělování paměti

Často je užitečné, aby člen třídy ukazoval do dynamicky přidělené paměti. Jestliže použijete pro přidělení dynamické paměti v konstruktoru operátor **new**, zavazujete se dodat vhodný destruktor, definovat explicitní kopírovací konstruktor a definovat explicitní operátor přiřazení. Tato kapitola vám ukáže jak a rozebírá chování členských funkcí generovaných implicitně v případě, že explicitní definice nedodáte. Také si své znalosti rozšíříte o zkušenosti se třídami při použití ukazatelů na objekty a studiem problému simulování fronty.

Kapitola 12: Dědičnost tříd

Jedním z nejmocnějších rysů objektově orientovaného programování je dědičnost, díky níž odvozená třída zdědí vlastnosti třídy základní a tak získáte možnost znovupoužití kódu základní třídy. Tato kapitola rozebírá veřejnou dědičnost, která modeluje vztah *je*, což znamená, že odvozený objekt je speciálním případem objektu základního. Fyzik je například speciálním případem vědce. Při implementování vztahů *je* je potřeba nový druh členské funkce, které se říká virtuální funkce. Tato kapitola se danou problematikou zabývá a ukazuje, kdy je a kdy není vhodná veřejná dědičnost.

Kapitola 13: Znovupoužitelný kód v jazyce C++

Veřejná dědičnost představuje pouze jeden ze způsobů znovupoužití kódu. Tato kapitola se dívá i na několik dalších způsobů. Kompozice je případ, kdy jedna třída obsahuje členy, které jsou objekty jiné třídy. Lze ji použít pro modelování vztahu *má*, kdy jedna třída obsahuje složky třídy jiné. Například automobil má motor. Tyto vztahy můžete také modelovat pomocí soukromé a chráněné dědičnosti. Tato kapitola ukazuje jak a zdůrazňuje rozdíly mezi rozdílnými přístupy. Také se dozvíte o třídních šablonách, které vám umožňují definovat třídu podle nějakého nespecifikovaného generického typu a potom podle této šablony vytvořit konkrétní třídy podle určitých typů. Například zásobníková šablona umožňuje vytvořit zásobník celých čísel nebo zásobník řetězců. Nakonec se dozvíte o vícenásobné veřejné dědičnosti, na jejímž základě může být třída odvozena od více než jedné třídy.

Kapitola 14: Přátelé, výjimky a další

Tato kapitola rozšiřuje diskusi o přátelích o spřátelené třídy a spřátelené členské funkce. Potom předkládá několik nových prvků ve vývoji C++ počínaje výjimkami, které nabízejí mechanismus pro zpracování neobvyklých událostí v programu jako jsou například neodpovídající hodnoty v parametrech funkcí nebo nedostatek paměti. Dále se dozvíte o RTTI (routine type information), což je mechanismus pro identifikování typů objektu. Nakonec se dozvíte o bezpečnějších alternativách k neomezenému přetypování.

Kapitola 15: Třída string a standardní knihovna šablon

Tato kapitola rozebírá některé užitečné knihovny tříd, které byly do jazyka přidány nedávno. Třída **string** je vhodnou a mocnou alternativou tradičních řetězců ve stylu jazyka C. Třída **auto_ptr** pomáhá spravovat dynamicky přidělenou paměť. Standardní knihovna šablon (Standard Template Library, STL) obsahuje několik generických kontejnerů včetně šablonových reprezentací polí, front, seznamů, množin a map. Obsahuje také efektivní knihovnu generických algoritmů, které lze použít u kontejnerů knihovny STL a rovněž u obyčejných polí.

Kapitola 16: Vstup, výstup a soubory

Tato kapitola se ohlíží za vstupy a výstupy v jazyce C++ a rozebírá formátování výstupu. Dozvíte se, jak pomocí metod tříd určit stav vstupního nebo výstupního proudu a také například uvidíte, jestliže na vstupu došlo k neshodě typů nebo byl detekován konec souboru. Jazyk C++ odvozuje třídy pro správu vstupních a výstupních souborů pomocí dědičnosti. Naučíte se, jak otevřít soubory pro vstup a výstup, jak do souboru přidat data, jak používat binární soubory a jak přistupovat k souboru přímo. Nakonec se dozvíte, jak použít standardní vstupně-výstupní metody pro čtení ze řetězců a zápis do nich.

Příloha A: Základy číselných soustav

Tato příloha rozebírá čísla v osmičkové, šestnáctkové a dvojkové soustavě.

Příloha B: Klíčová slova jazyka C++

Příloha uvádí seznam klíčových slov jazyka C++.

Příloha C: Znaková sada ASCII

Tato příloha uvádí seznam znakové sady ASCII společně s desítkovými, osmičkovými, šestnáctkovými a dvojkovými reprezentacemi.

Příloha D: Priorita operátorů

Tato příloha uvádí seznam operátorů v C++ podle jejich priority v sestupném pořadí.

Příloha E: Další operátory

Tato příloha shrnuje takové operátory jazyka C++, jako jsou operátory bitové a také operátory, které nebyly probrány v základní části této knihy.

Příloha F: Šablonová třída string

Tato příloha shrnuje metody a funkce třídy **string**.

Příloha G: Metody a funkce knihovny STL

Tato příloha shrnuje kontejnerové metody knihovny STL a funkce všeobecných algoritmů z této knihovny.

Příloha H: Vybraná literatura

Tato příloha uvádí seznam knih, které vám dále pomohou pochopit jazyk C++.

Příloha I: Konverze na ANSI/ISO standard C++

Tato příloha nabízí vodítka pro převod z jazyka C a starších implementací jazyka C++ na standard.

Příloha J: Odpovědi na opakovací otázky

Tato příloha obsahuje odpovědi na opakovací otázky uvedené na konci každé kapitoly.

KAPITOLA 1

Začínáme

Témata kapitoly:

- Jak C++ přidává jazyku C pojmy objektově orientovaného programování
- Jak C++ přidává jazyku C pojmy programování pomocí předloh
- Historie a filosofie C
- Procedurální proti objektově orientovanému programování
- Historie a filosofie C++
- Standardy programovacího jazyka
- Mechanismus vytváření programu
- Konvence použité v knize

Vítáme vás v jazyce C++! Tento vzrušující programovací jazyk, který smíchává jazyk C s podporou objektově orientovaného programování, se stal jedním z nejdůležitějších programovacích jazyků devadesátých let dvacátého století a slibuje, že bude výrazně pokračovat po roce 2000. Jeho předchůdce, jazyk C, přináší do jazyka C++ tradici efektivity, ucelený, rychlý a přenositelný jazyk. Jeho objektově orientované dědictví přináší C++ novou metodiku programování, která je navržena tak, aby si poradila se stupňováním obtížnosti moderních úkolů programování. Jeho nově rozšířené vlastnosti šablon přinášejí ještě další metodiku programování, všeobecně použitelné programování. Toto trojnásobné dědictví je jak božím požehnáním, tak zdrojem záhuby. Dělá jazyk velmi výkonným, ale také znamená, že se musíte více učit.

V této kapitole blíže prozkoumáme pozadí C++, potom projdeme několik základních pravidel vytváření programů v C++. Zbytek knihy vás naučí používat jazyk C++ od skromných základů jazyka až po slávu objektově orientovaného programování (OOP) a jeho podpůrnou formu nového slangu – objektů, tříd, skrývání dat, mnohotvárnosti a dědičnosti, po podporu programování pomocí předloh. (Samozřejmě, jak se budete učit C++, tyto výrazy budou transformovány z módních slov do nezbytného slovníku kultivovaného projevu.)

Učíme se C++

C++ spojuje tři oddělené programovací tradice: tradici procedurálního jazyka reprezentovanou C; tradici objektově orientovaného jazyka reprezentovanou rozšířením C o třídy C++; programování pomocí předloh podporované šablonami jazyka C++. Tato kapitola krátce nahlédne do těchto tradic. Ale nejprve uvažujme o tom, co z tohoto dědictví vyplývá při učení C++. Jeden důvod pro použití C++ je

použít jeho objektově orientované rysy. Abyste tak učinili, potřebujete pevný základ ve standardu C, pro tento jazyk poskytuje základní typy, operátory, řídící struktury a syntaktická pravidla. Tedy pokud již znáte C, jste připraveni učit se C++. Ale to není pouze otázka naučení několika klíčových slov a konstrukcí. Přechod z C na C++ zahrnuje především asi tolik práce, jako kdybyste se učili jazyku C. Tedy pokud znáte C, jakmile přecházíte na C++, musíte se odnaučit některým programovacím zvyklostem. Pokud neznáte C a chcete se naučit C++, musíte zvládnout prvky C, prvky OOP a programování pomocí předloh, ale možná se alespoň nemusíte odnaučovat programovacím návykům. Jestliže začínáte uvažovat o tom, že učení C++ může zahrnovat z vaší strany jisté úsilí namáhání mysli, máte pravdu. Tato kniha vás provede procesem jasným a nápomocným způsobem, krok po kroku tak, že namáhání mysli bude přiměřeně pozvolné, aby zanechalo váš mozek pružným.

C++ Primer Plus se blíží k C++ tím, že učí jak základy C, tak nové prvky, takže kniha nepředpokládá, že máte nějaké předběžné znalosti C. Začnete výukou společných rysů, které C++ sdílí s C. Dokonce i když znáte C možná zjistíte, že tato část knihy je dobrým přehledem. Také zdůrazňuje způsob, který se stane později důležitým a ukazuje, v čem se C++ liší od C. Až budete dostatečně zběhlí v základech C, přidáte nadstavbu C++. V tomto bodě se začnete učit o objektech a třídách a o tom, jak je C++ zavádí. Budete se učit o šablonách.

Tato kniha se nepovažuje za úplný manuál C++; nezkoumá každé zákoutí nebo skulinu jazyka. Ale naučíte se všem hlavním rysům jazyka včetně těch, jako jsou šablony, výjimky a prostory jmen, což jsou současné dodatky.

Nyní se stručně podívejme trochu na základy C++.

Trochu historie

Výpočetní technika se rozvinula během několika posledních desetiletí úžasnou rychlostí. Dnes může laptop pracovat rychleji a ukládat více informací než sálový počítač před 40ti lety. (Pouze několik programátorů si může připomenout široké nabídky sad děrných štítků, které byly dodávány mohutnému, celý sál vyplňujícímu počítačovému systému s majestátními 100 KB paměti – nedostačující k provádění dnešní dobré hry pro osobní počítač.) Počítačové jazyky se také rozvinuly. Změny možná nejsou tak dramatické, ale jsou důležité. Větší a výkonnější počítače plodí větší a složitější programy, které na oplátku způsobují nové problémy ve správě a údržbě programů.

V 70tých letech jazyky C a Pascal pomohly zavést období strukturovaného programování, základní principy, které přinesly jistý řád a kázeň do oblasti, která silně potřebovala tyto vlastnosti. Kromě poskytnutí prostředků pro strukturované programování jazyk C také produkoval kompaktní rychle běžící programy spolu se schopností adresovat hardwarové záležitosti, jako jsou řídící komunikační porty a ovladače disků. Tyto dary pomohly C, aby se stal dominantním programovacím jazykem 80tých let. Mezitím 80tá léta svědčila růstu nových přístupů: objektově orientovanému programování, neboli OOP, které je zakotvené v jazycích Smalltalk a C++. Podívejme se na tyto dva směry vývoje (C a OOP) trochu blíže.

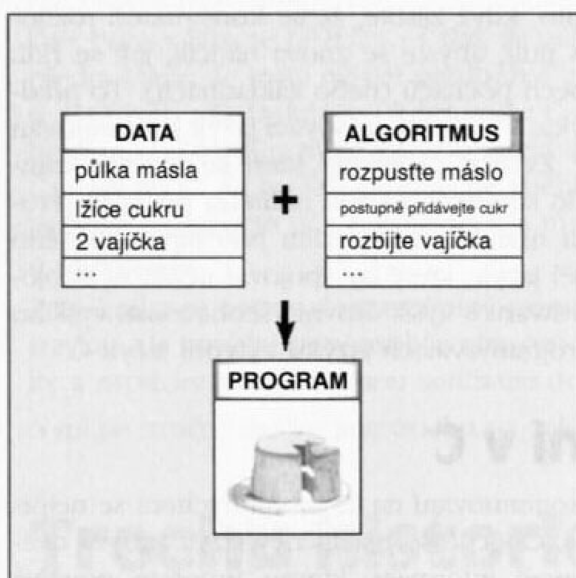
Jazyk C

Začátkem 70tých let Dennis Ritchie z Bell Laboratories pracoval na svém projektu rozvoje operačního systému UNIX. (Operační systém je sada programů, která spravuje počítačové zdroje a řídí vzájemné působení s uživatelem. Například je to operační systém, jenž vyšle na obrazovku systémovou výzvu a provede váš program.) Na tuto práci potřeboval Ritchie jazyk, který by byl stručný a výstižný, který by vytvářel kompaktní rychlé programy a jenž by mohl účinně řídit hardware. Tradičně se programátoři setkávají s těmito potřebami, když používají jazyk assembler, který je těsně svázán se strojovým kódem počítače. Avšak assembler je strojově orientovaný jazyk; to znamená, že je závislý na určitém procesoru počítače. Pokud tedy chcete přesunout program v assembleru na jiný druh počítače, možná musíte úplně program přepsat pomocí jiného assembleru. To by bylo trochu jako kdybyste si pokaždé koupili nové auto, když zjistíte, že se konstruktéři rozhodli změnit umístění ovládacích prvků, což vás nutí, abyste se znova naučili, jak se řídí. Ale UNIX byl určený pro práci na různých typech počítačů (nebo základnách). To předpokládalo použití vyššího programovacího jazyka. Vyšší programovací jazyk je orientován na řešení problému a nikoli na určitý hardware. Zvláštní programy, které se nazývají kompilátory, překládají vyšší programovací jazyk do strojového kódu určitého počítače. Proto můžete použít vyšší programovací jazyk na různých základnách použitím odlišného kompilátoru pro každou základnu. Ritchie chtěl jazyk, který by spojoval účinnost strojového jazyka na nízké úrovni s přístupem k hardwaru s vyšší úrovní všeobecnosti vyššího jazyka a přenositelnosti. Na základě starších programovacích jazyků vytvořil jazyk C.

Filozofie programování v C

Protože C++ roubuje nové základní principy programování na C, mohli bychom se nejprve podívat na starší principy, které C doprovází. Obecně se počítačový jazyk zabývá dvěma pojmy – daty a algoritmy. Data představují informaci, kterou program používá a zpracovává. Algoritmy jsou metody, které program používá (viz obrázek 1.1). Jazyk C, podobně jako dodnes většina jazyků, je procedurální jazyk. To znamená, že zdůrazňuje hledisko algoritmu programu. Pojmově procedurální programování sestává z vypočítávání činností, které by počítač mohl provádět, a potom používá programovací jazyk na uskutečnění těchto činností. Program předepisuje počítači množinu procedur, kterými se počítač řídí, aby provedl určitý výstup, téměř jako recept, který předepisuje množinu postupů, jimiž se řídí kuchař, aby vyrobil koláč.

Dřívější procedurální jazyky jako FORTRAN a BASIC narážely na organizační problémy, jakmile se programy zvětšovaly. Například programy často používaly příkazy skoku, které směrovaly provádění programu na jednu nebo druhou skupinu instrukcí v závislosti na výsledku jistých druhů testů. Mnoho starších programů mělo komplikovaný průběh (tzv. „špagetové programování"), prakticky nebylo možné porozumět programu čtením kódu a požadavek na změnu takového programu byl pobídkou ke katastrofě. Počítačoví vědci na to odpověděli tím, že vyvinuli ukázněnější styl programování, který se nazývá strukturovaným programováním. C zahrnuje rysy, které tento přístup usnadňují. Například strukturované programování omezuje skoky (rozhodnutí, která instrukce se má následně vykonat) na malou množinu způsobných konstrukcí. C obsahuje ve svém slovníku tyto konstrukce (cyklus `for`, cyklus `while`, cyklus `do while` a příkaz `if else`).



Obrázek 1.1 Data + algoritmy = program

Návrh seshora dolů byl dalším z nových postupů. Myšlenka spočívá v rozdělení velkého programu do menších, lépe zvladatelných úkolů. Je-li jeden z těchto úkolů stále příliš rozsáhlý, rozdělte ho do ještě menších úkolů. Pokračujte tímto způsobem dokud nebude program rozškatulkován do malých, jednoduše programovatelných modulů. (Uspořádejte si své studium. Ach! Uspořádejte si svůj pracovní stůl, stolní desku, kartotéku a poličky na knihy. Ach! Začněte pracovním stolem a uspořádejte každou zásuvku a začněte prostřední. Hm, snad mohu ten úkol zvládnout.) Návrh C tento přístup usnadňuje, podporuje vás, abyste navrhovali programové jednotky nazývané funkce, které představují jednotlivé moduly úlohy. Jak jste si mohli všimnout, postupy strukturovaného programování odrážejí postupný myšlenkový proces, jenž přemýšlí o programu v rámci činností, které provádí.

Objektově orientované programování

Ačkoli principy strukturovaného programování zlepšily srozumitelnost, spolehlivost a usnadnily údržbu programů, velká část programování stále ještě zůstává výzvou. *Objek-*

tově orientované programování (OOP) k této výzvě přináší nový přístup. Na rozdíl od procedurálního programování, které zdůrazňuje algoritmy, OOP zdůrazňuje data. Spíše než se pokoušet o to, aby problém vyhovoval procedurálnímu přístupu jazyka, OOP usiluje o to, aby jazyk vyhovoval problému. Myšlenka spočívá v návrhu datových tříd, které odpovídají základním rysům problému.

V C++ existuje slovo *class*, které popisuje specifikaci takové nové datové třídy a *objekt* je určitá datová struktura, která je konstruovaná podle tohoto plánu. Například třída by mohla popisovat obecné vlastnosti manažera společnosti (například jméno, titul, plat a neobvyklé schopnosti), zatímco objekt by mohl představovat určitého manažera (Guilford Sheppblat, viceprezident, 325 000$, ví, jak se používá soubor CONFIG.SYS). Obecně vzato, třída definuje všechny údaje, které se používají k reprezentaci objektu a činnosti, jež mohou být nad těmito údaji prováděny. Například předpokládejme, že jste vytvářeli počítačový kreslící program schopný namalovat čtyřúhelník. Mohli byste definovat třídu, která čtyřúhelník popisuje. Datová část specifikace by mohla zahrnovat takové věci, jako jsou umístění rohů, výška a šířka, barva a styl ohraničení, barvu a vzor použitý pro vyplnění čtyřúhelníka. Část činností specifikace by mohla zahrnovat metody pro posun čtyřúhelníka, změnu velikosti, otočení, změnu barev a vzorů a kopírování čtyřúhelníka na jiné místo. Když potom použijete svůj program, aby namaloval čtyřúhelník, vytvoří objekt podle specifikace třídy. Tento objekt bude obsahovat všechny datové hodnoty, které popisují čtyřúhelník, a můžete použít metody třídy, abyste čtyřúhelník modifikovali. Jestliže namalujete dva čtyřúhelníky, program vytvoří dva objekty, pro každý čtyřúhelník jeden.

Přístup OOP k programovému návrhu spočívá v prvé řadě v návrhu tříd, které přesně zachycují ty věci, se kterými program zachází. Například kreslící program může definovat třídy, které představují čtyřúhelníky, čáry, kruhy, štětce, pera a podobně. Připomínáme, že definice třídy zahrnuje popis přípustných operací pro každou třídu, jako je posun kruhu nebo otočení čáry. Potom pokračujete návrhem programu za použití objektů těchto tříd. Postup začínající z nejnižší organizační úrovně, jako jsou třídy, po nejvyšší úroveň, jako je návrh programu, se nazývá programování zdola nahoru.

Programování OOP je mnohem více než vázání dat a metod do definice třídy. Například OOP podporuje vytváření znovupoužitelného kódu, který případně může uspořit mnoho práce. Skrývání dat zabezpečuje data proti nevhodnému přístupu. Polymorfismus vás nechá vytvořit násobné definice operátorů a funkcí a spolu s programovací souvislostí určuje, jaká definice se má použít. Dědičnost vás nechá odvodit nové třídy ze starých. Jak můžete vidět, objektově orientované programování zavádí mnoho nových pojmů a vyžaduje jiný přístup k programování než procedurální programování. Místo soustředění na úkoly, se můžete soustředit na reprezentaci pojmů. Místo použití přístupu k programování shora dolů občas použijete přístup zdola nahoru. Kniha vás provede všemi těmito body se spoustou příkladů, které se dají jednoduše pochopit.

Návrh užitečné spolehlivé třídy může být těžkou úlohou. Naštěstí jazyky OOP umožňují do vašeho vlastního programování jednoduše začlenit existující třídy. Prodejci poskytují kolekce užitečných knihoven tříd, které zahrnují knihovny tříd navržených tak, že usnadňují vytváření programů pro prostředí jako je Windows nebo Macintosh. Jedna ze skutečných výhod C++ je, že vás nechá jednoduše znovupoužít a přizpůsobit existující dobře otestovaný kód.

Generické programování

Generické programování je ještě další programovací postup podporovaný C++. S OOP sdílí cíl vytváření jednoduššího kódu pro znovupoužití a postup abstrakce obecných představ. Zatímco OOP zdůrazňuje datové hledisko programování, programování pomocí šablon zdůrazňuje algoritmické hledisko. Ale jeho střed pozornosti je odlišný. OOP je prostředkem pro správu velkých projektů, zatímco programování pomocí předloh poskytuje prostředky pro provádění společných úloh jako jsou třídění dat a slučování seznamů. Pojem *všeobecný* znamená vytváření typově nezávislého kódu. Data v C++ se vyskytují v mnoha typech – celá čísla, čísla s desetinnou částí, znaky, řetězce znaků, uživatelsky definované složité struktury různých typů. Jestliže například chcete setřídit data těchto různých typů, obvykle musíte pro každý typ vytvořit odlišné třídící funkce. Programování pomocí předloh zahrnuje rozšíření jazyka, takže můžete napsat jedinou funkci pro všeobecný typ (tj. neurčený) a použít ji pro varianty různých typů. Šablony C++ poskytují mechanismus, který to provádí.

C++

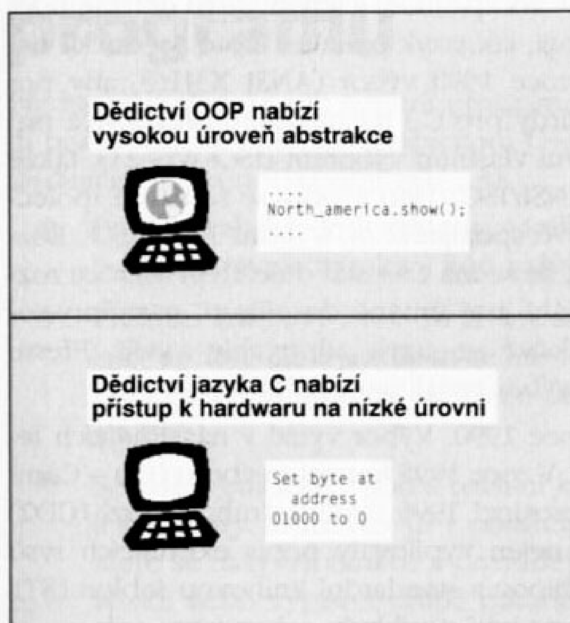
C++ začíná, podobně jako C, svůj život v Bell Labs, kde začátkem 80tých let vyvinul tento jazyk Bjarne Stroustrup. Podle jeho vlastních slov, „C++ byl původně navržen tak, abychom (mí přátelé a já) nemuseli programovat v assembleru nebo jiných moderních vyšších jazycích. Jeho hlavním účelem bylo udělat pro jednotlivého programátora jednodušší a příjemnější psaní dobrých programů" (Bjarne Stroustrup, *The C++ Programming Language*. Třetí vydání. Záznam MA: Addison-Wesley Publishing Company, 1997).

Stroustrup se více soustředil na vytvoření užitečného jazyka, než na prosazení určitých programovacích postupů nebo stylů. V určení jazykových rysů jsou skutečné programovací potřeby důležitější než teoretická čistota. Stroustrup založil C++ na stručnosti C, jeho vhodnosti pro systémové programování, široce rozšířené dostupnosti a jeho úzké vazby na operační systém UNIX. Hledisko OOP v C++ bylo inspirováno počítačovým simulačním jazykem, který se nazýval Simula67. Stroustrup přidal do C rysy OOP aniž by významně změnil složku C. C++ je tedy nadstavba C, znamenající, že každý platný program v C je také platným programem v C++. Existuje několik méně důležitých nesrovnalostí, ale nic rozhodujícího. Programy v C++ mohou používat softwarové knihovny C. *Knihovny* jsou kolekce programových modulů, které můžete volat z programu. Poskytují prověřená řešení mnoha programovacích problémů, proto vám šetří mnoho času a úsilí. To pomohlo rozšíření C++.

Jméno C++ pochází z přírůstkového operátoru C++, který přičítá k hodnotě proměnné 1. Jméno C++ přesně připomíná rozšířenou verzi C.

Počítačový program překládá problém skutečného života do řady činností, které mají být provedeny počítačem. Zatímco stránka OOP dává jazyku C++ schopnost uvést do spojitosti představy, které jsou zahrnuty v problému, C část C++ dává jazyku schopnost, aby se dostal blíže k hardwaru (viz obrázek 1.2). Toto spojení schopností rozšířilo C++. Možná také zahrne duševní posun v rychlosti, když přejdete z jednoho pohledu na program na jiný. (Vskutku, někteří puritáni OOP pohlížejí na rysy OOP přidané do C, jako na přidání křídel praseti, třebaže má sklon být výkonným žroutem.) Tedy, protože C++ roubu-

je OOP na C, můžete ignorovat objektově orientované rysy C++. Ale mnoho zmeškáte, když to bude všechno co uděláte.



Obrázek 1.2 Dualita C++

Jakmile C++ dosáhl jistého úspěchu, Stroustrup dodal šablony, umožňující programování pomocí šablon. Ale až teprve tehdy, když se začaly používat a rozšiřovat, začalo být zřejmé, že snad byly tak důležitým dodatkem jako OOP, dokonce, jak by někteří tvrdili, důležitějším. Skutečnost, že C++ zahrnuje jak OOP, tak programování pomocí šablon, ukazuje, že C++ zdůrazňuje užitkovost před ideovým přístupem, což je jedním z důvodů úspěchu jazyka.

Přenositelnost a standardy

Napsali jste v práci praktický program C++ pro starší 286 PC AT, když se vedení rozhodlo nahradit tento stroj pracovní stanicí Sun, počítačem, který používá jiný procesor a jiný operační systém. Můžete spustit svůj program na nové platformě? Samozřejmě musíte překompilovat program za použití kompilátoru, který je navržen pro novou platformu. Ale budete muset udělat nějaké změny do kódu, který jste napsali. Pokud můžete přeložit program bez provedení změn, a ten běží bez nesnází, říkáme, že je program přenositelný.

Pro přenositelnost existuje množství překážek, první z nich je hardware. Program, který je hardwarově závislý, pravděpodobně nebude přenositelný. Ten, kdo chce například přímo ovládat video kartu IBM PC VGA, tak si nebude vědět rady, pokud se to bude týkat Sunu. V této knize se tomuto druhu programování vyhneme. (The Waite Group's Object-Oriented Programming in C++, druhé vydání, Robert Lafore, se více dotýká hardwarově závislých sporných otázek.)

Druhou překážkou je jazyková odlišnost. Zajisté může nastat problém s mluveným jazykem. Popis denních událostí člověkem z Yorkshiru nebude přenositelný do Brooklynu,

i když se v obou oblastech mluví anglicky. Počítačové jazyky také mohou rozvíjet dialekty. Je implementace C++ na IBM PC stejná jako implementace na Sunu? Ačkoli by většina implementátorů ráda vytvořila své verze C++ kompatibilní s jinými, je to obtížné udělat bez zveřejnění standardů, které přesně popisují, jak jazyk pracuje. Proto Americký národní institut pro standardy (ANSI) vytvořil v roce 1990 výbor (ANSI X3J16), aby pro C++ vyvinul standard. (ANSI již vyvinulo standardy pro C.) Mezinárodní organizace pro standardy (ISO) brzy spojila tento proces se svým vlastním výborem (ISO-WG-21), takže na rozvoj standardů C++ existuje spojené úsilí ANSI/ISO. Tyto výbory se setkávají společně třikrát do roka a my je jednoduše celosvětově spojujeme do výboru ANSI/ISO. Rozhodnutí ANSI/ISO vytvořit standardy zdůrazňuje, že se má C++ stát důležitým a široce rozšířeným jazykem. Také to indikuje, že C++ dosáhl jisté úrovně dospělosti, navzdory tomu, že není produktivní zavádět standardy, když se jazyk tak rychle vyvíjí. Přesto C++ prodělal významné změny od té doby, co výbor začal svou práci.

Práce na standardech ANSI/ISO C++ začala v roce 1990. Výbor vydal v následujících letech několik prozatímních pracovních podkladů. V roce 1995 koncept výboru (CD – Committee Draft) k veřejnému připomínkování. V prosinci 1996 uvolnil druhou verzi (CD2) pro další veřejné posouzení. Tyto dokumenty nejen vypilovaly popis existujících rysů C++, ale také rozšířily jazyk o výjimky, RTTI, šablony a standardní knihovnu šablon (STL – Standard Template Library). Na prozatímní pracovní podklady a koncepty výboru se často odkazovalo jako na „standardní koncept" („standard draft"), ale první dokument, který dal právo k tomuto označení, je Final Draft International Standard (FDIS) vydaný v listopadu 1997. V době psaní této knihy se očekává, že má být v březnu 1998 schválen konečný Mezinárodní Standard (IS – International Standard). Tato kniha je primárně založena na CD2, který je velmi blízký FDIS. Rozdíly mezi FDIS a IS by mohly naproti tomu být v povaze korekcí, nikoli v dodatcích o změně rysů.

Standard ANSI/ISO C++ dodatečně dotahuje standard C, protože se předpokládá, dokud je to možné, že C++ je nadstavbou C. To znamená, jakýkoli platný program C by měl být platným programem C++. Mezi ANSI C a odpovídajícími pravidly C++ existuje několik rozdílů, ale jsou méně důležité. ANSI C zahrnuje některé rysy, které byly nejprve zavedeny do C++, jako jsou vytváření prototypů funkcí a kvalifikátor typu `const`.

Standard ANSI C nejen že definuje jazyk C, ale také definuje standardní knihovnu C, kterou musí implementace ANSI C podporovat. C++ tuto knihovnu také používá; tato kniha se na ni bude odkazovat jako na standardní knihovnu C nebo standardní knihovnu. Navíc výbor ANSI/ISO C++ poskytl standardní knihovnu tříd C++.

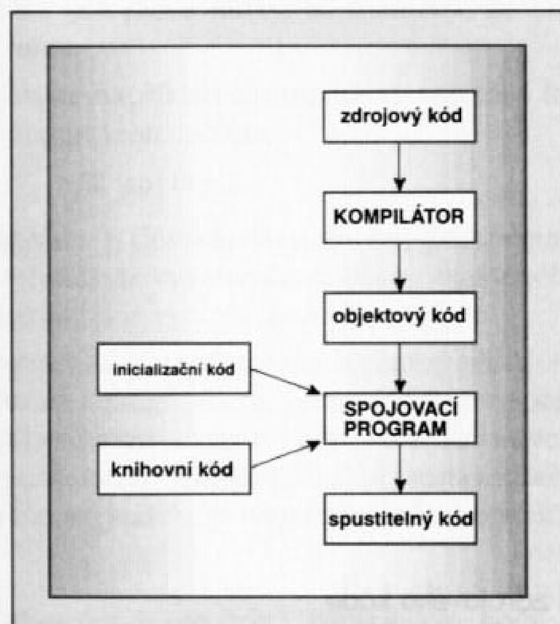
Než začal výbor ANSI/ISO pracovat, mnoho lidí přijalo nejnovější verzi standardu C++ od Bell Labs. Například kompilátor se sám mohl charakterizovat jako kompatibilní s vydáním C++ 2.0 nebo 3.0.

Než se řádně pustíme do jazyka C++, pojednáme trochu o podkladech o vytváření programů a o používání této knihy.

Mechanizmus vytváření programu

Předpokládejme, že jste napsali program. Jak ho spustíte? Přesné kroky závisí na prostředí počítače a na určitém kompilátoru C++, který používáte, ale ty se budou podobat následujícím krokům (viz obrázek 1.3):

- Použijte nějaký druh editoru, napište program a uložte ho do souboru. Tento soubor představuje zdrojový kód vašeho programu.
- Přeložte zdrojový kód. To znamená, že spustíte program, který překládá zdrojový kód do interního jazyka zvaného strojový jazyk, který se používá na hostitelském počítači. Soubor, který obsahuje přeložený program, je objektový kód vašeho programu.
- Sestavíte objektový kód s dalšími kódy. Programy C++ například obvykle používají *knihovny*. Knihovny C++ obsahují objektový kód pro kolekci počítačových rutin, které se nazývají funkce a provádějí třeba úlohy typu zobrazení informace na obrazovku nebo výpočet druhé odmocniny čísla. Sestavování spojí váš objektový kód s objektovým kódem funkcí, které používáte s jistým standardním startovacím kódem, což vytváří spustitelnou verzi vašeho programu. Program, který obsahuje tento výsledný produkt, se nazývá proveditelný kód.



Obrázek 1.3 Programové kroky

Všude v této knize narazíte na výraz zdrojový kód, tak se ujistěte, že jste ho zařadili do své vlastní paměti s přímým přístupem.

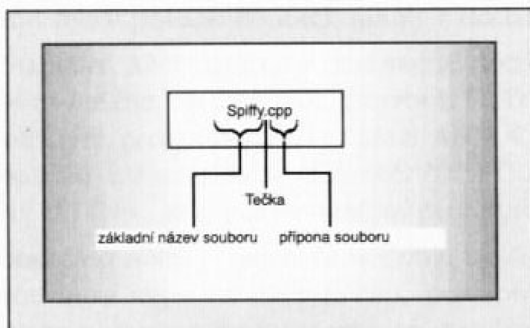
Programy v této knize jsou obecně použitelné a měly by běžet na libovolném systému, který podporuje moderní C++. (Avšak v době psaní mnoho kompilátorů nepodporuje

všechny rysy. Například jen některé podporují prostory jmen a novější vlastnosti šablon.) Kroky, které sestavují program, se mohou lišit. Podívejme se na tyto kroky trochu blíže.

Vytvoření zdrojového kódu

Některé implementace C++, jako jsou Microsoft Visual C++, Borland C++ (různé verze), Watcom C++, Symantec C++ a Metrowerks Code Warior, poskytují *integrovaná vývojová prostředí* (IDEs – *integrated development environments*), která vám umožňují z jednoho hlavního programu řídit všechny kroky vývoje programu včetně editace. Jiné implementace, jako jsou AT&T C++ nebo GNU C++ na Unixu nebo Linuxu, pouze ovládají etapy kompilace a sestavování a očekávají, že napíšete všechny příkazy v příkazové řádce systému. V takových případech můžete použít libovolný textový editor, pomocí kterého vytvoříte nebo modifikujete zdrojový kód. Například na Unixu můžete použít `vi` nebo `ed` nebo `ex` nebo `emacs`. V systému Dosu můžete použít `edlin` nebo `edit` nebo některý z dostupných editorů programu. Dokonce můžete použít libovolný textový editor za předpokladu, že soubor uložíte jako standardní ASCII dosový textový soubor namísto ve speciálním formátu textového editoru.

Při pojmenování zdrojového souboru musíte použít správný sufix, aby bylo zřejmé, že se jedná o soubor C++. Říká to nejen vám, že se jedná o soubor C++, ale také kompilátoru. (Když si vám kompilátor Unixu stěžuje na „chybné magické číslo“, jde právě o jeho roztomile nesrozumitelný způsob vyjádření toho, že jste použili chybný sufix.) Sufix se skládá z tečky následované znakem nebo skupinou znaků, který se nazývá přípona (viz obrázek 1.4).



Obrázek 1.4 Přípona zdrojového souboru

Tabulka 1.1 Přípona zdrojového souboru

Implementace C++	Přípona zdrojového kódu
UNIX AT&T	C, cc, cxx, c
GNU C++	C, cc, cxx, c
Symantec	cpp, cp
Borland C++	cpp
Watcom	cpp
Microsoft Visual C++	cpp, cxx
Metrowerks Code Warior	cp,cpp

Kompilace a sestavení

Stroustrup původně realizoval kompilátor C++ pomocí převodu z C++ do C místo toho, aby vyvinul kompilátor C++ přímo do objektového kódu. Tento program, který se jmenoval `cfront` (pro C front end), překládal zdrojový kód C++ do zdrojového kódu C, jenž mohl potom být kompilován standardním kompilátorem C. Tento přístup zjednodušil zavedení C++ do společenství C. Jiné implementace používaly tento přístup proto, aby přenesly C++ na jiné platformy. Když se C++ rozvíjel a získával na popularitě, tak se více a více realizátorů pouštělo do vytváření kompilátorů C++, které generují objektový kód přímo ze zdrojového kódu C++. Tento přímý přístup urychluje proces kompilace a zdůrazňuje, že C++ je jiný, i když podobný, jazyk. Pro uživatele je rozdíl mezi překladačem `cfront` a kompilátorem často téměř nepozorovatelný. Například na systému Unix může příkaz `CC` nejprve předat váš program překladači `cfront` a potom předá automaticky jeho výstup kompilátoru C, který se nazývá `cc`. Napříště budeme používat výrazu *kompilátor*, abychom spojili kombinaci překlad a kompilace. Mechanizmus kompilování závisí na implementaci a následující kapitoly nastíní několik společných rysů. Tyto přehledy nastíní základní kroky, ale nejsou náhražkou za nahlédnutí do dokumentace vašeho systému.

Kompilace a sestavování pod Unixem

Předpokládejme například, že jste na unixovém systému, který používá C++ AT&T verzi 3.0. Abyste zkompilovali svůj program, použijte příkaz `CC`. Jméno se píše velkými písmeny, aby se odlišilo od standardního unixového kompilátoru `cc`. Kompilátor `CC` je kompilátor příkazové řádky, to znamená, že zadáváte kompilační příkazy v příkazovém řádku Unixu.

Abyste například zkompilovali zdrojový kód C++ `spiffy.C`, měli byste napsat za unixový prompt tento příkaz:

```
CC spiffy.C
```

Jestliže z důvodu dovednosti, posvěcení nebo štěstí nemá váš program žádné chyby, kompilátor vygeneruje soubor objektového kódu s příponou `o`. V tomto případě by měl kompilátor vytvořit soubor `spiffy.o`.

Následně kompilátor automaticky předá soubor objektového kódu systémovému sestavovacímu programu, tj. programu, který spojuje váš kód spolu s knihovním kódem a vytváří proveditelný soubor. Standardně se proveditelný soubor nazývá `a.out`. Jestliže jste použili pouze jeden zdrojový soubor, sestavovací program také vymaže soubor `spiffy.o`, protože již není potřebný. Abyste spustili program, pouze napíšete jméno spustitelného souboru:

```
a.out
```

Všimněte si, že když kompilujete nový program, vytváří se nový proveditelný soubor `a.out`, který nahrazuje předchozí `a.out`. (to je důvodem, proč proveditelné soubory zabírají mnoho prostoru, takže přepsání starých proveditelných souborů napomáhá redukování požadavků na paměť). Ale když vyvíjíte proveditelný program a chcete, aby se zachoval, stačí když použije unixový příkaz `mv`, abyste změnili jméno proveditelného programu.

V C++, podobně jako v C, můžete program rozšířit o více než jeden soubor. (Mnoho programů v této knize, počínaje kapitolou 8 a konče 16, tak činí.) V tomto příkladě můžete kompilovat program spolu se seznamem všech souborů na příkazovém řádku:

```
CC my.C precious.C
```

Existují-li násobné soubory zdrojového kódu, kompilátor soubory objektových kódů nemaže. Pokud tímto způsobem změníte `my.C`, můžete znovu překompilovat program příkazem:

```
CC my.C, precious.o
```

Toto překompiluje soubor `my.C` a sestaví ho s již dříve zkompilovaným souborem `precious.o`.

Možná budete muset identifikovat některé knihovny. Abyste například měli přístup k funkcím definovaným v knihovně math, musíte do příkazového řádku přidat označení `-lm`.

```
CC usingmath.C -lm
```

Free Software Foundation dodává kompilátor GNU C++, který se nazývá g++ a pracuje podobně jako standardní unixový kompilátor.:

```
g++ spiffy.C
```

Některé verze mohou vyžadovat, že přidáte C++ knihovnu:

```
g++ spiffy.C -lg++
```

GNU g++ je dostupný pro mnoho základen včetně Linuxu, a běží na PC kompatibilních s IBM.

Turbo C++ 2.0, Borland C++ 3.1 (DOS)

V integrovaném prostředí Dosu verzí Turbo C++ a Borland C++, které zahrnují vestavěný editor, používáte řádkové menu, které je přístupné pomocí myši nebo kombinace ALT+KEY, které dávají na vědomí vaše přání. Například menu File vám dovolí vytvořit, uložit a otevřít soubory. Menu Edit napomáhá při editaci zdrojového souboru. Menu Compile nabízí několik možností kompilování a menu Run zobrazuje volby pro spuštění programu. Poté, co jste použili vestavěný editor k napsání programu, nejjednodušší volba je vybrat Run z menu Run. Toto způsobí, že Borland C++ nebo Turbo C++ zkompiluje, sestaví a vykoná váš program. Jestliže kompilátor narazí na chyby, samozřejmě program nespustí, ale zobrazí seznam chyb a zvýrazní pochybné řádky vašeho zdrojového kódu. Integrované prostředí také zahrnuje ladící program, který vám umožní po řádcích krokovat program a prověřit libovolnou hodnotu, kterou chcete vidět.

Pokud vyvíjíte program používající více než jeden soubor, použijte na otevření nového projektu menu Project. Potom můžete použít další volby menu, abyste přidali odpovídající soubory do seznamu projektu. Projektový soubor umožňuje Borlandu C++ a Turbo C++ sledovat, co se děje. Jestliže změníte jeden ze souborů v seznamu projektu, kompilátor ví, že by měl změnit proveditelný program. V Borlandu C++ a Turbo C++ můžete mít několik zdrojových souborů současně ve svém vlastním okně na obrazovce a můžete se snadno přepínat z jednoho souboru do druhého.

Jak Borland C++ tak Turbo C++ přicházejí s výukovým programem, který vám ukazuje spletité detaily prostředí Borlandu C++ a Turbo C++. Samozřejmě si můžete přečíst manuály.

Kompilátory Windows

Produktů pod Windows je také velké množství a jsou často modifikovány, takže by bylo zbytečné popisovat je jednotlivě. Mají však několik společných rysů.

Pro váš program musíte vytvořit projekt a přidat do něj soubor nebo soubory, které váš program vytváří. Každý dodavatel poskytuje spolu s volbami menu integrované vývojové prostředí (IDE – Integrated Development Environment) a pravděpodobně automatickou pomoc při vytváření projektu. Velmi důležitá skutečnost, kterou musíte stanovit, je druh programu, který vytváříte. Typicky bude kompilátor nabízet mnoho voleb, jako jsou aplikace windows, aplikace MFC Windows, dynamická sestavovací knihovna, řízení AktivX, dosový nebo znakový režim, statická knihovna nebo terminálová aplikace. Některé z těchto voleb jsou dostupné jak pro 16 tak pro 32bitové verze.

Protože programy v této knize jsou všeobecně použitelné, měli byste se vyhnout volbám, které vyžadují základny závislé na kódu, jako je aplikace Windows. Místo toho byste měli spouštět program ve znakovém režimu. Tato volba bude záviset na kompilátoru. Některé verze Microsoftu zdůrazňují volbu QuickWin, která napodobuje práci v Dosu; jiné verze zdůrazňují volbu Console. Některé verze Borlandu zdůrazňují volbu EasyWin, která napodobuje práci v Dosu, jiné verze nabízejí volbu Console. Obecně se podívejte, zdali nenaleznete volbu označenou Console, znakový režim nebo proveditelný v Dosu, a zkuste ji.

Jakmile jste nastavili projekt, musíte váš program zkompilovat a sestavit. IDE vám typicky nabídne několik voleb jako jsou Compile, Build, Make, Build All, Link, Execute a Run (ale ne nezbytně všechny ve stejném IDE!).

- ***Compile*** obvykle znamená zkompilovat kód právě otevřeného souboru.
- ***Build nebo Make*** obvykle znamená zkompilovat zdrojové kódy všech souborů v projektu. To je typicky přírůstkový proces. To jest, když projekt obsahuje tři soubory a vy změníte pouze jeden, tak se překompiluje pouze ten jeden.
- ***Build All*** obvykle znamená přeložit zdrojové soubory.
- ***Link*** znamená (jak bylo popsáno dříve) sestavení zkompilovaného zdrojového kódu spolu s nezbytným kódem knihovny.
- ***Run nebo Execute*** znamená vykonat program. Obvykle, pokud jste ještě neprovedli předchozí kroky, Run je také provede, než se pokusí program spustit.

Kompilátor vygeneruje chybové hlášení, porušíte-li pravidla jazyka a identifikuje řádky, ve kterých je problém. Bohužel, pokud jste v jazyce nováčkem, asi pro vás bude obtížné zprávě porozumět. Občas se může chyba vyskytnout před identifikovaným řádkem a občas jednoduchá chyba bude generovat řetězec chybových hlášení.

Tip:

Když opravujete chyby, opravte nejprve první. Nemůžete-li ji nalézt na řádku, který je identifikován jako chybový, ověřte předchozí řádek.

Tip:

Příležitostně se kompilátor po nekompletní kompilaci programu může zaplést a odpovídá nesmyslnými chybovými hlášeními, která nemohou být opravena. V takovém případě můžete věci vyjasnit výběrem Build All a znovu nastartováním celého procesu. Bohužel je velmi obtížné tuto situaci rozpoznat od běžného případu, ve kterém se chybové zprávy zdají nesmyslné.

Obvykle vám IDE dovolí provedení programu v pomocném okně. Některá IDE okno zavírají, když se program ukončí, jiná zůstávají otevřena. Když váš kompilátor zavře okno, sotva si stihnete prohlédnout výstup, pokud nemáte bystré oči nebo fotografickou paměť. Abyste se mohli podívat na výstup, musíte na konec programu umístit jistý dodatečný kód:

```
cin.get(); // přidejte tento příkaz
cin.get(); // a nebo snad také tento
return 0;
```

Příkaz `cin.get()` čte následující stisk klávesy, takže tento příkaz způsobí, že program čeká, dokud nestisknete klávesu `Enter`. (Do programu se neposílá žádný stisk klávesy, dokud nestisknete `Enter`, takže nemá smysl stisknout jiný klíč.) Druhý příkaz je potřebný, když program jinak opouští nezpracovaný znak po reguléroím vstupu. Například když zadáváte číslo, budete vkládat číslo a potom stisknete `Enter`. Program přečte číslo a zanechá stisk klávesy `Enter` nezpracovaný a ten pak bude přečten prvním `cin.get()`.

Kompilátor C++ Builder se trochu odchyluje od tradičního návrhu. Jeho primárním cílem je programování ve Windows. Abyste ho mohli použít pro obecně použitelné programování, vyberte z menu File New. Potom zvolte Console App. Okno, které se otevře, obsahuje verzi kostry `main()`. Některé položky můžete vymazat, ale měli byste zachovat dvě následující nestandardní řádky:

```
#include <vcl\condefs.h>
#pragma hdrstop
```

Kompilátory pro Macintosh

Nejznámější kompilátory C++ Macintosh jsou kompilátory Metrowerks CodeWarior a Symantec C++. Oba poskytují v základních ohledech podobné projektově orientované IDE, ke kterému byste nalezli kompilátor Windows. (Vskutku, obě společnosti nabízejí verze svých kompilátorů orientované na Windows.) V obou produktech začínáte výběrem New Project z menu File. Nabídne se vám volba typů projektu. Pro CodeWarior zvolte MacOs:C/C++:ANSI C++ Console; pro Symantec zvolte ANSII C++ (iostreams). Můžete ta-

ké zvolit mezi verzí 68K (pro Motorolu s řadou procesorů 680X0) nebo verzi PowerPC (pro procesory PowerPC).

Oba projekty obsahují malý zdrojový soubor jako součást inicializačního projektu. Můžete tento program zkompilovat a vykonat, abyste zjistili, zda máte svůj systém řádně nastavený. Nicméně, když jednou poskytnete svůj vlastní kód, měli byste tento soubor z projektu vymazat. Toto provedete zvýrazněním souboru v okně projektu a výběrem Remove z menu Project.

Dále musíte do projektu dodat váš zdrojový kód. Můžete použít New z menu File nebo Open z menu File a otevřít existující soubor. Použijte správnou příponu, jako jsou `.cp` nebo `.cpp`. Použijte menu Project a přidejte tento soubor do seznamu projektu. Některé programy v této knize vyžadují přidávat jeden nebo více zdrojových souborů. Když jste hotovi, vyberte Run z menu Project.

Tip:

Abyste ušetřili čas, můžete používat pouze jeden projekt pro všechny vzorové programy. Vymažte předchozí zdrojové programy ze seznamu projektu a přidejte současný zdrojový kód. To spoří prostor na disku.

Oba kompilátory obsahují ladící program, který vám pomůže lokalizovat příčiny problémů za běhu programu.

Konvence použité v této knize

Abychom rozlišili mezi různými druhy textů, použili jsme několik typografických konvencí. Typ kurzíva se používá pro důležitá slova nebo výrazy použité poprvé jako je *strukturované programování*. Písmo se stejnou roztečí označuje následující:

- Jména nebo hodnoty použité v programu, jako `x`, `starship` a `3.14`
- Klíčová slova jazyka, jako `int` a `if else`
- Jména souborů, jako `iostream`
- Funkce, jako `main()` a `puts()`

Zdrojový kód C++ je zobrazen následovně:

```
#include <iostream>
using namespace;
int main()
{
   cout << "What's up, Doc!\n";
   return 0;
}
```

Provedení vzorového programu používají stejný formát, jen s tím rozdílem, že uživatelský výstup se objevuje v tučném řezu:

```
Prosím, zadejte vaše jméno:
Plato
```

V následující struktuře občas naleznete pravidlo nebo návrh:

Tip:

Učíte se při práci, tedy zkoušejte příklady a experimentujte s nimi.

Mimochodem, právě jste přečetli skutečný a důležitý návrh, nikoli pouze příklad, jak pravidlo nebo návrh vypadá.

Konečně, když zavádíte programový vstup, obvykle musíte stisknout klávesu RETURN nebo ENTER, abyste programu poslali vstup. Některé klávesnice používají jednu a některé jinou; tato kniha používá ENTER.

Náš systém

Tato kniha popisuje ISO/ANSI CD2 návrh definice C++, takže příklady by měly pracovat s libovolnou implementací C++, která je kompatibilní s tímto standardem. (Přinejmenším je to vize naděje přenositelnosti.) Nicméně standard C++ je nový a můžete nalézt několik nesrovnalostí. Například v době psaní této knihy mnoho kompilátorů postrádá prostory jmen nebo novější rysy šablon. Podpora standardní knihovny šablon (STL – Standard Template Library), která je popsaná v kapitole 15, je v této chvíli neúplná. Systémy, které používají verzi 2.0 (nebo pozdější `cfront`) překladače, mohou potom předávat přeložený kód kompilátoru C, který není plně kompatibilní s ANSI, což může vést k tomu, že některé jazykové rysy nejsou implementovány a v některých standardních knihovnách ANSI a hlavičkových souborech nejsou podporovány. Také některé věci, jako je počet bajtů pro vyjádření celého čísla, jsou závislé na implementaci.

Za zmínku stojí, že příklady v této knize byly vyvíjeny pomocí Microsoft Visual C++ 5.0 a Metrowerks CodeWarior vydání 1 na PC pentium s pevným diskem, který běžel pod Windows 95. Programy byly ověřeny pomocí GNU g++ 2.7.1 na IBM kompatibilním s 486, na kterém běžel Linux, Watcomu 10.6 na PC pentium a Metrowerksu CodeWarior Professional vydání 1 na Macintoshi 7100 pod systémem 7.5.5 a 8.0. Tato kniha informuje o nesrovnalostech, které pocházejí všeobecně ze zaostávání za standardem, jako např. „starší implementace používají `ios::fixed` místo `ios_base::fixed`". Kniha informuje o některých skrytých chybách a výstřednostech, které by mohly být obtížnými nebo matoucími; avšak mohou být velmi dobře opraveny v následujících vydáních.

KAPITOLA

2

Vydáváme se do C++

Témata kapitoly:

- Jak vytvořit program v C++
- Direktiva #include
- Obecný formát programu C++
- Funkce main()
- Použití objektu cout pro výstup
- Umístění poznámek do programu C++
- Znak nového řádku \n
- Deklarace a použití proměnných
- Použití objektu cin pro vstup
- Definice a použití jednoduchých funkcí

Když budujete nový dům, začínáte se základy a kostrou. Když nemáte od začátku pevnou konstrukci, budete mít později potíže při doplnění detailů, jako jsou okna, dveře, rámy, pozorovací kopule a parketový taneční sál. Podobně, když se učíte počítačovému jazyku, měli byste začít učením základní struktury programu. Pouze tehdy můžete přejít na detaily, jako jsou cykly a objekty. Tato kapitola vám dává přehled o základní struktuře programu C++ a ukáže některé důležité body – zejména funkce a třídy – které kniha pokrývá později mnohem detailněji. (Myšlenka spočívá v postupném zavedení alespoň některých základních pojmů během cesty k velkému uvědomění, které přijde později.)

Zahájení C++

Začneme s jednoduchým programem, který zobrazuje zprávu. Výpis programu 2.1 používá prostředek C++ cout (vyslovujte sí aut), který produkuje výstup znaků. Zdrojový kód obsahuje pro čtenáře několik poznámek; tyto řádky začínají pomocí // a kompilátor je ignoruje. C++ je citlivý na velikost písma; tj. odlišuje velké a malé znaky. To znamená, že musíte být pečliví při používání a dodržovat stejnou velikost, jako je v příkladech. Jestliže podstrčíte `Cout` nebo `COUT`, kompilátor odmítne vaši nabídku a nařkne vás z používání neznámých identifikátorů. (Kompilátor je také citlivý na pravopis, tak nezkoušejte ani `kout` ani `coot`.) Přípona souboru `cpp` je běžný způsob indikace C++ programu; možná potřebujete použít jinou příponu, jak je popsáno v kapitole 1 „Začínáme“.

Výpis programu 2.1 myfirst.cpp

```
// myfirst.cpp–zobrazuje zprávu
  #include <iostream>                          // direktiva preprocesoru
  using namespace std;                         // zpřistupní definice
  int main()                                   // hlavička funkce
  {                                            // začátek těla funkce
      cout << "Come up and C++ me some time."; // zpráva
      cout << "\n";                            // začátek nové řádky
      return 0;                                // ukončuje main()
  }                                            // konec těla funkce
```

Kompatibilita:

Když používáte starší kompilátor, možná budete potřebovat `#include <iostream.h>` místo `#include <iostream>`; v tomto případě byste také měli vynechat řádek `using namespace std;`, to jest nahraďte

```
#include <iostream>     // budoucí způsob
using namespace std;    // to samé
```

pomocí

```
#include <iostream.h>  // v případě, kdy ještě budoucnosti nebylo dosaženo
```

(Některé velmi staré kompilátory používají `#include <stream.h>` místo `#include <iostream.h>`; pokud máte kompilátor tohoto stáří, měli byste získat buď novější kompilátor nebo starší knihu.) Změna z `iostream.h` na `iostream` je téměř současná a v době psaní této knihy ji někteří dodavatelé ještě neimplementovali.

Některá windowsovská prostředí vykonávají program v samostatném okně a potom automaticky okno zavírají, jakmile program skončí. Jak bylo pojednáno v kapitole 1, můžete donutit okno, aby zůstalo otevřené, dokud nestisknete klávesu Enter tím, že přidáte následující řádek kódu před příkaz return:

```
cin.get();
```

Pro některé programy musíte dodat tyto řádky dva. V kapitole 4 „Odvozené typy", se naučíte, co tento kód provádí.

Úpravy programu

Možná zjistíte, že musíte změnit příklady v této knize, aby na vašem systému běžely. Dvě nejběžnější změny jsou v této první Připomínce ve zmínkách kapitoly. První je věcí standardů jazyka, když je váš kompilátor zastaralý, musíte zahrnout `iostream.h` místo `iostream` a vynechat řádek `namespace`. Druhá je záležitostí programového prostředí; možná musíte přidat jeden nebo dva příkazy `cin.get()`, abyste udrželi programový výstup viditelný na obrazovce. Protože se tyto úpravy používají rovným dílem v každém příkladě této knihy, tato poznámka o kompatibilitě je pouze výstrahou na ta prostředí, která dostanete. Nezapomeňte na ně! Budoucí poznámky o kompatibilitě vás upozorní na další možné úpravy, které možná musíte udělat.

Až použijete váš zvolený editor na vytvoření kopie tohoto programu (nebo když použijete zdrojový soubor na CD ROMu, který doprovází tuto knihu), použijte váš kompilátor na vytvoření spustitelného kódu, jak bylo načrtnuto v kapitole 1. Tady je výstup z běžícího zkompilovaného programu:

```
Come up and C++ me some time.
```

Vstup a výstup v C

Pokud jste byli zvyklí na programování v C, to že vidíte `cout` namísto funkce `printf()` vám možná může přivodit menší šok. C++ ve skutečnosti může používat `printf()`, `scanf()` a všechny další standardní funkce C vstupu a výstupu za předpokladu, že vložíte obvyklý soubor C `stdio.h`. Ale toto je kniha o C++, proto používejte nové prostředky vstupu C++, které jsou v mnoha směrech lepší než ve verzi C.

Program v C++ vytváříte ze stavebních bloků, které se nazývají funkce. Typicky organizujete program do hlavních úloh a potom navrhujete samostatné funkce, které tyto úlohy řídí. Příklad ukázaný ve výpisu programu 2.1 je dostatečně jednoduchý na to, aby sestával z jediné funkce, která se nazývá `main()`. Příklad `myfirst.cpp` má následující prvky:

- Poznámky, které jsou označeny prefixem //
- Direktivu preprocesoru `#include`
- Direktivu using `namespace`
- Hlavičku funkce: `int main()`
- Tělo funkce ohraničené { a }
- Příkaz, který používá C++ `cout` pro zobrazení zprávy
- Příkaz return na ukončení funkce `main()`

Podívejme se na tyto různé prvky nyní podrobněji. Funkce `main()` je dobrým místem pro začátek, protože některé z rysů, které předcházejí `main()`, jako je direktiva preprocesoru, se jednodušeji chápou, až když uvidíte, co `main()` dělá.

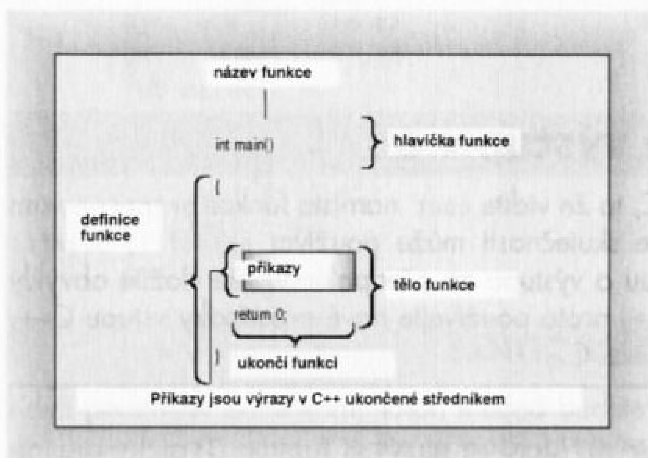
Funkce main()

Ořezaný příklad programu, který je ukázán na výpisu, má následující základní strukturu:

```
int main()
{
příkazy
return 0;
}
```

Tyto řádky vyjadřují, že máte funkci, která se jmenuje `main()` a popisují, jak se funkce chová. Společně vytvářejí *definici funkce*. Tato definice má dvě části. První řádek `int main()`, který se nazývá *hlavičkou funkce* a část uzavřená do složených závorek ({ a }), která je *tělem funkce*. Obrázek 2.1 zobrazuje funkci `main()`. Hlavička funkce je zhuštěný nástin rozhraní funkce a zbytku programu a tělo funkce představuje vaše in-

strukce pro počítač o tom, co by funkce měla provádět. V C++ se každá úplná instrukce nazývá příkazem. Každý příkaz musíte zakončit středníkem, takže středníky nesmíte vynechat, když píšete příklady.



Obrázek 2.1 Funkce main()

Poslední příkaz v `main()`, který se nazývá *příkazem návratu*, ukončuje funkci. O příkazu return se více dozvíte, jakmile přečtete tuto kapitolu.

Příkazy a středníky

Příkaz reprezentuje kompletní počítačovou instrukci. Aby kompilátor rozuměl vašemu kódu, musí vědět, kde jeden příkaz končí a jiný začíná. Některé jazyky používají oddělovače příkazů. Fortran například používá konce řádky, aby oddělil jeden příkaz od následujícího. Pascal používá středník. V Pascalu můžete v určitých případech středník vynechat, jako po příkazu hned před end, když vlastně neoddělujete dva příkazy. (Pragmatici a minimalisté nesouhlasí s tím, zda *může* implikuje *mohl by*.) Ale C++ podobně jako C, používá ukončovací znak spíše, než separátor. Ukončovací znak je středník, který označuje konec příkazu, je součástí příkazu spíše než značkou mezi příkazy. Praktickým závěrem je, že v C++ nikdy nemůžete vynechat středník.

Hlavička funkce jako rozhraní

Právě nyní je hlavním bodem pro zapamatování to, že syntaxe C++ vyžaduje, abyste definici funkce `main()` začínali touto hlavičkou: `int main()`. Kapitola toto později zkoumá detailněji, ale pro ty, kteří nemohou udržet svou zvědavost, je tady přehled.

Obecně je funkce C++ aktivována nebo *volána* jinou funkcí a hlavička funkce popisuje rozhraní mezi funkcí a funkcí, která ji volá. Část, která předchází jménu funkce se nazývá *návratový typ funkce*, který popisuje informační tok ven z funkce do funkce, která ji volá. Část mezi závorkami následujícími za jménem funkce se nazývá *seznam argumentů* nebo *parametrů*, který popisuje informační tok z volající funkce do volané funkce. Tento obecný formát je trochu zmatený, pokud ho aplikujete na `main()`, protože `main()` obvykle z jiné části vašeho programu nevoláte. Ve skutečnosti je však `main()` volaná startovacím kódem, který přidává do vašeho programu kompilátor, jenž dělá prostředníka

mezi programem a operačním systémem (Unix, Windows 95 nebo jakýmkoli jiným). Hlavička funkce tedy popisuje rozhraní mezi `main()` a operačním systémem.

Uvažujme rozhraní funkce `main()` začínající částí `int`. Funkce C++ volaná jinou funkcí může navracet hodnotu do aktivační (volající) funkce. Tato hodnota se nazývá *návratovou hodnotou*. V tomto případě může `main()` navracet celočíselnou hodnotu, jak je indikováno klíčovým slovem `int`. Dále si všimněte prázdných závorek. Obecně může jedna funkce C++ předávat informaci jiné funkci, když ji volá. Část hlavičky funkce uzavřená v závorkách popisuje tuto informaci. V tomto případě prázdné závorky znamenají, že funkce `main()` nedostává žádnou informaci nebo v obvyklé terminologii, `main()` nedostává žádné parametry. (Když řekneme, že `main()` nedostává žádné parametry, neznamená to, že je to nesmyslná, autoritářská funkce. Spíše jde o to, že *parametr* je výraz, který používají počítačoví nadšenci, aby poukázali na předávanou informaci z jedné funkce do druhé.)

Krátce řečeno, hlavička

```
int main()
```

stanovuje, že funkce může navracet celočíselnou hodnotu funkci, která ji volá, a že `main()` nedostává žádnou informaci od funkce, která ji volá.

Mnoho existujících programů používá místo toho klasickou hlavičku C:

```
main() // původní styl C
```

Pod C je vynechání návratového typu to samé, jako když se řekne, že funkce je typu `int`. C++ však tuto fázi použití vynechává.

Můžete také použít tuto variantu:

```
int main(void)  // velmi zřetelný typ
```

Použití klíčového slova `void` v závorkách je explicitní způsob vyjádření toho, že funkce nedostává žádné parametry. Pod C++ (ne pod C), když necháme závorky prázdné, je to totéž, jako kdybychom použili `void` uvnitř závorek. (V C, když necháme závorky prázdné, znamená, že ponecháme v tichosti to, zda má funkce parametry nebo nemá.)

Někteří programátoři používají tuto hlavičku:

```
void main()
```

a vynechávají příkaz return. To je logické, protože návratový typ `void` znamená, že funkce nemá návratovou hodnotu. Tato varianta pracuje na mnoha systémech, ale protože není povinná jako volba pod současnými standardy, na některých systémech nepracuje.

Konečně výbor ANSI/ISO C++ rozhodl, že se příkazem

```
return 0;
```

implicitně rozumí, že patří na konec funkce `main()` (ale ne v ostatních funkcích), pokud ho implicitně neposkytnete.

Proč main() s jiným jménem není to samé

Existuje extrémně nepřekonatelný důvod pojmenovat funkci v programu `myfirst.cpp` `main()`: musíte to udělat. Každý program C++ vyžaduje funkci, která se nazývá `main()`. (A ne mimochodem `Main()` nebo `MAIN()` nebo `mane()`. (Nezapomeňte rozlišovat velikost

písma nebo pravopis.) Protože program myfirst.cpp má pouze jednu funkci, musí nést zodpovědnost, že je main(). Když spustíte program C++, provedení vždy začíná na začátku funkce main(). Proto, když nemáte main(), nemáte kompletní program a kompilátor poukáže na to, že jste nedefinovali funkci main().

Existují výjimky. Například při programování pod Windows můžete napsat modul dynamicky vázané knihovny (dll). To je kód, který mohou používat jiné programy Windows. Protože modul dll není nezávislý program, nepotřebuje main(). Programy pro specializovaná prostředí, jako pro řídící čipy v robotu, možná nepotřebují main(). Ale váš obvykle nezávislý program funkci main() potřebuje; tato kniha pojednává o takovém druhu programu.

Komentáře v C++

Dvojité lomítko (//) začíná poznámku v C++. Poznámka je programátorova připomínka pro čtenáře, která obvykle identifikuje oblast programu nebo vysvětluje jistou stránku kódu. Kompilátor poznámky ignoruje. Konec konců zná C++ alespoň tak dobře jako vy a není schopný poznámkám rozumět. Pokud se týká kompilátoru, výpis programu 2.1 vypadá, jako by byl napsán bez poznámek:

```
#include <iostream>
  using namespace std;
  int main()
  {
      cout << "Come up and C++ me some time.";
      cout << "\n";
      return 0;
  }
```

Poznámky C++ začínají od // a končí koncem řádku. Poznámka může být na svém vlastním řádku, nebo může být na stejném řádku jako kód. Mimochodem, všimněte si prvního řádku Programového výpisu 2.1:

```
// myfirst.cpp - zobrazuje zprávu
```

V této knize začínají všechny programy poznámkou, která poskytuje jméno programu a stručný popis programu. Jak bylo zmíněno v kapitole 1, přípona souboru zdrojového kódu závisí na vašem systému. Některé systémy mohou pro jména používat myfirst.C nebo myfirst.cxx.

Tip:

Pro komentování vašeho programu byste měli používat poznámky. Čím je program složitější, tím se poznámky stávají cennějšími. Nejenže pomáhají jiným porozumět, co jste udělali, ale také pomáhají vám, zvláště, když jste určitou dobu program neviděli.

Styl komentářů C

C++ také rozpoznává komentáře C, které jsou uzavřeny mezi symboly /* a */:

```
#include <iostream> /* styl komentáře v C */
```

Protože komentář stylu C je ukončen */ spíše než koncem řádku, můžete ho rozšířit na několik řádků. Snažte se však držet stylu C++. To způsobí, že programátor C, který pohlédne přes vaše rameno bude vědět, že jste se posunuli na vyšší úroveň programování.

Preprocesor C++ a soubor iostream

Tady je krátký popis toho, co potřebujete vědět. Má-li váš program C++ obvyklé prostředky vstupu nebo výstupu, poskytněte tyto dva řádky:

```
#include <iostream>
using namespace std;
```

Pokud váš kompilátor nemá tyto řádky rád (například když si stěžuje, že nemůže nalézt soubor `iostream`), zkuste místo toho následující řádek:

```
#include <iostream.h> //kompatibilní se staršími kompilátory
```

To je vše, co skutečně potřebujete vědět, abyste přinutili váš program pracovat, ale nyní se podíváme na věc trochu hlouběji.

C++, podobně jako C, používá preprocesor, to je program, který zpracovává zdrojový soubor než dostane prostor hlavní kompilace. (Některé implementace C++, jak si můžete připomenout z kapitoly 1, používají překlad programu z C++ do C. Ačkoli překladač je také forma preprocesoru, neučíte se zde o něm, místo toho se učíte o programu, který zpracovává direktivy, jejichž jména začínají na #.) Nemusíte dělat nic zvláštního, abyste tento preprocesor vyvolali. Automaticky funguje, když kompilujete program.

Výpis používá direktivu `#include`:

```
#include <iostream>   // direktiva preprocesoru
```

Tato direktiva způsobí to, že preprocesor přidá obsah souboru `iostream` do vašeho programu. To je typická akce preprocesoru: přidání nebo záměna textu před jeho kompilací.

To vyvolává otázku, proč byste měli přidat obsah souboru `iostream` do programu. Odpověď se týká komunikace mezi programem a okolním světem. Předpona `io` v iostream se odkazuje na vstup, což je informace, která se programu dodává, a na výstup, což je informace posílaná z programu ven. Váš první program tyto definice potřebuje, aby mohl použít prostředek cout na zobrazení zprávy. Direktiva `#include` způsobuje, že se obsah souboru `iostream` pošle spolu s obsahem vašeho souboru kompilátoru. V podstatě obsah souboru `iostream` nahradí v programu řádku `#include <iostream>`. Váš původní soubor se nezmění, ale složený soubor vytvořený z vašeho souboru a `iostream` jdou do další etapy kompilace.

Pamatujte:

Program, který používá cin a `cout` pro vstup a výstup, musí obsahovat soubor `iostream`.

Jména hlavičkových souborů

Soubory jako je `iostream`, se nazývají *vložené soubory* (protože jsou vkládány do jiných souborů) nebo *hlavičkové soubory* (protože jsou vkládány na začátek souborů). Kompilátory C++ přicházejí s mnoha hlavičkovými soubory, každý podporuje určitou skupinu prostředků. Tradice C používala s hlavičkovým souborem příponu `h`, jako jednoduchý způsob identifikace typu souboru podle jeho jména. Například hlavičkový soubor `math.h` podporuje různé matematické funkce C. Původně dělal C++ to samé. Například hlavičkový soubor podporující vstup a výstup se jmenoval `iostream.h`. Avšak teprve nedávno se použití C++ změnilo. Nyní je přípona `h` rezervována pro staré hlavičkové soubory C (které programy C++ stále mohou používat), zatímco hlavičkové soubory C++ nemají žádnou příponu. Také existují hlavičkové soubory C, které byly konvertovány do hlavičkových souborů C++. Tyto soubory byly přejmenovány odhozením přípony h (což vytvořilo jméno stylu C++) a přidáním předpony c ke jménu souboru (indikující, že podporuje rysy C++). Například C++ verze `math.h` je hlavičkový soubor `cmath`. Občas jsou verze C a C++ identické, zatímco v jiných případech novější verze může mít několik změn. Pro čistě C++ hlavičkové soubory, jako je `iostream`, je vynechání h mnohem více než kosmetická změna, a to proto, že hlavičkové soubory zbavené h také začleňují prostory jmen, následující téma v tomto přehledu. Tabulka 2.1 shrnuje konvence pro pojmenování hlavičkových souborů.

Tabulka 2.1 Konvence pro pojmenování hlavičkových souborů

Druh hlavičky	Konvence	Příklad	Poznámky
starý styl C++	končí na .h	iostream.h	použitelné v programech v C++
starý styl C	končí na .h	math.h	použitelné v programech v C, C++
nový styl C++	žádná přípona	iostream	použitelné v programech v C++, používá jméno prostoru jmen std
konvertované C	předpona c, žádná přípona	cmath	použitelné v programech v C++, může používat rysy nepatřící C, jako namespace std

Se zřetelem ke tradici C při používání různých přípon souborů na označení různých typů souborů se zdá rozumné mít nějaké speciální přípony, které by indikovaly hlavičkové soubory C++. Výbor ANSI/ISO to také pociťoval. Problém byl dohodnout se, jakou používat příponu, tak se nakonec nedohodl na ničem.

Prostory jmen

Používáte-li `iostream` místo `iostream.h`, měli byste použít následující direktivu namespace, abyste zpřístupnili vašemu programu definice dostupné v `iostream`:

```
using namespace std;
```

Toto se nazývá *direktiva using*. Nejjednodušší je udělat to, že ji nyní přijmete a trápit se s ní budete později (například v kapitole 8, „Dobrodružství ve funkcích"). Ale tady je přehled o tom, co se děje, takže nebudete ponecháni úplně ve tmě.

Podpora prostoru jmen je nový rys C++, který je navržen ke zjednodušení psaní programů, které spojují předem existující kód několika dodavatelů. Jeden potenciální problém spočívá v tom, že můžete použít dva předem zabalené produkty, které oba mají, řekněme, funkci pojmenovanou `wanda()`. Když potom použijete funkci `wanda()`, kompilátor neví, kterou verzi potřebujete. Prostředek *prostoru jmen* zanechá balíčku dodavatele jeho zboží v jednotce, která se nazývá prostor jmen, takže můžete použít jméno prostoru jmen k identifikaci výrobku dodavatele, který chcete. Tak Microflop Industries by mohl umístit své definice do prostoru jmen nazvaného `Microflop`. Potom `Microflop::wanda()` by se mohl stát úplným jménem pro jeho funkci `wanda()`. Podobně, `Piscine::wanda()` by mohl označit `wanda()` verzi Piscine Corporation. Takže nyní by mohl váš kompilátor použít prostory jmen pro rozlišení mezi různými verzemi:

```
Microflop::wanda(" go dancing?);   // použití verze prostoru jmen Microflop
Priscine::wanda("a fish named Desire");//použití verze prostoru jmen Pris-
cine
```

V tomto duchu, třídy, funkce a proměnné, které jsou standardními komponentami kompilátorů C++, jsou nyní umístěny do prostoru jmen, který se nazývá `std`. To platí pro hlavičkové soubory, které nemají příponu `h`. To například znamená, že proměnná `cout`, která je používaná pro výstup a definovaná v `iostream`, se ve skutečnosti nazývá `std::cout`. Avšak většina uživatelů se nedomnívá, že by měla konvertovat kód předdefinovaného prostoru jmen, který používá `iostream.h` a `cout` pomocí kódu používajícího `iostream` a `std::cout`, pokud to mohou dělat bez mnoha potíží. Tady přichází direktiva `using`. Řádek:

```
using namespace std;
```

znamená, že můžete použít jména definovaná v prostoru jmen `std`, aniž použijete předponu `std::`.

Výstup C++ pomocí cout

Podívejme se, jak se zobrazuje zpráva. Program `myfirst.cpp` používá následující příkaz C++:

```
cout << "Come up and C++ me some time.";
```

Část, uzavřená do apostrofů, je zpráva pro tisk. V C++ se jakákoli skupina znaků uzavřená do apostrofů nazývá řetězec, podle všeho proto, že obsahuje několik znaků společně zřetězených do větší jednotky. Označení << signalizuje, že příkaz posílá řetězec do `cout`; symboly zdůrazňují, že teče informace. A co je `cout`? To je předdefinovaný objekt, který ví, jak zobrazit různé věci zahrnující řetězce, čísla a samostatné znaky. (Objekt, jak si můžete pamatovat z kapitoly 1, je určitá instance třídy a třída definuje, jak se ukládají a používají data.)

Dobře, toto je trochu neobratné. Po několik následujících kapitol nebudete v pozici, abyste se učili o objektech, nicméně jeden musíte používat. Ve skutečnosti to odhaluje jednu

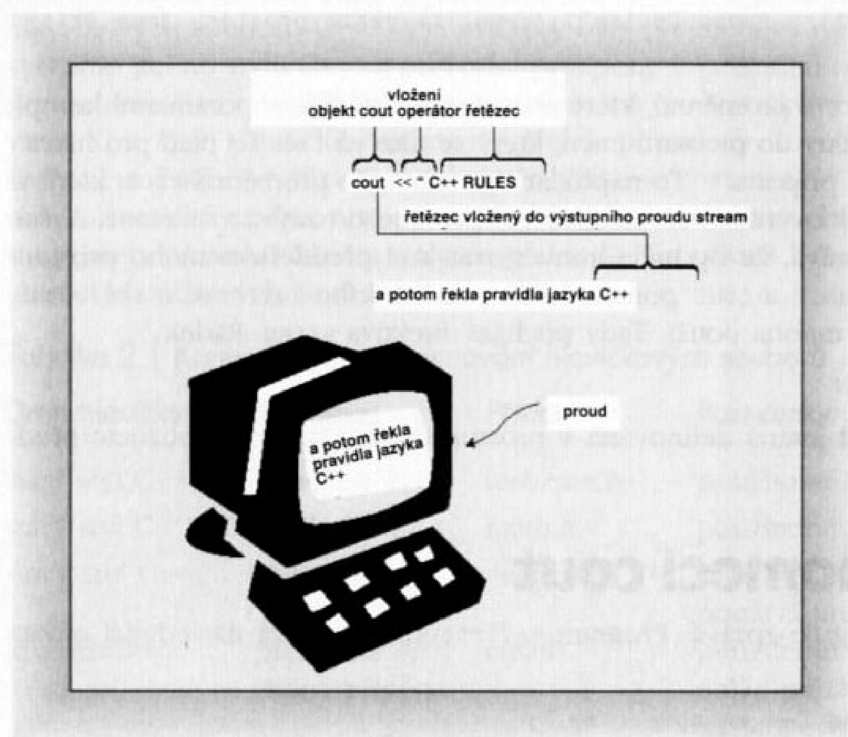
ze silných stránek objektů. Nemusíte znát vnitřnosti objektu, abyste ho mohli používat. Vše co potřebujete vědět, je jeho rozhraní, to jest, jak ho používat. Objekt `cout` má jednoduché rozhraní. Jestliže `string` reprezentuje řetězec, abyste ho zobrazili, provedete následující:

```
cout << string;
```

Toto je vše, co potřebujete na zobrazení řetězce vědět. Ale nyní se podívejme na to, jak pojmové stanovisko C++ tento proces reprezentuje. Z tohoto pohledu je výstupem řetězec – to jest skupina znaků, která proudí z programu. Objekt `cout`, jehož vlastnosti jsou definovány v souboru `iostream`, představuje řetězec. Vlastnosti objektu cout zahrnují operátor vložení (<<), který vkládá informaci do řetězce na jeho pravou stranu. Tedy příkaz (všimněte si ukončující středník)

```
cout << "Come up and C++ me some time.";
```

vkládá řetězec „Come up and C++ me some time." do výstupního proudu. Tedy spíš než bychom řekli, že váš program zobrazuje zprávu, můžeme říci, že vkládá řetězec do výstupního proudu. Zní to jaksi působivěji (viz obrázek 2.2).



Obrázek 2.2 Zobrazení řetězce pomocí cout

Pokud jste přešli z C do C++, pravděpodobně jste si všimli, že operátor vložení (<<) vypadá právě tak, jako bitový operátor (<<) posun vlevo. Toto je příklad *přetížení operátoru*, podle kterého může mít stejný symbol operátoru různé významy. Kompilátor používá textovou souvislost, aby rozpoznal, o který význam se jedná. Samotný C přetížení operátoru trochu rozumí. Například symbol `&` představuje jak operátor adresování, tak bitový operátor `AND`. Symbol `*` představuje jak násobení, tak dereferenční ukazatel. Nejdůležitějším bodem tady není přesná funkce těchto operátorů ale to, že stejný symbol může mít více než jeden význam, jehož přesný význam určí kompilátor z textové souvislosti. (Provádíte bez-

mála to samé, když určujete význam „shoot" ve „shoot the breeze" proti „shoot the piano player".) C++ rozšiřuje pojem přetížení operátoru tím, že vás nechá předefinovat významy operátorů pro uživatelsky definované typy zvané třídy.

Znak nového řádku (\n)

Nyní vyšetříme zvláštně vypadající označení, které se vyskytuje ve druhém příkazu výstupu:

```
cout << "\n";
```

Dvojznak \n je zvláštní označení C++, které představuje důležitý pojem, kterému se přezdívá *znak nového řádku*. Ačkoli píšete znak nového řádku pomocí dvou znaků (\ a n), počítají se jako jeden znak. Všimněte si, že používáte zpětné lomítko (\) namísto běžného lomítka (/). Když zobrazíte znak nového řádku, kurzor obrazovky se přesune na začátek dalšího řádku, a když vyšlete znak nového řádku na tiskárnu, tiskací hlavička se přesune na začátek nového řádku. Znak nového řádku si nese své jméno.

Všimněte si, že prostředek cout se nepřesouvá automaticky na další řádek, když tiskne řetězec, takže první příkaz cout na výpisu 2.1 opouští kurzor umístěný hned za tečkou na konci výstupního řetězce. Abyste přesunuli kurzor na začátek dalšího řádku, musíte na výstup poslat znak nového řádku. Nebo podle vžité hantýrky, musíte znak nového řádku vložit do výstupního proudu.

Znak nového řádku můžete použít právě tak, jako běžný znak. Výpis 2.1 ho používá pomocí oddělených příkazů, ale mohl by ho použít v původním řetězci. To jest, dva původní příkazy můžete nahradit následovně:

```
cout << "Come up and C++ me some time. \n";
```

Znak nového řádku dokonce můžete vložit doprostřed řetězce. Například uvažujme následující příkaz:

```
cout << "I am a mighty stream\nof lucid\nclarity.\n";
```

Každý znak nového řádku posouvá kurzor na začátek dalšího řádku, což vytváří následující výstup:

```
I am a mighty stream
of lucid
clarity.
```

Vynecháním nových řádků můžete vytvořit úspěšné příkazy cout, které tisknou na stejný řádek. Například příkazy:

```
cout << "The Good, the";
cout << "Bad, ";
cout << "and the Ukelele\n";
```

produkují následující výstup:

```
The Good, theBad, and the Ukelele
```

Všimněte si, že začátek jednoho řetězce následuje okamžitě za koncem předchozího. Pokud chcete mezeru v místě, kde se dva řetězce spojují, musíte ji zahrnout do jednoho z ře-

tězců. (Pamatujte, abyste vyzkoušeli tyto příklady pro výstup, musíte je vložit do kompletního programu spolu s funkcí main() a otevírací a uzavírací složené závorky.)

C++ má ještě jiný způsob vyjádření nového řádku na výstupu: slovo endl:

```
cout << "What's next?" << endl; // endl znamená začátek nové řádky
```

Tento výraz je definovaný v `iostream`. Pro většinu je jeho psaní trochu jednodušší než `"\n"`, ale můžete ho použít pouze samostatně, ne jako část řetězce. To jest, řetězec `"What's next?\n"` obsahuje znak nového řádku, ale `"What's next?endl"` je pouze řetězec, který končí čtyřmi písmeny e, n, d a l.

Formátování zdrojového kódu C++

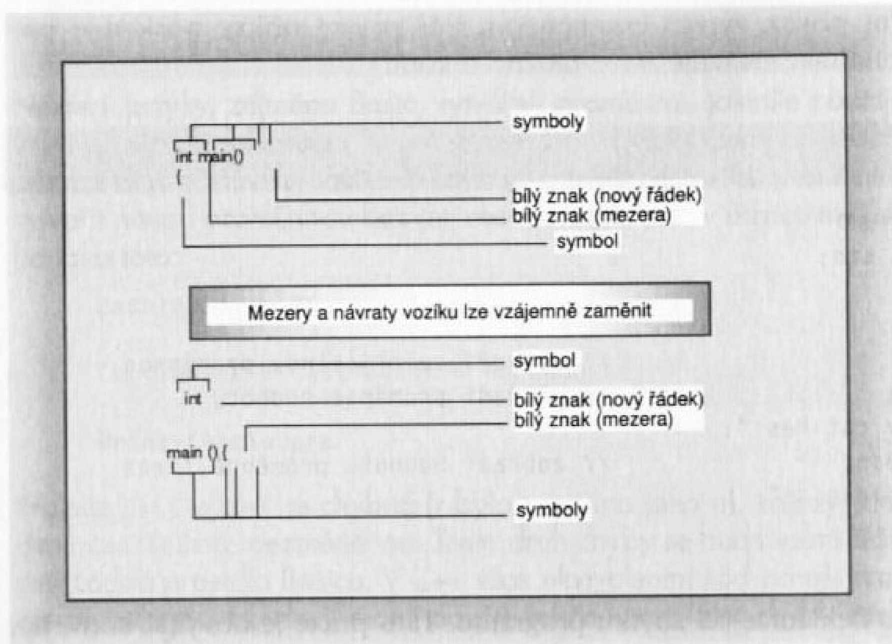
Některé jazyky, jako je Fortran, jsou řádkově orientované s jedním příkazem na řádku. Pro tyto jazyky slouží příkaz nový řádek k oddělení příkazů. Avšak v C++ označuje konec každého příkazu středník. C++ vám dává volnost v nakládání příkazem nový řádek stejným způsobem jako s mezerou nebo tabulátorem. To jest v C++ můžete obvykle použít mezeru, kde byste mohli použít příkaz nový řádek a naopak. To znamená, že můžete rozšířit jediný příkaz na několik řádků nebo umístit několik příkazů na jediný řádek. Například byste mohli naformátovat myfirst.cpp následovně:

```
#include <iostream>
        using
         namespace
          std;
          int
main
() { cout
          <<
"Come up and C++ me some time.";
        cout << "\n"; return 0; }
```

Toto je strašný, ale správný kód. Musíte vypozorovat některá pravidla, v C a C++ nemůžete vložit mezeru, tabulátor nebo nový řádek doprostřed prvku, jako je jméno, nebo nemůžete umístit nový řádek doprostřed řetězce.

```
int ma in() // CHYBNÉ - mezera ve jménu
re
turn 0; // CHYBNÉ - slovo na více řádcích
cout << "Behold the Beans
of Beauty!"; // CHYBNÉ - řetězec na více řádcích
```

Nedělitelné prvky v řádku kódu se nazývají *lexikálními jednotkami* (tokens) (viz obrázek 2.3). Obecně musíte oddělit jednu lexikální jednotku od následují mezerou, tabulátorem nebo novým řádkem, které se společně nazývají *oddělovače*.



Obrázek 2.3 Lexikální jednotky a oddělovače

```
returnO;          // CHYBNĚ, musí být return 0;
return(0);        // SPRÁVNĚ, oddělovač vynechán
return (0);       // SPRÁVNĚ, oddělovač použit
int main()        // SPRÁVNĚ, oddělovač vynechán
int main ( )      // TAKÉ SPRÁVNĚ, oddělovač použit
```

Styl zdrojového kódu C++

Ačkoli vám C++ dává mnoho volnosti ve formátování, bude snazší číst váš program, když budete dodržovat jasný styl. Pokud byste měli správný, ale strašný kód, byli byste nespokojeni. Většina programátorů používá styl podle výpisu 2.1, který zachovává tato pravidla:

- Jeden příkaz na řádku
- Otevírací a uzavírací složená závorka funkce, každá z nich má svůj vlastní řádek
- Příkazy ve funkcích odsazené od složených závorek
- Žádné oddělovače kolem závorek, které jsou spojené se jménem funkce

Prvá tři pravidla mají jednoduchý záměr, udržet dokonalý a čitelný kód. Čtvrté pomáhá odlišit funkce od některých vestavěných struktur C++, jako jsou cykly, které také používají závorky. Kniha vás upozorní na ostatní pravidla, jakmile se objeví.

Více o příkazech C++

Program C++ je skupina funkcí a každá funkce je skupina příkazů. C++ má několik druhů příkazů, tak se podívejme na některé možnosti. Výpis 2.2 poskytuje dva druhy příka-

zů. Za prvé, deklarační příkaz vytváří proměnnou. Přiřazovací příkaz poskytuje proměnné hodnotu. Program také ukazuje nové možnosti cout.

Výpis 2.2 fleas.cpp

```
#include <iostream>
using namespace std;
int main()
{
      int fleas;                    // vytvoří celočíselnou proměnnou
      fleas = 38;                   // přiřadí proměnné hodnotu
      cout << "My cat has ";
      cout << fleas;                // zobrazí hodnotu proměnné fleas
      cout << " fleas.\n";
      return 0;
}
```

Prázdný řádek odděluje deklarace od zbytku programu. Tato praxe je obvyklá konvence C, ale je o něco méně běžná v C++. Zde je výstup programu:

```
My cat has 38 fleas.
```

Následujících několik stránek prozkoumá tento program.

Deklarační příkazy a proměnné

Počítače jsou přesné, systematické stroje. Abyste uložili položku informace do počítače, musíte určit jak umístění paměti, tak množství paměťového prostoru, který informace vyžaduje. Jeden relativně bezbolestný způsob jak to udělat, je použití *deklaračního příkazu* pro označení typu paměti a poskytnutí návěští pro místo. Například program má tento deklarační příkaz (všimněte si středníku) :

```
int fleas;
```

Tento příkaz zaručuje, že program poskytuje dostatek paměti pro to, co se nazývá `int`. Kompilátor se úloze postará o detaily umístění a označení paměti. C++, může obsluhovat několik druhů nebo typů dat a `int` je nejzákladnějším datovým typem. Odpovídá celému číslu bez desetinné čárky. V C++ může být typ `int` kladný nebo záporný, ale rozsah velikosti závisí na implementaci. Kapitola 3, „Práce s daty", poskytuje o `int` a dalších základních typech detaily.

Kromě poskytnutí typu, deklarační příkaz říká, že od nynějška bude program používat jméno `fleas`, které ztotožňuje uloženou hodnotu s umístěním. `Fleas` nazýváme proměnnou, protože můžete měnit její hodnotu. Kdybyste vynechali ve `fleas.cpp` deklaraci, kompilátor by hlásil chybu, pokud by se program dále pokoušel `fleas` použít. (Možná budete chtít ve skutečnosti vynechat deklaraci právě proto, abyste viděli, jak kompilátor odpovídá. Jakmile potom tuto odpověď v budoucnosti uvidíte, budete umět posoudit chybějící deklarace.)

Proč musí být proměnné deklarovány

Některé jazyky, zejména Basic, vytvářejí proměnné, jakmile použijete nové jméno bez pomoci explicitní deklarace. To může být pro uživatele více přátelské, a také za omezených podmínek je. Problém nastane, když se spletete ve jménu proměnné, nedopatřením můžete vytvořit novou proměnnou bez její realizace. To jest, v Basicu můžete udělat něco podobného jako toto:

```
CastleDark = 34
...
CastleDank = CastleDank + MoreGhosts
...
Print CastleDark
```

Protože `CastleDank` je chybná (r bylo zapsáno jako n), změny, které s ní provedete, zanechají `CastleDark` nezměněnou. Tento druh chyby se bude velmi těžce hledat, protože neporuší žádná pravidla Basicu. V C++ však ekvivalentní kód poruší pravidlo o potřebě deklarovat proměnnou, abyste ji mohli použít, takže kompilátor chybu zachytí a označí potenciální skrytou chybu.

Obecně tedy deklarace ukáže typ dat, která mají být uložena, a jméno, které pro ně bude program používat. V tomto určitém případě program vytváří proměnnou, která se jmenuje `fleas`, a do níž může uložit celé číslo (viz obrázek 2.4.).

Obrázek 2.4 Deklarace proměnné

Deklarační příkaz se v programu nazývá *definičním deklaračním* příkazem nebo krátce *definicí*. To znamená, že jeho výskyt zapříčiní, že kompilátor alokuje pro proměnnou paměťový prostor. Ve složitějších situacích můžete mít také *referenční deklarace*. Ty říkají počítači, aby použil proměnné, které již byly někde definovány. Obecně, deklarace nemusí být definicí, ale v tomto příkladě je.

Pokud jste obeznámeni s C nebo Pascalem, tak jste již obeznámeni s deklaracemi proměnných. Také to pro vás může být skromným překvapením. V C a Pascalu se deklarace proměnných dostávají bezprostředně na začátek funkcí nebo procedur. Ale C++ taková omezení nemá. Vskutku, styl C++ je deklarování proměnné právě před jejím prvním použitím. Tímto způsobem nemusíte zpětně hledat v programu, abyste viděli, jakého je typu. Příklad na toto téma uvidíte v kapitole později. Tento styl má nevýhodu, že nemů-

žete shromáždit všechna vaše jména proměnných v jednom místě; tedy nemůžete najednou říct, jaké proměnné funkce používá.

Připomínka:
Styl C++ deklarování proměnných je deklarovat proměnné co nejblíže, jak je to možné, k jejich prvnímu použití.

Přiřazovací příkaz

Přiřazovací příkaz přiřazuje hodnotu do paměťového místa. Například příkaz

```
fleas = 38;
```

přiřazuje celé číslo 38 lokaci, která je reprezentovaná proměnnou `fleas`. Symbol = se nazývá *přiřazovacím operátorem*. Jeden neobvyklý rys C++ (i C) je, že můžete přiřazovací operátor použít za sebou. Jako například:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

Přiřazení probíhá zprava doleva. Nejprve se `88` přiřadí do `stainwey`; potom hodnota `stainway`, která je nyní `88` do `baldwin`; potom se hodnota `88` proměnné `baldwin` přiřadí do `yamaha`. (C++ následuje náklonnost C pro povolení tajemně vyhlížejícího kódu.)

Nový trik pro cout

Doposud byly cout pro tisk dodávány řetězce. Výpis 2.2 dodatečně dává cout proměnnou, jejíž hodnota je celočíselná.

```
cout << fleas;
```

Program netiskne slovo fleas, namísto toho tiskne hodnotu do `flea` uloženou, která je 38. Skutečně, tady existují dva triky v jednom. Za prvé, cout nahrazuje fleas její současnou numerickou hodnotou 38. Za druhé, hodnotu překládá do správných výstupních znaků.

Jak můžete vidět, cout pracuje jak s řetězci, tak celými čísly. Možná se vám to nezdá zvlášť významné, ale uvědomte si, že celé číslo 38 je zcela něco jiného než řetězec „38". Řetězec obsahuje znaky, pomocí kterých číslo píšete, to jest znak 3 a znak 8. Program interně ukládá kód znaků 3 a 8. Abyste vytiskli řetězec, `cout` jednoduše tiskne každý znak řetězce. Ale číslo 38 je uloženo jako numerická hodnota. Spíše než uložení každé číslice jednotlivě, počítač ukládá 38 jako binární číslo. (Dodatek A, „Číselné soustavy", o této reprezentaci pojednává.) Hlavním bodem zde je, že `cout` musí přeložit číslo v celočíselném tvaru do znakové podoby předtím, než ho vytiskne. Dále, příkaz `cout` je dostatečně chytrý, aby rozpoznal, že fleas je celé číslo vyžadující konverzi.

Možná, že porovnání se starým C ukáže, jak je `cout` chytrý. Abyste vytiskli řetězec „38" a celé číslo 38 v C, měli byste použít víceúčelovou funkci C `printf()`:

```
printf("Printing a string: %s\n", "38");
printf("Printing an integer: %d\n", 38);
```

Aniž byste se dostali do spletitostí printf(), povšimněte si, že musíte použít zvláštní kódy (%s a %d), které indikují, zda se chystáte tisknout řetězec nebo celé číslo. A když řeknete printf(), aby vytiskla řetězec a dodáte jí omylem celé číslo, `printf()` je příliš hloupá na to, aby si všimla chyby. Pouze pokračuje dále a zobrazí nesmysly.

Inteligentní způsob, kterým se chová `cout` pramení z objektově orientovaných rysů C++. C++ operátor vložení (<<) v podstatě přizpůsobuje své chování, aby vyhověl typu dat, která ho následují. Toto je příklad přetížení operátoru. V pozdějších kapitolách, až začnete s přetížením funkcí a operátorů, se naučíte, jak takové chytré návrhy můžete implementovat sami.

cout a printf()

Pokud jste byli zvyklí na C a `printf()`, můžete si myslet, že cout vypadá podivně. Možná dokonce budete dávat přednost lpění na těžce nabitém vítězství nad `printf()`. Ale `cout` ve skutečnosti není ve vzhledu podivnější než `printf()`se všemi svými specifikacemi konverze. Důležitější je, že `cout` má značné výhody. Jeho schopnost rozeznat typy odráží inteligentnější a spolehlivější návrh. Tedy, je schopné rozšíření. To jest, můžete operátor << předefinovat tak, že `cout` může rozpoznat nové datové typy, které navrhnete. A když si pochvalujete pěkné řízení, které `printf()`poskytuje, můžete dosáhnout stejných efektů pomocí pokročilých způsobů použití `cout` (viz kapitola 16, „Vstup, výstup a soubory").

Další příkazy C++

Podívejte se na pár dalších příkladů příkazů. Program na výpisu 2.3 dále rozšiřuje předchozí příklad tím, že vám dovoluje za běhu programu zadávat hodnotu. Abyste to udělali, používá `cin` (vyslovujte sí-in), vstupní doplněk cout. Program také ukazuje ještě další způsob použití objektu cout, tohoto mistra přizpůsobivosti.

Výpis 2.3 yourcat.cpp

```
// yourcat.cpp - vstup a výstup
#include <iostream>
using namespace std;
int main()
{
    int fleas;
    cout << "How many fleas does your cat have?\n";
    cin >> fleas;                    // vstup C++
    // další řádka spojuje výstup
    cout << "Well, that's " << fleas   << " fleas too many!\n";
return 0;
}
```

Zde je vzorek výstupu:

```
How many fleas does your cat have?
112
Well, that's 112 fleas too many!
```

Program má dva nové rysy: používá cin ke čtení vstupu z klávesnice a spojuje tři výstupní příkazy do jednoho. Podívejme se na to.

Použití cin

Jak výstup ukazuje, hodnota zadaná z klávesnice (112) je nakonec přiřazena proměnné `fleas`. Tady je příkaz, který přání provádí:

```
cin >> fleas;
```

Když se díváte na tento příkaz, můžete prakticky vidět informaci tekoucí z `cin` do `fleas`. Přirozeně existuje trochu více formálních vlastností tohoto procesu. Právě tak jak C++ považuje výstup za proud znaků tekoucích z programu, tak považuje vstup za proud znaků tekoucích do programu. Soubor `iostream` definuje cin jako objekt, který představuje tento proud. Pro výstup operátor << vkládá znaky do výstupního proudu. Pro vstup `cin` používá >> operátor, aby získal znaky ze vstupního proudu. Typicky dodáte na pravou stranu operátoru proměnnou, aby obdržela získanou informaci. (Symboly << a >> byly vybrány tak, aby vizuálně připomněly směr, kterým informace teče.)

Podobně jako `cout` je i `cin` chytrý objekt. Konvertuje vstup, který je pouze skupinou zadaných znaků z klávesnice, do tvaru, který odpovídá proměnné přijímající informaci. V tomto příkladě deklaroval celočíselnou proměnnou `fleast` , takže je vstup konvertován do číselného tvaru, který počítač používá pro ukládání celých čísel.

Více o cout

Druhým novým rysem `yourcat.cpp` je kombinace tří výstupních příkazů v jednom. Soubor iostream definuje operátor << tak, že můžete následně kombinovat (spojovat) vstup:

```
cout << "Well, that's " << fleas  << " fleas too many.\n";
```

Toto vám umožňuje kombinovat výstup řetězce a celého čísla do jediného příkazu. Výsledný výstup je stejný, jako to, co produkuje následující kód:

```
cout << "Well, that's ";
cout << fleas;
cout << " fleas too many.\n";
```

Máte-li ještě náladu na radu o `cout`, můžete také spojenou verzi přepsat tím způsobem, že jednotlivé příkazy rozšíříte přes tři řádky:

```
cout   << "Well, that's "
       << fleas
       << " fleas too many.\n";
```

To je proto, že pravidla volného formátování C++ považují nové řádky a mezery mezi syntaktickými jednotkami za zaměnitelné. Tento poslední postup je vhodný tehdy, když šíře řádku omezuje váš styl.

„...pravidla volného formátování C++ považují nové řádky a mezery mezi syntaktickými jednotkami za zaměnitelné."

Dotek třídy

O `cin` a `cout` jste viděli dost, abyste si srovnali svou orientaci blíže k objektové tradici. Především se více naučíte o pojmu tříd. Třídy jsou jedním ze základních pojmů objektově orientovaného programování v C++.

Třída je datový typ, který definuje uživatel. Abyste definovali třídu, popište, jaký druh informace může představovat a jaký druh činností může s daty provádět. Třída má stejný poměr k objektu, jako má typ k proměnné. To jest, definice třídy popisuje formát dat a jak může být použita, zatímco objekt je jednotka vytvořená podle specifikace formátu dat. Nebo pomocí nepočítačových výrazů, jestliže je třída analogická s kategorií jako jsou známí herci, potom je objekt analogický s jednotlivým vzorem této kategorie, jako je žába Kermit. Abychom rozšířili tuto analogii, třída, která reprezentuje herce, by měla zahrnovat definice možných činností, jako jsou studium textu, vyjádření smutku, zdůraznění nebezpečí, přijetí odměny apod. Jestliže jste byli vystaveni různé terminologii OOP, možná vám pomůže vědět, že třídy C++ odpovídají tomu, co některé jazyky nazývají objektovým typem a objekty C++ odpovídají instanci objektu nebo instanci proměnné.

Buďme nyní více konkrétní. Připomeňme tuto deklaraci proměnné:

```
int fleas;
```

Vytváří určitou proměnnou (`fleast`), která má vlastnosti typu `int`. To jest, `fleast` může ukládat celá čísla a může být používána určitými způsoby, například pro sčítání a odčítání. Nyní uvažujme `cout`. To je objekt, který je vytvořený proto, aby měl vlastnosti třídy `ostream`. Definice třídy `ostream` (jiný obyvatel souboru `iostream`) popisuje druh dat, které reprezentuje objekt `ostream` a operace, které s ním nebo na něm můžeme provádět, jako jsou vložení čísla nebo řetězce do výstupního proudu. Podobně je cin objekt vytvořený s vlastnostmi třídy `istream`, také definovaným v `iostream`.

Pamatujte:

Třída popisuje všechny vlastnosti datového typu a objekt je entita vytvořená podle tohoto popisu.

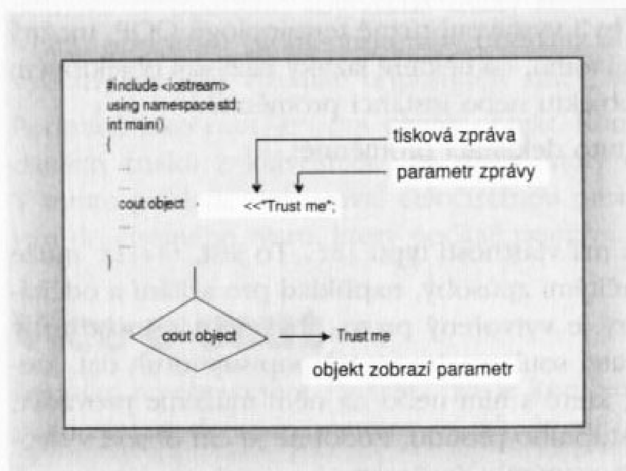
Učili jste se, že třídy jsou uživatelsky definované typy, ale jako uživatel jste určitě nenavrhoval třídy `ostream` a `istream`. Zrovna tak jako mohou vstoupit funkce do knihoven funkcí, tak třídy mohou vstoupit do knihoven tříd. To je případ tříd `ostream` a `istream`. Technicky nejsou vestavěny do jazyka C++, ale jsou příklady tříd, kterým se poštěstilo přijít spolu s jazykem. Definice tříd jsou rozmístěny v souboru `iostream`, a nejsou vestavěny do kompilátoru. Dokonce můžete tyto definice modifikovat, když chcete, avšak to není dobrý nápad. (Přesněji, je to doopravdy strašný nápad.) Skupina tříd `iostream` a příbuz-

né skupiny `fstream` (nebo soubor I/O) jsou pouze množiny definic tříd, které přicházejí se všemi rannými implementacemi C++. Avšak výbor ANSI/ISO C++ přidal ke standardu ještě několik tříd. Také většina implementací poskytuje dodatečné definice tříd jako součást programového balíku. Velká část současné přitažlivosti C++ spočívá v existenci rozsáhlých a užitečných knihoven, které podporují programování pod Unixem, Macintoshem a Windows.

Popis třídy specifikuje všechny operace, které mohou být nad objekty tříd prováděny. Abyste na určitém objektu provedli takovou povolenou činnost, posíláte objektu zprávu. Například, když chcete, aby objekt `cout` zobrazil řetězec, posíláte mu zprávu, která ve skutečnosti říká „Objekte! Zobraz toto!". C++ poskytuje několik způsobů zasílání zpráv. Jeden způsob, nazvaný použití metody třídy, je v podstatě volání funkce, podobně jako způsob, který jste již viděli. Druhý, který se používá pro `cin` a `cout`, je předefinování operátoru. Tedy příkaz

```
cout << "I am not a crook."
```

používá předefinovaný operátor <<, aby poslal `cout` „zobraz zprávu". V tomto případě zpráva přichází spolu s parametrem, což je řetězec, který se má zobrazit. (Viz obrázek 2.5.)



Obrázek 2.5 Poslání zprávy objektu

Funkce

Protože jsou funkce moduly, ze kterých jsou postaveny programy C++ a protože jsou podstatné pro OOP definice C++, měli byste se s nimi pečlivě seznámit. Jelikož některé stránky funkcí jsou pokročilá témata, hlavní diskuse o funkcích přijde později v kapitole 7 „Funkce – programové moduly C++" a v kapitole 8 „Dobrodružství ve funkcích". Ale když se budete nyní zabývat některými základními charakteristikami funkcí, budete to mít později s funkcemi snazší a nacvičenější. Zbytek této kapitoly vás seznámí s těmito základy funkcí.

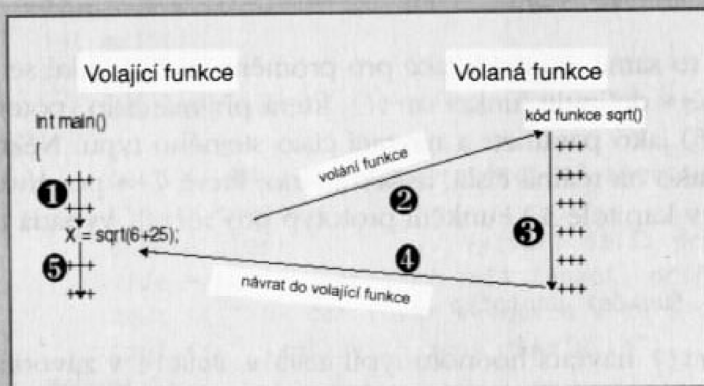
Funkce C++ přicházejí ve dvou variantách: takových, které vracejí hodnoty a takových, které je nevracejí. Ve standardní knihovně funkcí C++ můžete nalézt příklady každého druhu a můžete vytvořit své vlastní funkce každého typu. Podívejte se na knihovní funkci, která má návratovou hodnotu, a potom vyzkoušejte, jak umíte napsat své vlastní jednoduché funkce.

Použití funkce s návratovou hodnotou

Funkce, která má návratovou hodnotu, produkuje hodnotu, kterou můžete přiřadit proměnné. Například standardní knihovna C/C++ obsahuje funkci nazvanou `sqrt()`, která vrací druhou odmocninu čísla. Předpokládejme, že chcete vypočítat druhou odmocninu z čísla 6,25 a přiřadíte ji do proměnné x. Mohli byste ve svém programu použít následující příkaz:

```
x = sqrt(6.25);  // navrací hodnotu 2.5 a přiřazuje ji do x
```

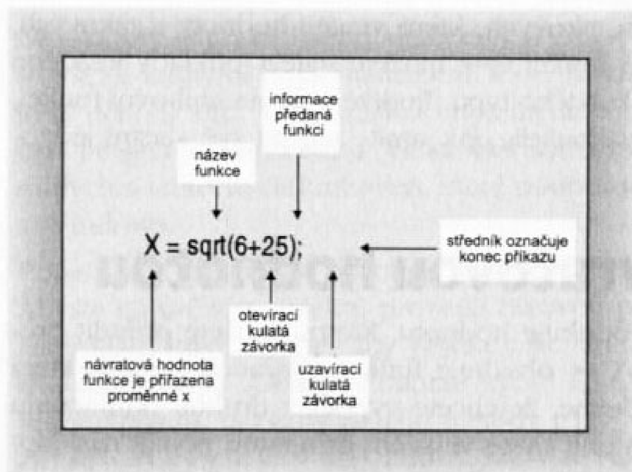
Výraz `sqrt(6.25)` vyvolává nebo volá funkci sqrt. Výraz `sqrt(6.25)` se označuje *volaná funkce* a funkce obsahující funkční volání se označuje *volající funkce*. (Viz obrázek 2.6.)



Obrázek 2.6 Volání funkce

Hodnota v závorkách (v tomto příkladě 6.25) je informace posílaná funkci; tím je řečeno, že má být předána funkci. Tímto způsobem funkci posílaná hodnota se nazývá parametr. (Viz obrázek 2.7.) Funkce `sqrt()` vypočítá odpověď, jež je 2,5, a posílá tuto hodnotu zpět volající funkci; zpět poslaná hodnota se nazývá návratová hodnota funkce. Uvažujte o návratové hodnotě jako o tom, co nahrazuje v příkazu funkční volání, jakmile funkce skončí svou práci. Tedy příklad přiřazuje návratovou hodnotu proměnné x. Krátce řečeno, parametr je informace posílaná funkci a návratová hodnota je posílaná z funkce zpět.

To je k této záležitosti prakticky všechno, kromě toho, že kompilátor musí před použitím funkce vědět, jaký druh parametrů funkce používá a jaký má druh návratové hodnoty. To jest, zda je funkční návrat celé číslo, znak, číslo s desetinnou částí, rozsudek obvinění nebo něco jiného? Pokud kompilátor tuto informaci ztrácí, jak interpretovat návratovou hodnotu? Způsob, kterým C++ tlumočí tuto informaci, je příkaz funkčního prototypu.



Obrázek 2.7 Syntaxe funkčního volání

Pamatujte:

Program C++ by měl pro každou funkci v programu použít prototyp.

Funkční prototyp dělá pro funkci to samé, co deklarace pro proměnnou říká, jaké se vyžadují typy. Například knihovna C++ definuje funkci `sqrt()`, která přijímá číslo (potenciálně) s desetinnou částí (jako 6.25) jako parametr a navrací číslo stejného typu. Některé jazyky poukazují na taková čísla jako na reálná čísla, avšak jméno, které C++ používá, je `double`. (O `double` se více dozvíte v kapitole 3.) Funkční prototyp pro `sqrt()` vypadá takto:

```
double sqrt(double);    // funkční prototyp
```

Počáteční `double` znamená, že `sqrt()` navrací hodnotu typu `double`. `Double` v závorkách znamená, že `sqrt()` vyžaduje parametr `double`. Takže tento prototyp popisuje `sqrt()` přesně tak, jak se používá v následujícím výrazu:

```
x = sqrt (6.25);
```

Mimochodem ukončovací středník identifikuje tento řádek textu jako příkaz, čili vytvoří místo hlavičky funkce prototyp. Pokud vynecháte středník, kompilátor interpretuje řádek jako hlavičku funkce a očekává od vás, že budete pokračovat tělem funkce, které funkci definuje.

Použijete-li `sqrt()` v programu, také musíte poskytnout prototyp. To můžete učinit dvěma způsoby:

- Můžete sami napsat prototyp funkce do zdrojového souboru.
- Můžete zahrnout hlavičkový soubor `cmath` (`math.h` pro starší systémy), který má prototyp v sobě.

Druhý způsob je lepší, protože hlavičkový soubor, dokonce pravděpodobněji než vy, dodá správný prototyp. Každá funkce v knihovně C++ má prototyp v jednom nebo více hlavičkových souborech. Proto ověřte popis funkce ve svém manuálu nebo v přímé nápo-

vědě, pokud ji máte, popis vám řekne, jaký hlavičkový soubor máte použít. Například popis funkce `sqrt()` by vám řekl, abyste použili hlavičkový soubor `cmath`. (Znovu možná musíte použít starší hlavičkový soubor `math.h`, který pracuje pro oba programy C a C++.)

Nesměšujte funkční prototyp s definicí funkce. Prototyp, jak jste viděli, pouze popisuje funkční rozhraní. To jest, popisuje informaci posílanou funkci a informaci posílanou zpět. Definice však zahrnuje kód, jak funkce pracuje, například kód pro výpočet druhé odmocniny čísla. C a C++ pro knihovnu funkcí rozdělují tyto dva pojmy – prototyp a definice. Knihovní soubory obsahují zkompilovaný kód funkcí, zatímco hlavičkové soubory obsahují prototypy.

Měli byste funkční prototyp umístit dopředu před první použití funkce. Obvyklá praxe je umístit prototyp právě před funkci `main()`. Výpis 2.4 demonstruje použití knihovní funkce `sqrt()`; prototyp poskytuje zahrnutím souboru `cmath`.

Výpis 2.4 sqrt.cpp

```
// sqrt.cpp - použití funkce pro výpočet druhé odmocniny
#include <iostream>
using namespace std;
#include <cmath>          // nebo math.h
int main()
{
     double cover;        // double používá pro reálná čísla

     cout << "How many square feet of sheets do you have?\n";
     cin >> cover;
     double side;         // vytváří další proměnnou
     side = sqrt(cover); // volá funkci, přiřazuje návratovou hodnotu
     cout << "You can cover a square with sides of " << side;
     cout << " feet\nwith your sheets.\n";
return 0;
}
```

Kompatibilita:

Pokud používáte starší kompilátor, můžete ve výpisu 2.4 namísto `#include <cmath>` použít `#include <math.h>`

Použití knihovních funkcí

Knihovny funkcí C++ jsou uloženy v knihovních souborech. Když kompilátor kompiluje program, musí pro funkce, které jste použili, hledat knihovní soubory. Kompilátory se liší v názoru, jaké knihovní soubory mají hledat automaticky. Když se pokusíte spustit výpis 2.4 a dostanete zprávu, že `_sqrt` je nedefinovaná externí proměnná (zní to jako podmínka, které se máte vyhnout!), je naděje, že váš kompilátor nenalezl automaticky matematickou knihovnu. (Kompilátory rády přidávají ke jménu funkce předponu podtržítko – další jemná připomínka toho, že mají poslední slovo k vašemu programu.) Pokud dostanete takovou zprávu, zkontro-

lujte dokumentaci vašeho kompilátoru, abyste zjistili, jak má kompilátor najít správnou knihovnu. Obvyklá unixová implementace například vyžaduje, že použijete na konci příkazové řádky volbu –lm (*pro* knihovnu *math*):

```
CC sqrt.C -lm
```

Pouhé zavedení hlavičkového souboru `cmath` poskytne prototyp, ale ne nezbytně přiměje kompilátor, aby vyhledal správný knihovní soubor.

Zde je vzorek běhu programu:

```
How many square feet of sheets do you have?
123.21
You can cover a square with sides of 11.1 feet
with your sheets.
```

Protože `sqrt()` pracuje s typem hodnot `double`, příklad vytváří proměnné tohoto typu. Všimněte si, že deklarujete proměnnou `double` použitím stejného způsobu nebo syntaxe, jako když deklarujete proměnnou typu `int`:

```
typ_jména jméno_proměnné;
```

Typ `double` dovoluje proměnným `cover` a `side`, aby obsahovaly hodnotu s desetinnou částí, například 123,21 a 1,1. Jak uvidíte v kapitole 3, typ double zahrnuje mnohem větší rozsah hodnot než typ `int`.

C++ vám umožňuje deklarovat nové proměnné kdekoli v programu, takže `sqrt.cpp` nedeklaruje `side` dříve než ji použijete. C++ vám také dovoluje přiřadit hodnotu proměnné, když ji vytváříte, takže byste také mohli udělat toto:

```
double side = sqrt (cover);
```

K tomuto procesu, který se nazývá inicializace, se vrátíme v kapitole 3.

Všimněte si, že `cin` ví, jak má konvertovat informaci ze vstupního proudu do typu `double` a `cout` ví, jak má vložit typ `double` do výstupního proudu. Jak bylo dříve poznamenáno, jsou to chytré objekty.

Varianty funkcí

Některé funkce vyžadují více než jednu informaci. Tyto funkce používají několikanásobné parametry oddělené čárkami. Například funkce `pow()` z knihovny math má dva parametry a navrací hodnotu, která je rovna prvnímu parametru umocněnému na druhý. Má tento prototyp:

```
double pow(double, double);  // prototyp funkce se dvěma parametry
```

Jestliže, řekněme, chcete umocnit 5 na 8-ou, měli byste použít takovouto funkci:

```
answer = pow(5.0, 8.0);  // volání funkce se seznamem parametrů
```

Jiné funkce nemají žádné parametry. Například jedna z knihoven C (ta, která je spojená s hlavičkovým souborem `cstdlib` nebo `stdlib.h`) má funkci `rand()`, která nemá žádné parametry a navrací náhodné celé číslo. Jeho prototyp vypadá takto:

```
int rand(void);  // prototyp funkce, která nemá parametry
```

Klíčové slovo `void` explicitně oznamuje, že funkce nemá žádné parametry. Jestliže vynecháte void a necháte závorky prázdné, C++ to interpretuje tak, že nejsou parametry. Funkci můžete použít tímto způsobem:

```
myGuess = rand();  //  volání funkce s žádnými parametry
```

Všimněte si, že na rozdíl od některých programovacích jazyků musíte použít ve funkci závorky, dokonce i když nejsou žádné parametry.

Také existují funkce, které nemají žádnou návratovou hodnotu. Předpokládejme například, že jste napsali funkci, která zobrazuje číslo ve formátu, který spolu s číslem zobrazí znak dolaru nebo centu. Pošlete jí parametr, řekněme 23,5, a měla by na obrazovce zobrazit $23,50. Protože funkce posílá hodnotu na obrazovku místo volajícímu programu, není to návratová hodnota. Pro návratovou hodnotu to v prototypu označíte klíčovým slovem void:

```
void bucks(double);  // prototyp funkce s žádnou návratovou hodnotou
```

Protože nevrací hodnotu, nemůžete funkci použít jako část přiřazovacího příkazu nebo jiného výrazu. Místo toho máte čistý příkaz volání funkce:

```
bucks(1234.56);  //  volání funkce, žádná návratová hodnota
```

Některé jazyky rezervují název *funkce* pro funkce s návratovou hodnotou a výrazy *procedure* nebo *subroutine* používají pro jazyky, které nemají návratovou hodnotu, ale C++, podobně jako C, používají výraz *funkce* pro obě varianty.

Uživatelsky definované funkce

Standardní knihovna C poskytuje více než 140 předdefinovaných funkcí. Pokud se některá hodí pro vaše potřeby, rozhodně ji použijte. Avšak často musíte napsat svoji vlastní, zvláště, když navrhujete třídy. Mimochodem při návrhu vlastních funkcí je mnohem více legrace, takže si nyní tento proces vyzkoušíte. Používali jste již několik uživatelsky definovaných funkcí, ale všechny se jmenovaly `main()`. Každý program C++ musí mít funkci `main()`, kterou uživatel musí definovat. Předpokládejme, že chcete přidat další uživatelsky definovanou funkci. Podobně jako u knihovní funkce budete volat uživatelsky definovanou funkci jejím jménem. A podobně jako u knihovní funkce, musíte před použitím funkce poskytnout funkční prototyp, což prakticky uděláte umístěním prototypu před definici `main()`. Novým prvkem je to, že také musíte poskytnout zdrojový kód nové funkce. Nejjednodušším způsobem je umístění kódu do stejného souboru za kód `main()`. Výpis programu 2.5 objasňuje tyto prvky.

Výpis programu 2.5

```
// ourfunc.cpp - definování vaší vlasní funkce
#include <iostream>
using namespace std;
void simon(int);     // funkční prototyp pro simon()
int main()
{
    simon(3);        // volání funkce simon()
```

```
        cout << "Pick an integer: ";
        int count;
        cin >> count;
        simon(count);    // opětné volání
        return 0;
    }

void simon(int n)    // definice funkce simon()
{
        cout << "Simon says touch your toes " << n << " times.\n";
}                       // funkce void nepotřebují příkaz return
```

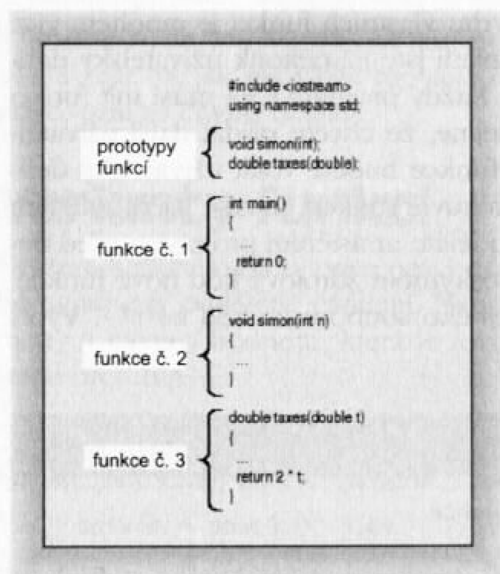
Funkce `main()` volá dvakrát funkci `simon()`, jednou s parametrem 3 a jednou s proměnným parametrem `count`. Mezitím uživatel zadává celé číslo, které se používá pro nastavení hodnoty `count`. Příklad nepoužívá ve výzvě znak nového řádku. To má za následek, že se uživatelský vstup vyskytuje na stejném řádku jako výzva. Zde je vzorek běhu programu:

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
```

Tvar funkce

Definice funkce `simon()` má stejnou formu, jako definice `main()`. Za prvé je tam hlavička funkce. Za druhé, uzavřené do závorek, přichází tělo funkce. Tvar definice funkce můžete zobecnit následovně:

```
typ jméno_funkce(seznam_parametrů)
{
   příkazy
}
```



Obrázek 2.8 Definice funkcí se nachází v souboru za sebou

Všimněte si, že zdrojový kód funkce simon() následuje za uzavírací složenou závorkou main(). Podobně jako v C, a ne jako v Pascalu, C++ nedovoluje vkládat jednu definici

funkce do druhé. Každá definice funkce stojí od ostatních odděleně; všechny funkce se vytvářejí stejně. (Viz obrázek 2.8.)

Hlavičky funkcí

Funkce simon() má tuto hlavičku:

```
void simon(int n)
```

Počáteční void znamená, že simon() nemá návratovou hodnotu. Takže, když voláte funkci simon(), tak neprodukuje číslo, které můžete přiřadit proměnné v main(). Tedy první volání funkce vypadá takto:

```
simon(3);     // správně pro funkce void
```

Protože ubohá simon() ztratila návratovou hodnotu, nemůžete použít tento způsob:

```
simple = simon(3);  // není povoleno kvůli funkcím void
```

Výraz int n uvnitř závorek znamená, že se od vás očekává, že použijete simon() s jediným parametrem typu int. Proměnné n se přiřadí nová hodnota předaná během volání funkce. Tedy volání funkce

```
simon(3);
```

proměnné n definované v hlavičce simon() hodnotu 3. Když příkaz cout v těle funkce použije n, použije hodnotu předanou ve volání funkce. Proto tedy simon(3) zobrazí ve svém výstupu 3. Volání simon(count) ve vzorku běhu programu způsobí, že funkce zobrazí 512, protože je to hodnota dodaná do count. Krátce řečeno, hlavička simon() vám říká, že funkce používá jediný parametr typu int a že nemá návratovou hodnotu.

Přezkoumání funkční hlavičky main():

```
int main()
```

Počáteční int znamená, že main() navrací celočíselnou hodnotu. Prázdné závorky (které by nepovinně mohly obsahovat void) znamenají, že main()nemá žádné parametry. Funkce, které mají návratové hodnoty, by měly používat klíčové slovo return na poskytnutí návratové hodnoty a ukončení funkce. Proto jste tedy použili na konci main()následující příkaz:

```
return 0;
```

Toto je logicky konzistentní: od main()se předpokládá, že navrátí hodnotu typu int a vy jí máte doručit celé číslo 0. Ale možná, že jste zvědavi, komu vracíte hodnotu? Konec konců, v žádném ze svých programů jste nic neviděli, co by volalo main():

```
squeeze = main();    // chybí ve všech našich programech
```

Odpověď spočívá v tom, že můžete přemýšlet o vašich operačních systémech (řekněme Unix nebo Dos), jak volají váš program. Takže návratová hodnota main()není navrácena jiné části programu, ale operačnímu systému. Mnoho operačních systémů může používat návratovou hodnotu programu. Například příkazový soubor Unixu nebo dávkové soubory Dosu mohou být navrženy tak, aby prováděly programy a testovaly jejich návratové hodnoty, které se obvykle nazývají výstupní hodnoty. Obvyklá konvence spočívá v tom, že nulová výstupní hodnota znamená, že program proběhl úspěšně, zatímco nenulová hodnota znamená, že nastal nějaký problém. Můžete tedy navrhnout váš program

v C++ tak, aby navracel nenulovou hodnotu, řekněme tehdy, když selže otevření souboru. Potom můžete navrhnout příkazový nebo dávkový soubor, který by vykonal tento program a použijete jisté alternativní akce, jestliže program signalizuje selhání.

Klíčová slova

Klíčová slova jsou slovníkem počítačového jazyka. V této kapitole jste použili čtyři klíčová slova: `int`, `void`, `return` a `double`. Protože jsou tato klíčová slova pro C++ specifická, nemůžete je použít pro jiné účely. To znamená, že nemůžete použít `return` pro jméno proměnné nebo `double` pro jméno funkce. Ale můžete je použít jako část jména, třeba v `painter` (se skrytým `int`) nebo `return_aces`. Dodatek B, „Klíčová slova v jazyce C++", poskytuje úplný seznam klíčových slov C++. Mimochodem `main` není klíčovým slovem, protože není částí jazyka. Spíše je jménem požadované funkce. `Main` můžete použít jako jméno proměnné. (To může způsobit problém za okolností, které jsou těžko srozumitelné, než abychom je zde popsali, a protože je to v jakémkoli případě matoucí, raději byste to neměli dělat.) Podobně, ostatní jména funkcí a objektů, nejsou klíčová slova. Avšak použijete-li stejné jméno, řekněme `cout`, jak pro objekt, tak pro proměnnou v programu, spletete kompilátor. To jest, můžete použít `cout` jako jméno proměnné ve funkci, která nepoužívá objekt `cout` pro výstup, ale nemůžete použít cout oběma způsoby ve stejné funkci.

Uživatelsky definované funkce s návratovou hodnotou

Nyní popojděme o jeden krok dále a napišme funkci, která používá příkaz return. Funkce main() již přece ukazuje postup pro funkci s návratovou hodnotou: dodejte hlavičce funkce návratový typ a použijte na konci těla funkce `return`. Tento způsob můžete použít na řešení váhového problému pro ty, kteří navštíví Spojené království. Ve Spojeném království je mnoho koupelnových měřidel cejchováno ve váhových jednotkách o čtrnácti librách (stone), v USA v librách nebo mezinárodně v kilogramech. Slovo stone je v tomto kontextu jak v jednotném, tak v množném čísle. (Anglický jazyk postrádá vnitřní konzistenci, řekněme u C++.) Jeden stone je 14 liber a program ve výpisu 2.6 používá funkci, která tuto konverzi provádí.

Výpis programu 2.6 convert.cpp

```
// convert.cpp - konvertuje stouny na libry
#include <iostream>
using namespace std;
int stonetolb(int);        // funkční prototyp
int main()
{
    int stone;
    cout << "Enter the weight in stone: ";
    cin >> stone;
    int pounds = stonetolb(stone);
```

```
    cout << stone << " stone are ";
    cout << pounds << " pounds.\n";
    return 0;
}

int stonetolb(int sts)
{
    return 14 * sts;
}
```

Zde je vzorek běhu programu:

```
Enter the weight in stone: 14
14 stone are 196 pounds.
```

V `main()`používá program funkci `cin`, aby poskytl celočíselné proměnné `stone` hodnotu. Tato hodnota se předává funkci `stonetolb()` jako parametr a je přiřazena ve funkci proměnné `sts`. Funkce `stonetolb()` používá klíčové slovo `return`, aby navrátila `main()`hodnotu `14 * sts`. Výraz ukazuje, že nejste omezeni jednoduchým číslem, které následuje `return`. Zde se použitím složitějšího výrazu vyhnete trápení tak, že nemusíte vytvořit novou proměnnou, které přiřadíte hodnotu předtím, než ji navrátíte. Program vypočítá hodnotu výrazu (196 v tomto příkladu) a navrátí výslednou hodnotu. Pokud vás navrácení hodnoty výrazu obtěžuje, můžete použít delší cestu:

```
int stonetolb(int sts)
{
        int pounds = 14 * sts;
        return pounds;
}
```

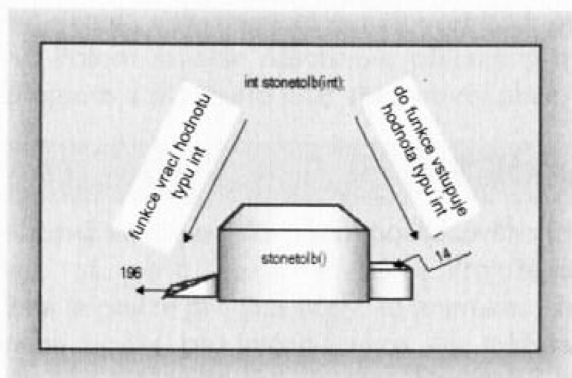
Obě verze produkují stejný výsledek, ale druhá to provádí trochu déle.

Obecně můžete použít funkci s návratovou hodnotou kdekoli byste mohli použít jednoduchou konstantu stejného typu. Například `stonetolb()` navrací hodnotu typu `int`. To znamená, že funkci můžete použít následujícím způsobem:

```
int aunt = stonetolb(20);
int aunts = aunt + stonetolb(10);
cout << "Ferdie weighs " << stonetolb(16) << " pounds.\n";
```

V každém případě program počítá návratovou hodnotu a potom toto číslo v těchto příkazech použije.

Jak příklady ukazují, prototyp funkce popisuje rozhraní funkce, to jest, jak funkce působí na zbytek programu. Seznam parametrů ukazuje, jaký druh informace vstupuje do funkce a typ funkce ukazuje typ návratové hodnoty. Programátoři často popisují funkce jako černé skříňky (výraz z elektroniky) specifikované tokem informace dovnitř a ven. Funkční prototyp perfektně toto hledisko vykresluje (viz obrázek 2.9).



Obrázek 2.9 Prototyp funkce a funkce jako černá skříňka

Funkce stonetolb() je krátká a jednoduchá, přesto vyjadřuje úplný rozsah funkčních vlastností:

- Má hlavičku a tělo.
- Přijímá parametr.
- Navrací hodnotu.
- Vyžaduje prototyp.

Považujme `stonetolb()` za standardní tvar funkčního návrhu. Funkce budete blíže probírat v kapitolách 7 a 8. Mezitím by vám měl materiál z této kapitoly dodat dobrý pocit, jak funkce pracují a jak zapadají do C++.

Shrnutí příkazů

Následující seznam je shrnutím několika příkazů C++, o kterých jste se učili a které jste v této kapitole používali.

- Deklarační příkaz oznamuje jméno a typ proměnné použité ve funkci.
- Přiřazovací příkaz používá k přiřazení hodnoty do proměnné přiřazovací operátor (=).
- Příkaz zprávy posílá zprávu objektu, inicializuje některé druhy činností.
- Volání funkce aktivuje funkci. Když volaná funkce končí, program se vrací na příkaz, který bezprostředně následuje volání funkce.
- Prototyp funkce deklaruje návratový typ funkce spolu s počtem a typem parametrů, které funkce očekává.
- Příkaz return posílá hodnotu z volané funkce zpět do volající funkce.

Shrnutí

Program v C++ sestává z jednoho nebo více modulů zvaných funkce. Programy se začínají vykonávat na začátku funkce, která se nazývá `main()`, takže byste vždy měli mít funkci tohoto jména. Funkce, na oplátku, sestává z hlavičky a těla. Hlavička funkce vám říká, jaký druh návratové hodnoty, je-li nějaký, funkce produkuje a jaký druh informace očekává, aby jí byl předán pomocí parametrů. Tělo funkce sestává ze skupiny příkazů C++ uzavřených do páru složených závorek: {}.

Typy příkazů C++ zahrnují deklarační příkazy, příkazy volání funkce, příkazy objektových hlášení a příkazy return. Deklarační příkazy oznamují jméno proměnné a stanovují typ dat, který mohou obsahovat. Přiřazovací příkaz dodává proměnné hodnotu. Volání funkce předává řízení programu volané funkci. Když funkce končí, řízení se navrací na příkaz ve volající funkci bezprostředně následující za funkčním voláním. Zpráva dává pokyn objektu aby vykonal určitou činnost. Příkaz return je mechanismus, pomocí kterého funkce navrací hodnotu volající funkci.

Třída je uživatelsky definovaná specifikace datového typu. Tato specifikace líčí, jak má být informace reprezentována a také jaké operace mohou být s daty prováděny. Objekt je entita vytvořená podle stanovení třídy, podobně jako jednoduchá proměnná, je entita vytvořená podle popisu datového typu.

C++ poskytuje pro řízení vstupu a výstupu dva předdefinované objekty (`cin` a `cout`). Jsou to příklady tříd `istream` a `ostream`, které jsou definovány v souboru `iostream`. Tyto třídy pohlížejí na vstup a výstup jako na proud znaků. Operátor vložení (<<), který je definován pro třídu `ostream`, vám dovoluje vkládat data do výstupního proudu a operátor extrakce (>>), který je definovaný pro třídu `istream`, vám dovoluje extrahovat informaci ze vstupního proudu. Jak `cin` tak `cout` jsou chytré objekty, schopné automatické konverze informace z jedné formy na jinou v závislosti na programovém kontextu.

C++ může používat rozsáhlou množinu knihovních funkcí C. Abyste knihovní funkci mohli používat, měli byste zahrnout hlavičkový soubor, který poskytuje prototyp funkce.

Nyní, když máte celkový přehled o programech C++, můžete pokračovat v následujících kapitolách, abyste doplnili podrobnosti a rozšířili horizont.

Opakovací otázky

Odpovědi na tyto a následující otázky pro přezkoušení můžete nalézt v Dodatku I, „Odpovědi na otázky pro přezkoušení".

1. Jak se nazývají moduly programů C++?
2. Co provádí následující direktiva preprocesoru?

```
#include <iostream>
```

3. Co provádí následující příkaz?

```
using namespace std;
```

4. Jaký příkaz byste použili pro vytištění výrazu „Hello, world" a nastavení nového řádku?

5. Jaký příkaz byste použili pro vytvoření celočíselné proměnné se jménem `cheese`?
6. Jaký příkaz byste použili pro přiřazení hodnoty 32 do proměnné `cheese`?
7. Jaký příkaz byste použili pro čtení vstupní hodnoty z klávesnice do proměnné `cheese`?
8. Jaký příkaz byste použili pro tisk „Máme X druhů sýra", kde okamžitá hodnota proměnné `cheese` nahradí proměnnou X.
9. Co vám o funkci říká následující hlavička funkce?

```
int froop(double t)
```

10. Kdy při definici funkce nepoužíváte klíčové slovo `return`.

Programovací cvičení

1. Napište program v C++, který zobrazí vaše jméno a adresu.
2. Napište program v C++, který se vás zeptá na vzdálenost 1/8 míle (furlong) a konvertuje jí do vzdálenosti 0,9144 m (yard) (jeden furlong je 220 yardů).
3. Napište program v C++, který používá tři uživatelsky definované funkce (`main()` se počítá za jednu) a produkuje následující výstup:

```
Three blind mice
Three blind mice
See how they run
See how they run
```

Jedna funkce je volaná dvakrát a produkuje první dvě řádky a zbývající funkce také volaná dvakrát, by měla poskytovat zbývající výstup.

4. Napište program v C++, v jehož funkci `main()` se volá uživatelsky definovaná funkce, která jako parametr přijímá teplotu ve stupních Celsia a navrací ekvivalentní hodnotu ve stupních Fahrenheita. Program by měl požádat uživatele o vstup hodnoty ve stupních Celsia a zobrazit výsledek, jak je ukázáno v následujícím kódu:

```
Please enter A Celsius value: 20
20 degrees Celsius is 68 degrees Fahrenheit.
```

Pro nahlédnutí je zde vzorec, který provádí konverzi:

Fahrenheit = 1.8 * Celsius +32.0

Práce s daty

KAPITOLA

3

Podstata objektově orientovaného programování spočívá v navrhování a rozšiřování vašich vlastních datových typů. Navržené typy představují úsilí na vytvoření typů, které odpovídají datům. Děláte-li to pořádně, zjistíte, že se s daty pracuje později mnohem jednodušeji. Ale než budete umět vytvářet své vlastní typy, musíte znát a rozumět vestavěným typům v C++, protože z těchto typů budou vaše stavební bloky.

Vestavěné typy C++ se hlásí ve dvou skupinách: základní typy a odvozené typy. V této kapitole potkáte základní typy, které reprezentují celá čísla a čísla v pohyblivé řádové čárce. To může znít jako pouze dva typy; avšak C++ připouští, že žádný celočíselný typ, ani žádný typ v pohyblivé řádové čárce neodpovídá všem programovým požadavkům, proto nabízí několik variant na tato dvě datová témata. Dále v kapitole 4, „Odvozené typy", následuje zpracování několika odvozených typů ze základních typů; tyto odvozené typy zahrnují pole, řetězce, ukazatele a struktury.

Samozřejmě program také potřebuje prostředky pro identifikaci uložených dat. Vyšetříte jednu metodu používající proměnné, která to tak provádí. Dále se podíváte, jak se v C++ provádí aritmetické operace. Nakonec uvidíte, jak C++ konvertuje hodnoty z jednoho typu na druhý.

Jednoduché proměnné

Typický program musí uložit informaci – současnou cenu akcií IBM, průměrnou vlhkost v New Yorku v srpnu, nejčastější písmeno ve slově Constitution a jeho relativní četnost nebo počet dostupných imitátorů Elvise. Abychom uložili položku informace do počítače, musí mít program ponětí o třech základních vlastnostech:

- Kde je informace uložena
- Jaké jsou tam hodnoty
- Jaký druh informace je uložen

Témata kapitoly:

Pravidla pro pojmenování proměnných

Vestavěné celočíselné typy v C++: `unsigned long`, `long`, `unsigned int`, `int`, `unsigned short`, `short`, `char`, `unsigned char`, `signed char` a `bool`

Soubor `climits`, reprezentující systémové meze pro různé typy celočíselných proměnných

Numerické konstanty různých celočíselných typů

Použití kvalifikátoru `const` na vytvoření symbolických konstant

Vestavěné typy C++ s pohyblivou řádovou čárkou: `float`, `double` a `long double`

Soubor `cfloat`, reprezentující systémové meze pro typy s pohyblivou řádovou čárkou

Numerické konstanty různých typů pro pohyblivou řádovou čárku

Aritmetické operátory v C++

Automatická konverze typů

Vynucená konverze typů (přetypování)

Strategie, kterou dosud příklady používaly, je deklarování proměnné. Typ používaný v deklaracích popisuje druh informace a jména proměnných znázorňují symbolicky hodnotu. Například předpokládejme, že zástupce vedoucího laboratoře Igor používá následující příkazy:

```
int braincount;
braincount = 5;
```

Tyto příkazy programu říkají, že se ukládá celé číslo a jméno braincount představuje v našem případě celočíselnou hodnotu 5. V podstatě program lokalizuje dostatečně velký kousek paměti, aby se do něj vešlo celé číslo a na toto místo nakopíruje hodnotu 5. Tyto příkazy vám (nebo Igorovi) neříkají, kde je v paměti hodnota uložena, ale program o této informaci ví. Samozřejmě můžete použít operátor & na zjištění adresy braincount v paměti. Operátor probereme v následující kapitole, až budeme zkoumat druhou strategii identifikace dat – použití ukazatelů.

Jména proměnných

C++ vás podporuje v používání smysluplných jmen pro proměnné. Pokud proměnná cost představuje náklady na výlet (trip), nazvěte ji costoftrip nebo costOfTrip a ne pouze x nebo cot. Měli byste uposlechnout několik jednoduchých pravidel C++ pro pojmenování:

- Jediné znaky, které můžete použít ve jménu, jsou alfabetické znaky, číslice a znak podtržení (_).
- První znak ve jménu nemůže být číslice.
- Velká písmena se považují za jiná, než malá písmena.
- Nemůžete pro jména používat klíčová slova C++.
- Jména začínající dvěma nebo jedním znakem podtržení následovaná velkým písmenem jsou rezervována pro použití v implementaci. Jména začínající jediným znakem podtržení jsou rezervována pro implementaci jako globální identifikátory.
- C++ neklade žádná omezení na délku jmen a všechny znaky ve jménu jsou významné.

Předposlední bod se trochu liší od předcházejících bodů, protože použití názvu jako _time_stop nebo _Donut nezpůsobí chybu kompilátoru; místo toho to vede k neurčitému chování. Jinými slovy, nedá se říct, jaký bude výsledek. Příčina nespočívá v tom, že není chyba kompilátoru, jména jsou legální, ale spíše v tom, že jsou rezervována pro použití v implementaci. Část o globálních jménech se odvolává na to, kde jsou jména deklarována; kapitola 4 se tohoto tématu dotýká.

Poslední bod odlišuje C++ od ANSI C, který zaručuje, že prvních 31 znaků jména je významných. (V ANSI C jsou dvě jména, která mají prvních 31 znaků stejných, považována za stejná, dokonce i když se liší ve 32. znaku.)

Zde je několik platných a neplatných jmen v C++:

```
int poodle;    // platné
int Poodle;    // platné, ale různé od poodle
```

```
int POODLE;     // platné a dokonce se ještě více liší
Int terrier;    // neplatné - má být int, nikoli Int
int my_stars3   // platné
int _Mystars3; // platné, ale rezervované - začíná znakem podtržení
int 4ever;      // neplatné, protože začíná číslicí
int double;     // neplatné – double je klíčovým slovem C++
int begin;      // platné – begin je klíčovým slovem Pascalu
int __fools;    // platné, ale rezervované - začíná dvěma znaky podtržení
int the_very_best_variable_i_can_be_version_112;  // platné, velmi dlouhé
int honky-tonk;           // neplatné - pomlčka není povolena
```

Chcete-li vytvářet jméno ze dvou nebo více slov, obvyklá praxe spočívá v oddělení slov znakem podtržení, jako v `my_onions` nebo zavedením velkého počátečního písmene v každém slově, které následuje první, jako v `myEyeTooth`. (Veteráni z C mají sklon k používání metody se znakem podtržení podle zvyku z C, příznivci Pascalu dávají přednost cestě s velkými písmeny.) Oba tvary usnadňují nahlížení na jednotlivá slova a rozlišení mezi, řekněme `carDrip` a `cardRip` nebo `boat_sport` nebo `boats_port`.

Celočíselné typy

Celá čísla jsou čísla, která nemají desetinnou část, jako například 2, 98, -5286 a 0. Existuje mnoho celých čísel, předpokládejme, že pokládáte nekonečný počet za mnoho, takže žádné konečné množství paměti počítače nemůže zachytit všechna možná celá čísla. Proto může jazyk zachytit pouze podmnožinu celých čísel. Některé jazyky, jako je například standardní Pascal, nabízejí pouze jeden typ celého čísla (jeden typ se hodí na všechno!), C++ však poskytuje několik voleb. Dává vám možnost výběru celočíselného typu, který nejlépe odpovídá určitým požadavkům programu. Tato záležitost týkající se shody typu a dat je znamením navržených datových typů z OOP.

Různé celočíselné typy C++ se liší v objemu paměti, kterou používají pro celé číslo. Velký blok paměti může reprezentovat větší rozsah celočíselných hodnot. Rovněž některé typy (typy se znaménkem) mohou reprezentovat jak kladné tak záporné hodnoty. Základní celočíselné typy v C++, které jsou srovnány podle rostoucí velikosti, se nazývají `char`, `short`, `int` a `long`. Každý se vyskytuje jak ve verzi se znaménkem, tak bez znaménka. To vám dává volbu osmi různých celočíselných typů! Podívejme se na tyto typy podrobněji. Protože typ `char` má jisté zvláštní vlastnosti (nejčastěji se používá na reprezentaci znaků místo čísel), kapitola pojedná nejprve o ostatních typech.

Celočíselné typy short, int a long

Paměť počítače se skládá z jednotek, které se nazývají *bity*. (Podívejte se na následující poznámku Bity a bajty.) Typy C++, `short`, `int` a `long`, mohou použitím různého počtu bitů na uložení hodnoty představovat až tři různé velikosti celých čísel. Bylo by výhodné, kdyby měl každý typ pro všechny systémy vždy nějakou určitou velikost, například, pro `short` by bylo 16 bitů, pro `int` vždy 32 bitů a tak dále. Ale život není tak jednoduchý. Příčinou je to, že neexistuje jediný výběr, který je vhodný pro všechny návrhy počítačů. C++ nabízí přizpůsobivý standard s jistými zaručenými minimálními velikostmi. Tady je to, co dostanete:

- Celé číslo `short` má nejméně 16 bitů.
- Celé číslo `int` má přinejmenším tolik bitů jako `short`.
- Celé číslo `long` má alespoň 32 bitů a přinejmenším je tak velké jako `int`.

Bity a bajty

Základní jednotkou počítačové paměti je *bit*. Považujte ho za elektronický přepínač, který můžete nastavit na vypnuto nebo zapnuto. Vypnuto představuje hodnotu 0 a zapnuto hodnotu 1. 8bitový kousek paměti může být nastaven na 256 různých kombinací. Číslo 256 vychází ze skutečnosti, že každý bit má dvě možná nastavení, což vytváří pro 8 bitů celkový počet kombinací 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2, neboli 256. Proto 8bitová jednotka může reprezentovat řekněme hodnoty od 0 do 255 nebo hodnoty od –128 do 127. Každý dodatečný bit zdvojnásobuje počet kombinací. To znamená, že můžete 16bitovou jednotku nastavit na 65 536 různých hodnot a 32bitovou jednotku na 4 294 672 296 různých hodnot.

Bajt obvykle znamená 8bitovou jednotku paměti. Bajt je v tomto smyslu jednotkou míry, která popisuje objem paměti v počítači pomocí kilobajtu rovném 1 024 bajtů a megabajtu rovném 1 024 kilobajtů. C++ však definuje bajt odlišně. *Bajt* v C++ sestává alespoň z tolika přilehlých bitů, aby vyhověl základní znakové sadě implementace. To znamená, počet možných hodnot se musí rovnat nebo musí být větší než počet různých znaků. V USA jsou základními znakovými sadami obvykle ASCII a EBCDIC, každá z nich může být přizpůsobena 8 bitům, takže bajt v C++ je na systémech používajících tyto sady 8bitový. Avšak mezinárodní programování může vyžadovat mnohem větší znakové sady, jako například Unicode, takže některé aplikace mohou požívat 16bitový bajt.

Mnoho systémů v současné době uplatňuje minimální záruku, vytváří 16bitový `short` a 32bitový `long`. To ještě zanechává mnoho otevřených možností pro `int`. Mohla by mít 16, 24 nebo 32 bitů a vyhovět standardu. Typicky má `int` pro starší typy implementací IBM PC 16 bitů (stejně jako `short`) a 32 bitů (stejné jako `long`) pro Windows 95, Windows NT, Macintosh, VAX a řadu dalších minipočítačových implementací. Některé implementace vám dávají výběr, jak řídit `int`. (Co vaše implementace používá? Následující příklad vám ukáže, jak určit hranice vašeho systému, aniž byste museli otevřít manuál.) Rozdíly velikostí typů mezi implementacemi mohou způsobit problémy, pokud přesunete program v C++ z jednoho prostředí na jiné. Avšak trochu péče, jak se později v této kapitole dozvíte, může tyto problémy minimalizovat.

Použijte tato typová jména na deklarování proměnných přesně tak, jako byste použili `int`:

```
short score;            // vytváří proměnnou celočíselného typu short
int temperature;        // vytváří proměnnou celočíselného typu int
long position;          // vytváří proměnnou celočíselného typu long
```

Skutečně, `short` je zkratka pro `short int` a `long` pro `long int`, ale sotva někdo použije delší tvary. Tyto tři typy, `int`, `short` a `long`, jsou typy se znaménkem a mají ten význam, že každý rozděluje svůj rozsah téměř rovnoměrně mezi kladné a záporné hodnoty. Například 16bitové `int` se může měnit od –32 768 do 32767.

Pokud chcete vědět, jak vypadají celá čísla vašeho systému, tak vám C++ nabízí prostředky, které vám dovolují prozkoumat velikosti typů pomocí programu. Například operátor

sizeof navrací velikost typu nebo proměnné v bajtech. (Operátor je vestavěný prvek jazyka, který pracuje nad jednou nebo více položkami a produkuje hodnotu. Například operátor sčítání, reprezentovaný pomocí +, sčítá dvě hodnoty.) Všimněte si, že význam „bajt“ je implementačně závislý, takže dvoubajtové int by mohlo mít na jednom systému 16 bitů a na jiném 32. Za druhé, hlavičkový soubor climits (nebo pro starší implementace hlavičkový soubor limits.h) obsahuje informaci o mezích celočíselných typů. Především definuje symbolická jména, která reprezentují různé meze. Například definuje INT_MAX, což je největší možná hodnota int. Výpis programu 3.1 ukazuje, jak se tyto prostředky používají. Program také ukazuje *inicializaci*, což je použití deklaračního příkazu pro přidělení hodnoty proměnné.

Výpis programu 3.1 limits.cpp

```
// limits.cpp - některé hodnoty celočiselných mezi
#include <iostream>
using namespace std;
#include <climits>            // pro starší systémy použijte limits.h
int main()
{
    int n_int = INT_MAX;      // inicializace n_int na maximální hodnotu
int
    short n_short = SHRT_MAX; // symboly jsou definovány v souboru climits
    long n_long = LONG_MAX;

    // operátor sizeof poskytuje velikost typu nebo proměnné
    cout << "int is " << sizeof (int) << " bytes.\n";
    cout << "short is " << sizeof n_short << " bytes.\n";
    cout << "long is " << sizeof n_long << " bytes.\n\n";

    cout << "Maximum values:\n";
    cout << "int: " << n_int << "\n";
    cout << "short: " << n_short << "\n";
    cout << "long: " << n_long << "\n\n";

    cout << "Minimum int value = " << INT_MIN << "\n";
    return 0;
}
```

Kompatibilita:

Hlavičkový soubor climits je verzí C++ ANSI C hlavičkového souboru limits.h. Některé dřívější platformy C++ neměly dostupný ani jeden hlavičkový soubor. Pokud takové systémy používáte, musíte se omezit na zažití příkladu pouze v duchu.

Zde je výstup, který používá Microsoft Visual C++ 5.0:

```
int is 4 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 2147483647
short: 32767
long: 2147483647

Minimum int value = -2147483648
```

Tady je výstup pro druhý systém (Borland C++ 3.1 pro Dos):

```
int is 2 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 32767
short: 32767
long: 2147483647

Minimum int value = -32768
```

Poznámky k programu

Tato sekce shrnuje hlavní programové rysy tohoto programu.

Operátor `sizeof` oznamuje, že `int` na základním systému, jenž používá 8bitový bajt, zaujímá 4 bajty. Operátor `sizeof` můžete použít na jméno typu nebo jméno proměnné. Když používáte operátor `sizeof` se jménem typu, jako je `int`, uzavíráte jméno do závorek. Ale když používáte operátor se jménem proměnné, jako je n_short, tak jsou závorky volitelné:

```
cout << "int is " << sizeof (int) << " bytes.\n";
cout << "short is " << sizeof n_short << " bytes.\n";
```

Hlavičkový soubor `climits` definuje symbolické konstanty na zachycení typů mezí (viz poznámka Symbolické konstanty). Jak bylo řečeno, `INT_MAX` představuje největší hodnotu, kterou může typ `int` obsahovat; na výstupu vidíme, že má být pro náš systém Dos 32 767. Výrobce kompilátoru poskytuje soubor `climits`, který odráží vhodné hodnoty tohoto kompilátoru. Například, soubor `climits` pro systém používající 32bitové `int`, by mohl definovat `INT_MAX` tak, aby reprezentovala 2 147 483 647. Tabulka 3.1 shrnuje symbolické konstanty definované v tomto souboru; některé se týkají typů, o nichž jste se ještě neučili.

Tabulka 3.1 Symbolické konstanty ze souboru climits

Symbolická konstanta	Představuje
CHAR_BIT	počet bitů v char
CHAR_MAX	maximální hodnota char
CHAR_MIN	minimální hodnota char
SCHAR_MAX	maximální hodnota signed char
SCHAR_MIN	minimální hodnota signed char
UCHAR_MAX	maximální hodnota unsigned char
SHRT_MAX	maximální hodnota short
SHRT_MIN	minimální hodnota short
USHRT_MAX	maximální hodnota unsigned short
INT_MAX	maximální hodnota int
INT_MIN	minimální hodnota int
UINT_MAX	maximální hodnota unsigned int
LONG_MAX	maximální hodnota long
LONG_MIN	minimální hodnota long
ULONG_MAX	maximální hodnota unsigned long

Inicializace spojuje přiřazení s deklarací. Například příkaz

```
int n_int = INT_MAX;
```

deklaruje proměnnou `n_int` a nastavuje ji na největší možnou hodnotu typu `int`. Pro inicializaci hodnot můžete také použít obvyklé konstanty. Proměnnou můžete inicializovat pomocí jiné za předpokladu, že tato jiná proměnná byla definována první. Dokonce můžete inicializovat proměnnou pomocí výrazu, za předpokladu, že všechny hodnoty výrazu jsou v době kompilace známy:

```
int uncles = 5;                    // initializace uncles na 5
int aunts = uncles;                // initializace aunts na 5
int chairs = aunts + uncles + 4;   // initializace chairs na 14
```

Kdybyste přesunuli deklaraci `uncles` na konec tohoto seznamu příkazů, zrušili byste platnost dalších dvou inicializací, protože by hodnota `uncles` v době, kdy se kompilátor snaží inicializovat ostatní proměnné, nebyla známa.

Pamatujte:

Pokud neinicializujete proměnnou definovanou uvnitř funkce, hodnota je *nedefinovaná*. To znamená, že by hodnotou mohlo být cokoli, co je usazené v paměťové lokaci před vytvořením proměnné.

Pokud víte jaká by měla být počáteční hodnota proměnné, inicializujte ji. Pravda, oddělení deklarace proměnné od přiřazení hodnoty může vytvořit momentální nejistotu:

```
short year;        // co by to mohlo být?
year = 1492;       // jé
```

Ale inicializace proměnné v době vytváření vás chrání před pozdějším opomenutím přiřadit hodnotu.

Symbolické konstanty, postup preprocesoru

Soubor `climits` obsahuje řádky, které jsou podobné následujícímu:

```
#define INT_MAX 32767
```

Připomínáme, že proces kompilace nejprve předá zdrojový kód preprocesoru. Zde `#define`, podobně jako `#include`, jsou direktivy preprocesoru. To, co tato určitá direktiva preprocesoru říká, je: Prohlédni si program, zda tam nejsou instance `INT_MAX` a nahraď každý výskyt hodnotou `32 767`. Takže direktiva `#define` pracuje podobně jako příkaz globálního vyhledávání a nahrazování v editoru nebo v textovém editoru. Jakmile se tyto změny uskuteční, upravený program se zkompiluje. Preprocesor hledá nezávislé syntaktické jednotky (oddělená slova), přeskakuje zabudovaná slova. To znamená, že preprocesor nenahrazuje `PINT_MAXIM` hodnotou `P32767IM`. Také můžete použít `#define` pro své vlastní symbolické konstanty. (Viz výpis programu 3.2.) Direktiva `#define` je však přežitek z C. C++ má lepší způsob vytváření symbolických konstant (klíčové slovo `const`, o kterém bude pojednáno v pozdější sekci), takže byste `#define` neměli moc používat. Ale některé hlavičkové soubory, zvláště ty, které byly navrženy jak pro C tak pro C++ , ji používají.

Typy bez znaménka

Každý ze tří celočíselných typů, o kterých jste se právě učili, se vykytuje ve variantě bez znaménka, která nemůže obsahovat záporné hodnoty. To má výhodu ve zvětšení největší hodnoty, kterou může proměnná obsahovat. Například, jestliže `short` reprezentuje rozsah -32 768 až +32 767, potom verze bez znaménka reprezentuje rozsah 0 až 65 535. Samozřejmě, že můžete typy bez znaménka používat pouze pro veličiny, které nejsou nikdy záporné, jako například populace, stav zásob a výrazy šťastné tváře. Abyste ze základních celočíselných typů vytvořili verze bez znaménka, pouze použijte na změnu deklarace klíčové slovo `unsigned`:

```
unsigned short change;      // typ unsigned short
unsigned int rovert;        // typ unsigned int
unsigned quarterback;       // také typ unsigned int
unsigned long gone;         // typ unsigned long
```

Všimněte si, že `unsigned` je samo o sobě zkratka pro `unsigned int`.

Výpis programu 3.2 ilustruje použití typů bez znaménka. Také ukazuje, co by se mohlo přihodit, kdyby se program pokusil přejít za hranice celočíselných typů. Nakonec vám dává poslední pohled na příkaz preprocesoru `#define`.

Výpis programu 3.2 exceed.cpp

```
// exceed.cpp - překročení některých celočíselných mezí
#include <iostream>
using namespace std;
#define ZERO 0          // vytváří symbol ZERO pro hodnotu 0
```

```
#include <climits>  // definuje INT_MAX jako největší celočíselnou hodnotu
int main()
{
    short sam = SHRT_MAX;     // inicializuje proměnnou na maximální hodno-
tu
    unsigned short sue = sam;// v pořádku, je-li proměnná sam již defino-
vána
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nAdd $1 to each account.\nNow ";
    sam = sam + 1;
    sue = sue + 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nPoor Sam!\n";
    sam = ZERO;
    sue = ZERO;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\n";
    cout << "Take $1 from each account.\nNow ";
    sam = sam - 1;
    sue = sue - 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nLucky Sue!\n"; return 0;
}
```

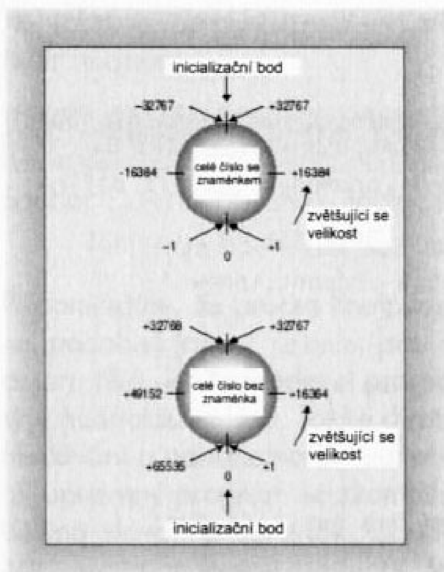
Kompatibilita:

Výpis programu 3.2, podobně jako 3.1, používá soubor `climits`; starší kompilátory možná musí použít `limits.h` a některé velmi staré možná nemají oba soubory dostupné.

Zde je výstup:

```
Sam has 32767 dollars and Sue has 32767 dollars deposited.
Add $1 to each account.
Now Sam has -32768 dollars and Sue has 32768 dollars deposited.
Poor Sam!
Sam has 0 dollars and Sue has 0 dollars deposited.
Take $1 from each account.
Now Sam has -1 dollars and Sue has 65535 dollars deposited.
Lucky Sue!
```

Program nastavuje proměnnou (`sam`) `short` a proměnnou (sue) `unsigned short` na největší hodnotu `short`, která je na našem systému 32 767. Potom ke každé hodnotě přičítá 1. To nezpůsobuje žádné problémy pro `sue`, protože nová hodnota je stále mnohem menší, než maximální hodnota celého čísla bez znaménka. Ale `sam` přechází z 32 767 na -32 768! Podobně odečtení 1 od nuly nevytváří pro `sam` žádný problém, ale nutí proměnnou bez znaménka `sue` přejít z 0 na 65 535. Jak můžete vidět, celá čísla se chovají skoro stejně jako počítadlo ujetých kilometrů nebo čítač VCR. Když překročíte mez, hodnoty začínají právě na druhé straně rozsahu. Viz obrázek 3.1. C++ zaručuje, že se typy bez znaménka chovají tímto způsobem. Avšak C++ nezaručuje, že celočíselné typy nemohou překročit své meze (přetečení a podtečení) bez potíží, ale to je nejběžnější chování na současných implementacích.



Obrázek 3.1 Typické chování celých čísel při přetečení

Jaký typ použít

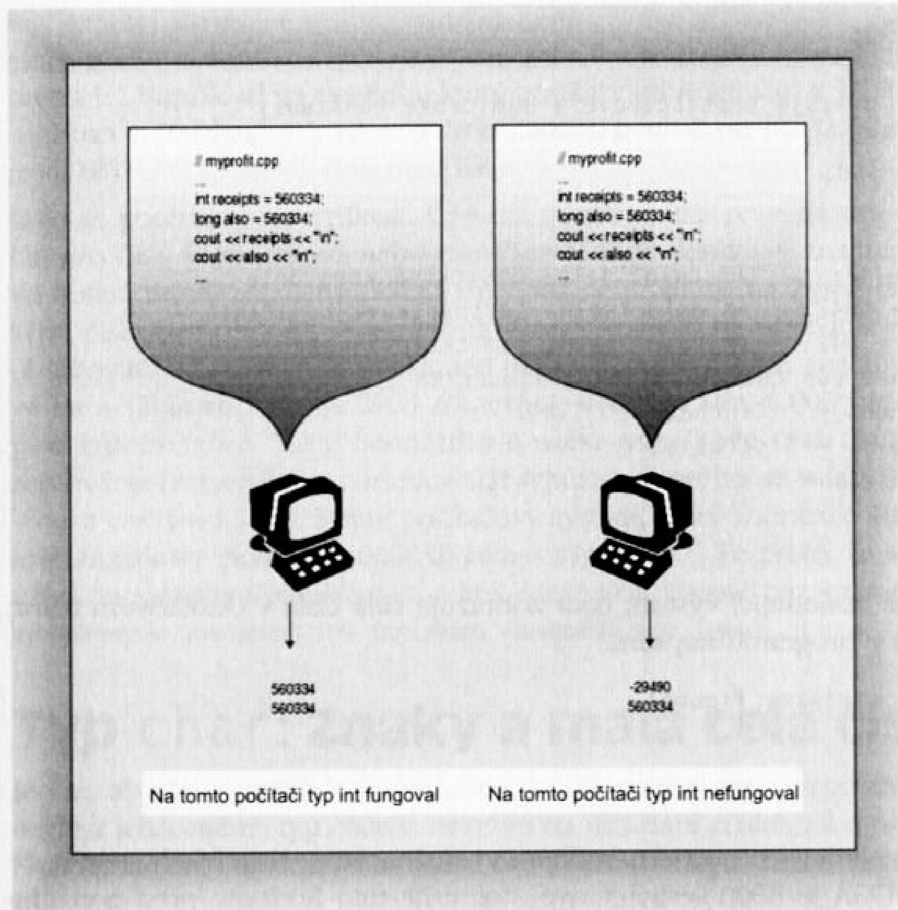
Jaký typ byste měli při takové bohatosti celočíselných typů použít? Obecně se `int` považuje pro cílový počítač za „nejpřirozenější" velikost celého čísla. *Přirozená velikost* znamená, že počítač zachází s tvarem celého čísla nejefektivněji. Neexistuje-li žádný vynucený důvod pro použití jiného typu, použijte `int`.

Nyní se podívejte na důvody, kdybyste mohli použít jiné typy. Jestliže proměnná představuje něco, co není nikdy záporné, jako například počet slov v dokumentu, můžete použít typ bez znaménka; tímto způsobem může proměnná představovat větší hodnoty.

Jestliže víte, že by proměnná musela obsahovat celočíselné hodnoty, které jsou příliš velké na 16bitové celé číslo, použijte `long`. To platí dokonce i tehdy, když má `int` na vašem systému 32 bitů. Tímto způsobem, přenášíte-li váš program na systém s 16bitovým `int`, program by vás nepřivedl do rozpaků náhodným selháním při správné činnosti. Viz obrázek 3.2.

Jestliže je `short` menší než `int`, použití `short` vám šetří paměť. Prakticky je to důležité pouze tehdy, máte-li velké pole celých čísel. (Pole je datová struktura, která se skládá z několik hodnot stejného typu sekvenčně seřazených v paměti.) Je-li důležité ušetřit prostor, měli byste použít `short` namísto `int`, dokonce i když jsou oba stejné velikosti. Předpokládejme například, že přesouváte váš program z 16bitového systému Dos PC na 32bitový `int` systému Windows NT. To duplikuje množství paměti potřebné na úschovu pole `int`, ale neovlivňuje požadavky na pole `short`. Pamatujte, ušetřený bit je získaný bit.

Potřebujete-li pouze jediný bajt, můžete použít `char`. Tuto možnost brzy vyzkoušíte.



Obrázek 3.2 Pro přenositelnost použijte pro velká celá čísla long

Celočíselné konstanty

Celočíselná konstanta je ta, kterou můžete napsat explicitně, jako 212 nebo 1776. C++, podobně jako C, vám dovoluje zapsat celá čísla pomocí tří různých číselných základů: základ 10 (dává mu přednost veřejnost), základ 8 (dává mu přednost starý Unix) a základ 16 (dávají mu přednost hardwaroví fandové). Dodatek A, „Číselné soustavy“, tyto soustavy popisuje; zde se podíváme na vyjádření v C++. C++ používá první nebo první dvě číslice na identifikaci základu číselné konstanty. Je-li první číslice v rozsahu 1–9, číslo má základ 10 (desítkový); tedy 93 má základ 10. Jestliže je první číslice 0 a druhá je v rozsahu 1-7, číslo má základ 8 (osmičkový); tedy 042 je osmičkové a je rovno 34 desítkově. Jestliže jsou první dva znaky 0x nebo 0X, číslo má základ 16 (šestnáctkový); tedy 0x42 je šestnáctkové a je rovno 66 desítkově. Pro šestnáctkové hodnoty znaky a–f a A–F reprezentují šestnáctkové číslice, které odpovídají hodnotám 10–15. 0xF je 15 a 0xA5 je 165 (10 krát 16 plus 5 krát jedna). Výpis programu 3.3 je ušitý na předvedení těchto tří číselných soustav.

Výpis programu 3.3 hexoct.cpp

```
// hexoct.cpp - ukazuje šestnáctkové a osmičkové konstanty
#include <iostream>
using namespace std;
int main()
{
    int chest = 42;     // celočíselná desítková konstanta
    int waist = 0x42;   // celočíselná šestnáctková konstanta
    int inseam = 042;   // celočíselná osmičková konstanta
    cout << "Monsieur cuts a striking figure!\n";
    cout << "chest = " << chest << "\n";
    cout << "waist = " << waist << "\n";
    cout << "inseam = " << inseam << "\n";
    return 0;
}
```

Standardně, jak ukazuje následující výstup, cout zobrazuje celá čísla v desítkovém tvaru, nehledě na to, jak jsou v programu napsána.

```
Monsieur cuts a striking figure!
chest = 42
waist = 66
inseam = 34
```

Pamatujte si, že tyto záznamy jsou pouze národními vymoženostmi. Například, čtete-li, že segment video paměti CGA je B000 šestnáctkově, nemusíte tuto hodnotu před použitím vašeho programu konvertovat do desítkové soustavy na 45 056. Místo toho jednoduše použijete 0xB000. Ale když napíšete hodnotu deset jako 10, 012 nebo 0xA, ukládá se v počítači stejným způsobem – jako binární hodnota (základ 2). Mimochodem, chcete-li zobrazit hodnotu šestnáctkově nebo osmičkově, můžete použít některé zvláštní rysy `cout`. Teď se do toho nepustíme, ale tuto informaci naleznete v kapitole 16, „Vstup, výstup a soubory". (Abyste tuto informaci získali, můžete kapitolu přelétnout a ignorovat vysvětlení.) Na vstupu rozpoznává `cin` zápis C++ v různých číselných soustavách. Takže když napíšete 0x20, `cin` to interpretuje jako šestnáctkovou hodnotu, ekvivalent 32.

Jak C++ určuje typ konstanty

Deklarace programu oznamují kompilátoru C++ typ určité celočíselné proměnné. Ale co konstanty? To jest, předpokládejme, že v programu reprezentujete číslo konstantou:

```
cout << "Year = " << 1492 << "\n";
```

Uloží program 1492 jako `int`, `long` nebo jako nějaký jiný celočíselný typ? Odpověď zní, že C++ ukládá celočíselné konstanty jako typ int, dokud není důvod udělat to jinak. Dva takové důvody existují, použijete-li zvláštní příponu, která indikuje určitý typ nebo jestliže je hodnota na `int` příliš velká.

Nejprve se podívejme na přípony. Jsou to písmena umístěná na konec číselné konstanty, která signalizují typ. Přípona `l` nebo `L` u celého čísla znamená, že celé číslo je konstanta typu long, přípona u nebo U signalizuje konstantu `unsigned int` a `ul` (v jakékoli kombi-

naci pořadí malých a velkých písmen) signalizuje typ konstanty `unsigned long`. (Protože malé písmeno l může vypadat jako číslice 1, měli byste pro přípony používat velké písmeno L.) Například na systému, který používá 16bitové `int` a 32bitové `long` je číslo 22022 uloženo v 16 bitech jako `int` a číslo 22022L je uloženo na 32 bitech jako `long`. Podobně jsou 22022LU 22022UL unsigned long.

Dále se podívejme na velikost. C++ má nepatrně jiná pravidla pro celá desítková čísla než má pro čísla šestnáctková nebo osmičková. (Zde desítkový znamená základ 10, přesně jako šestnáctkový znamená základ 16; výraz neimplikuje nezbytně desetinnou tečku.) Desítkové číslo bez přípony je reprezentováno nejmenším z následujících typů, který ho může obsahovat: `int`, `long` nebo unsigned long. Na počítačovém systému, který používá 16bitové int a 32bitové `long`, se 2000 znázorňuje jako typ `int`, 40000 jako typ `long` 3000000000 jako typ `unsigned long`. Šestnáctkové nebo osmičkové celé číslo bez přípony je reprezentováno nejmenším z následujících typů, do kterého se může vejít: `int`, `unsigned int`, `long` a `unsigned long`. Stejný počítačový systém, který zobrazuje 40000 jako `long`, zobrazuje šestnáctkový ekvivalent 0x9C40 jako `unsigned int`. To proto, že se šestnáctkové číslo používá na vyjádření paměťových adres, které jsou vlastně bez znaménka. Takže pro 16bitové adresy je `unsigned int` mnohem vhodnější než `long`.

Typ `char`: Znaky a malá celá čísla

Je čas, abyste se vrátili k poslednímu celočíselnému typu, typu `char`. Jak pravděpodobně tušíte z jeho jména, typ char je navržen na ukládání znaků, jako například písmen a číslic. Nyní, poněvadž ukládání čísel není pro počítač velkým kšeftem, ukládání písmen je jinou záležitostí. Programovací jazyky mají jednoduché východisko při použití číselného kódu pro písmena. Typ char je tudíž jiným typem celého čísla. Je zaručeno, aby byl dostatečně velký na zobrazení celého rozsahu základních symbolů na všech cílových počítačových systémech – všechna písmena, číslice, interpunkční znaménka a podobně. V praxi většina systémů podporuje méně než 256 druhů znaků, takže jediný bajt může reprezentovat celý rozsah. Nicméně, ačkoli se `char`nejčastěji používá pro zacházení se znaky, můžete ho také používat jako celočíselný typ, který je prakticky menší než `short`.

Nejběžnější sadou symbolů ve Spojených státech je znaková sada ASCII (vyslovuje se askey), která je popsána v příloze C, „Znaková sada ASCII". Číselný kód (ASCII kód) reprezentuje znaky této sady. Například 65 je kód pro znak A. Pro pohodlí tato kniha předpokládá ASCII kód ve svých příkladech. Avšak, implementace C++ používají jakýkoli kód, který je přirozený pro jejich hostitelský systém – například EBCDIC (vyslovuje se eb-se-dik) na sálovém počítači IBM. Ani ASCII ani EBCDIC neslouží dobře mezinárodním potřebám, C++ podporuje široký typ znaků, který může obsahovat větší rozsah hodnot, jako se například používají mezinárodní znakovou sadou Unicode. V této kapitole se o tomto typu `wchar_t` naučíte později.

Vyšetřete typ `char` ve výpisu programu 3.4.

Výpis programu 3.4 chartype.cpp

```
// chartype.cpp - typ char
#include <iostream>
```

```
using namespace std;
int main( )
{
        char ch;        // deklarujte proměnnou char
        cout << "Enter a character:\n";
        cin >> ch;
        cout << "Holla! ";
        cout << "Thank you for the " << ch << " character.\n";
        return 0;
}
```

Jako obvykle, označení `\n` představuje znak nového řádku. Tady je výstup:

```
Enter a character:
M
Holla! Thank you for the M character.
```

Zajímavou věcí je to, že napíšete `M`, nikoli odpovídající kód znaku `77`. Program také tiskne `M`, ne `77`. Avšak když se podíváte do paměti zjistíte, že v proměnné `ch` je uložena hodnota `77`. Kouzlo, jako je například toto, nespočívá v typu `char`, ale v `cin` a `cout`. Tyto výtečné prostředky provádějí konverze ve vašem zájmu. Na vstupu konvertuje `cin` klávesový vstup M na hodnotu `77`. Na výstupu `cout` konvertuje hodnotu `77` na zobrazený znak `M`; `cin` a `cout` se řídí typem proměnné. Když umístíte stejnou hodnotu `77` do proměnné `int`, potom ji `cout` zobrazí jako `77`. (To jest, `cout` zobrazí dva znaky `7`.) Výpis programu 3.5 tuto vlastnost ilustruje. Také ukazuje, jak psát v C++ znakovou konstantu. Uzavřete znak mezi dvě jednoduché uvozovky, jako je v `'M'`. (Všimněte si, že příklad nepoužívá dvojité uvozovky. C++ používá jednoduché uvozovky pro znaky a dvojité uvozovky pro řetězce. Jak se probírá v kapitole 4, jsou obě zcela různé.) Nakonec program představuje vlastnost `cout`, funkci `cout.put()`, která zobrazuje jediný znak.

Výpis programu 3.5 morechar.cpp

```
// morechar.cpp - typy char a int v protikladu
#include <iostream>
using namespace std;
int main()
{
     char c = 'M';          // přiřaďte ASCII kód M do c
     int i = c;             // uložte stejný kód do int
     cout << "The ASCII code for " << c << " is " << i << "\n";
     cout << "Add one to the character code:\n";
     c = c + 1;
     i = c;
     cout << "The ASCII code for " << c << " is " << i << '\n';
     // používá cout.put() na zobrazení znaku
     cout << "Displaying char c using cout.put(c): ";
     cout.put(c);
     // používá cout.put() na zobrazení znakové konstanty
     cout.put('!');
     cout << "\nDone\n";
```

```
    return 0;
}
```

Zde je výstup:

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char c using cout.put(c): N!
Done
```

Poznámky k programu

Označení 'M' reprezentuje číselný kód znaku M, takže inicializací proměnné char c na 'M', se c nastaví na hodnotu 77. Program potom přiřadí identickou hodnotu do proměnné int i. Jak c tak i mají hodnotu 77. Jak bylo dříve stanoveno, typ hodnoty řídí cout, jako by vybíral, jak má tuto hodnotu zobrazit – pouze další příklad chytrých objektů.

Protože je c skutečně celé číslo, můžeme na něj použít celočíselné operace, jako je například přičtení 1. To změní hodnotu c na 78. Program potom znovu nastaví i na tuto novou hodnotu. (Ekvivalentně můžete jednoduše přičíst 1 k i.) Ještě jednou, cout zobrazí verzi hodnoty char jako znak a verzi int jako číslo.

Skutečnost, že C++ reprezentuje znaky jako celá čísla je opravdovou výhodou, která usnadňuje manipulaci se znakovými hodnotami. Nemusíte používat zastaralé konverzní funkce na konvertování znaků do ASCII a zpět.

Nakonec program používá funkci cout.put() na zobrazení jak c tak znakové konstanty.

Členská funkce: cout.put()

Co je vlastně cout.put() a proč má tečku ve svém jménu? Funkce cout.put() je vaším prvním příkladem důležitého pojmu OOP v C++, členské funkce. Vzpomeňte si, že třída definuje jak reprezentovat data a jak s nimi manipulovat. Členská funkce patří do třídy a popisuje metodu na manipulaci s daty třídy. Třída ostream má například členskou funkci put(), která je navržena pro výstup znaků. Členskou funkci můžete použít pouze s určitým objektem třídy, jako například v tomto případě objekt cout. Abyste mohli použít členskou funkci třídy spolu s objektem jako například cout, použijete ke spojení jména objektu (cout) se jménem funkce (put()) tečku. Tečka se nazývá členským operátorem. Zápis cout.put() znamená použití členské funkce put()s objektem třídy cout. Samozřejmě se o tom budete učit podrobněji, až dosáhnete na třídy v kapitole 9, „Objekty a třídy". Nyní jedinými třídami, které máte, jsou třídy istream a ostream, a abyste se s tímto pojetím cítili pohodlněji, můžete s jejich členskými funkcemi experimentovat.

Členská funkce cout.put() poskytuje alternativu v použití operátoru << na zobrazení znaku. Tady byste mohli být zvědavi, proč je potřeba nějaká cout.put(). Před Verzí 2.0 C++ by cout zobrazoval znakové proměnné jako znaky, ale znakové konstanty, jako například 'M' a '\n', jako čísla. Problém spočíval v tom, že dřívější verze C++, podobně jako C, ukládaly znakové konstanty jako typ int. To jest, kód 77 pro 'M' býval uložen do 16bitové nebo 32bitové jednotky. Zatímco proměnné char typicky zaujímají 8 bitů. A příkaz jako

```
char c = 'M';
```

kopíroval 8 bitů (důležitých 8 bitů) z konstanty 'M' do proměnné c. To naneštěstí znamenalo, že se 'M' a c jevily cout zcela různě, dokonce i když obě obsahovaly stejnou hodnotu. Takže příkaz jako

```
cout << '$';
```

by vytiskl ASCII kód znaku $, místo aby jednoduše zobrazil $. Ale

```
cout.put('$');
```

by vytiskl znak, jak se požaduje. Nyní, po Verzi 2.0, C++ ukládá jednotlivé znakové konstanty jako typ `char`, ne jako typ `int`. To znamená, že `cout` nyní zachází se znakovými konstantami správně. C++ mohl vždy používat řetězec „`\n`" na odstartování nového řádku; nyní také může použít znakovou konstantu `'\n'`:

```
cout << "\n"; // používá řetězec
cout << '\n'; // používá znakovou konstantu
```

Řetězec je uzavřen do dvojitých uvozovek namísto do jednoduchých a může obsahovat více než jeden znak. Řetězce, dokonce jednoznakové řetězce, nejsou to samé, jako typ `char`. K řetězcům se vrátíme v následující kapitole.

Objekt `cin` ovládá několik různých způsobů čtení znaků ze vstupu. Můžete je snadno prozkoumat uplatněním programu, který používá cykly na čtení několika znaků, takže na toto téma se dostanete, až budete probírat cykly v kapitole 5, „Cykly a relační výrazy".

Konstanty char

Na psaní znakových konstant máte v C++ několik možností. Nejjednodušší zvolenou věcí týkající se běžných znaků, jako například písmen, interpunkčních znamének a číslic, je uzavření znaku do jednoduchých uvozovek. Tento zápis ručí za numerický kód znaku. Například systém ASCII rozumí následujícím shodám:

```
'A' je 65, ASCII kód pro A
'a' je 97, ASCII kód pro a
'5' je 53, ASCII kód pro číslici 5
' ' je 32, ASCII kód pro znak mezery
'!' je 33, ASCII kód pro vykřičník
```

Používání tohoto zápisu je lepší, než použití numerických kódů explicitně. Je jasnější a nepředpokládá určitý kód. Pokud systém používá EBCDIC, potom 65 není kód pro A, avšak `'A'` tento znak stále představuje.

Některé znaky nemůžete zavádět do programu přímo z klávesnice. Nemůžete například vytvořit znak nového řádku jako část řetězce stiskem klávesy Enter; namísto toho editovací program tento znak interpretuje jako požadavek na začátek nového řádku ve vašem zdrojovém souboru. Další znaky mají potíže, protože jazyk C++ je naplňuje zvláštním významem. Například znak dvojité uvozovky ohraničuje řetězce, takže ho sotva můžete strčit doprostřed řetězce. C++ má pro některé z těchto znaků zvláštní označení, která se nazývají *escape sekvence*, jak se ukazuje v Tabulce 3.2. Například \a představuje výstrahu, která vyšle na reproduktor vašeho terminálu zvukový signál nebo zazvoní jeho zvonkem. A \" symbolizuje znak dvojité uvozovky jako běžný znak, namísto oddělovače řetězce. V řetězcích nebo ve znakových konstantách můžete použít tyto zápisy:

```
char alarm = '\a';
cout << alarm << "Don't do that again!\a\n";
cout << "Ben \"Buggsie\" Hacker was here!\n";
```

Poslední řádek produkuje následující výstup:

```
Ben "Buggsie" Hacker was here!
```

Všimněte si, že jste považovali escape sekvenci, jako například `\a`, za regulérní znak, jako například `Q`. To jest, uzavíráte ho do jednoduchých uvozovek na vytvoření znakové konstanty a nepoužíváte je, když ho začleňujete jako část řetězce.

Tabulka 3.2 Kódy escape sekvencí v C++

Jméno proměnné	Symbol ASCII	C++ kód	Desítkový ASCII kód	Hexadecimální ASCII kód
nový řádek	NL (LF)	\n	10	0xA
horizontální tabulátor	HT	\t	9	0x9
vertikální tabulátor	VT	\v	11	0xB
návrat na předchozí znak	BS	\b	8	0x8
návrat vozu	CR	\r	13	0xD
zvuková výstraha	BEL	\a	7	0x7
zpětné lomítko	\	\\	92	0x5C
otazník	?	?	63	0x3F
jednoduchá uvozovka	'	\'	39	0x27
dvojitá uvozovka	"	\"	34	0x22

Nakonec můžete použít escape sekvence pro znaky založené na osmičkovém nebo šestnáctkovém kódu. Například CTRL-Z má ASCII kód 26, který je osmičkově 032 a šestnáctkově 0x1a. Tyto znaky můžete reprezentovat každou ze dvou escape sekvencí: `\032` nebo `\x1a`. Uzavřením do jednoduchých uvozovek z nich můžete vytvořit znakové konstanty, například '`\032`', a můžete je použít jako část řetězce, jako v „`hi\x1a there`".

Tip:

Máte-li možnost volby mezi použitím numerické nebo symbolické escape sekvence, jako je v `\0x8` proti `\b`, použijte symbolický kód. Numerická reprezentace se váže na určitý kód, například ASCII, avšak symbolická reprezentace pracuje se všemi kódy a je čitelnější.

Výpis programu 3.6 předvádí několik escape sekvencí. Používá znaku zvukové výstrahy na upoutání vaší pozornosti, znaku nového řádku na posunutí kurzoru (jeden malý krok pro kurzor, jeden obrovský krok pro skupinu kurzorů), znaku klávesa zpět na posunutí kurzoru o jedno místo do leva. (Houdini jednou maloval obraz řeky Hudson pouze za použití escape sekvencí; byl ovšem velkým mistrem úniku.)

Výpis programu 3.6 bondini.cpp

```
// bondini.cpp - použití escape sekvencí
#include <iostream>
using namespace std;
int main()
{
    cout << "\aOperation \"HyperHype\" is now activated!\n";
    cout << "Enter your agent code:________\b\b\b\b\b\b\b\b";
    long code;
    cin >> code;
    cout << "\aYou entered " << code << "...\n";
    cout << "\aCode verified! Proceed with Plan Z3!\n";
    return 0;
}
```

Kompatibilita:

Některé systémy C++ založené na kompilátorech před ANSI C neumí rozpoznat `\a`. Na systémech, které používají znakový kód ASCII, můžete `\a` nahradit pomocí `\007`. Některé systémy se mohou chovat odlišně, místo návratu na předchozí znak například zobrazují \b jako malý obdélník nebo znak vymažou.

Když spustíte program, tak na obrazovku vkládá následující text:

```
Operation "HyperHype" is now activated!
Enter your agent code:________
```

Po vytisknutí znaků podtržení program použije znaku návrat na předchozí znak na nastavení kurzoru na první znak podtržení. Potom můžete zavést svůj tajný kód a pokračovat. Zde je úplný běh programu:

```
Operation "HyperHype" is now activated!
Enter your agent code:42007007
You entered 42007007...
Code verified! Proceed with Plan Z3!
```

Char se znaménkem a bez znaménka

Na rozdíl od `int`, `char` standardně nemá znaménko. Ani není standardně bez znaménka. Pro implementaci C++ se volba vynechává, aby se vývojáři kompilátoru umožnilo co nejlépe tento typ přizpůsobit vlastnostem hardwaru. Pokud je pro vás životně důležité, aby měl `char` určité chování, můžete explicitně použít `signed char` nebo `unsigned char` jako například typy:

```
char fodo;              // může být se znaménkem, může být bez znaménka
unsigned char bar;      // rozhodně bez znaménka
signed char snark;      // rozhodně se znaménkem
```

Tyto rozdíly jsou zvláště důležité, používáte-li char jako numerický typ. Typ unsigned char prakticky reprezentuje rozsah 0 až 255 a signed char prakticky reprezentuje rozsah –128 až 127. Například předpokládejme, že chcete použít proměnnou char na uschování hodnoty tak velké jako 200. Na některých systémech to pracuje, na jiných selhává. Avšak pro tento účel můžete na jakýchkoli systémech úspěšně použít unsigned char. Na druhé straně, použijete-li proměnnou char na úschovu standardních znaků ASCII, skutečně nezáleží na tom, zda je char se znaménkem nebo bez znaménka, proto jednoduše můžete použít char.

Když potřebujete větší velikost: wchar_t

Programy možná zacházejí se znakovými sadami, které se nevejdou do hranic jediného 8bitového bajtu; například japonský kanja systém. C++ s tím zachází několika způsoby. Za prvé, jestliže je velká sada znaků základní znakovou sadou implementace, dodavatel kompilátoru může definovat char jako 16bitový bajt nebo větší. Za druhé, implementace může podporovat jak malou základní znakovou sadu, tak větší, rozšířenou znakovou sadu. Obvyklý 8bitový char představuje základní znakovou sadu a nový typ, který se jmenuje wchar_t (pro *široký datový typ*), může představovat rozšířenou znakovou sadu. Typ wchar_t je celočíselný typ s dodatečným prostorem na vyjádření větších rozšířených znakových sad, které se používají na systému. Tento typ má stejnou velikost jako jeden z celočíselných typů, který se nazývá základním typem. Volba základního typu závisí na implementaci, takže by mohl být unsigned short na jednom systému a int na jiném.

Rodina cin a cout považuje vstup a výstup za proudy char, proto nejsou vhodné pro zacházení s typem wchar_t. Poslední verze hlavičkového souboru iostream poskytuje paralelní možnosti ve formě wcin a wcout na zacházení s proudy wchar_t. Rozšířenou znakovou konstantu nebo řetězec můžete tedy vyjádřit prefixem L .

```
wchar_t bob = L'P';            // široká znaková konstanta
wcout << L"tall" << endl;      // výstup širokého znakového řetězce
```

Na systému s dvoubajtovým wchar_t tento kód ukládá každý znak do dvoubajtové jednotky paměti. Tato kniha nepoužívá široký znakový typ, ale měli byste si toho být vědomi, zvláště, když se začleníte do mezinárodního programování nebo do používání Unicodu.

Unicode

Unicode poskytuje řešení na zobrazení různých znakových sad poskytnutím standardních numerických kódů pro větší počet znaků a symbolů tím, že je seskupí do typu. Například ASCII je začleněný jako podmnožina Unicodu, takže latinské znaky ve Spojených státech, jako například A a Z mají stejnou reprezentaci pod oběma systémy. Avšak Unicode začleňuje další latinské znaky, jako například znaky používané v evropských jazycích; znaky z dalších abeced zahrnující znaky řecké, cyrilici, hebrejské, arabské, thajské a bengálské; a ideogramy, které se například používají v Číně a Japonsku. Unicode reprezentuje doposud přes 38 000 symbolů a stále je ještě ve výstavbě. Chcete-li se dozvědět více, můžete si prohlédnou webovou stránku asociace Unicode.

Nový typ bool

Výbor ANSI/ISO C++ pro standardy přijal nový typ (to znamená, nový pro C++), který se jmenuje `bool`. Jmenuje se na počest anglického matematika George Boola, který vyvinul matematickou představu zákonů logiky. Ve výpočtech může *booleovská proměnná* nabývat hodnot pravda nebo nepravda. V minulosti neměl C++, podobně C, žádný booleovský typ. Místo toho, jak ve větším detailu uvidíte v kapitole 5, „Cykly a relační výrazy" a 6, „Příkazy větvení a logické operátory", C++ interpretoval nenulovou hodnotu jako pravda a nulovou hodnotu jako nepravda. Avšak nyní na vyjádření pravdy a nepravdy můžete používat typ `bool` a předdefinované literály `true` a `false`, které tyto hodnoty představují. To jest, můžete vytvořit příkazy, které jsou podobné následujícím:

```
bool isready = true;
```

Literály `true` a `false` se mohou konvertovat do typu `int`, `true` se konvertuje na 1 a `false` na 0.

```
int ans = true;           // ans se přiřadí 1
int promise = false;      // promise se přiřadí 0
```

Jakákoli numerická hodnota nebo hodnota ukazatele se také může konvertovat implicitně (to jest bez explicitního přetypování) na hodnotu `bool`. Jakákoli nenulová hodnota se konvertuje na `true`, zatímco nulová hodnota se konvertuje na `false`.

```
bool start = -100;        // start se přiřadí true
bool stop = 0;            // stop se přiřadí false
```

Další kapitoly ukazují, jak se může tento typ hodit.

Kvalifikátor const

Vraťme se nyní k tématu o symbolických jménech konstant. Symbolické jméno může naznačit, co konstanta představuje. Tedy, používá-li program konstantu v několika místech, a vy musíte změnit hodnotu, můžete změnit právě jedinou definici symbolu. Poznámka o `#define` dříve v této kapitole (Symbolické konstanty a postup preprocesoru) slíbila, že C++ má lepší způsob zacházení se symbolickými konstantami. Tento způsob spočívá v použití klíčového slova `const` na modifikaci deklarace proměnné a inicializaci. Předpokládejme, že chcete například symbolickou konstantu na počet měsíců v roce. Zaveďte tento řádek do programu:

```
const int MONTHS = 12;  // Months je symbolická konstanta pro 12
```

Nyní můžete `MONTHS` použít v programu namísto 12. (Pouhé 12 může v programu reprezentovat počet palců ve stopě nebo počet koblih v tuctu, ale jméno `MONTHS` vám říká, co představuje.) Po inicializaci konstanty, jako například `MONTHS`, se nastaví její hodnota. Kompilátor vám následně nedovolí změnit hodnotu `MONTHS`. Například Borland C++ vydává chybové hlášení, které říká, že je překročena `Lvalue`. To je to samé hlášení, které dostanete, když se, řekněme, pokusíte přiřadit hodnotu 4 do 3. (`Lvalue` je hodnota, jako například proměnná, která se vyskytuje na levé straně přiřazovacího operátoru.) Klíčové slovo `const` se nazývá kvalifikátor, protože blíže určuje význam deklarace.

Psaní jména velkými písmeny vám pomůže, abyste si připomněli, že `MONTHS` je konstanta. Toto není v žádném případě univerzální konvence, ale pomáhá rozlišit konstanty od proměnných, když čtete program. Jinou konvencí je psaní pouze prvého znaku jména velkým písmenem. Ještě další konvencí je začínat konstantu písmenem k, jako například `kmonths`. A existují ještě další konvence. Mnoho organizací má určité kódovací konvence a očekává, že se jich budou jejich programátoři držet.

Obecný tvar pro vytvoření konstanty je tento:

```
const typ jméno = hodnota;
```

Všimněte si, že konstantu v deklaraci inicializujete. Následující postup není správný:

```
const int toes;    // hodnota toes v tomto okamžiku není definována
toes = 10;         // příliš pozdě!
```

Neposkytnete-li hodnotu, když deklarujete konstantu, skončí s nespecifikovanou hodnotou, kterou nemůžete modifikovat.

Máte-li základy z C možná cítíte, že příkaz `#define`, který se probíral dříve, již provádí práci odpovídajícím způsobem. Ale `const` je lepší. Z jednoho důvodu, dovoluje vám specifikovat explicitně typ. Za druhé, můžete použít pravidla o rozsahu platnosti C++ na omezení definic pro určité funkce nebo soubory. (Pravidla o rozsahu platnosti popisují, jak široce je jméno známé v různých modulech; budete se o tom podrobněji učit v kapitole 8, „Dobrodružství ve funkcích".) Za třetí, `const` můžete použít pro spletitější typy, jako jsou například pole a struktury, které přijdou na pořad v následující kapitole.

Tip:

Přicházíte-li do C++ z C a chystáte se použít `#define` na definování symbolické konstanty, použijte místo toho `const`.

ANSI C také používá kvalifikátor `const`. Jste-li obeznámeni s verzí ANSI C, měli byste si být vědomi, že verze C++ se trochu liší. Jeden rozdíl se vztahuje k pravidlům o rozsahu platnosti, kapitola 8 tento bod pokrývá. Dalším hlavním rozdílem je, že v C++ (ale ne v C) můžete použít hodnotu `konstanty` na deklaraci velikosti pole. Příklad uvidíte v následující kapitole.

Čísla v pohyblivé řádové čárce

Nyní, když jste viděli úplnou řadu celočíselných typů C++, podívejme se na typy v pohyblivé řádové čárce, které tvoří druhou hlavní skupinu základních typů C++. Tato čísla vám dovolují reprezentovat čísla s desetinnou částí, jako například kilometrový výkon benzínové nádrže M1 (056 MPG). Poskytují také mnohem větší rozsah hodnot. Je-li číslo příliš velké na to, aby se dalo reprezentovat typem `long`, například počet hvězd v naší galaxii (odhadováno 400 000 000 000), můžete použít jeden z typů pohyblivé řádové čárky. Pomocí typů pohyblivé řádové čárky můžete reprezentovat čísla, jako jsou například 2,5 a 3,14159 nebo 122442,32, to jest čísla s desetinnou částí. Počítač takové hodnoty ukládá do dvou částí. Jedna část reprezentuje hodnotu a druhá váhu, která hodnotu zvyšuje ne-

bo snižuje. Zde je analogie. Uvažujme dvě čísla 34,1245 a 34124,5. Jsou stejná kromě měřítka. První můžete reprezentovat jako 0,341245 (základní hodnota) a 100 (měřítko), Druhé můžete reprezentovat jako 0,341245 (stejná základní hodnota) a 100 000 (větší měřítko). Měřítko slouží k posunu desetinné čárky, odtud pojem pohyblivá řádová čárka. C++ používá na interní reprezentaci čísel v pohyblivé řádové čárce podobné metody, kromě toho, že je založena na binárních číslech, takže se posun měřítka děje pomocí faktoru 2 místo 10. Naštěstí o vnitřní reprezentaci nemusíte vědět mnoho. Hlavními body jsou, že čísla v pohyblivé řádové čárce vám dovolují vyjadřovat zlomky, velmi velké a velmi malé hodnoty a mají značně rozdílnou vnitřní reprezentaci oproti celočíselným typům.

Zápis čísel v pohyblivé řádové čárce

C++ má dva způsoby zápisu čísel v pohyblivé řádové čárce. První je standardní notace s desetinnou tečkou, kterou během vašeho života často používáte:

```
12.34           // pohyblivá řádová čárka
939001.32       // pohyblivá řádová čárka
0.00023         // pohyblivá řádová čárka
8.0             // stále pohyblivá řádová čárka
```

Dokonce i když je desetinná část rovna 0, jako například v 8.0, tak vám desetinná tečka zajišťuje, že se číslo ukládá ve formátu v pohyblivé řádové čárce a ne jako celé číslo.

Druhý způsob reprezentace hodnot s pohyblivou řádovou čárkou se nazývá zápisem E a vypadá například takto: `3.45E6`. To znamená, že číslo se násobí 1000000; E6 znamená 10 umocněno na 6, což je 1 následovaná 6 nulami. Proto `3.45E6` znamená 3450000. Číslo 6 se nazývá exponentem a číslo 3.45 se nazývá mantisou. Zde je více příkladů:

```
2.52e+8         // můžete použít E nebo e, + je volitelné
8.33E-4         // exponent může být záporný
7E5             // stejné jako 7.0E+05
-18.32e13       // může mít vepředu znaménko + nebo -
2.857e12        // státní dluh USA, 1989
5.98E24         // hmotnost Země v kilogramech
9.11e-31        // hmotnost elektronu v kilogramech
```

Jak jste si mohli všimnout, označení E je mnohem vhodnější pro velká a velmi malá čísla.

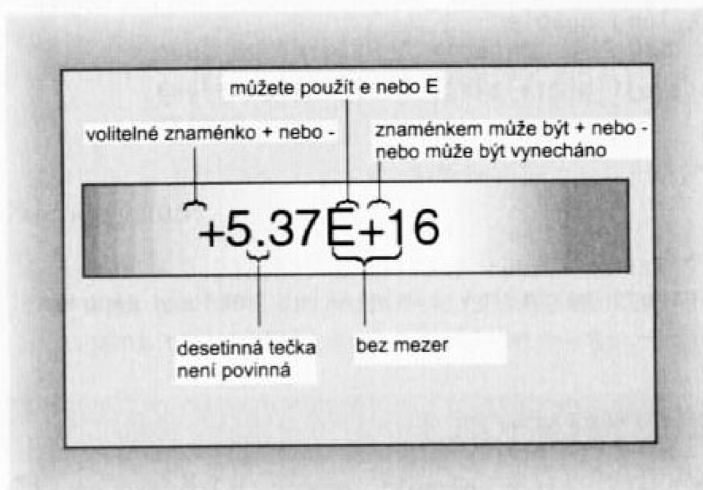
Označení E zaručuje, že je číslo uloženo ve formátu pohyblivé řádové čárky, dokonce i když nebyla použita desetinná tečka. Všimněte si, že můžete použít jak `E` tak `e` a exponent může mít jak kladné tak záporné znaménko (viz obrázek 3.3). Nemůžete však mít v čísle mezery: `7.2 E6` je chybné.

Použití záporného exponentu znamená dělení mocninou 10 namísto násobení mocninou 10. Takže 8.33E-4 znamená $8.33 \div 10^4$, nebo 0.000833. Podobně hmotnost elektronu 9.11e-31 kg znamená:

```
0.000000000000000000000000000000911 kg
```

Vyberte si. (Mimochodem všimněte si, že 911 je ve Spojených státech telefonní číslo pro případ nouze a že jsou telefonní zprávy přenášeny elektrony. Shoda okolností nebo vědecké spiknutí? (Buďte soudcem.) Všimněte si, že –8.33E4 znamená –83300. Znaménko

před číslem se aplikuje na hodnotu čísla, zatímco znaménko u exponentu se aplikuje na měřítko.



Obrázek 3.3 Označení E

Pamatujte:

Tvar d.dddE+n znamená posun desetinné tečky o n míst doprava a tvar d.dddE-n znamená posun desetinné tečky o n míst doleva .

Typy pohyblivé řádové čárky

Podobně jako ANSI C, má C++ tři typy v pohyblivé řádové čárce: `float`, `double` a `long double`. Tyto typy se popisují pomocí počtu významných číslic, které je mohou představovat, a minimálního přípustného rozsahu exponentů. *Podstatnými znaky* v čísle jsou významné číslice. Například, píšeme-li výšku Mt. Shasty v Kalifornii jako 14 162 stop, používá pět významných číslic, ale píšeme-li, že je Mt. Shasta asi 10 000 stop vysoká, používá dvě významné číslice, neboť je výsledek zaokrouhlen na nejbližších tisíc stop. V tomto případě tři číslice pouze rezervují místo. Počet významných číslic nezávisí na umístění desetinné tečky. Můžete například napsat 14.162 tisíc stop. Znovu, výraz používá pět významných číslic, protože hodnota má přesnost na pět číslic.

Ve skutečnosti jsou požadavky C a C++ na množství na významné číslice pro `float` alespoň 32 bitů, pro `double` alespoň 48 bitů a samozřejmě ne menší než pro `float` a pro `long double` alespoň tolik jako pro `double`. Všechny tři mohou mít stejnou velikost. Avšak prakticky má `float` 32 bitů, `double` 64 bitů a `long double` 80, 96 nebo 128 bitů. Také rozsah exponentů je pro tyto tři typy přinejmenším -37 až 37. Abyste zjistili meze vašeho systému, můžete se podívat do hlavičkových souborů `cfloat` nebo `float.h` (`cfloat` je C++ verze souboru `float.h` z C.) Zde se například dozvíte vstupy opatřené poznámkami ze souboru `float.h` pro Borland C++ Builder:

```
// následující definice představují minimální počty významných číslic
#define DBL_DIG 15              // double
#define FLT_DIG 6               // float
#define LDBL_DIG 18             // long double

// následující definice představují počty bitů, které se používají
// pro reprezentaci mantisy
#define DBL_MANT_DIG     53
#define FLT_MANT_DIG     24
#define LDBL_MANT_DIG    64

// následující definice představují maximální a minimální hodnoty exponen-
tů
#define DBL_MAX_10_EXP   +308
#define FLT_MAX_10_EXP   +38
#define LDBL_MAX_10_EXP  +4932

#define DBL_MIN_10_EXP   -307
#define FLT_MIN_10_EXP   -37
#define LDBL_MIN_10_EXP  -4931
```

Kompatibilita:

Některé implementace C++ ještě nedodaly hlavičkový soubor `cfloat` a některé implementace, které jsou založené na předchůdcích kompilátorů ANSI C, neposkytují hlavičkový soubor `float.h`.

Výpis programu 3.7 vyšetří typy `float` a `double` a to, jak se mohou lišit v přesnosti, ve které reprezentují čísla (to je význačná stránka rysu). Program ukazuje metodu `ostream` z kapitoly 16, „Vstup, výstup a soubory", která se nazývá `setf()`. Toto určité volání nutí výstup, aby zápis zůstal v pevné řádové čárce, takže můžete lépe sledovat přesnost. Zabraňuje to programu, aby se pro velká čísla přepnul do zápisu E a nutí ho, aby zobrazoval vpravo od desetinné tečky šest číslic. Parametry `ios_base::fixed` a `ios_base::floatfield`, jsou konstanty, které poskytuje přidružený `iostream`.

Výpis programu 3.7 floatnum.cpp

```
// floatnum.cpp - typy v pohyblivé řádové čárce
#include <iostream>
using namespace std;
int main()
{
    cout.setf(ios_base::fixed, ios_base::floatfield); //pevná řádová čárka
    float tub = 10.0 / 3.0;      // přesnost asi na 6 míst
    double mint = 10.0 / 3.0;    // přesnost asi na 15 míst
    const float million = 1.0e6;

    cout << "tub = " << tub;
```

```
    cout << ", a million tubs = " << million * tub;
    cout << ",\nand ten million tubs = ";
    cout << 10 * million * tub << "\n";

    cout << "mint = " << mint << " and a million mints = ";
    cout << million * mint << "\n";
    return 0;
}
```

Zde je výstup:

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

Kompatibilita:

C++ se nachází v procesu přeměny z `ios::fixed` na `ios_base::fixed` a z `ios::floatfield` na `ios_base::floatfield`. Pokud váš kompilátor neakceptuje tvary `ios_base`, zkuste místo toho `ios`. Starší verze C++, když zobrazují hodnoty v pohyblivé řádové čárce, zobrazují standardně šest číslic vpravo od desetinné tečky, jako v příkladu 2345.831541. Novější verze standardně zobrazují celek šesti číslic (2345.83) a poté, co hodnota dosáhne miliónu nebo více, tak se přepínají do E notace (2.34583E+06). Avšak nestandardní zobrazovací režimy, jako například `fixed` v příkladu, zobrazují napravo od desetinné tečky šest číslic jak pro starou, tak pro novou verzi.

Standardní nastavení také potlačuje zadní nuly, zobrazuje 23.4500 jako 23.45. Implementace se liší v tom, jak odpovídají na použití příkazu `setf()`, který přepisuje standardní nastavení. Starší verze, jako je například Borland C++ 3.1 pro DOS, potlačují zadní nuly v tomto režimu stejně dobře. Novější verze, jako například Microsoft Visual C++, Metrowerks CodeWarior a Borland C++ Builder, zobrazují nuly tak, jak je uvedeno na výpisu programu 3.7.

Poznámky k programu

Obvykle funkce `cout` koncové nuly nezobrazuje. Například by zobrazila 3333333.250000 jako 3333333.25. Volání `cout.setf()`, alespoň v nových implementacích, toto chování potlačuje. Hlavní, co zde stojí za povšimnutí, je to, že `float` má menší přesnost než `double`. Jak `tub`, tak `mint` jsou inicializovány na 10.0/3.0. To by se mělo rozvinout na 3.33333333333333333...(atd.). Protože `cout` tiskne napravo od desetinné tečky šest číslic, můžete vidět, že obě `tub` a `mint` jsou tak stejně přesné. Ale poté, co program vynásobí každé číslo milionem, vidíte, že se `tub` rozchází od přesné hodnoty po sedmé 3. Proměnná `tub` je přesná do sedmé významné číslice. (Tento systém zaručuje šest významných číslic, ale je nejhorší případ scénáře.) Proměnná typu double ukazuje třináct 3, takže je dobrá alespoň pro třináct významných číslic. Protože systém garantuje patnáct, nemělo by to být pro vás překvapením. Všimněte si také, že násobením `tub` desetkrát, nemělo za následek zcela správnou odpověď; to opět poukazuje na omezení přesnosti `float`.

Třída `ostream`, ke které `cout` patří, má členské funkce třídy, které vám umožňují přesné řízení toho, jak se výstup formátuje – šířky polí, místa vpravo od desetinné tečky, desítkový nebo E tvar atd. Kapitola 16 tyto volby vysvětluje. Příklady této knihy to zjednodušují a obvykle pouze používají operátor <<. Občas tento postup zobrazuje více číslic než je nezbytné, ale to pouze způsobuje estetické poškození. Máte-li chuť, přelétněte kapitolu 16, abyste viděli, jak se používají formátovací metody. Avšak neočekávejte, že se budeme v tomto bodě věnovat úplnému vysvětlení.

Konstanty v pohyblivé řádové čárce

Když napíšete v programu konstantu v pohyblivé řádové čárce, v jakém typu pohyblivé řádové čárky ji program ukládá? Standardně jsou konstanty v pohyblivé řádové čárce, jako například 8.24 a 2.4E8, typu `double`. Chcete-li, aby byla konstanta typu `float`, použijte příponu `f` nebo `F`. Pro typ `long double` použijte příponu `l` nebo `L`.

```
1.234f          // konstanta float
2.45E20F;       // konstanta float
2.345324E28     // konstanta double
2.2L            // konstanta long double
```

Výhody a nevýhody pohyblivé řádové čárky

Čísla v pohyblivé řádové čárce mají oproti celým číslům dvě výhody. Za prvé můžete reprezentovat čísla, která leží mezi celými čísly. Za druhé kvůli faktoru měřítka mohou reprezentovat mnohem větší rozsah hodnot. Na druhé straně, operace v pohyblivé řádové čárce jsou pomalejší než operace nad celými čísly, alespoň na počítačích bez matematického koprocesoru, a můžete také ztratit přesnost. Výpis programu 3.8 tento poslední bod ilustruje.

Výpis programu 3.8 fltadd.cpp

```
// fltadd.cpp - problém přesnosti u float
#include <iostream>
using namespace std;
int main()
{
    float a = 2.34E+22f;
    float b = a + 1.0f;

    cout << "a = " << a << "\n";
    cout << "b - a = " << (b - a) << "\n";
    return 0;
}
```

Kompatibilita:

Některé staré implementace C++ na kompilátorech před ANSI C nepodporují pro konstanty pohyblivé řádové čárky příponu f. Zjistíte-li, že jste tváří v tvář tomuto problému, můžete nahradit `2.34E+22f` pomocí `2.34E+22` a `1.0f` pomocí `(float) 1.0`.

Program vezme číslo, přičte 1 a potom odečte původní číslo. Výsledkem by měla být hodnota 1. Je? Tady je výstup z jednoho systému:

```
a = 2.34e+022
b - a = 0
```

Problém spočívá v tom, že 2.34+22 představuje číslo s 23 číslicemi vlevo od desetinného místa. Přidáním 1 se pokoušíte přidat 1 k 23 číslici tohoto čísla. Ale float reprezentuje pouze 6 nebo 7 prvních číslic z čísla, takže snaha změnit 23. číslici nemá žádný vliv na hodnotu.

Klasifikace typů

Typy `bool`, `char`, `wchar_t`, se označují za celočíselné typy bez znaménka. Typy `signed char`, `short`, `int` a `long` se nazývají celočíselné typy se znaménkem. Celočíselné typy se znaménkem a celočíselné typy bez znaménka se dohromady nazývají *integrální typy* nebo *celočíselné typy*. `float`, `double` a `long double` se nazývají *typy v pohyblivé řádové čárce*. Celočíselné typy a typy v pohyblivé řádové čárce se společně nazývají *aritmetické* typy.

Aritmetické operátory v C++

Možná, že jste si zahřívali paměť, když jste na základní škole prováděli aritmetická cvičení. Vašemu počítači můžete dopřát stejné potěšení. C++ na provádění výpočtů používá operátory. Poskytuje operátory pro pět základních aritmetických výpočtů: sčítání, odčítání, násobení, dělení a výpočet zbytku po celočíselném dělení. Každý z těchto operátorů používá na výpočet konečné odpovědi dvě hodnoty (které se nazývají operandy). Společně vytvářejí operátor a jeho operandy *výraz*. Například uvažujme následující příkaz:

```
int wheels = 4 + 2;
```

Hodnoty `4` a `2` jsou operandy, symbol + je operátorem sčítání a `4 + 2` je výraz, jehož hodnota je 6.

Zde je pět základních aritmetických operátorů C++:

- Operátor + sčítá své operandy. Například `4 + 20` dává hodnotu `24`.
- Operátor – odečítá druhý operand od prvého. Například `12 - 3` dává hodnotu 9.
- Operátor * násobí své operandy. Například `28 * 4` dává hodnotu 112.

- Operátor / dělí první operand druhým. Například 1000 / 5 dává hodnotu 200. Jsou-li oba operandy celá čísla, výsledkem je celá část podílu. Například 17 / 3 je 5 s odhozenou desetinnou částí.
- Operátor % najde zbytek po celočíselném dělení jeho prvního operandu s ohledem na druhý. To jest produkuje zbytek po dělení prvého operandu druhým. Například 19 % 6 je 1, protože se 6 vejde do 19 třikrát se zbytkem 1. Je-li jeden z operandů záporný, znaménko výsledku závisí na implementaci.

Samozřejmě můžete pro operandy použít proměnné stejně tak jako konstanty. Výpis programu 3.9 to právě dělá. Protože operátor % pracuje pouze s celými čísly, použijeme ho v pozdějším příkladu.

Výpis programu 3.9 arith.cpp

```
// arith.cpp - některé aritmetické výpočty v C++
#include <iostream>
using namespace std;
int main()
{
    float hats, heads;

    cout.setf(ios_base::fixed, ios_base::floatfield); // pevná řádová teč-
ka
    cout << "Enter a number: ";
    cin >> hats;
    cout << "Enter another number: ";
    cin >> heads;

    cout << "hats = " << hats << "; heads = " << heads << "\n";
    cout << "hats + heads = " << (hats + heads) << "\n";
    cout << "hats - heads = " << (hats - heads) << "\n";
    cout << "hats * heads = " << (hats * heads) << "\n";
    cout << "hats / heads = " << (hats / heads) << "\n";
    return 0;
}
```

Kompatibilita:

Jestliže váš kompilátor neakceptuje v `setf()` tvar `ios_base`, zkuste namísto toho starší `ios`.

Zde je vzorek výstupu. Jak vidíte, můžete C++ v provádění jednoduchých aritmetických výpočtů důvěřovat:

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.250000; heads = 11.170000
hats + heads = 61.419998
hats - heads = 39.080002
hats * heads = 561.292480
hats / heads = 4.498657
```

Dobrá, možná mu nemůžete důvěřovat úplně. Součet 11.17 s 50.25 by měl poskytnout 61.42 ale výstup oznamuje 61.419998. To není aritmetický problém; je to problém s omezenou kapacitou typu `float` na reprezentaci významných číslic.

Pamatujte, C++ zaručuje pro `float` pouze šest významných číslic. Zaokrouhlíte-li 61.419998 na šest číslic, dostanete 61.4200, což je správná hodnota k zaručené přesnosti. Zásadní je to, že potřebujete-li větší přesnost, použijte `double` nebo `long double`.

Jaké pořadí: Priorita operátorů a asociativita

Můžete C++ věřit v provádění komplikovaných aritmetických výpočtů? Ano, ale musíte vědět, jaká pravidla C++ používá. Například mnoho výrazů obsahuje více než jeden operátor. Mohou vyvstat otázky o tom, který operátor se aplikuje první. Uvažujme například tento příkaz:

```
int flyingpigs = 3 + 4 * 5;  // 35 nebo 23?
```

Vypadá to, že 4 je operandem pro oba operátory – pro + i pro *. Pokud se použije na stejný operand více než jeden operátor, C++ uplatní pravidla o prioritě na rozhodnutí, který operátor se použije první. Aritmetické operátory dodržují obvyklé algebraické pořadí, násobení, dělení a zjištění zbytku po celočíselném dělení se provádějí před sčítáním a odčítáním. Proto 3 + 4 * 5 znamená 3 + (4 * 5), nikoli (3 + 4) * 5. Takže odpověď je 23, ne 35. Samozřejmě na posílení vašich vlastních priorit můžete použít závorek. Dodatek D, „Priorita operátorů", ukazuje prioritu všech operátorů v C++. Všimněte si v něm, že *, / a % jsou všechny na stejném řádku. To znamená, že mají stejnou prioritu. Podobně sčítání a odčítání sdílejí nižší prioritu.

Občas seznam priorit nepostačuje. Uvažujte následující příkaz:

```
float logs = 120 / 4 * 5;    // 150 nebo 6?
```

Opět máme operand 4 pro dva operátory. Ale operátory / a * mají stejnou prioritu, takže sama priorita programu nepoví, zda má nejprve 120 dělit 4 nebo 4 násobit 5. Protože první volba vede k výsledku 150 a druhá k výsledku 6, je volba důležitá. Mají-li dva operátory stejnou prioritu, C++ se dívá, zda mají asociativitu zleva doprava nebo zprava doleva. Asociativita zleva doprava znamená, že mají-li dva operátory co do činění se stejným operandem, pak se levostranný operátor aplikuje první. Pro asociativitu zprava doleva se aplikuje první operátor na pravé straně. Informace o asociativitě jsou také v Dodatku D. Tam vidíte, že násobení a dělení jsou asociativní zleva doprava. To znamená, že použijete 4 nejprve s levým operátorem. To jest 120 dělíte 4, dostanete 30 jako výsledek a ten potom násobíte 5, abyste dostali 150.

Všimněte si, že pravidla o prioritě a asociativitě se dostávají do hry pouze tehdy, když dva operátory sdílejí stejný operand. Uvažujte následující výraz:

```
int dues = 20 * 5 + 24 * 6;
```

Priorita operátorů vám říká dvě věci: program musí před provedením sčítání vyčíslit 20 * 5 a program musí před provedením sčítání vyčíslit 24 * 6. Ale ani priorita ani asociativita neříká, které násobení se má provést první. Možná si myslíte, že asociativita říká provést

levé násobení, ale v tomto případě operátory nesdílejí společný operand, takže pravidla nelze použít. Ve skutečnosti to C++ nechává na implementaci, aby rozhodla, které pořadí pracuje na systému nejlépe. V našem příkladu dává obojí pořadí stejný výsledek, ale existují okolnosti, za kterých pořadí může vytvořit rozdíl. Jeden uvidíte v kapitole 5, kde se pojednává o přírůstkovém operátoru.

Odchylky při dělení

Musíte ještě vidět zbytek příběhu o operátoru dělení. Chování tohoto operátoru závisí na typu operandů. Jsou-li oba operandy celočíselné, C++ provádí celočíselné dělení. To znamená, že se od výsledku odhazuje celá desetinná část, což způsobí, že je výsledek celým číslem. Jsou-li jeden nebo oba operandy v pohyblivé řádové čárce, desetinná část zůstává, což způsobí, že je výsledek v pohyblivé řádové čárce. Výpis programu 3.10 ukazuje, jak dělení v C++ pracuje s různými typy hodnot. Podobně, jako ve výpisu programu 3.8, vyvolává členskou funkci `setf()` na přizpůsobení toho, jak se data zobrazí.

Výpis programu 3.10 divide.cpp

```
// divide.cpp - celočíselné dělení a dělení v pohyblivé řádové čárce
#include <iostream>
using namespace std;
int main()
{
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Integer division: 9/5 = " << 9 / 5  << "\n";
    cout << "Floating-point division: 9.0/5.0 = ";
    cout << 9.0 / 5.0 << "\n";
    cout << "Mixed division: 9.0/5 = " << 9.0 / 5  << "\n";
    cout << "double constants: 1e7/9.0 = ";
    cout << 1.e7 / 9.0 <<  "\n";
    cout << "float constants: 1e7f/9.0f = ";
    cout << 1.e7f / 9.0f <<  "\n";
    return 0;
}
```

Kompatibilita:

Nepřijme-li váš kompilátor v `setf()` tvary `ios_base`, zkuste místo toho použít starší tvary `ios`.

Některé implementace, založené na kompilátorech předcházejících ANSI C, nepodporují příponu f pro konstanty v pohyblivé řádové čárce. Střetnete-li se s tímto problémem, můžete nahradit `1.e7f / 9.0f` pomocí `(float) 1.7f /(float) 9.0`.

Některé implementace potlačují koncové nuly.

Zde je výstup pro jednu implementaci:

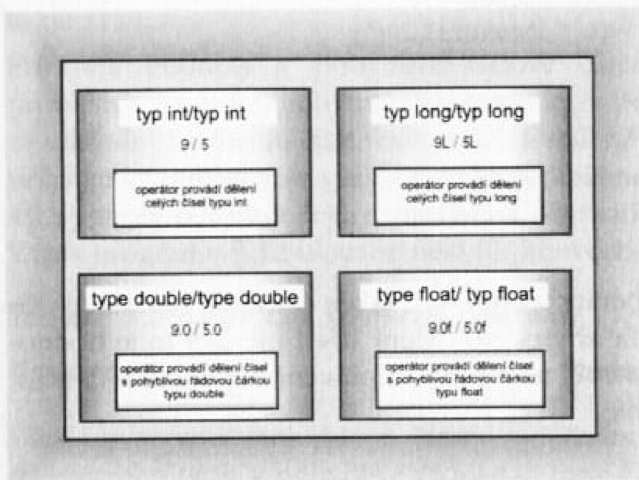
```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.800000
```

```
Mixed division: 9.0/5 = 1.800000
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125000
```

První výstupní řádek ukazuje, že dělení celého čísla 9 celým číslem 5 poskytuje celé číslo 1. Zlomková část 4 / 5 (nebo 0.8) se odhazuje. Pro tento druh dělení uvidíte praktické použití, když se budete učit o operátoru pro nalezení zbytku z celočíselného dělení. Následující dva řádky ukazují, že pokud je alespoň jeden z operandů v pohyblivé řádové čárce, dostanete výsledek 1.8 v pohyblivé řádové čárce. Skutečně, pokusíte-li se kombinovat smíšené typy, C++ konvertuje všechny typy, kterých se to týká, na stejný typ. O těchto automatických konverzích se v této kapitole dozvíte později. Relativní přesnost posledních dvou řádků ukazuje, že výsledkem je typ `double`, jsou-li oba operandy `double` a že je `float`, jsou-li oba operandy `float`. Pamatujte si, konstanty v pohyblivé řádové čárce jsou standardně typu `double`.

Letmý pohled na přetížení operátorů

Ve výpisu programu 3.10 představují operátory dělení tři různé operace: dělení `int`, dělení `float` a dělení `double`. C++ se chová podle kontextu, v tomto případě typu operandu, na určení, který operátor se míní. Postup, při kterém se používá stejného symbolu pro více než jednu operaci, se nazývá *přetížení operátoru*. C++ má několik příkladů přetížení, které jsou vestavěny do jazyka. C++ vám také dovoluje rozšířit přetížení operátoru na uživatelsky definované třídy, takže to, co tady vidíte je předzvěstí důležité vlastnosti OOP (viz obrázek 3.4).



Obrázek 3.4 Různá dělení

Operátor modulo (zbytek po celočíselném dělení)

Většině lidí jsou více než operátor modulo známy sčítání, odčítání, násobení a dělení, takže se na okamžik podívejte na tento operátor v akci. Operátor modulo, připomínáme, na-

vrací zbytek po celočíselném dělení. Spolu v kombinaci s celočíselným dělením je operátor modulo zvláště užitečný v problémech, které vyžadují rozdělení množství na různé celočíselné jednotky, jako například konvertování palců na stopy, nebo konvertování dolarů na čtvrťáky, desetníky, pěticenty a centy. V kapitole 2, „Vytyčení trasy k C++", výpis programu 2.6 konvertoval váhu v britských stone na libry. Výpis programu 3.11 tento proces obrací, konvertuje váhu v librách na stone. Stone, jak si pamatujete, je 14 liber a většina britských koupelnových měřidel se cejchuje v těchto jednotkách. Program používá celočíselné dělení na nalezení největšího čísla v celých stonech ve váze a používá operátor modulo na nalezení počtu liber, který je navíc.

Výpis programu 3.11 modulus.cpp

```
// modulus.cpp - používá operátor % na konverzi lbs na stone
#include <iostream>
using namespace std;
int main()
{
    const int Lbs_per_stn = 14;
    int lbs;

    cout << "Enter your weight in pounds: ";
    cin >> lbs;
    int stone = lbs / Lbs_per_stn;      // celé stone
    int pounds = lbs % Lbs_per_stn;     // zbytek v librách
    cout << lbs << " pounds are " << stone;
    cout << " stone, " << pounds << " pound(s).\n";
    return 0;
}
```

Zde je vzorek běhu programu:

```
Enter your weight in pounds: 184
184 pounds are 13 stone, 2 pound(s).
```

Ve výrazu `lbs / Lbs_per_stn`, jsou oba operandy typu `int`, takže počítač provádí celočíselné dělení. Součin 13 krát 14 je 182, takže zbytek po dělení 184 14 je 2, a to je hodnota `lbs % Lbs_per_stn`. Nyní jste, když ne citově, technicky připraveni odpovědět na otázky o vaší váze, cestujete-li do Velké Británie.

Konverze typů

Hojnost typů v C++ vám dovoluje nalézt potřebný typ. To komplikuje počítači také život. Například, sečtení dvou hodnot `short` může vyžadovat jiné hardwarové instrukce, než sečtení dvou hodnot `long`. S jedenácti celočíselnými typy a třemi v pohyblivé řádové čárce může mít počítač mnoho různých případů, se kterými zachází, zvláště, když začínáte typy míchat. Aby vám pomohl zacházet s tímto potenciálním zmatkem, C++ provádí mnoho typových konverzí automaticky:

- C++ konvertuje hodnoty, když přiřazujete hodnotu jednoho aritmetického typu do proměnné jiného aritmetického typu.

- C++ konvertuje hodnoty, když kombinujete ve výrazu smíšené typy.
- C++ konvertuje hodnoty, když předáváte parametry funkcím.

Když nerozumíte, co se v těchto automatických konverzích děje, možná se vám zdají některé výsledky programů záhadné, tak se na tato pravidla podívejme podrobněji.

Konverze při přiřazení

C++ je zcela liberální, když vám dovoluje přiřazovat numerickou hodnotu jednoho typu do proměnné jiného typu. Kdykoli tak činíte, hodnota se konvertuje na typ přijímací proměnné. Například předpokládejme, že `so_long` je typu `long`, `thirty` typu `short` a vy máte v programu následující příkaz:

```
so_long = thirty;          // přiřadí short do long
```

Program vezme hodnotu `thirty` (typicky 16bitová hodnota) a rozšíří ji během provádění programu na hodnotu `long` (typicky 32bitová hodnota). Všimněte si, že když se zakládá hodnota do místa v `so_long`, uskutečňuje se rozšíření; obsah `thirty` se nemění.

Přiřazení hodnoty do typu s větším rozsahem obvykle nevytváří problém. Například, přiřazení hodnoty `short` do proměnné `long`, nemění hodnotu; pouze dodává hodnotě více bajtů, ve kterých lenoší. Avšak přiřazení velké hodnoty `long`, jako například 2111222333 do proměnné `float` vyúsťuje ve ztrátu jisté přesnosti. Protože `float` může mít pouze šest významných číslic, hodnota se může zaokrouhlit na 2.11122E9. Tabulka 3.3 ukazuje některé možné konverzní problémy.

Nulová hodnota přiřazená do proměnné `bool` se konvertuje na `false`, a nenulová hodnota na `true`.

Přiřazení hodnoty v pohyblivé řádové čárce do celočíselného typu vnáší několik problémů. Za prvé, konvertování hodnoty v pohyblivé řádové čárce do celého čísla má za následek odříznutí čísla (odhození desetinné části). Za druhé, hodnota `float` může být příliš velká, než aby se vešla do úzké proměnné `int`. V tomto případě C++ nedefinuje, jaký by mohl být výsledek; to znamená, že různé implementace mohou odpovídat různě. Výpis programu 3.12 ukazuje několik konverzí při přiřazení.

Tabulka 3.3 Potenciální konverzní problémy

Konverze	Potenciální problémy
Větší typ v pohyblivé řádové čárce do menšího typu v pohyblivé řádové čárce, jako například double do float	Ztráta přesnosti (významné číslice), hodnota může být mimo rozsah cílového typu, v tomto případě je výsledek nedefinovaný
Typ v pohyblivé řádové čárce do celočíselného typu	Ztráta zlomkové části, původní hodnota může být mimo rozsah cílové hodnoty, v tomto případě může být výsledek nedefinovaný
Větší celočíselný typ do menšího celočíselného typu, jako například long do short	Původní hodnota může být mimo rozsah cílové hodnoty, ve skutečnosti se kopírují pouze bajty nižšího řádu

Výpis programu 3.12 assign.cpp

```
// assign.cpp - změny typu při přiřazení
#include <iostream>
using namespace std;
int main()
{
    float tree = 3;      // int se konvertuje na float
    int guess = 3.9832; // float se konvertuje na int
    int debt = 3.0E12;  // výsledek není v C++ definován
    cout << "tree = " << tree << "\n";
    cout << "guess = " << guess << "\n";
    cout << "debt = " << debt << "\n";
    return 0;
}
```

Zde je výstup z jednoho systému:

```
tree = 3
guess = 3
debt = 0
```

Tady se do three přiřazuje hodnota 3.0 v pohyblivé řádové čárce. Avšak protože cout nezobrazuje na výstupu koncové nuly, zobrazuje 3.0 jako 3. Přiřazení 3.9832 do int proměnné guess způsobuje, že se hodnota zkrátí na 3; C++ používá zkrácení (odříznutí zlomkové části) a nikoli zaokrouhlení (nalezení nejbližší celočíselné hodnoty), když konvertuje typy v pohyblivé řádové čárce na celočíselné typy. Nakonec si všimněte, že int proměnná debt je nezpůsobilá podržet hodnotu 3.0E12. To vytváří situaci, ve které C++ nedefinuje výsledek. Na tomto systému debt končí s hodnotou 0. Nuže, to je nebývalý způsob řešení masivní zadluženosti.

Některé kompilátory vás varují o možné ztrátě dat pro ty příkazy, které inicializují celočíselné proměnné hodnotami v pohyblivé řádové čárce. Tedy zobrazená hodnota debt se mění od kompilátoru ke kompilátoru. Například, provedení stejného programu na druhém systému vyprodukovalo hodnotu 2112827392.

Konverze ve výrazech

Dále uvažujme, co se přihodí, když zkombinujeme dva různé aritmetické typy v jednom výrazu. C++ v tomto případě vytváří dva druhy automatických konverzí. Za prvé, některé typy se automaticky konvertují, kdykoli se vyskytnou. Za druhé, některé typy se konvertují, když se kombinují ve výrazu s jinými typy.

Za prvé, vyzkoušíme automatické konverze. Když vyhodnotí výrazy, C++ konvertuje hodnoty bool, char, unsigned char, signed char a short na int. Především se true povyšuje na 1 a false na 0. Tyto konverze se nazývají *celočíselnými konverzemi*. Například uvažujme následující drůbežářské příkazy:

```
short chickens = 20;            // řádek 1
short ducks = 35;               // řádek 2
short fowl = chickens + ducks;  // řádek 3
```

Abychom provedli příkaz na řádku 3, program C++ vezme hodnoty chickens a duck a obě je konvertuje na int. Potom program konvertuje výsledek zpět na typ short, protože odpověď je přiřazena proměnné typu short. Možná to shledáváte trochu rozvláčné, ale má to význam. Typ int se obecně považuje za nejpřirozenější počítačový typ, což znamená, že počítač pravděpodobně s tímto typem provádí operace nejrychleji.

Existuje několik dalších celočíselných postupů: typ unsigned short se konvertuje na int, jestliže je short menší než int. Jestliže mají oba typy stejnou velikost, tak se unsigned short konvertuje na unsigned int. Toto pravidlo zajišťuje, že se během konverze na unsigned short neztrácejí žádná data. Podobně se wchar_t konvertuje na první z následujících typů, který je dostatečně veliký na přizpůsobení jeho rozsahu: int, unsigned int, long nebo unsigned long.

Potom existují konverze, ke kterým dojde, když aritmeticky kombinujete různé typy, jako například při přičítání int do float. Když operace zahrnuje dva typy, menší se konvertuje do většího. Například program ve výpisu programu 3.10 9.0 dělí 5. Protože 9.0 je typem double, program dříve, než provede dělení, konvertuje 5 na typ double. Obecněji, program prochází seznamem kvůli rozhodnutí, jakou má v aritmetickém výrazu provést konverzi. Zde je seznam, kterým kompilátor po řadě prochází:

1. Když je jeden operand typu long double, druhý operand se konvertuje na long double.
2. Jinak, když je jeden operand typu double, druhý operand se konvertuje na double.
3. Jinak, když je jeden operand typu float, druhý operand se konvertuje na float.
4. Jinak, když jsou operandy celočíselné typy, provede se celočíselná konverze.
5. V případě, že je jeden operand typu unsigned long, druhý se konverguje na unsigned long.
6. Jinak, je-li jeden operand long int a druhý unsigned int, konverze závisí na relativní velikosti obou typů. Jestliže může long reprezentovat možné hodnoty typu unsigned int, pak se unsigned int konvertuje na long.
7. Jinak se oba operandy konvertují na unsigned long.
8. Jinak, když je jeden nebo druhý operand long, pak se druhý konvertuje na long.
9. Jinak, když je jeden operand unsigned int, pak se druhý konvertuje na unsigned int.
10. Když kompilátor dosáhne tohoto bodu seznamu, oba operandy by se měly konvertovat na typ int.

ANSI C sleduje stejná pravidla jako C++, ale klasický K&R C měl pravidla trochu odlišná. Například klasický C konvertoval vždy float na double, dokonce i když byly oba operandy float.

Konverze při předávání parametrů

Obvykle vytváření prototypů funkcí při předávání parametrů v C++ řídí typ konverzí, jak se budete učit v kapitole 7, „Funkce – programové moduly C++". Avšak je pravděpodobné, ačkoli obvykle nemoudré, abychom se zřekli kontroly prototypu při předávání parametru. V tomto případě C++ aplikuje celočíselné konverze na typy char a short (signed a unsigned). Také, aby se zachovala kompatibilita s obrovským množstvím programo-

vého kódu v klasickém C, C++ konvertuje při předávání funkci parametry float na double, pokud se nevytvářel prototyp parametru typu float.

Přetypování

C++ vám také umožňuje vynutit konverze typu explicitně pomocí mechanizmu přetypování. (C++ je takovým mistrovským jazykem.) Přetypování přichází ve dvou formách. Například na konvertování hodnoty int uložené v proměnné, která se jmenuje thorn na typ long, můžete použít oba z následujících výrazů:

```
(long) thorn            // vytváří z hodnoty thorn typ long
long (thorn)            // vytváří z hodnoty thorn typ long
```

Přetypování nemění samu proměnnou thorn; spíše vytváří novou hodnotu naznačeného typu, který potom můžete použít ve výrazu.

Obecněji můžete udělat následující:

```
(jméno_typu) hodnota     // konvertuje hodnotu na typ jméno_typu
jméno_typu (hodnota)     // konvertuje hodnotu na typ jméno_typu
```

První tvar je pravé C. Druhý tvar je čisté C++. Myšlenka druhého tvaru spočívá ve vytvoření pohledu na přetypování jako na volání funkce. Vytváří přetypování pro vestavěné typy, které vypadají jako typové konverze, které můžete navrhnout pro uživatelské třídy.

Výpis programu 3.13 vám dává ukázku obou tvarů. Představte si, že první část tohoto výpisu je částí výkonného programu na modelování ekologie, který provádí výpočty v pohyblivé řádové čárce, které se konvertují na celé počty ptáků nebo zvířat. Výpočet za alky (auks) nejprve sečte hodnoty v pohyblivé řádové čárce a potom součet konvertuje při přiřazení na int. Ale výpočty pro netopýry (bats) a lysky (coots) nejprve používají přetypování pro zkonvertování hodnoty v pohyblivé řádové čárce na int a potom hodnoty sečte. Poslední část programu ukazuje, jak můžete použít přetypování na zobrazení ASCII kódu hodnoty typu char.

Výpis programu 3.13 typecast.cpp

```
// typecast.cpp - vynucení změny typu
#include <iostream>
using namespace std;
int main()
{
    int auks, bats, coots;

    // následující příkaz sečte hodnoty jako double,
    // potom konvertuje výsledek na int
    auks = 19.99 + 11.99;

    // tyto příkazy sečtou hodnoty jako int
    bats = (int) 19.99 + (int) 11.99;   // stará syntaxe v C
    coots = int (19.99) + int (11.99);  // nová syntaxe v C++
    cout << "auks = " << auks << ", bats = " << bats;
    cout << ", coots = " << coots << '\n';
```

```
    char ch = 'Z';
    cout << "The code for " << ch << " is ";    // tiskne se jako char
    cout << int(ch) << '\n';                    // tiskne se jako int
    return 0;
}
```

Zde je výsledek:

```
auks = 31, bats = 30, coots = 30
The code for Z is 90
```

Nejprve sečtení 19.99 s 11.99 poskytne 31.98. Když se tato hodnota přiřadí do `int` proměnné `auks`, zkrátí se na 31. Ale použití přetypování zkrátí dvě stejné hodnoty na 19 a 11 před sečtením, vytváří výsledek 30 jak pro `bats` tak pro `coots`. Poslední příkaz `cout`, než výsledek zobrazí, používá přetypování na konverzi hodnoty typu `char` na `int`. To způsobí, že `cout` vytiskne hodnotu jako celé číslo radši než jako znak.

Program ukazuje dva důvody pro použití přetypování. Za prvé, můžete mít hodnoty, které jsou uloženy jako typ `double`, ale používají se na výpočet hodnoty typu `int`. Třeba byste mohli upravit polohu vzhledem k mřížce nebo byste mohli modelovat celočíselné hodnoty pomocí čísel v pohyblivé řádové čárce, jako například soubory položek. Možná byste chtěli, aby výpočty zacházely s hodnotami jako `int`. Přetypování vám to umožňuje provádět takto přímo. Všimněte si, alespoň pro tyto hodnoty, že dostáváte různý výsledek, konvertujete-li na `int` a potom sčítáte, než když nejprve sečtete a potom konvertujete na `int`.

Druhá část programu ukazuje nejběžnější důvod přetypování – schopnost přinutit data v jednom tvaru, aby vyhověla různým očekáváním. V tomto programovém výpisu například `char` proměnná `ch` obsahuje kód pro písmeno Z. Uplatněním `cout` na `ch` se zobrazí znak Z, protože se `cout` zafixuje na skutečnost, že `ch` je typu `char`. Ale přetypováním ch na typ `int`, přinutíte `cout`, aby se přepnula na režim `int` a vytiskla ASCII kód uložený v `ch`.

Shrnutí

Základní typy C++ padají do dvou skupin. Jedna skupina sestává z hodnot, které se ukládají jako celá čísla. Druhá skupina sestává z hodnot, které se ukládají ve formátu pohyblivé řádové čárky. Celočíselné typy se jeden od druhého liší ve velikosti paměti, kterou používají na uložení hodnot a v tom, zda mají nebo nemají znaménko. V pořadí od nejmenšího do největšího existují tyto celočíselné typy `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned integer`, `long` a `unsigned long`. Existuje také typ `wchar_t`, jehož umístění do této posloupnosti velikostí závisí na implementaci. C++ zaručuje, že `char` je dostatečně velký na uchování jakéhokoli členu základní znakové sady systému, `wchar_t` může uchovávat libovolný člen rozšířené znakové sady systému, `short` je alespoň 16bitový, `long` je alespoň 32bitový a přinejmenším tak velký jako `int`. Přesné velikosti závisejí na implementaci.

Znaky jsou reprezentovány svými numerickými kódy. Systém V/V určuje, zda-li se kód interpretuje jako znak nebo jako číslo.

Typy pohyblivé řádové čárky reprezentují zlomkové hodnoty nebo hodnoty mnohem větší, než mohou reprezentovat celá čísla. Tři typy v pohyblivé řádové čárce se nazývají `float`, `double` a `long double`. C++ zaručuje, že `float` není větší než `double` a že `double` není větší než `long double`. Typicky `float` používá 32 bitů paměti, double používá 64 bitů a `long double` používá 80 až 128 bitů.

Poskytnutím rozmanitosti typů různých velikostí a ve variantách se znaménkem a bez znaménka vás C++ nechá přizpůsobit typ určitým datovým požadavkům.

Pro numerické typy používá C++ operátory, které poskytují obvyklou aritmetickou podporu: sčítání, odčítání, násobení, dělení a zjištění zbytku po celočíselném dělení. Snaží-li se dva operátory pracovat se stejnou hodnotou, pravidla priority a asociativity v C++ určují, které operace se provádějí nejdříve.

C++ konvertuje hodnoty z jednoho typu na jiný, když přiřazujete hodnoty do proměnné, mícháte typy v aritmetických operacích a používáte přetypování na vynucení konverze typů. Mnoho typových konverzí je „bezpečných", což znamená, že je můžete provádět bez ztráty nebo změny dat. Například můžete bez jakýchkoli problémů konvertovat hodnotu `int` na hodnotu `long`. Jiné, jako například konverze typů v pohyblivé řádové čárce na celočíselné typy, vyžadují více péče.

Napoprvé možná shledáváte velký počet základních typů C++ za trochu přehnaný, zvláště, když berete v úvahu různá pravidla konverze. Ale pravděpodobně časem naleznete příležitosti, kdy jeden z typů je právě to, co teď potřebujete a poděkujete C++, že ho má.

Opakovací otázky

1. Proč má C++ více než jeden celočíselný typ?
2. Definujte následující:

   a) Celé číslo `short` s velikostí 80.

   b) Celé číslo `unsigned int` s hodnotou 42110

   c) Celé číslo s hodnotou 3000000000
3. Jaké zabezpečení vám C++ poskytuje, aby vám zabránilo v překročení mezí celočíselných typů?
4. Jaký je rozdíl mezi 33L a 33?
5. Uvažujte dva následující příkazy v C++. Jsou ekvivalentní?

   ```
   char grade = 65;
   char grade = 'A';
   ```
6. Jak byste mohli použít C++, abyste zjistili, jakou znakovou hodnotu reprezentuje kód 88? Postupujte alespoň dvěma způsoby.
7. Přiřazení hodnoty `long` do `float` může mít za následek chybu zaokrouhlení. A co když přiřadíte `long` do `double`?
8. Vyčíslete následující výrazy tak, jak by to provedl C++:

```
a) 8 * 9 + 2
b) 6 * 3 / 4
c) 3 / 4 * 6
d) 6.0 * 3 / 4
e) 15 % 4
```

9. Předpokládejte, že `x1` a `x2` jsou dvě proměnné typu `double`, které chcete sečíst jako celá čísla a přiřadit je do celočíselné proměnné. Vytvořte příkaz v C++, který to udělá.

Programovací cvičení

1. Napište krátký program, který se zeptá na vaši výšku v celých palcích, a potom ji konvertuje na stopy a palce. Přimějte program, aby použil znak podtržení na naznačení, kde chcete napsat odpověď. Také použijte symbolickou konstantu const na reprezentaci konverzního činitele.
2. Napište krátký program, který se zeptá na vaši výšku ve stopách a palcích a vaši váhu v librách. (Použijte tři proměnné na uložení informace.) Přimějte program, aby vypsal vaše BMI (Body Mass Index – index tělesné hmotnosti). Abyste mohli vypočítat BMI, nejprve konvertujte vaši výšku ve stopách a palcích do vaší výšky v palcích. Potom konvertujte svou výšku v palcích na výšku v metrech násobením 0.0254. Potom konvertujte svou váhu v librách na váhu v kilogramech dělením 2.2. Konečně vypočítejte své BMI dělením váhy v kilogramech čtvercem váhy v metrech. Použijte symbolické konstanty na vyjádření různých konverzních faktorů.
3. Napište program, který se vás zeptá, kolik mil jste řídili a kolik galonů benzínu jste spotřebovali a potom podá zprávu o mílích na galon, které ujelo vaše auto. Nebo, preferujete-li, že se může program dotázat na vzdálenost v kilometrech, na benzín v litrech a potom vydat zprávu o výsledku v evropském stylu, v litrech na 100 kilometrů. (Nebo snad můžete použít litry na 100 kilometrů.)

KAPITOLA 4

Odvozené typy

Témata kapitoly:

- Vytváření a používání polí
- Vytváření a používání řetězců
- Metody getline() a get() pro čtení řetězců
- Směšování řetězcových a numerických vstupů
- Vytváření a používání struktur
- Vytváření a používání unionů
- Vytváření a používání výčtových typů
- Vytváření a používání ukazatelů
- Správa dynamické paměti pomocí operátorů new a delete
- Vytváření dynamických polí
- Vytváření dynamických struktur
- Automatické, statické a dynamické paměti

Vytvořili jste počítačovou hru, jež se jmenuje User-Hostile (uživatelsky nepřátelská), ve které se rozumem utkávají hráči s tajuplným a nemístným rozhraním počítače. Nyní musíte napsat program, který sleduje vaše měsíční prodeje po pět ročních období. Nebo chcete pořídit seznam hromadění svých obchodních karet počítačového fandy-hrdiny (hacker-hero trading cards).

Brzy dojdete k závěru, že potřebujete trochu víc, než jsou jednoduché základní typy, abyste vyhověli těmto datovým požadavkům a C++ vám nabízí něco navíc – odvozené typy. Jsou to typy odvozené ze základních celočíselných typů a typů v pohyblivé řádové čárce. Nejdále dosažitelným odvozeným typem je třída, ta lahůdka OOP, ke které se propracujeme. Ale C++ také podporuje několik skromnějších typů převzatých z C. Například pole může obsahovat několik hodnot stejného typu. Určitý druh pole může obsahovat řetězec, což je skupina znaků. Struktury mohou obsahovat několik hodnot různých typů. Pak existují ukazatele, což jsou proměnné, které říkají počítači, kde jsou umístěna data. V této kapitole vyzkoušíte všechny tyto odvozené tvary (mimo tříd) a také nakouknete na `new` a `delete` a na to, jak je můžete použít na správu dat.

Úvod do polí

Pole je datová forma, která může obsahovat několik hodnot, všechny stejného typu. Například pole může obsahovat 60 hodnot typu `int`, které představují data za pět let prodeje her, 12 hodnot `short`, které reprezentují počet dnů v měsíci nebo 365 hodnot `float`, které indikují vaše výdaje na potraviny každý den v roce. Každá hodnota se ukládá do samostatného prvku pole a počítač všechny tyto prvky pole ukládá za sebou do paměti. Na vytvoření pole použijete deklarační příkaz. Deklarace pole by měla signalizovat tři věci:

- Typ hodnoty, která má být uložena do každého prvku
- Jméno pole
- Počet prvků v poli

To v C++ provedete změnou deklarace jednoduché proměnné přidáním hranatých závorek, které obsahují počet prvků. Například deklarace

```
short months[12]; // vytváří pole 12 short
```

vytváří pole, které se jmenuje `months` a které má 12 prvků, z nichž každý může obsahovat hodnotu typu `short`. Každý prvek je v podstatě proměnná, kterou můžete považovat za jednoduchou proměnnou.

Obecný tvar deklarace pole je tento:

```
jméno_typu jméno_pole [velikost_pole];
```

Výraz `velikost_pole`, což je počet prvků, musí být konstantou, jako například 10 nebo hodnota `const`, nebo konstantní výraz, jako například `8 * sizeof (int)`, pro který jsou všechny hodnoty v době kompilace známé. Zejména nemůže být `velikost_pole` proměnnou, jejíž hodnota se nastaví během vykonávání programu. Avšak později vám tato kapitola ukáže, jak používat operátor `new`, abyste tato omezení obešli.

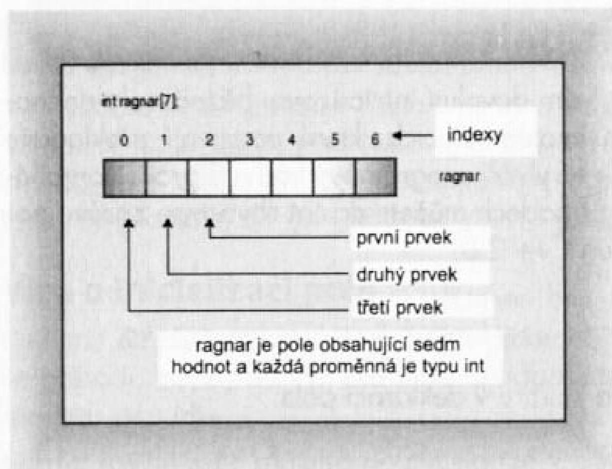
Pole jako odvozený typ

Pole se nazývá odvozeným typem, protože je založeno na jistých dalších typech. Nemůžete jednoduše deklarovat, že něco je polem; vždy to musí být pole jistého určitého typu. Neexistuje žádný zobecněný typ pole. Místo toho existuje mnoho specifických typů pole, jako například pole `char` nebo pole `long`. Například uvažujme tuto deklaraci:

```
float loans[20];
```

Typ `loans` neplatí pro „pole"; spíše je to pole `float` typů. Takhle se zdůrazňuje, že pole `loans` je odvozeno z typu `float`.

Mnoho užitečného, co se týká pole, pochází ze skutečnosti, že můžete přistupovat k prvkům pole jednotlivě. Způsob, jak se to dá realizovat, je použití indexu nebo ukazatele na očíslování prvků. Číslování pole v C++ začíná 0. (To je dané, musíte začínat od 0. Uživatelé Pascalu a Basicu se budou muset přizpůsobit.) C++ používá na určení prvku pole označení pomocí hranatých závorek s indexem. Například `months[0]` je prvním prvkem pole `month` a `months[11]` je jeho posledním prvkem. Všimněte si, že index posledního prvku je o jednotku menší než velikost pole. Viz obrázek 4.1. Deklarace pole vám tedy dovoluje vytvořit mnoho proměnných jedinou deklarací a vy potom můžete použít index na identifikaci jednotlivého prvku.



Obrázek 4.1 Vytvoření pole

Program na analýzu sladkých brambor ve výpisu programu 4.1 ukazuje několik vlastností polí, včetně deklarace pole, přiřazení hodnoty prvkům pole a inicializace pole.

Výpis programu 4.1 arrayone.cpp

```
// arrayone.cpp _ malá pole celých čísel
#include <iostream>
using namespace std;
int main()
{
    int yams[3];      // vytváří tříprvkové pole
    yams[0] = 7;      // přiřazuje hodnotu prvnímu prvku
    yams[1] = 8;
    yams[2] = 6;

    int yamcosts[3] = {20, 30, 5}; // vytváří, inicializuje pole
    // Poznámka: Jestliže váš C++ kompilátor nebo překladač neumí
    // inicializovat toto pole, použijte static int yamcosts[3] namísto
    // int yamcosts[3]

    cout << "Total yams = ";
    cout << (yams[0] + yams[1] + yams[2]) << "\n";
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
return 0;
}
```

Kompatibilita:

Současné verze C++ stejně tak jako ANSI C, vám dovolují inicializovat běžná pole definovaná ve funkci. Avšak v některých starších implementacích, které používají překladače C++ místo pravého kompilátoru, překladač C++ vytváří programový kód v C pro C kompilátor, který zcela nevyhovuje ANSI C. V těchto případech můžete dostat chybovou zprávu podobnou následujícímu příkladu ze systému Sun C++ 2.0:

```
"arrayone.cc", line 10: sorry, not implemented:
initialization of yamcosts (automatic aggregate) Compilation failed
```

Náprava se provede použitím klíčového slova `static` v deklaraci pole:

```
// initializace před ANSI
static int yamcosts[3] = {20, 30, 5};
```

Klíčové slovo `static` přiměje kompilátor, aby používal pro uložení pole jiné paměťové schéma, které umožňuje inicializaci dokonce pod kompilátory před ANSI C. Kapitola 8, „Příběhy ve funkcích", probírá `static` v sekci o paměťových třídách.

Zde je výstup:

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.

Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

Poznámky k programu

Za prvé, program vytváří tříprvkové pole, které se nazývá `yams`. Protože `yams` má tři prvky, prvky se číslují od 0 do 2 a `arrayone.cpp` používá hodnoty indexu 0–2, aby přiřadil těmto třem jednotlivým prvkům hodnoty. Každý jednotlivý prvek `yam` je `int` se všemi právy a privilegii typu `int`, takže `arryone.cpp` může provádět a provádí přiřazení hodnoty prvkům, sčítání, násobení a zobrazení prvků.

Program používá na přiřazení hodnot prvkům yam zdlouhavý způsob. C++ vás také nechá inicializovat prvky pole v deklaračním příkazu. Výpis programu 4.1 používá na přiřazení hodnot do pole yamcosts tento zkrácený zápis:

```
int yamcosts[3] = {20, 30, 5};
```

Jednoduše používá seznam hodnot oddělený čárkou (inicializační seznam), který je uzavřený do složených závorek. Mezery v seznamu nejsou povinné. Pokud neinicializujete pole, které je definované uvnitř funkce, hodnoty prvků zůstanou nedefinované. To znamená, že prvek může obsahovat jakoukoli hodnotu, která se dříve nacházela v této lokaci paměti.

Dále program používá hodnoty pole na několik výpočtů. Tato část programu vypadá se všemi indexy a složenými závorkami přecpaně. Cyklus `for`, který se objeví v kapitole 5

„Cykly a relační výrazy", poskytne výkonný způsob práce s poli a vyloučí potřebu psát každý prvek explicitně. Prozatím se budeme držet malých polí.

Vzpomeňte si, že operátor `sizeof` navrací velikost typu nebo datového objektu v bajtech. Všimněte si, že když používáte operátor `sizeof` se jménem pole, dostanete počet bajtů celého pole. Avšak když používáte sizeof s prvkem pole, dostanete velikost prvku v bajtech. To ukazuje, že `yams` je pole, `yams[1]` je pouze `int`.

Více o inicializaci pole

C++ má na inicializaci pole několik pravidel. Omezují, když ji můžete provést, určují, co se přihodí, když počet prvků pole neodpovídá počtu hodnot v inicializátoru. Vyzkoušejme tato pravidla.

Formu inicializace můžete použít pouze tehdy, když definujete pole. Nemůžete ji použít později a nemůžete přiřadit jedno celé pole do druhého:

```
int cards[4] = {3, 6, 8, 10};        // v pořádku
int hand[4];                         // v pořádku
hand[4] = {5, 6, 7, 9};              // není povoleno
hand = cards;                        // není povoleno
```

Avšak vždy na přiřazení hodnot do prvků pole jednotlivě můžete použít indexy.

Inicializujete-li pole, můžete poskytnout méně hodnot než je prvků pole. Například následující příkaz inicializuje pouze první dva prvky `hotelTips`:

```
float hotelTips[5] = {5.0, 2.5};
```

Inicializujete-li pole částečně, kompilátor nastaví zbývající prvky na nulu. Je tedy snadné inicializovat všechny prvky pole na nulu – inicializujete explicitně pouze první prvek pole na nulu, a potom necháte kompilátor, aby na nulu inicializoval zbývající prvky.

```
float totals[500] = {0};
```

Necháte-li při inicializaci pole hranaté závorky prázdné, kompilátor C++ za vás spočte prvky pole. Předpokládejme například, že jste vytvořili následující deklaraci:

```
short things[] = {1, 5, 3, 8};
```

Kompilátor vytvoří čtyř prvkové pole `things`.

Nechte kompilátor, ať to udělá.

Normálně je ubohým způsobem nechat kompilátor spočítat počet prvků, protože jeho výpočet může být jiný, než zamýšlíte. Avšak, jak brzy uvidíte, tento přístup může být bezpečnější na inicializaci znakového pole řetězcem. A je-li vaším největším znepokojením to, že počítač, ne vy, ví, jak velké je pole, můžete udělat něco podobného jako je toto:

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things / sizeof (short);
```

Zda je to užitečné nebo pomalé, záleží na okolnostech.

Řetězce

Řetězec je skupina znaků, která se ukládá do za sebou jdoucích bajtů paměti. C++ má dva způsoby práce s řetězci. První, převzatý z C, se často nazývá řetězcovým stylem C, je to metoda, kterou se zde naučíme. Kapitola 15, „Třída string a astandardní knihovna šablon", pokračuje alternativní metodou na knihovně třídy `string`. Mezitím, myšlenka řady znaků uložených do za sebou jdoucích bajtů má za následek, že můžete ukládat řetězce do pole `char`, jehož každý znak se uchovává ve svém vlastním prvku pole. Řetězce poskytují vhodný způsob ukládání textové informace, jako například zprávy pro uživatele („Prosím řekněte mi své tajné číslo švýcarského účtu:") nebo jeho odpovědi („Musíte si dělat legraci"). Styl řetězců v C má zvláštní vlastnost: Posledním znakem každého řetězce je *prázdný znak*. Tento znak, psáno \0, je znakem s ASCII kódem 0 a slouží pro označení konce řetězce. Například uvažujme následující dvě deklarace:

```
char dog [5] = {'b', 'e', 'a', 'u', 'x'};          // není řetězcem!
char cat[5] = {'f', 'a', 't', 's', '\0'};          // je řetězcem!
```

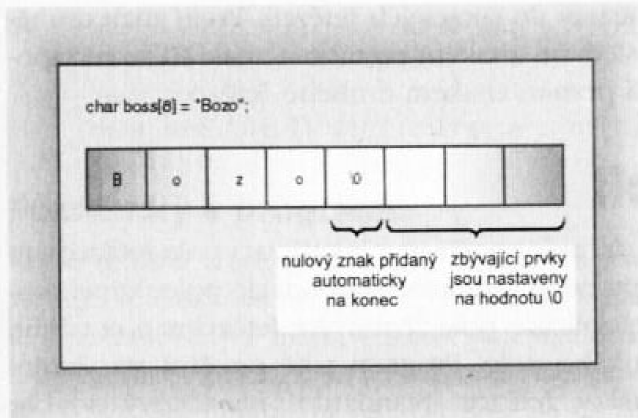
Obě pole jsou pole `char`, ale pouze druhé je řetězcem. Prázdný znak hraje základní roli v řetězcích stylu C. Například C++ má mnoho funkcí, které pracují s řetězci, včetně těch, které byly použity v `cout`. Všechny fungují tak, že zpracovávají řetězce znak po znaku, dokud nedosáhnou nulového znaku. Požádáte-li `cout`, aby zobrazil výše uvedený hezký řetězec `cat`, zobrazí první čtyři znaky, detekuje prázdný znak a zastaví se. Avšak budete-li dostatečně neslušný a řeknete `cout`, aby zobrazil výše uvedené pole `dog`, které není řetězcem, `cout` vytiskne pět znaků pole a potom pokračuje v mašírování pamětí bajt po bajtu, a dokud nedosáhne nulového znaku, vyhodnocuje každý znak jako vhodný pro tisk. Protože nulové znaky, což jsou skutečně bajty nastavené na nulu, mají sklon často se v paměti vyskytovat, havárie se zvládne obvykle rychle; přesto byste neměli považovat neřetězcová znaková pole za řetězce.

Příklad pole `cat`, který provádí inicializaci pole řetězcem, vypadá obtížně – všechny ty jednoduché uvozovky a pamatovat si prázdný znak. Nedělejte si starosti. Existuje lepší způsob inicializace znakových polí řetězcem. Použijete pouze řetězec v uvozovkách, který se nazývá *řetězcová konstanta* nebo *řetězcový literál*, jak vidíte zde:

```
char bird[10] = "Mr. Cheep";        // \0 se rozumí samo sebou
char fish[] = "Bubbles";            // nechá kompilátor počítat
```

Řetězce v uvozovkách zahrnují koncový prázdný znak implicitně, takže mu nemusíte rozumět. Viz obrázek 4.2. Také různé dovednosti C++ pro čtení řetězce z klávesnicového vstupu do pole `char` za vás automaticky přidávají ukončovací prázdný znak. (Jestliže za běhu programu z výpisu programu 4.1 objevíte, že musíte použít klíčové slovo `static` na inicializaci pole, musíte ho s těmito poli `char` také použít.)

Samozřejmě byste si měli být jisti, že je pole dostatečně veliké na to, aby mohlo obsahovat všechny znaky řetězce, včetně znaku null. Inicializace znakového pole řetězcovou konstantou je jedním způsobem, kde by mohlo být bezpečnější nechat za vás kompilátor spočítat počet znaků. Když vytváříte pole větší než je řetězec, neexistuje žádné jiné poškození, než plýtvání prostorem. Je to proto, že funkce, které pracují s řetězci, se řídí umístěním nulového znaku, nikoli velikostí pole. C++ neklade žádné meze na délku řetězců.



Obrázek 4.2 Inicializace pole řetězcem

Pamatujte:

Když určujete minimální velikost pole nezbytnou na udržení řetězce, nezapomeňte vzít v úvahu koncový prázdný znak.

Všimněte si, že řetězcová konstanta (dvojité uvozovky) je nezaměnitelná se znakovou konstantou (jednoduché uvozovky). Znaková konstanta, jako například 'S', je zkráceným označením kódu znaku. Na ASCII systému je 'S' jiným způsobem zápisu 83. Proto příkaz

```
char shirt_size = 'S'; // toto je v pořádku
```

přiřazuje hodnotu 83 do shirt_size. Ale "S" reprezentuje řetězec sestávající ze dvou znaků, znaků S a \0. Ještě horší, "S" ve skutečnosti představuje adresu paměti, na které je řetězec uložen. Takže příkaz jako

```
char shirt_size = "S"; // nepovolené míchání typů
```

se pokouší přiřadit paměťovou adresu do shirt_size. Protože je adresa izolovaným typem v C++, kompilátor C++ nepřipustí tento druh nesmyslu. (K tomuto bodu se vrátíme později po probrání ukazatelů.) Avšak C, které je při ověřování souhlasu typů mnohem shovívavější, nechá příkaz projít s varováním a výsledkem je nesmysl.

Spojování řetězců

Občas mohou být řetězce příliš dlouhé, aby se pohodlně vešly na jede řádek programového kódu. C++ vám umožňuje spojovat řetězcové konstanty; to jest kombinovat dva řetězce v apostrofech do jednoho. Vskutku, libovolné dvě řetězcové konstanty, oddělené pouze oddělovačem (mezery, tabulátory a znaky nového řádku) se automaticky spojují do jednoho. Tedy všechny následující příkazy výstupu jsou ekvivalentní jeden druhému:

```
cout << "I'd give my right arm to be" " a great violinist.\n";
cout << "I'd give my right arm to be a great violinist.\n";
cout << "I'd give my right ar"
        "m to be a great violinist.\n";
```

Všimněte si, že spojení nepřidává žádné mezery do spojených řetězců. První znak druhého řetězce bezprostředně následuje za posledním znakem prvního a znak \0 se nezapočítává. Znak \0 prvního řetězce se nahradí prvním znakem druhého řetězce.

Použití řetězců v poli

Dva nejběžnější způsoby dosazení řetězce do pole spočívají v inicializaci pole řetězcovou konstantou a načítání klávesnicového vstupu nebo vstupu ze souboru do pole. Výpis programu 4.2 ukazuje tyto přístupy pomocí inicializace pole znakovým řetězcem a použitím `cin` na umístění vstupního řetězce do druhého pole. Program také používá standardní knihovní funkci `strlen()` na zjištění délky řetězce. Standardní hlavičkový soubor `cstring` (nebo `string.h` pro starší implementace) poskytuje deklarace pro mnoho dalších funkcí, které souvisejí s řetězci.

Výpis programu 4.2 strings.cpp

```
// strings.cpp _ ukládá řetězce do pole
#include <iostream>
#include <cstring>  // kvůli funkci strlen()
using namespace std;
int main()
{
    const int Size = 15;
    char name1[Size];                   // prázdné pole
    char name2[Size] = "C++owboy";   // inicializované pole
    // Poznámka: některé implementace mohou vyžadovat na inicializaci pole
    // name2 klíčové slovo static
    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;
    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof name1 << " bytes.\n";
    cout << "Your initial is " << name1[0] << ".\n";
    name2[3] = '\0';            // null character
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << "\n";
    return 0;
}
```

Kompatibilita:

Pokud váš systém neposkytuje hlavičkový soubor `cstring`, zkuste starší verzi `string.h`.

Tady je ukázka běhu programu:

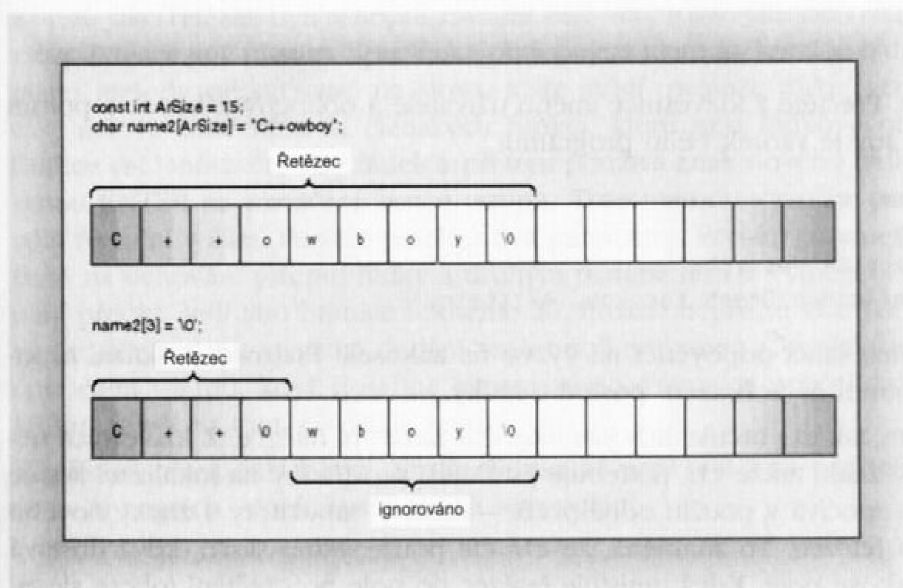
```
Howdy! I'm C++owboy! What's your name?
Basicman
```

```
Well, Basicman, your name has 8 letters and is stored
in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my name: C++
```

Poznámky k programu

Co se můžete z tohoto příkladu naučit? Za prvé, všimněte si operátoru `sizeof`, který poskytuje velikost celého pole, 15 bajtů, ale funkce `strlen()` navrací velikost řetězce, který je v poli uložený a nikoli velikost pole samotného. Tedy `strlen()` počítá pouze viditelné znaky a nikoli prázdný znak. Proto navrací pro délku `Basicman` hodnotu 8, ne 9. Je-li `cosmic` řetězcem, minimální velikost pole na uložení tohoto řetězce je `strlen(cosmic) + 1`.

Protože `name1` a `name2` jsou pole, můžete na přístup ke konkrétnímu znaku pole použít index. Například program používá `name1[0]` na nalezení prvního znaku pole. Program tedy nastavuje `name2[0]` na prázdný znak. To vytváří konec řetězce po třech znacích, dokonce i když v poli zůstává více znaků. Viz obrázek 4.3.



Obrázek 4.3 Zkrácení řetězce pomocí \0

Všimněte si, že program používá pro velikost pole symbolickou konstantu. Velikost pole se často vyskytuje v několika příkazech programu. Uplatnění symbolické konstanty na zachycení velikosti pole zjednodušuje úpravu programu na použití jiné velikosti pole; musíte pouze změnit jedinou hodnotu tam, kde je konstanta definovaná.

Dobrodružství na vstupu řetězce

Program `string.cpp` má skvrnu, která byla skryta kvůli často používané technice pečlivě vybraného vstupního vzorku. Výpis programu 4.3 odstraňuje závoj a ukazuje, že vstup řetězce může být ošidný.

Výpis programu 4.3 instr1.cpp

```
// instr1.cpp - čtení více než jednoho řetězce
#include <iostream>
using namespace std;
int main()
{
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin >> name;
    cout << "Enter your favorite dessert:\n";
    cin >> dessert;
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Záměr je jednoduchý. Přečtěte z klávesnice jméno uživatele a oblíbený zákusek a potom zobrazte informaci. Tady je vzorek běhu programu:

```
Enter your name:
Alistair Dreeb
Enter your favorite dessert:
I have some delicious Dreeb for you, Alistair.
```

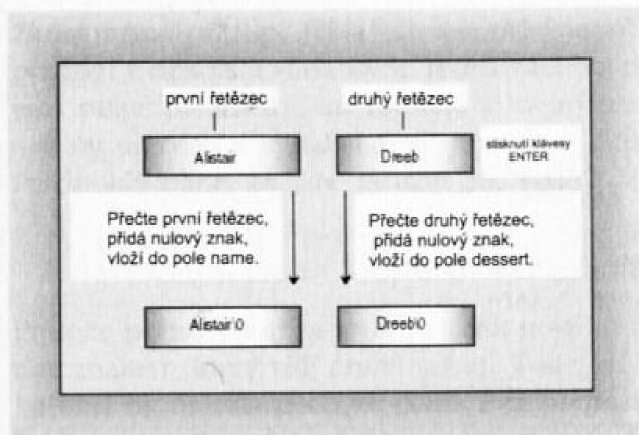
Nedostali jsme dokonce šanci odpovědět na výzvu na zákusek! Program ji ukázal a potom se okamžitě přesunul na zobrazení poslední řádky.

Problém spočívá v tom, jak `cin` určuje, kdy jste ukončili zadávání řetězce. Z klávesnice nemůžete zadat prázdný znak, takže `cin` potřebuje jisté další prostředky na lokalizaci konce řetězce. Technika `cin` spočívá v použití oddělovače – mezery, tabulátory a znaky nového řádku – na zobrazení řetězce. To znamená, že `cin` čte pouze jedno slovo, když dostává vstupní údaje pro znakové pole. Když umisťuje řetězec do pole po přečtení tohoto slova, `cin` automaticky přidává ukončovací prázdný znak.

Praktickým důsledkem tohoto příkladu je, že `cin` přečte `Alistair` jako celý první řetězec a umístí ho do pole name. Toto zanechá ubohé `Dreeb` stále sedět ve vstupní frontě. Když `cin` prohledává vstupní frontu na odpověď na otázku o oblíbeném zákusku, nalezne tam ještě `Dreeb`. `cin` zhltne `Dreeb` a pošle ho do pole dessert. Viz obrázek 4.4.

Dalším problémem, který se nevynořil na povrch v ukázkovém běhu programu je, že by se mohlo stát, že by vstupní řetězec mohl být delší, než je přijímací pole. Použití `cin`, jak ukázal tento příklad, nenabízí ochranu proti umístění 30znakového řetězce do 20znakového pole.

Mnoho programů závisí na řetězcovém vstupu, takže bude mít cenu rozvíjet toto téma chvíli dál. Musíme se zastavit u několika dalších pokročilých rysů, které se popisují v kapitole 16, „Vstup, výstup a soubory“.



Obrázek 4.4 Pohled cin na vstupní řetězec

Řádkově orientovaný vstup: getline() a get()

Abyste jako řetězec byli schopni zavádět celé věty místo jednotlivých slov, potřebujete pro vstup řetězce jiný přístup. Specificky potřebujete metodu, která je řádkově orientovaná, namísto metody orientované na slova. Máte štěstí, protože třída `istream`, jejíž ukázkou je `cin`, má ve třídě několik členských funkcí, které jsou řádkově orientované. Například funkce `getline()` čte celý řádek a při tom používá znak nového řádku, který se vysílá klávesou ENTER na označení konce vstupu. Tuto metodu vyvoláte použitím `cin.getline()` jako funkční volání. Funkce používá dva parametry. Prvním parametrem je jméno pole určené na uchování vstupní řádky a druhým parametrem je vymezení počtu znaků, které se mají přečíst. Je-li tato hranice řekněme 20, funkce nepřečte více jak 19 znaků a zanechává prostor pro automatické dodání znaku null na konec. Členská funkce `getline()` ukončuje čtení vstupu, když dosáhne této numerické hranice nebo když přečte znak nového řádku, záleží na tom, co přijde první.

Například předpokládejme, že chcete použít `getline()` na přečtení jména do pole `name` o 20 znacích. Mohli byste použít toto volání:

```
cin.getline(name,20);
```

Toto přečte celý řádek do pole `name` za předpokladu, že řádek obsahuje 19 nebo méně znaků. (Členská funkce `getline()` má také volitelný třetí parametr, o němž pojednává kapitola 16.)

Výpis programu 4.4 upravuje výpis programu 4.3 na použití `cin.getline()` namísto jednoduchého `cin`. Jinak se program nemění.

Výpis programu 4.4 instr2.cpp

```
// instr2.cpp _ čtení více než jednoho slova pomocí getline
#include <iostream>
using namespace std;
int main()
{
    const int ArSize = 20;
```

```
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin.getline(name, ArSize);  // čte celý řádek
    cout << "Enter your favorite dessert:\n";
    cin.getline(dessert, ArSize);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Kompatibilita:

Některé ranné verze C++ plně neimplementují všechny stránky současného balíku V/V v C++. Zvláště členská funkce `getline()` není vždy dostupná. Jestliže se vás to týká, o tomto příkladu si pouze přečtěte a pokračujte dalším, který používá členskou funkci, jež předchází `getline()`. Ranné verze Turbo C++ implementují `getline()` trochu rozdílně, takže ukládají znak nové řádky do řetězce. Microsoft Visual C++ 5.0 má v `getline()` chybu, která se nachází v hlavičkovém souboru `iostream`, ale nikoli ve verzi `ostream.h`.

Zde je několik ukázek výstupu:

```
Enter your name:
Melanie Ploops
Enter your favorite dessert:
Raspberry Torte
I have some delicious Raspberry Torte for you, Melanie Ploops.
```

Program nyní přečte úplné jméno a doručí uživatelce její oprávněný zákusek! Funkce `getline()` obyčejně získá řádek najednou. Přečte vstup až do znaku nového řádku, který označuje její konec, ale neuchovává ho. Místo toho, když ukládá řetězec, nahrazuje ho nulovým znakem. Viz obrázek 4.5.

Obrázek 4.5 getline() čte a nahrazuje znak nového řádku

Zkusme jiný přístup. Třída `istream` má jinou členskou funkci, jež se nazývá `get()`, která přichází v několika variantách. Jedna varianta pracuje téměř podobně jako `getline()`. Přijímá stejné parametry, interpretuje je stejným způsobem a čte do konce řádku. Ale spíše než by přečetla a vyřadila znak nového řádku, zanechává tento znak ve vstupní frontě. Předpokládejme, že jsme použili dvě volání `get()` v řadě:

```
cin.get(name, ArSize);
cin.get(dessert, Arsize);   // problém
```

Protože první volání zanechává znak nového řádku ve vstupní frontě, tento znak je prvním znakem, který vidí druhé volání. Tedy `get()` dospívá k závěru, že dosáhla konce řádku aniž by nalezla něco ke čtení. Bez pomoci nemůže `get()` správně tímto novým řádkem projít.

Naštěstí existuje pomoc ve formě variantyget(). Volání `cin.get()` (žádné parametry) čte samotný následující znak, dokonce i když to je znak nového řádku, takže jej můžete použít pro oddělení nového řádku a přípravu na další řádek vstupu. To jest, tento postup pracuje:

```
cin.get(name, ArSize);          // čte první řádek
cin.get();                      // čte znak nového řádku
cin.get(dessert, Arsize);       // čte druhý řádek
```

Jiným způsobem použití `get()` je *zřetězení* nebo spojení dvou členských funkcí třídy následovně:

```
cin.get(name, ArSize).get();    // Zřetězení členských funkcí
```

Co to umožňuje je to, že `cin.get(name, ArSize)` vrací objekt cin, který se použije jako objekt, který vyvolá funkci `get()`. Podobně čte příkaz

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

dva za sebou jdoucí vstupní řádky do polí `name1` a `name2`; je to ekvivalentní vytvoření dvou oddělených volání `cin.getline()`.

Výpis programu 4.5 používá zřetězení. V kapitole 10, „Práce s třídami", se naučíte, jak zahrnout tyto rysy do vašich definic tříd.

Výpis programu 4.5 instr3.cpp

```
// instr3.cpp _ čte více než jedno slovo get() & get()
#include <iostream>
using namespace std;
int main()
{
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";
    cin.get(name, ArSize).get();    // čte řetězec, nový řádek
    cout << "Enter your favorite dessert:\n";
```

```
    cin.get(dessert, ArSize).get();
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Kompatibilita:

Některé starší verze C++ neimplementují variantu `get()`, která nemá žádné parametry. Přesto však implementují jinou variantu `get()`, která přijímá jediný parametr `char`. Použijete-li ji namísto ybez parametrů, musíte nejprve deklarovat proměnnou `char`:

```
char ch;
cin.get(name, ArSize).get(ch);
```

Tento kód můžete použít namísto toho, který naleznete ve výpisu programu 4.5. Kapitoly 5, 6, „Příkazy větvení a Logické operátory" a 16 probírají další varianty `get()`.

Tady je vzorek běhu programu:

```
Enter your name:
Mai Parfait
Enter your favorite dessert:
Chocolate Mousse
I have some delicious Chocolate Mousse for you, Mai Parfait.
```

Jednou věcí, která stojí za povšimnutí je to, jak C++ povoluje násobné verze funkcí za předpokladu, že mají různý seznam parametrů. Použijete-li, řekněme, `cin.get(name, ArSize)`, kompilátor si všimne, že jste použili tvar, který vkládá řetězec do pole a vyvolává vhodnou členskou funkci. Když místo toho použijete `cin.get()`, kompilátor si uvědomí, že chcete tvar, který čte jeden znak. Kapitola 8 bude tento rys, který se jmenuje přetížení, zkoumat.

Proč vůbec požíváme `get()` místo `getline()`? Za prvé, starší implementace možná `getline()` nemají. Za druhé vás nutí být trochu pečlivější. Předpokládejme například, že jste používali funkci `get()` na čtení řádku do pole. Jak můžete uhodnout, jestli přečetla celý řádek spíše, než že se zastavila, protože se pole naplnilo? Podívejte se na následující vstupní znak. Je-li to znak nového řádku, pak se přečetl celý řádek. Jestliže to nebyl znak nového řádku, pak jsou na tomto řádku ještě další vstupní data. Kapitola 16 tuto techniku zkoumá. Krátce, `getline()` je na použití trochu jednodušší, ale `get()` zjednodušuje ošetření chyb.

Prázdné řádky a další problémy

Co se stane, když `getline()` nebo `get()` přečtou prázdný řádek? Původní postup byl, že následující příkaz zachytil, kde poslední `getline()` nebo `get()` přestaly. Avšak současný postup po přečtení prázdného řádku pomocí `get()` (ale ne `getline()`) je nastavení něčeho, čemu se říká *chybový bit*. Důsledky této činnosti jsou, že je další vstup zablokován, ale můžete jej obnovit pomocí následujícího příkazu:

```
cin.clear();
```

Jiným potenciálním problémem je to, že by vstupní řetězec mohl být delší než alokovaný prostor. Je-li vstupní řádek delší než počet stanovených znaků, jak `getline()` tak `get()` zanechávají zbylé znaky ve vstupní frontě.

Kapitoly 5, 6 a 16 dále zkoumají, jak se mají využít tyto vlastnosti.

Míchání řetězcového a numerického vstupu

Míchání numerického vstupu spolu s řádkově orientovaným vstupem řetězců může zapříčinit problémy. Uvažujme jednoduchý program na výpisu programu 4.6.

Výpis programu 4.6 numstr.cpp

```
// numstr.cpp - následující vstup čísla pomocí řádkového vstupu
#include <iostream>
using namespace std;
int main()
{
    cout << "What year was your house built?\n";
    int year;
    cin >> year;
    (cin >> year).get();
    cout << "What is its street address?\n";
    char address[80];
    cin.getline(address, 80);
    cout << "Year built: " << year << "\n";
    cout << "Address: " << address << "\n";
    return 0;
}
```

Vykonání tohoto programu by mělo vypadat skoro jako toto:

```
What year was your house built?
1966
What is its street address?
Year built: 1966
Address:
```

Nikdy nedostanete příležitost zavést adresu. Problém spočívá v tom, že když `cin` přečte rok, tak zanechá znak nového řádku generovaný klávesou `ENTER` ve vstupní frontě. Potom `cin.getline()` přečte nový řádek jako prázdný a přiřadí prázdný řetězec do pole `address`. Oprava spočívá v přečtení a odhození znaku nového řádku před čtením adresy. Může to být provedeno několika způsoby, včetně použití `get()` bez žádného parametru nebo s parametrem `char`, jak se popisuje v předchozím příkladu. Volání můžete provést odděleně:

```
cin >> year;
cin.get(); // nebo cin.get(ch);
```

Nebo můžete volání spojit, když vezmete v úvahu skutečnost, že výraz `cin >> year` navrací objekt `cin`:

```
(cin >> year).get(); // nebo (cin >> year).get(ch);
```

Uděláte-li jednu z těchto změn do výpisu programu 4.6, tak to bude pracovat pořádně:

```
What year was your house built?
1966
What is its street address?
43821 Unsigned Short Street
Year built: 1966
Address: 43821 Unsigned Short Street
```

Programy C++ často používají na zacházení s řetězci ukazatele místo polí. Tuto stránku řetězců proberemte, až se něco o ukazatelích naučíme. Mezitím se podívejme na další odvozený typ struktury.

Úvod do struktur

Předpokládejme, že chcete ukládat informaci o hráči košíkové. Možná budete chtít uložit jeho nebo její jméno, plat, výšku, váhu, průměrné skórování, procento volných hodů, nahrávky a tak dále. Určitě by se vám líbil jistý druh datové formy, která by mohla všechnu tuto informaci obsahovat v jediné jednotce. Pole to neudělá. Ačkoli pole může obsahovat několik položek, každá položka musí být stejného typu. To jest, jediné pole může obsahovat 20 celých čísel a jiné 10 čísel v pohyblivé řádové čárce, ale jediné pole nemůže v některých prvcích obsahovat celá čísla a v jiných čísla v pohyblivé řádové čárce.

Odpovědí na vaše přání (to o uložení informace o hráči košíkové) je *struktura* v C++. Struktura je všestrannější datovou formou než pole, protože jediná struktura může obsahovat položky více než jednoho datového typu. Umožňuje vám unifikovat vaši datovou reprezentaci uložením všech informací souvisejících s košíkovou do jediné strukturní proměnné. Chcete-li mít přehled o celém týmu, můžete použít pole struktur. Typ struktura je také prostředkem vedoucím do lahůdky C++ v OOP, třídy. Když se nyní budeme trochu učit o strukturách, tak nás to přivede blíže k OOP, srdci C++.

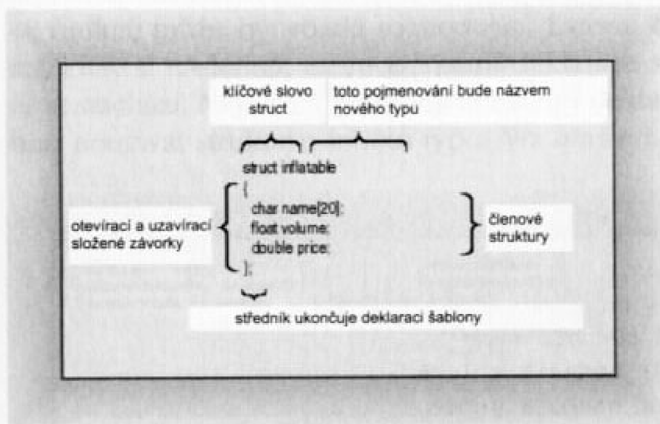
Struktura je uživatelsky definovaným typem s deklarací struktury, která slouží k definování vlastností datových typů. Jakmile definujete typ, můžete vytvořit proměnnou tohoto typu. Tedy vytvoření struktury je procesem ze dvou částí. Za prvé definujete popis struktury. Popisuje a označuje různé typy dat, které mohou být do struktury uloženy. Potom vytvoříte strukturní proměnné nebo obecněji, datové objekty struktury, které odpovídají záměru popisu.

Například předpokládejme, Bloataire, Inc., chce vytvořit typ, který by popisoval součásti jejich výrobní linky pro návrh nafukovacích věcí. Zvláště by měl typ obsahovat jméno položky, její objem v kubických stopách a prodejní cenu. Tady je popis struktury, která odpovídá těmto potřebám:

```
struct inflatable // popis struktury
{
        char name[20]; // člen typu pole
```

```
        float volume; // člen typu float
        double price; // člen typu double
    };
```

Klíčové slovo `struct` indikuje, že kód definuje návrh struktury. Identifikátor `inflatable` je jméno, neboli jmenovka tohoto tvaru; vytváří z `inflatable` jméno pro nový typ. Takto nyní můžete vytvořit proměnné typu `inflatable` stejně, jako byste vytvářeli proměnné typu `char` nebo `int`. Dále, mezi složenými závorkami se zúčastňuje seznam datových typů, které mají být obsaženy ve struktuře. Můžete zde použít libovolný z typů C++, včetně polí nebo jiných struktur. Příklad používá pole `char`, vhodné pro uložení řetězce, `float` a `double`. Každá jednotlivá položka seznamu se nazývá členem struktury, takže struktura `inflatable` má tři členy. Viz obrázek 4.6.



Obrázek 4.6 Části popisu struktury

Poté, co máte šablonu, můžete vytvořit proměnnou tohoto typu:

```
inflatable hat;             // hat je strukturní proměnná typu inflatable
inflatable woopie_cushion;  // typ proměnné inflatable
inflatable mainframe;       // typ proměnné inflatable
```

Znáte-li struktury v C, všimli jste si (pravděpodobně s potěšením), že C++ dovoluje odhodit klíčové slovo `struct`, když deklarujete strukturní proměnnou:

```
struct inflatable goose;   // klíčové slovo struct se vyžaduje v C
inflatable vincent;        // klíčové slovo struct se v C++ nevyžaduje
```

V C++ se jmenovky struktury používá pouze jako základní typové jméno. Tato změna zdůrazňuje, že deklarace struktury definuje nový typ. Také odstraňuje ze seznamu opomenutí `struct`, chyby přivozující neštěstí.

Je-li dáno, že `hat` je typu `inflatable`, použijte na přístup k jednotlivým členům operátor příslušnosti (.). Například `hat.volume` se odkazuje na člen `volume` struktury a `hat.price` se odvolává na člen `price`. Podobně `vincent.price`, je `price` členem proměnné `vincent`. Zkrátka, jména členů vám dovolují přistupovat ke členům struktury skoro jak vám to dovolují indexy k prvkům pole. Protože člen `price` je deklarován jako typ double, `hat.price` a `vincent.price` jsou oba ekvivalentní proměnným typu double a mohou se používat libovolným způsobem, jako se může používat obyčejná proměnná typu double. `hat` je struktura, ale `hat.price` je `double`. Mimochodem, metoda, která se používá na pří-

stupu ke členským funkcím třídy, jako například cin.getline(), má svůj původ v metodě použité v přístupu ke členským proměnným struktury, jako například `vincent.price`.

Výpis programu tyto vlastnosti struktury ilustruje. Také ukazuje, jak se má struktura inicializovat.

Výpis programu 4.7 structur.cpp

```
// structur.cpp - jednoduchá struktura
#include <iostream>
using namespace std;
struct inflatable   // šablona struktury
{
    char name[20];
    float volume;
    double price;
};

int main()
{
    inflatable guest =
    {
        "Glorious Gloria",  // hodnota name
        1.88,               // hodnota volume
        29.99               // hodnota price
    };  // guest je strukturní proměnnou typu inflatable
    // inicializuje se uvedenými hodnotami
    inflatable pal =
    {
        "Audacious Arthur",
        3.12,
        32.99
    };  // pal je druhou proměnnou typu inflatable
    // Poznámka: některé implementace vyžadují použití
    // static inflatable guest =

    cout << "Expand your guest list with " << guest.name;
    cout << " and " << pal.name << "!\n";
    // pal.name is the name member of the pal variable
    cout << "You can have both for $";
    cout << (guest.price + pal.price) << "!\n";
    return 0;
}
```

Kompatibilita:

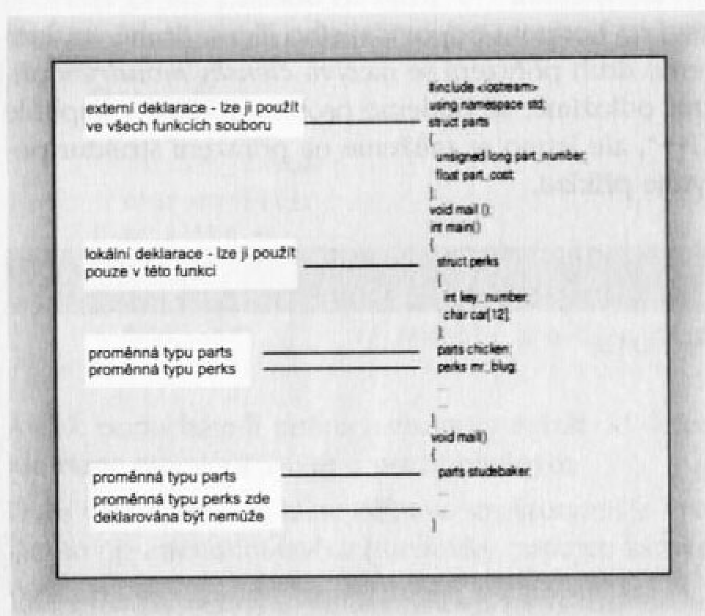
Stejně tak, jako některé implementace nejsou způsobilé inicializovat obyčejné pole definované ve funkci, také nejsou způsobilé inicializovat ve funkci obyčejnou strukturu. Znovu je řešením použití klíčového slova `static` v deklaraci.

Zde je výstup:

```
Expand your guest list with Glorious Gloria and Audacious Arthur!
You can have both for $62.98!
```

Poznámky k programu

Důležitou věcí je, kam umístit deklaraci struktury. Pro `structur.cpp` existují dvě volby. Mohli byste umístit deklaraci do nitra funkce `main()`, hned za otevírací složenou závorku. Druhou volbou, která je zde provedena, je umístit ji mimo a před `main()`. Vyskytuje-li se deklarace vně funkce, nazývá se *externí deklarací.* Pro tento program neexistuje žádný praktický rozdíl mezi oběmi volbami. Ale pro programy, které sestávají ze dvou nebo více funkcí, může být rozdíl rozhodující. Externí deklarace se může použít všemi funkcemi, které jí následují, zatímco interní deklarace se může použít pouze ve funkci, ve které se nachází. Nejčastěji potřebujete externí deklaraci struktury, takže všechny funkce mohou používat strukturu tohoto typu. Viz obrázek 4.7.



Obrázek 4.7 Lokální a externí deklarace struktury

Proměnné s externími proměnnými sdílenými všemi funkcemi mohou být také deklarovány interně nebo externě. (Kapitola 8 se na toto téma dívá blíže.) Praxe v C++ odrazuje od použití externích proměnných, ale podporuje použití deklarací externích funkcí. Často má také smysl externě deklarovat symbolické konstanty.

Všimněte si inicializační procedury:

```
inflatable guest =
{
        "Glorious Gloria",        // hodnota jména
         1.88,                    // hodnota objemu
        29.99                     // hodnota ceny
};
```

Stejně tak jako u polí, používáte seznam hodnot oddělený čárkami uzavřený do páru složených závorek. Program umisťuje jednu hodnotu na řádku, ale můžete je umístit všechny na tom stejném řádku. Pouze si pamatujte, abyste položky oddělili čárkou:

```
inflatable duck = {"Daphne", 0.12, 9.98};
```

Každý člen struktury se považuje za proměnnou tohoto typu. Tedy `pal.price` je proměnná double a `pal.name` pole `char`. Když program používá `cout` na zobrazení `pal.name`, zobrazuje člen jako řetězec. Mimochodem, protože je `pal.name` znakové pole, můžete pro přístup k jednotlivým znakům pole použít indexy. Například `pal.name[0]` je znak A. Avšak `pal[0]` nemá význam, protože pal je strukturou, nikoli polem.

Další vlastnosti struktury

C++ vytváří uživatelsky definované typy tak jednoduché jako jsou třeba vestavěné typy. Například můžete předat struktury funkci jako parametry a můžete mít funkci, která používá strukturu jako návratovou hodnotu. Také, abyste přiřadili jednu strukturu stejného typu do druhé, můžete použít přiřazovací operátor (=). Když to uděláte, tak to způsobí, že se každý člen jedné struktury nastaví na hodnotu odpovídajícího členu druhé struktury, dokonce i když je členem pole. Tento druh přiřazení se nazývá *člensky moudrým* přiřazením. Předávání a navracení struktur odložíme, až budeme probírat funkce v kapitole 7, „Funkce – programové moduly v C++", ale letmo se můžeme na přiřazení struktur podívat nyní. Výpis programu 4.8 poskytuje příklad.

Výpis programu 4.8 assgn_st.cpp

```
// assgn_st.cpp - přiřazení struktur
#include <iostream>
using namespace std;
struct inflatable
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    inflatable bouquet =
    {
        "sunflowers",
        0.20,
        12.49
    };
    inflatable choice;
    cout << "bouquet: " << bouquet.name << " for $";
    cout << bouquet.price << "\n";

    choice = bouquet;  // přiřadí jednu strukturu druhé
    cout << "choice: " << choice.name << " for $";
```

```
    cout << choice.price << "\n";
    return 0;
}
```

Zde je výstup:

```
bouquet: sunflowers for $12.49
choice: sunflowers for $12.49
```

Jak můžete vidět, *člensky moudré* přiřazení je v činnosti, protože členům struktury `choice` se přiřazují stejné hodnoty, které jsou uloženy ve struktuře `bouquet`.

Definici tvaru struktury a vytvoření strukturní proměnné můžete spojit. Abyste tak učinili, následujte uzavírací složené závorky jménem nebo jmény proměnné:

```
struct perks
{
    int key_number;
    char car[12];
} mr_smith, ms_jones;    // dvě proměnné perks
```

Dokonce můžete proměnné, které jste vytvořili tímto způsobem, inicializovat:

```
struct perks
{
    int key_number;
    char car[12];
} mr_glitz =
{
    7,                // hodnota pro člen mr_glitz.key_number
    "Packard"         // hodnota pro člen mr_glitz.car
};
```

Avšak ponecháte-li definici struktury zvlášť od deklarací proměnných, program se obvykle stane snáze čitelným a pochopitelným.

Další věcí, kterou můžete dělat se strukturami, je vytvořit strukturu beze jména typu. Uděláte to opomenutím jména jmenovky, zatímco zároveň definujete tvar struktury a proměnnou:

```
struct          // žádná jmenovka
{
        int x;  // 2 členy
        int y;
} position;          // strukturní proměnná
```

Vytváří to jedinou strukturní proměnnou, která se nazývá `position`. K jejím členům můžete přistupovat pomocí operátoru příslušnosti, jako například v `position.x`, ale žádné obecné jméno tohoto typu neexistuje. Současně nemůžete vytvořit další proměnnou stejného typu. Kniha nebude používat tohoto omezeného tvaru struktury.

Nehledě na skutečnost, že program může používat jmenovky struktury jako jméno typu, struktury v C mají všechny rysy, které jsme dosud probrali o strukturách v C++. Avšak struktury v C++ jdou dále. Například struktury v C++, na rozdíl od struktur v C, mohou mít spolu s členskými proměnnými členské funkce. Ale tyto pokročilejší rysy jsou

spíše používány s třídami, než se strukturami, takže o nich pojednáme, až budeme probírat třídy.

Pole struktur

Struktura `inflatable` obsahuje pole (pole `name`). Je také možné vytvořit pole, jehož prvky jsou struktury. Postup je přesně ten samý, jako pro vytvoření polí základních typů. Například, abyste vytvořili pole o 100 strukturách `inflatable`, proveďte následující:

```
inflatable gifts[100]; // pole o 100 strukturách inflatable
```

Toto vytváří pole `gifts` typu `inflatable`. Z tohoto důvodu je každý prvek pole, jako například `gifts[0]` nebo `gifts[99]`, objektem `inflatable` a může se použít s operátorem příslušnosti:

```
cin >> gifts[0].volume; // použijte první člen struktury volume
cout << gifts[99].price << endl; // zobrazte poslední člen struktury price
```

Pamatujte, že sama `gifts` je polem, ne strukturou, takže konstrukce jako je `gifts.price` jsou nepřípustné.

Na inicializování pole struktur spojte pravidlo inicializace polí (seznam hodnot uzavřený do složených závorek, kde je každý prvek oddělený čárkou) spolu s pravidlem pro struktury (seznam hodnot uzavřený do složených závorek, kde je každý člen oddělený čárkou). Protože každý prvek pole je strukturou, jeho hodnota se reprezentuje inicializací struktury. Vložíte tedy se složenými závorkami seznam hodnot oddělených čárkami, každá z nichž, sama uzavřena do složených závorek, je seznamem hodnot oddělených čárkou:

```
inflatable guests[2] = // inicializace pole struktur
{
        {"Bambi", 0.5, 21.99},          // první struktura v poli
        {"Godzilla", 2000, 565.99}      // další struktura v poli
};
```

Jako obvykle to můžete formátovat, jak se vám zlíbí. Obě inicializace mohou být na stejném řádku, nebo každá inicializace samostatného členu struktury může například dostat svůj vlastní řádek.

Bitová pole

C++, podobně jako C, vám umožňují specifikovat strukturní členy, které zaujímají určitý počet bitů. To může být šikovné na vytvoření datové struktury, která například odpovídá registru nějakého hardwarového zařízení. Typ pole by měl být celočíselným nebo výčtovým typem (výčtové typy se budou v kapitole probírat později) následovaným dvojtečkou s číslem, které určuje skutečný počet bitů, které se mají použít. Na poskytnutí mezer můžete použít nepojmenovaná pole. Každý člen má označení *bitové pole*. Zde je příklad:

```
struct torgle_register
{
        int SN : 4;             // 4 bity pro hodnotu SN
        int : 4;                // 4 nevyužité bity
```

```
        bool goodIn : 1;            // platný vstup (1 bit)
        bool goodTorgle : 1;        // úspěšný torgling
};
```

Pro přístup k bitovým polím použijete standardního značení struktury:

```
torgle_register tr;
...
if (tr.goodIn)
```

Bitová pole se převážně používají na úrovni strojově orientovaného programování. Používání celočíselného typu a bitových operátorů z Dodatku E, „Další operátory", poskytuje často alternativní přístup.

Uniony

Union je datovým formátem, který může používat různé datové typy, ale pouze jeden typ v daném okamžiku. To jest, zatímco struktura může obsahovat řekněme `int` *a* `long` *a* `double`, `union` může obsahovat *buď* `int` *nebo* `long` *nebo* `double`. Syntaxe je podobná jako u struktury, ale význam je různý. Například uvažujme následující deklaraci:

```
union one4all
{
        int int_val;
        long long_val;
        double double_val;
};
```

Proměnnou one4all můžete využít na uchování `int`, `long` nebo `double` pouze tak dlouho, dokud je potřebujete:

```
one4all pail;
pail.int_val = 15;                // uloží int
cout <lt; pail.int_val;
pail.double_val = 1.38;    // uloží double, hodnota int se ztratí
cout << pail.double_val;
```

Tedy, `pail` může sloužit jako proměnná `int` při jedné příležitosti a jako proměnná `double` při jiné. Jméno členu určuje kapacitu, při které proměnná funguje. Protože v jednom okamžiku proměnná obsahuje pouze jednu hodnotu, musí mít dostatek prostoru na úschovu svého největšího členu. Z toho důvodu je velikost unionu velikostí jejího největšího členu.

Jedním upotřebením unionu je úspora prostoru, když datová položka může používat dva nebo více formátů, ale nikdy ne současně. Například předpokládejme, že spravujete smíšený inventář blíže neurčených věcí, některé z nich mají celočíselnou identifikaci ID a jiné řetězcovou. Potom byste mohli udělat následující:

```
struct widget
{
        char brand[20];
        union id // formát závisí na typu věcičky
```

```
        {
          long id_num; // typ věcičky 1
          char id_char[20]; // jiné věcičky
        };
        int type;
};
...
widget prize;
...
if (prize.type == 1)
cin >> prize.id.id_num; // jméno členu použijte na určení režimu
else
cin >> prize.id.id_char;
```

Union *neznámého původu* nemá žádné jméno; v podstatě se jeho členy stávají proměnnými, které sdílejí stejnou adresu. Přirozeně, běžný v jednom okamžiku může být pouze jeden člen:

```
struct widget
{
        char brand[20];
        union // formát závisí na typu věcičky
        {
          long id_num; // typ věcičky 1
          char id_char[20]; // jiné věcičky
        };
        int type;
};
...
widget prize;
...
if (prize.type == 1)
cin >> prize.id_num; // jméno členu použijte na určení režimu
else
cin >> prize.id_char;
```

Protože je union anonymní, `id_num` a `id_char` se považují za dva členy price, které sdílejí stejnou adresu. Potřeba prostředního identifikátoru id se odstranila. Na programu záleží, aby sledoval, která z voleb je aktivní.

Výčtové typy

Vybavení `enum` v C++ poskytuje alternativní prostředek ke `const` na vytváření symbolických konstant. Dovoluje také definovat nové typy, ale zcela omezeným způsobem. Syntaxe použití enum se podobá syntaxi struktury. Například uvažujme následující příkaz:

```
enum spectrum {red, orange, yellow, green, blue, violet, indigo,
    ultraviolet};
```

Tento příkaz dělá dvě věci:

Ze `spectrum` vytváří jméno nového typu; `spectrum` se vztahuje k *enumeration* (výčtu), téměř jako proměnná `struct` se nazývá *structure* (strukturou).

Stanoví `red, orange, yelow` a tak dále symbolickými konstantami pro celá čísla 0–7. Tato čísla se nazývají enumerátory.

Standardně se enumerátorům přiřazují celočíselné hodnoty počínaje nulou pro první enumerátor, 1 pro další atd. Standard můžete přepsat přiřazením celočíselných hodnot. Jak se to dělá, vám ukážeme později.

Jméno výčtu můžete použít na deklaraci proměnné tohoto typu:

```
spectrum band;
```

Výčtová proměnná má některé speciální vlastnosti, které nyní prozkoumáme.

Jediné platné hodnoty, které můžete přiřadit výčtové proměnné bez přetypování, jsou výčtové proměnné použité v definici typu. Proto máme následující:

```
band = blue;             // platné, blue je enumerátorem
band = 2000;             // neplatné, 2000 není enumerátorem
```

Tedy proměnná `spectrum` je omezena pouze na osm možných hodnot. Některé kompilátory vydávají chybové hlášení, když se pokoušíte přiřadit neplatnou hodnotu, jiné vydávají varování. Pro maximální přenositelnost byste měli pohlížet na přiřazení hodnoty, která není `enum`, proměnné typu `enum`, jako na chybu. Pro výčtové typy je definován pouze přiřazovací operátor. Především nejsou definovány aritmetické operace:

```
band = orange;           // platné
++band;                  // neplatné
band = orange + red;     // neplatné
...
```

Avšak některé implementace toto omezení neakceptují. To může způsobit porušení mezí typu. Například, má-li `band` hodnotu `ultraviolet`, neboli 7, potom `++band`, ačkoli platné, inkrementuje `band` na 8, což není platnou hodnotou pro typ `spectrum`. Znovu, kvůli maximální přenositelnosti, byste se měli přizpůsobit přísnějším omezením.

Enumerátory jsou výčtového typu a mohou výt povýšeny na typ int, ale typy int nejsou automaticky změnitelné na výčtový typ:

```
int color = blue;        // platné, typ spectrum je přeměnitelný na int
band = 3;                // neplatné, int není přeměnitelné na spectrum
color = 3 + red;         // platné, red je přeměnitelný na int
...
```

Všimněte si, že ačkoli dokonce 3 odpovídá enumerátoru `green`, přiřazení 3 do `band` je chybou typu. Znovu, některé implementace toto omezení nezesilují. Ve výrazu `3 + red`, není sčítání pro enumerátory definováno. Avšak `red` se přeměňuje na typ `int` a výsledkem je typ `int`. Ačkoli sčítání není pro enumerátory definováno, enumerátory můžete používat v aritmetických výrazech.

Hodnotu `int` můžete přiřadit `enum` za předpokladu, že je hodnota platná a že použijete explicitní přetypování:

```
band = spectrum(3);      // přetypování 3 na typ spectrum
```

Jak můžete vidět, pravidla, která řídí výčty jsou docela omezující. V praxi se výčty používají častěji jako způsob definování symbolických konstant, než jako prostředek definování nového typu. Například můžete výčty použít na definování symbolických konstant pro příkaz `switch`. (Viz kapitola 6 kvůli příkladu.) Plánujete-li použít pouze konstanty a nevytvářet proměnné výčtového typu, můžete opustit jméno výčtového typu:

```
enum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Nastavení hodnot enumerátorů

Hodnoty enumerátorů můžete nastavit explicitně použitím přiřazovacího operátoru:

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
```

Přiřazované hodnoty musí být celá čísla. Také můžete definovat pouze některé z enumerátorů explicitně:

```
enum bigstep{first, second = 100, third};
```

V tomto případě je `first` standardně 0. Následující neinicializované enumerátory jsou větší o jedničku než jejich předchůdci. Takže `third` by měl mít hodnotu 101.

Konečně můžete vytvořit více než jeden enumerátor se stejnou hodnotou:

```
enum {zero, null = 0, one, numero_uno = 1};
```

Zde jak `zero` tak `null` jsou 0, a oba `one` a `numero_uno` jsou 1. V dřívějších verzích C++, jste mohli přiřadit enumerátorům pouze hodnoty `int` (nebo hodnoty, které se dají na `int` povýšit), ale toto omezení bylo odstraněno, takže můžete použít hodnoty typu `long`.

Rozsahy hodnot pro výčty

Původně jedinými platnými hodnotami pro výčty byly hodnoty, které byly vyjmenované v deklaraci. Avšak C++ nyní podporuje čistý způsob, pomocí kterého můžete platně přiřadit hodnoty prostřednictvím přetypování na výčtovou proměnnou. Každý výčet má *rozsah* a vy můžete přiřadit libovolnou celočíselnou hodnotu v tomto rozsahu, dokonce i když není výčtovou hodnotou, využitím přetypování na výčtovou hodnotu. Například předpokládejme, že `myflag` je proměnnou typu `bits` (jak bylo dříve definováno). Potom je následující platné:

```
myflag = bits(6); // platné, protože 6 je v rozsahu bits
```

Zde 6 není jedním z enumerátorů, ale leží v rozsahu definice výčtu.

Rozsah se definuje následovně. Za prvé, abyste zjistili horní mez, vezměte největší hodnotu enumerátoru. Zjistěte nejmenší mocninu dvou, která je větší než tato největší hodnota, odečtěte jednotku a to je horním koncem rozsahu. Například největší hodnota `bigstep`, jak se dříve definovalo, je 101. Nejmenší mocnina dvou větší než tato hodnota je 128, takže horním koncem rozsahu je 127. Dále nalezněte spodní hranici, zjistěte nejmenší hodnotu enumerátoru. Je-li nula nebo větší, spodní hranicí rozsahu je nula. Je-li nejmenší enumerátor záporný, použijte stejného způsobu, jako pro nalezení horní hranice, ale dodejte záporné znaménko. Například je-li nejmenší enumerátor –6, následující mocnina dvou (násobte záporným znaménkem) je –8, a spodní hranicí je –7.

Myšlenkou je to, že kompilátor může zvolit dostatek prostoru na uchování výčtu. Může použít jeden nebo méně bajtů na výčty s malým rozsahem a čtyři bajty pro výčty s hodnotami typu long.

Ukazatele a volná paměť

Začátek kapitoly 3, „Práce s daty", se zmínil o třech základních vlastnostech, které musí počítačový program sledovat, když ukládá data. Abyste ušetřili knihu od odření a slz z vašeho listování zpět k této kapitole, tak jsou zde tyto vlastnosti znovu:

- Kde je informace uložena
- Jaké jsou tam hodnoty
- Jaký druh informace je uložen

Na dosažení těchto konců jste použili jednu strategii: Definování jednoduché proměnné. Deklarační příkaz poskytuje pro hodnotu typ a symbolické jméno. Také má za příčinu, že program alokuje pro hodnotu paměť a místo vnitřně sleduje.

Podívejme se nyní na druhou strategii, tu, která se stane zvláště důležitou v rozvoji tříd v C++. Tato strategie je založena na ukazatelích, což jsou proměnné, které spíše než hodnotu samu, ukládají její adresu. Ale dříve než pojednáme o ukazatelích, podívejme se jak zjistit adresu obyčejných proměnných. Na proměnnou pouze aplikujte adresový operátor, který je reprezentován `&`, abyste získali její umístění; například, je-li `home` proměnná, `&home` je její adresa. Výpis programu 4.9 tento operátor demonstruje.

Výpis programu 4.9 address.cpp

```
// address.cpp _ použití operátoru & na zjištění adresy
#include <iostream>
using namespace std;
int main()
{
    int donuts = 6;
    double cups = 4.5;

    cout << "donuts value = " << donuts;
    cout << " and donuts address = " << &donuts << "\n";
    // Poznámka: možná musíte použít unsigned (&donuts)
    //    a unsigned (&cups)
    cout << "cups value = " << cups;
    cout << " and cups address = " << &cups << "\n";
return 0;
}
```

Kompatibilita:

Cout je chytrý objekt, ale některé verze jsou chytřejší než jiné. Proto některé implementace mohou selhat při rozpoznání typu ukazatele. V tomto případě musíte adresu přetypovat, aby se stala rozpoznatelným typem, jako je například `unsigned int`. Vhodné přetypování závisí na paměťovém modelu. Standardní paměťový model Dos používá 2-bajtovou adresu, proto je `unsigned int` správným přetypováním. Některé paměťové modely Dos však používají 4-bajtovou adresu, která vyžaduje přetypování na `unsigned long`.

Zde je výstup z jednoho systému:

```
donuts value = 6 and donuts address = 0x8566fff4
cups value = 4.5 and cups address = 0x8566ffec
```

Když `cout` zobrazuje adresy, používá hexadecimální značení, protože je to obvyklé značení pro popis adresy. Naše implementace ukládá `cups` do nižší pozice paměti než `donuts`. Rozdíl mezi oběmi adresami je 0x856fff4-0x8566ffec, neboli 8. To dává smysl, protože `cups` je typu `double`, který používá osm bajtů. V případě, že místo zajímá, tak tyto určité reprezentace adres odrážejí metodu na PC, která popisuje adresy pomocí hodnoty segmentu a ofsetu. Hodnota segmentu, v tomto případě `8566`, identifikuje blok paměti, který se používá pro uložení dat; to je skutečná adresa dělená 16. Ofsety, v tomto případě `ffec` a `fff4`, reprezentují pozici paměti relativně k začátku segmentu. Jestliže jsou všechna data v jednom segmentu, programy pro PC mohou používat pouze 2-bajtové ukazatele obsahující pouze ofset. Nebo mohou použít 4-bajtové ukazatele, kde první dva bajty obsahují hodnotu segmentu a druhé dva ofset.

Použijeme-li obyčejné proměnné, potom nakládá s hodnotou, jako s pojmenovanou veličinou a s lokací, jako s odvozenou veličinou. Nyní se podívejme na strategii ukazatele, která je základní pro metodiku programování v C++ při správě paměti. (Viz následující poznámku.)

Ukazatele a filozofie C++

Objektově orientované programování se liší od tradičního procedurálního programování v důrazu OOP na rozhodování během běhu programu namísto v době kompilace. Běh programu znamená, když se program provádí a *doba kompilace* znamená, když kompilátor sestavuje program dohromady. Rozhodnutí v době běhu je jako, kdy jet na prázdniny, výběr pamětihodností, které chceme vidět, závisí na počasí a na náladě v daném okamžiku, zatímco rozhodnutí v době kompilace je více jako dodržování nastaveného rozvrhu nehledě na podmínky.

Rozhodnutí v době běhu poskytuje pružnost v přizpůsobení se aktuálním okolnostem. Například uvažujme alokaci paměti pro pole. Tradičním způsobem je deklarovat pole. Abychom deklarovali pole v C++, musíme se svěřit určité velikosti pole. Tedy, velikost pole se nastaví, když se program kompiluje; to je rozhodnutí v době kompilace. Možná si myslíte, že pole o 20 prvcích je dostačující v 80% času, ale že příležitostně bude muset program spravovat 200 prvků. Aby to bylo bezpečné, použijete pole s 200 prvky. To má ve vašem programu za následek plýtvání pamětí po většinu času, kdy se používá. OOP se pokouší vytvořit program mnohem pružnější tím, že ponechává taková rozhodnutí až do doby běhu. Tímto způ-

sobem, poté co program běží, můžete říct, že potřebujete v jednom okamžiku pouze 20 prvků nebo v jiném 205 prvků.

Zkrátka, velikost pole je rozhodnutím v době běhu. Jazyk vám musí na tento přístup umožnit vytvoření pole – nebo ekvivalent, zatímco program běží. Metoda v C++, kterou brzy uvidíte, používá na vyžádání správného množství paměti klíčové slovo new a ukazatele, které sledují, kde se nově alokovaná paměť nachází.

Nová metoda správy uložených dat mění věci ohledně nakládání s lokací jako je jméno veličiny a hodnoty jako je odvozená veličina. Zvláštní typ proměnné – ukazatel obsahuje adresu hodnoty. Tedy jméno ukazatele reprezentuje umístění. Použití operátoru *, který se nazývá *nepřímou hodnotou* nebo *dereferenčním operátorem*, poskytuje hodnotu v lokaci. (Ano, je to stejný symbol * používaný pro násobení; C++ používá kontext na určení, zda míníte násobení nebo dereferencování.) Předpokládejme například, že `manly` je ukazatel. Potom `manly` reprezentuje adresu a `*manly` hodnotu na této adrese. Spojení `*manly` se stává ekvivalentem k obyčejnému typu proměnné int. Výpis programu 4.10 tyto body ukazuje. Také ukazuje, jak deklarovat ukazatel.

Výpis programu 4.10 pointer.cpp

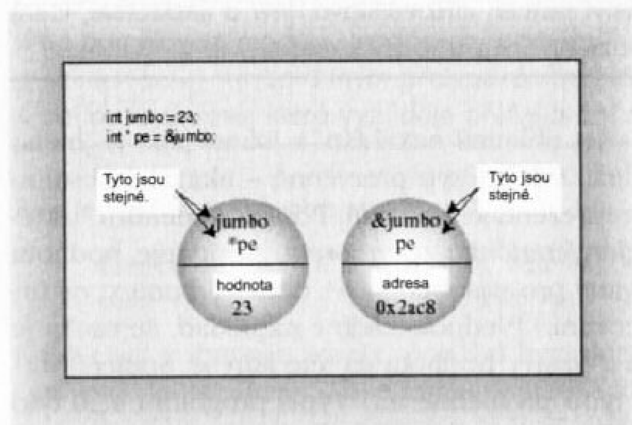
```
// pointer.cpp _ naše první ukazatelová proměnná
#include <iostream>
using namespace std;
int main()
{
    int updates = 6;        // deklaruje proměnnou
    int * p_updates;        // deklaruje ukazatel na int
    p_updates = &updates;   // přiřazuje adresu int ukazateli
    // vyjadřuje hodnoty dvěma způsoby
    cout << "Values: updates = " << updates;
    cout << ", *p_updates = " << *p_updates << "\n";
    // vyjadřuje hodnoty dvěma způsoby
    cout << "Addresses: &updates = " << &updates;
    cout << ", p_updates = " << p_updates << "\n";
    // používá ukazatel na změnu hodnoty
    *p_updates = *p_updates + 1;
    cout << "Now updates = " << updates << "\n";
    return 0;
}
```

Zde je výstup:

```
Values: updates = 6, *p_updates = 6
Addresses: &updates = 0x85b0fff4, p_updates = 0x85b0fff4
Now updates = 7
```

Jak můžete vidět, `int` proměnná updates a ukazatelová proměnná `p_updates`, jsou pouze dvě strany jedné mince. Proměnná updates reprezentuje primárně hodnotu a používá operátor & na získání adresy, zatímco proměnná `p_updates` reprezentuje primárně adresu a používá operátor * na získání hodnoty. Viz obrázek 4.8. Protože `p_updates` ukazuje na

updates, `*p_updates` a `updates` jsou úplně ekvivalentní. Můžete použít `*p_updates` přesně tak, jako byste použili typ proměnné int. Jak program ukazuje, dokonce můžete přiřadit do `*p_updates` hodnotu. Uděláte-li to, změní se hodnota proměnné na kterou se ukazuje, `update`.



Obrázek 4.8 Dvě strany jedné mince

Deklarování a inicializace ukazatelů

Vyšetříme proces deklarování ukazatelů. Počítač musí sledovat typ proměnné, na kterou se ukazatel odvolává. Například adresa `char` vypadá stejně, jako adresa double, ale `char` a `double` používají různé počty bajtů a různé vnitřní formáty pro uložení hodnot. Proto musí deklarace ukazatele specifikovat, jaký je typ dat, na který ukazatel ukazuje.

Například poslední příklad má tuto deklaraci:

```
int * p_updates;
```

To stanovuje, že spojení `* p_update` je typu `int`. Protože se operátor `*` používá ve vztahu k ukazateli, sama proměnná `p_update` musí být ukazatelem. Říkáme, že `p_update` ukazuje na typ `int`. Říkáme také, že typ pro `p_update` je ukazatelem na `int`, nebo výstižněji, `int *`. Abychom zopakovali: `p_updates` je ukazatel (adresa) a `*p_updates` je `int` a nikoli ukazatel. Viz obrázek 4.9.

Mimochodem, použití mezer kolem operátoru `*` je volitelné. Programátoři v C používali tradičně tento tvar:

```
int *ptr;
```

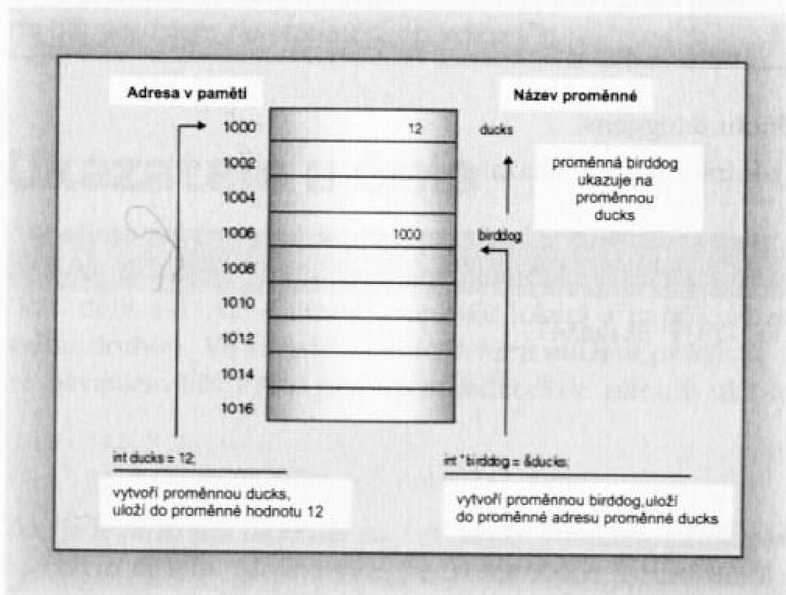
To zdůrazňuje myšlenku, že spojení `*ptr` je hodnotou typu `int`. Mnoho programátorů v C++, na druhé straně, používá tuto formu:

```
int* ptr;
```

To zdůrazňuje myšlenku, že `int*` je typem, ukazatelem na `int`. Kam umístíte mezery, v tom nedělá kompilátor rozdíl. Buďte si vědomi, že deklarace

```
int* p1, p2;
```

vytváří jeden ukazatel (p1) a jednu obyčejnou proměnnou (p2) typu int. Pro každé jméno ukazatelové proměnné potřebujete *.



Obrázek 4.9 Ukazatele ukládají adresy

Pamatujte:

V C++ je spojení int * odvozeným typem, ukazatelem na int.

Na deklaraci ukazatelů ostatních typů použijte stejnou syntaxi:

```
double * tax_ptr;          // tax_ptr ukazuje na typ double
char * str;                // str ukazuje na typ char
```

Protože deklarujete tax_ptr jako ukazatel na double, kompilátor ví, že *tax_ptr je hodnotou typu double. To jest ví, že *tax_ptr reprezentuje číslo uložené ve formátu pohyblivé řádové čárky, které zaujímá (na většině systémů) osm bajtů. Proměnná ukazatel nikdy není jednoduchým ukazatelem. Vždy je to ukazatel na určitý typ. tax_ptr je typem ukazatel na double (neboli typem double *) a str je typem ukazatel na char (neboli char *). Ačkoli jsou oba ukazateli, jsou to ukazatele dvou rozdílných typů. Podobně jako pole, ukazatele jsou odvozeny z jiných typů.

Všimněte si, že zatímco tax_ptr a str ukazují na datové typy různých velikostí, obě samotné proměnné tax_ptr a str jsou typicky stejné velikosti. To jest, adresa char má stejnou velikost jako adresa double, téměř jako 1016 by mohla být adresou ulice obchodního domu, zatímco 1024 by mohla být adresou ulice malé chalupy. Velikost nebo hodnota adresy vám ve skutečnosti nic neříká o velikosti nebo druhu proměnné nebo budovy, kterou na adrese hledáte. Obvykle adresa vyžaduje dva nebo čtyři bajty, to záleží na počítačovém systému. (Některé systémy mohou mít větší adresy a systém může použít různé velikosti adres pro různé typy.)

Na inicializaci ukazatele můžete použít deklarační příkaz. V tomto případě se inicializuje ukazatel, nikoli hodnota, na kterou se ukazuje. To jest příkazy:

```
int higgens = 5;
int * pi = &higgens;
```

nastavují pi a nikoli *pi na hodnotu &higgens.

Výpis programu 4.11 ukazuje, jak inicializovat ukazatel adresou.

Výpis programu 4.11 init_ptr.cpp

```
// init_ptr.cpp - inicializuje ukazatel
#include <iostream>
using namespace std;
int main()
{
    int higgens = 5;
    int * pi = &higgens;
    cout << "Value of higgens = " << higgens
         << "; Address of higgens = " << &higgens << "\n";
    cout << "Value of *pi = " << *pi
         << "; Value of pi = " << pi << "\n";
    return 0;
}
```

Zde je výstup:

```
Value of higgens = 5; Address of higgens = 0068FDF0
Value of *pi = 5; Value of pi = 0068FDF0
```

Můžete vidět, že program inicializuje `pi` adresou `higgens`, nikoli `*pi`.

Na ty, kteří nepozorně používají ukazatele, čeká nebezpečí. Jedním zvláště důležitým bodem je, že když vytváříte v C++ ukazatel, počítač alokuje paměť na úschovu adresy, ale nealokuje paměť na úschovu dat, na která adresa ukazuje. Vytvoření prostoru pro data zahrnuje oddělený krok. Opomenutím tohoto kroku, podobně jako v následující ukázce, je pozvánkou k pohromě:

```
long * fellow;            // vytváří ukazatel na long
*fellow = 223323;         // umístění hodnoty do země nikoho
```

Jistě, `fellow` je ukazatel. Ale kam ukazuje? Programový kód selhal při přiřazení adresy do `fellow`. Tak kam se uložila hodnota 223323? To nemůžeme říci. Protože `fellow` nebyla inicializována. Cokoli, co je hodnotou, program interpretuje jako adresu, na kterou se uloží 223323. Když se přihodí, že `fellow` má hodnotu 1200, potom se program pokusí umístit data na adresu 1200, dokonce i když se stane, že je to adresa uprostřed vašeho programového kódu. Rizikem je, že na cokoli fellow ukazuje, není to místo, kam chcete uložit číslo 223323. Tento druh chyby může vyprodukovat některé z nejzákeřnějších a těžce vystopovatelných opomenutí.

Upozornění:
Zlaté pravidlo ukazatele: VŽDY inicializujte ukazatel jednoznačnou a vhodnou adresou, než na něj použijete dereferenční operátor (*).

Ukazatele a čísla

Ukazatele nejsou celočíselné typy, třebaže počítače spravují adresy jako celá čísla. Pojmově jsou ukazatele jinými typy než celá čísla. Celá čísla jsou čísla, která můžete sčítat, odčítat, dělit atd. Ale ukazatel popisuje lokaci a například nemá smysl násobit dvě lokace jednu druhou. Ve smyslu operací s nimi můžete pracovat, ukazatele a celá čísla se od sebe navzájem liší. Proto nemůžete jednoduše přiřadit ukazateli celé číslo:

```
int * pi;
pi = 0xB8000000;  // nesoulad typů
```

Zde je levá strana ukazatel na `int`, takže jí můžete přiřadit adresu, ale pravá strana je pouze celým číslem. Možná víte, že 0xB8000000 je složenou adresou segmentu a ofsetu video paměti vašeho systému, ale nic v příkazu programu neříká, že toto číslo je adresou. C vám taková přiřazení dovoluje. Avšak C++ podporuje mnohem přísněji souhlas typů a kompilátor vám dá chybové hlášení, které říká, že máte nesoulad typů. Chcete-li použít numerickou hodnotu jako adresu, měli byste použít přetypování, které konvertuje číslo na vhodný adresový typ:

```
int * pi;
pi = (int *) 0xB8000000; // typy se nyni shodují
```

Nyní obě strany přiřazovacího příkazu reprezentují adresy celých čísel, takže je přiřazení platné. Všimněte si, že je to pouze proto, že je adresa typu `int`, hodnota neznamená, že `pi` je sama typu `int`. Například ve velkém paměťovém modelu na IBM PC je typ `int` 2-bajtovou hodnotou, zatímco adresy jsou 4-bajtovou hodnotou.

Ukazatele mají některé další zajímavé vlastnosti, o kterých pojednáme, jakmile budou relevantní. Mezitím se podívejme, jak se mohou ukazatele používat na alokaci paměťového prostoru v době běhu programu.

Alokace paměti pomocí operátoru new

Nyní, když máte jistý pocit, jak ukazatele pracují, podívejme se jak mohou implementovat tento důležitý postup OOP, alokování paměti za běhu programu. Dosud jsme inicializovali proměnné adresami proměnných; proměnné jsou *pojmenovanou* pamětí alokovanou v době kompilace a ukazatele pouze poskytují druhé jméno pro paměť, ke které byste stejně mohli přistupovat podle jména. Pravá cena ukazatelů vstupuje do hry, když alokujete *nepojmenovanou* paměť na úschovu hodnot v době běhu programu. V tomto případě se ukazatele stávají jediným přístupem k této paměti. V C byste mohli alokovat paměť pomocí knihovní funkce `malloc()`. Stále to tak ještě můžete v C++ provádět, ale C++ má také lepší způsob, operátor `new`.

Vyzkoušejme tento nový postup vytvořením nepojmenované paměti za běhu programu pro hodnotu typu int a přístupem k hodnotě pomocí ukazatele. Klíčem je v C++ operátor new. Řeknete new, pro jaký typ dat chcete paměť; new nalezne blok správné velikosti a navrátí jeho adresu. Tuto adresu přiřadíte ukazateli a máte to. Zde je vzorek postupu:

```
int * pn = new int;
```

Část new int říká programu, že chcete jistou novou paměť vhodnou pro úschovu int. Operátor new používá na vyčíslení množství bajtů, které je potřeba na typ. Potom nalezne paměť a vrátí adresu. Dále přiřadí tuto adresu do pn, která se deklaruje jako typ ukazatele na int. Nyní je pn adresou a *pn hodnotou tam uloženou. Porovnejte to s přiřazením adresy proměnné ukazateli:

```
int higgens;
int * pi = &higgens;
```

V obou případech (pn a pi) přiřazujete ukazateli adresu int. Ve druhém případu můžete přistoupit k int pomocí jména: higgens. V prvém případu je váš jediný přístup prostřednictvím ukazatele. To vyvolává otázku: Protože paměť, na kterou pn ukazuje, postrádá jméno, jak ji voláte? Říkáme, že pn ukazuje na *datový objekt*. To není „objekt" ve smyslu „objektově orientovaného programování"; je to pouze „objekt" ve smyslu „věci". Výraz „datový objekt" je obecnější než výraz „proměnná", protože znamená jakýkoli blok paměti, který je alokovaný pro datovou položku. Tedy proměnná je také datovým objektem, ale paměť na kterou pn ukazuje, není proměnnou. Metoda ukazatelů na správu datových objektů může vypadat napoprvé nešikovně, ale nabízí větší dohled nad tím, jak váš program spravuje paměť.

Obecný tvar na získání přiřazení paměti pro jednoduchý datový objekt, který může být také strukturou stejně tak jako základním typem, je:

```
jméno_typu jméno_ukazatele = new jméno_typu;
```

Datový typ používáte dvakrát: jednou na určení druhu požadované paměti a jednou na deklaraci vhodného ukazatele. Samozřejmě, jestliže jste již deklarovali ukazatel správného typu, můžete ho použít radši než deklarovat nový. Výpis programu 4.12 ukazuje použití new se dvěma různými typy.

Výpis programu 4.12 use_new.cpp

```
// use_new.cpp _ použití operátoru new
#include <iostream>
using namespace std;
int main()
{
    int * pi = new int;          // alokuje prostor pro int
    *pi = 1001;                  // ukládá tam hodnotu

    cout << "int ";
    cout << "value = " << *pi << ": location = " << pi << "\n";

    double * pd = new double;   // alokuje prostor pro double
    *pd = 10000001.0;           // ukládá tam double
```

```
    cout << "double ";
    cout << "value = " << *pd << ": location = " << pd << "\n";
    cout << "size of pi = " << sizeof pi;
    cout << ": size of *pi = " << sizeof *pi << "\n";
    cout << "size of pd = " << sizeof pd;
    cout << ": size of *pd = " << sizeof *pd << "\n";
    return 0;
}
```

Zde je výstup:

```
int value = 1001: location = 007B0A60
double value = 1e+007: location = 007B0CD0
size of pi = 4: size of *pi = 4
size of pd = 4: size of *pd = 8
```

Poznámky k programu

Program používá `new` na alokaci paměti pro datové objekty typu `int` a `double`. To nastává, když program běží. Ukazatele `pi` a `pd` ukazují na tyto dva datové objekty. Bez nich nemůžete k těmto paměťovým lokacím přistupovat. S nimi můžete použít `*pi` a `*pn` zrovna tak, jako byste použili proměnné. Přiřazujete hodnoty do `*pi` a `*pd` přiřazením hodnot novým datovým objektům. Podobně, tisknete `*pi` a `*pd`, abyste tyto hodnoty zobrazili.

Program také ukazuje jeden z důvodů, proč musíte deklarovat typ, na který ukazatel ukazuje. Adresa sama o sobě odhalí pouze začátek adresy, kde je objekt uložen, nikoli jeho typ nebo počet použitých bajtů. Podívejte se na adresy obou hodnot. Jsou to pouze čísla s žádným typem nebo informací o velikosti. Také si všimněte, že velikost ukazatele na `int` je stejná jako velikost ukazatele na `double`. Oba jsou pouze adresami. Ale protože `use_new.cpp` deklaroval typy ukazatelů, program ví, že `*pd` je hodnotou `double` o 8 bajtech, zatímco `*pi` je hodnotou `int` o 4 bajtech. Když `use_new.cpp` tiskne hodnotu `*pd`, `cout` může říci, kolik bajtů má přečíst a jak je má interpretovat.

Mimo rozsah paměti?

Je pravděpodobné, že počítač možná nemá dostatek dostupné paměti , aby vyhověl požadavku `new`. Když nastane tento případ, `new` navrací hodnotu `0`. V C++ se ukazatel s hodnotou `0` nazývá *prázdným ukazatelem*. C++ zaručuje, že prázdný ukazatel nikdy neukazuje na platná data, takže se často používá na indikování selhání operátorů nebo funkcí, které jinak navracejí použitelné ukazatele. Až se trochu naučíte o příkazech `if` (v kapitole 6), můžete si ověřit, abyste viděli, zda `new` navrací prázdný ukazatel, a tedy chrání váš program před pokusem překročit jeho hranice. Navíc, po navrácení prázdného ukazatele po selhání alokace paměti, může `new` shodit výjimku `bad_alloc`. Kapitola 14 pojednává o mechanizmu výjimek.

Uvolnění paměti pomocí operátoru delete

Použití `new` na vyžádání paměti, když ji potřebujete, to je pouze fascinující půlka balíku C++ pro správu paměti. Druhou půlkou je operátor `delete`, který vám umožňuje vrátit paměť do paměťové oblasti, když jste s ní skončili. To je důležitým krokem směrem k vytvoření nejefektivnějšího využití paměti. Paměť, kterou vrátíte, neboli *uvolníte*, může být potom znovu použita jinou částí vašeho programu. Použijete `delete` následně s ukazatelem na blok paměti, který byl původně alokován pomocí `new`:

```
int * ps = new int; // alokace paměti pomocí new
delete ps;          // uvolnění paměti pomocí delete
```

Toto odstraňuje paměť, na kterou ukazuje `ps`; neodstraňuje to samotný ukazatel. Můžete ho znovu použít, například aby ukazoval na jinou novou lokaci. Měli byste vždy dát do rovnováhy použití `new` s použitím `delete`; jinak se vám bude paměť likvidovat pomocí úniku; to jest paměť, která se alokovala, ale dále se nemůže používat. Jestliže se únik paměti příliš zvětší, tak to může program při hledání další paměti zastavit.

Neměli byste se pokoušet uvolňovat blok paměti, který jste již uvolnili. Výsledek takového úsilí není definován. Také nemůžete použít `delete` na uvolnění paměti, která byla vytvořena deklarací proměnných:

```
int * ps = new int;      // v pořádku
delete ps;............   // v pořádku
delete ps;............   // nyní není v pořádku
int jugs = 5;            // v pořádku
int * pi = & jugs;       // v pořádku
delete pi;               // není povoleno, paměť nebyla alokována pomocí new
```

Upozornění: Použijte `delete` pouze k uvolnění paměti alokované pomocí `new`. Je však bezpečné aplikovat `delete` na nulový ukazatel.

Všimněte si, že rozhodujícím testem na použití `delete` je to, že ho použijete na paměť alokovanou pomocí `new`. To neznamená, že musíte použít stejný ukazatel, který jste potřebovali s `new`; místo toho musíte použít stejné adresy:

```
int * ps = new int;      // alokujte paměť
int * pq = ps;           // nastavte druhý ukazatel na stejný blok
delete pq;               // vymažte pomocí druhého ukazatele
```

Obyčejně dva ukazatele na stejný blok paměti nechcete vytvořit, protože tím vzrůstá pravděpodobnost, že se chybně pokusíte vymazat stejný blok dvakrát. Ale jak brzy uvidíte, použití druhého ukazatele má smysl, když pracujete s funkcí, která navrací ukazatel.

Vytváření dynamických polí pomocí operátoru new

Jestliže vše, co program potřebuje, je jednoduchá proměnná, měli byste rovněž deklarovat jednoduchou proměnnou, protože je to jednodušší, ne-li méně působivější, než pou-

žití new a ukazatele na správu jednoduchých malých objektů. Typičtěji používáte new spolu s pořádným kusem dat, jako například s poli, řetězci a strukturami. To znamená, kde je new užitečné. Předpokládejme například, že píšete program, který možná potřebuje a možná nepotřebuje pole, to závisí na informaci, kterou program dostane během svého běhu. Vytvoříte-li pole deklarováním, prostor se alokuje v době kompilace. Zda program nakonec pole používá nebo nepoužívá, existuje tam a spotřebovává paměť. Alokace pole během kompilace se nazývá *statickou vazbou*, znamenající, že je pole do programu vestavěno během kompilace. Ale pomocí new můžete pole vytvořit v době běhu programu, když ho potřebujete, nebo vytvoření přeskočit, když ho nepotřebujete. Nebo můžete zvolit velikost pole, až když program běží. To se nazývá *dynamickou vazbou*, která znamená, že se pole vytváří během běhu programu. Takové pole se nazývá *dynamickým polem*. Pomocí statické vazby musíte specifikovat velikost pole, když program píšete. Pomocí dynamické vazby se může program rozhodnout o velikosti pole, když běží.

Nyní se podíváme na dvě základní věci, které se týkají dynamických polí: jak použít operátor C++ new na vytvoření pole a jak použít ukazatel na přístup k prvkům pole.

Vytvoření dynamického pole pomocí operátoru new

V C++ je jednoduché vytvořit dynamické pole; sdělíte new typ prvku pole a jejich počet, který chcete. Syntaxe vyžaduje, že za jménem pole následuje počet prvků v hranatých závorkách. Například, potřebujete-li pole o 10 prvcích int, uděláte toto:

```
int * psome = new int [10]; // dostanete blok o 10 prvcích int
```

Operátor new navrací adresu prvního prvku bloku. V tomto příkladu se tato hodnota přiřadí ukazateli psome. Měli byste, až program přestane používat tento blok paměti, vyrovnat volání new s voláním delete.

Když použijete new na vytvoření pole, měli byste použít alternativní tvar delete, který indikuje, že uvolňujete pole:

```
delete [] psome; // uvolnění dynamického pole
```

Přítomnost hranatých závorek programu říká, že by měl uvolnit celé pole, ne pouze prvek, na který ukazoval ukazatel. Všimněte si, že hranaté závorky jsou mezi delete a ukazatelem. Použijete-li new bez hranatých závorek, použijte bez nich delete. Dřívější verze C++ neuměly rozpoznat notaci s hranatými závorkami. Se současnými verzemi není však výsledek záměny tvarů new a delete definován, což znamená, že se nemůžete spoléhat na některá určitá chování.

```
int * pi = new int;
short * ps = new short [500];
delete [] pi;      // výsledek není definovaný, nepoužívejte to
delete ps;         // výsledek není definovaný, nepoužívejte to
```

Zkrátka, používáte-li `new` a `delete`, zachovávejte tato pravidla:

- Nepoužívejte `delete` na uvolnění paměti, kterou nealokovala funkce `new`.
- Nepoužívejte `delete` na uvolnění stejného bloku paměti dvakrát za sebou.
- Použijte `delete []`, když jste na alokování paměti použili `new []`.
- Použijte `delete` (žádné hranaté závorky), když jste na alokování jednoduché entity použili `new`.
- Je bezpečné aplikovat `delete` na nulový ukazatel (nic se neděje).

Nyní se vrátíme k dynamickému poli. Všimněte si, že `psome` je ukazatelem na jednoduché `int`, první prvek bloku. Je na vaší zodpovědnosti sledovat, kolik prvků je v bloku. To znamená, když kompilátor nesleduje tu skutečnost, že `psome` ukazuje na první z 10 celých čísel, musíte napsat váš program tak, aby počet prvků sledoval.

Ve skutečnosti program sleduje množství alokované paměti tak, aby mohla být později správně uvolněna použitím operátoru `delete []`. Ale tato informace není veřejně přístupná; nemůžete použít operátoru `sizeof`, například na nalezení počtu bajtů v dynamicky alokované paměti.

Obecný tvar alokování a přiřazení paměti poli je tento:

```
jméno_typu jméno_ukazatele new jméno_typu [počet_prvků]:
```

Uplatnění operátoru `new` s ukazatelem `jméno_ukazatele`, který ukazuje na první prvek, zajišťuje dostatečně velký blok paměti na úschovu `počet_prvků` prvků typu `jméno_typu`. Jak si asi snažíte představit, můžete jako jméno pole používat `jméno_ukazatele` mnoha podobnými způsoby.

Použití dynamického pole

Jak použijete dynamické pole poté, co ho vytvoříte? Za prvé, přemýšlejte o problému koncepčně. Příkaz

```
int * psome = new int [10]; // dostane blok o 10 prvcích int
```

vytváří ukazatel `psome`, který ukazuje na první prvek bloku o 10 `int` hodnotách. Přemýšlejme o tom, jako o prstu ukazujícím na tento prvek. Předpokládejte, že `int` zaujímá čtyři bajty. Potom, posunutím palce o čtyři bajty správným směrem, můžete ukazovat na další prvek. Společně existuje 10 prvků, to je oblast, přes kterou můžete váš prst posouvat. Tedy příkaz `new` vám opatřuje veškerou informaci, kterou potřebujete na identifikaci každého prvku bloku.

Nyní přemýšlejte o problému prakticky. Jak přistoupíte k jednomu z těchto prvků? První prvek není problémem. Protože `psome` ukazuje na první prvek pole, `*psome` je hodnotou prvního prvku. To zanechává na zpřístupnění dalších devět prvků. Následující způsob, pokud neznáte C, vás může překvapit: Pouze použijete ukazatel, jako by byl jménem pole. To jest pro první prvek můžete použít `psome[0]` namísto `*psome`, `psome[1]` pro druhý atd. Použití ukazatele pro přístup k dynamickému poli se ukazuje velmi jednoduchým, třebaže to, proč metoda pracuje, možná není bezprostředně zřejmé. Důvodem, proč to můžete udělat je, že C i C++ obsluhují pole v každém případě vnitřně použitím ukazatelů. Tato vnitřní rovnocennost polí a ukazatelů je jednou z krás C a C++. Tuto rovnocennost budeme za chvíli rozvíjet. Za prvé, výpis programu 4.13 ukazuje, jak můžete použít

new na vytvoření dynamického pole a potom použít jeho označení na přístup k prvkům. Poukazuje také na základní rozdíl mezi ukazatelem a pravým jménem pole.

Výpis programu 4.13 arraynew.cpp

```
// arraynew.cpp _ používá pro pole operátor new
#include <iostream>
using namespace std;
int main()
{
    double * p3 = new double [3]; // místo na 3 double
    p3[0] = 0.2;                 // zachází s p3 jako se jménem pole
    p3[1] = 0.5;
    p3[2] = 0.8;
    cout << "p3[1] is " << p3[1] << ".\n";
    p3 = p3 + 1;                 // inkrementuje ukazatel
    cout << "Now p3[0] is " << p3[0] << " and ";
    cout << "p3[1] is " << p3[1] << ".\n";
    p3 = p3 - 1;                 // ukazuje zpět na začátek
    delete [] p3;                // uvolňuje paměť
    return 0;
}
```

Zde je výstup:

```
p3[1] is 0.5.
Now p3[0] is 0.5 and p3[1] is 0.8.
```

Jak můžete vidět, arraynew.cpp používá ukazatel p3, jako by to bylo jméno pole, p3[0] jako první prvek atd. Základní rozdíl mezi jménem pole a ukazatelem vidíte v následujícím řádku:

```
p3 = p3 + 1; // v pořádku pro ukazatele, chybně pro jména polí
```

Nemůžete měnit hodnotu jména pole. Ale ukazatel je proměnná, proto její hodnotu můžete měnit. Všimněte si efektu přidání 1 do p3. Výraz p3[0] se nyní odkazuje na prvek před druhým prvkem pole. Proto přidání 1 do p3 má za příčinu, že se nyní odkazuje na druhý prvek, ne na první. Odečtení jednotky vrací ukazatel zpět na původní hodnotu, takže program může poskytnout delete [] správnou adresu.

Skutečná adresa za sebou jdoucích prvků int se ve skutečnosti liší o dva nebo čtyři bajty, takže událost, že přidání 1 do p3 poskytuje adresu následujícího prvku naznačuje, že existuje něco zvláštního v aritmetice ukazatelů. A existuje.

Ukazatele, pole a aritmetika ukazatelů

Blízká rovnocennost ukazatelů a jmen polí pramení z aritmetiky ukazatelů a z toho, jak C++ vnitřně pracuje s poli. Za prvé prověřme aritmetiku. Přidání jednotky do celočíselné proměnné zvyšuje její hodnotu o 1, ale přidání 1 do proměnné ukazatel zvyšuje jeho hodnotu o počet bajtů typu, na který ukazuje. Přidání 1 k ukazateli na `double` přidává 8 k numerické hodnotě na systému s 8-bajtovým `double`, zatímco přidání 1 k ukazateli na `short`, přidává hodnotě ukazatele 2, má-li `short` 2 bajty. Výpis programu 4.14 tuto úžasnou vlastnost ukazuje. Ukazuje také další důležitou vlastnost: C++ interpretuje jméno pole jako adresu.

Výpis programu 4.14 addpntrs.cpp

```
using namespace std;
int main()
{
    double wages[3] = {10000.0, 20000.0, 30000.0};
    short stacks[3] = {3, 2, 1};

    // Zde jsou dva způsoby získání adresy pole
    double * pw = wages;     // jméno pole = adresa
    short * ps = &stacks[0]; // nebo použití adresového operátoru
                             // with array element
    cout << "pw = " << pw << ", *pw = " << *pw << "\n";
    pw = pw + 1;
    cout << "add 1 to the pw pointer:\n";
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";

    cout << "ps = " << ps << ", *ps = " << *ps << "\n";
    ps = ps + 1;
    cout << "add 1 to the ps pointer:\n";
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";
    cout << "access two elements with array notation\n";
    cout << stacks[0] << " " << stacks[1] << "\n";
    cout << "access two elements with pointer notation\n";
    cout << *stacks << " " << *(stacks + 1) << "\n";

    cout << sizeof wages << " = size of wages array\n";
    cout << sizeof pw << " = size of pw pointer\n";
    return 0;
}
```

Zde je výstup:

```
pw = 0068FDE0, *pw = 10000
add 1 to the pw pointer:
pw = 0068FDE8, *pw = 20000
ps = 0068FDD0, *ps = 3
```

```
add 1 to the ps pointer:
ps = 0068FDD2, *ps = 2
access two elements with array notation
3 2
access two elements with pointer notation
3 2
24 = size of wages array
4 = size of pw pointer
```

Poznámky k programu

Ve většině souvislostí C++ interpretuje jméno pole jako adresu jeho prvního prvku. Tedy příkaz

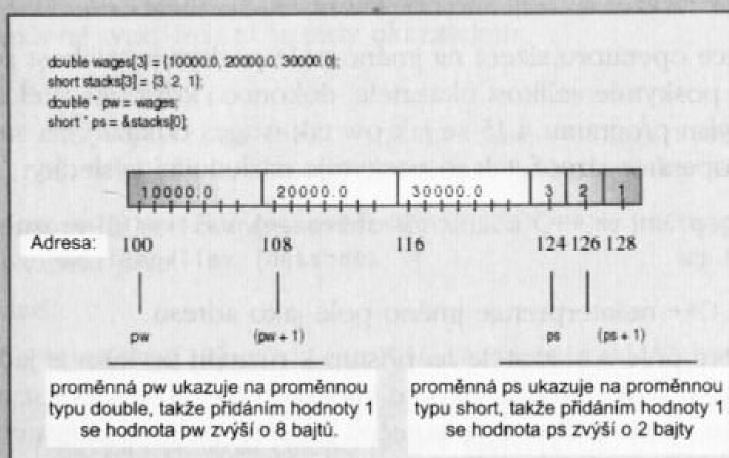
```
double * pw = wages;
```

vytváří z `pw` ukazatel na typ `double` a inicializuje ho na `wages`, což je adresa prvního prvku pole `wages`. Pro `wages`, stejně tak pro jiná pole, máme následující rovnost:

```
wages = &wages[0] = adresa prvního prvku pole
```

Pouze abychom ukázali, že to není žádný nesmysl, program explicitně používá adresový operátor ve výrazu `&stack[0]` na inicializaci ukazatele `ps` prvním prvkem pole `stacks`.

Dále program dohlíží na hodnoty `pw` a `*pw`. První je adresou, druhá je hodnotou na této adrese. Protože `pw` ukazuje na první prvek, zobrazenou hodnotou `*pw` je hodnota prvého prvku, 10000. Potom program do `pw` přičítá 1. Jak bylo slíbeno, toto přidává 8 (E0 + 8 = E8 hexadecimálně) k numerické hodnotě adresy. To způsobí, že se `pw` rovná adrese druhého prvku. Tedy, `*pw` je nyní 20000, hodnota druhého prvku. Viz obrázek 4.10. (Adresové hodnoty na obrázku jsou přizpůsobené, aby byl obrázek jasnější.)



Obrázek 4.10 Sčítání ukazatelů

Poté program prochází obdobnými kroky pro `ps`. Nyní, protože `ps` ukazuje na typ `short` a protože `short` má 2 bajty, přidání 1 k ukazateli zvýší jeho hodnotu o 2. Výsledkem je opět vytvoření ukazatele, který ukazuje na další prvek pole.

Pamatujte:

Pamatujte, přidání 1 do ukazatelové proměnné zvyšuje její hodnotu o počet bajtů typu, na který ukazuje.

Nyní uvažujme výraz pole `stacks[1]`. Kompilátor C++ zachází s výrazem přesně tak, jako byste napsali `*(stacks + 1)`. Druhý výraz znamená vypočítejte adresu druhého prvku pole a potom zjistěte hodnotu, která je tam uložena. Konečným výsledkem je přesně to, co znamená `stacks[1]`. (Priorita operátorů vyžaduje, abyste použili závorky. Bez nich by se 1 přičetla k `*stacks` namísto k `stacks`.)

Výstup programu ukazuje, že jsou `*(stacks + 1)` a `stacks[1]` stejné. Podobně, `*(stacks + 2)` je stejný jako `stacks[2]`. Obecně, kdykoli použijete označení pole, C++ vytváří následující konverzi:

```
jméno_pole[i] se stává *(jméno_pole + i)
```

A použijete-li ukazatel místo jména pole, C++ vytváří stejnou konverzi:

```
jméno_ukazatele[i] se stává *(jméno_ukazatele + 1)
```

Tedy, v mnoha ohledech můžete používat jména ukazatelů a jména polí stejným způsobem. S oběma můžete používat označení pole s hranatými závorkami. Na obě můžete aplikovat dereferenční operátor (*). Ve většině výrazů navzájem představují adresu. Jedním rozdílem je to, že hodnotu ukazatele můžete měnit, zatímco jméno pole je konstantou:

```
jméno_ukazatele = jméno_ukazatele + 1;      // platné
jméno_pole = jméno_pole + 1;                // neplatné
```

Druhým rozdílem je, že aplikace operátoru sizeof na jméno pole poskytuje velikost pole, ale aplikace sizeof na ukazatel poskytuje velikost ukazatele, dokonce i když ukazatel ukazuje na pole. Například ve výpisu programu 4.15 se jak pw tak wages odkazují na stejné pole. Ale aplikujeme-li na ně operátor sizeof, tak to poskytuje následující výsledky:

```
24 = velikost pole wages              // zobrazení velikosti wages
4 = velikost ukazatele pw             // zobrazení velikosti pw
```

To je jeden případ, ve kterém C++ neinterpretuje jméno pole jako adresu.

Zkrátka, použití `new` na vytvoření pole a ukazatele na přístup k různým prvkům je jednodušší záležitostí. Pouze zacházení s ukazatelem jako se jménem pole. Avšak porozumění, proč to pracuje, je zajímavou výzvou. Chcete-li skutečně rozumět polím a ukazatelům, měli byste si opět pečlivě prohlédnout jejich vzájemný vztah. Ve skutečnosti byste měli být vystaveni doopravdy pořádně znalostem o ukazatelích později, takže shrneme, co bylo o ukazatelích a polích dosud odhaleno.

Shrnutí vlastností ukazatelů

Deklarace ukazatelů: Abyste deklarovali ukazatel určitého typu, použijte tento tvar:

```
jméno_typu * jméno_ukazatele
```

Příklady:

```
double * pn;        // pn ukazuje na hodnotu double
char * pc;          // pc ukazuje na hodnotu char
```

Zde jsou ukazatele `pn` a `pc` a `double *` a `char *` jsou zápisy v C++ pro typy ukazatel na `double` a na `char`.

Přiřazení hodnoty ukazateli: Ukazateli byste měli přiřadit adresu paměti. Můžete použít operátor `&` na jméno proměnné na získání adresy pojmenované paměti a operátor `new` vrací adresu nepojmenované paměti.

Příklady:

```
double bubble = 3.2;
pn = &bubble;  // přiřazuje adresu bubble do pn
pc = new char; // přiřazuje adresu nově alokované paměti char do pc
```

Dereferencování ukazatelů: Dereferencování ukazatele znamená odkazovat se na ukazovanou hodnotu. Použijte na ukazatel dereferenční operátor (*) neboli nepřímou hodnotu a dereferencujte ho. Tedy, jestliže `pn` je ukazatelem na `bubble`, jako v posledním příkladě, potom `*pn` je ukazovanou hodnotou, neboli v tomto případě 3.2.

Příklady:

```
cout << *pn;  // tiskne hodnotu bubble
*pc = 'S';    // umístěte 'S' na paměťovou lokaci, jejíž adresa je pc
```

Nikdy nedereferencujte ukazatel, který nebyl inicializován řádnou adresou.

Rozlišení mezi ukazatelem a hodnotou, na kterou se ukazuje: Pamatujte, jestliže je `pi` ukazatel na `int`, potom `*pi` není ukazatelem na `int`; místo toho je `*pi` úplným ekvivalentem proměnné typu `int`. `pi` je tedy ukazatelem.

Příklady:

```
int * pi = new int;    // přiřazení adresy ukazateli
pi *pi = 5;            // uloží hodnotu 5 na tuto adresu
```

Jména polí: Ve většině souvislostí nakládá C++ se jmény polí jako by to byla adresa prvního prvku pole.

Příklad:

```
int tacos[10];             // tacos je nyní to samé jako &tacos[0]
```

Existuje jedna výjimka, kdy používáte jméno pole s operátorem `sizeof`. V tomto případě `sizeof` navrací velikost celého pole v bajtech.

Aritmetika ukazatelů: C++ vám dovoluje přičítat k ukazateli celé číslo. Výsledek přičtení 1 se rovná původní hodnotě adresy plus hodnotě, která se rovná počtu bajtů objektu, na který se ukazuje. Také můžete celé číslo od ukazatele odečítat a dostanete rozdíl mezi dvěma ukazateli. Poslední operace, která poskytuje celé číslo, má význam pouze tehdy,

když oba ukazatele ukazují do stejného pole (ukazovat na jednu pozici za koncem je také povoleno); poskytuje to potom odstup mezi těmito dvěma prvky.

Příklady:

```
int tacos[10] = {5,2,8,4,1,2,2,4,6,8};
int * pt = tacos;    // předpokládejme, že pt a pe mají stejnou adresu 3000
pt = pt + 1;         // nyní je pt 3004 pokud je int čtyři bajty
int *pe = &tacos[9];// pe je 3036 pokud je int čtyři bajty
pe = pe -1;          // nyní je pe 3032, adresa tacos[8]
int diff = pe - pt; // rozdíl je 7, odstup mezi
                     // tacos[8] a tacos[1]
```

Dynamická a statická vazba pro pole: Použijte deklaraci pole na vytvoření pole se statickou vazbou, to jest pole, jehož velikost se nastaví během doby kompilace:

```
int tacos[10]; // statická vazba, pevná velikost v době kompilace
```

Použijte operátor `new []` na vytvoření pole s dynamickou vazbou (dynamické pole), to jest pole, které se alokuje a jehož velikost se může nastavit během běhu programu. Uvolněte paměť pomocí `delete []`, když jste skončili:

```
int size; cin >> size;
int * pz = new int [size]; // dynamická vazba, velikost se nastaví v době
běhu
...
delete [] pz; // uvolněte paměť, když skončíte
```

Značení polí a ukazatelů: Použití označení pole s hranatými závorkami je ekvivalentní dereferencování ukazatele:

```
tacos[0] znamená *tacos znamená hodnotu na adrese tacos
tacos[3] znamená *(tacos + 3) znamená hodnotu na adrese tacos + 3
```

To platí jak pro jména polí, tak pro ukazatelové proměnné, takže můžete na ukazatele nebo jména polí použít jak značení ukazatelů, tak polí.

Příklady:

```
int * pi = new int [10];   // pi ukazuje na blok 10 int
*pi = 5;                    // nastavuje nulový prvek na 5
pi[0] = 6;                  // přenastavuje nulový prvek na 6
pi[9] = 44;                 // nastavuje desátý prvek na 44
int tacos[10];
*(tacos + 4) = 12;         // nastavuje tacos[4] na 12
```

Ukazatele a řetězce

Zvláštní vztah mezi poli a ukazateli se rozšiřuje na řetězce. Uvažujme následující programový kód:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

Jméno pole je adresou jeho prvého prvku, takže flower v příkazu cout je adresou prvku char, který obsahuje znak r. Objekt cout předpokládá, že adresa char je adresou řetězce, takže tiskne znak, který je na této adrese, a potom pokračuje tisknutím znaků dokud nenarazí na prázdný znak (\0). Zkrátka, dáte-li cout adresu znaku, vytiskne vše od tohoto znaku až po první prázdný znak, který ho následuje.

Kritickým faktorem tady není, že flower je jménem pole, ale že se chová jako adresa char. To má za následek, že můžete použít ukazatel na proměnnou char jako parametr pro cout, tedy proto, že je také adresou char. Samozřejmě, že by měl ukazatel ukazovat na začátek řetězce. Vyzkoušíte si to za chvíli.

Ale nejprve, co takhle poslední část předcházejícího příkazu cout? Je-li flower skutečně adresou prvního znaku řetězce, co je výraz "s are red\n" ? Abychom byli konzistentní se správou řízení výstupu řetězce pomocí cout, tento řetězec v uvozovkách by měl také být adresou. A on je, co se týče řetězce v uvozovkách v C++, podobně jako jméno pole, slouží jako adresa jeho prvního prvku. Předcházející programový kód skutečně neposílá do cout celý řetězec, posílá pouze adresu řetězce. To znamená, že řetězce v poli, řetězcové konstanty v uvozovkách a řetězce, které jsou popsány pomocí ukazatelů, jsou všechny ovládány rovnocenně. Každý je skutečně posílán jako adresa. To je určitě méně práce, než posílat navzájem každý znak v řetězci.

Pamatujte:

V cout a ve většině výrazů C++ je jméno pole char, ukazatel na char a řetězcová konstanta v uvozovkách považována za adresu prvého znaku řetězce.

Výpis programu 4.15 ukazuje použití různých tvarů řetězců. Používá dvě funkce z řetězcové knihovny. Funkce strlen(), kterou jsme použili dříve, navrací délku řetězce. Funkce strcpy() kopíruje řetězec z jednoho místa na jiné. Obě mají funkční prototypy v hlavičkovém souboru cstring (nebo string.h na starších implementacích). Program také předvádí některá nesprávná použití ukazatelů, kterým byste se měli vyhnout.

Výpis programu 4.15 ptrstr.cpp

```
// ptrstr.cpp _ používá ukazatele na řetězce
#include <iostream>
using namespace std;
#include <cstring>              // deklaruje strlen(), strcpy()
int main()
{
    char animal[20] = "bear";   // animal obsahuje bear
    const char * bird = "wren"; // bird obsahuje adresu řetězce
    char * ps;                  // neinicializováno

    cout << animal << " and ";  // zobrazuje bear
    cout << bird << "\n";       // zobrazuje wren
    cout << ps << "\n";         // hrubá chyba - zobrazuje nesmysl
    cout << "Enter a kind of animal: ";
    cin >> animal;              // ok if input < 20 chars
```

```
    // cin >> ps; Zkoušet to je příliš strašnou hrubou chybou;
    // ps neukazuje na alokovaný prostor

    ps = animal;                    // nastavuje ps, aby ukazovala na řetězec
    cout << ps << "s!\n";           // v pořádku, stejné jako použití animal
    cout << "Before using strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;
    ps = new char[strlen(animal) + 1];  // získá novou paměť
    strcpy(ps, animal);             // kopíruje řetězec do nové paměti
    cout << "After using strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;
    delete [] ps;
    return 0;
}
```

Kompatibilita:

Nemá-li váš systém hlavičkový soubor `cstring`, použijte starší verzi `string.h`. První pokus o zobrazení `ps` může vést k chybě za běhu programu.

Zde je vzorek běhu programu:

```
bear and wren
          " output from cout << ps << "\n";
Enter a kind of animal: fox
foxs!
Before using strcpy():
fox at 0068FDE4
fox at 0068FDE4
After using strcpy():
fox at 0068FDE4
fox at 007B0D80
```

Poznámky k programu

Program na výpisu programu 4.15 vytváří jedno pole `char` (`animal`) a dvě proměnné ukazatele na `char` (`bird` a `ps`). Program začíná inicializací pole `animal` řetězcem „`bear`", stejně jako jsme je inicializovali dříve. Inicializuje ukazatel na `char` řetězcem:

```
const char * bird = "wren"; // bird obsahuje adresu řetězce
```

Pamatujte, „wren" ve skutečnosti představuje adresu řetězce, takže tento příkaz přiřazuje adresu „wren" ukazateli `bird`. (Ve skutečnosti si kompilátor rezervuje oblast v paměti na uchování všech použitých řetězců ve zdrojovém kódu programu spojením každého uloženého řetězce s jeho adresou.) To znamená, že můžete použít ukazatel bird stejně, jako byste použili řetězec „`wren`", jako v `cout << "A concerned " << bird << " speaks\n"`. Řetězcové literály jsou konstanty, a proto programový kód používá v deklaraci klíčové slovo `const`. Použití `const` v této podobě znamená, že můžete použít `bird` pro přístup k ře-

těžci, ale ne k jeho změně. Kapitola 7 pojednává o tématu `konstantních` ukazatelů mnohem podrobněji. Konečně, ukazatel `ps` zůstává neinicializován, takže neukazuje na žádný řetězec. (Vzpomeňte si, toto je obvyklý a špatný nápad, ale tento příklad není výjimkou.)

Dále program ukazuje, že můžete použít v `cout` jméno pole `animal` a ukazatel `bird` ekvivalentně. Oba jsou především adresami řetězců a `cout` zobrazuje oba řetězce („`bear`" a „`wren`"), které jsou na těchto adresách uloženy. Když programový kód chybuje při pokusu o zobrazení `ps`, dostaneme prázdný řádek. Vytvoření neinicializovaného ukazatele je něco jako distribuce prázdného podepsaného šeku; ztrácíte kontrolu nad tím, jak bude použit. My jsme zde, co se týče `ps`, trochu šťastnější, ač neinicializovaný, mohl by ukazovat náhodně na nějakou nevhodnou paměťovou pozici. V tomto případě ukazuje na místo, které obsahuje nulu, takže se nic nezobrazí. Jinak byste mohli vytvořit nějaký nesmyslný výstup.

Pro vstup je situace trochu jiná. Pokud je vstup dostatečně krátký, pole `animal` pro vstupní data vyhovuje. Avšak správné by nebylo pro vstupní data použít `bird`.

Některé kompilátory považují řetězcové literály za konstanty, které se dají pouze číst, což vede k chybě běhu programu při pokusu přepsat je novými daty. Že jsou řetězcové literály konstantou, je nařízeným chováním v C++, ale všechny kompilátory ještě neprovedly tuto změnu ze staršího chování.

Některé kompilátory používají na reprezentování všech výskytů literálu v programu pouze jedinou kopii řetězcového literálu.

Rozšiřme druhý bod. C++ nezaručuje, že jsou řetězcové literály uloženy jedinečně. To jest, používáte-li řetězcový literál „`wren`" v programu několikrát, kompilátor může uložit několik kopií tohoto řetězce nebo pouze jedinou. Nastane-li druhý případ, potom nastavíme-li ukazatel `bird`, aby ukazoval na jediný řetězec „`wren`", tak ho to přinutí, aby ukazoval pouze na jedinou kopii řetězce. Načtení hodnoty do jednoho řetězce by mohlo ovlivnit nikoli to, co jste mysleli, ale mohl by to být nezávislý řetězec kdekoli. V každém případě, protože ukazatel `bird` je deklarován jako konstanta, kompilátor zabraňuje jakékoli snaze změnit obsah lokace, na kterou `bird` ukazuje.

Ještě horší je, když se pokoušíte uložit informaci na lokaci, kam ukazuje `ps`. Protože `ps` není inicializován, nevíte, kam se informace uloží. Dokonce to může přepsat informaci již v paměti uloženou. Naštěstí je snadné se těmto problémům vyhnout, pouze použijte dostatečně velké pole `char` pro přijetí vstupních dat. Na přijetí vstupních dat nepoužívejte řetězcové konstanty nebo neinicializované ukazatele.

Upozornění:

Když načítáte do programu řetězec, měli byste vždy použít adresy dříve alokované paměti. Tato adresa může být ve tvaru jména pole nebo ukazatele, který byl inicializován pomocí `new`.

Dále si všimněte, co uskutečňuje následující programový kód:

```
ps = animal;                    // nastaví ps, aby ukazoval na řetězec
  ...
```

```
cout << animal << " at " << (int *) animal << endl;
cout << ps << " at " << (int *) ps << endl;
```

```
Produkuje následující výstup:
```

```
fox at 0068FDE4
fox at 0068FDE4
```

Normálně, když dodáte objektu `cout` ukazatel, tiskne adresu. Ale když je ukazatel typu `char *`, `cout` zobrazuje řetězec, na který se ukazuje. Chcete-li vidět adresu řetězce, musíte přetypovat ukazatel jiným typem ukazatele, jako například na `int *`, což dělá tento programový kód. Takže `ps` zobrazuje „`fox`" jako řetězec, ale `(int *) ps` ho zobrazuje jako adresu, na které se nalézá. Všimněte si, že přiřazením `animal` do `ps` nekopíruje řetězec, ale kopíruje adresu. To má za důsledek dva ukazatele (`animal` a `ps`) na stejnou lokaci a řetězec.

Abyste získali kopii řetězce, musíte udělat více. Za prvé musíte alokovat na úschovu řetězce paměť. Můžete to udělat deklarací dalšího pole nebo použitím `new`. Druhý přístup vám umožňuje uživatelsky přizpůsobit paměť řetězci:

```
ps = new char[strlen(animal) + 1]; // získá novou paměť
```

Řetězec „`fox`" nenaplní úplně pole `animal`, takže plýtváme prostorem. Tento kousek programového kódu používá `strlen()` na zjištění délky řetězce; přidává 1 na získání délky včetně prázdného znaku. Potom používá `new` na alokování pouze takového paměťového prostoru, aby se do něj řetězec vešel.

Nyní potřebujete způsob, jak kopírovat řetězec z pole `animal` do nově alokovaného místa. Přiřazení `animal` do `ps` nefunguje, protože to pouze mění adresu v `ps` uloženou, a tedy ztrácí jedinou cestu, kterou musel program přistupovat k nově alokované paměti. Místo toho musíte použít knihovní funkci `strcpy()`:

```
strcpy(ps, animal); // kopíruje řetězec do nové paměti
```

Funkce `strcpy()` má dva parametry. První je adresou příjemce, druhý je adresou řetězce, který se má kopírovat. Je na vás, abyste zajistili, že příjemce je skutečně alokován a má dostatek prostoru na úschovu kopie. Toto se zde zajišťuje použitím `strlen()` k nalezení správné velikosti a použitím `new` na získání volné paměti.

Často narazíte na potřebu umístit řetězec do paměti. Operátor = použijte, když pole inicializujete; jinak použijte `strcpy()` nebo `strncpy()`. Viděli jste funkci strcpy():

```
char food[20] = "carrots"; // inicializace
strcpy(food, "flan");      // jinak
```

Všimněte si, že něco jako

```
strcpy(food, "a picnic basket filled with many goodies");
```

může zapříčinit problémy, protože pole food je menší než řetězec. V tomto případě funkce kopíruje zbytek řetězce do paměťových bajtů bezprostředně následujících za polem, což může přepsat další paměť, kterou váš program používá. Abyste se vyhnuli tomuto problému, použijte místo toho `strncpy()`. Používá třetí parametr: maximální počet znaků, který se má kopírovat. Buďte si však vědomi, že jestliže funkce překročí prostor dříve, než dosáhne konce řetězce, nepřidá prázdný znak. Mohli byste tedy funkci použít takto:

```
strncpy(food, "a picnic basket filled with many goodies", 19);
food[19] = '\0';
```

Kopíruje až 19 znaků do pole a potom nastaví poslední prvek na prázdný znak. Je-li řetězec kratší než 19 znaků, `strncpy()` přidá prázdný znak na označení správného konce řetězce dříve.

Pamatujte

Pro přiřazení řetězce do pole používejte funkce `strcpy()` a `strncpy()`, nikoli operátor přiřazení.

Vytváření dynamických struktur pomocí operátoru new

Viděli jste, jak může být výhodné vytvářet pole za běhu programu než během kompilace. To samé platí pro struktury. Musíte alokovat prostor pouze pro tolik struktur, kolik jich program během určitého běhu potřebuje. Ještě jednou, operátor `new` je prostředkem, který potřebujete. Pomocí něho můžete vytvořit dynamické struktury. Ještě jednou, „dynamický„ znamená, že je paměť alokována během běhu programu, nikoli během kompilace. Mimochodem, protože třídy jsou téměř jako struktury, budete schopni používat postupy, které se naučíte pro struktury, také se třídami.

Použití `new` s třídami má dvě části: vytvoření struktury a přístup k jejím členům. Abyste vytvořili strukturu, použijte typ struktury spolu s `new`. Například na vytvoření nepojmenované struktury typu `inflatable` a přiřazení její adresy vhodnému ukazateli můžete udělat následující:

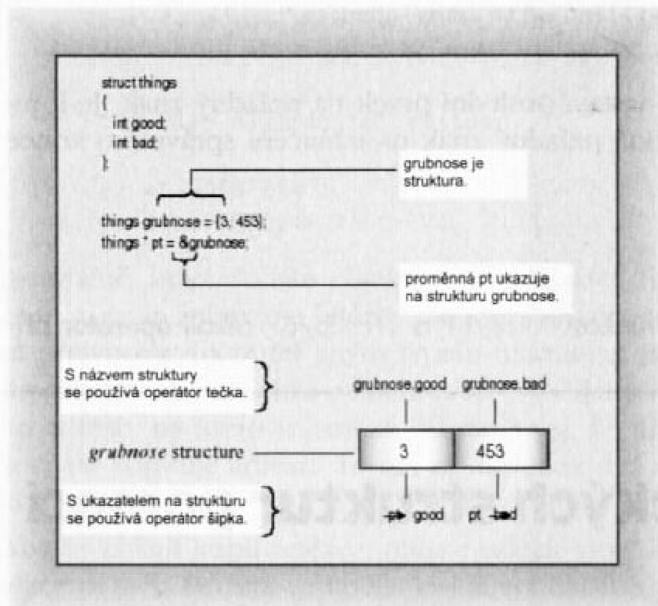
```
inflatable * ps = new inflatable;
```

Toto přiřazuje ukazateli ps adresu dostatečně velkého kusu paměti na úschovu struktury typu `inflatable`. Všimněte si, že syntaxe je přesně stejná jako pro vestavěné typy v C++.

Ožehavou věcí je přístup ke členům. Když vytváříte dynamickou strukturu, nemůžete použít tečkového operátoru příslušnosti se jménem struktury, protože struktura nemá jméno. Vše, co máte, je její adresa. C++ poskytuje operátor právě pro tuto situaci: operátor příslušnosti šipka (->). Tento operátor, který se formuje pomocí pomlčky a symbolu větší než, je pro ukazatele na struktury tím samým, co tečka pro jména struktur. Například, ukazuje-li ps na jméno struktury `inflatable`, potom je `ps->price` členem struktury `price`, na kterou se ukazuje. Viz obrázek 4.11.

Pamatujte:

Občas bývají noví uživatelé na pochybách, kdy mají používat operátor tečka a kdy operátor šipka na určení členu struktury. Pravidlo je jednoduché. Je-li identifikátorem struktury její jméno, použijte operátor tečku. Je-li identifikátorem ukazatel na strukturu, použijte operátor šipku.



Obrázek 4.11 Identifikace členů struktury

Druhý přístup horší na pochopení je, že jestliže `ps` je ukazatelem na strukturu, potom `*ps` představuje hodnotu, na kterou se ukazuje, na strukturu samotnou. Tedy, protože `ps` je strukturou, `(*ps).price` je člen struktury `price`. Pravidla priority v C++ vyžadují, abyste v této konstrukci použili závorky.

Výpis programu 4.16 používá `new` na vytvoření nepojmenované struktury a ukazuje oba zápisy s ukazateli pro přístup ke členům struktury.

Výpis programu 4.16 newstrct.cpp

```
// newstrct.cpp _ použití new se strukturou
#include <iostream>
using namespace std;
struct inflatable   // šablona struktury
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    inflatable * ps = new inflatable; // alokuje prostor pro strukturu

    cout << "Enter name of inflatable item: ";
    cin.get(ps->name, 20);          // metoda 1 pro přístup ke členu
    cout << "Enter volume in cubic feet: ";
    cin >> (*ps).volume;            // metoda 2 pro přístup ke členu
    cout << "Enter price: $";
    cin >> ps->price;
    cout << "Name: " << (*ps).name << "\n";  // metoda 2
```

```
    cout << "Volume: " << ps->volume << " cubic feet\n";
    cout << "Price: $" << ps->price << "\n"; // metoda 1
    return 0;
}
```

Zde je ukázka běhu programu:

```
Enter name of inflatable item: Fabulous Frodo
Enter volume in cubic feet: 1.4
Enter price: $17.99
Name: Fabulous Frodo
Volume: 1.4 cubic feet
Price: $17.99
```

Příklad na použití new a delete

Podívejme se na příklad, který používá `new` a `delete` na řízení ukládání vstupních řetězců z klávesnice. Výpis programu 4.17 definuje funkci, která navrací ukazatel na vstupní řetězec. Tato funkce čte vstupní data do velkého dočasného pole a potom používá `new []` na vytvoření části paměti takové velikosti, aby se do ní vstupní řetězec vešel. Potom funkce navrací ukazatel na blok. Tento přístup by mohl ušetřit mnoho paměti programům, které čtou velký počet řetězců.

Předpokládejme, že váš program musí přečíst 1000 řetězců a že největší řetězec může být 79 znaků dlouhý, ale většina řetězců je mnohem kratších. Kdybyste použili `znaková` pole na úschovu řetězců, potřebovali byste 1000 polí po 80 znacích. To jest 80 000 bajtů a většina tohoto bloku paměti by byla nevyužitá. Alternativně byste mohli vytvořit pole o 1000 ukazatelích na `char` a potom použít `new` na alokování pouze takového množství paměti, které je potřebné pro každý řetězec. To by mohlo uspořit desítky tisíc bajtů. Místo toho, abyste museli mít pro každý řetězec velké pole, přizpůsobte paměť vstupním datům. Ještě lépe, `new` můžete také použít na zjištění prostoru na uložení pouze tolika ukazatelů, kolik jich potřebujete. Dobrá, nyní je to poněkud ambiciózní. Dokonce použití pole o 1000 ukazatelích je nyní trochu přehnaný požadavek, ale výpis programu 4.17 ukazuje část z postupu. Používá ji také pouze pro uvolnění a znovupoužití paměti, aby ukázal, jak `delete` pracuje.

Výpis programu 4.17 delete.cpp

```
// delete.cpp _ použití operátoru delete
#include <iostream>
#include <cstring>      // nebo string.h
using namespace std;
char * getname(void);   // funkční prototyp
int main()
{
    char * name;          // vytváří ukazatel, ale ne paměť

    name = getname();   // přiřazuje adresu řetězce do name
    cout << name << " at " << (int *) name << "\n";
    delete [] name;     // uvolnil paměť
```

```
    name = getname();   // nové použiti uvolněné paměti
    cout << name << " at " << (int *) name << "\n";
    delete [] name;     // opět uvolnil paměť
    return 0;
}

char * getname()        // vrací ukazatel novému řetězci
{
    char temp[80];      // dočasná paměť

    cout << "Enter last name: ";
    cin >> temp;
    char * pn = new char[strlen(temp) + 1];
    strcpy(pn, temp);   // kopíruje řetězec do menšího prostoru

return pn;          // temp se ztrácí, když funkce končí
}
```

Zde je ukázka běhu programu:

```
Enter last name: Fredeldumpkin
Fredeldumpkin at 007B0B40
Enter last name: Pook
Pook at 007B0DC0
```

Poznámky k programu

Za prvé uvažujme funkci `getname()`. Používá objekt `cin` na umístění vstupního slova do pole `temp`. Dále používá `new` na alokování nové paměti na úschovu toho slova. Program potřebuje `strlen(temp) + 1` znak na uložení řetězce včetně prázdného znaku, což je hodnota, která je předává metodě `new`. Poté, co je prostor přístupný, `getname()` používá standardní knihovní funkci `strcpy()` na zkopírování řetězce z `temp` do nového bloku. Funkce neprovádí kontrolu na prozkoumání, zda se řetězec vejde nebo ne, ale `getname()` to pokrývá žádostí o přesný počet znaků pro `new`. Nakonec funkce vrací v ukazateli `pn` adresu kopie řetězce.

V `main()` je návratová hodnota (adresa) přiřazena ukazateli `name`. Tento ukazatel se definuje v `main()`, ale ukazuje na blok paměti, která je alokována ve funkci `getname()`. Program potom tiskne řetězec a jeho adresu.

Když uvolní blok, na který `name` ukazuje, `main()` volá funkci `getname()` podruhé. C++ nezaručuje, když se příště použije `new`, že nově uvolněná paměť bude první, která má být vybrána, a ani v této ukázce běhu programu tomu tak není.

Abyste ocenili některé z důvtipnějších rysů programu, měli byste vědět trochu více o tom, jak C++ spravuje paměť. Proto si prohlédneme některé prostředky, které se plněji pokrývají v kapitole 8.

Automatická, statická a volná paměť

C++ má pro data tři způsoby řízení paměti, které jsou závislé na použitém způsobu alokace paměti: *automatická, statická* a *volná paměť*. Datové objekty alokované těmito třemi způsoby se jeden od druhého liší tím, jak dlouho existují. Letmo se na každý typ podíváme.

Automatické proměnné

Obvykle se proměnné definované uvnitř funkce nazývají *automatickými proměnnými*. Přicházejí na svět automaticky, když se vyvolá funkce, která je obsahuje, a jejich platnost skončí, když se funkce ukončí. Například pole `temp` ve výpisu programu 4.17 existuje pouze tehdy, když je funkce `getname()` aktivní. Když program vrací řízení do `main()`, použitá paměť pro `temp` se automaticky uvolní. Jestliže `getname()` vrátila adresu `temp`, ukazatel `name` ve funkci `main()` by měl přestat ukazovat na paměťovou lokaci, která by měla být brzy uvolněna. To je jeden z důvodů, proč jsme použili `new` ve funkci `getname()`.

Skutečně, automatické hodnoty jsou lokální v bloku, který je obsahuje. Blok je sekcí programového kódu uzavřeného mezi složenými závorkami. Ale jak uvidíte v následující kapitole, bloky můžete mít uvnitř funkcí. Definujete-li proměnnou uvnitř jednoho z těchto bloků, existuje pouze tehdy, když program provádí příkazy uvnitř tohoto bloku.

Statická paměť

Statická paměť existuje během provádění celého programu. Existují dva způsoby vytváření statické proměnné. Jeden je definovat ji externě, vně funkce. Druhý je použití klíčového slova `static`, když se proměnná deklaruje:

```
static double fee = 56.50;
```

Na K&R C můžete pouze inicializovat statická pole a struktury, zatímco C++ verze 2.0 (a pozdější) a ANSI C vám dovolují inicializovat také automatická pole a struktury. Avšak, jak někteří z vás mohli zjistit, některé implementace C++ inicializaci automatických polí a struktur nezahrnují.

Kapitola 8 o statické paměti pojednává detailněji. Hlavní bod, který se týká automatické a statické paměti, spočívá v tom, že tyto metody přísně definují dobu života proměnné. Proměnná buď existuje během celého trvání programu (statická proměnná) nebo existuje pouze tehdy, když se určitá funkce vykonává (automatická proměnná).

Volná paměť

Operátory `new` a `delete` však poskytují mnohem pružnější přístup. Řídí zásobu paměti, na kterou se C++ odvolává jako na volnou paměť. Tato volná paměť je oddělená od paměti používané pro statické a automatické proměnné. Jak ukazuje výpis programu 4.17, `new` a `delete` vám umožňují alokovat paměť v jedné funkci a uvolňovat v jiné. Tedy doba života dat se libovolně neváže na život programu nebo funkce. Společné použití `new` a `delete` vám dává mnohem více vlády nad tím, jak používáte paměť, než jak to umožňuje použití obvyklých proměnných.

Připomínka:

Ukazatele patří mezi nejmocnější prostředky C++. Jsou také nejnebezpečnější, protože povolují počítači nepřátelské akce, jako například použití neinicializovaného ukazatele na přístup do paměti nebo snahu uvolnit stejný blok paměti dvakrát. Kromě toho, dokud vám praxe neumožní, abyste si zvykli na značení a systém ukazatelů, ukazatele mohou být matoucí. Tato kniha se vrací k ukazatelům několikrát v naději, že každé osvícení vám umožní, abyste se s nimi cítili pohodlněji.

Shrnutí

Pole, struktury a ukazatele jsou tři odvozené typy v C++. Pole může obsahovat několik hodnot, všechny stejného typu, v jediném datovém objektu. Použitím ukazatele nebo indexu můžete přistupovat k jednotlivým prvkům pole.

Struktura může obsahovat v jednom datovém objektu několik hodnot různých typů a pro přístup k jednotlivým prvkům můžete použít operátor příslušnosti (.). Prvním krokem v použití struktur je vytvoření šablony struktury tím, že definujete, které členy struktura obsahuje. Jméno, neboli návěští, této šablony se potom stává novým identifikátorem typu. Potom můžete deklarovat proměnné tohoto typu.

Union může obsahovat jedinou hodnotu, ale ta může být různých typů, členské jméno určuje, který režim se má používat.

Ukazatele jsou proměnné navržené pro úschovu adres. Říkáme, že ukazatel ukazuje na adresu, kterou obsahuje. Deklarace ukazatele vždy stanoví, na který typ objektu ukazatel ukazuje. Použití dereferenčního operátoru (*) na ukazatel poskytuje hodnotu v lokaci, na kterou ukazatel ukazuje.

Řetězec je skupina znaků ukončená prázdným znakem. Řetězec se může reprezentovat řetězcovou konstantou uzavřenou do dvojitých uvozovek, v tomto případě se předpokládá prázdný znak samo sebou. Řetězec můžete uložit do pole typu `char` a můžete jej přestavit pomocí ukazatele na `char`, který se inicializuje tak, aby ukazoval na řetězec. Funkce `strlen()` vrací délku řetězce bez započítání prázdného znaku. Funkce `strcpy()` kopíruje řetězec z jedné lokace na jinou. Když používáte tyto funkce, zahrňte hlavičkový soubor `cstring` nebo `string.h`.

Operátor `new` vám umožní požádat paměť o datový objekt za běhu programu. Operátor navrací adresu paměti, kterou získá, a vy tuto adresu můžete přiřadit ukazateli. Jediným prostředkem pro přístup k této paměti je použití ukazatele. Je-li datový objekt jednoduchou proměnnou, můžete použít dereferenční operátor (*) na určení hodnoty. Je-li datovým objektem pole, můžete použít ukazatel na přístup k prvkům, jakoby to bylo jméno pole. Je-li datovým objektem struktura, můžete na přístup ke členům struktury použít ukazatelový dereferenční operátor (->).

Ukazatele a pole jsou těsně spojeny. Je-li `ar` jménem pole, potom se výraz `ar[i]` interpretuje jako `*(ar + i)` s tím, že jméno pole se interpretuje jako adresa prvního prvku pole. Tedy, jméno pole hraje stejnou roli jako ukazatel. Na oplátku můžete použít na přístup k prvkům v poli, které se alokuje pomocí `new`, jméno ukazatele s označením pole.

Operátory `new` a `delete` vám explicitně dovolují řídit, kdy se mají datové objekty alokovat a kdy se mají navracet do volné paměti. Automatické proměnné, to jsou ty, které jsou deklarovány ve funkci a statické proměnné, které jsou definovány mimo funkci nebo pomocí klíčového slova `statici`, jsou méně flexibilnější. Automatická proměnná oživne, když se vstoupí do bloku, který ji obsahuje (typicky definice funkce) a zaniká, když se blok opouští. Statická proměnná přetrvává po celou dobu trvání programu.

Opakovací otázky

1. Jak byste deklarovali každý z následujících bodů?

   a) `actors` je polem o 30 prvcích typu `char`.

   b) `betsie` je polem o 100 prvcích typu `short`.

   c) `chuck` je polem o 13 prvcích typu `float`.

   d) `dispea` je polem o 64 prvcích typu `long double`.

2. Deklarujte pole, které obsahuje 5 prvků typu `int`, inicializujte ho prvními pěti lichými celými čísly.
3. Napište příkaz, který přiřazuje součet prvého a posledního prvku pole v otázce 2 do proměnné `even`.
4. Napište příkaz, který zobrazuje hodnotu druhého prvku typu `float` pole `ideas`.
5. Deklarujte pole typu `char` a inicializujte ho řetězcem „`cheesburger`".
6. Navrhněte deklaraci struktury, která popisuje rybu. Struktura by měla zahrnovat druh (kind), váhu (weight) v celých uncích a délku (lenght) ve zlomcích palců.
7. Deklarujte proměnnou typu definovaného v otázce 6 a inicializujte ji.
8. Použijte klíčové slovo `enum` na definici typu, který se nazývá `Response` s možnými hodnotami `Ano`, `Ne` a `Snad`. `Ano` by mělo být 1, `Ne` 0 a `Snad` 2.
9. Předpokládejte, že `ted` je proměnnou typu `double`. Deklarujte ukazatel, který ukazuje na `ted` a použijte ho na zobrazení hodnoty `ted`.
10. Předpokládejte, že `treacle` je polem o 10 prvcích typu `float`. Deklarujte ukazatel, který ukazuje na první prvek pole `treacle` a použijte ukazatel na zobrazení jeho prvého a posledního prvku.
11. Napište část programového kódu, který požádá uživatele o zadání kladného celého čísla a vytvořte dynamické pole, které obsahuje tolik prvků typu `int`.
12. Je následující programový kód platný? Jestliže ano, co vytiskne?

    ```
    cout << (int *) "Home of the jolly bytes";
    ```

13. Napište část programového kódu, který dynamicky alokuje strukturu typu popsané v otázce 6 a přečtěte hodnotu členu `kind` (druh) této struktury.
14. Výpis programu 4.6 ilustruje problém řádkově orientovaného vstupu řetězce, který je následován číselným vstupem. Jak by nahrazení

    ```
    cin.getline(address,80);
    ```

    pomocí

```
cin >> address;
```

ovlivnilo práci programu?

Programovací cvičení

1. Napište program v C++, který požádá o informaci a zobrazí ji, jak je ukázáno níže. Všimněte si, že by měl být program schopný akceptovat rodná jména, která se skládají z více než jednoho slova. Také si všimněte, že program snižuje známku směrem dolů, to jest nahoru o jedno písmeno. Předpokládejte, že uživatel požádá o známky A, B nebo C, takže se nemusíte obávat o mezery mezi D a F.

```
What is your first name? Betty Sue
What is your last name? Yew
What letter grade do you deserve? B
What is your age? 22
Name: Yew, Betty Sue
Grade: C
Age: 22
```

2. Struktura CandyBar obsahuje tři členy. První člen obsahuje značku cukrové tyčinky. Druhý člen obsahuje váhu (která může mít desetinnou část) a třetí člen obsahuje počet kalorií (celé číslo) v cukrové tyčince. Napište program, který takovou strukturu deklaruje a vytváří proměnnou typu `CandyBar`, která se nazývá `snack`, inicializujte každý její člen postupně na „`Mocha Munch`", `2.3` a `350`. Inicializace by měla být částí deklarace `snack`.
3. Struktura `CandyBar` obsahuje tři členy, jak je popsáno v programovém cvičení 2. Napište program, který vytváří pole tří struktur `CandyBar`, inicializuje je hodnotami dle vašeho výběru, a potom zobrazte obsah každé struktury.
4. William Wingate poskytuje službu, která analyzuje pizzy. Pro každou pizzu musí zaznamenat následující informaci:

   - Jméno společnosti, která pizzu vyrábí, které může sestávat z více než jednoho slova
   - Průměr pizzy
   - Váhu pizzy

   Navrhněte strukturu, která bude obsahovat tuto informaci a napište program, jenž používá proměnnou tohoto typu. Program by měl požádat uživatele, aby zavedl každou z předchozích položek informace a pak by ji měl zobrazit. Použijte objekty `cin` (nebo jeho metody) a `cout`.
5. Proveďte programové cvičení 2, ale použijte na alokaci struktury `new` namísto deklarace strukturní proměnné. Také by měl program požádat o poloměr pizzy dříve než požádá o jméno společnosti, která pizzu vyrábí.

Cykly a relační výrazy

KAPITOLA

5

Témata kapitoly:

- Cyklus for
- Výrazy a příkazy
- Operátory inkrementování a dekrementování: ++ a —
- Kombinované přiřazovací operátory
- Složené příkazy (bloky)
- Operátor čárka
- Relační operátory: >, >=, ==, <=, < a !=
- Cyklus while
- Příkaz typedef
- Cyklus do while
- Metoda pro vstup znaku get()
- Podmínka konec_souboru
- Vložené cykly a dvourozměrná pole

Počítače provádějí více než ukládání dat. Analyzují, spojují, přeskupují, extrahují, mění, extrapolují, syntetizují a jiným způsobem manipulují s daty. Občas také data komolí a oklešťují, ale my se budeme snažit tento druh chování odstranit. Na provádění svých manipulačních zázraků programy potřebují nástroje na vykonávání opakovatelných činností a na učinění rozhodnutí. C++ samozřejmě takové nástroje poskytuje. Vskutku, používá stejné cykly `for`, `while`, `do while`, příkazy `if` a `switch`, které používá běžný C, takže pokud znáte C, můžete tuto a následující kapitoly prolétnout. (Ale neprolétněte je příliš rychle – nechcete minout, jak `cin` obsluhuje vstup znaků!) Tyto rozličné řídicí příkazy programu často používají na řízení svého chování relační a logické výrazy. Tato kapitola pojednává o cyklech a relačních operátorech a následující sleduje příkazy skoku a logické výrazy.

Úvod do cyklu for

Okolnosti často vyzývají program, aby prováděl opakovatelné úlohy, jako například sečtení prvků pole jeden po druhém nebo vytištění některých oslavných básní na produktivitu 20krát. Cyklus for v C++ provádí takové úlohy snadno. Podívejme se na cyklus ve výpisu programu 5.1, podívejme se, co dělá a potom diskutujme, jak pracuje.

Výpis programu 5.1 forloop.cpp

```
// forloop.cpp — představení cvyklu for
#include <iostream>
using namespace std;
int main()
```

```
{
    int i;  // vytvoří čítač
    //   initializace; test ; změna
    for (i = 0; i < 5; i++)
        cout << "C++ knows loops.\n";
    cout << "C++ knows when to stop.\n";
    return 0;
}
```

Zde je výstup:

```
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows when to stop.
```

Cyklus začíná nastavením celočíselného `i` na 0:

```
i = 0
```

Toto je *inicializační částí* cyklu. Potom v *testování cyklu* program testuje, zda je `i` menší než 5:

```
i < 5
```

Je-li tomu tak, program provede následující příkaz, který se nazývá *tělem cyklu*:

```
cout << "C++ knows loops.\n";
```

Potom cyklus použije *změnovou část* cyklu, aby zvýšil `i` o 1:

```
i++
```

Používá operátor ++, který se nazývá přírůstkovým operátorem. Inkrementuje hodnotu svého operandu o 1. (Přírůstkový operátor není omezen na cykly. Například můžete použít

```
i++;
```

místo

```
i = i + 1;
```

jako příkaz v programu.) Inkrementací `i` se dokončí první průběh cyklem.

Dále cyklus začíná další průběh porovnáním nové hodnoty `i` s hodnotou 5. Protože je nová hodnota (1) také menší než 5, cyklus vytiskne další řádek a potom končí opětnou inkrementací `i`. Tím se nastaví fáze na nové testování cyklu, provedení příkazu a změnu hodnoty `i`. Proces pokračuje, dokud cyklus nezmění `i` na 5. Potom další test selže a program se přesune na další příkaz za cyklem.

Části cyklu for

Cyklus `for` tedy poskytuje postupný předpis na provádění opakovaných činností. Podívejme se nyní detailněji, jak se to nastavuje. Obvyklé části cyklu `for` obsluhují tyto kroky:

- Nastavení hodnoty na začátku
- Provedení testu, aby se zjistilo, zda má cyklus pokračovat
- Provedení činností cyklu
- Změna použité hodnoty (hodnot) pro test

Návrh cyklu v C++ nastavuje tyto prvky tak, že je můžete hned spatřit. Inicializace, test a změna činností vytvářejí tři části řídící sekce, která je uzavřena do závorek. Každá část je výrazem a středníky tyto výrazy oddělují jeden od druhého. Příkaz, který následuje za řídící sekcí se nazývá ***tělem*** cyklu a provádí se tak dlouho, dokud testovací výraz zůstává pravdivým:

```
for (inicializace; testovací-výraz; změnový-výraz)
   tělo
```

Syntaxe v C++ považuje úplný příkaz `for` za jediný příkaz, dokonce i když může zahrnovat jeden nebo více příkazů v tělové části.

Inicializace cyklu se provádí pouze jednou. Prakticky používá program tohoto výrazu na nastavení proměnné na startovací hodnotu a potom proměnnou používá na počítání průchodů cyklem.

`Testovací-výraz` určuje, zda se dá tělo cyklu provést. Ve skutečnosti je tento výraz relačním výrazem, to jest výrazem, který porovnává dvě hodnoty. Náš příklad třeba porovnává hodnoty `i` a 5, zjišťuje, zda je `i` menší než 5. Je-li srovnání pravdivé, program vykoná tělo cyklu. Ve skutečnosti C++ neomezuje `testovací-výraz` na srovnání pravda nepravda. Můžete použít libovolný výraz. Když se výraz vyhodnotí jako nula, cyklus končí. Když se výraz vyhodnotí jako nenulový, cyklus pokračuje. Výpis programu 5.2 to ukazuje použitím výrazu `i` jako testovací podmínky. (Ve změnové sekci je `i--` podobné `i++` kromě toho, že snižuje hodnotu `i` o 1 pokaždé, když se použije.)

Výpis programu 5.2 num_test.cpp

```
// num_test.cpp - použijte numerický test v cyklu for
#include <iostream>
using namespace std;
int main()
{
    cout << "Enter the starting countdown value: ";
    int limit;
    cin >> limit;
    int i;
    for (i = limit; i; i--)      // konec, když je i rovno 0
        cout << "i = " << i << "\n";
```

```
    cout << "Done now that i = " << i << "\n";
    return 0;
}
```

Zde je výstup:

```
Enter the starting countdown value: 4
i = 4
i = 3
i = 2
i = 1
Done now that i = 0
```

Všimněte si, že cyklus končí, když `i` dosáhne 0.

Jak relační výrazy, jako například i < 5, zapadají do této konstrukce ukončení cyklu s hodnotou 0? Původně se relační výrazy vyhodnocovaly na 1, když byl výraz pravdivý a na 0, když byl nepravdivý. Tedy hodnota výrazu 3 < 5 byla 1 a hodnota 5 < 5 byla 0. Nyní C++ přidal typ `bool`, avšak relační výrazy se vyhodnocují na `booleovské` literály `true` a `false` namísto na 1 a 0. Tato změna nevedla k neslučitelnosti, avšak co se týče C++, program konvertuje `true` a `false` na 0 a 1, kde se očekávají celočíselné hodnoty a konvertuje 0 na `false` a nenulovou hodnotu na `true`, kde se očekávají `booleovské` hodnoty.

Cyklus `for` je cyklem se vstupní podmínkou. To znamená, že se výraz vyhodnotí *před* každým průchodem cyklem. Cyklus nikdy neprovede tělo cyklu, když je testovací výraz nepravda. Například předpokládejme, že se vrátíte do programu ve výpisu programu 5.2, ale nastavíte jako startovací hodnotu 0. Protože testovací podmínka selže pokaždé, když se vyhodnotí, tělo cyklu se nikdy neprovede:

Enter the starting countdown value: **0**

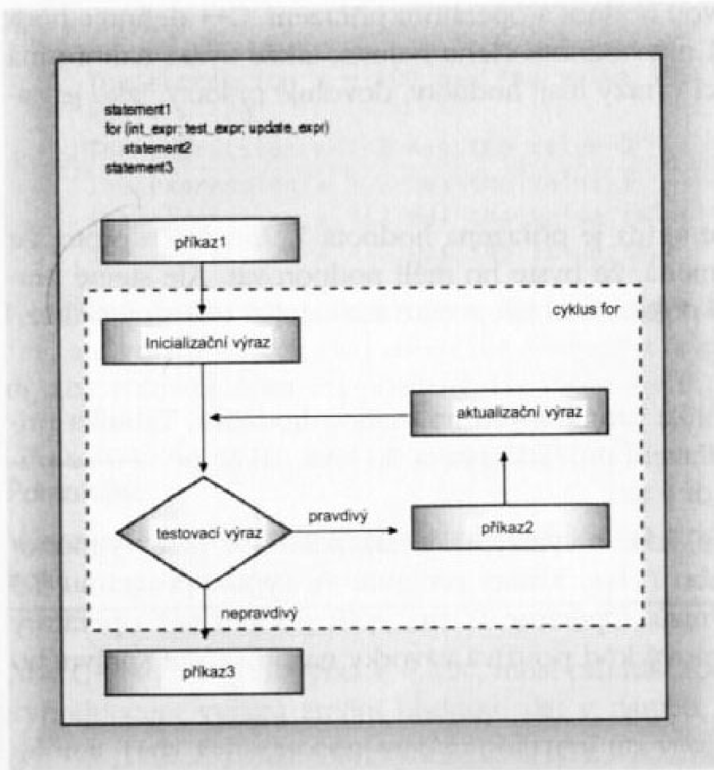
```
Done now that i = 0
```

Tento postoj, podívat se než začnete provádět cyklus, může pomoci zbavit program potíží.

`Změnový-výraz` se vyhodnotí na konci cyklu, až když se provedlo tělo. Prakticky se používá na zvýšení nebo snížení hodnoty proměnné, přičemž sleduje počet průchodů cyklem. Může však existovat libovolný platný výraz v C++, tak jako mohou existovat jiné řídící výrazy. To umožňuje, aby cyklus `for` byl mnohem schopnější vykonávat více práce, než pouhé počítání od 0 do 5, to je způsob, jak pracoval první příklad cyklu. Několik příkladů uvidíte později.

Tělo cyklu `for` sestává z jediného příkazu, ale brzy se naučíte, jak toto pravidlo rozšířit. obrázek 5.1 shrnuje návrh cyklu `for`.

Cyklus `for` vypadá téměř podobně jako volání funkce, protože používá jméno následované párem závorek. Avšak status `for` jako klíčového slova v C++ zabraňuje kompilátoru, aby přemýšlel o `for` jako o funkci. Vám také zabraňuje, abyste funkci pojmenovali `for`.



Obrázek 5.1 Cyklus for

Tip:

Běžným stylem v C++ je umístit mezeru mezi `for` a následující závorku a vynechat mezeru mezi jménem funkce a následující závorkou:

```
for (int i = 6; i < 10; i++)
        smart_function(i);
```

S jinými řídícími příkazy, jako jsou například `if` a `while`, se zachází podobně jako s `for`. To slouží k vizuálnímu posílení rozdílu mezi řídícím příkazem a voláním funkce. Také je běžnou praxí odsadit tělo od příkazu `for`, aby vystupovalo samostatně.

Výrazy a příkazy

Řídící sekce `for` používá tři výrazy. Spolu se svými dobrovolně přijatými vymezeními syntaxe je C++ velmi dokonalým jazykem. Libovolná hodnota nebo libovolná platná kombinace hodnot a operátorů vytváří výraz. Například výrazem je hodnota 10 (žádné překvapení) a 28 * 20 je výrazem s hodnotou 560. V C++ má každý výraz hodnotu. Často je hodnota zřejmá. Například výraz

```
22 + 27
```

se formuje ze dvou hodnot a operátoru sčítání a má hodnotu 49. Občas je hodnota méně zřejmá. Například

```
x = 20
```

je výrazem, protože je formován ze dvou hodnot a operátoru přiřazení. C++ definuje hodnotu přiřazovacího výrazu tak, že má mít hodnotu členu nalevo, takže výraz nahoře má hodnotu 20. Skutečnost, že přiřazovací výrazy mají hodnoty, dovoluje příkazy, jako je například následující:

```
maids = (cooks = 4) + 3;
```

Výraz `cooks = 4` má hodnotu 4, takže `maids` je přiřazena hodnota 7. Avšak jen proto, že C++ dovoluje takové chování, neznamená, že byste ho měli podporovat. Ale stejné pravidlo, které umožňuje tento nezvyklý příkaz, umožňuje také následující užitečný příkaz:

```
x = y = z = 0;
```

Toto je rychlý způsob nastavení několika proměnných na stejnou hodnotu. Tabulka priorit (Dodatek D) prozrazuje, že se přiřazení provádí zprava do leva, takže první 0 se přiřadí `z`, potom se hodnota `z = 0` přiřadí `y` atd.

Nakonec, jak bylo poznamenáno dříve, relační výrazy, jako například `x < y`, se vyhodnocují na `booleovskou` hodnotu `true` nebo `false`. Krátký program ve výpisu programu 5.3 ilustruje některé vlastnosti hodnot výrazů. Operátor << má vyšší prioritu než operátory používané ve výrazech, takže programový kód používá závorky na prosazení správného pořadí.

Výpis programu 5.3 express.cpp

```
// express.cpp — hodnoty výrazů
#include <iostream>
using namespace std;
int main()
{
    int x;

    cout << "The expression x = 100 has the value ";
    cout << (x = 100) << "\n";
    cout << "Now x = " << x << "\n";
    cout << "The expression x < 3 has the value ";
    cout << (x < 3) << "\n";
    cout << "The expression x > 3 has the value ";
    cout << (x > 3) << "\n";
    cout.setf(ios_base::boolalpha);
    cout << "The expression x < 3 has the value ";
    cout << (x < 3) << "\n";
    cout << "The expression x > 3 has the value ";
    cout << (x > 3) << "\n";
    return 0;
}
```

Kompatibilita:

Starší implementace C++ mohou jako parametru pro `cout.setf()` vyžadovat použití `ios::boolalpha` namísto `ios_base::boolalpha`. Ještě starší implementace možná nerozpoznají žádnou formu.

Zde je výstup:

```
The expression x = 100 has the value 100
Now x = 100
The expression x < 3 has the value 0
The expression x > 3 has the value 1
The expression x < 3 has the value false
The expression x > 3 has the value true
```

Normálně objekt `cout` konvertuje hodnoty typu `bool` před jejich zobrazením na typ `int`, ale volání funkce `cout.setf(ios_base::boolalpha)` nastaví příznak, který `cout` nařídí, aby zobrazil slova `true` nebo `false` místo 1 a 0.

Pamatujte:

Výraz v C++ je hodnotou nebo kombinací hodnot a operátorů a každý výraz v C++ má hodnotu.

Aby C++ vyhodnotil výraz `x = 100`, musí přiřadit hodnotu `100` do `x`. Když ta samá činnost vyhodnocení výrazu změní hodnotu dat v paměti, říkáme, že vyhodnocení má *vedlejší účinek*. Tedy vyhodnocení výrazu přiřazení má vedlejší účinek změny hodnoty prvku, do kterého se přiřazuje. O přiřazení můžete uvažovat jako o zamýšleném účinku, ale z hlediska toho, jak je C++ konstruován, vyhodnocení výrazu je primárním účinkem. Ne všechny výrazy mají vedlejší účinek. Například vyhodnocení výrazu `x + 15` vypočítá novou hodnotu, ale nemění hodnotu `x`. Avšak vyhodnocení `++x + 15` má vedlejší účinek, protože zahrnuje zvýšení `x`.

Od výrazu k příkazu je jen kousek; přidáme pouze středník. Tedy

```
age = 100
```

je výrazem, zatímco

```
age = 100;
```

je příkazem. Libovolný výraz se může stát příkazem, přidáte-li středník, ale výsledek možná nebude mít programovací smysl. Například, je-li `rodents` proměnnou, potom

```
rodents + 6;     // platný, ale nepoužitelný příkaz
```

je platným příkazem v C++. Kompilátor to povolí, ale příkaz neprovede nic užitečného. Program pouze spočítá součet, nic s ním neudělá a pokračuje dalším příkazem. (Chytrý kompilátor dokonce tento příkaz přeskočí.)

Nevýrazy a příkazy

Některé pojmy, jako například znát strukturu cyklu `for`, jsou pro porozumění C++ zásadní. Ale také existují relativně méně důležitá hlediska syntaxe, která vás náhle mohou zmást, právě když si myslíte, že rozumíte jazyku. Na pár z nich se nyní podíváme.

Ačkoli je pravdou, že přidáním středníku k libovolnému výrazu vytváříme příkaz, opak není pravdou. To jest, odstraněním středníku z příkazu ho nezbytně nekonvertujeme na

výraz. Z těch druhů příkazů, které jsme dosud použili, příkazy return, deklarační příkazy a příkazy `for` nevyhovují tvaru *příkaz = výraz + středník*. Například, ačkoli

```
int toad;
```

je příkazem, část `int toad` není výrazem a nemá hodnotu. To způsobí, že je následující programový kód neplatný:

```
eggs = int toad * 1000;     // neplatné, není výrazem
cin >> int toad;            // nemůžeme kombinovat deklaraci s objektem
cin
```

Podobně nemůžete přiřadit cyklus `for` proměnné:

```
int fx = for (int i = 0; i< 4; i++)
        cout >> i; // není možné
```

Tady cyklus `for` není výrazem, takže nemá hodnotu a vy ho nemůžete přiřadit.

Přizpůsobování pravidel

C++ přidává cyklům v C jisté vlastnosti, které vyžadují některé obratné úpravy syntaxe cyklu `for`. Toto byla původní syntaxe:

```
for (výraz; výraz; výraz)
  příkaz
```

Především, jak bylo dříve definováno, řídící část struktury `for` obsahovala tři výrazy oddělené středníkem. Cykly v C++ vám však dovolují provádět záležitosti, jako je následující:

```
for (int i = 0; i < 5; i++)
```

Například můžete deklarovat proměnnou v inicializační oblasti cyklu `for`. To může být vhodné, ale nevyhovuje to původní syntaxi, protože deklarace není výrazem. Toto nezákonné chování se původně přizpůsobilo definování nového druhu výrazu, *deklarační příkaz výrazu*, který byl holou deklarací středníku, a který jediný se mohl v příkazu `for` vyskytovat. Od této úpravy se však upustilo. Namísto toho byla syntaxe příkazu `for` pozměněna na následující:

```
for  (inicializační-příkaz-for podmínka; výraz)
  příkaz
```

Na první pohled to vypadá zvláštně, protože existuje pouze jediný středník místo dvou. Ale je to v pořádku, protože `inicializační-příkaz-for` je definován jako příkaz a příkaz má svůj vlastní středník. Co se týče `inicializační-příkaz-for`, je identifikován buď jako výrazový příkaz nebo deklarace. Toto syntaktické pravidlo nahrazuje výraz následovaný středníkem příkazem, který má svůj vlastní středník. Přínosem je to, že programátoři v C++ chtějí být schopni deklarovat a inicializovat proměnnou v inicializační části cyklu `for` a udělají možným vše, co je nezbytné pro syntaxi v C++ a anglickém jazyce. Existuje praktický důsledek pro deklaraci proměnné v `inicializační-příkaz-for`, o kterém byste měli vědět. Taková proměnná existuje pouze uvnitř příkazu `for`. To jest, jakmile program opustí cyklus, proměnná se odstraní:

```
for (int i = 0; i < 5; i++)
        cout << "C++ knows loops.\n";
```

```
cout << i << endl; // překvapení! i není dále definováno
```

Jiná věc, kterou byste měli vědět, je, že některé implementace v C++ dodržují dřívější pravidla a zacházejí s předcházejícím cyklem, jako by se i deklarovalo před cyklem, tedy zůstává dostupné po ukončení cyklu. Použití této nové volby v deklaraci proměnné při inicializaci cyklu for má za následek, alespoň nyní, odlišné chování na různých systémech.

Upozornění:

V době psaní této knihy se ne všechny kompilátory držely současného pravidla, že proměnná deklarovaná v řídící sekci cyklu for zaniká, když se cyklus ukončí.

Zpět do cyklu for

Buďme na cykly trochu náročnější. Výpis programu 5.4 používá cyklus na výpočet a uložení prvních 16 faktoriálů. Faktoriály, které jsou vhodné pro výpočet pravděpodobnosti, se počítají následujícím způsobem. Nulový faktoriál, psáno jako 0! je implicitně roven 1, potom 1! je 1 * 0!, neboli 1. Dále 2! je 2 * 1!, neboli 2. Potom 3! je 3 * 2!, neboli 6 atd., faktoriál každého celého čísla je součin tohoto celého čísla s předchozím faktoriálem. (Jeden z nejznámějších monologů pianisty Victora Borge charakterizuje vlastnosti hlasové interpunkce, ve které je vykřičník vyslovován něco jako phffft pptz, sentimentálním způsobem. Avšak v tomto případě se „!" vyslovuje „faktoriál".) Program používá jeden cyklus na výpočet následných hodnot faktoriálů a přitom je ukládá do pole. Potom používá druhý cyklus na uložení výsledků. Program také zavádí použití externích deklarací hodnot.

Výpis programu 5.4 formore.cpp

```
// formore.cpp - více cyklů for
#include <iostream>
using namespace std;
const int ArSize = 16;          // příklad externí deklarace
int main()
{
    double factorials[ArSize];
    factorials[1] = factorials[0] = 1.0;
    int i;
    for (i = 2; i < ArSize; i++)
        factorials[i] = i * factorials[i-1];
    for (i = 0; i < ArSize; i++)
        cout << i << "! = " << factorials[i] << "\n";
    return 0;
}
```

Zde je výstup:

```
0! = 1
1! = 1
```

```
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3.6288e+006
11! = 3.99168e+007
12! = 4.79002e+008
13! = 6.22702e+009
14! = 8.71783e+010
15! = 1.30767e+012
```

Faktoriály rostou rychle!

Poznámky k programu

Program vytváří pole na úschovu hodnot faktoriálů. Prvek 0 je 0!, prvek 1 je 1! atd. Protože první dva faktoriály jsou rovny 1, program nastaví prvé dva prvky v poli `factorials` na 1.0. (Pamatujte, že první prvek pole má hodnotu indexu 0.) Poté používá program cyklus na nastavení každého faktoriálu na součin indexu s předchozím faktoriálem. Cyklus ukazuje, že můžete použít čítač cyklu jako proměnné v těle cyklu.

Program ukazuje, jak cyklus `for` pracuje ruku v ruce s polem, poskytujíc vhodný prostředek pro přístup postupně ke každému členu pole. Tedy `formore.cpp` používá `const` na vytvoření symbolického znaku (`ArSize`) pro velikost pole. Potom používá `ArSize` kdekoli to přichází v úvahu, jako například při definici pole a pro meze cyklů, které pole obsluhují. Nyní, přejete-li si rozšířit program na řekněme 20 faktoriálů, musíte nastavit pouze `ArSize` na 20 a program překompilovat. Použitím symbolické konstanty se vyhnete tomu, že musíte změnit individuálně všechny výskyty čísla 16 na 20.

Tip:

Obvykle je dobrým nápadem definovat `konstantní` hodnotu na vyjádření počtu prvků v poli. Použijte `konstantní` hodnotu v deklaraci pole a všech dalších odkazech na velikost pole, jako například v cyklu `for`.

Výraz `limit i < ArSize` odráží tu skutečnost, že se index pole s prvky `ArSize` mění od jedné do `ArSize - 1`, takže by se index pole měl zastavit na hodnotě o 1 menší než `ArSize`. Mohli byste použít test `i <= ArSize - 1`, ale to vypadá v porovnání k předchozímu neohrabaně.

Jednou z vedlejších informací, kterou program poskytuje, je to, že deklaruje proměnnou `ArSize` typu `const int` vně těla funkce `main()`. Jak se na konci kapitoly 4, „Odvozené typy", uvádí, to vytváří z `ArSize` externí údaj. Dva důsledky tohoto způsobu deklarace jsou, že `ArSize` existuje během trvání programu a že ji mohou používat všechny funkce v programovém souboru. V tomto konkrétním případě má program pouze jednu funkci, takže

deklarování ArSize externě má malý praktický význam. Ale programy s více funkcemi často profitují ze sdílení externích konstant, takže si nyní procvičíme jejich používání.

Změna velikosti kroku

Dosud příklady o cyklu zvyšovaly nebo snižovaly čítač cyklu o 1 při každém průchodu. Můžete to ovlivnit změnou změnového výrazu. Program ve výpisu programu 5.5 zvyšuje například čítač cyklu o velikost kroku zvolenou uživatelem. Spíše než použití i++ jako změnového výrazu, používá výraz i = i + by, kde by je uživatelem zvolená velikost kroku.

Výpis programu 5.5 bigstep.cpp

```
// bigstep.cpp - načítání jak nařízeno
#include <iostream>
using namespace std;
int main()
{
    cout << "Enter an integer: ";
    int by;
    cin >> by;
    cout << "Counting by " << by << "s:\n";
    for (int i = 0; i < 100; i = i + by)
        cout << i << "\n";
    return 0;
}
```

Zde je ukázka běhu programu:

```
Enter an integer: 17
Counting by 17s:
0
17
34
51
68
85
```

Jakmile i dosáhne hodnoty 102, cyklus končí. Hlavním bodem je zde to, že změnový výraz může být libovolným platným výrazem. Například, chcete-li i umocnit na druhou a přidat 10 v každém průchodu, můžete použít i = i * i + 10.

Procházení řetězců pomocí cyklu for

Cyklus for poskytuje přímou cestu přístupu postupně ke každému znaku v řetězci. Výpis programu 5.6 vám například umožňuje zavést řetězec a potom ho znak po znaku v obráceném pořadí zobrazit. Strlen() poskytuje počet znaků v řetězci; cyklus používá tuto hodnotu ve svém inicializačním výrazu na nastavení i na index posledního znaku řetězce vyjma znaku null. Na zpětné počítání používá program operátor dekrement (—) na

snižování indexu pole o 1 při každém průchodu cyklem. Výpis programu také používá relační operátory větší než nebo rovno (>=), aby otestoval, zda cyklus dosáhl prvního prvku. Brzy všechny relační operátory shrneme.

Výpis programu 5.6 forstr1.cpp

```
// forstr1.cpp – použití for s řetězcem
#include <iostream>
#include <cstring>
using namespace std;
const int ArSize = 20;
int main()
{
    cout << "Enter a word: ";
    char word[ArSize];
    cin >> word;

    // zobrazuje písmena v opačném pořadí
    for (int i = strlen(word) - 1; i >= 0; i--)
        cout << word[i];
    cout << "\n";
    return 0;
}
```

Kompatibilita:

Pokud vaše implementace ještě nepřidala nový hlavičkový soubor, měli byste použít `string.h` namísto `cstring`.

Zde je ukázka běhu programu:

```
Enter a word: animal
lamina
```

Ano, program je při tisknutí slova `animal` v opačném pořadí úspěšný; použití `animal` jako testovacího slova mnohem jasněji ilustruje smysl tohoto programu, než kdybychom vybrali, řekněme, `redder` nebo `stats`.

Operátory inkrementování (++) a dekrementování (—)

C++ se věnuje několika operátorům, které se často používají v cyklech, takže se nyní na chvilku zastavme a vyzkoušejme je. Dva jste již viděli: operátor inkrementování (++), který ovlivnil jméno C++ a operátor dekrementování (—). Oba provádějí mimořádně časté operace cyklu: zvyšování a snižování čítače cyklu o 1. Avšak v jejich popisu toho existuje více, než jste dosud viděli. Každý operátor se vyskytuje ve dvou variantách. *Prefixová* verze se připojuje před operand, jako v ++x. *Postfixová* verze se připojuje za operand, ja-

ko v x++. Obě verze mají na operand stejný vliv, ale liší se v tom, kdy se používají. To je jako zaplatit za posečení trávníku předem nebo potom; obě metody mají stejný konečný vliv na vaši peněženku ale liší se v tom, kdy se peníze dodají. Výpis programu 5.7 tento rozdíl ukazuje pro operátor inkrementování.

Výpis programu 5.7 plus_one.cpp

```
// plus_one.cpp - oprátor inkrementování
#include <iostream>
using namespace std;
int main()
{
int a = 20;
int b = 20;

cout << "a   = " << a << ":   b = " << b << "\n";
cout << "a++ = " << a++ << ": ++b = " << ++b << "\n";
cout << "a   = " << a << ":   b = " << b << "\n";
return 0;
}
```

Zde je výstup:

```
a     = 20:   b = 20
a++   = 20: ++b = 21
a     = 21:   b = 21
```

Stručně řečeno, značení a++ znamená „použij okamžitou hodnotu a a vyhodnoť výraz, potom hodnotu a inkrementuj". Podobně značení ++b znamená „nejprve inkrementuj hodnotu b a potom použij novou hodnotu na vyhodnocení výrazu". Například máme následující vztah:

```
int x = 5;
int y = ++x;  // změňte x, potom přiřaďte do y
              // y je 6, x je 6

int z = 5;
int y = z++;  // přiřaďte do y, potom změňte z
              // y je 5, z je 6
```

Operátory inkrementování a dekrementování jsou zhuštěným zápisem, vhodným na obsluhování běžných úloh pro zvyšování a snižování hodnoty o 1. Můžete je použít s ukazateli stejně tak jako se základními proměnnými. Připomínáme, že přičtení 1 k ukazateli zvyšuje jeho hodnotu o počet bajtů typu, na který ukazuje. Stejné pravidlo platí i pro inkrementaci a dekrementaci ukazatelů.

Pamatujte:

Inkrementování a dekrementování ukazatelů dodržuje pravidla aritmetiky ukazatelů. Tedy, jestliže pt ukazuje na první člen pole, ++pt změní pt tak, že ukazuje na druhý člen.

Operátory inkrementování a dekrementování jsou trochu inteligentními operátory, ale nenechte se unést a neinkrementujte nebo nedekrementujte stejnou hodnotu více než jednou ve stejném příkazu. Problémem je, že pravidla použij-a-potom-změň a změň-a-potom-použij se mohou stát dvojsmyslnými. To jest, například příkaz jako

```
x = 2 * x++ * (3 - ++x); // nedělejte to
```

může vyprodukovat zcela odlišné výsledky na jiných systémech. C++ nedefinuje přesné chování pro tento druh příkazu.

Sdružování přiřazovacích operátorů

Výpis programu 5.5 používá na změnu čítače cyklu následující výraz:

```
i = i + by
```

C++ má sdružený operátor sčítání a přiřazení, který dosáhne stejného výsledku mnohem výstižněji:

```
i += by
```

Operátor += sečte hodnotu jeho dvou operandů a výsledek přiřadí operandu vlevo. To má za důsledek, že operátor vlevo musí být něčím, čemu můžete přiřadit hodnotu, jako například proměnná, prvek pole, člen struktury nebo údaj, který identifikujete dereferenčním ukazatelem:

```
int k = 5;
k += 3;                  // v pořádku, k se nastaví na 8
int *pa = new int[10];   // pa ukazuje na pa[0]
pa[4] = 12;
pa[4] += 6;              // v pořádku, pa[4] se nastaví na 18
*(pa + 4) += 7;          // v pořádku, pa[4] se nastaví na 25
pa += 2;                 // v pořádku, pa ukazuje na dříve uvedený pa[2]
34 += 10;                // zcela chybné
```

Každý aritmetický operátor má odpovídající přiřazovací operátor, jak se shrnuje v Tabulce 5.1. Každý operátor pracuje podobně jako +=. Tedy příkaz

```
k *= 10;
```

nahradí současnou hodnotu k hodnotou 10 krát větší.

Tabulka 5.1 Sdružené přiřazovací operátory

Operátor	Výsledek (L = levý operand, R = pravý operand)
+=	přiřadí L + R do L
-=	přiřadí L – R do L
*=	přiřadí L * R do L
/=	přiřadí L / R do L
%=	přiřadí L % R do L

Složené příkazy neboli bloky

Formát, neboli syntaxe, psaní příkazu for v C++ se vám může zdát omezující, protože tělo cyklu musí být jediným příkazem. To je nešikovné, jestliže chcete, aby tělo cyklu obsahovalo více příkazů. Naštěstí C++ poskytuje zadní vrátka, kterými můžete do těla cyklu vložit tolik příkazů, kolik je libo. Trik spočívá v použití páru složených závorek na sestrojení *složeného příkazu*, neboli *bloku*. Blok sestává z páru složených závorek a příkazů, které uzavírají, a z důvodů syntaxe se považuje za jediný příkaz. Například program ve výpisu programu 5.8 používá složených závorek na sloučení tří oddělených příkazů do jednoho bloku. To umožňuje tělu cyklu vyzvat uživatele, přečíst vstup a provést výpočet. Program počítá pohyblivý součet členů, které zavedete a poskytuje přirozenou příležitost pro použití operátoru +=.

Výpis programu 5.8 block.cpp

```
// block.cpp - použijte blokový příkaz
#include <iostream>
using namespace std;
int main()
{
    cout << "The Amazing Accounto will sum and average ";
    cout << "five numbers for you.\n";
    cout << "Please enter five values:\n";
    double number;
    double sum = 0.0;
    for (int i = 1; i <= 5; i++)
    {                                           // blok začíná zde
        cout << "Value " << i << ": ";
        cin >> number;
        sum += number;
    }                                           // blok zde končí
    cout << "Five exquisite choices indeed! ";
    cout << "They sum to " << sum << "\n";
    cout << "and average to " << sum / 5 << ".\n";
    cout << "The Amazing Accounto bids you adieu!\n";
    return 0;
}
```

Zde je vzorek běhu programu:

```
The Amazing Accounto will sum and average five numbers for you.

Please enter five values:
Value 1: 1942
Value 2: 1948
Value 3: 1957
Value 4: 1974
Value 5: 1980
Five exquisite choices indeed! They sum to 9801
and average to 1960.2.
The Amazing Accounto bids you adieu!
```

Předpokládejme, že ponecháte odsazení, ale vynecháte složené závorky:

```
for (int i = 1; i <= 5; i++)
        cout << "Value " << i << ": ";          // cyklus končí zde
        cin >> number;                           // po cyklu
        sum += number;
cout << "Five exquisite choices indeed! ";
```

Kompilátor ignoruje odsazení, takže v cyklu by byl pouze první příkaz. Tedy cyklus by vytiskl pět výzev a nic víc. Až by se cyklus ukončil, program by se přesunul na následující řádky a přečetl a sečetl by pouze jedno číslo.

Složené příkazy mají další zajímavou vlastnost. Definujete-li uvnitř bloku novou proměnnou, proměnná přetrvává pouze tak dlouho, dokud program provádí příkazy uvnitř bloku. Jakmile výpočet opouští blok, proměnná se dealokuje. To znamená, že je známa pouze uvnitř bloku:

```
#include <iostream>
using namespace std;
int main()
{
        int x = 20;
        {                           // blok začíná
                int y = 100;
                cout << x << "\n";       // v pořádku
                cout << y << "\n";       // v pořádku
        }                          // blok končí
        cout << x << "\n";   // v pořádku
        cout << y << "\n";            // neplatné, nezpracuje se
        return 0;
}
```

Operátor čárka (neboli více syntaktických triků)

Blok, jak jste viděli, vám umožňuje ukrýt jeden nebo více příkazů na místo, kde syntaxe C++ dovoluje pouze jediný příkaz. Operátor čárka provádí to samé pro výrazy, umožňuje vám ukrýt dva výrazy na místo, kde syntaxe C++ dovoluje pouze jeden. Například předpokládejme, že máte cyklus, ve kterém se jedna proměnná zvyšuje v každém průchodu o 1 a druhá proměnná se o 1 v každém průchodu snižuje. Bylo by příhodné provádět oba výrazy ve změnové části řídící sekce cyklu `for`, ale syntaxe cyklu vám tam umožňuje pouze jediný. Řešením je použití operátoru čárka ke spojení dvou výrazů do jednoho:

```
j++, i--  //pro syntaktické účely se dva výrazy počítají jako jeden
```

Čárka není vždy operátorem čárka. Například čárka v deklaraci

```
int i, j; // čárka je zde oddělovačem, nikoli operátorem
```

slouží k oddělení přilehlých jmen v seznamu proměnných.

Výpis programu 5.9 používá v programu, který otáčí obsah znakového pole, operátor čárka dvakrát. Všimněte si, že výpis programu 5.6 zobrazuje obsah pole v obráceném pořadí, ale výpis programu 5.9 přesouvá znaky v poli skutečně dokola. Program také používá pro seskupení několika příkazů do jednoho blok.

Výpis programu 5.9 forstr2.cpp

```
// forstr2.cpp - obrácení pole
#include <iostream>
#include <cstring>
using namespace std;
const int ArSize = 20;
int main()
{
    cout << "Enter a word: ";
    char word[ArSize];
    cin >> word;

    // fyzicky modifikuje pole
    char temp;
    int i, j;
    for (j = 0, i = strlen(word) - 1; j < i; i--, j++)
    {                              // začátek bloku
        temp = word[i];
        word[i] = word[j];
        word[j] = temp;
    }                              // konec bloku
    cout << word << "\n";
    return 0;
}
```

Zde je vzorek výstupu:

```
Enter a word: parts
strap
```

Poznámky k programu

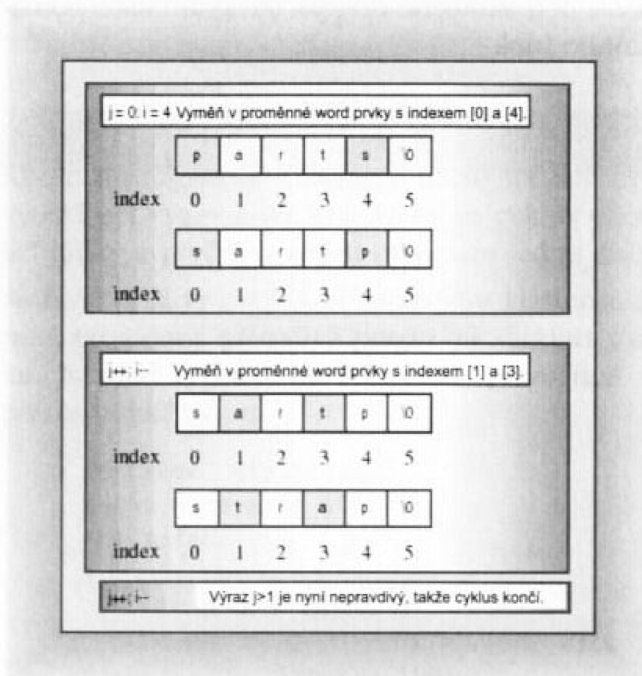
Podívejte se na řídící sekci `for`:

```
for (j = 0, i = strlen(word) - 1; j < i; i--, j++)
```

Za prvé, používá operátor čárka na vtěsnání dvou inicializací do jednoho výrazu do první části řídící sekce. Potom opět používá čárkový operátor na sloučení dvou řídících údajů do jediného výrazu do poslední části řídící sekce.

Dále se podívejte na tělo. Program používá složené závorky na sloučení několika příkazů do jediné jednotky. V těle program obrací pořadí ve slově záměnou prvého a posledního znaku pole. Potom inkrementuje `j` a dekrementuje `i`, takže se nyní odkazují na následující prvek za prvním prvkem a na předchozí prvek před posledním prvkem. Jakmile je to hotovo, program tyto prvky zamění. Všimněte si testovací podmínky `j < i`, která způso-

buje ukončení cyklu, když se dosáhne středu pole. Kdyby pokračoval přes tento bod, začal by zaměňovat přehozené prvky do jejich původní pozice. Viz obrázek 5.2.



Obrázek 5.2 Otočení řetězce

Další věc, která stojí za povšimnutí, je umístění deklarace proměnných `temp`, `i` a `j`. Programový kód deklaruje `i` a `j` před cyklem, protože dvě deklarace nemůžete spojovat operátorem čárka. To je proto, že deklarace vždy používá čárku pro jiný účel, oddělení položek v seznamu. Na vytvoření a inicializaci dvou proměnných můžete použít jediný výraz deklaračního příkazu, avšak to je vizuálně trochu matoucí:

```
int j = 0, i = strlen(word) - 1;
```

V tomto případě je čárka pouze oddělovačem v seznamu, nikoli operátorem čárka, takže výraz deklaruje a inicializuje jak `j` tak `i`. Ale vypadá to, jako by deklaroval pouze `j`. Mimochodem můžete deklarovat proměnnou `temp` uvnitř cyklu `for`:

```
int temp = word[i];
```

To má za následek, že se `temp` při každém průchodu cyklem alokuje a dealokuje. Může to být trochu pomalejší, než deklarování `temp` jednou před cyklem. Na druhé straně, pokud byla uvnitř cyklu, tak se odhodí, když cyklus skončí.

Lahůdky operátoru čárka

Zdaleka nejběžnějším použitím operátoru čárka je zabudování dvou nebo více výrazů do jediného výrazu cyklu `for`. Avšak C++ vybavuje operátor dvěma dodatečnými vlastnostmi. Za prvé zaručuje, že se první výraz vyhodnotí před druhým. Například následující výrazy jsou bezpečné:

```
i = 20, j = 2 * i // i nastavuje na 20, j nastavuje na 40
```

Za druhé C++ stanoví, že hodnotou výrazu čárka je hodnota druhé části. Hodnotou předchozího výrazu je například 40, protože je hodnotou `j = 2 * i`.

Operátor čárka má mezi všemi operátory nejnižší prioritu. Například příkaz

```
cats = 17, 240;
```

se čte jako

```
(cats = 17), 240;
```

To jest, `cats` se nastaví na 17 a 240 nic nedělá. Ale protože závorky mají vyšší prioritu,

```
cats = (17,240);
```

má za následek, že se `cats` nastaví na 240, hodnota výrazu napravo.

Relační výrazy

Počítače jsou více než neúprosnými pojídači čísel. Mají schopnost porovnávat hodnoty a tato schopnost je základem počítačového rozhodování. V C++ tuto schopnost ztělesňují relační operátory. C++ poskytuje na porovnání čísel šest relačních operátorů. Protože se znaky zachycují pomocí jejich ASCII-kódu, můžete na ně tyto operátory také použít, ale nepracují s řetězci ve stylu C. Každý relační výraz se redukuje na `booleovskou` hodnotu `true`, je-li porovnání pravdivé a na `false`, je-li nepravdivé, takže se velmi dobře hodí v testovacím výrazu cyklu. (Starší implementace vyhodnocují pravdivé relační výrazy na 1 a nepravdivé na 0.) Tabulka 5.2 tyto operátory shrnuje.

Těchto šest relačních operátorů vyčerpává porovnání, která vám C++ umožňuje provádět s čísly. Chcete-li porovnat dvě hodnoty, abyste viděli, která je hezčí nebo šťastnější, musíte se podívat někam jinam.

Tabulka 5.2 Relační operátory

Operátor	Význam
<	je menší než
<=	je menší než nebo rovno
==	je rovno
>	je větší
>=	je větší než nebo rovno
!=	není rovno

Zde je několik vzorových testů:

```
for (x = 20; x > 5; x--)         // pokračuj, dokud je x větší než 5
for (x = 1; y != x; x++)         // pokračuj, dokud není y rovno x
for (cin >> x; x == 0; cin >> x)) // pokračuj, dokud je x rovno 0
```

Relační operátory mají nižší prioritu než aritmetické. To znamená, že výraz

```
x + 3 > y - 2                    // výraz 1
```

odpovídá

```
(x + 3) > (y - 2)          // výraz 2
```

neodpovídá následujícímu:

```
x + (3 > y) - 2            // výraz 3
```

Protože výraz `(3 > y)` je buď 1 nebo 0 poté co se hodnota typu `bool` konvertuje na `int`, jsou oba výrazy 2 a 3 platné. Ale většina z nás by chtěla, aby výraz jedna znamenal výraz 2 a to právě C++ dělá.

Chyba, kterou možná děláte

Nespleťte si při testovací operátor je-rovno (==) s přiřazovacím operátorem (=). Výraz

```
musicians == 4 // porovnání
```

se zeptá na hudební otázku, je `musicians` rovno 4? Výraz má hodnotu `true` nebo `false`. Výraz

```
musicians = 4 // přiřazení
```

přiřadí hodnotu 4 do `musicians`. Celý výraz má v tomto případě hodnotu 4, protože je to hodnota na levé straně.

Pružný návrh cyklu `for` vytváří zajímavou příležitost pro vznik chyby. Jestliže nešťastnou náhodou odhodíte rovnítko (=) z operátoru == a použijete přiřazovací výraz místo relačního v testovací části cyklu `for`, stále ještě vyprodukujete platný programový kód. To je proto, že můžete použít jakýkoli platný výraz C++ pro testovací podmínku cyklu `for`. Pamatujte, nenulová hodnota oceňuje test jako `true` a nulová jako `false`. Výraz, který přiřazuje 4 do `musicians`, má hodnotu 4 a je považován za `true`. Přecházíte-li z jazyka, jako například Pascal nebo Basic, který používá = na testování rovnosti, pravděpodobně budete k tomuto uklouznutí náchylní.

Výpis programu 5.10 ukazuje situaci, ve které můžete tento druh chyby vytvořit. Program se pokouší prověřit pole výsledků testu a zastaví se, když dosáhne prvého výsledku, který není 20. Zobrazuje cyklus, který správně používá srovnání a potom druhý, který používá chybně přiřazení v testovací podmínce. Program má také jinou, do nebe volající chybu v návrhu, jejíž odstranění uvidíte později. (Učte se svými chybami, výpis programu 5.10 je šťastný, že v tomto ohledu pomůže.)

Výpis programu 5.10 equal.cpp

```
// equal.cpp - rovnost proti přiřazení
#include <iostream>
using namespace std;
int main()
{
    int quizscores[10] =
        { 20, 20, 20, 20, 20, 19, 20, 18, 20, 20};

    cout << "Doing it right:\n";
```

```
    int i;
    for (i = 0; quizscores[i] == 20; i++)
        cout << "quiz " << i << " is a 20\n";

    cout << "Doing it dangerously wrong:\n";
    for (i = 0; quizscores[i] = 20; i++)
        cout << "quiz " << i << " is a 20\n";

    return 0;
}
```

Protože tento program má závažný problém, měli byste dát přednost jeho prostudování před skutečným provedením. Zde je jedna ukázka výstupu:

```
Doing it right:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
Doing it dangerously wrong:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
quiz 5 is a 20
quiz 6 is a 20
quiz 7 is a 20
quiz 8 is a 20
quiz 9 is a 20
quiz 10 is a 20
quiz 11 is a 20
quiz 12 is a 20
quiz 13 is a 20
...
```

První cyklus se správně zastavuje po zobrazení prvních pěti výsledků testu. Ale druhý začíná zobrazením celého pole. Horší ovšem je, že tvrdí, že každá hodnota je 20. A ještě horší je, že se na konci pole nezastaví! Místo, kde se něco pokazí, je samozřejmě v následující testovací podmínce:

```
quizscores[i] = 20
```

Za prvé jednoduše proto, že přiřazuje nenulovou hodnotu prvku pole, je výraz vždy nenulový, odtud vždy pravdivý. Za druhé, protože výraz přiřazuje hodnoty prvkům pole, skutečně mění údaje. Za třetí, protože testovací výraz zůstává pravdivým, program pokračuje měnit údaje za konec pole. Pouze ukládá více a více 20 do paměti! To není dobré.

Obtížnost tohoto druhu chyby spočívá v tom, že je programový kód syntakticky správný, takže ho kompilátor neoznačí jako chybu. (Jenomže programátoři v C a C++ tuto chybu dělají roky a roky, což časem přimělo mnoho kompilátorů k tomu, aby vydávaly varování, které se ptá, zda to co děláte je skutečně tím, co míníte.)

Upozornění:

Nepoužívejte = na porovnání rovnosti; použijte ==.

Jak C, tak C++ vám poskytují více volnosti, než většina programovacích jazyků. Je to za cenu, vyžadující větší zodpovědnost na vaší straně. Nic než vaše vlastní dobré plánování zabrání programu v překročení hranice standardního pole v C++. Avšak pomocí tříd v C++ můžete navrhnout chráněný typ pole, který tomuto druhu nesmyslu zamezí. Kapitola 12, „Dědičnost tříd", poskytuje příklad. Zatím byste měli zabudovat ochranu do vašich programů, když ji potřebujete. Například cyklus by měl zahrnovat test, který by zabránil přejít za poslední člen. To je užitečné dokonce pro „dobrý" cyklus. Kdyby byly všechny výsledky 29, také by se překročily hranice pole. Zkrátka, cyklus potřebuje otestovat hodnoty pole a index pole. Kapitola 6 ukazuje, jak použít logické operátory na spojení dvou takových testů do jediné podmínky.

Porovnání řetězců

Předpokládejme, že chcete vidět, je-li řetězec ve znakovém poli roven slovu `mate`. Je-li `word` jménem pole, následující test by nemohl udělat to, co zamýšlíte:

```
word == "mate"
```

Pamatujte, že jméno pole je synonymem své adresy. Podobně je znaková konstanta v uvozovkách synonymem své adresy. Tedy předchozí relační výraz netestuje, zda jsou řetězce stejné, ale testuje, zda jsou uloženy na stejných adresách. Výsledkem testu je ne, dokonce i když oba řetězce mají stejné znaky.

Když se pokusíte použít relační operátory na srovnání řetězců, nezískáte uspokojivé výsledky, protože C++ zachází s řetězci jako s adresami. Místo toho můžete jít do řetězcové knihovny stylu C a na porovnání řetězců použít funkci `strcmp()`. Funkce má jako parametry dvě adresy řetězců. To znamená, že parametry mohou být ukazatele, řetězcové konstanty nebo jména znakových polí. Jsou-li oba řetězce stejné, funkce navrací nulovou hodnotu. Jestliže první řetězec předchází druhý alfabeticky, `strcmp()` navrací zápornou hodnotu a jestliže první řetězec následuje alfabeticky druhý, `strcmp()` navrací kladnou hodnotu. Ve skutečnosti „je srovnávané pořadí na systému" mnohem přesnější než „alfabetické". To znamená, že znaky jsou srovnávány podle znakového kódu systému. Například v kódu ASCII mají všechna velká písmena menší kód než malá písmena, takže ve srovnávaném pořadí velká písmena předcházejí malá. Proto řetězec „`Zoo`" předchází řetězec „`aviary`". To, že porovnání jsou závislá na hodnotě kódu, také znamená, že se velká a malá písmena liší, takže řetězec „`FOO`" je různý od řetězce „`foo`".

V některých jazycích, jako například Basic a standardní Pascal, se řetězce uložené v polích různé velikosti navzájem nutně nerovnají. Ale řetězce stylu C jsou definovány ukončovacím znakem null, nikoli velikostí pole, ve kterém jsou uloženy. To znamená, že dva řetězce mohou být identické, dokonce i když jsou uloženy v různě velkých polích:

```
char big[80] = "Daffy";         // 5 pismen plus \0
char little[6] = "Daffy";       // 5 pismen plus \0
```

Mimochodem, ačkoli nemůžete používat relační operátory na porovnání řetězců, můžete je použít na porovnání znaků, protože znaky jsou ve skutečnosti celočíselnými typy. Tak

```
for (ch = 'a'; ch <= 'z'; ch++)
        cout << ch;
```

je platným programovým kódem, alespoň pro znakovou sadu ASCII, pro zobrazení znaků abecedy.

Výpis programu 5.11 používá `strcmp()` v testovací podmínce cyklu `for`. Program zobrazuje slovo, mění jeho první písmeno, zobrazuje ho znovu a pokračuje, dokud `strcmp()` neurčí, že je slovo stejné jako řetězec „`mate`". Všimněte si, že výpis programu zahrnuje soubor `cstring`, protože poskytuje funkční prototyp pro `strcmp()`.

Výpis programu 5.11 compstr.cpp

```
// compstr.cpp - porovnání řetězců
#include <iostream>
#include <cstring>      // prototyp pro strcmp()
using namespace std;
int main()
{
    char word[5] = "?ate";

    for (char ch = 'a'; strcmp(word, "mate"); ch++)
    {
        cout << word << "\n";
        word[0] = ch;
    }
    cout << "After loop ends, word is " << word << "\n";
    return 0;
}
```

Kompatibilita:

Možná musíte použít `string.h` místo `cstring`. Programový kód také předpokládá, že systém používá znakovou kódovou sadu ASCII. V této sadě jsou kódy pro písmena a až z za sebou a kód pro znak ? bezprostředně předchází kódu pro a.

Zde je výstup:

```
?ate
aate
bate
cate
date
eate
fate
gate
hate
```

```
iate
jate
kate
late
After loop ends, word is mate
```

Poznámky k programu

Program má několik zajímavých vlastností. Jednou z nich je test. Chceme, aby cyklus pokračoval tak dlouho, dokud `word` neznamená `mate`. To jest, chceme, aby test pokračoval tak dlouho, dokud `strcmp()` neřekne, že jsou oba řetězce stejné. Nejočividnějším testem pro to je:

```
strcmp(word, "mate") != 0    // řetězce nejsou stejné
```

Tento výraz má hodnotu 1 (true), jestliže se řetězce nerovnají a 0 (false), jestliže se rovnají. Ale co `strcmp(word, "mate")` sama o sobě? Má nenulovou hodnotu (true), když se řetězce nerovnají a hodnotu 0 (false), když se rovnají. V podstatě funkce vrací true, jestliže se řetězce nerovnají a false, jestliže jsou stejné. Místo celého relačního výrazu můžete použít pouze funkci. To produkuje stejné chování a obsahuje méně zápisu. Také je to způsob, jak programátoři v C a v C++ tradičně používali `strcmp()`.

Pamatujte

Pro testování rovnosti a pořadí řetězců použijte funkci `strcmp()`. Výraz

```
strcmp(str1, str2) == 0
```

je pravdivý, jestliže jsou řetězce `str1` a `str2` totožné. Výrazy

```
strcmp(str1, str2) != 0
```

a

```
strcmp(str1, str2)
```

jsou pravdivé, jestliže řetězce `str1` a `str2` totožné nejsou. Výraz

```
strcmp(str1, str2) < 0
```

je pravdivý, jestliže řetězec `str1` předchází řetězec `str2` a výraz

```
strcmp(str1, str2) > 0
```

je pravdivý, jestliže řetězec `str1` za řetězecem `str2` následuje. Funkce `strcmp()` tedy může hrát roli operátorů ==, !=, < a > podle toho, jak testovací podmínku vytvoříte.

Dále `compstr.cpp` používá operátor inkrement pro pochod proměnné `ch` abecedou:

```
ch++
```

Operátory inkrementování a dektrementování můžete použít se znakovými proměnnými, protože typ `char` je opravdu celočíselným typem, takže operace skutečně mění celočíselný kód obsažený v proměnné. Také si všimněte, že použití indexu pole zjednodušuje změnu jednotlivých znaků v řetězci:

```
word[0] = ch;
```

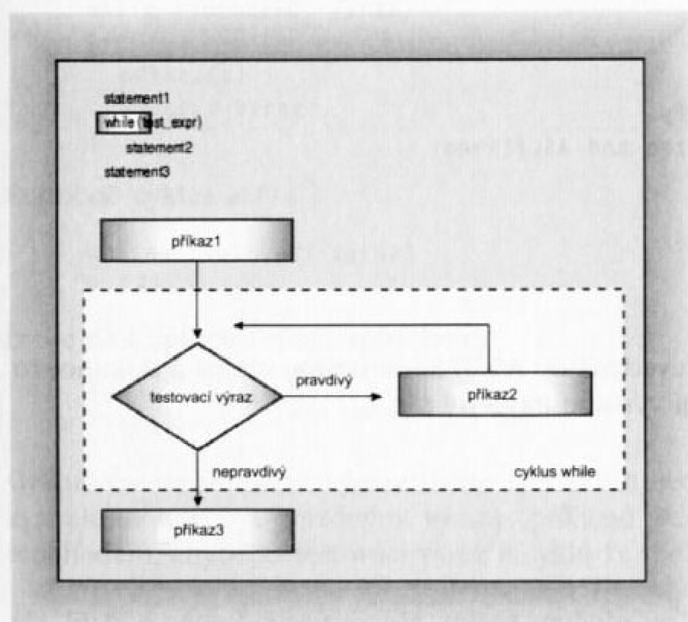
Konečně, na rozdíl od většiny dosud uvedených cyklů `for`, tento cyklus není čítacím cyklem. To jest, neprovádí blok příkazů blíže určeným počtem kroků. Místo toho cyklus čía na určité okolnosti (`word` se stane „`mate`"), aby signalizoval, že je čas zastavit. Častěji používají programy pro tento druh testů cyklů `while`, takže tento tvar nyní vyzkoušíme.

Cyklus while

Cyklus `while` je cyklus `for` zbavený inicializační a změnové části; má pouze testovací podmínku a tělo.

```
while (testovaci-podminka)
  tělo
```

Za prvé program vyhodnocuje výraz `testovaci-podminka`. Jestliže se výraz vyhodnotí na `true`, program vykoná příkaz(y) v těle. Podobně jako u cyklu `for`, tělo sestává z jediného příkazu nebo bloku, definovaného pomocí složených závorek. Jakmile ukončí tělo, program se vrátí na testovací podmínku a vyhodnotí ji. Je-li podmínka nenulová, program vykoná tělo znovu. Cyklus testování a vykonávání pokračuje, dokud se testovací podmínka nevyhodnotí jako `false`. Viz obrázek 5.3. Zřejmě, chcete-li, aby se cyklus konečně ukončil, cosi v těle cyklu musí něco provádět, aby se ovlivnil výraz testovací-podmínka. Například cyklus může inkrementovat proměnnou, která je použita v testovací podmínce nebo přečíst novou hodnotu ze vstupu z klávesnice. Podobně jako cyklus `for`, je cyklus `while` cyklem se vstupní podmínkou. Tedy jestliže se testovací-podmínka vyhodnotí na začátku jako `false`, program nikdy neprovede tělo cyklu.



Obrázek 5.3 Cyklus while

Výpis programu 5.12 nařizuje cyklu while, aby pracoval. Cyklus běží přes každý znak řetězce a zobrazuje znak a jeho ASCII-kód. Cyklus končí, když dosáhne prázdného znaku. Tento postup krokování řetězce znak po znaku, dokud se nedosáhne prázdného znaku, je standardní metodou v C++ pro zpracování řetězců. Protože řetězce obsahují svoji ukončovací značku, programy většinou nepotřebují explicitní informaci o tom, jak je řetězec dlouhý.

Výpis programu 5.12 while.cpp

```
// while.cpp - představení cyklu while
#include <iostream>
using namespace std;
const int ArSize = 20;
int main()
{
    char name[ArSize];

    cout << "Your first name, please: ";
    cin >> name;
    cout << "Here is your name, verticalized and ASCIIized:\n";
    int i = 0;                  // start na začátku řetězce
    while (name[i] != '\0')     // zpracování do konce řetězce
    {
        cout << name[i] << ": " << int(name[i]) << '\n';
        i++;                    // tento krok nezapoměňte
    }
    return 0;

}
```

Zde je vzorek běhu programu:

```
Your first name, please: Muffy
Here is your name, verticalized and ASCIIized:
M: 77
u: 117
f: 102
f: 102
y: 121
```

(Ne, vertikálně uspořádaná slova a převedená do ASCII kódu nejsou reálná a dokonce to jsou dobrá pseudo slova. Jen dodávají výstupu moderní tón.)

Poznámky k programu

Podmínka while vypadá takto:

```
while (name[i] != '\0')
```

Testuje, zda je určitý znak pole roven prázdnému znaku. Aby se test nakonec podařil, tělo cyklu musí změnit hodnotu i. Dělá to inkrementací i na konci těla. Vynechání tohoto kroku by přilepilo cyklus na stejný prvek pole a tisklo by stejný znak a jeho kód, dokud by se nepodařilo cyklus vypnout. Takový nekonečný cyklus je jedním z nejběžnějších

problémů cyklů. Často to můžete zapříčinit, když zapomenete měnit nějakou hodnotu uvnitř těla cyklu.

Řádek `while` můžete přepsat tímto způsobem:

```
while (name[i])
```

S touto změnou program pracuje stejně jako předtím. To je proto, že `name[i]` je řádným znakem, jehož hodnotou je kód znaku, který je nenulový, neboli true. Ale je-li `name[i]` prázdným znakem, hodnota kódu znaku je 0, neboli false. Tento zápis je mnohem zhuštěnější, ale méně zřejmý, než ten, který jsme použili. Hloupé kompilátory možná vytvářejí rychlejší kód pro druhou verzi, ale chytré kompilátory budou produkovat stejný kód pro obě.

Abyste přiměli program tisknout ASCII-kód znaku, používá pro konverzi `name[i]` na celočíselný typ přetypování. Potom objekt `cout` tiskne hodnotu jako celé číslo, spíše než by ji interpretoval jako znakový kód.

`for` oproti `while`

V C++ jsou cykly `for` a `while` v podstatě ekvivalentní. Například cyklus `for`

```
for (inicializační-výraz; testovací-výraz; změnový-výraz)
{
   příkaz(y)
}
```

by se mohl přepsat tímto způsobem:

```
inicializační-výraz;
while (testovací-výraz)
{
   příkaz(y)
   změnový-výraz;
}
```

Podobně cyklus `while`

```
while (testovací-výraz)
   tělo
```

by se mohl přepsat tímto způsobem:

```
for ( ; testovací-výraz;)
   tělo
```

Cyklus `for` vyžaduje tři výrazy (nebo techničtěji, jeden příkaz následovaný dvěma výrazy), ale mohou být prázdnými výrazy (příkazy). Pouze dva středníky jsou povinné. Mimochodem, chybějící testovací výraz v cyklu `for` se chápe jako true, takže cyklus

```
for ( ; ; )
     tělo
```

nikdy nekončí.

Protože cykly while a for jsou téměř ekvivalentní, je záležitostí stylu, který z nich použijete. (Existuje jemný rozdíl, jestliže tělo obsahuje příkaz continue, o kterém se pojednává v kapitole 6, „Příkazy větvení a logické operátory".) Většinou programátoři používají cyklu for pro čítací cykly, protože formát cyklu for umožňuje umístit veškerou důležitou informaci, počáteční hodnotu, konečnou hodnotu a metodu změny čítače do jednoho místa. Mnohem častěji používáte cyklus while, když předem přesně nevíte, kolikrát se bude cyklus provádět.

Když navrhujete cyklus, měli byste mít na paměti následující metodické pokyny:

1. Identifikujte podmínku, která ukončuje cyklus.
2. Inicializujte podmínku před prvním testem.
3. Změňte podmínku v každém průchodu cyklem předtím, než ji znovu otestujete.

Jednou z pěkných věcí, týkající se cyklu for, je, že jeho struktura poskytuje důvod na implementaci těchto tří metodických pokynů, tudíž vám pomáhá, abyste si vzpomněli to tak udělat.

Chybné rozdělení

Oba cykly for a while mají těla, sestávající z jediného výrazu, za kterými jsou příkazy v závorkách. Jak jste viděli, jediný příkaz může být blokem, který může obsahovat několik příkazů. Pamatujte si, že blok definují závorky, nikoli odsazení. Uvažujme například následující cyklus:

```
i = 0;
while (name[i] != '\0')
        cout << name[i] << "\n";
        i++;
cout << "Done\n";
```

Odsazení nám říká, že autor programu zamýšlel, že příkaz i++; je částí těla cyklu. Nepřítomnost složených závorek však říká kompilátoru, že tělo obsahuje pouze první příkaz cout. Proto program trvale tiskne první znak pole. Program nikdy nedosáhne na příkaz i++;, protože je mimo cyklus.

Následující příklad ukazuje další léčku:

```
i = 0;
while (name[i] != '\0');     // problém středníku
{
        cout << name[i] << "\n";
        i++;
}
cout << "Done\n";
```

Tentokrát dostal program složené závorky v pořádku, ale zahrnul také středník navíc. Pamatujte, středník ukončuje příkaz, takže tento středník ukončuje cyklus while. Jinými slovy, tělo cyklu je prázdný příkaz, tj. po středníku nic nenásleduje. Vše podstatné ve složených závorkách nyní přichází po cyklu. Nikdy toho není dosaženo. Místo toho cyklus stále probíhá a nic nedělá. Dejte si pozor na zbloudilý středník.

Čekání

Občas je užitečné do programu zabudovat časové zpoždění. Možná jste například narazili na programy, které vysvítí na obrazovku zprávu a pokračují něčím jiným, než ji můžete přečíst. Jste ponecháni v obavě, že jste propásli nenahraditelnou informaci životní důležitosti. Bylo by mnohem milejší, kdyby se program před pokračováním na pět vteřin zastavil. Cyklus `while` je vhodný pro vytvoření tohoto efektu. Jedním z dřívějších postupů bylo vytvořit na chvíli počítačový čítač, který by spotřebovával čas:

```
long wait = 0;
while (wait < 10000)
    wait++;                // počítá potichu
```

Problém tohoto přístupu spočívá v tom, že musíte změnit čítací mez, změníte-li rychlost procesoru počítače. Různé hry, napsané například pro původní IBM PC, přestaly být rychle ovladatelné, když běžely na jeho rychlejších následovnících. Lepším přístupem je nechat systémové hodiny, aby za vás prováděly časování.

Knihovny ANSI C a C++ mají funkci, která vám to pomůže realizovat. Funkce se nazývá `clock()` a navrací systémový čas, který uplynul od spuštění programu. Existuje několik komplikací. Za prvé, `clock()` nevrací nezbytně čas v sekundách. Za druhé, funkční návratový typ může být na některých systémech `long`, na jiných `unsigned long` nebo snad ještě jiný.

Avšak hlavičkový soubor `ctime` (`time.h` na starších implementacích) poskytuje na tyto problémy řešení. Za prvé, definuje symbolickou konstantu `CLOCKS_PER_SEC`, která se rovná počtu systémových časových jednotek za sekundu. Takže dělení systémového času touto hodnotou poskytne sekundy. Nebo můžete sekundy násobit `CLOCKS_PER_SEC` na získání času v systémových jednotkách. Za druhé, `ctime` stanovuje `clock_t` jako alias pro návratový typ `clock()`. (Viz poznámku o typech alias.) To znamená, že můžete deklarovat proměnnou typu `clock_t` a kompilátor ji potom konvertuje na `long` nebo `unsigned int` nebo na cokoli, co je pro váš systém řádným typem.

Kompatibilita:

Systémy, které nedodávají hlavičkový soubor `ctime`, mohou místo toho použít `time.h`. Některé implementace mohou mít problémy s `waiting.cpp`, jestliže knihovna implementačních komponent není plně kompatibilní s ANSI-C. To je proto, že funkce clock() je dodatkem ANSI do tradiční knihovny C. Také některé nezralé implementace ANSI C používaly `CLK_TCK` nebo `TCK_CLK` namísto delšího `CLOCKS_PER_SEC`. Některé starší verze g++ nerozeznávají z těchto definovaných konstant žádnou. Některá prostředí (jako například MSVC++ 1.0, ale ne MSVC++ 5.0) mají potíže se znakem alarm \a a sladěním displeje s časovým zpožděním.

Výpis programu ukazuje, jak se používají `clock()` a hlavička `ctime` na vytvoření zpožďovací smyčky.

Výpis programu 5.13 waiting.cpp

```
// waiting.cpp — použití clock() ve zpožďovací smyčce
#include <iostream>
#include <ctime> // popisuje funkci clock(), typ clock_t
using namespace std;
int main()
{
    cout << "Enter the delay time, in seconds: ";
    float secs; cin >> secs;
    clock_t delay = secs * CLOCKS_PER_SEC; // konvertuje na hodinové takty
    cout << "starting\a\n";
    clock_t start = clock();
    while (clock() - start < delay )        // počkejte, dokud nevyprší čas
        ;                                   // všimněte si středníku
    cout << "done \a\n";
    return 0;
}
```

Výpočtem času zpoždění v systémových jednotkách namísto v sekundách se program vyhýbá povinnosti konvertovat systémový čas na sekundy během každého cyklu.

Alias typy

C++ má dva způsoby stanovení nového jména jako druhého jména typu. Jeden spočívá v použití preprocesoru:

```
#define BYTE char // preprocesor nahrazuje BYTE pomocí char
```

Když kompilujete program, preprocesor potom nahrazuje všechny výskyty `BYTE` pomocí `char`, tedy vytváří pro `char` druhé jméno.

Druhým způsobem na vytvoření alias-jména je v C++ (a C) použití klíčového slova `typedef`. Například na vytvoření alias-jména `byte` pro `char`, uděláte toto:

```
typedef char byte;   // dělá z byte alias-jméno pro char
```

Zde je obecný tvar:

```
typedef jméno_typu alias_jméno;
```

Jinými slovy, chcete-li, aby alias_jméno bylo druhým jménem pro určitý typ, deklarujte `alias_jméno` jako by to byla proměnná tohoto typu a potom před deklaraci napište klíčové slovo `typedef`. Například, abyste vytvořili alias `byte_pointer` pro ukazatel na `char`, deklarujte `byte_pointer` jako ukazatel na `char` a potom dopředu přilepte `typedef`:

```
typedef char * byte_pointer; // ukazatel na typ char
```

Mohli byste zkusit něco podobného pomocí `#define`, ale nebude to fungovat, když použijete seznam proměnných. Například uvažujme následující:

```
#define FLOAT_POINTER float *
FLOAT_POINTER pa, pb;
```

Substituce preprocesoru konvertuje deklaraci takto:

```
float * pa, pb;    // pa je ukazatel na float, pb pouze float
```

Přístup pomocí `typedef` tento problém nemá.

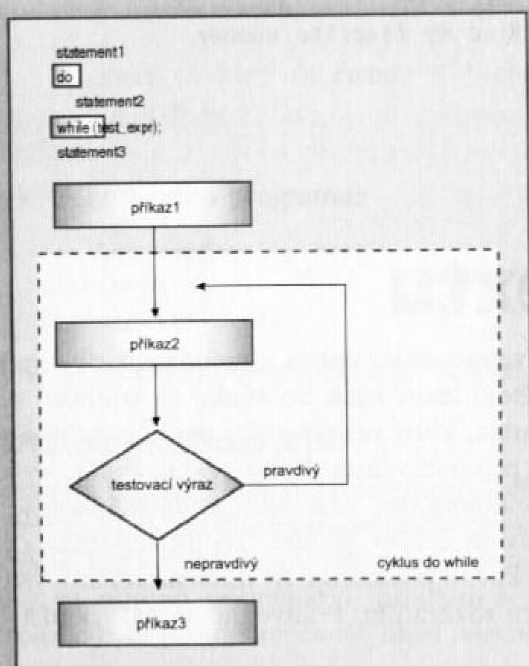
Všimněte si, že `typedef` nevytváří nový typ. Vytváří pouze nové jméno tohoto typu. Vytvoříte-li `word` druhým jménem pro `int`, `cout` zachází s hodnotou typu `word` jako s `int`, kterou skutečně je.

Cyklus do while

Nyní jste viděli cykly `for` a `while`. Třetím cyklem v C++ je `do while`. Liší se od obou předchozích, protože je cyklem s výstupní podmínkou. To znamená, že bezstarostný cyklus nejprve provede tělo cyklu a teprve potom vyhodnotí testovací výraz, aby viděl, zda by měl pokračovat dalším průběhem. Když se podmínka vyhodnotí na `false`, cyklus končí; jinak začíná nový cyklus prováděním a testováním. Takový cyklus se vždy provede alespoň jednou, protože jeho programový tok musí projít tělem cyklu, než dosáhne testu. Zde je syntaxe:

```
do
   tělo
while (testovací_výraz);
```

Část tělo může být jediným příkazem nebo blokem příkazů ohraničeným složenými závorkami. Obrázek 5.4 sumarizuje tok programu cyklem `do while`.



Obrázek 5.4 Cyklus do while

Obvykle je cyklus se vstupní podmínkou lepší volbou než cyklus s výstupní podmínkou, protože se podmínka ověří před průběhem cyklu. Například předpokládejme, že výpis programu 5.12 používal `do while` namísto `while`. Potom by měl cyklus nejprve vytisknout prázdný znak a jeho kód předtím, než by zjistil, že již dosáhl konce řetězce. Ale občas má test `do while` smysl. Například, žádáte-li uživatele o vstup, program ho musí před otestováním získat. Výpis programu 5.14 ukazuje, jak používat `do while` v této situaci.

Výpis programu 5.14 dowhile.cpp

```
// dowhile.cpp - výstupní podmínka cyklu
#include <iostream>
using namespace std;
int main()
{
    int n;

    cout << "Enter numbers in the range 1-10 to find ";
    cout << "my favorite number\n";
    do
    {
        cin >> n;          // provádí tělo
    } while (n != 7);     // potom testuje
    cout << "Yes, 7 is my favorite.\n" ;
    return 0;
}
```

Zde je ukázka běhu programu:

```
Enter numbers in the range 1-10 to find my favorite number
9
4
7
Yes, 7 is my favorite.
```

Cykly a vstup textu

Nyní, když jste viděli, jak cykly pracují, podívejme se na jednu z nejběžnějších a nejdůležitějších úloh, která se vztahuje k cyklům: čtení textu znak po znaku ze souboru nebo klávesnice. Například byste chtěli napsat program, který počítá počet znaků, řádků a slov na vstupu. Tradičně, jak C++ tak C, používají pro tento druh úlohy cyklus `while`. Vyšetříme nyní, jak se to udělá. Znáte-li již C, nepřelétněte tuto část příliš rychle. Ačkoli je cyklus `while` v C++ stejný jako v C, V/V vybavení v C++ je jiné. To může dodat cyklu v C++ poněkud odlišný vzhled. Vskutku, objekt `cin` podporuje tři rozličné režimy vstupu po jednom znaku, každý s jiným uživatelským rozhraním. Podívejme se, jak použít tyto volby s cykly `while`.

Použití nepřizdobeného cin pro vstup

Chystá-li se program použít cyklus na čtení vstupu z klávesnice, musí znát několik způsobů, kdy zastavit. Jak ví, kdy má zastavit? Jedním způsobem je výběr určitého znaku, jenž se často nazývá zarážkou, která slouží jako signál zastav. Například výpis programu 5.15 zastaví čtení vstupu, když program narazí na znak #. Program počítá počet znaků, které čte, a opakuje je. To jest, znovu zobrazuje znaky, které byly přečteny. (Stisknutí klávesy na klávesnici automaticky neumisťuje znaky na obrazovku; programy musí dělat otrockou práci tím, že provádějí odezvu na vstupní znaky.) Když se ukončí, vydá zprávu o počtu zpracovaných znaků. Výpis programu 5.15 ukazuje program.

Výpis programu 5.15 textin1.cpp

```
// textin1.cpp — čtení znaků cyklem while
#include <iostream>
using namespace std;
int main()
{
    char ch;
    int count = 0;          // použije základní vstup

    cin >> ch;              // získá znak
    while (ch != '#')       // test znaku
    {
        cout << ch;         // zobrazení znaku
        count++;            // přičtení znaku
        cin >> ch;          // získá další znak
    }
    cout << "\n" << count << " characters read\n";
    return 0;
}
```

Zde je ukázka běhu programu:

```
see ken run#really fast
seekenrun
9 characters read
```

Zřejmě Ken běží tak rychle, že sám vymaže místo nebo alespoň prostor znaků na vstupu.

Poznámky k programu

Nejprve si všimněte struktury. Program, dříve než dojde na začátek cyklu, přečte první vstupní znak. Způsob, kterým se může první znak otestovat, když program dosáhne příkazu cyklu. Je to důležité, protože prvním znakem může být #. Protože `textin1.cpp` používá cyklus se vstupní podmínkou, program správně v tomto případě přeskočí celý cyklus. A protože byla proměnná `count` dříve nastavena na nulu, `count` má správnou hodnotu.

Předpokládejme, že prvním přečteným znakem není #. Potom program vstoupí do cyklu, zobrazí znak, inkrementuje čítač a přečte další znak. Poslední krok je důležitý. Bez něj by cyklus stále zpracovával první vstupní znak. Pomocí něho program pokračuje dalším znakem.

Všimněte si, že program dodržuje dříve uvedené pokyny. Podmínka, která ukončuje cyklus je, když posledním přečteným znakem je #. Podmínka se inicializuje přečtením znaku dříve, než cyklus začne. Podmínka se mění čtením dalšího znaku na konci cyklu.

Vše to zní rozumně. Tak proč program vynechává na výstupu mezery? Příčinou je objekt `cin`. Když čte hodnoty typu `char`, stejně tak jako při čtení dalších základních typů, přeskakuje mezery a znaky nového řádku. Mezery na vstupu nejsou potvrzovány, a tak se nepočítají.

Aby se věci dále komplikovaly, vstup do objektu `cin` se ukládá do vyrovnávací paměti. To znamená, že se znaky, které zadáváte, neposílají do programu, dokud nestisknete ENTER. Proto jsme byli schopni psát znaky po #. Jakmile jsme stiskli ENTER, celá skupina znaků se poslala do programu, ale program ukončil zpracování vstupu poté, co dosáhl znaku #.

`cin.get(char)` vede k vysvobození

Programy, které obvykle čtou znak po znaku, potřebují každý znak včetně mezer, tabulátorů a znaku nového řádku vyzkoušet. Třída `istream` (definovaná v `iostream`), do které `cin` patří, zahrnuje členské funkce, které vyhovují těmto potřebám. Zvláště členská funkce `cin.get(ch)` čte následující znak, dokonce i když je mezera, do proměnné `ch`. Nahrazením `cin>>ch` tímto voláním funkce, můžete výpis programu 5.15 upravit. Výpis programu 5.16 ukazuje výsledek.

Výpis programu 5.16 textin2.cpp

```
//textin2.cpp
#include <iostream>
using namespace std;
int main()
{
    char ch;
    int count = 0;

    cin.get(ch);          // použijte funkci cin.get(ch)
    while (ch != '#')
    {
        cout << ch;
        count++;
        cin.get(ch);      // použijte ji znovu
    }
    cout << "\n" << count << " characters read\n";
    return 0;
}
```

Zde je ukázka běhu programu:

```
Did you use a #2 pencil?
Did you use a
14 characters read
```

Nyní program odpovídá a počítá všechny znaky včetně mezer. Vstup se stále ukládá do vyrovnávací paměti, takže je stále možné zadávat více vstupních dat, než na která program eventuálně dosáhne.

Znáte-li C, program vám může připadat nefunkčním. Volání `cin.get(ch)` umisťuje hodnotu do proměnné `ch`, to znamená, že mění hodnotu proměnné. V C musíte funkci předat adresu proměnné, chcete-li její hodnotu změnit. Avšak volání `cin.get()` ve výpisu programu 5.16 předává `ch`, nikoli `&ch`. V C podobný programový kód nepracuje. V C++ je to možné za předpokladu, že funkce deklaruje tento parametr jako *odkaz*. To je nový odvozený typ v C++. Hlavičkový soubor `iostream` deklaruje parametr pro `cin.get()` jako typ *odkaz*, takže funkce může měnit hodnotu svého parametru. K detailům se dostaneme v kapitole 8, „Příběhy ve funkcích". Mezitím mohou odborníci na C mezi vámi odpočívat, předávání parametrů v C++ pracuje obvykle stejně jako v C. Avšak pro `cin.get(ch)` to neplatí.

Kterou `cin.get()`?

Kapitola 4 používá tento programový kód:

```
char name[ArSize];
...
cout << "Enter your name:\n";
cin.get(name, ArSize).get();
```

Poslední řádek je ekvivalentní dvěma funkčním voláním za sebou:

```
cin.get(name, ArSize).get();
cin.get();
```

Jedna verze `cin.get()` má dva parametry: jméno pole, které je adresou řetězce (technicky typu `char *`), a `ArSize`, který je celým číslem typu `int`. (Připomeňme, že jméno pole je adresou jeho prvního prvku, takže jméno znakového pole je typu `char *`.) Program potom používá `cin.get()` bez parametrů. A zcela nedávno jsme použili `cin.get()` tímto způsobem:

```
char ch;
cin.get(ch);
```

Tentokrát nemá `cin.get()` parametr a je typu char.

Ještě jednou nastává vhodná chvíle pro ty, kteří jsou obeznámeni s C, aby se rozčílili nebo spletli. V C, má-li funkce parametry ukazatel na `char` a `int`, nemůžete úspěšně použít tutéž funkci s jediným parametrem jiného typu. Ale v C++ to můžete udělat, protože jazyk podporuje rysy OOP, které se nazývají *přetížení funkcí*. Přetížení funkcí vám dovoluje vytvořit různé funkce, které mají stejné jméno, za předpokladu, že mají jiný seznam parametrů. Jestliže například v C++ můžete použít `cin.get(name, ArSize)`, kompilátor nalezne verzi `cin.get()`, která používá jako parametry `char *` a `int`. A jestliže programový kód neposkytne žádné parametry, kompilátor použije verzi `cin.get()`, která nemá žádné parametry. Přetížení funkcí vám umožní použít stejné jméno pro příbuzné funkce, které provádějí stejný základní úkol různými způsoby nebo pro různé typy. To je další téma, které na vás čeká v kapitole 8. Mezitím si na přetížení funkcí můžete zvyknout, když použijete

příklady, které přicházejí se třídou `istream`. Abychom odlišili různé verze funkcí, zahrneme seznam parametrů, když se na ně odkazujeme. Tedy `cin.get()` znamená verzi bez parametrů a `cin.get(ch)` verzi, která má jeden parametr.

Podmínka konce souboru

Jak ukazuje výpis programu 5.16, použití symbolu, jako například #, na signalizaci konce vstupu není vždy uspokojivé, protože takový symbol může být částí skutečného vstupu. To stejné platí o jiných libovolně zvolených symbolech, jako například @ nebo %. Je-li vstup ze souboru, můžete použít mnohem výkonnější postup – rozpoznání konce souboru (end-of-file – EOF). Prostředky vstupu v C++ spolupracují s operačním systémem, aby rozpoznaly, kdy vstup dosáhl konce souboru a oznamují tuto informaci zpět programu.

Na první pohled vypadá čtení informace ze souborů, že má málo společného s objektem `cin` a vstupem z klávesnice, ale existují dvě spojení. Za prvé, mnoho operačních systémů, včetně Unixu a MS-DOSu, podporuje *přesměrování*, které vám umožní nahradit soubor vstupem z klávesnice. Například předpokládejme, že v MS-DOSu máme proveditelný program, který se nazývá `gofish.exe` a textový soubor se nazývá `fishtale`. Potom můžete dodat za prompt Dosu tento příkazový řádek:

```
gofish < fishtale
```

To přiměje program, aby vzal vstup ze souboru `fishtale` místo z klávesnice. Symbol < je operátorem přesměrování pro oba, Unix a Dos. Mnoho operačních systémů vám dovoluje simulovat podmínku konce souboru z klávesnice. Na Unixu to učiníte stisknutím `CTRL-D` na začátku řádku. V Dosu stisknete kdekoli na řádku `CTRL-Z, ENTER`. Některé implementace podporují podobné chování, dokonce i když podpůrný operační systém ne. Koncept konce souboru pro vstup z klávesnice je skutečně zákonným prostředím příkazového řádku. Avšak Symantec C++ pro Mac imituje Unix a rozpoznává `CTRL-D` jako simulovaný `EOF`. Microsoft Visual C++ 5.0 a Borland C++ Builder, prostředí Windows, podporují režim terminálu, ve kterém `CTRL-Z` pracuje bez `ENTER`. Avšak mechanismus je zde trochu odlišný. Režim terminálu zachytí CTRL-Z předtím, než na něj dosáhne program a ukončí další V/V interakci, takže tento postup má v tomto prostředí omezenou užitečnost. Kompilátory Metrowerks neposkytují simulovaný `EOF`.

Může-li váš program testovat konec souboru, můžete ho použít s přesměrovaným souborem a můžete ho použít pro vstup z klávesnice, na které simulujete konec souboru. Zní to užitečně, tak se podíváme, jak se to dělá.

Když objekt `cin` detekuje konec souboru (`EOF`), nastavuje bit (`eofbit`) na 1. Můžete použít členskou funkci `eof()`, abyste viděli, zda byl bit nastaven; volání `cin.eof()` vrací `booleovskou` hodnotu `true`, když se detekoval `EOF` a jinak `false`. Všimněte si, že metoda `eof()` oznamuje výsledek většiny současných pokusů čtení; to jest podává zprávu o minulosti spíše než, že se dívá dopředu. Takže test `cin.eof()` by měl vždy následovat po pokusu o čtení. Návrh výpisu programu 5.17 tuto skutečnost odráží.

Kompatibilita:

Některé systémy, včetně Metrowerks CodeWarrior, nepodporují simulovaný EOF z klávesnice. Jiné systémy, včetně Microsoft Visual C++ 5.0 a Borland C++ Builder, ho podporují nedokonale.

Výpis programu 5.17 textin3.cpp

```
// textin3.cpp - čtení znaků do konce souboru
#include <iostream>
using namespace std;
int main()
{
    char ch;
    int count = 0;
    cin.get(ch);            // pokus přečíst znak
    while (cin.eof() == false)  // test na EOF
    {
        cout << ch;         // zobrazení znaku
        count++;
        cin.get(ch);        // pokus o čtení dalšího znaku
    }
    cout << count << " characters read\n";
    return 0;
}
```

Zde je ukázka výstupu. Protože jsme vykonali program pod systémem Windows 95, abychom simulovali podmínku konce souboru, stiskli jsme CTRL-Z. Uživatelé Dosu by měli místo toho stisknout CTRL-Z, ENTER. Uživatelé Unixu a Symantec C++ pro Mac by místo toho měli stisknout CTRL-D.

```
The green bird sings in the winter. Enter
The green bird sings in the winter.
Yes, but the crow flies in the dawn. Enter
Yes, but the crow flies in the dawn.
Ctrl-Z
73 characters read
```

Použitím přesměrování můžete využít tento program na zobrazení textového souboru a vydat zprávu, kolik znaků obsahuje. Tentokrát máme program, který četl, zobrazoval a spočítal dvouřádkový soubor na systému Unix ($ je unixová výzva):

```
$ textin3 < stuff
I am a UNIX file. I am proud
to be a UNIX file.
49 characters read
$
```

Konec souboru končí vstup

Když metoda objektu cin detekuje konec souboru, připomínáme, že nastaví příznak v objektu cin, který indikuje podmínku konce souboru. Když je tento příznak nastavený, cin nečte žádný další vstup a další volání cin nemá žádný účinek. Pro vstup souboru to má smysl, protože byste neměli číst za konec souboru. Avšak pro vstup z klávesnice byste na ukončení cyklu mohli použít simulovaný konec souboru, ale potom byste chtěli číst další vstupní údaje později. Metoda cin.clear() vymaže příznak konce souboru a nechá vstup opět pokračovat. Kapitola 16, „Vstup, výstup a soubory", o tom dále pojednává. Pamatujte si však, že v režimu emulace terminálu ve Windows 95 stisknutí CTRL-Z efektivně ukončuje vstup i výstup, i přes schopnost metody cin.clear() je obnovit.

Běžná spojení

Základním návrhem vstupního cyklu je:

```
cin.get(ch);                        // pokus o přečtení znaku
while (cin.eof() == false)          // test na EOF
{
    ...                             // zpracování příkazů
    cin.get(ch);                    // pokus o přečtení dalšího znaku
}
```

Existuje několik zkrácených operací, které můžete udělat. Kapitola 6 představuje operátor !, který zaměňuje true na false a naopak. Můžete ho použít na přepsání testu while, aby vypadal takto:

```
while (!cin.eof())    // když není eof
```

Návratovou hodnotou metody cin.get(char) je cin, objekt. Avšak třída istream poskytuje funkci, která konvertuje objekt istream jako například cin na hodnotu bool; tato konverzní funkce se volá, když se cin vyskytuje v místě, kde se očekává bool, jako například v testovací podmínce cyklu while. Dále, booleovská hodnota pro konverzi je true, když byl poslední pokus o čtení úspěšný a v opačném případě false. To znamená, že můžete přepsat test while, aby vypadal takto:

```
while (cin)    // pokud je vstup úspěšný
```

Je to trochu obecnější než použití !cin.eof(), protože to detekuje další možné příčiny selhání, jako je například selhání disku.

Nakonec, protože návratovou hodnotou cin.get(char) je objekt cin, můžete cyklus stěsnat do tohoto formátu:

```
while (cin.get(ch))    // když je vstup úspěšný
{
        ...            // zpracování příkazů
}
```

Aby program vyhodnotil test cyklu, musí nejprve provést vyvolání cin.get(ch), které, je-li úspěšné, umístí hodnotu do ch. Potom program získá návratovou hodnotu z funkčního volání, která je cin. Potom aplikuje na cin booleovskou konverzi, která poskytne true, jestliže vstup pracoval, v opačném případě false. Tři metodické pokyny (identifi-

kace ukončovací podmínky, inicializace a změna podmínky) jsou všechny vtěsnány do jedné testovací podmínky.

Ještě další cin.get()

Nostalgičtější uživatelé C mezi vámi by mohli toužit po znakových V/V funkcích getchar() a puchar(). Jsou stále dostupné, pokud je chcete. Pouze použijete hlavičkového souboru stdio.h, když byste chtěli do C (nebo použijte novější cstdio). Nebo můžete použít členské funkce ze tříd istream a ostream, které pracují téměř stejným způsobem. Na tento přístup se nyní podíváme.

Kompatibilita:

Některé starší implementace nepodporují členskou funkci cin.get() (žádné parametry), o které se zde pojednává.

Členská funkce cin.get(), která nemá žádné parametry, vrací následující znak ze vstupu. To jest, můžete použít tento způsob:

```
ch = cin.get();
```

(Připomínáme, že členská funkce cin.get(ch) navrací objekt, nikoli přečtený znak.) Tato funkce pracuje téměř stejným způsobem jako funkce getchar() v C, navrací znakový kód jako hodnotu typu int. Podobně můžete na zobrazení znaku použít funkci cout.put() (viz kapitola 3, „Práce s daty"):

```
cout.put(ch());
```

Pracuje téměř jako funkce putchar() v C, kromě toho, že její parametr by měl být typu char místo int.

Kompatibilita:

Původně měl člen put() jediný prototyp put(char). Mohli jste mu předat parametr typu char, který potom mohl být přetypován na int. Návrh standardu také vyžaduje jediný prototyp. Avšak mnoho současných implementací poskytuje tři prototypy: put(char), put(signed char) a put(unsigned char). Použití put() s parametrem typu int v těchto aplikacích může generovat chybové hlášení, protože existuje více než jediná volba na konverzi int. Explicitní přetypování, jako je například cin.put(char(c)), pracuje s typy int.

Abyste použili cin.get() úspěšně, potřebujete vědět, jak zachází s podmínkou konce souboru. Když funkce dosáhne konce souboru, neexistuje více znaků, které by se měly vrátit. Místo toho cin.get() vrací speciální hodnotu pomocí symbolické konstanty EOF. Tato konstanta se definuje v hlavičkovém souboru iostream. Hodnota EOF se musí lišit od libovolného znaku, takže by program neměl poplést EOF s regulérním znakem. Prakticky se EOF definuje jako hodnota -1, protože žádný znak nemá ASCII kód této hodnoty, ale vy nemusíte skutečnou hodnotu znát. Pouze použijte EOF v programu. Například srdce výpisu programu 5.15 vypadá takto:

```
    cin >> ch;
    while (ch != '#')
    {
        cout << ch;
        count++;
        cin >> ch;
    }
```

Objekt `cin` můžete nahradit pomocí `cin.get()`, objekt `cout` pomocí `cout.put()` a '#' pomocí `EOF`:

```
    ch = cin.get();
    while (ch != EOF)
    {
        cout.put(ch);   // cout.put(char(ch)) for some implementations
        count++;
        ch = cin.get();
    }
```

Je-li `ch` znakem, tak ho cyklus zobrazí. Je-li `ch` `EOF`, cyklus se ukončí.

Tip:

Měli byste si uvědomit, že `EOF` nereprezentuje znak na vstupu. Spíše je signálem, že tam již nejsou další znaky.

Kromě změn, které se dosud udělaly, existuje důvtipný, ale důležitý moment o použití `cin.get()`. Protože EOF reprezentuje hodnotu vně platných znakových kódů, je pravděpodobné, že možná nebude kompatibilní s typem `char`. Například na některých systémech je typ `char` unsigned, takže proměnná `char` by nikdy nemohla mít obvyklé `EOF` o hodnotě -1. Z tohoto důvodu, používáte-li `cin.get()` (žádný parametr) a testujete-li `EOF`, musíte přiřadit návratovou hodnotu do typu `int` místo typu `char`. Také vytvoříte-li proměnnou `ch` typu `int` místo typu `char`, možná budete muset přetypovat na `char`, když zobrazujete `ch`.

Výpis programu 5.18 včleňuje postup `cin.get()` do nové verze výpisu programu 5.15. Také zhušťuje programový kód spojením znakového vstupu s testem cyklu `while`.

Výpis programu 5.18 textin4.cpp

```
// textin4.cpp — čtení znaků pomocí cin.get()
#include <iostream.h>
using namespace std;
int main(void)
{
    int ch;                         // mělo by být int, ne char
    int count = 0;

    while ((ch = cin.get()) != EOF) // test na konec souboru
    {
```

```
        cout.put(char(ch));
        count++;
    }
    cout << count << " characters read\n";
    return 0;
}
```

Kompatibilita:

Některé systémy buď nepodporují simulovaný EOF nebo ho podporují nedokonale.

Zde je ukázka běhu programu:

```
The sullen mackerel sulks in the shadowy shallows.
The sullen mackerel sulks in the shadowy shallows.
Yes, but the blue bird of happiness harbors secrets.
Yes, but the blue bird of happiness harbors secrets.
^Z
104 characters read
```

Analyzujme podmínku cyklu:

```
while ((ch = cin.get()) != EOF)
```

Závorky, které uzavírají podvýraz `ch = cin.get()`, nutí program, aby vyhodnotil tento výraz první. Aby provedl vyhodnocení, program nejprve volá funkci `cin.get()`. Potom přiřazuje její návratovou hodnotu do `ch`. Protože hodnota přiřazovacího výrazu je hodnotou levého operandu, celý podvýraz se redukuje na hodnotu `ch`. Je-li tato hodnota EOF, cyklus končí; jinak pokračuje. Testovací podmínka potřebuje všechny závorky. Předpokládejme, že některé vynecháme:

```
while (ch = cin.get() != EOF)
```

Operátor != má vyšší prioritu než =, takže program nejprve porovná návratovou hodnotu `cin.get()` s `EOF`. Porovnání vyprodukuje pravdivý nebo nepravdivý výsledek; `bool`eovská hodnota se konvertuje na 0 nebo 1 a to je hodnota, která se přiřadí do `ch`.

Použití `cin.get(ch)` (s parametrem) na vstup na druhé straně nevytváří žádné problémy typu. Funkce `cin.get(char)`, připomínáme, nepřiřazuje `ch` zvláštní hodnotu na konci souboru. Ve skutečnosti se v tomto případě nic do `ch` nepřiřazuje. Proměnná `ch` se nikdy nevyzve, aby uchovávala hodnotu, která není znakem. Tabulka 5.3 shrnuje rozdíly mezi `cin.get(char)` a `cin.get()`.

Tak kterou metodu byste měli použít, `cin.get()` nebo `cin.get(char)`? Tvar se znakovým parametrem je plněji začleněn do objektového přístupu, protože jeho návratovou hodnotou je objekt `istream`. To například znamená, že můžete zřetězit jeho použití. Například následující prostředky kódu čtou jeden vstupní znak do `ch1` a další vstupní znak do `ch2`:

```
cin.get(ch1).get(ch2);
```

Toto pracuje, protože volání funkce `cin.get(ch1)` vrací objekt `cin`, který potom funguje jako objekt, ke kterému je připojena `get(ch2)`.

Pravděpodobně hlavním použitím tvaru `get()` je nechat vás provádět rychlé a špinavé konverze (quick-and-dirty) z funkcí `getchar()` a `putchar()` ze `stdio.h` na metody `cin.get()` a `cout.put()` z `iostream`. Pouze záměnou jednoho hlavičkového souboru jiným a globální náhradou `getchar()` a `putchar()` ekvivalenty, které provádějí stejnou činnost. (Používá-li starý programový kód proměnnou typu `int`, musíte udělat další přizpůsobení, jestliže vaše implementace má násobné prototypy pro funkci `put()`.

Tabulka 5.3 `cin.get(ch)` **oproti** `cin.get()`

Vlastnost	cin.get(ch)	cin.get()
Metoda pro zprostředkování znakového vstupu	Přiřazení do parametru `ch`	Použijte návratové hodnoty funkce pro přiřazení do `ch`
Návratová hodnota funkce pro znakový vstup	Objekt třídy `istream` (`true` po `booleovské` konverzi)	Kód znaku jako hodnota typu `int`
Návratová hodnota funkce na konci souboru	Objekt třídy `istream` (`false` po `booleovské` konverzi)	EOF

Vnořené cykly a dvourozměrná pole

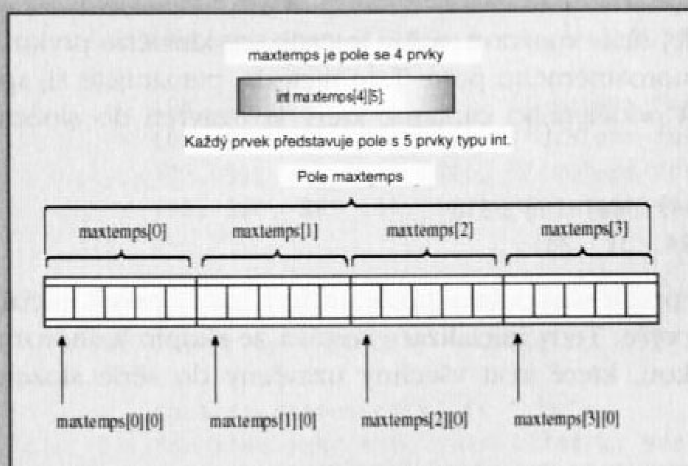
Dříve jste viděli, jakým přirozeným prostředkem je cyklus `for` pro zpracování polí. Pojďme o krok dál a podívejme se, jak cyklus `for` s cyklem `for` (vnořené cykly) slouží na zacházení s dvoudimenzionálními poli.

Nejprve vyšetříme, co je dvourozměrným polem. Pole dosud používaná se nazývají jednorozměrná, protože si můžeme představit každé pole jako jediný řádek dat. Dvourozměrné pole si můžete představit spíše jako tabulku, která má jak řádky, tak sloupce dat. Dvourozměrné pole můžete použít například na zobrazení kvartálních částek za prodej pro šest různých obvodů s jedním řádkem pro každý. Nebo můžete použít dvoudimenzionální pole na zobrazení umístění RoboDork na hrací desku vybavenou počítačem.

C++ neposkytuje zvláštní typ dvourozměrného pole. Místo toho vytváříte pole, jehož každým prvkem je samo pole. Například předpokládejme, že chcete uložit data o maximálních teplotách pěti velkých měst během čtyřletého období. V tomto případě můžete deklarovat pole následovně:

```
int maxtemps[4][5];
```

Deklarace znamená, že `maxtemps` je polem o čtyřech prvcích. Každý z těchto prvků je polem o pěti celých číslech. Viz obrázek 5.5. Pole `maxtemps` můžete považovat za reprezentaci čtyř řádků o pěti teplotních hodnotách v každém.



Obrázek 5.5 Pole polí

Výraz `maxtemps[0]` je prvním prvkem pole `maxtemps`; odtud `maxtemps[0]` je sám polem o pěti prvcích typu `int`. Prvním prvkem pole `maxtemps[0]` je `maxtemps[0][0]` a tento prvek je jediným `int`. Pro přístup k prvkům typu `int` musíte tedy použít dva indexy. O prvém indexu můžete uvažovat jako o reprezentantu řádku a o druhém jako o reprezentantu sloupce. Viz obrázek 5.6.

int maxtemps[4][5];

Zobrazení pole maxtemps ve formě tabulky:

		0	1	2	3	4
maxtemps[0]	0	maxtemps[0][0]	maxtemps[0][1]	maxtemps[0][2]	maxtemps[0][3]	maxtemps[0][4]
maxtemps[1]	1	maxtemps[1][0]	maxtemps[1][1]	maxtemps[1][2]	maxtemps[1][3]	maxtemps[1][4]
maxtemps[2]	2	maxtemps[2][0]	maxtemps[2][1]	maxtemps[2][2]	maxtemps[2][3]	maxtemps[2][4]
maxtemps[3]	3	maxtemps[3][0]	maxtemps[3][1]	maxtemps[3][2]	maxtemps[3][3]	maxtemps[3][4]

Obrázek 5.6 Přístup k prvkům pole pomocí indexů

Předpokládejme, že chcete vytisknout celý obsah pole. Můžete tedy použít jeden cyklus `for` na změnu řádků a druhý, vložený, na změnu sloupců:

```
for (int row = 0; row < 4; row++)
{
        for (int col = 0; col < 5; col++)
             cout << maxtemps[row][col] << "\t";
        cout << "\n";
}
```

Inicializace dvourozměrného pole

Když vytváříte dvourozměrné pole, máte možnost volby inicializace každého prvku. Postup je založen na inicializaci jednorozměrného pole. Tato metoda, pamatujete si, spočívá v poskytnutí seznamu hodnot odděleného čárkami, který je uzavřen do složených závorek:

```
// inicializace jednodimenzionálního pole
int btus[5] = { 23, 26, 24, 31, 28};
```

Ve dvourozměrném poli je každý prvek sám polem, takže můžete každý prvek inicializovat pomocí tvaru, jak je uvedeno výše. Tedy inicializace sestává ze skupin jednorozměrných inicializací oddělených čárkou, které jsou všechny uzavřeny do série složených závorek:

```
int maxtemps[4][5] =                    // 2-D pole
{
        {94, 98, 87, 103, 101},         // hodnoty pro maxtemps[0]
        {98, 99, 91, 107, 105},         // hodnoty pro maxtemps[1]
        {93, 91, 90, 101, 104},         // hodnoty pro maxtemps[2]
        {95, 100, 88, 105, 103}         // hodnoty pro maxtemps[3]
};
```

Výraz `{94, 98, 87, 103, 101}`, který inicializuje první řádek, je reprezentován prvkem `maxtemps[0]`. Umístění každé řady dat na svůj vlastní řádek je věcí stylu a je-li to možné, pak to usnadňuje čtení dat.

Výpis programu 5.19 začleňuje do programu inicializované dvourozměrné pole a vložený cyklus. Tentokrát program obrací pořadí cyklů, umísťuje sloupcový cyklus (index města) na vnější stranu a řádkový cyklus (index roku) dovnitř. Také používá v C++ běžný postup inicializace pole ukazatelů sadou řetězcových konstant. To jest, `cities` se deklaruje jako pole ukazatelů na `char`. To vytváří z každého prvku, jako například `cities[0]`, ukazatel na `char`, který se může inicializovat adresou řetězce. Program inicializuje `cities[0]` adresou řetězce „`Gribble City`" atd. Toto pole ukazatelů je tedy v podstatě polem řetězců.

Výpis programu 5.19 nested.cpp

```
// nested.cpp - vložené cykly a 2-D pole
#include <iostream>
using namespace std;
const int Cities = 5;
const int Years = 4;
int main()
{
    const char * cities[Cities] =   // pole ukazatelů
    {                                  // až 5 řetězců
        "Gribble City",
        "Gribbleton",
        "New Gribble",
        "San Gribble",
        "Gribble Vista"
```

```
};

int maxtemps[Years][Cities] =   // 2-D pole
{
        {94, 98, 87, 103, 101}, // values for maxtemps[0]
        {98, 99, 91, 107, 105}, // values for maxtemps[1]
        {93, 91, 90, 101, 104}, // values for maxtemps[2]
        {95, 100, 88, 105, 103} // values for maxtemps[3]
};

cout << "Maximum temperatures for 1995 - 1998\n\n";
for (int city = 0; city < Cities; city++)
{
        cout << cities[city] << ":\t";
        for (int year = 0; year < Years; year++)
            cout << maxtemps[year][city] << "\t";
        cout << "\n";
}

return 0;
}
```

Zde je výstup programu:

```
Maximum temperatures for 1995 - 1998
Gribble City:     94     98     93     95
Gribbleton:       98     99     91     100
New Gribble:      87     91     90     88
San Gribble:      103    107    101    105
Gribble Vista:    101    105    104    103
```

Uplatnění tabulátorů ve výstupu rozmisťuje data mnohem pravidelněji, než by bylo použití mezer. Kapitola 16 ukazuje preciznější, ale složitější metody formátování výstupu.

Shrnutí

C++ nabízí tři varianty cyklů: `for`, `while` a `do while`. Cyklus probíhá stejnou skupinou instrukcí opakovaně tak dlouho, dokud se testovací podmínka vyhodnocuje jako `true` nebo nenulová, cyklus se ukončí, když se podmínka vyhodnotí jako `false` nebo nula. Cykly `for` a `while` jsou cykly se vstupní podmínkou, to znamená, že prověřují testovací podmínku před provedením příkazů v těle cyklu. Cyklus `do while` je cyklem s výstupní podmínkou, to znamená, že vyhodnotí podmínku po provedení příkazů v těle cyklu.

Syntaxe každého cyklu vyžaduje, aby se tělo cyklu skládalo z jediného příkazu. Avšak tento příkaz může být příkazem složeným, neboli blokem, který se formuje několika příkazy uzavřenými párem složených závorek.

Relační výrazy, které porovnávají dvě hodnoty, se často používají jako testovací podmínky cyklu. Relační výrazy se utvářejí použitím jednoho ze šesti relačních operátorů: <, <=, ==, >=, > nebo !=. Relační výrazy se vyhodnocují jako hodnoty typu `bool` na `true` anebo `false`.

Mnoho programů čte textový vstup nebo textové soubory znak po znaku. Třída `istream` poskytuje několik způsobů jak to udělat. Jestliže je `ch` proměnnou typu `char`, příkaz

```
cin >> ch;
```

čte další vstupní znak do `ch`. Přeskakuje však mezery, znaky nového řádku a tabulátory. Volání členské funkce

```
cin.get(ch);
```

čte další vstupní znak bez ohledu na jeho hodnotu a umisťuje ho do `ch`. Volání členské funkce `cin.get()` vrací další vstupní znak včetně mezer, znaků nového řádku a tabulátorů, takže se může použít následovně:

```
ch = cin.get();
```

Volání členské funkce `cin.get(char)` informuje o setkání s podmínkou konce souboru navrácením hodnoty `booleovské` konverze na `false`, zatímco volání členské funkce `cin.get()` ohlásí konec souboru navrácením hodnoty `EOF`, která se definuje v souboru `iostream`. Vložený cyklus je cyklus uvnitř cyklu. Vložené cykly poskytují přirozený způsob zpracování dvoudimenzionálních polí.

Opakovací otázky

1. Jaký je rozdíl mezi cyklem se vstupní a cyklem s výstupní podmínkou? Jakým druhem je každý cyklus v C++?
2. Co by vytiskl následující kousek programového kódu, kdyby byl částí platného programu?

```
int i;
for (i = 0; i < 5; i++)
        cout << i;
        cout << "\n";
```

3. Co by vytiskl následující kousek programového kódu, kdyby byl částí platného programu?

```
int j;
for (j = 0; j < 11; j += 3)
        cout << j;
cout << "\n" << j << "\n";
```

4. Co by vytiskl následující kousek programového kódu, kdyby byl částí platného programu?

```
int j = 5;
while ( ++j < 9)
        cout << j++ << "\n";
```

5. Co by vytiskl následující kousek programového kódu, kdyby byl částí platného programu?

```
int k = 8;
```

```
do
    cout <<" k = " << k << "\n";
while (k++ < 5);
```

6. Napište cyklus for, který tiskne hodnoty 1 2 4 8 16 32 64 zvyšováním hodnoty čítací proměnné činitelem 2 v každém průběhu.
7. Jak vytvoříte tělo cyklu, aby obsahovalo více než jeden příkaz?
8. Je následující příkaz platný? Jestliže ne, proč ne? Jestliže ano, co dělá?

```
int x = (1,024);
```

Co třeba následující?

```
int y;
y = 1,024;
```

9. Jak se liší `cin >> ch` od `cin.get(ch)` a `ch = cin.get()` v tom, jak pohlíží na vstup?

Programovací cvičení

1. Napište program, který požádá uživatele o vstup dvou celých čísel. Program by potom měl vypočítat součet celých čísel mezi těmito čísly včetně a oznámit výsledek. V tomto bodě předpokládáme, že se menší celé číslo zavede první. Například, když uživatel zavede 2 a 9, program oznámí, že součet všech celých čísel mezi 2 a 9 je 44.
2. Napište program, který vás požádá, abyste zadali čísla. Po každém vstupu čísla oznamuje doposud narůstající součet všech vstupů. Program se ukončí, když zavedete nulu.
3. Daphne investuje 100$ na jednoduchý 10% úrok. To jest, každý rok investice vzroste o 10 % původní investice, neboli o 10$ každý rok:

   úrok = 0.10 x původní zůstatek

   Stejnou dobou Cleo investuje 100$ na složený 5% úrok. To jest úrok je 5 % z běžného zůstatku včetně předchozího přírůstku úroku:

   úrok = 0.05 x běžný zůstatek

   Cleo vydělá 5 % ze 100$ první rok, což jí poskytne 105$. Další rok vydělá 5 % ze 105$, neboli 5.25$ atd. Napište program, který zjistí, kolik let je třeba pro Cleinu investici, aby překročila investici Daphniinu a potom zobrazte hodnoty obou investic v té době.
4. Prodáváte *C++ For Fools*. Napište program, kterým máte zadat roční hodnotu měsíčních prodejů (v pojmech počtu knih, nikoli peněz). Program by měl použít cyklus, aby vás vyzval pomocí jména měsíce, který používá pole `char *` inicializované řetězci ze jmen měsíců a který ukládá vstupní data do pole typu `int`. Potom by měl najít součet obsahů pole a zobrazit celkový prodej za rok.

5. Proveďte programovací cvičení 4, ale použijte na uložení vstupních hodnot měsíčních prodejů za tři roky dvourozměrné pole. Zobrazte celkové prodeje za každý jednotlivý rok a za všechny roky.

6. Navrhněte strukturu, která by se jmenovala `car` a obsahovala následující informace o automobilu: jeho značku jako řetězec ve znakovém poli a rok, kdy byl vyroben jako celé číslo. Napište program, který požádá uživatele, kolik aut má zavést do katalogu. Program by měl potom použít metodu `new` na vytvoření dynamického pole tohoto množství struktur typu `car`. Dále by měl požádat uživatele, aby zavedl značku (která může sestávat z několika slov) a informaci o roku pro každou strukturu. Všimněte si, že to vyžaduje určitou pečlivost, protože se střídá čtení řetězců s numerickými daty (viz kapitola 4). Nakonec byste měli zobrazit obsah každé struktury. Vzorek běhu programu by měl vypadat jako tento následující:

```
How many cars do you wish to catalog? 2
Car #1:
Please enter the make: Hudson Hornet
Please enter the year made: 1952
Car #2:
Please enter the make: Kaiser
Please enter the year made: 1951
Here is your collection:
1952 Hudson Hornet
1951 Kaiser
```

Příkazy větvení a logické operátory

KAPITOLA 6

Témata kapitoly:

- Příkaz `if`
- Příkaz `if else`
- Logické operátory `&&`, `||` a `!`
- Knihovna `cctype` obsahující funkce pro práci se znaky
- Podmíněný operátor ?:
- Příkaz `switch`
- Příkazy `continue` a `break`
- Cykly, které čtou čísla

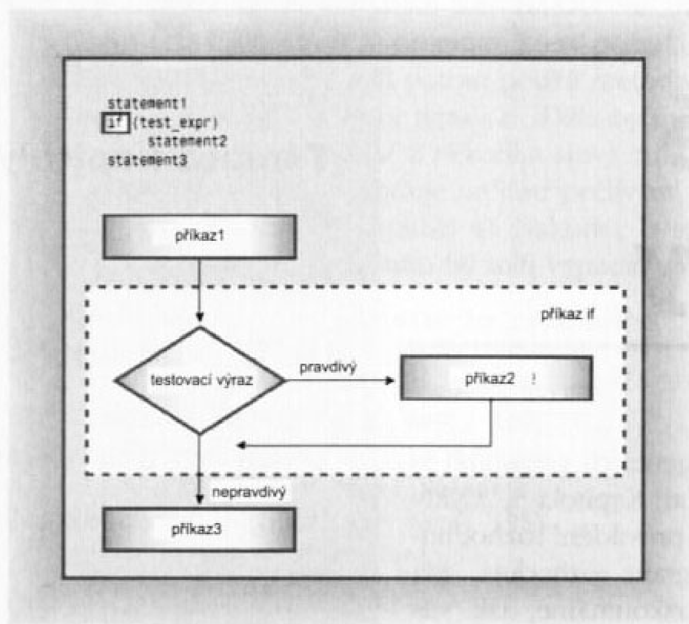
Jedním z klíčů pro navrhování inteligentních programů je dát jim schopnost provádět rozhodnutí. Kapitola 5, „Cykly a relační výrazy", ukazuje jeden druh provádění rozhodnutí – cyklování – ve kterém se program rozhoduje, zda v něm bude pokračovat. Nyní prozkoumáme, jak vás C++ nechá používat příkazy větvení, aby se rozhodl mezi alternativními činnostmi. Jaké schéma na ochranu proti upírům (česnek nebo kříž) by měl program používat? Zadal uživatel nulu? C++ poskytuje na implementaci rozhodnutí operátory `if` a `switch` a ty jsou hlavním tématem kapitoly. Také se podíváte na podmíněný operátor, který poskytuje jiný způsob provedení rozhodnutí a na logické operátory, které vám umožní spojovat více testů do jednoho.

Příkaz if

Když si program v C++ musí vybrat, zda udělá určitou činnost, můžete použít operátor `if`. `If` se vyskytuje ve dvou podobách: `if` a `if else`. Vyšetříme nejprve jednoduchý if. Je modelován podle běžné angličtiny, jako například v „If you have a Captain Cookie card, you get a free cookie" (Máte-li kartu kapitána Cookie, dostanete zadarmo zákusek). Příkaz `if` směruje program k provedení příkazu nebo bloku příkazů, je-li testovací podmínka true a tyto příkazy nebo blok přeskočí, je-li `false`. Syntaxe je podobná syntaxi `while`:

```
if (testovaci_podminka)
  příkaz
```

True nebo nenulová testovaci_podminka způsobí, že program provede příkaz, který může být jediným příkazem nebo blokem. False nebo nulová testovaci_podminka působí, že program příkaz přeskočí. Viz obrázek 6.1. Celá konstrukce if se považuje za jediný příkaz.



Obrázek 6.1 Příkaz if

Obvykle je testovaci_podminka relačním výrazem, který byl například použit na řízení cyklů. Předpokládejme například, že chcete, aby program počítal mezery na vstupu a zároveň celkový počet znaků. V cyklu while můžete použít na čtení znaků funkci cin.get(char) a potom příkaz if, abyste identifikovali a spočítali znaky mezera. To právě dělá výpis programu 6.1 a na rozpoznání konce věty používá tečku.

Výpis programu 6.1 if.cpp

```
#include <condefs.h>
// if.cpp - použití příkazu if
#include <iostream>
using namespace std;
int main()
{
    char ch;
    int spaces = 0;
    int total = 0;
    cin.get(ch);
    while (ch != '.')   // ukončení na konci věty
    {
        if (ch == ' ')  // test, zda je ch mezera
            spaces++;
        total++;        // provádí se pokaždé
```

```
        cin.get(ch);
    }
    cout << spaces << " mezer, " << total;
    cout << " celkovy pocet znaku ve vete\n";
    return 0;
}
```

Zde je jeden vzorek výstupu:

```
Vzduchoplavec byl vzdusnym predmostim
s vysokymi cili.
5 mezer, 53 znaku celkem ve vete
```

Jak ukazují poznámky, příkaz `spaces++;` se provádí pouze když je `ch` mezerou. Příkaz `total++;` se provádí v každém průchodu cyklem, protože je vně příkazu `if`. Všimněte si, že čítač `total` zahrnuje znak nového řádku, která se generuje stiskem `Enter`.

Příkaz if else

Příkaz `if` způsobí, že se program rozhodne, zda provede *určitý* příkaz nebo blok. Příkaz `if else` způsobí, že program rozhodne, který ze *dvou* příkazů nebo bloků provede. Je neocenitelným příkazem na vytvoření alternativních průběhů činností. Příkaz `if else` se v C++ modeluje podle jednoduché angličtiny, jako například ve větě „If you have a Captain Cookie Card, you get a Cookie Plus Plus, else you just get a Cookie d'Ordinaire". (Máte-li kartu kapitána Cookie, dostanete zákusek Plus Plus, jinak pouze zákusek d'Ordinaire). Příkaz `if else` má tento obecný tvar:

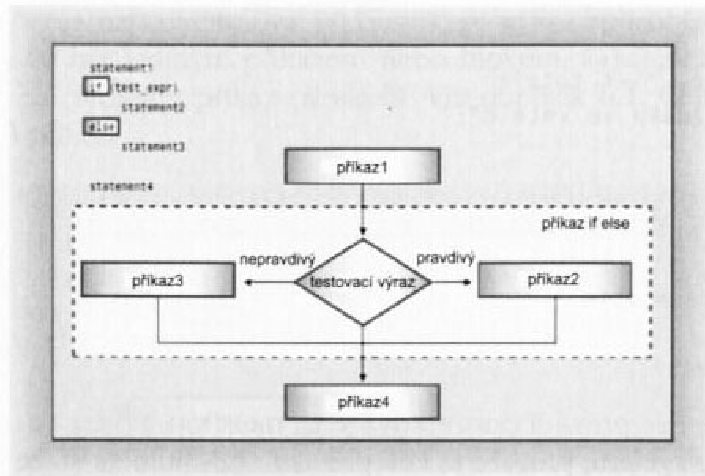
```
if (testovaci _podminka)
  příkaz1
else
  příkaz2
```

Je-li `testovaci-podminka` true nebo nenulová, program provede `příkaz1` a přeskočí `příkaz2`. Ale, jestliže je `testovaci_podminka` false nebo nula, program přeskočí `příkaz1` a provede místo něj `příkaz2`. Takže fragment programového kódu

```
if (answer == 1492)
    cout << "To je spravne!\n";
else
     cout << "Meli byste si lepe zopakovat Kapitolu 1.\n";
```

tiskne první zprávu, je-li `answer` rovna 1492, jinak tiskne druhou zprávu. Každý příkaz může být buď jednoduchým příkazem nebo blokem příkazů, ohraničeným složenými závorkami. Viz obrázek 6.2. Celá konstrukce se považuje syntakticky za jediný příkaz.

Předpokládejte například, že chcete změnit zadávaný text promícháním písmen, zatímco znak nového řádku necháte nedotčeným. Tímto způsobem se každý vstupní řádek konvertuje na výstupní o stejné délce. To znamená, že chcete, aby program zacházel jedním způsobem se znaky nového řádku a jiným se všemi ostatními znaky. Jak ukazuje výpis programu 6.2, `if else` vykonává tuto úlohu jednoduše.



Obrázek 6.2 Příkaz if else

Výpis programu 6.2 ifelse.cpp

```
#include <condefs.h>
// ifelse.cpp — použití příkazu if else
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout << "Napis a ja budu opakovat.\n";
    cin.get(ch);
    while (ch != '.')
    {
        if (ch == '\n')
            cout << ch;     // provádí se, když ch je znak nového řádku
        else
            cout << ++ch;   // provádí se jindy
        cin.get(ch);
    }
    // zkuste ch + 1 místo ++ch kvůli zajímavému důsledku
    cout << "\nProsim, omluvte mirnou zamenu.\n";
    return 0;
}
```

Zde je jeden vzorek výstupu:

```
Napis a ja budu opakovat.
Jsem vyjimecne potesen.
Ktfn!wzkjnfdof!qpuftfo-
ze mohu pouzivat takovy vykonny pocitac.
{f!npiv!qpv{jwbu!ublpwz!wzlpooz!qpdjubd
Prosim, omluvte mirnou zamenu.
```

Všimněte si jednoho z komentářů v programu, který napovídá, že přeměna `++ch` na `ch+1` má zajímavý důsledek. Mohli byste usoudit, jaký bude? Jestliže ne, vyzkoušejte to a potom uvidíte, zda můžete vysvětlit, co se přihodilo. (Rada: Přemýšlejte o tom, jak `cout` zachází s různými typy.)

Formátování vašich příkazů `if else`

Pamatujte si, že obě alternativy `if else` musí být jednotlivými příkazy. Potřebujete-li více než jeden příkaz, použijte na jejich spojení do jediného příkazového bloku složené závorky. Na rozdíl od některých jazyků, jako jsou například BASIC nebo Fortran, nepovažuje C++ automaticky vše mezi `if` a `else` za blok, takže abyste z příkazů vytvořili blok, musíte použít složené závorky. Následující programový kód například produkuje chybu kompilátoru. Kompilátor se na to dívá jako na jednoduchý příkaz `if` a končí příkazem `zorro++;`. Potom následuje příkaz `cout`. Dosud také dobré. Avšak potom se vyskytuje něco, co kompilátor chápe jako nezávislý `else` a to se označí jako chyba.

```
if (ch == 'Z')
    zorro++;            // if tady končí
    cout << "Dalsi kandidat Zorro\n";
else                    // chybně
    dull++;
    cout << "To neni kandidat Zorro\n";
```

Jestliže potřebujete přeměnit programový kód na co potřebujete, dodejte složené závorky:

```
if (ch == 'Z')
{                              // tento blok, je-li pravdivostni hodnota true
    zorro++;
    cout << "Dalsi kandidat Zorro\n";
}
else
{                              // tento blok, je-li pravdivostni hodnota false
    dull++;
    cout << "To neni kandidat Zorro\n";
}
```

Protože má jazyk C++ volný formát, můžete uspořádat závorky, jak se vám líbí, pokud budou uzavírat příkazy. Předchozí programový kód ukazuje jeden z populárních formátů. Zde je jiný:

```
if (ch == 'Z') {
    zorro++;
    cout << " Dalsi kandidat Zorro\n";
    }
else {
    dull++;
    cout << "To neni kandidat Zorro\n";
    }
```

První tvar zdůrazňuje blokovou strukturu příkazů, zatímco druhý těsněji váže bloky ke klíčovým slovům `if` a `else`. Pokud nenarazíte na vášnivého obhájce nějakého určitého stylu, oba styly by vám měly dobře sloužit.

Konstrukce `if else if else`

Počítačové programy, podobně jako život, by vám mohly umožňovat více než dvě alternativy výběru. Abyste vyhověli těmto možnostem, příkaz `if else` můžete v C++ rozšířit. `Else`, který jste viděli, by mohl být následován jediným příkazem, který může být i blokem. Protože příkaz `if else` je sám o sobě jediným příkazem, může následovat za `else`:

```
if (ch == 'A')
    a_grade++;                  // alternativa # 1
else
    if (ch == 'B')              // alternativa # 2
        b_grade++;              // dílčí alternativa # 2a
    else
        soso++;                 // dílčí alternativa # 2b
```

Jestliže `ch` není `'A'`, program jde na `else`. Tam druhý `if else` rozděluje alternativy na další dvě volby. Volné formátování v C++ vám umožňuje uspořádat tyto prvky do lépe čitelné podoby:

```
if (ch == 'A')
    a_grade++;              // alternativa # 1
else if (ch == 'B')
    b_grade++;              // alternativa # 2
else
    soso++;                 // alternativa # 3
```

Vypadá to jako nová řídící struktura – struktura `if else if else`. Ale ve skutečnosti je to jediný `if else` spojený s dalším. Takto upravený formát vypadá mnohem čistěji a umožňuje vám přelétnout programový kód a rozeznat různé alternativy. Celá konstrukce se stále považuje za jediný příkaz.

Výpis programu 6.3 toto upřednostňované formátování používá na vytvoření jednoduchého dotazovacího programu.

Výpis programu 6.3 ifelseif.cpp

```
#include <condefs.h>
// ifelseif.cpp - použití if else if else
#include <iostream>
using namespace std;
const int Fave = 27;
int main()
{
    int n;

    cout << "Zadejte cislo v rozsahu 1-100 ke zjisteni ";
    cout << "mého oblibeneho cisla: ";
```

```
    do
    {
        cin >> n;
        if (n < Fave)
            cout << "Prilis nizke - hadejte znovu: ";
        else if (n > Fave)
            cout << "Prilis vysoke - hadejte znovu: ";
        else
            cout << Fave << " je spravne!\n";
    } while (n != Fave);
    return 0;
}
```

Zde je jeden vzorek výstupu:

```
Zadejte cislo v rozsahu 1-100 ke zjisteni mého oblibeneho cisla: 50
Prilis vysoke - hadejte znovu: 25
Prilis nizke - hadejte znovu: 37
Prilis vysoke - hadejte znovu: 31
Prilis vysoke - hadejte znovu in: 28
Prilis vysoke - hadejte znovu: 27
27 je spravne!
```

Logické výrazy

Často musíte testovat více než jednu podmínku. Například, aby byl znak malým písmenem, jeho hodnota musí být větší nebo rovna 'a' a menší nebo rovna 'z'. Nebo požadujete-li od uživatele, aby odpověděl a (ano) nebo n (ne), chcete akceptovat jak velká (A a N) tak malá písmena. Abyste tomuto druhu požadavku vyhověli, C++ poskytuje na spojení nebo úpravu existujících výrazů tři logické operátory. Operátory jsou logické NEBO, psáno ||; logické A, psáno &&; a logické NE, psáno !. Vyzkoušejme je nyní:

Logický operátor NEBO: ||

V angličtině slovo or (nebo) může označovat jednu nebo obě ze dvou podmínek, které vyhovují požadavku. Například můžete jít na piknik do společnosti MegaMicro, jestliže vy *nebo* vaše manželka pracujete v MegaMicro, Inc. V C++ je ekvivalentem logický operátor NEBO, psáno ||. Tento operátor spojuje dva výrazy do jednoho. Když je jeden nebo druhý z původních výrazů true nebo nenulový, výsledný výraz má hodnotu true; v opačném případě má výraz hodnotu false. Zde je několik příkladů:

```
5 == 5 || 5 == 9    // true protože je první výraz true
5 > 3  || 5 > 10    // true protože je první výraz true
5 > 8  || 5 < 10    // true protože je druhý výraz true
5 < 8  || 5 > 2     // true protože jsou oba výrazy true
5 > 8  || 5 < 2     // false protože jsou oba výrazy false
```

Protože || má nižší prioritu, než relační operátory, nemusíte v těchto příkazech použít závorky. Tabulka 6.1 shrnuje, jak operátor || pracuje.

Tabulka 6.1 Operátor ||

Hodnoty expr1 || expr2

	expr1 == true	expr1 == false
expr2 == true	true	true
expr2 == false	true	false

C++ stanovuje, že operátor || je sekvenčním článkem. To znamená, že jakékoli změny hodnot uvedené na levé straně se uskutečňují před tím, než se vyhodnotí pravá strana. Například uvažujme následující výraz:

```
i++ < 6 || i == j
```

Předpokládejme, že i má původně hodnotu 10. Jakmile nastane porovnání s j, má hodnotu 11. Tedy C++ se nebude zatěžovat vyhodnocením výrazu napravo, bude-li výraz nalevo pravdivý, protože pouze jediný pravdivý výraz činí celý logický výraz pravdivým.

Výpis programu 6.4 používá operátor || v příkazu if, aby ověřil verze znaku jak pro velká, tak pro malá písmena. Také používá na rozšíření jediného řetězce přes tři řádky vlastností C++ pro spojení řetězců (viz kapitola 4, „Odvozené typy“).

Výpis programu 6.4 or.cpp

```
#include <condefs.h>
// or.cpp - použití logického operátoru OR
#include <iostream>
using namespace std;
int main()
{
    cout << "Tento program muze preformatovat vas harddisk\n"
            "a zrusit vsechna vase data.\n"
            "Prejete si pokracovat? <a/n> ";
    char ch;
    cin >> ch;
    if (ch == 'a' || ch == 'A')              // a or A
        cout << "Byl jste upozornen!\a\a\n";
    else if (ch == 'n' || ch == 'N')         // n or N
        cout << "Moudra volba ... sbohem\n";
    else
        cout << "Nebylo zadano ani a ani n, tak se tedy domnivam,\n"
        "ze vas disk v kazdem pripade znicim.\n";
    return 0;
}
```

Zde je vzorek běhu programu:

```
Tento program muze preformatovat vas harddisk
a zrusit vsechna vase data.
Prejete si pokracovat? <a/n> N
Moudra volba ... sbohem
```

Program čte pouze jeden znak, takže v odpovědi má význam jen první znak. To znamená, že by uživatel mohl odpovědět NE! místo N. Program by přečetl pouze N. Avšak kdyby se program pokusil přečíst později více vstupních veličin měl by začít na E.

Logický operátor A: &&

Logický operátor A, psáno `&&`, spojuje do jednoho dva výrazy. Výsledný výraz má hodnotu `true` pouze tehdy, mají-li oba původní výrazy hodnotu `true`. Zde je několik příkladů:

```
5 == 5 && 4 == 4   // true protože jsou oba výrazy true
5 == 3 && 4 == 4   // false protože je první výraz false
5 > 3 && 5 > 10    // false protože je druhý výraz false
5 > 8 && 5 < 10    // false protože je první výraz false
5 < 8 && 5 > 2     // true protože jsou oba výrazy true
5 > 8 && 5 < 2     // false protože jsou oba výrazy false
```

Protože `&&` má nižší prioritu než relační operátory, nemusíte v těchto výrazech použít závorky. Podobně jako operátor `||`, operátor `&&` funguje jako sekvenční článek, takže se vyhodnotí levá strana a provedou se ještě boční efekty před tím, než se vyhodnotí pravá strana. Je-li levá strana nepravdivá, celý logický výraz musí být nepravdivý, takže se C++ v tomto případě nezatěžuje pravou stranou. Tabulka 6.2 shrnuje, jak operátor `&&` pracuje.

Tabulka 6.2 Operátor &&

Hodnoty `expr1 && expr2`

	expr1 == true	expr1 == false
expr2 == true	true	false
expr2 == false	false	false

Výpis programu 6.5 ukazuje, jak se použití `&&` umí vypořádat s běžnou situací ukončením cyklu `while` ze dvou různých důvodů. Ve výpisu programu načítá cyklus `while` hodnoty do pole. První test (`i < ArSize`) ukončuje cyklus, když je pole plné. Druhý test (`temp >= 0`) dává uživateli možnost skončit dříve zadáním záporného čísla. Operátor `&&` vám umožní spojit oba testy do jediné podmínky. Program také používá dva příkazy `if`, příkaz `if else` a cyklus `for`, takže předvádí několik témat z této a předchozí kapitoly.

Výpis programu 6.5 and.cpp

```
#include <condefs.h>
// and.cpp - použití logického operátoru AND
#include <iostream>
using namespace std;
const int ArSize = 6;
int main()
{
    float naaq[ArSize];
    cout << "Zadejte KUNV (Koeficienty uvedomeni noveho veku) "
```

```
        << "\nvasich sousedu. Program konci, "
        << "kdyz provedete\n" << ArSize << " zadani "
        << "nebo vlozite zapornou hodnotu.\n";

    int i = 0;
    float temp;
    cin >> temp;
    while (i < ArSize && temp >= 0) // 2 kritéria na ukončení
    {
        naaq[i++] = temp;
        if (i < ArSize)              // v poli ještě zbývá prostor,
            cin >> temp;             // takže získání další hodnoty
    }
    if (i == 0)
        cout << "Zadne udaje—sbohem\n";
    else
    {
        cout << "Vlozte svuj KUNV: ";
        float you;
        cin >> you;
        int count = 0;
        for (int j = 0; j < i; j++)
            if (naaq[j] > you)
                count++;
            cout << count;
        cout << " z vasich sousedu ma vetsi uvedomeni\n"
            << "Noveho veku, nez vy.\n";
    }
    return 0;
}
```

Všimněte si, že program umisťuje vstupní data do dočasné proměnné `temp`. Pouze až když ověří, že vstupní data jsou platná, přiřadí hodnotu do pole.

Zde je pár vzorových běhů programu. Jeden se ukončí po šesti vstupech, druhý po zadání záporné hodnoty:

```
Zadejte KUNV (Koeficienty uvedomeni noveho veku
vasich sousedu. Program konci, kdyz provedete
6 zadani nebo vlozite zapornou hodnotu.
28 72 19
6
130 145
Vlozte svuj KUNV: 50
3 of your neighbors have greater awareness of
the New Age than you do.

Zadejte KUNV (Koeficienty uvedomeni noveho veku
vasich sousedu. Program konci, kdyz provedete
6 zadani nebo vlozite zapornou hodnotu.
123 119
4
```

```
89
-1
Vlozte svuj KUNV: 123.027
0 z vasich sousedu ma vetsi uvedomeni
Noveho veku, nez vy.
```

Poznámky k programu

Podívejte se na vstupní část programu:

```
cin >> temp;
while (i < ArSize && temp >= 0)        // 2 ukončovací kritéria
{
        naaq[i++] = temp;
        if (i < ArSize)                 // je-li prostor v poli,
                cin >> temp;            // tak čtěte další hodnotu
}
```

Program začíná čtením první vstupní hodnoty do dočasné proměnné, která se nazývá `temp`. Potom testovací podmínka ve `while` ověří, zda je ještě v poli ponechán dostatečný prostor (`i < ArSize`) a zdali je vstupní hodnota nezáporná (`temp >= 0`). Jestliže je tomu tak, kopíruje hodnotu `temp` do pole a zvyšuje index o 1. Tady, protože číslování pole začíná od 0, se `i` rovná celkovému počtu dosud zavedených vstupů. To znamená, že jestliže `i` začíná na 0, první průběh cyklem přiřadí hodnotu do `naaq[0]` a `i` se nastaví na 1. Cyklus končí, když se pole naplní nebo když uživatel zavede záporné číslo. Všimněte si, že cyklus čte další hodnotu do `temp` pouze tehdy, jestliže je `i` menší než `ArSize`, to jest, jestliže je ještě v poli ponechán dostatek prostoru.

Po získání údajů program použije příkaz `if else`, aby oznámil, že nebyla zavedena žádná data (v případě, že prvním vstupem bylo záporné číslo) a jestliže jsou údaje přítomny, zpracuje je.

Nastavení rozmezí pomocí &&

Operátor `&&` vám také umožní nastavit skupiny příkazů `if else if else` každý na alternativu, která odpovídá určitému rozsahu hodnot. Výpis programu 6.6 tento postup ukazuje. Také ukazuje užitečný postup na zacházení se skupinou zpráv. Stejně tak, jako může proměnná ukazatel-na-`char` rozpoznat řetězec tím, že ukáže na jeho začátek, pole ukazatelů-na-`char` může rozpoznat skupinu řetězců. Jednoduše přiřaďte adresu každého řetězce do různého prvku pole. Výpis programu 6.6 používá pole `qualify` na uchování adres čtyř řetězců. Například `qualify[1]` obsahuje adresu řetězce „`pretahovani v blate.\n`". Program potom může použít `qualify[1]` jako libovolný další ukazatel na řetězec, například, s `cout` nebo s `strlen()` nebo s `strcmp()`. Použití kvalifikátoru `const` chrání řetězce před náhodnými změnami.

Výpis programu 6.6 more_and.cpp

```
#pragma hdrstop
#include <condefs.h>
// more_and.cpp — použití logického operátoru AND
```

```
#include <iostream>
using namespace std;
const char * qualify[4] =                        // pole ukazatelů
{                                                // na řetězce
    "zavod na 10,000 metru.\n",
    "pretahovani v blate.\n",
    "mistrovske utkani v jizde na kanoich.\n",
    "svatek vrhani kolacu.\n"
};
int main()
{
    int age;
    cout << "Zadejte vas vek v rocich: ";
    cin >> age;
    int index;
    if (age > 17 && age < 35)
        index = 0;
    else if (age >= 35 && age < 50)
        index = 1;
    else if (age >= 50 && age < 65)
        index = 2;
    else
        index = 3;
    cout << "Kvalifikujete se na " << qualify[index];
    return 0;
}
```

Kompatibilita:

Možná si vzpomenete, že některé implementace vyžadují v deklaraci pole použití klíčového slova `static`, aby umožnily toto pole inicializovat. Omezení, jak se o něm pojednává v kapitole 8, se používá na pole deklarovaná uvnitř těla funkce. Když se pole deklaruje vně těla funkce, jako je `qualify` ve výpisu programu 6.6, nazývá se *externím polem* a může se inicializovat dokonce i v implementacích, které předcházejí ANSI C.

Zde je vzorek běhu programu:

```
Zadejte vas vek v rocich: 87
Kvalifikujete se na svatek vrhani kolacu.
```

Zavedený věk neodpovídá žádnému z rozsahů testů, takže program nastaví index na 3 a potom vytiskne odpovídající řetězec.

Poznámky k programu

Výraz `age > 17 && age < 35` testuje věk mezi dvěma hodnotami; to jest, věk je v rozsahu 18–34. Výraz `age >= 35 && age < 50` používá operátor `<=`, aby se do jeho rozsahu zahrnulo 35, tedy rozsah 35–49. Kdyby program použil `age > 35 && age < 50`, hodnota 35 by se všemi testy opomenula. Když použijete testování rozsahů, měli byste ověřit, že rozsahy nemají mezi sebou díry a že se nepřekrývají. Také se ujistěte, že jsou všechny roz-

sahy nastaveny správně; viz poznámka na téma Rozsahy testů. Příkaz if else slouží na výběr indexu pole, které na oplátku identifikuje určitý řetězec.

Rozsahy testů

Všimněte si, že by každá část rozsahu testu měla na spojení dvou úplných relačních výrazů používat operátor A:

```
if (age > 17 && age < 35)    // v pořádku
```

Nepůjčujte si z matematiky a nepoužívejte následující zápis:

```
if (17 < age < 35) // Nedělejte to!
```

Uděláte-li tuto chybu, kompilátor ji nezachytí, protože je to stále platná syntaxe v C++. Operátor < se spojuje zleva doprava, takže předchozí výraz znamená následující:

```
if ( (17 < age) < 35)
```

Avšak `17 < age` je buď `true`, neboli 1, anebo `false`, neboli 0. V obou případech je výraz `17 < age` menší než 35, takže celý test je vždy pravdivý!

Logický operátor NE: !

Operátor ! neguje, neboli obrací pravdivostní hodnotu výrazu, který ho následuje. To jest, jestliže je `výraz true`, `!výraz` je `false` a naopak. Přesněji, je-li `výraz true` nebo nenulový, potom `!výraz` je `false`. Mimochodem mnoho lidí nazývá vykřičník výstřel, vytvoření !x výstřel-exe a !!x výstřel-výstřel-exe (bang, bang-exe, bang-bang-exe).

Obvykle můžete relační vztah vyjádřit bez použití tohoto operátoru:

```
if (!(x > 5))                    // if (x <= 5) je jasné
```

Avšak operátor ! může být užitečný ve funkcích, které navracejí pravdivostní hodnoty nebo hodnoty, které se tímto způsobem mohou interpretovat. Například `strcmp(s1, s2)` navrací nenulovou (`true`) hodnotu, jestliže se oba řetězce navzájem liší a nulovou hodnotu, jsou-li stejné. To má za následek, že `!strcmp(s1, s2)` je `true`, jsou-li si oba řetězce rovny.

Výpis programu 6.7 tuto techniku používá (aplikuje operátor ! na návratovou hodnotu funkce), aby prověřil numerický vstup, je-li vhodný pro přiřazení do typu `int`. Funkce `is_int()`, o které za chvíli pojednáme, navrací `true`, je-li její parametr v rozsahu hodnot, které se mohou přiřadit do typu `int`. Program potom na odmítnutí hodnot, které rozsahu nevyhovují, používá test `while (!is_int(num))`.

Výpis programu 6.7 not.cpp

```
#include <condefs.h>
// not.cpp - použití operátoru not
#include <iostream>
#include <climits>
using namespace std;
```

```
bool is_int(double);
int main()
{
    double num;

    cout << "Halo, frajere! Zadejte celociselnou hodnotu: ";
    cin >> num;
    while (!is_int(num))    // pokračování, dokud není num v rozsahu
    {                        // platnosti typu int
        cout << "Mimo rozsah - prosim, zkuste znovu:   ";
        cin >> num;
    }
    int val = num;
    cout << "Zadali jste cele cislo " << val << "\n";
    return 0;
}
bool is_int(double x)
{
    if (x <= INT_MAX && x >= INT_MIN)   // použití hodnot z climits
        return true;
    else
        return false;
}
```

Připomínka:

Jestliže váš systém neposkytuje `climits`, použijte `limits.h`

Zde je vzorek běhu programu ze systému s 32 bitovým `int`:

```
Halo, frajere! Zadejte celociselnou hodnotu: 6234128679
Mimo rozsah - prosim, zkuste znovu: -8000222333
Mimo rozsah - prosim, zkuste znovu: 99999
Zadali jste cele cislo 99999
```

Poznámky k programu

Zavedete-li příliš velkou hodnotu do programu, který čte typ `int`, většina implementací jednoduše hodnotu zkrátí, aby vyhovovala bez toho, aniž by vás informovala, že se data ztratila. Typ `double` má více než dostatečnou přesnost na úschovu typické hodnoty `int` a její rozsah je mnohem větší.

Booleovská funkce `is_int()` používá dvě symbolické konstanty (`INT_MAX` a `INT_MIN`), které jsou definované v souboru `climits` (bylo o něm pojednáno v kapitole 3, „Práce s daty") na určení, zda je parametr ve správných mezích. Je-li tomu tak, program navrací hodnotu `true`, v opačném případě `false`.

Program `main()` používá na odmítnutí neplatného vstupu cyklus `while`, dokud uživatel nedodá data správně. Můžete vytvořit program přátelštější zobrazením mezí `int`, je-li vstup dat mimo rozsah. Poté, co byla vstupní data ověřena, program je přiřadí do proměnné `int`.

Fakta o logických operátorech

Jak jsme si všimli, logické operátory NEBO a A mají nižší prioritu než relační operátory. To znamená, že výraz jako například

```
x > 5 && x < 10
```

se čte tímto způsobem:

```
(x > 5) && (x < 10)
```

Operátor ! má na druhé straně vyšší prioritu než jakékoli relační nebo aritmetické operátory. Proto, abyste negovali výraz, měli byste ho uzavřít do závorek:

```
!(x > 5) // je false, když je x větší než 5
!x > 5   // !x je větší než 5
```

Mimochodem, druhý výraz je vždy `false`, protože `!x` může mít hodnoty pouze `true` nebo `false`, které se dají konvertovat na 1 nebo 0.

Logický operátor A má vyšší prioritu než NEBO. Proto výraz

```
age > 30 && age < 45 || weight > 300
```

znamená následující:

```
(age > 30 && age < 45) || weight > 300
```

To jest, jedna podmínka znamená, že `age` má být v rozsahu 31 až 44 a druhá, že `weight` má být větší než 300. Celý výraz je `true`, je-li první nebo druhá nebo obě z těchto podmínek pravdivá.

Můžete samozřejmě použít závorky, abyste programu sdělili interpretaci, kterou chcete. Například předpokládejme, že chcete použít `&&` na spojení podmínky, že `age` má být větší než 50 nebo `weight` je větší než 300 s podmínkou, že `donation` má být větší než 1000. Musíte část `NEBO` uzavřít do závorek:

```
(age > 50 || weight > 300) && donation > 1000
```

Jinak kompilátor spojí podmínku `weight` s podmínkou `donation` místo s podmínkou `age`. Nejjednodušším způsobem, jak to učinit, je použít na seskupení testů závorky bez ohledu na to, zda jsou či nejsou potřebné. Vytváří to čitelnější kód.

C++ zaručuje, že když program vyhodnocuje logický výraz, vyhodnocuje ho zleva doprava a zastaví vyhodnocování, jakmile ví, jaká je odpověď. Předpokládejme nyní, že máme tuto podmínku:

```
x != 0 && 1.0 / x > 100.0
```

Je-li první podmínka nepravdivá, celý výraz musí být nepravdivý. To je proto, aby byl tento výraz pravdivý, musí být každá podmínka pravdivá. Víme-li, že je první podmínka nepravdivá, program se neobtěžuje vyhodnocováním druhé. Je to v tomto příkladě příznivé, protože vyhodnocení druhé podmínky by mělo za následek dělení nulou, což není v doméně možných činností počítače.

Knihovna cctype obsahující funkce pro práci se znaky

C++ zdědil z C příhodný balík funkcí, které souvisí se znaky, jejichž prototypy jsou v hlavičkovém souboru cctype (ctype.h ve starším typu), které zjednodušují takové úkoly, jako je určení, zda je znak velkým písmenem nebo číslicí nebo interpunkčním znaménkem a podobně. Například funkce isalpha(ch) vrací nenulovou hodnotu, je-li ch písmeno a nulovou, jestliže není. Podobně ispunct(ch) vrací hodnotu true pouze tehdy, je-li ch interpunkčním znakem, jako jsou například čárka a tečka. (Tyto funkce mají návratový typ int místo bool, ale obvykle vám konverze na bool dovoluje zacházet s nimi jako s typem bool.)

Použití těchto funkcí je vhodnější než použití operátorů A a NEBO. Například tady vidíte, jak byste mohli použít A a NEBO na otestování, zda je znak ch alfabetickým znakem:

```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
```

Porovnejte to s použitím isalpha():

```
if (isalpha(ch))
```

Nejenže je isalpha() na použití snazší, ale je obecnější. Tvary A, NEBO předpokládají, že znakové kódy pro A až Z jdou za sebou a v tomto rozsahu neexistují žádné další kódy. Tento předpoklad je pravdivý pro ASCII kód, ale nemusí být pravdivý obecně.

Výpis programu 6.8 ukazuje některé funkce z této rodiny. Zvláště používá isalpha(), která testuje alfabetické znaky; isdigits(), která testuje číslicové znaky, jako je například 3; isspace(), která testuje znaky oddělovačů, jako jsou například znaky nového řádku, mezery a tabulátory; ispunct(), která testuje znaky interpunkce. Program také referuje o struktuře if else if a používá cyklus while s funkcí cin.get(char).

Výpis programu 6.8 cctypes.cpp

```
#include <condefs.h>
// cctypes.cpp-použití knihovny ctype.h
#include <iostream>
#include <cctype>              // prototypy pro znakové funkce
using namespace std;
int main()
{
    cout << "Zadejte text pro analýzu, a @"
         " na ukončení vstupu.\n";
    char ch;
    int whitespace = 0;
    int digits = 0;
    int chars = 0;
    int punct = 0;
    int others = 0;

    cin.get(ch);                // získá první znak
```

```
    while(ch != '@')            // test na nárazník
    {
        if(isalpha(ch))           // je to alfabetický znak?
            chars++;
        else if(isspace(ch))      // je to netisknutelný znak?
            whitespace++;
        else if(isdigit(ch))      // je to číslice?
            digits++;
        else if(ispunct(ch))      // je to znak interpunkce?
            punct++;
        else
            others++;
        cin.get(ch);               // získá další znak
    }
    cout << chars << " pismen, "
        << whitespace << " netisknutelnych znaku, "
        << digits << " cislic, "
        << punct << " znaku interpunkce, "
        << others << " ostatnich.\n";
    return 0;
}
```

Zde je vzorek běhu programu; všimněte si, že oddělovače jsou včetně znaku nového řádku:

```
Zadejte text pro analyzu, a @ na ukonceni vstupu.
Jody "Java-Java" Joystone, vyznacna posuzovatelka restauraci,
slavila sve 39 narozeniny s karafou Chateau Panda
z roku 1982.@
96 pismen, 16 netisknutelnych znaku, 6 cislic, 6 znaku interpunkce, 0 os-
tatnich.
```

Tabulka 6.3 shrnuje dostupné funkce z balíku cctype. Některé systémy mohou nějaké funkce postrádat nebo mají navíc jiné.

Tabulka 6.3 Znakové funkce z cctype

Jméno funkce	Návratová hodnota
isalnum()	True, je-li parametr alfanumerický, písmeno nebo číslice
isalpha()	True, je-li parametr alfabetický
iscntrl()	True, je-li parametr řídící znak
isdigit()	True, je-li parametr desítková číslice (0-9)
isgraph()	True, je-li parametr libovolný znak, který se dá vytisknout, jiný než mezera
islower()	True, je-li parametr malé písmeno
isprint()	True, je-li parametr libovolný znak, který se dá vytisknout, včetně mezery
ispunct()	True, je-li parametr interpunkční znak
isspace()	True, je-li parametr standardním oddělovacím znakem, to jest mezera, posun formuláře, nový řádek, návrat vozu, horizontální tabulátor, vertikální tabulátor

`isupper()`	True, je-li parametr velké písmeno
`isxdigit()`	True, je-li parametr hexadecimální číselný znak, to jest 0-9, a-f nebo A-F
`tolower()`	Je-li parametr velkým písmenem, `tolower()` navrací verzi malého písmene tohoto znaku, jinak navrací nezměněný parametr
`toupper()`	Je-li parametr malým písmenem, `toupper()` navrací verzi velkého písmene tohoto znaku, jinak navrací nezměněný parametr

Operátor ?:

C++ má operátor, který se velmi často dá použít namísto příkazu `if else`. Tento operátor se nazývá *podmíněným operátorem*, psáno `?:` a pro vás každodenní nadšenci, je to jediný operátor v C++, který vyžaduje tři operandy. Obecný tvar vypadá takto:

```
výraz1 ? výraz2 : výraz3
```

Je-li `výraz1` pravdivý, potom je hodnota celého podmíněného výrazu hodnotou `výraz2`. Jinak je hodnota celého výrazu hodnotou `výraz3`. Zde jsou dva příklady, které ukazují, jak operátor pracuje:

```
5 > 3 ? 10 : 12  // 5 > 3 je pravda, tak je hodnota výrazu 10
3 == 9? 25 : 18  // 3 == 9 je nepravda, tak je hodnota výrazu 18
```

První příklad můžeme přepsat tímto způsobem: Je-li 5 větší než 3, výraz se vyhodnotí na 10; jinak se vyhodnotí na 12. V reálných programovacích situacích by výrazy samozřejmě obsahovaly proměnné.

Výpis programu 6.9 condit.cpp

```
#include <condefs.h>
// condit.cpp — použití podmíněného operátoru
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout << "Zadejte dve cela cisla: ";
    cin >> a >> b;
    cout << "Vetsi z " << a << " a " << b;
    int c = a > b ? a : b;    // c = a jestliže a > b, jinak c = b
    cout << " je " << c << "\n";
    return 0;
}
```

Zde je vzorek běhu programu:

```
Zadejte dve cela cisla: 25 27
Vetsi z 25 ad 27 je 27
```

Klíčovou částí programu je tento příkaz:

```
int c = a > b ? a : b;
```

Produkuje stejný výsledek, jako následující příkazy:

```
int c;
if (a > b)
      c = a;
else
      c = b;
```

Ve srovnání s uspořádáním `if else` je podmíněný operátor mnohem stručnější, ale méně zřejmý. Jeden rozdíl mezi oběma přístupy je, že podmíněný operátor produkuje výraz a odtud jedinou hodnotu, která se může přiřadit nebo začlenit do většího výrazu, tak jak to program udělal, když přiřadil hodnotu podmíněného výrazu do proměnné `c`. Stručný tvar podmíněného operátoru, neobvyklá syntaxe a celkově tajemný vzhled, ho dělají velkým favoritem mezi programátory, kteří tyto kvality oceňují. Jedním z oblíbených triků pro cíl zatajení účelu programového kódu, který zasluhuje pokárání, je vnořit podmíněný operátor do jiného, jak ukazuje slabý následující příklad:

```
const char x[2] [20] = {"Jason ","k vasim sluzbam\n"};
const char * y = "Quillstone ";

for (int i = 0; i < 3; i++)
      cout << ((i < 2)? !i ? x [i] : y : x[1]);
```

To je pouze zatemnění (ale nikterak maximální zatemnění) způsobu tisku tří řetězců v následujícím pořadí:

```
Jason Quillstone k vasim sluzbam
```

Příkaz `switch`

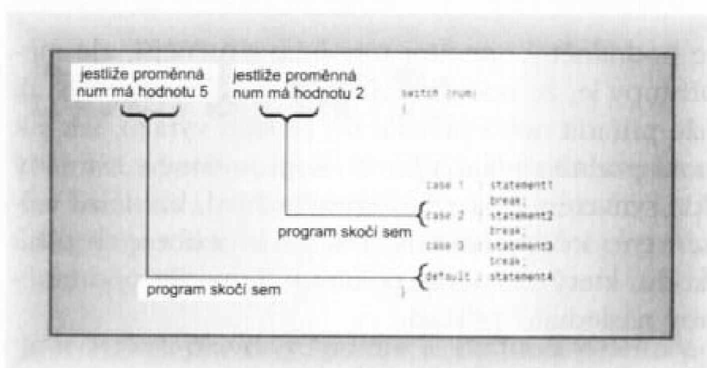
Předpokládejme, že vytváříte obrazovkové menu, které požádá uživatele, aby vybral jednu z pěti voleb, například Levný, Průměrný, Drahý, Marnivý a Přehnaný. Abyste mohli zacházet s těmito pěti alternativami, můžete rozšířit sekvenci `if else if else`, ale příkaz `switch` v C++ pracuje s výběrem možností z rozšířeného seznamu mnohem snadněji. Zde je obecný tvar příkazu `switch`:

```
switch (celočíselný_výraz)
{
      case návěští1 : příkaz(y)
      case návěští2 : příkaz(y)
     ...
      default     : příkaz(y)

}
```

Příkaz `switch` v C++ funguje jako směrovací zařízení, které říká počítači, jaký řádek programového kódu má dále provést. Při čtení `switch` skáče program na řádek, který je označen

hodnotou odpovídající hodnotě celočíselný_výraz. Například, má-li celočíselný_výraz hodnotu 4, program jde na řádek, který má návěští case 4:. Hodnota celočíselný_výraz, jak jméno napovídá, musí být výrazem, který produkuje celočíselnou hodnotu. Rovněž každé návěští musí být celočíselným konstantním výrazem. Nejčastěji jsou návěští jednoduché konstanty int nebo char, jako jsou například 1 nebo q, nebo enumerátory. Jestliže celočíselný_výraz neodpovídá žádnému z návěští, program přeskočí na řádek označený default. Návěští default je nepovinné. Vynecháte-li ho a neexistuje-li žádná shoda, program přeskočí na další příkaz, který následuje za switch. Viz obrázek 6.3.



Obrázek 6.3 Příkaz switch

Příkaz switch se liší velmi důležitým způsobem od podobných příkazů v jazycích jako je například Pascal. Každé návěští case funguje pouze jako návěští řádku, nikoli jako hranice mezi volbami. To jest, jakmile program přeskočí na určitý řádek ve switch, potom sekvenčně provádí všechny příkazy, které tento řádek následují, dokud ho jinak nepřesměrujete. Vykonávání se automaticky NEZASTAVÍ na dalším case. Abyste zastavili na konci určité skupiny příkazů, musíte použít příkaz break. To způsobí, že provádění přeskočí na příkaz následující za switch.

Výpis programu 6.10 ukazuje, jak se switch a break společně implementují do jednoduchého menu pro manažery. Program používá na zobrazení sady možností funkci showmenu(). Příkaz switch potom vybírá činnost závislou na uživatelově odpovědi.

Kompatibilita:

Některé implementace nakládají s escape sekvencí \a mlčky.

Výpis programu 6.10 switch.cpp

```
#include <condefs.h>
// switch.cpp - použití příkazu switch
#include <iostream>
using namespace std;
void showmenu();    // prototypy funkcí
void report();
void comfort();
```

```
int main()
{
    showmenu();
    int choice;
    cin >> choice;
    while (choice != 5)
    {
        switch(choice)
        {
            case 1  :  cout << "\a\n";
                        break;
            case 2  :  report();
                        break;
            case 3  :  cout << "Vedouci byl pritomen cely den.\n";
                        break;
            case 4  :  comfort();
                        break;
            default :  cout << "To neni volba.\n";
        }
        showmenu();
        cin >> choice;
    }
    cout << "Sbohem!\n";
    return 0;
}

void showmenu()
{
    cout << "Prosim, zadejte 1, 2, 3, 4, or 5:\n"
            "1) poplach             2) zprava\n"
            "3) vymluva             4) uklidneni\n"
            "5) ukonceni\n";
}
void report()
{
    cout << "To byl vytecny tyden pro obchod.\n"
    "Tzby vzrostly na 120%. Vydaje poklesly o 35%.\n";
}
void comfort()
{
    cout << "Vasi zamestnanci si mysli, ze jste nejlepsi vykonny redi-
tel\n"
            "prumyslu. Spravni rada si mysli,\n"
            "ze jste nejlepsi vykonny reditel v prumyslu.\n";
}
```

Zde je vzorek běhu programu s menu pro manažery:

```
Prosim, zadejte 1, 2, 3, 4, or 5:
1) poplach        2) zprava
3) vymluva        4) uklidneni
5) ukonceni
```

```
4
Vasi zamestnanci si mysli, ze jste nejlepsi vykonny reditel
prumyslu. Spravni rada si mysli,
ze jste nejlepsi vykonny reditel v prumyslu.
Prosim, zadejte 1, 2, 3, 4, or 5:
1) poplach          2) zprava
3) vymluva          4) uklidneni
5) ukonceni
2
To byl vytecny tyden pro obchod.
Tzby vzrostly na 120%. Vydaje poklesly o 35%.
Prosim, zadejte 1, 2, 3, 4, or 5:
1) poplach          2) zprava
3) vymluva          4) uklidneni
5) ukonceni
6
To neni volba.
Prosim, zadejte 1, 2, 3, 4, or 5:
1) poplach          2) zprava
3) vymluva          4) uklidneni
5) ukonceni
5
Sbohem!
```

Cyklus `while` se ukončí, když uživatel zavede 5. Zavedení čísel 1 až 4 aktivuje odpovídající volbu ze seznamu `switch` a zavedení čísla 6 spustí standardní příkazy.

Jak bylo poznamenáno dříve, program potřebuje příkazy `break`, aby omezil vykonávání programu na určitou část příkazu switch. Abyste viděli, že to tak funguje, odstraňte z výpisu programu 6.10 příkazy `break` a podívejte se, jak bude pracovat. Například zjistíte, že zavedení čísla 2 zapříčiní, že program provede *všechny* příkazy přičleněné k návěštím 2, 3, 4 a standardnímu návěští case. C++ pracuje tímto způsobem, protože takový druh chování může být užitečný. Předpokládejme například, že jste přepsali výpis programu 6.10 pomocí znaků celá čísla ve volbách menu a návěštích přepínače. Potom byste mohli použít v návěštích jak malých tak velkých písmen pro stejné příkazy:

```
char choice;
cin >> choice;
while (choice != 'Q' && choice != 'q')
{
    switch(choice)
    {
        case 'a':
        case 'A': cout << "\a\n";
                  break;
        case 'r':
        case 'R': report();
                  break;
        case 'l':
        case 'L': cout << "Sef byl pritomen cely den.\n";
                  break;
        case 'c'
```

```
        case 'C': comfort();
                  break;
        default : cout << "To neni volba.\n";
    }
    showmenu();
    cin >> choice;
}
```

Protože bezprostředně za `case 'a'` neexistuje žádný `break`, vykonávání programu přechází na další řádek, což je následující příkaz `case 'A'`.

Použití enumerátorů jako návěští

Výpis programu 6.11 používá na definování skupiny příbuzných konstant klíčové slovo `enum` a potom je používá v příkazu `switch`. Obvykle objekt `cin` nerozpoznává výčtové typy (nemůže vědět, jak je budete definovat), takže program přečte volbu jako typ `int`. Když příkaz `switch` srovnává hodnotu typu `int` s enumerátorem návěští `case`, konvertuje enumerátor na typ `int`. Výčtové typy se také konvertují na typ `int` v testovací podmínce cyklu `while`.

Výpis programu 6.11 `enum.cpp`

```
#include <condefs.h>
// enum.cpp — použití enum
#include <iostream>
using namespace std;
// vytvoření pojmenovaných konstant pro 0 - 6
enum {cervena, oranzova, zluta, zelena, modra, fialova, indigova};

int main()
{
    cout << "Zadejte kod barvy (0-6): ";
    int code;
    cin >> code;
    while (code >= cervena && code <= indigova)
    {
        switch (code)
        {
            case cervena  : cout << "Jeji rty byly cervene.\n"; break;
            case oranzova : cout << "Jeji vlasy byly oranzove.\n"; break;
            case zluta    : cout << "Jeji boty byly zlute.\n"; break;
            case zelena   : cout << "Jeji nehty byly zelene.\n"; break;
            case modra    : cout << "Jeji atleticky ubor byl mo-
dry.\n"; break;
            case fialova  : cout << "Jeji oci byly fialove.\n"; break;
            case indigova : cout << "Jeji nalada byla indigova.\n"; bre-
ak;
        }
        cout << "Zadejte kod barvy (0-6): ";
        cin >> code;
    }
```

```
    cout << "Sbohem\n";
    return 0;
}
```

Zde je ukázka výstupu:

```
Zadejte kod barvy (0-6): 3
Jeji nehty byly zelene.
Zadejte kod barvy (0-6): 5
Jeji oci byly fialove.
Zadejte kod barvy (0-6): 2
Jeji boty byly zlute.
Zadejte kod barvy (0-6): 8
Sbohem
```

Příkazy `switch` a `if else`

Jak příkaz `switch` tak `if else` umožňují programu vybrat ze seznamu alternativ. Příkaz `if else` je z obou všestrannější. Například s rozsahy může zacházet následujícím způsobem:

```
if (age > 17 && age < 35)
        index = 0;
    else if (age >= 35 && age < 50)
        index = 1;
    else if (age >= 50 && age < 65)
        index = 2;
    else
        index = 3;
```

Příkaz `switch` však na zacházení s rozsahy není navržen. Každé návěští case příkazu `switch` musí být jediná hodnota. Ale také tato hodnota musí být celočíselného typu (který zahrnuje `char`), takže `switch` nemůže zacházet s testy v pohyblivé řádové čárce. A hodnota návěští case musí být konstanta. Pokud vaše alternativy zahrnují rozsahy nebo testy v pohyblivé řádové čárce nebo porovnání dvou hodnot, použijte příkaz `if else`.

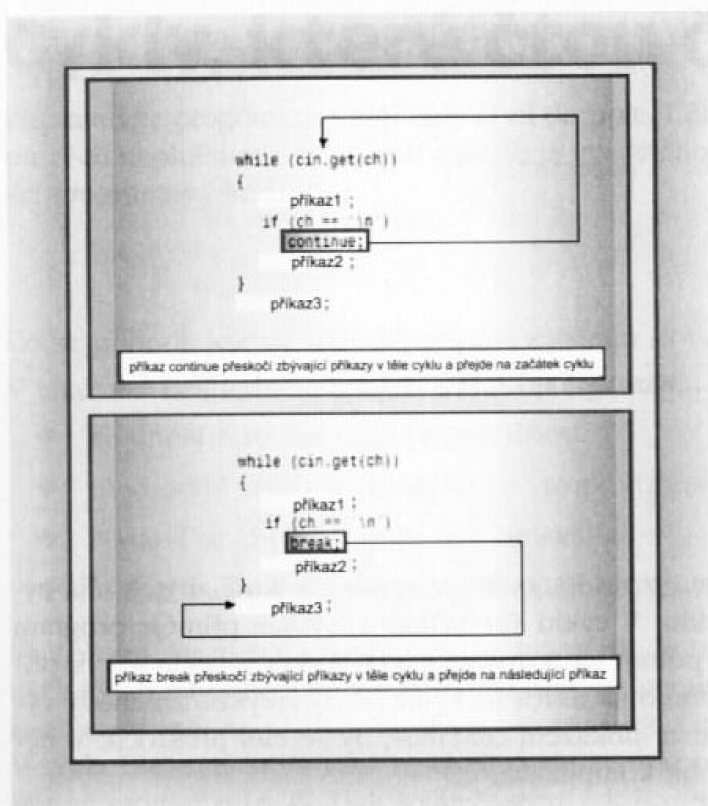
Jestliže se však všechny alternativy mohou identifikovat pomocí celočíselných konstant, můžete použít příkaz `switch` nebo `if else`. Protože toto je přesně ta situace, pro kterou je příkaz `switch` navržen, je obvykle efektivnější volbou ve smyslu velikosti programového kódu a rychlosti provádění, pokud neexistuje pouze jediný pár alternativ, ze kterých se má vybírat.

Tip:

Můžete-li použít jak sekvenci `if else`, tak příkaz `switch`, obvyklé pravidlo pro použití příkazu `switch` je, když máte tři nebo více alternativ.

Příkazy `break` a `continue`

Příkazy `break` a `continue` umožňují programu přeskočit část programového kódu. Příkaz `break` můžete použít v příkazu `switch` nebo v libovolném z cyklů. To způsobí, že se vykonávání programu přenese na další příkaz, který následuje za příkazem `switch` nebo za cyklem. Příkaz `continue` se používá v cyklech a způsobuje, že program přeskočí zbytek těla cyklu a začíná novým průběhem cyklu. Viz obrázek 6.4.



Obrázek 6.4 Příkazy break a continue

Výpis programu 6.12 ukazuje, jak tyto dva příkazy pracují. Program vám umožní zavést textový řádek. Cyklus zobrazí každý znak a na ukončení cyklu, jestliže je znak tečka, používá příkaz `break`. Ukazuje, jak můžete použít příkaz `break` na ukončení cyklu z nitra jisté podmínky, která se stane pravdivou. Dále program počítá mezery, ale nikoli další znaky. Cyklus používá příkaz `continue` na přeskočení zbývající části cyklu, jakmile znak není mezera.

Výpis programu 6.12 jump.cpp

```
#include <condefs.h>
// jump.cpp - použiti continue a break
#include <iostream>
using namespace std;
const int ArSize = 80;
int main()
{
```

```
    char line[ArSize];
    int spaces = 0;
    cout << "Zadejte textovy radek:\n";
    cin.get(line, ArSize);
    for (int i = 0; line[i] != '\0'; i++)
    {
        cout << line[i];     // zobrazi znak
        if (line[i] == '.') // ukončení, když je tečka
            break;
        if (line[i] != ' ') // přeskočení zbytku příkazů v cyklu
            continue;
        spaces++;
    }
    cout << "\n" << spaces << " mezer\n";
    return 0;
}
```

Zde je ukázka běhu programu:

```
Zadejte textovy radek:
Nechte me dnes zaplatit obed. Muzete zaplatit.
Nechte me dnes zaplatit obed.
4 mezer
```

Poznámky k programu

Všimněte si, že zatímco příkaz `continue` způsobuje, že program přeskočí zbytek těla cyklu, nepřeskakuje změnový výraz cyklu. V cyklu `for` příkaz `continue` přiměje program přeskočit přímo na změnový výraz a potom na testovací výraz. Avšak v cyklu `while` příkaz `continue` přiměje program přejít přímo na testovací výraz. Tedy jakýkoli změnový výraz v těle cyklu `while`, který následuje za příkazem `continue`, by se měl přeskočit. V některých případech by to mohlo způsobit komplikace.

Tento program nemusel použít příkaz `continue`. Místo toho by mohl použít tento programový kód:

```
if (line[i] == ' ')
    spaces++;
```

Avšak příkaz `continue` může způsobit, že je program čitelnější, když za `continue` následuje několik příkazů. Tímto způsobem nemusíte učinit všechny tyto příkazy částí příkazu `if`.

C++, podobně jako C, má příkaz `goto`. Příkaz jako

```
goto paris
```

znamená přeskočení na místo, které nese `paris:` jako návěští. To jest, můžete mít následující programový kód:

```
char ch;
cin >> ch;
if (ch == 'P')
    goto paris;
```

```
cout < ...
...
paris: cout << "Prave jste prijel do Parize.\n";
```

Ve většině situací je použití příkazu `goto` nevhodné (bad hack) a měli byste na řízení průběhu programu používat strukturované řídící příkazy, jako například `if else`, `switch`, `continue` a podobně.

Cykly, které čtou čísla

Připravujete program na čtení řady čísel do pole. Uživateli chcete umožnit ukončení vstupu před naplněním pole. Jeden způsob je využít chování objektu `cin`. Uvažujme následující programový kód:

```
int n;
cin >> n;
```

Co se přihodí, jestliže uživatel odpoví zadáním slova místo čísla?

V takovém nevhodném spojení nastávají čtyři věci:

- Hodnota `n` se ponechá nezměněnou.
- Nevhodný vstup se ponechá ve vstupní frontě.
- Nastaví se chybový příznak v objektu `cin`.
- Volání metody objektu `cin` navrací false, jestliže se konvertuje na typ `bool`.

Skutečnost, že metoda navrací `false` znamená, že můžete použít na ukončení počtu čtených cyklů nenumerický vstup. Skutečnost, že nenumerický vstup nastavuje chybový příznak znamená, že příznak musíte resetovat před tím, než můžete číst další vstupní data. Metoda `clear()`, která také resetuje podmínku konec-souboru (viz kapitola 5), resetuje špatný vstupní příznak. (Jak špatný vstup, tak konec souboru mohou způsobit, že objekt `cin` navrací `false`. Kapitola 16, „Vstup, výstup a soubory", pojednává o tom, jak tyto dva případy rozlišit.) Podívejme se na pár příkladů, které tyto postupy ilustrují.

Chcete napsat program na výpočet průměrné hmotnosti vašeho denního úlovku ryb. Úlovek pěti ryb je limit, takže pole o pěti prvcích může obsahovat všechny údaje, ale je možné, že byste mohli chytit méně ryb. Výpis programu 6.13 používá cyklus, který končí, když je pole plné, nebo když jste zavedli nenumerický vstupní údaj.

Výpis programu 6.13 cinfish.cpp

```
#include <condefs.h>
// cinfish.cpp — nenumerický vstup ukončuje cyklus
#include <iostream>
using namespace std;
const int Max = 5;
int main()
{
    // získání dat
```

```
    double fish[Max];
    cout << "Prosim, zadejte vahy vasich ryb.\n";
    cout << "Muzete zadat az " << Max
         << " ryb <q pro ukonceni>.\n";
    cout << "ryba #1: ";
    int i = 0;
    while (i < Max && cin >> fish[i]) {
         if (++i < Max)
              cout << "ryba #" << (i+1) << ": ";
    }
    // výpočet průměru
    double total = 0.0;
    for (int j = 0; j < i; j++)
         total += fish[j];
    // zobrazení výsledků
    if (i == 0)
         cout << "Zadna ryba\n";
    else
         cout << total / i << " = prumerna vaha "
              << i << " ryb\n";
    return 0;
}
```

Kompatibilita:

Některé kompilátory od Borlandu dávají varovnou zprávu u příkazu

```
cout << "ryba #" << (i+1) << ": ";
```

kvůli tomu, že u nejednoznačných operátorů je potřeba použít závorky. Nelamte si s tím hlavu. Pouze vás upozorňují na možnou chybu, pokud by byl operátor << použitý ve svém původním významu jako operátor posunu vlevo.

Výraz `cin >> fish[i]` je ve skutečnosti metoda funkčního volání objektu `cin` a funkce navrací objekt `cin`. Jestliže je `cin` částí testovací podmínky, konvertuje se na typ `bool`. Konvertovaná hodnota je `true`, je-li vstup úspěšný, jinak je `false`. Hodnota výrazu `false` ukončuje cyklus. Mimochodem, zde je ukázka běhu programu:

```
Prosim, zadejte vahy vasich ryb.
Muzete zadat az 5 ryb <q pro ukonceni>.
ryba #1: 30
ryba #2: 35
ryba #3: 25
ryba #4: 40
ryba #5: q
32.5 = prumerna vaha 4 ryb
```

Všimněte si následujícího řádku programového kódu:

```
while (i < Max && cin >> fish[i]) {
```

Připomínáme, že C++ nevyhodnocuje pravou stranu logického výrazu AND, jestliže je levá strana nepravdivá. V tomto případě vyhodnocení pravé strany znamená použití objektu `cin` na umístění vstupních dat do pole. Jestliže se `i` rovná `Max`, cyklus se ukončí bez pokusu vložit hodnotu na místo za koncem pole.

Poslední příklad se nepokusil po zadání nenumerického vstupu přečíst žádné vstupní údaje. Podívejme se na příklad, který je čte. Předpokládejme, že jste požádáni do programu v C++ dodat přesně pět výsledků z golfu na stanovení vašeho průměrného skóre. Jestliže uživatel zavede nenumerický vstup, program by měl mít námitky a trvat na numerickém vstupu. Jak jste viděli, můžete na otestování nenumerického vstupu použít hodnotu vstupního výrazu `cin`. Předpokládejme, že jste zjistili, že uživatel zavedl chybný údaj. Potřebujete podniknout tři kroky:

- Resetovat objekt `cin` pro přijetí nových vstupních dat.
- Zbavit se chybných vstupních dat.
- Vyzvat uživatele, aby se pokusil znovu.

Všimněte si, že musíte objekt `cin` resetovat před tím, než se zbavíte špatných vstupních dat. Výpis programu 6.14 ukazuje, jak se mohou tyto úkoly uskutečnit.

Výpis programu 6.14 cingolf.cpp

```
#include <condefs.h>
// cingolf.cpp - nenumerický vstup se přeskočí
#include <iostream>
using namespace std;
const int Max = 5;
int main()
{
    // get data
    int golf[Max];
    cout << "Prosim, zadejte vase skore v golfu.\n";
    cout << "Musite zadat " << Max << " her.\n";
    int i;
    for (i = 0; i < Max; i++)
    {
        cout << "kolo #" << (i+1) << ": ";
        while (!(cin >> golf[i])) {
            cin.clear();    // obnovení vstupu
            while (cin.get() != '\n')
                continue;   // odhození chybných vstupních dat
            cout << "Prosim, zadejte cislo: ";
        }
    }
    // výpočet průměru
    double total = 0.0;
    for (i = 0; i < Max; i++)
        total += golf[i];
    // zobrazení výsledků
    cout << total / Max << " = prumerne skore "
```

```
        << Max << " kol\n";
    return 0;
}
```

Zde je ukázka běhu programu:

```
Prosim, zadejte vase skore v golfu.
Musite zadat 5 her.
kolo #1: 88
kolo #2: 87
kolo #3: musim?
Prosim, zadejte cislo: 103
kolo #4: 94
kolo #5: 86
91.6 = prumerne skore 5 kol
```

Poznámky k programu

Srdce programového kódu na zpracování chyb je následující:

```
while (!(cin >> golf[i])) {
    cin.clear();      // obnoveni vstupu
    while (cin.get() != '\n')
        continue; // odhození chybných vstupních dat
    cout << " Prosim, zadejte cislo: ";
}
```

Jestliže uživatel zavede 88, výraz `cin` je `true` a hodnota se umístí do pole, výraz `!(cin >> golf[i])` je `false` a vnitřní cyklus se ukončí. Ale jestliže uživatel zavede `musim?`, výraz `cin` je `false`, do pole se nic neumístí, výraz `!(cin >> golf[i])` je `true` a program vstoupí do vnitřního cyklu `while`. První příkaz v cyklu používá na resetování vstupu metodu `clear()`. Jestliže tento příkaz vynecháte, program odmítne číst další vstupní data. Dále program používá v cyklu `while` na přečtení zbývajících vstupních dat do konce řádku metodu `cin.get()`. Ta se zbavuje chybných vstupních dat spolu se vším, co je na řádku. Jiný přístup je čtení až do dalšího oddělovače, který by mohl zbavit chybný vstup jednoho slova najednou, místo najednou celé řádky. Nakonec program požádá uživatele o zadání čísla.

Shrnutí

Programy a programování se stávají zajímavější, když zavedete příkazy, které provázejí program alternativními činnostmi. (Otázku, zda to také způsobí, že se o ně bude programátor více zajímat, jsme plně neprozkoumali.) C++ poskytuje příkazy `if`, `if else` a `switch` jako prostředky na řízení výběrů. Příkaz `if` vám v C++ dovolí podmíněně provádět příkaz nebo příkazový blok, to znamená, že program vykoná příkaz nebo blok, je-li splněna určitá podmínka. Příkaz `if else` v C++ umožní programu vybrat ze dvou voleb, který příkaz nebo příkazový blok má provést. K příkazu na vyjádření skupiny voleb můžete přidat dodatečné příkazy `if else`. Příkaz `switch` v C++ směruje program k určitému případu ze seznamu voleb.

C++ také poskytuje na pomoc v rozhodování operátory. Kapitola 5 pojednává o relačních výrazech, které porovnávají dvě hodnoty. Příkazy if a if else používají na testování podmínek typicky relační výrazy. Použitím relačních operátorů (&&, || a !) v C++ můžete spojovat nebo modifikovat relační výrazy a vytvářet komplikovanější testy. Podmíněný operátor (?:) poskytuje zhuštěný způsob volby ze dvou hodnot.

Knihovna znakových funkcí cctype poskytuje vhodnou a účinnou sadu prostředků na analýzu vstupních znakových dat.

Spolu s cykly a příkazy, umožňujícími rozhodnutí, máte v C++ prostředky na psaní zajímavých, inteligentních a účinných programů. My jsme pouze začali zkoumat reálnou sílu C++. Dále se podíváme na funkce.

Opakovací otázky

1. Uvažujte dva následující fragmenty programového kódu na spočítání mezer a nových řádků:

```
// Version 1
while (cin.get(ch))    // ukončit při eof
{
        if (ch == ' ')
                spaces++;
        if (ch == '\n')
               newlines++;
}
// Version 2
while (cin.get(ch))    // ukončit při eof
{
        if (ch == ' ')
               spaces++;
        else if (ch == '\n')
               newlines++;
}
```

   Jaké výhody, jsou-li nějaké, má druhý tvar před prvním?

2. Jaký účinek má nahrazení výrazu ++ch výrazem ch+1 ve výpisu programu 6.2?
3. Promyslete si pečlivě následující pogram:

```
#include <iostream>
using namespace std;
int main()
{
     char ch;
     int ct1, ct2;

     ct1 = ct2 = 0;
     while ((ch = cin.get()) != '$')
     {
          cout << ch;
```

```
        ct1++;
        if (ch = '$')
            ct2++;
        cout << ch;
    }
    cout <<"ct1 = " << ct1 << ", ct2 = " << ct2 << "\n";
    return 0;
}
```

Předpokládejte, že poskytneme následující vstupní data, kde o představuje stisknutí ENTER:

```
Ahoj!o
Posli nyni $10 nebo $20 !o
```

Jaký je výstup? (Připomínáme, že vstupní data jsou ve vyrovnávací paměti.)

4. Vytvořte na znázornění následujících podmínek logické výrazy :
   a) `Weight` je větší nebo rovno 115, ale menší než 125.
   b) `Ch` je `q` nebo `Q`.
   c) `X` je sudé, ale není `26`.
   d) `Donation` je v rozmezí 1000–2000 nebo `guest` je `1`.
   e) Ch je malé nebo velké písmeno (předpokládejte, že se jak malá, tak velká písmena kódují sekvenčně, ale mezi malými a velkými písmeny existuje mezera).
5. V anglické větě „I will not not speak" znamená to samé jako „I will speak". Je v C++ `!!x` to samé jako `x`!?
6. Vytvořte podmíněný výraz, který je roven absolutní hodnotě proměnné. To jest, jestliže je proměnná `x` kladná, hodnota výrazu je právě `x`, je-li záporná, hodnota výrazu je `-x`, což je kladné.
7. Přepište následující fragment pomocí příkazu `switch`:

```
if (ch == 'A')
        a_grade++;
    else if (ch == 'B')
        b_grade++;
    else if (ch == 'C')
        c_grade++;
    else if (ch == 'D')
        d_grade++;
    else
        f_grade++;
```

8. Jaká by byla výhoda při použití znakových návěští, jako například `a` a `c` namísto čísel ve volbách menu a případech příkazu `switch` ve výpisu programu 6.10? (Rada: Přemýšlejte o tom, co se přihodí, když uživatel zadá `q` v každém případě a co se stane, když zadá 5.)
9. Uvažujte následující fragment programového kódu:

```
int line = 0;
char ch;
```

```
while (cin.get(ch))
{
    if (ch == 'Q')
            break;
    if (ch != '\n')
            continue;
    line++;
}
```

Přepište tento kód pomocí příkazů `break` nebo `continue`.

Programovací cvičení

1. Napište program, který čte vstupní data z klávesnice až po symbol `@`, a který zobrazuje vstupní data kromě číslic a konvertuje malé znaky na velké a naopak. (Nezapomeňte na skupinu `cctype`.)
2. Napište program, který čte až 10 darovaných hodnot do pole typu `double`. Program by měl ukončit vstup při zadání nenumerické hodnoty. Měl by vydat zprávu o průměru z čísel a také kolik čísel v poli je větších než průměr.
3. Napište předchůdce k programu řízeného pomocí menu. Program by měl zobrazit menu, které nabízí čtyři volby, každou označenou písmenem. Odpoví-li uživatel jiným písmenem než jsou tyto čtyři platné volby, program by ho měl vyzvat, aby zadával platnou odpověď, dokud se nepodrobí. Potom by měl program použít přepínač na výběr jednoduché akce založené na uživatelově výběru. Běh programu by mohl vypadat podobně jako:

```
Prosim, zadejte jednu z nasledujicich voleb:
m) masozravec          p) pianista
s) strom               h) hra
f
Prosim, zadejte m, p, s, nebo h: k
Please enter c, p, t, or g: s
Javor je strom.
```

4. Když se připojíte k Dobročinné skupině programátorů, můžete být znám na setkáních DSP podle svého skutečného jména, názvu zaměstnání, tajného jména nebo členské volby. Postavte program na následující struktuře:

```
// jméno struktury Dobročinná skupina programátorů
struct bop {
    char fullname[strsize]; // skutečné jméno
    char title[strsize];    // název zaměstnání
    char bopname[strsize];  // tajné jméno DSK
    int preference;       // 0 = plné jméno, 1 = zaměstnání, 2 = jméno DSK
};
```

V programu vytvořte malé pole takových struktur a inicializujte ho vhodnými hodnotami. Dovolte programu ať vykoná cyklus, který nechá uživatele vybrat z různých alternativ:

```
a. zobrazeni podle jmena        b. zobrazeni podle zaměstnání
c. zobrazeni podle tajné jméno  d. zobrazeni podle volby
k. konec
```

Ukázkový běh programu může vypadat něco jako následující:

```
Oznameni Dobrocinne skupiny programatoru
a. zobrazeni podle jmena        b. zobrazeni podle titulu
c. zobrazeni podle tajné jméno  d. zobrazeni podle volby
k. konec
Zadejte svou volbu: a
Wimp Macho
Raki Rhodes
Celia Laiter
Hoppy Hipman
Pat Hand
Další volba: d
Wimp Macho
Junior Programmer
MIPS
Analyst Trainee
LOOPY
Další volba: q
Ahoj!
```

5. Království Neutronie, kde je jednotkou měny tvarp, má následující zásadu pro daň z příjmu:

prvních	5000 tvarpů:	0% daň
dalších	10000 tvarpů:	10% daň
dalších	20000 tvarpů:	15% daň
tvarpy nad	35000:	20% daň

Například někdo, kdo vydělá 38000 tvarpů, by měl dlužit 5000 x 0.00 + 10000 x 0.10 + 20000 x 0.15 + 3000 x 0.20, neboli 4600 tvarpů. Cyklus končí, když uživatel zavede záporné číslo nebo nenumerický vstup.

Funkce – programové moduly v C++

KAPITOLA 7

Témata kapitoly:

Základy funkcí (přehled)

Funkční prototypy

Předávání parametrů do funkce hodnotou

Navrhování funkcí na zpracování polí

Použití konstantních ukazatelů jako parametrů

Navrhování funkcí na zpracování textových řetězců

Navrhování funkcí na zpracování struktur

Funkce, které volají samy sebe (rekurze)

Ukazatele na funkce

Veselá zábava je tam, kde ji hledáte. Podívejte se zblízka a můžete ji najít ve funkcích. C++ poskytuje velikou knihovnu užitečných funkcí (standardní knihovna ANSI C a několik tříd v C++), ale skutečné programové potěšení nastává při psaní vašich vlastních funkcí. V této a následující kapitole vyzkoušíte, jak se funkce definují, jak se jim předává informace a jak z nich informaci získáte. Po seznámení s tím jak funkce pracují se kapitola soustředí na to, jak se používají ve spojení s poli, řetězci a strukturami. Nakonec se dotkne rekurze a ukazatelů na funkce. Jestliže jste věnovali pozornost C, zjistíte, že mnoho z této kapitoly je vám důvěrně známo. Ale neuklidněte se vědomím odborných znalostí. C++ má několik dodatků o tom, co mohou funkce dělat a následující kapitola o nich primárně pojednává. Mezitím dohlédneme na podstatu.

Přehled funkcí

Za prvé shrňme, co jste se již o funkcích naučili. Abyste mohli používat funkce, musíte udělat následující:

- Poskytněte definici funkce.
- Poskytněte prototyp funkce.
- Vyvolejte funkci.

Jestliže používáte knihovní funkci, funkce již pro vás byla definována a překompilována. Můžete tedy na obstarání prototypu použít standardní knihovní hlavičkový soubor. Vše, co zbývá udělat, je vyvolat řádně funkci. Příklady v této knize to několikrát dělaly. Například standardní knihovna C zahrnuje funkci `strlen()` k nalezení délky řetězce. Přidružený standardní hlavičkový soubor `cstring` obsahuje pro funkci `strlen()` a několik dalších funkcí, které ma-

jí vztah k řetězcům, funkční prototyp. Tato předem zaplacená práce vám umožní ve vašich programech bez dalších starostí použít funkci `strlen()`.

Avšak když vytváříte své vlastní funkce, musíte se sami vypořádat se všemi třemi aspekty – definováním, vytvořením prototypů a voláním. Výpis programu 7.1 tyto kroky ukazuje na krátkém příkladu.

Výpis programu 7.1 calling.cpp

```
#include <condefs.h>
// calling.cpp — definování, vytvoření prototypu a vyvolání funkce
#include <iostream>
using namespace std;

void simple();     // prototyp funkce

int main()
{
    cout << "main() bude volat funkci simple():\n";
    simple();          // volání funkce
    return 0;
}

// definice funkce
void simple()
{
    cout << "Jsem to ale jednoducha funkce.\n";
}
```

Zde je výstup:

```
main() bude volat funkci simple():
Jsem to ale jednoducha funkce.
```

Podívejme se na tyto kroky mnohem podrobněji.

Definování funkce

Funkce můžete seskupit do dvou kategorií: funkce, které nemají návratové hodnoty a funkce, které je mají. Funkce bez návratových hodnot se nazývají funkce typu `void` a mají následující obecný tvar:

```
void jménoFunkce(seznamParametrů)
{
    příkaz(y)
    return;          // nepovinné

}
```

Zde `seznamParametrů` určuje typ a počet parametrů předaných funkci. Tato kapitola podrobněji zkoumá tento seznam později. Nepovinný příkaz return označuje konec

funkce. Jinak funkce končí uzavírací složenou závorkou. Funkce typu `void` odpovídají procedurám v Pascalu, subroutinám ve FORTRANu a moderním procedurám subprogram v BASICu. Funkce `void` typicky používáte k provedení nějakého typu činnosti. Například funkce na vytištění pozdravu `Tak ahoj!` (n) krát může vypadat takto:

```
void cheers(int n)              // žádná návratová hodnota
{
    for (int i = 0; i < n; i++)
        cout << "Tak ahoj! ";
    cout << "\n";
}
```

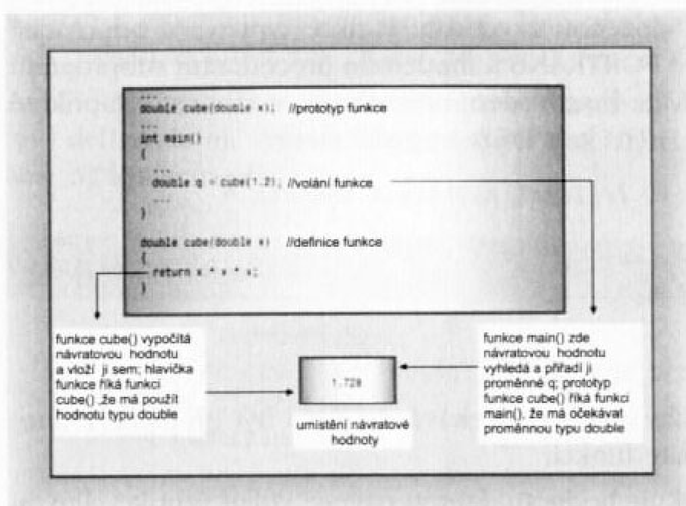
Parametr `int n` v seznamu znamená, že `cheers()` očekává, že jí má být předána hodnota typu `int` jako parametr, když zavoláte funkci.

Funkce s návratovou hodnotou produkuje hodnotu, kterou navrací volající funkci. Jinými slovy, navrací-li funkce druhou odmocninu z 9.0 (`sqrt(9.0)`), potom je funkční volání nahrazeno hodnotou 3.0. Taková funkce se deklaruje se stejným typem, jako je typ hodnoty, kterou navrací. Zde je obecný tvar:

```
jménoTypu JménoFunkce(seznamParametrů)
{
      příkazy
      return hodnota;   // hodnota má stejný typ jako jménoTypu
}
```

Funkce s návratovými hodnotami vyžadují používání příkazu return tak, že se hodnota vrací volající funkci. Samotná hodnota může být konstanta, proměnná nebo mnohem obecněji výraz. Jediný požadavek je, že se výraz redukuje na hodnotu, která má typ `jménoTypu` nebo se dá na něj konvertovat. (Je-li, řekněme, deklarovaný návratový typ `double` a funkce navrací výraz typu `int`, hodnota `int` se přetypuje na typ `double`.) Funkce potom navrací konečnou hodnotu volající funkci. C++ klade omezení na typy, které můžete používat pro návratovou hodnotu. Návratová hodnota nemůže být pole. Vše ostatní je možné – celá čísla, čísla v pohyblivé řádové čárce, ukazatele, dokonce struktury a objekty! (Zajímavé, dokonce ačkoli funkce nemůže navracet pole přímo, může navracet pole, které je částí struktury nebo objektu.)

Jako programátor nepotřebujete znát, jak funkce navrací hodnotu, ale znalost metody by vám mohla tuto představu objasnit. (Tedy dává vám něco, o čem můžete se svými přáteli a rodinou hovořit.) Typicky funkce navrací hodnotu jejím nakopírováním do předepsaného registru CPU nebo do paměťového místa. Potom volající program toto místo zkontroluje. Jak vracející, tak volající funkce musí souhlasit s typem dat na této lokaci. Funkční prototyp říká volajícímu programu, co má očekávat a funkční definice říká volanému programu, co má navrátit. Viz obrázek 7.1. Poskytnutí stejné informace jak prototypu tak definici se může jevit jako práce navíc, ale má to svůj dobrý význam. Zajisté, chcete-li, aby kurýr vyzvedl něco z vašeho stolu v kanceláři, zvýšíte pravděpodobnost toho, že se úkol provede správně, jestliže poskytnete popis toho, co chcete jak kurýrovi tak někomu v kanceláři.



Obrázek 7.1 Typický mechanismus pro návrat hodnoty

Funkce se ukončí po provedení příkazu return. Má-li funkce více než jeden příkaz return – například jako alternativy k různým volbám výběrů `if else` – ukončí se po provedení prvního příkazu return, na který dosáhne:

```
int bigger(int a, int b)
{
    if (a > b )
        return a;  // je-li a > b, funkce se ukončí zde
    else
        return b;  // jinak se funkce ukončí zde
}
```

Zde `else` není potřeba, ale pomáhá nahodilému uživateli pochopit záměr.

Funkce s návratovými hodnotami jsou téměř jako funkce v Pascalu, FORTRANu a BASICu. Vracejí hodnotu volajícímu programu, který ji potom může přiřadit proměnné, zobrazit ji nebo ji jinak použít. Zde je jednoduchý příklad, který navrací třetí mocninu hodnoty typu `double`:

```
double cube(double x)      // x krát x krát x
{
    return x * x * x; // hodnota typu double
}
```

Například funkční volání `cube(1.2)` navrací hodnotu 1.728. Všimněte si, že tento návratový příkaz používá výraz. Funkce počítá hodnotu výrazu (v tomto případě `1.728`) a navrací ji.

Vytváření funkčních prototypů a volání funkcí

Nyní jste obeznámeni s vytvořením funkčního volání, ale možná budete méně uspokojeni s vytvářením funkčních prototypů, protože jsou často skryty v začleněných souborech.

Použijme funkce `cheers()` a `cube()` v programu (výpis programu 7.2); všimněte si funkčních prototypů:

Výpis programu 7.2 protos.cpp

```
#include <condefs.h>
// protos.cpp — použití prototypů a volání funkcí
#include <iostream>
using namespace std;
void cheers(int);          // prototyp: žádná návratová hodnota
double cube(double x);     // prototyp: vrací hodnotu typu double
int main(void)
{
    cheers(5);               // volání funkce
    cout << "Dejte mi cislo: ";
    double side;
    cin >> side;
    double volume = cube(side);    // volání funkce
    cout << "Kostka o velikosti strany " << side <<"-stop ma objem ";
    cout << volume << " kubickych stop.\n";
    cheers(cube(2));         // ochrana prototypu v činnosti
    return 0;
}
void cheers(int n)
{
    for (int i = 0; i < n; i++)
    cout << "Tak ahoj! ";
    cout << "\n";
}

double cube(double x)
{
    return x * x * x;
}
```

Zde je ukázka běhu programu:

```
Tak ahoj! Tak ahoj! Tak ahoj! Tak ahoj! Tak ahoj!
Dejte mi cislo: 5
Kostka o velikosti strany 5-stop ma objem 125 kubickych stop.
Tak ahoj! Tak ahoj! Tak ahoj! Tak ahoj! Tak ahoj! Tak ahoj! Tak ahoj! Tak
ahoj!
```

Všimněte si, že `main()` volá funkci `cheers()` typu `void` pomocí jména funkce a parametrů, za kterými následuje středník: `cheers(5);`. To je příklad příkazu volání funkce. Ale protože `cube()` má návratovou hodnotu, `main()` ji může použít jako část přiřazovacího příkazu:

```
double volume = cube(side);
```

Řekli jsme ale, že byste se měli soustředit na prototypy. Co byste měli o prototypech vědět? Za prvé byste měli rozumět tomu, proč je C++ vyžaduje. Potom, protože je C++ vy-

žaduje, byste měli znát jejich správnou syntaxi. Nakonec byste měli ocenit, co pro vás prototyp dělá. Podívejme se po řadě na tyto body pomocí výpisu programu 7.2, jako základu pro diskusi.

Proč prototypy?

Prototyp popisuje rozhraní funkce ke kompilátoru. To jest říká kompilátoru, jakého typu je návratová hodnota, má-li funkce nějakou a říká mu počet a typ parametrů funkce. Uvažujte například, jak prototyp ovlivňuje funkční volání z výpisu programu 7.2:

```
double volume = cube(side);
```

Za prvé prototyp říká kompilátoru, že `cube()` by měla mít jeden parametr typu `double`. Když program selže při poskytnutí parametru, vytvoření prototypu dovolí kompilátoru zachytit chybu. Za druhé, když funkce `cube()` končí svůj výpočet, umístí svou návratovou hodnotu do jisté specifikované lokace – možná do registru CPU nebo do paměti. Potom volající funkce, v tomto případě `main()`, z této lokace hodnotu získá. Protože prototyp stanoví, že `cube()` je typu `double`, kompilátor ví, kolik bajtů má získat a jak je má interpretovat. Bez této informace by kompilátor mohl pouze hádat.

Stále můžete být zvědavi, proč kompilátor potřebuje prototyp? Nemůžete se pouze podívat dále do souboru a vidět jak jsou funkce definované? Jeden problém tohoto přístupu je, že je méně efektivní. Kompilátor by musel `main()` uložit, zatímco by hledal zbytek souboru. Dokonce mnohem vážnějším problémem je skutečnost, že by funkce nesměla být v souboru. C++ vám dovoluje rozšířit program do několika souborů, které můžete nezávisle zkompilovat a potom je později sestavit. Je-li to tento případ, kompilátor možná nemá přístup k funkčnímu kódu, když kompiluje `main()`. Totéž platí, jestliže je funkce částí knihovny. Jediný způsob jak se vyhnout funkčnímu prototypu je umístit definici funkce před její první použití. Toto není vždy možné. Styl programování v C++ dává funkci `main()` dopředu, protože většinou poskytuje strukturu celému programu.

Syntaxe prototypu

Prototyp funkce je příkaz, proto musí mít ukončovací středník. Nejjednodušší způsob získání prototypu je nakopírovat hlavičku funkce z její definice a přidat středník. Je to totéž, co provádí program s funkcí `cube()`:

```
double cube(double x); // přidejte ; k hlavičce na získání prototypu
```

Nicméně prototyp funkce nevyžaduje poskytnutí jmen proměnných; seznam typů postačuje. Program vytváří prototyp `cheers()` pouze pomocí typu parametru:

```
void cheers(int); // vypustit jména proměnných v prototypu je v pořádku
```

Obecně můžete v seznamu parametrů prototypů buď začlenit nebo vyloučit jména proměnných. Jména proměnných v prototypu pouze fungují jako držitelé místa, tudíž používáte-li jména, nemusí odpovídat jménům v definici funkce.

Porovnání prototypů v C++ a v ANSI C

ANSI C si vypůjčil prototypy z C++, ale tyto dva jazyky mají jisté rozdíly. Nejdůležitější je, že ANSI C, aby zachoval kompatibilitu s klasickým C, prováděl vytváření prototypů volitelně, ale C++ povinně. Například uvažujme následující deklaraci funkce:

```
void say_hi();
```

Necháme-li v C++ prázdné závorky, tak je to stejné, jako když uvnitř bude klíčové slovo void. To znamená, že funkce nemá žádné parametry. Necháme-li v ANSI C prázdné závorky, znamená to, že odmítáme stanovit, jaké jsou parametry. To znamená, že se zříkáte vytvoření prototypu seznamu parametrů.

Co pro vás prototypy dělají

Viděli jste, že prototypy pomáhají kompilátoru, ale co dělají pro vás? Znatelně redukují možnost programových chyb. Zvláště zajišťují následující:

- Kompilátor se správně postará o návratovou hodnotu funkce.
- Kompilátor ověří, zda používáte přesný počet parametrů ve funkci.
- Kompilátor ověří, zda používáte přesný typ parametrů. Jestliže ne a je-li to možné, konvertuje parametry na správný typ.

Diskutovali jsme již, jak se správně postarat o návratovou hodnotu funkce. Podívejme se nyní, co se stane, když použijete chybný počet parametrů. Například předpokládejte, že jste udělali následující volání:

```
double z = cube();
```

Bez vytváření prototypů funkcí to nechá kompilátor projít. Když se volá funkce, dívá se, kam by volání `cube()` mělo umístit číslo a použije jakoukoli hodnotu, která tam bude. To je například stejné, jak pracoval C před ANSI C, který si vytváření prototypů vypůjčil od C++. Protože je vytváření prototypů pro ANSI C volitelné, odpovídá to tomu, jak pracují některé programy v C. Avšak v C++ to volitelné není, takže máte zajištěnu ochranu před tímto druhem chyby.

Dále předpokládejme, že poskytnete parametr, ale ten má špatný typ. V C by to mohlo vytvořit příšerné chyby. Například, jestliže funkce očekává hodnotu typu `int` (předpokládejme, že má 16 bitů) a vy předáte `double` (předpokládejme, že má 64 bitů), funkce se podívá pouze na prvních 16 z 64 bitů a pokusí se je interpretovat jako hodnotu typu `int`. C++ však konvertuje hodnotu, kterou předáváte na typ specifikovaný v prototypu za předpokladu, že to jsou oba aritmetické typy. Například výpis programu 7.2 zvládne obdržení dvou neshodných typů v jednom příkazu:

```
cheers(cube(2));
```

Za prvé program předá hodnotu 2 typu `int` do `cube()`, která očekává typ `double`. Kompilátor si všimne, že prototyp `cube()` specifikuje parametr typu `double`, konvertuje 2 na 2.0, což je hodnota typu `double`. Potom navrací hodnotu (8.0) typu `double`, která se má použít jako parametr pro `cheers()`. Opět kompilátor prověří prototypy a poznamená, že `cheers()` vyžaduje `int`. Konvertuje návratovou hodnotu na celočíselnou hodnotu 8. Obec-

ně vytváření prototypů produkuje automatické přetypování na očekávané typy. (Přetížení funkcí, o kterém se pojednává v kapitole 8, „Příběhy ve funkcích", může vytvářet nejednoznačné situace, avšak brání některým typům v automatickém přetypování.)

Automatická konverze typu nepředejde všem možným chybám. Například, jestliže předáte hodnotu 8.33E27 funkci, která očekává `int`, tak velké číslo se nemůže správně konvertovat do obyčejného `int`. Některé kompilátory vás varují před možnou ztrátou dat, když dochází k automatické konverzi z většího na menší typ.

Vytváření prototypů má také za následek typovou konverzi pouze tehdy, když to má smysl. Například nebude provádět konverzi celého čísla na strukturu nebo na ukazatel.

Vytváření prototypů má své místo v době kompilace a nazývá se *statické ověření typu*. Statické ověření typu, jak jsme právě viděli, zachycuje mnoho chyb, které je obtížné zachytit při běhu programu.

Parametry funkcí a předávání hodnotou

Je čas, abychom se blíže podívali na parametry funkcí. C++ obvykle předává parametry hodnotou. To znamená, že se číselná hodnota parametru předá funkci, kde se přiřadí do nové proměnné. Například výpis programu 7.2 má toto volání funkce:

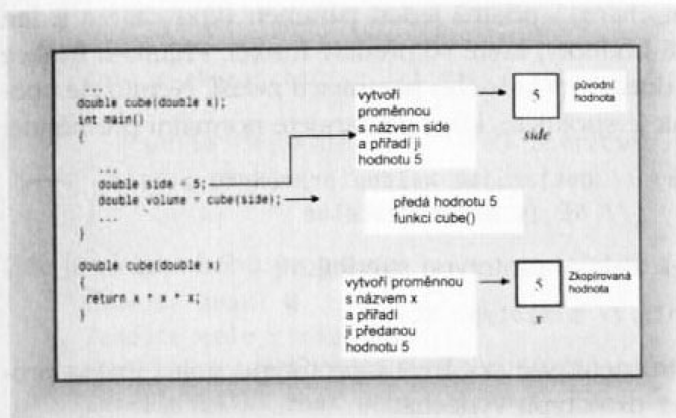
```
double volume = cube(side);
```

Zde je proměnná side, která měla za běhu programu hodnotu 5. Připomínáme, že hlavička funkce `cube()` byla:

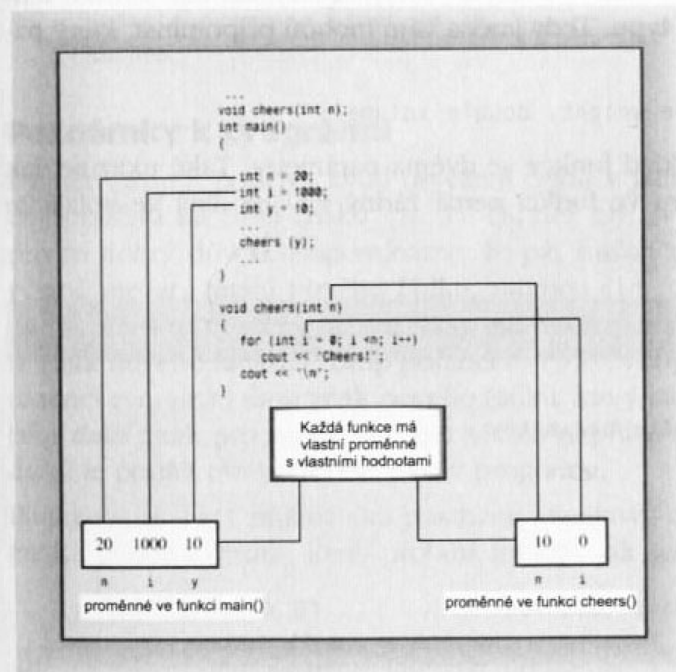
```
double cube(double x)
```

Když se tato funkce vyvolá, vytváří novou proměnnou typu `double`, která se nazývá `x` a přiřadí jí hodnotu 5. To izoluje data v `main()` před činnostmi, které mají své místo v `cube()`, protože `cube()` pracuje raději s kopií proměnné `side` než s původním údajem. Brzy příklad této ochrany uvidíte. Proměnná, která se používala na přijetí předávané hodnoty se nazývá *formální parametr*. Hodnota, která se předala do funkce se nazývá *skutečný parametr*. Tedy předání parametru přiřazuje skutečný parametr formálnímu. Viz obrázek 7.2.

Proměnné, včetně formálních parametrů, které jsou deklarovány uvnitř funkce, jsou vzhledem k funkci privátní. Když se funkce vyvolá, počítač alokuje pro tyto proměnné potřebnou paměť. Když se funkce ukončí, počítač uvolní paměť, která byla pro tyto proměnné použita. (Některá literatura o C++ se zmiňuje o alokování a uvolňování paměti, jako o vytvoření a destrukci proměnných. To vyznívá mnohem vzrušivěji.) Takové proměnné se nazývají *lokální proměnné*, protože jsou umístěny ve funkci. Jak jsme se zmínili, napomáhá to uchovat datovou integritu. Také to znamená, že když v `main()` deklarujete proměnnou, která se nazývá `x` a jinou proměnnou stejného jména v další funkci, obě to jsou různé proměnné, které k sobě nemají žádný vztah, podobně jako se liší Albany v Kalifornii od Albany v New Yorku. Viz obrázek 7.3.



Obrázek 7.2 Předání hodnotou



Obrázek 7.3 Lokální proměnné

Násobné parametry

Funkce mohou mít více než jeden parametr. Ve volání funkce pouze oddělte parametry čárkou:

```
n_chars('R', 25);
```

To předává funkci `n_chars()`, která bude zakrátko definována, dva parametry. Podobně, když deklarujete funkci, používejte v hlavičce funkce seznam deklarací oddělených čárkou:

```
void n_chars(char c, int n)  // dva parametry
```

Hlavička funkce stanoví, že funkce `n_chars()` přijímá jeden parametr typu `char` a jeden typu `int`. Proměnným `c` a `n` se přiřadí hodnoty, které se předaly funkci. Přijímá-li funkce dva parametry stejného typu, musíte dodat typ každého parametru zvlášť. Nemůžete spojovat deklarace stejným způsobem, jak je spojujete, když deklarujete normální proměnné:

```
void fifi(float a, float b) // deklarujte každou proměnnou zvlášť
void fifi(float a, b)       // NE je akceptovatelné
```

Stejně jako u dalších funkcí použijte k získání prototypu středník:

```
void n_chars(char c, int n); // prototyp, styl 1
```

Podobně jako u jednotlivých parametrů nemusíte používat v prototypu stejná jména proměnných jako v definici a můžete je v prototypu vynechat:

```
void n_chars(char, int);     // prototyp, styl 2
```

Avšak poskytnutí jmen proměnných může vytvářet prototyp srozumitelnější zvláště tehdy, když jsou dva parametry stejného typu. Tedy jména vám mohou připomínat, který parametr je který:

```
double melon_density(double weight, double volume);
```

Výpis programu 7.3 vám ukazuje příklad funkce se dvěma parametry. Také ukazuje, jak změna hodnoty formálního parametru ve funkci nemá žádný vliv na data ve volajícím programu.

Výpis programu 7.3 twoarg.cpp

```
#include <condefs.h>
// twoarg.cpp – funkce se dvěma parametry
#include <iostream>
using namespace std;
void n_chars(char, int);
int main()
{
    int times;
    char ch;

    cout << "Zadejte znak: ";
    cin >> ch;
    while (ch != 'q')          // q kvůli ukončení
    {
        cout << "Zadejte cele cislo: ";
        cin >> times;
        n_chars(ch, times);  // funkce se dvěma parametry
        cout << "\nZadejte dalsi znak nebo stisknete"
                " klavesu q kvuli ukonceni: ";
        cin >> ch;
    }
    cout << "Hodnota promenne times je " << times << ".\n";
    cout << "Sbohem\n";
    return 0;
```

```
}

void n_chars(char c, int n) // zobrazí c n-krát
{
    while (n-- > 0)          // pokračuje, dokud n nedosáhne 0
        cout << c;
}
```

Zde je ukázka běhu programu:

```
Zadejte znak: W
Zadejte cele cislo: 50
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
Zadejte dalsi znak nebo stisknete klavesu q kvuli ukonceni: a
Zadejte cele cislo: 20
aaaaaaaaaaaaaaaaaaaa
Zadejte dalsi znak nebo stisknete klavesu q kvuli ukonceni: q
Hodnota promenne times je 20.
Sbohem
```

Poznámky k programu

Funkce `main()` používá kvůli osvěžení cyklů v paměti cyklus `while`. Všimněte si, že raději používá na čtení znaků `cin >> ch`, než `cin.get(ch)` nebo `ch = cin.get()`. Existuje pro to dobrý důvod. Připomínáme, že pár funkcí `cin.get()` čte všechny vstupní znaky včetně mezer, znaků nového řádku, zatímco `cin >> ch` je přeskakuje. Když odpovídáte na programovou výzvu, musíte stisknout na konci každého řádku ENTER, tedy generujete znak nového řádku. Přístup pomocí `cin >> ch` obyčejně tyto znaky přeskakuje, ale sourozenci `cin.get()` čtou znak nového řádku, který následuje za každým číslem zavedeným jako další znak pro zobrazení. Bez těchto nepříjemností můžete programovat, ale jednodušší je použít `cin` podobně jako v programu.

Funkce `n_chars()` přijímá dva parametry: znakový `c` a celočíselný `n`. Potom na zobrazení znaku používá cyklus, který probíhá tolikrát, jak specifikuje celočíselný parametr:

```
while (n-- > 0)          // pokračuj dokud n nedosáhne 0
    cout << c;
```

Všimněte si, že program udržuje čítač snižováním proměnné `n`, kde `n` je formální parametr ze seznamu parametrů. Této proměnné se přiřadí hodnota proměnné `times` z `main()`. Cyklus `while` potom snižuje `n` na 0, ale jak běh programu demonstruje, změna hodnoty `n` nemá žádný vliv na `times`.

Jiná funkce se dvěma parametry

Vytvořme náročnější funkci, která provádí netriviální výpočty. Funkce také bude předvádět použití lokálních proměnných, které jsou jiné než formální parametry funkce.

Mnoho států ve Spojených státech nyní podporuje loterii jisté formy hry Lotto. Lotto vám umožní vybrat určitý počet výběrů z hrací sady. Například z hrací sady, která má 51 čísel, byste mohli získat výběr šesti čísel. Potom Lotto zařídí náhodný výběr šesti čísel. Jestliže jim vaše volba přesně odpovídá, vyhrajete několik miliónů dolarů a podobně. Naše

funkce bude počítat pravděpodobnost, že jste zvolili správná čísla. (Ano, funkce, která úspěšně předpovídá správná čísla by mohla být užitečnější, ale C++, ačkoli je účinný, musí ještě implementovat duševní schopnosti.)

Za prvé potřebujeme vzorec. Předpokládejte, že musíte vybrat šest hodnot z 51. Potom matematika říká, že máte jednu šanci výhry ku R, kde R poskytuje následující vzorec:

$$R = \frac{51 \times 50 \times 49 \times 48 \times 47 \times 46}{6 \times 5 \times 4 \times 3 \times 2 \times 1}$$

Pro šest výběrů je jmenovatel součinem šesti prvních celých čísel, neboli šest faktoriál. Čitatel je také součinem šesti za sebou jdoucích čísel, která tentokrát začínají od 51 a jdou směrem dolů. Obecněji, jestliže vyberete `picks` hodnot z `numbers` čísel, jmenovatel je `picks` faktoriál a čitatel je součinem `picks` celých čísel počínaje hodnotou `numbers`, který funguje směrem dolů. K provedení tohoto výpočtu můžete použít cyklus `for`:

```
long double result = 1.0;
for (n = numbers, p = picks; p > 0; n--, p--)
    result = result * n / p ;
```

Spíše než aby násobil nejprve všechny členy, cyklus začíná násobením 1.0 prvním členem čitatele a potom dělením prvním členem jmenovatele. Potom, v dalším průběhu cyklem, násobí a dělí druhými členy čitatele a jmenovatele. To udržuje průběžný součin menší, než kdybyste provedli nejprve celé násobení. Například porovnejte

```
(10 * 9) / (2 * 1)
```

s následujícím:

```
(10 / 2) * (9 / 1)
```

První se vyhodnotí na `90/2` a potom na `45`, zatímco druhý na `5*9` a potom na `45`. Oba poskytují stejnou odpověď, ale první metoda produkuje větší mezihodnotu (`90`) než druhá. Čím máte více činitelů, tím dostanete větší rozdíl. Pro velká čísla strategie střídajícího se násobení a dělení může ochránit před přetečením maximální možné hodnoty v pohyblivé řádové čárce. Výpis programu 7.4 zahrnuje tento vzorec do funkce `odds()`. Protože počet tahů a celkový počet výběrů by měly být kladné hodnoty, program používá pro tyto veličiny typ `unsigned int` (zkráceně `unsigned`). Vynásobení několika celých čísel dohromady může vyprodukovat pořádně velký výsledek, proto `lotto.cpp` používá pro návratovou hodnotu funkce typ `long double`. Také výrazy, co se týče celočíselných typů, jako například 49/6, produkují chybu odříznutí části čísla.

Výpis programu 7.4 lotto.cpp

```
#include <condefs.h>
// lotto.cpp - šance na výhru
#include <iostream>
using namespace std;
// Poznámka: některé implementace vyžadují double místo long double
long double odds(unsigned numbers, unsigned picks);
int main()
```

```
{
    double total, choices;
    cout << "Zadejte celkovy pocet hracich karet a\n"
         "pocet povolenych tahu:\n";
    while ((cin >> total >> choices) && choices <= total)
    {
        cout << "Mate sanci jedna ku ";
        cout << odds(total, choices);        // výpočet pravděpodobnosti
        cout << ", ze vyhrajete.\n";
        cout << "Dalsi dve cisla (q na ukonceni): ";
    }
    cout << "Sbohem\n";
    return 0;
}

// následující funkce počítá správně pravděpodobnost výběru
// čísel z počtu možností
long double odds(unsigned numbers, unsigned picks)
{
    long double result = 1.0;  // zde je několik lokálních proměnných
    long double n;
    unsigned p;

    for (n = numbers, p = picks; p > 0; n--, p--)
        result = result * n / p ;
    return result;
}
```

Kompatibilita:

Některé implementace C++ nepodporují typ `long double`. Jestliže vaše implementace selže v této kategorii, zkuste místo toho obvyklé `double`.

Zde je ukázka běhu programu. Všimněte si, že zvyšování počtu výběrů z hrací sady značně zvyšuje pravděpodobnost pro případ výhry.

```
Zadejte celkovy pocet hracich karet a
pocet povolenych tahu:
49 6
Mate sanci jedna ku 1.39838e+07, ze vyhrajete.
Dalsi dve cisla (q na ukonceni): 51 6
Mate sanci jedna ku 1.80095e+07, ze vyhrajete.
Dalsi dve cisla (q na ukonceni): 38 6
Mate sanci jedna ku 2.76068e+06, ze vyhrajete.
Dalsi dve cisla (q na ukonceni): q
Sbohem
```

Poznámky k programu

Funkce `odds()` uvádí příklad dvou druhů lokálních proměnných, které můžete mít ve funkci. Za prvé, existují formální parametry (`numbers` a `picks`), které se deklarují v hlavičce funkce před otevírací složenou závorkou. Potom přicházejí další lokální proměnné (`result`, `n` a `p`). Deklarují se mezi složenými závorkami, které ohraničují definici funkce. Hlavní rozdíl mezi formálními parametry a dalšími lokálními proměnnými spočívá v tom, že formální parametry dostávají své hodnoty z funkce, která volá `odds()`, zatímco ostatní proměnné dostávají hodnoty z vnitřku funkce.

Funkce a pole

Dosavadní ukázky funkcí byly jednoduché, používaly pouze základní typy parametrů a návratových hodnot. Ale funkce mohou být klíčem k zacházení s mnohem komplikovanějšími typy, jako jsou například pole a struktury. Podívejme se nyní, jak si spolu vedou pole a funkce.

Předpokládejme, že používáte pole, které sleduje, kolik zákusků snědla každá osoba na rodinném pikniku. (Každý index pole odpovídá osobě a počet prvků koresponduje s počtem zákusků, které osoba snědla.) Nyní chcete součet. Je to jednoduché; pouze dodáte cyklus na sečtení všech prvků pole. Avšak sčítání prvků pole je taková běžná úloha, že má smysl navrhnout funkci, která je provádí. Potom byste nemuseli psát nový cyklus pokaždé, když musíte pole sečíst.

Uvažujme, co zahrnuje funkční rozhraní. Protože funkce počítá součet, měla by tuto odpověď navracet. Jestliže ponecháte vaše zákusky celé, můžete použít funkci s návratovou hodnotou typu `int`. Aby funkce věděla, jaká má sčítat pole, potřebujete jí předat jméno pole jako parametr. A abyste vytvořili funkci obecně, takže se neomezuje na pole určité velikosti, předáte jí jeho velikost. Jedinou novou součástí zde je to, že musíte deklarovat, že jeden z formálních parametrů je jméno pole. Podívejme se, jaký je a jak vypadá zbytek hlavičky funkce:

```
int sum_arr(int arr[], int n) // arr = jméno pole, n = velikost
```

Vypadá to přijatelně. Hranaté závorky patrně indikují, že `arr` je pole a skutečnost, že jsou prázdné, zdá se ukazuje, že můžete použít funkci s polem libovolné velikosti. Avšak věci se nemají vždy tak jak vypadají: `arr` ve skutečnosti není pole; je to ukazatel! Dobrou zprávou je to, že můžete napsat zbytek funkce, právě tak, jako by `arr` bylo polem. Za prvé, podívejme se, že tento přístup pracuje a pak se podívejme na to, proč pracuje.

Výpis programu 7.5 ukazuje použití ukazatele jako by byl jménem pole. Program inicializuje pole jistými hodnotami a potom používá funkci `sum_arr()` na výpočet součtu. Všimněte si, že funkce `sum_arr()` používá `arr` jako by byl jménem pole.

Výpis programu 7.5 arrfun1.cpp

```
// arrfun1.cpp - funkce s parametrem typu pole
#include <iostream>
using namespace std;
const int ArSize = 8;
```

```
int sum_arr(int arr[], int n);          // prototyp
int main()
{
    int cookies[ArSize] = {1,2,4,8,16,32,64,128};
    // některé systémy vyžadují pro umožnění inicializace pole,
    // aby static předcházelo int

    int sum = sum_arr(cookies, ArSize);
    cout << "Celkovy pocet snedenych zakusku: " << sum << "\n";
    return 0;
}

// vrací součet prvků celočíselného pole
int sum_arr(int arr[], int n)
{
    int total = 0;

    for (int i = 0; i < n; i++)
        total = total + arr[i];
    return total;
}
```

Zde je výstup programu:

```
Celkovy pocet snedenych zakusku: 255
```

Jak vidíte, program pracuje. Nyní se podíváme, proč pracuje.

Pole a ukazatele (znovu)

Klíčem je, že C++, jako C, ve většině souvislostí nakládá se jménem pole, jako by to byl ukazatel. Připomínáme z kapitoly 4, že C++ interpretuje jméno pole jako adresu jeho prvního prvku:

```
cookies == &cookies[0]  // jméno pole je adresou jeho prvního prvku
```

(K tomuto pravidlu existují dvě výjimky. Za prvé, deklarace pole používá jméno pole k označení paměti. Za druhé, použití `sizeof` na jméno pole poskytuje velikost celého pole v bajtech.)

Výpis programu 7.5 provádí následující volání funkce:

```
int sum = sum_arr(cookies, ArSize);
```

Zde je `cookies` jméno pole, z toho důvodu je podle pravidel C++ adresou jeho prvního prvku. Funkce předává adresu. Protože má pole prvky typu `int`, `cookies` musí být typem ukazatel-na-`int`, neboli `int *`. To nasvědčuje, že by správnou hlavičkou funkce mělo být:

```
int sum_arr(int *arr, int n) // arr = jméno pole, n = velikost
```

Zde `int *arr` nahradilo `int arr[]`. Ukazuje se, že obě hlavičky jsou správné, co se týče C++, zápisy `int *arr` a `int arr[]` mají stejný význam, když (a pouze tehdy když) se používají v hlavičce funkce nebo v prototypu funkce. Oba znamenají, že `arr` je ukazatel-na-`int`. Avšak verze zápisu pole (`int arr[]`) nám symbolicky připomíná, že `arr` nejen uka-

zuje na int, ale ukazuje na první int v poli celých čísel. Budeme používat zápis pomocí pole, když se ukazatel vztahuje k prvnímu prvku pole a budeme používat zápis pomocí ukazatele, když se vztahuje k izolovanému prvku. Nezapomeňte, že zápisy int *arr a int arr[] nejsou v žádné jiné souvislosti synonymy. Například nemůžete použít zápis int tip[] na deklaraci ukazatele v těle funkce.

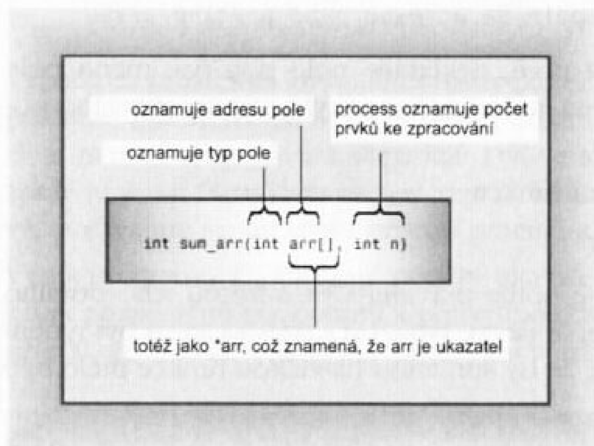
Je-li dáno, že proměnná arr je ve skutečnosti ukazatel, zbytek funkce dává smysl. Jak byste si mohli připomenout z diskuse o dynamických polích v kapitole 4, můžete pro přístup k jeho prvkům použít zápis pole s hranatými závorkami stejně dobře se jmény polí jako s ukazateli. Jelikož je arr ukazatel neboli jméno pole, výraz arr[3] znamená čtvrtý prvek pole. A pravděpodobně to ničemu neublíží, když vám připomeneme následující dvě identity:

```
arr[i] == *(ar + i)    // hodnoty ve dvou zápisech
&arr[i] == ar + i      // adresy ve dvou zápisech
```

Připomínáme, přidání 1 k ukazateli, jež zahrnuje jméno pole, vlastně přidává hodnotu, která se rovná velikosti typu v bajtech, na který ukazatel ukazuje. Přičítání k ukazateli a indexování pole jsou dva ekvivalentní způsoby odpočítávání prvků od začátku pole.

Důsledky použití polí jako parametrů

Podívejme se na důsledky výpisu programu 7.5. Volání funkce sum_arr(cookies, ArSize) předává adresu prvního prvku pole cookies a počet prvků pole do funkce sum_arr(). Funkce sum_arr() přiřazuje adresu cookies ukazatelové proměnné arr a ArSize do proměnné n typu int. To znamená, že výpis programu 7.5 ve skutečnosti funkci nepředává obsah pole. Místo toho funkci říká, kde se pole nachází (adresa), jaký druh prvků má (typ) a kolik prvků má (proměnná n). Viz obrázek 7.4. Funkce vyzbrojená touto informací potom používá původní pole. Když se předá obyčejná proměnná, funkce pracuje s kopií. Ale když se předá pole, funkce pracuje s originálem. Ve skutečnosti tento rozdíl neporušuje přístup předání-hodnotou v C++. Funkce sum_arr() stále předává hodnotu, která se přiřadí nové proměnné. Ale hodnota je jediná adresa, ne obsah pole.



Obrázek 7.4 Působení na funkci pomocí pole

Je shoda mezi jmény pole a ukazateli dobrou věcí? Vskutku je. Rozhodnutí při návrhu použít jméno pole jako parametru šetří čas a paměť, která je potřeba na zkopírování celého pole. Režie spojená s použitím kopie může být odrazující, když pracujete s velkými poli.

Nejen, že program potřebuje více počítačové paměti, ale musí spotřebovat čas na zkopírování velkých bloků dat. Na druhé straně, práce s původními daty zvyšuje možnost nenávratného poškození dat. To je skutečný problém v klasickém C, ale modifikátor `const` v ANSI C a C++ poskytuje nápravu. Brzy ukážeme příklad. Ale nejprve změňme výpis programu 7.5 na ilustraci některých vlastností, které se týkají toho, jak fungují funkce s poli. Výpis programu 7.6 ukazuje, že `cookies` a `arr` mají stejnou hodnotu. Také ukazuje, jak ukazatelové pojetí dělá funkci `sum_arr()` mnohem všestrannější, než jak by se z počátku mohlo zdát.

Výpis programu 7.6 arrfun2.cpp

```
// arrfun2.cpp – funkce s parametrem typu pole
#include <iostream>
using namespace std;
const int ArSize = 8;
int sum_arr(int arr[], int n);
int main()
{
      int cookies[ArSize] = {1,2,4,8,16,32,64,128};
      // některé systémy vyžadují pro umožnění inicializace pole,
      // aby static předcházelo int

      cout << cookies << " = adresa pole, ";
      //  některé systémy vyžadují přetypování: unsigned (cookies)

      cout << sizeof cookies << " = velikost pole cookies\n";
      int sum = sum_arr(cookies, ArSize);
      cout << "Celkovy pocet snedenych zakusku: " << sum << "\n";
      sum = sum_arr(cookies, 3);            // a lie
      cout << "Prvni tri jedlici snedli " << sum << " zakusku.\n";
      sum = sum_arr(cookies + 4, 4);       // jiné zadání
      cout << "Posledni ctyri jedlici snedli " << sum << " zakusku.\n";
      return 0;
}

// navrací součet prvků celočíselného pole
int sum_arr(int arr[], int n)
{
      int total = 0;
      cout << arr << " = arr, ";
// některé systémy vyžadují přetypování: unsigned (arr)

      cout << sizeof arr << " = velikost pole arr\n";
      for (int i = 0; i < n; i++)
           total = total + arr[i];
      return total;
}
```

Zde je výstup (hodnoty adres, polí a velikostí celých čísel se budou od systému k systému měnit):

```
0065FDE0 = adresa pole, 32 = velikost pole cookies
0065FDE0 = arr, 4 = velikost pole arr
Celkovy pocet snedenych zakusku: 255
0065FDE0 = arr, 4 = velikost pole arr
Prvni tri jedlici snedli 7 zakusku.
0065FDF0 = arr, 4 = velikost pole arr
Posledni ctyri jedlici snedli 240 zakusku.
```

Poznámky k programu

Výpis programu 7.6 ukazuje některé velmi zajímavé vlastnosti o funkcích, které pracují s poli. Za prvé, všimněte si, že se `cookies` a `arr` obě vyhodnocují na stejnou adresu, přesně jak se tvrdilo. Ale `sizeof cookies` je 16, zatímco `sizeof arr` je pouze 4. To je proto, že `sizeof cookies` je velikost celého pole, zatímco `sizeof array` je velikost ukazatelové proměnné. (Provedení tohoto programu se odehrává na systému, který používá 4-bajtové adresy.) Mimochodem, proto musíte explicitně předat velikost pole spíše než použít `sizeof arr` ve funkci `sum_arr()`. Například, když program používá podruhé funkci, provádí toto volání:

```
sum = sum_arr(cookies, 3);
```

Tím, že řeknete funkci, že má `cookies` pouze tři prvky, přimějete ji vypočítat součet prvních tří prvků.

Proč se zde zastavujeme? Můžete totiž zalhat, kde pole začíná:

```
sum = sum_arr(cookies + 4, 4);
```

Protože `cookies` funguje jako adresa prvního prvku, `cookies + 4` funguje jako adresa pátého. Tento příkaz sčítá pátý, šestý, sedmý a osmý prvek v poli. Všimněte si ve výstupu, jak třetí volání funkce přiřazuje do `arr` jinou adresu, než to provedla první dvě volání. A ovšem můžete jako parametr místo `cookies + 4` použít `&cookies[4]`; oba znamenají stejnou věc.

Pamatujte:

Indikaci druhu pole a počtu prvků funkce zpracovávající pole provedete předáním informací prostřednictvím dvou samostatných argumentů:

```
void fillArray(int arr[], int size);    //prototyp
```

Nesnažte se předávat velikost pole v závorkách:

```
void fillArray(int arr[size]);          //NE --nesprávný prototyp
```

Více funkcí, které pracují s poli

Když si na znázornění dat vyberete použití pole, provádíte rozhodovací návrh. Avšak rozhodovací návrhy by se měly dostat na to, jak jsou data uložena; také by měly zahrnovat, jak se používají. Často zjistíte, že je užitečné psát konkrétní funkce na zacházení s konkrétními datovými operacemi. (Tím dosáhnete zvýšení programové spolehlivosti, usnadnění modifikace a odstranění chyb.) Kromě toho, když začínáte při přemýšlení o programu spojovat vlastnosti paměti s operacemi, uvažujete o důležitém kroku názorového kompletu OOP; který by se také mohl ukázat užitečným v budoucnosti.

Vyzkoušejme jednoduchý případ. Předpokládejme, že chcete použít pole na sledování hodnoty vašeho skutečného majetku v dolarech. (Je-li to nezbytné, předpokládejte, že máte skutečný majetek.) Musíte se rozhodnout, jaký použijete typ. Ovšem, `double` je mnohem méně omezující ve svém rozsahu než `int` nebo `long` a poskytuje dostatek významných číslic na přesné zobrazení hodnot. Dále byste se měli rozhodnout o počtu prvků v poli. (Použitím dynamických polí vytvořených pomocí `new` můžete rozhodnutí odložit, ale nechme věci jednoduché.) Předpokládejme, že nemáte více než pět nemovitostí, tedy můžete použít pole o pěti prvcích typu `double`.

Uvažujte nyní o možných operacích, které byste mohli chtít provádět s polem nemovitého majetku. Dvě naprosto základní jsou na čtení hodnot do pole a zobrazení obsahu pole. Do seznamu přidejme ještě jednu další operaci: přecenění hodnoty nemovitosti. Pro jednoduchost předpokládejme, že všechny vaše nemovitosti zvyšují nebo snižují hodnotu ve stejném poměru. (Pamatujte, toto je kniha o C++, nikoli o správě nemovitého majetku.) Dále přizpůsobte funkci každé operaci a potom podle toho napište programový kód. Tyto kroky dále projdeme.

Naplnění pole

Protože funkce s parametrem jména pole přistupuje k původnímu poli, nikoli ke kopii, můžete pro přiřazení hodnot prvkům pole použít volání funkce. Jedním parametrem funkce bude jméno pole, které se má naplnit. Obecně by měl program spravovat více než jednu investici jedince, tedy více než jedno pole, takže byste neměli chtít zabudovat velikost pole do funkce. Místo toho předejte velikost pole jako druhý parametr, jako v předchozím příkladě. Je také možné, že byste chtěli ukončit načítání dat před naplněním pole, takže byste měli tento rys do funkce zabudovat. Protože byste mohli zavést méně, než je maximální počet prvků, má smysl, aby funkce navracela skutečný počet zavedených hodnot. Tyto úvahy naznačují následující prototyp funkce:

```
int fill_array(double ar[], int limit);
```

Funkce bere parametr jméno pole a parametr, který specifikuje maximální počet položek, který se má přečíst, a navrací skutečný počet přečtených položek. Například, jestliže použijete funkci s polem o pěti prvcích, předáte 5 jako druhý parametr. Zavedete-li pouze tři hodnoty, funkce vrátí 3.

K načtení po sobě jdoucích hodnot do pole můžete použít cyklus, ale jak ho dříve ukončíte? Jeden způsob spočívá v použití zvláštní hodnoty na indikaci konce vstupu. Je-li to dáno, můžete kódovat funkci následovně:

```
int fill_array(double ar[], int limit)
{
    double temp;
    int i;
    for (i = 0; i < limit; i++)
    {
        cout << "Zadejte hodnotu #" << i + 1 << ": ";
        cin >> temp;
        if (temp < 0)      // signál pro ukončení
            break;
        ar[i] = temp;
```

```
    }
    return i;
}
```

Všimněte si, že programový kód obsahuje v programu výzvu uživateli. Jestliže uživatel zavede nezápornou hodnotu, pak se přiřadí do pole. Jinak cyklus končí. Zavede-li uživatel pouze platné hodnoty, cyklus končí poté, co přečte limitní hodnotu. Poslední věcí, kterou cyklus provádí, je inkrementace i, takže když cyklus končí, i je o 1 větší než poslední index pole, tedy je roven počtu naplněných prvků. Funkce potom tuto hodnotu navrací.

Zobrazení pole a jeho ochrana pomocí const

Vytvoření funkce na zobrazení obsahu pole je jednoduché. Funkci předáte jméno pole a počet plněných prvků, a funkce potom používá cyklus k zobrazení každého prvku. Ale existuje další důvod – zajišťující, že zobrazovací funkce nezmění původní pole. Pokud účelem funkce není změnit data, která se jí předávají, měli byste ji zabezpečit, aby je neměnila. Zabezpečení nastává automaticky s obvyklými parametry, protože je C++ předává hodnotou a funkce pracuje s jejich kopií. Ale funkce, které používají pole, pracují s originálem. To je koneckonců důvod, proč je funkce fill_array() schopna svou práci vykonat. Abychom zabránili funkci v nevhodné změně obsahů parametrů pole, můžete použít klíčové slovo const (o kterém se pojednávalo v kapitole 3, „Práce s daty"), když jste deklarovali formální parametr:

```
void show_array(const double ar[], int n);
```

Deklarace stanoví, že ukazatel ar ukazuje na konstantní data. To znamená, že ho nemůžete použít na jejich změnu. To jest, můžete použít hodnotu jako například ar[0], ale nemůžete ji měnit. Všimněte si, že to neznamená, že původní pole musí být konstantou; pouze to znamená, že nemůžete použít ar ve funkci show_array() na změnu dat. Tedy, show_array() zachází s polem jako s daty pouze pro čtení. Předpokládejme, že nešťastnou náhodou toto omezení porušíte provedením něčeho, jako je v následující funkci show_array():

```
bar[0] += 10;
```

Potom kompilátor ukončí váš chybný postup. Borland C++ například vydává chybovou zprávu takto (mírně upraveno):

```
Cannot modify a const object in function
        show_array(const double *,int)
```

```
(nelze modifikovat konstantní objekt ve funkci)
```

Zpráva nám připomíná, že C++ interpretuje deklaraci const double ar[] ve významu const double *ar. Tedy deklarace skutečně říká, že ar ukazuje na konstantní hodnotu. Podrobněji to probereme, až skončíme se současným příkladem. Zatím je tu programový kód funkce show_array():

```
void show_array(const double ar[], int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "Majetek #" << i + 1 << ": $"
```

```
            cout << ar[i] << "\n";
        }
    }
```

Modifikace pole

Třetí operací pro naše pole je násobení každého prvku stejným oceňovacím koeficientem. Potřebujete předat funkci tři parametry: koeficient, pole a počet prvků. Není potřeba žádné návratové hodnoty, takže funkce může vypadat takto:

```
void reassess(double r, double ar[], int n)
{
        for (int i = 0; i < n; i++)
            ar[i] *= r;
}
```

Protože se od této funkce předpokládá, že bude měnit hodnoty, nepoužívejte `const`, když deklarujete `ar`.

Sestavení částí dohromady

Nyní, když jsme definovali datové typy ve smyslu, jak jsou uloženy (pole) a použity (tři funkce), můžeme program, který používá tento návrh, sestavit dohromady. Protože jsme již vytvořili všechny prostředky, které zacházejí s polem, ohromně jsme zjednodušili programování `main()`. Většina zbývající programovací práce sestává z ponechání `main()` vyvolat funkce, které jsme právě vyvinuli. Výpis programu 7.7 ukazuje výledek.

Výpis programu 7.7 arrfun3.cpp

```
#include <condefs.h>
// arrfun3.cpp - funkce pracující s poli a kvalifikátor const
#include <iostream>
using namespace std;
const int Max = 5;

// prototypy funkcí
int fill_array(double ar[], int limit);
void show_array(const double ar[], int n);  // nemění data
void reassess(double r, double ar[], int n);

int main()
{
    double properties[Max];

    int size = fill_array(properties, Max);
    show_array(properties, size);
    cout << "Zadejte koeficient pro preceneni: ";
    double rate;
    cin >> rate;
    reassess(rate, properties, size);
    show_array(properties, size);
```

```
    return 0;
}

int fill_array(double ar[], int limit)
{
    double temp;
    int i;
    for (i = 0; i < limit; i++)
    {
        cout << "Zadejte hodnotu #" << (i + 1) << ": ";
        cin >> temp;
        if (temp < 0)
            break;
        ar[i] = temp;
    }
    return i;
}

// následující funkce může používat, ale nikoli měnit
// pole, jehož adresa je ar
void show_array(const double ar[], int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "Majetek #" << (i + 1) << ": $";
        cout << ar[i] << "\n";
    }
}

// násobí každý prvek pole ar[] koeficientem r
void reassess(double r, double ar[], int n)
{
    for (int i = 0; i < n; i++)
    ar[i] *= r;
}
```

Zde jsou dvě ukázky běhů programu. Připomínáme, že vstup údajů by se měl ukončit, když uživatel zavede pět nemovitostí nebo záporné číslo, podle toho, co nastane dříve. První příklad ukazuje dosažení limitu pěti nemovitostí a druhý zadání záporné hodnoty.

```
Zadejte hodnotu #1: 100000
Zadejte hodnotu #2: 80000
Zadejte hodnotu #3: 222000
Zadejte hodnotu #4: 240000
Zadejte hodnotu #5: 118000
Majetek #1: $100000
Majetek #2: $80000
Majetek #3: $222000
Majetek #4: $240000
Majetek #5: $118000
Zadejte koeficient pro preceneni: 1.1
Majetek #1: $110000
Majetek #2: $88000
```

```
Majetek #3: $244200
Majetek #4: $264000
Majetek #5: $129800
Zadejte hodnotu #1: 200000
Zadejte hodnotu #2: 84000
Zadejte hodnotu #3: 160000
Zadejte hodnotu #4: -2
Majetek #1: $200000
Majetek #2: $84000
Majetek #3: $160000
Zadejte koeficient pro preceneni: 1.20
Majetek #1: $240000
Majetek #2: $100800
Majetek #3: $192000
```

Poznámky k programu

Právě jsme pojednali o důležitých programovacích detailech, tak nyní přemýšlejme o postupu. Začali jsme přemýšlením o typu dat a navrhli vhodné funkce k zacházení s nimi. Potom jsme tyto funkce sestavili do programu. To se občas nazývá programování zdola nahoru, protože se postup návrhu přesouvá z dílčích částí na celek. Tento postup se dobře hodí na OOP, které se nejprve soustředí na prezentaci a manipulaci s daty. Tradiční programovací postupy na druhé straně mají sklon k programování seshora dolů, ve kterém nejprve vyvíjíte modulární celkový návrh a potom obrátíte svoji pozornost na detaily. Obě metody jsou užitečné a obě vedou k modulárním programům.

Ukazatele a `const`

Použití `const` s ukazateli má některé jemné aspekty (ukazatele mají vždy jemné aspekty), tak se na to podívejme blíže. S ukazateli můžete použít klíčové slovo `const` dvěma různými způsoby. První způsob je vytvořit ukazatel, který ukazuje na konstantní objekt, což vám brání ho použít na změnu hodnoty, na kterou se ukazuje. Druhý způsob je udělat ze samotného ukazatele konstantu a to vám zabraňuje změnit adresu, kam ukazatel ukazuje. Nyní detaily.

Za prvé deklarujme ukazatel `pt`, který ukazuje na konstantu:

```
int age = 39;
const int * pt = &age;
```

Tato deklarace stanoví, že `pt` ukazuje na `const int` (v tomto případě 39). Proto nemůžete `pt` použít na změnu hodnoty. Jinými slovy hodnota `*pt` je `const` a nemůže být změněna:

```
*pt += 1;          // CHYBNÉ, protože pt ukazuje na const int
cin >> *pt;        // CHYBNÉ ze stejného důvodu
```

Nyní jemná pointa. Naše deklarace `pt` neznamená nezbytně, že hodnota, na kterou ukazuje je skutečně konstantou; to pouze znamená, že hodnota je konstantou, pokud se uvažuje `pt`. Například `pt` ukazuje na `age` a `age` není `const`. Hodnotu `age` můžete měnit přímo použitím proměnné `age`, ale nemůžete ji přímo měnit prostřednictvím ukazatele `pt`:

```
*pt = 20;          // CHYBNÉ, protože pt ukazuje na const int
age = 20;          // SPRÁVNÉ protože se age nedeklaruje jako konstanta
```

V minulosti jsme přiřadili adresu řádné proměnné řádnému ukazateli. Nyní jsme přiřadili adresu řádné proměnné ukazateli-na-const. To zanechává další dvě možnosti: přiřazení adresy const proměnné ukazateli-na-const a adresy konstanty řádnému ukazateli. Jsou obě možné? První je a druhá není:

```
const float g_earth = 9.80;
const float * pe = &g_earth;    // SPRÁVNÉ

const float g_moon = 1.63;
float * pm = &g_moon;           // CHYBNÉ
```

V prvním případě nemůžete použít ani g_earth, ani pe pro změnu hodnoty 9.80. C++ nedovoluje druhý případ z jednoduchého důvodu – když můžete přiřadit adresu g_moon do pm, potom můžete podvádět a použít pm pro změnu hodnoty g_moon. To ovšem zpochybňuje konstantni status proměnné g_moon, takže vám C++ zabraňuje v přiřazení adresy konstanty k ne-konstantnímu ukazateli.

Pamatujte:

Můžete přiřadit jak adresu konstantních tak ne-konstantních dat ukazateli-na-konstantu, avšak adresu ne-konstantního údaje můžete přiřadit pouze ne-konstantnímu ukazateli.

Předpokládejme, že máte pole konstantních dat:

```
const int months[12] = {31,28,31,30,31,30, 31, 31,30,31,30,31};
```

Zákaz pro případ přiřazení adresy konstantního pole znamená, že nemůžete předat jméno pole jako parametr funkci, která používá ne-konstantní formální parametr:

```
int sum(int arr[], int n);  // mělo by být const int arr[]
...
int j = sum(months, 12);    // není povoleno
```

Volání funkce usiluje o přiřazení konstantního ukazatele (months) ne-konstantnímu ukazateli (arr) a kompilátor nedovolí volání funkce.

Použijte const, když můžete

Existují dva silné důvody na deklarování ukazatelových parametrů jako ukazatelů na konstantní data:

- Zabraňuje to programovým chybám, které nenávratně mění data.
- Použití const dovoluje funkci zpracovat jak konstantní, tak ne-konstantní skutečné parametry, zatímco když funkce v prototypu const vynechá, může přijmout pouze ne-konstantní data.

Měli byste deklarovat formální ukazatelové parametry jako ukazatele na konstantu kdykoli je vhodné to tak učinit.

Nyní další jemná pointa: deklarace

```
int age = 39;
const int * pt = &age;
```

vám pouze zabraňují ve změně hodnoty, na kterou pt ukazuje, což je 39. Nezabraňují vám ve změně hodnoty samotného pt. To jest, do pt můžete přiřadit novou adresu:

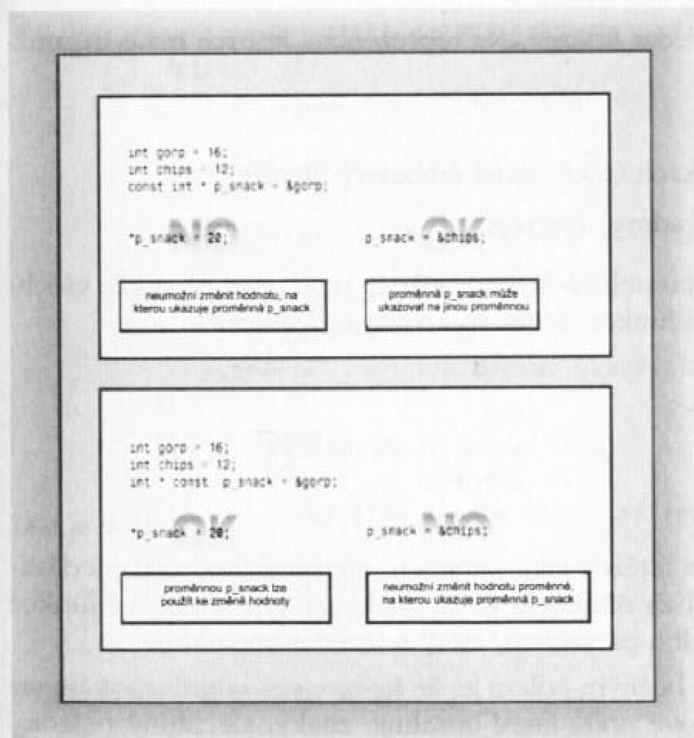
```
int sage = 80;
pt       = &sage; // v pořádku při ukazování na novou lokaci
```

Ale pořád nemůžete použít pt ke změně hodnoty, na kterou ukazuje (nyní 80).

Druhý způsob použití const znemožňuje změnit hodnotu samotného ukazatele:

```
int sloth = 3;
const int * ps = &sloth;        // ukazatel na const int
int * const finger = &sloth;    // konstantní ukazatel na int
```

Všimněte si, že poslední deklarace přemístila klíčové slovo const. Tento tvar deklarace nutí finger, aby ukazoval pouze na sloth. Avšak dovoluje vám použít finger na změnu hodnoty sloth. Prostřední deklarace nepovoluje použití ps na změnu hodnoty sloth, ale povoluje vám mít ps, které ukazuje na jinou lokaci. Zkrátka, finger a *ps jsou obě konstanty a *finger a ps konstanty nejsou. Viz obrázek 7.5.



Obrázek 7.5 Ukazatele-na-konstantu a konstantní ukazatele

Jestliže se vám to líbí, můžete deklarovat konstantní ukazatel na konstantní objekt:

```
double trouble = 2.0E30;
const double * const stick = &trouble;
```

Tady může ukazatel stick ukazovat pouze na proměnnou trouble a nemůže být použit na změnu její hodnoty. Zkrátka, jak stick, tak *stick jsou konstanty.

Obvykle, když předáváte ukazatele jako parametry funkce, používáte tvar ukazatel-na-konstantu na ochranu dat. Například připomínáme prototyp show_array() z výpisu programu 7.7:

```
void show_array(const double ar[], int n);
```

Použití const v této deklaraci znamená, že show_array() nemůže měnit hodnoty v libovolném poli, které se jí předá.

Funkce a řetězce ve stylu jazyka C

Připomínáme vám, že styl řetězců v C sestává ze série znaků ukončených prázdným znakem. Mnohé, co jste se dozvěděli při navrhování funkcí, které pracují s poli, platí také pro funkce, které pracují s řetězci. Ale k řetězcům existuje několik speciálních cest, které nyní rozpleteme.

Předpokládejme, že chcete funkci předat řetězec. Na reprezentaci řetězce máte tři možnosti:

- Pole znaků.
- Řetězcová konstanta v uvozovkách (také zvaná řetězcový literál).
- Ukazatel-na-znak nastavený na adresu řetězce.

Všechny tři volby jsou však typem ukazatel-na-znak (výstižněji typu char *), takže všechny můžete použít jako parametry pro funkce, které zpracovávají řetězce:

```
char ghost[15] = "letani";
char * str = "dupani";
int n1 = strlen(ghost);              // ghost je &ghost[0]
int n2 = strlen(str);                // ukazatel na znak
int n3 = strlen("poskakovani");       // adresa řetězce
```

Neformálně můžete říct, že předáváte řetězec jako parametr, ale ve skutečnosti předáváte adresu jeho prvního znaku. To má za následek, že by měl prototyp řetězcové funkce používat typ char * jako typ formálního parametru, který řetězec reprezentuje.

Jeden důležitý rozdíl mezi řetězcem a běžným polem je, že řetězec má zabudovaný ukončovací znak. (Připomínáme, že znakové pole, které obsahuje znaky, ale žádný prázdný znak, je pouze pole a nikoli řetězec.) To znamená, že nemusíte předávat velikost řetězce jako parametr. Místo toho může funkce použít cyklus na prověření každého znaku v řetězci po řadě, dokud nenarazí na ukončovací prázdný znak. Výpis programu 7.8 ilustruje tento přístup pomocí funkce, která počítá, kolikrát se daný znak vyskytuje v řetězci.

Výpis programu 7.8 strgfun.cpp

```
// strgfun.cpp - funkce s parametrem typu řetězec
#include <iostream>
using namespace std;
int c_in_str(const char * str, char ch);
int main()
{
    char mmm[15] = "minimum";     // řetězec v poli
    // některé systémy vyžadují pro inicializaci pole, aby
    // static předcházelo int

    char *wail = "ululate";    // wail ukazuje na řetězec (skučet)

    int ms = c_in_str(mmm, 'm');
    int us = c_in_str(wail, 'u');
    cout << ms << " m znaku v " << mmm << "\n";
    cout << us << " u znaku v " << wail << "\n";
    return 0;
}

// tato funkce počítá počet znaků ch
// v řetězci str
int c_in_str(const char * str, char ch)
{
    int count = 0;

    while (*str)        // konec, když je *str rovno '\0'
    {
        if (*str == ch)
            count++;
        str++;          // přesun ukazatele na další znak
    }
    return count;
}
```

Zde je výstup:

```
3 m znaku v minimum
2 u znaku v ululate
```

Poznámky k programu

Protože by funkce `c_int_str()` neměla měnit původní řetězec, když deklaruje formální parametr `str`, používá modifikátor `const`. Potom, když chybně necháte funkci změnit část řetězce, kompilátor zachytí vaši chybu. Samozřejmě můžete v hlavičce funkce použít zápis pomocí pole namísto deklarace `str`:

```
int c_in_str(const char str[], char ch) // také v pořádku
```

Avšak použití zápisu pomocí ukazatele vám připomíná, že parametr nemusí být jméno pole, ale může být nějaký další tvar ukazatele.

Samotná funkce ukazuje standardní způsob zpracování znaků v řetězci:

```
while (*str)
{
    příkazy
    str++;
}
```

Z počátku `str` ukazuje na první znak v řetězci, takže sama `*str` představuje první znak. Například bezprostředně po prvním zavolání funkce má hodnotu `m`, první znak ve slově `minimum`. Takže cyklus pokračuje tak dlouho, dokud znak není prázdným znakem (\0), `*str` je nenulová. Na konci každého cyklu výraz `str++` inkrementuje ukazatel o jeden bajt na další znak v řetězci. Nakonec `str` ukazuje na ukončovací prázdný znak, což způsobí, že se `*str` rovná `0`, což je numerický kód prázdného znaku. Tato podmínka ukončuje cyklus. (Proč jsou funkce zpracovávající řetězce tak nemilosrdné? Protože se jinak zastavují na ničem.)

Funkce, které navracejí řetězce

Nyní předpokládejme, že chcete napsat funkci, která navrací řetězec. Tak dobře, funkce to neumí udělat. Ale může vrátit adresu řetězce, a to je dokonce lepší. Například výpis programu 7.9 definuje funkci, která se jmenuje `buildstr()` a navrací ukazatel. Tato funkce přijímá dva parametry: znak a číslo. Použitím `new` funkce vytváří řetězec, jehož délka se rovná číslu, a potom každý prvek inicializuje znakem. Potom navrací ukazatel na nový řetězec.

Výpis programu 7.9 strgback.cpp

```
// strgback.cpp — funkce, která vrací ukazatel na char
#include <iostream>
using namespace std;
char * buildstr(char c, int n);      // prototyp
int main()
{
int times;
char ch;

cout << "Zadejte znak: ";
cin >> ch;
cout << "Zadejte cele cislo: ";
cin >> times;
char *ps = buildstr(ch, times);
cout << ps << "\n";
delete [] ps;                       // uvolnění paměti
ps = buildstr('+', 20);            // opětné použití ukazatele
cout << ps << "-HOTOVO-" << ps << "\n";
delete [] ps;                       // uvolnění paměti
return 0;
}
```

```
// vytváří řetězec o n znacích c
char * buildstr(char c, int n)
{
char * pstr = new char[n + 1];
pstr[n] = '\0';                    // ukončení řetězce
while (n-- > 0)
pstr[n] = c;              // naplní zbytek řetězce
return pstr;
}
```

Zde je ukázka běhu programu:

```
Zadejte znak: V
Zadejte cele cislo: 48
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
++++++++++++++++++++-HOTOVO-++++++++++++++++++++
```

Poznámky k programu

Abyste měli místo pro prázdné znaky na vytvoření řetězce o n viditelných znacích, potřebujete paměť pro n + 1 znaků. Takže funkce na jeho úschovu požádá o n + 1 znaků. Dále nastavuje poslední bajt na prázdný znak. Potom vyplní zbytek pole od konce do začátku. Cyklus

```
while (n-- > 0)
     pstr[n] = c;
```

probíhá n krát, až se n sníží na nulu. Na začátku posledního průběhu má n hodnotu 1. Protože n-- znamená použij hodnotu a potom ji sniž, testovací podmínka cyklu while porovnává 1 k 0, zjistí, zda test má hodnotu true a pokračuje. Ale po provedení testu funkce sníží n na 0, takže pstr[0] je posledním prvkem, který se nastaví na c. Důvod plnění řetězce od konce na začátek je vyhnout se použití dodatečné proměnné. Použití jiného pořadí by vyžadovalo něco jako toto:

```
int i = 0;
while (i < n)
      pstr[i++] = c;
```

Všimněte si, že proměnná pstr je lokální vzhledem k funkci buildstr, takže když funkce skončí, použitá paměť pro pstr (ale ne pro řetězec) se uvolní. Ale protože funkce navrací hodnotu pstr, program je schopný přistoupit k novému řetězci prostřednictvím ukazatele ps v main().

Program k uvolnění paměti použité pro řetězec používá delete, jakmile ho již nepotřebuje. Potom znovu používá ps, aby ukazoval na nový blok paměti získaný pro další řetězec a předchozí paměť uvolní. Nevýhoda tohoto druhu návrhu (mít funkci, která navrací ukazatel na paměť alokovanou pomocí new) je, že nechává na zodpovědnosti programátora, aby si pamatoval, že má použít delete. Šablona auto_ptr, o které se pojednává v kapitole 15, „Třída string a standardní knihovna šablon", může pomoci tento postup zautomatizovat.

Funkce a struktury

Přesuňme se z polí na struktury. Je jednodušší napsat funkce pro struktury než pro pole. Ačkoli se strukturní proměnné podobají polím v tom, že obě mohou obsahovat několik datových položek, strukturní proměnné se chovají jako základní, jednohodnotové proměnné, když vstupují do funkcí. Struktury můžete předat hodnotou, stejně tak jako předáváte obyčejné proměnné. V tomto případě funkce pracuje s kopií původní struktury. Funkce může také vrátit strukturu. Neexistují žádné komické triky, jako že je jméno pole adresou svého prvního prvku. Jméno struktury je jednoduše jménem struktury a chcete-li její adresu, musíte použít adresní operátor `&`.

Nejpřímější způsob programování za použití struktur je nakládat s nimi jako by to byly základní typy; to jest, předávat je jako parametry a používat je, je-li to nezbytné, jako návratové hodnoty. Avšak při předávání struktur hodnotou existuje jedna nevýhoda. Je-li struktura veliká, prostor a úsilí spojené s vytvořením kopie struktury může zvýšit požadavky na paměť a zpomalit systém. Z těchto důvodů (a protože nejprve C nepovoloval předání struktury hodnotou) mnoho programátorů v C dává přednost předávání struktury adresou a potom pro přístup k jejímu obsahu používají ukazatel. C++ poskytuje třetí alternativu, jež se nazývá předání odkazem, o které pojednáme v kapitole 8. Prověříme nyní další dvě volby a začneme předáním a navrácením celých struktur.

Předání a navrácení struktur

Předání struktur hodnotou má největší smysl, když je struktura relativně kompaktní, tak prozkoumejme pár příkladů v tomto směru. První příklad se zabývá dobou cestování. Některé mapy vám poví, že cesta trvá tři hodiny 50 minut z Thunder Falls do Bingo City a jednu hodinu 25 minut z Bingo City do Grotesquo. K reprezentaci takových časů můžete použít strukturu, přičemž jeden člen slouží pro hodnotu v hodinách a druhý v minutách. Sečtení dvou časů je trochu ošidné, protože možná musíte transformovat nějaké minuty do hodinového členu. Například dva předchozí časy se sčítají na čtyři hodiny a 75 minut, což by se mělo zkonvertovat na 5 hodin 15 minut. Vytvořme strukturu na vyjádření časové hodnoty a potom funkci, která vezme dvě takové struktury jako parametry a vrátí strukturu, která představuje jejich součet.

Definování struktury je jednoduché:

```
struct travel_time
{
    int hours;
    int mins;
};
```

Dále uvažujme prototyp funkce `sum()`, která navrací součet dvou takových struktur. Návratová hodnota by měla být typu `travel_time`, a tudíž by měla mít dva parametry. Tedy prototyp by měl vypadat takto:

```
travel_time sum(travel_time t1, travel_time t2);
```

Abychom sečetli dva časy, nejprve sečteme minutové členy. Celočíselné dělení 60 poskytne počet hodin, které se mají převést a operátor modulo (%) poskytne počet zbývajících

minut. Výpis programu 7.10 tento přístup zahrnuje do funkce `sum()` a přidává funkci `show_time()` na zobrazení obsahu struktury `travel_time`.

Výpis programu 7.10 travel.cpp

```
#include <condefs.h>
// travel.cpp – použiti struktur ve funkcich
#include <iostream>
using namespace std;
struct travel_time
{
    int hours;
    int mins;
};
const int Mins_per_hr = 60;

travel_time sum(travel_time t1, travel_time t2);
void show_time(travel_time t);

int main()
{
    travel_time day1 = {5, 45};     // 5 hodin, 45 min

    travel_time day2 = {4, 55};     // 4 hodiny, 55 min

    travel_time trip = sum(day1, day2);
    cout << "Celkem za dva dny: ";
    show_time(trip);

    travel_time day3= {4, 32};
    cout << "Celkem za tri dny: ";
    show_time(sum(trip, day3));

    return 0;
}

travel_time sum(travel_time t1, travel_time t2)
{
    travel_time total;

    total.mins = (t1.mins + t2.mins) % Mins_per_hr;
    total.hours = t1.hours + t2.hours +
                  (t1.mins + t2.mins) / Mins_per_hr;
    return total;
}

void show_time(travel_time t)
{
    cout << t.hours << " hodin, "
         << t.mins << " minut\n";
}
```

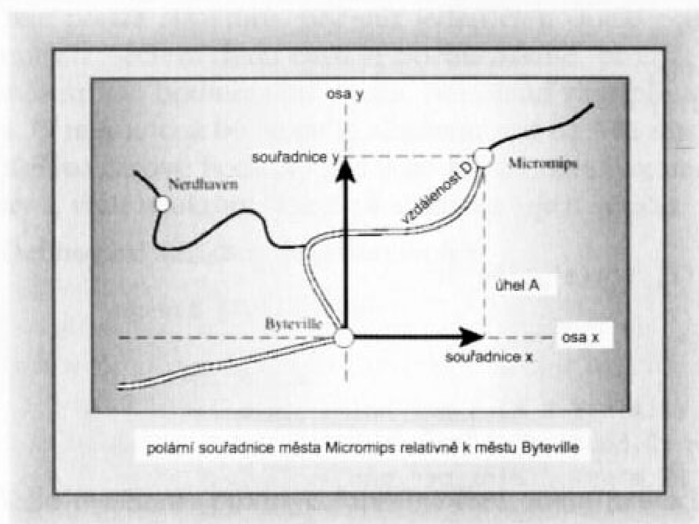
Zde `travel_time` funguje pouze jako standardní jméno typu; můžete ho použít na deklaraci proměnných, funkčních návratových typů a typů parametrů funkce. Protože proměnné jako `total` a `t1` jsou struktury `travel_time`, můžete na ně použít operátor příslušnosti tečka. Všimněte si, že protože funkce `sum()` navrací strukturu `travel_time`, můžete ji použít jako parametr ve funkci `show_time()`. Protože funkce v C++ standardně předávají parametry hodnotou, volání funkce `show_time(sum(trip, day3))` nejprve vyhodnotí funkční volání `sum(trip, day3)` na získání její návratové hodnoty. Volání `show_time()` potom předá návratovou hodnotu `sum()` do `show_time()`, nikoli funkci samotnou. Zde je výstup programu:

```
Celkem za dva dny: 10 hodin, 40 minut
Celkem za tri dny: 15 hodin, 12 minut
```

Další příklad

Mnohé, co se učíte o funkcích a strukturách v C++, se přenáší do tříd C++, takže se vyplatí podívat se na další příklad. Tentokrát se budeme zabývat prostorem místo časem. Příklad zvláště definuje dvě struktury, které představují dva různé způsoby popisu pozice, a potom vyvíjí funkce na konverzi jedné formy do druhé a ukazuje výsledek. Tento příklad je trochu více matematický než poslední, ale nemusíte rozumět matematice, abyste porozuměli C++.

Předpokládejme, že chcete popsat pozici bodu na obrazovce nebo na mapě vzhledem k nějakému počátku. Jeden způsob je stanovit horizontální a vertikální posunutí bodu od počátku. Matematici tradičně používají symbol `x` pro horizontální posunutí a `y` pro vertikální. Viz obrázek 7.6. Společně `x` a `y` vytvářejí pravoúhlé souřadnice. Můžete vytvořit strukturu, která sestává ze dvou souřadnic na stanovení pozice:



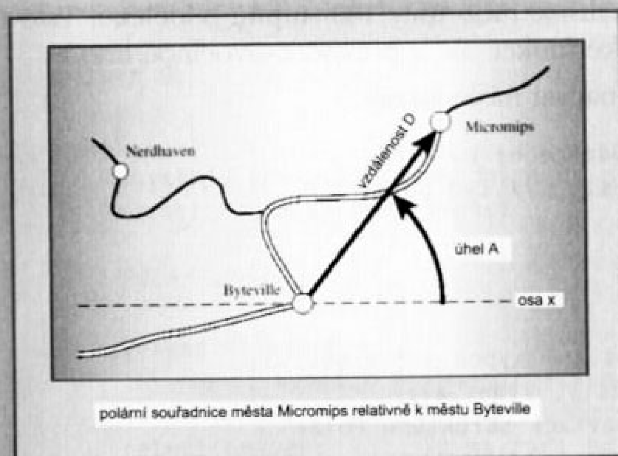
Obrázek 7.6 Pravoúhlé souřadnice

```
struct rect
  {
        double x;              // horizontální vzdálenost od počátku
```

```
        double y:               // vertikální vzdálenost od počátku
    };
```

Druhý způsob popisu pozice bodu je stanovit, jak daleko je od počátku a ve kterém je směru (například 40 stupňů severně od počátku). Tradičně matematici měří úhly proti směru hodinových ručiček od kladné horizontální osy. Viz obrázek 7.7. Společně vzdálenost a úhel vytvářejí polární souřadnice. Pro znázornění tohoto pohledu na pozici můžete použít další strukturu:

```
struct Polar
{
        double distance:    // vzdálenost od počátku
        double angle:       // směr od počátku
};
```



Obrázek 7.7 Polární souřadnice

Sestrojme funkci, která zobrazuje obsah struktury typu `Polar`. Matematické funkce v knihovně C++ předpokládají, že jsou úhly v radiánech, takže úhly měříme v těchto jednotkách. Ale pro zobrazovací účely konvertujeme míru v radiánech na stupně. To znamená násobením `180/(pi)`, což je přibližně `57.29577951`. Zde je funkce:

```
// ukaž polární souřadnice konvertováním úhlů na stupně
void show_polar (Polar dapos)
{
        const double Rad_to_deg = 57.29577951;

        cout << "vzdalenost = " << dapos.distance;
        cout << ", uhel = " << dapos.angle * Rad_to_deg;
        cout << " stupnu\n";
}
```

Všimněte si, že formální proměnná je typu `Polar`. Když předáte funkci strukturu `Polar`, obsah se kopíruje do struktury `dapos` a funkce potom používá tuto kopii pro svou

práci. Protože je dapos struktura, funkce používá k identifikaci členů struktury operátor příslušnosti (tečka) (viz kapitola 4).

Pokusme se dále o něco ambicióznějšího a napišme funkci, která konvertuje pravoúhlé souřadnice na polární. Napište funkci tak, že se jí předává struktura rect a navrací do volajícího programu strukturu Polar. To vyžaduje použití funkcí z matematické knihovny, takže program musí zahrnout hlavičkový soubor math.h. Na některých systémech musíte také říct kompilátoru, aby natáhl matematickou knihovnu (viz kapitola 1, „Začínáme"). Pro získání vzdálenosti z vertikální a horizontální komponenty můžete použít Pythagorovu větu:

```
distance = sqrt( x * x + y * y)
```

Funkce atan2() z matematické knihovny počítá z hodnot x a y úhel:

```
angle = atan2(y, x)
```

(Existuje také funkce atan(), ale ta nerozlišuje mezi úhly 180 stupňů odděleně. Tato neurčitost se dále nevyžaduje v matematické funkci, ale v průvodci divočinou ano.)

Jsou-li dány tyto vzorce, můžete funkci napsat následovně:

```
// konvertuje pravoúhlé souřadnice na polární
Polar rect_to_polar(rect xypos)   // typ polar
{
    Polar answer;

    answer.distance =
        sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;       // navrací strukturu Polar
}
```

Nyní jsou funkce připraveny, napsání zbytku programu je jednoduché. Výpis programu 7.11 ukazuje výsledek.

Výpis programu 7.11 strctfun.cpp

```
// strctfun.cpp — funkce s parametrem typu struktura
#include <iostream>
#include <cmath>
using namespace std;

// šablony struktur
struct Polar
{
    double distance;// vzdálenost od počátku
    double angle;   // směr od počátku
};
struct rect
{
    double x;       // horizontální vzdálenost od počátku
    double y;       // vertikální vzdálenost od počátku
```

```
};

// prototypy
Polar rect_to_polar(rect xypos);
void show_polar(Polar dapos);
int main()
{
    rect rplace;
    Polar pplace;

    cout << "Zadejte hodnoty x a y: ";
    while (cin >> rplace.x >> rplace.y)  // úhledné použití cin
    {
        pplace = rect_to_polar(rplace);
        show_polar(pplace);
        cout << "Dalsi dve cisla (q na ukoceni): ";
    }
return 0;
}

// konveruje pravoúhlé souřadnice na polární
Polar rect_to_polar(rect xypos)
{
    Polar answer;

    answer.distance =
        sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;       // navraci Polární strukturu
}

// zobrazení polárních souřadnic spolu s konverzí úhlu na stupně
void show_polar (Polar dapos)
{
    const double Rad_to_deg = 57.29577951;

    cout << "vzdalenost = " << dapos.distance;
    cout << ", uhel = " << dapos.angle * Rad_to_deg;
    cout << " stupne\n";
}
```

Kompatibilita:

Některé implementace stále používají hlavičkový soubor `math.h` namísto cmath. Některé kompilátory vyžadují explicitní instrukce na vyhledání matematické knihovny. Například starší verze g++ používají tento příkazový řádek:

```
g++ structfun.C -lm
```

Zde je ukázka běhu programu:

```
Zadejte hodnoty x a y: 30 40
vzdalenost = 50, uhel = 53.1301 stupne
Dalsi dve cisla (q na ukoceni): -100 100
vzdalenost = 141.421, uhel = 135 stupne
Dalsi dve cisla (q na ukoceni): q
```

Poznámky k programu

O těchto dvou funkcích jsme již pojednávali, takže znovu pohlédneme na to, jak program používá na řízení cyklu `while` objekt `cin`:

```
while (cin >> rplace.x >> rplace.y)
```

Připomínáme, že `cin` je objekt třídy `istream`. Extrakční operátor (>>) je navržen takovým způsobem, že `cin >> rplace.x` je také objektem tohoto typu. Jak uvidíte v kapitole 10 „Práce se třídami", operátory třídy jsou implementovány funkcemi. Co se skutečně přihodí, když použijete `cin >> rplace.x` je, že program zavolá funkci, která navrací hodnotu typu `istream`. Aplikujte extrakční operátor na objekt `cin >> rplace.x` (`cin >> rplace.x >> rplace.y`) a znovu dostanete objekt třídy `istream`. Tedy celý testovací výraz cyklu `while` se nakonec vyhodnotí na `cin`, který, jak si můžete připomenout, používá-li se v kontextu s testovacím výrazem, se konvertuje na `booleovskou` hodnotu `true` nebo `false` v závislosti na tom, je-li vstup úspěšný nebo ne. V tomto cyklu například `cin` očekává, že uživatel zavede dvě čísla. Zavedete-li místo toho `q`, jak jsme udělali, cyklus rozpozná, že `q` není číslo. Zanechá `q` ve vstupní frontě a vrátí hodnotu, která se konvertuje na `false` a ukončí cyklus.

Porovnejte tento přístup čtení čísel s přístupem ve výpisu programu 7.7:

```
for (i = 0; i < limit; i++)
    {
        cout << "Zadejte hodnotu #" << i + 1 << ": ";
        cin >> temp;
        if (temp < 0)
            break;
        ar[i] = temp;
    }
```

Abyste tento cyklus ukončili dříve, zadejte záporné číslo. To omezuje vstup na nezáporné hodnoty. Omezení vyhovuje potřebám programu, ale raději byste měli požadovat prostředek na ukončení cyklu, který by nevylučoval určité číselné hodnoty. Použití `cin >>` jako testovací podmínky eliminuje taková omezení, protože akceptuje všechny platné numerické vstupy. Také si pamatujte, že nenumerický vstup nastavuje chybovou podmínku, která zabraňuje čtení jakéhokoli dalšího vstupu. Potřebuje-li program následná data do vstupního cyklu, musíte na resetování vstupu použít `cin.clear()`, jak se popisuje v kapitolách 6 a 16, a možná se potom budete muset zbavit porušeného vstupu jeho přečtením.

Předání adres struktur

Předpokládejme, že chcete uspořit čas a prostor předáním adresy struktury namísto předání celé struktury. To vyžaduje přepsání funkcí, aby na struktury používaly ukazatele. Nejprve se podíváme, jak přepsat funkci `show_polar()`. Musíte provést tři změny:

- Když voláte funkci, spíše jí předejte adresu struktury (`&pplace`) než strukturu samotnou (`pplace`).
- Deklarujte formální parametr, který má být ukazatel-na-`Polar`, to jest typ `Polar *`. Protože by funkce neměla modifikovat strukturu, použijte modifikátor `const`.
- Protože je formální parametr ukazatelem místo strukturou, použijte raději operátor nepřímého adresování (->) místo operátoru příslušnosti (tečka).

Po těchto změnách vypadá funkce takto:

```
// ukazuje polární souřadnice, konvertujíc úhel na stupně
  void show_polar (const Polar * pda)
  {
      const double Rad_to_deg = 57.29577951;

      cout << "vzdalenost = " << pda->distance;
      cout << ", uhel = " << pda->angle * Rad_to_deg;
      cout << " stupnu\n";
  }
```

Nyní změňme `rect_to_polar`. To je komplikovanější, protože původní funkce `rect_to_polar` navrací strukturu. Abychom plně využili výhody schopnosti ukazatele, měli byste ho použít namísto návratové hodnoty. Způsob, jak to udělat, je předat funkci dva ukazatele. První ukazuje na strukturu, která se má konvertovat a druhý ukazuje na strukturu, která má být platnou konverzí. Namísto navrácení nové struktury funkce *modifikuje* existující strukturu ve volající funkci. Z toho důvodu je první parametr `konstantním` ukazatelem a druhý `není`. Jinak na ukazatelové parametry použijte stejné principy, které jste použili na `show_polar()`. Výpis programu 7.12 ukazuje přepracovaný program.

Výpis programu 7.12 strctptr.cpp

```
// strctptr.cpp - funkce s parametry typu ukazatel na strukturu
#include <iostream>
#include <cmath>
using namespace std;

// šablony struktur
struct Polar
{
    double distance;     // vzdálenost od počátku
    double angle;        // směr od počátku
};
struct rect
{
    double x;          // horizontální vzdálenost od počátku
```

```
    double y;           // vertikální vzdálenost od počátku
};

// prototypy
void rect_to_polar(const rect * pxy, Polar * pda);
void show_polar (const Polar * pda);

int main()
{
    rect rplace;
    Polar pplace;

    cout << "Zadejte hodnoty x a y: ";
    while (cin >> rplace.x >> rplace.y)
    {
        rect_to_polar(&rplace, &pplace);    // předání adresy
        show_polar(&pplace);            // předání adresy
        cout << "Dalsi dve cisla (q na ukonceni): ";
    }
    return 0;
}

// konvertuje pravoúhlé souřadnice na polární
void rect_to_polar(const rect * pxy, Polar * pda)
{
    pda->distance =
        sqrt(pxy->x * pxy->x + pxy->y * pxy->y);
    pda->angle = atan2(pxy->y, pxy->x);
}
// ukazuje souřednice struktury Polar a konvertuje úhel na stupně
void show_polar (const Polar * pda)
{
    const double Rad_to_deg = 57.29577951;

    cout << "vzdalenost = " << pda->distance;
    cout << ", uhel = " << pda->angle * Rad_to_deg;
    cout << " stupne\n";
}
```

Kompatibilita:

Některé implementace stále používají namísto novějšího hlavičkového souboru `cmath` soubor `math.h`. Některé kompilátory vyžadují k vyhledání matematické knihovny explicitní instrukce.

Z hlediska uživatele se program ve výpisu programu 7.12 chová jako program ve výpisu programu 7.11. Skrytý rozdíl je, že 7.11 pracuje s kopií struktury, zatímco 7.12 používá ukazatel na původní strukturu.

Rekurze

A nyní úplně něco jiného. Funkce v C++ má zajímavou vlastnost, může volat sama sebe. (Na rozdíl od C však C++ nedovoluje funkci `main()` volat samu sebe.) Tato schopnost se nazývá *rekurze*. Rekurze je důležitý prostředek pro určité typy programování, jako například umělá inteligence, ale my se podíváme pouze povrchně (vykonstruovaná povrchnost) na to, jak funguje.

Když rekurzivní funkce volá sama sebe, pak nově vyvolaná funkce volá opět sama sebe a tak dále až do nekonečna, dokud programový kód neobsahuje něco na ukončení tohoto řetězce volání. Obvyklým způsobem je udělat rekurzivní volání částí příkazu `if`. Například rekurzivní funkce typu `void`, která se jmenuje `recurs()`, může mít takovýto tvar:

```
void recurs(seznamparametrů)
{
        příkazy1
        if (test)
                recurs(příkazy)
        příkazy2
}
```

Díky štěstí nebo prozíravosti se *test* nakonec stane nepravdivý a řetěz volání se přeruší.

Rekurzivní volání produkuje spletitý řetěz událostí. Pokud zůstává příkaz `if` pravdivý, každé volání `recurs()` provádí `příkazy1` a potom vyvolává její nové ztělesnění bez dosažení `příkazy2`. Až se příkaz `if` stane nepravdivým, aktuální volání pokračuje na `příkazy2`. Potom, když aktuální volání končí, řízení programu se navrací na předchozí verzi `recurs()`, která ji vyvolala. Potom tato verze `recurs()` dokončí provedení své sekce `příkazy2` a končí s navrácením řízení na předchozí volání, a tak dále. Tedy, jestliže `recurs()` podstoupí pět rekurzivních volání, nejprve se provede sekce `příkazy1` pětkrát v pořadí, ve kterém se funkce volaly a potom se provede pětkrát sekce `příkazy2` v opačném pořadí než byly funkce volány. Po proběhnutí pěti úrovní rekurze potom program musí vycouvat přes pět stejných úrovní. Výpis programu 7.13 toto chování ukazuje.

Výpis programu 7.13 recur.cpp

```
// recur.cpp – použiti rekurze
#include <iostream>
using namespace std;
void countdown(int n);

int main()
{
    countdown(4);              // volá rekurzivní funkci
    return 0;
}

void countdown(int n)
{
    cout << "Pocitani smerem dolu ... " << n << "\n";
```

```
    if (n > 0)
        countdown(n-1);        // funkce volá sama sebe
    cout << n << ": Kaboom!\n";
}
```

Zde je výstup:

```
Pocitani smerem dolu ... 4 - začátek přidávání úrovní rekurze
Pocitani smerem dolu ... 3
Pocitani smerem dolu ... 2
Pocitani smerem dolu ... 1
Pocitani smerem dolu ... 0
0: Kaboom!                - úroveň 5 – začátek návratu přes sérii volání
1: Kaboom!                - úroveň 4
2: Kaboom!                - úroveň 3
3: Kaboom!                - úroveň 2
4: Kaboom!                - úroveň 1
```

Všimněte si, že každé rekurzivní volání vytváří svou vlastní sadu proměnných, takže jakmile program dosáhne pátého volání, má pět jednotlivých proměnných, které se jmenují `n` a každá má jinou hodnotu.

Rekurze je zvláště užitečná v situacích, které volají po opakovaném rozdělení úlohy na dvě menší, podobné části. Například uvažujme tento přístup na nakreslení pravítka. Označte dva konce, lokalizujte a označte střed. Potom použijte stejný postup na levou polovinu pravítka a potom na pravou. Chcete-li další rozdělení, použijte stejný postup na každou nynější sekci. Tento rekurzivní přístup se občas nazývá *strategie rozděl a panuj*. Výpis programu 7.14 tento přístup ukazuje pomocí funkce `subdivide()`. Používá řetězec, který je na počátku naplněný mezerami kromě znaku | na každém konci. Hlavní program používá šestkrát cyklus na vyvolání funkce `subdivide()`, pokaždé zvyšuje počet úrovní rekurze a tiskne výsledný řetězec. Tedy každý řádek výstupu představuje dodatečnou úroveň rekurze.

Výpis programu 7.4 ruler.cpp

```
// ruler.cpp - použití rekurze na rozdělení pravítka
#include <iostream>
using namespace std;
const int Len = 66;
const int Divs = 6;
void subdivide(char ar[], int low, int high, int level);
int main()
{
    char ruler[Len];
    int i;
    for (i = 1; i < Len - 2; i++)
        ruler[i] = ' ';
    ruler[Len - 1] = '\0';
    int max = Len - 2;
    int min = 0;
    ruler[min] = ruler[max] = '|';
```

```
    cout << ruler << "\n";
    for (i = 1; i <= Divs; i++)
    {
        subdivide(ruler,min,max, i);
        cout << ruler << "\n";
        for (int j = 1; j < Len - 2; j++)
            ruler[j] = ' '; // nastavení prázdného pravítka
    }

    return 0;
}
void subdivide(char ar[], int low, int high, int level)
{
    if (level == 0)
        return;
    int mid = (high + low) / 2;
    ar[mid] = '|';
    subdivide(ar, low, mid, level - 1);
    subdivide(ar, mid, high, level - 1);
}
```

Zde je výstup programu:

```
|                                                               |
|                               |                               |
|               |               |               |               |
|       |       |       |       |       |       |       |       |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

Poznámky k programu

Funkce `subdivide()` používá proměnnou, která se nazývá `level` a řídí úroveň rekurze. Když funkce volá sama sebe, redukuje úroveň o 1 a funkce, která má úroveň 0 končí. Všimněte si, že `subdivide()` volá sama sebe dvakrát, jednou pro levou a jednou pro pravou sekci. Původní střed se stane pravým koncem jednoho a levým druhého volání. Všimněte si, že počet volání roste geometricky. To jest, jedno volání generuje dvě, což generuje čtyři, což generuje osm a tak dále. Proto je volání 6 úrovní schopno naplnit 64 prvků ($2^6 = 64$).

Ukazatele na funkce

Žádné pojednání o funkcích v C nebo C++ by nebylo úplné bez zmínky o ukazatelích na funkce. Na toto téma se zběžně podíváme a úplné vysvětlení možností ponecháme pro pokročilejší učebnice.

Funkce, podobně jako datové položky, mají adresy. Adresa funkce je v paměti adresou, na které je uložen strojový kód jazyka, kde funkce začíná. Obvykle pro nás nebo uživatele není důležité ani užitečné znát tuto adresu, ale pro program to užitečné může být. Například je možné napsat funkci, která přijímá adresu jiné funkce jako parametr. To prv-

ní funkci umožňuje nalézt druhou a vykonat ji. Tento přístup je méně šikovný než když jednoduše první funkce zavolá druhou přímo, ale otevírá možnost předání různých adres funkcí v různé době. To znamená, že první funkce může v různé době použít různé funkce.

Základy ukazatelů na funkce

Vyjasněme tento postup pomocí příkladu. Předpokládejme, že chcete navrhnout funkci, která odhaduje množství času na napsání zadaného počtu řádků programového kódu a od různých programátorů chcete, aby tuto funkci použili. Část programového kódu `estimate()` bude pro všechny uživatele stejná, ale povolí každému programátorovi poskytnout jeho nebo její algoritmus na odhad času. Mechanismus bude spočívat v předání adresy určitého algoritmu funkce, který chce programátor použít, funkci `estimate()`. Pro implementaci tohoto plánu musíte být schopni udělat následující:

- Vzít adresu funkce.
- Deklarovat ukazatel na funkci.
- Použít ukazatel na funkci na vyvolání funkce.

Získání adresy funkce

Převzetí adresy funkce je jednoduché: pouze použijete jméno funkce bez zadních závorek. To jest, je-li `think()` funkce, potom je `think` její adresou. Abyste předali funkci jako parametr, předejte jméno funkce. Ujistěte se, že rozlišujete mezi předáním adresy funkce a její návratovou hodnotou:

```
process(think);    // předává adresu think() do process()
thought(think());  // předává návratovou hodnotu think() do thought()
```

Volání funkce `process()` umožňuje vyvolat funkci `think()` z funkce `process()`. Volání `thought()` nejprve vyvolá funkci `think()` a potom předá její návratovou hodnotu funkci `thought()`.

Deklarování ukazatele na funkci

Když jste deklarovali ukazatele na datové typy, deklarace musela přesně specifikovat, na jaký typ ukazatel ukazuje. Podobně musí ukazatel na funkci specifikovat, na jaký typ funkce ukazatel ukazuje. To znamená, že by deklarace měla identifikovat návratový kód funkce a její signaturu (její seznam parametrů). To jest, deklarace by nám měla o funkci říct stejné věci jako prototyp. Například předpokládejme, že Pam LeCoder napsal funkci na odhad času, která měla následující prototyp:

```
double pam(int);  // prototyp
```

Tady je deklarace odpovídajícího typu ukazatele:

```
double (*pf)(int);  // pf ukazuje na funkci, která přijímá
                    // jeden parametr typu int
                    // a navrací typ double
```

Všimněte si, že to vypadá téměř jako deklarace `pam()`, kde `(*pf)` hraje roli `pam`. Protože je pam funkce, je jí i (*pf). A jestliže je `(*pf)` funkce, potom je `pf` ukazatel na funkci.

Tip:

Obecně, abyste deklarovali ukazatel na určitý druh funkce, nejprve můžete napsat prototyp regulérní funkce požadovaného druhu a potom nahradit jméno funkce výrazem ve tvaru (`*pf`). To z `pf` vytvoří ukazatel na funkci tohoto typu.

Deklarace vyžaduje kolem `*pf` závorky na zajištění správné priority operátoru. Závorky mají vyšší prioritu než operátor *, takže `*pf(int)` znamená, že `pf()` je funkce, která vrací ukazatel, zatímco `(*pf)(int)` znamená, že `pf` je ukazatel na funkci:

```
double (*pf)(int); // pf ukazuje na funkci, která navrací double
double *pf(int);   // pf() je funkce, která navrací ukazatel-na-double
```

Jakmile jste jednou řádně deklarovali `pf`, můžete mu přiřadit adresu odpovídající funkce:

```
double pam(int);
double (*pf)(int);
pf = pam;            // pf nyní ukazuje na funkci pam()
```

Všimněte si, že `pam()` musí odpovídat `pf` jak popisem, tak typem návratu. Kompilátor odmítne neodpovídající přiřazení:

```
double ned(double);
int ted(int);
double (*pf)(int);
pf = ned;       // chybně – neodpovídající popis
pf = ted;       // chybně – neodpovídající návratový typ
```

Vraťme se k funkci `estimate()`, o které jsme se zmínili dříve. Předpokládejme, že jí chcete předat počet řádků programového kódu, který se má napsat a adresu odhadovacího algoritmu, jako například funkci `pam()`. Potom by mohla mít následující prototyp:

```
void estimate(int lines, double (*pf)(int));
```

Deklarace říká, že druhý parametr je ukazatel na funkci, která má atribut typu `int` a návratovou hodnotu typu `double`. Aby `estimate()` použila funkci `pam()`, předejte jí její adresu:

```
estimate(50, pam); // funkční volání říká estimate(), aby použila pam()
```

Jasněji, choulostivá část týkající se použití ukazatelů na funkce je napsání prototypů, zatímco předání adres je velmi jednoduché.

Použití ukazatelů na vyvolání funkce

Nyní jsme se dostali ke konečné části postupu, který spočívá v použití ukazatele na vyvolání funkce, na kterou se ukazuje. Záchytný bod je v deklaraci ukazatele. Tam, připomínáme, `(*pf)` hraje stejnou roli, jako jméno funkce. Tedy vše, co musíme udělat, je použít `(*pf)`, jako by to bylo jméno funkce:

```
double pam(int);
```

```
double (*pf)(int);
pf = pam;              // pf nyní ukazuje na funkci pam()
double x = pam(4);     // volání pam() používá jméno funkce
double y = (*pf)(5);   // volání pam() používá ukazatel pf
```

Dokonce vám C++ dovoluje použít `pf`, jakoby to bylo jméno funkce:

```
double y = pf(5);      // také volání pam() pomocí ukazatele pf
```

Použijeme první tvar. Je strašnější, ale poskytuje silnou vizuální připomínku, že programový kód používá ukazatel na funkci.

Historie proti logice

Ach ta syntaxe! Jak mohou být pf a (`*pf`) ekvivalentní? Historicky vzato, jedna myšlenková škola zastává názor, že je to z toho důvodu, protože pf ukazuje na funkci, je `*pf` funkce; a proto byste měli pro volání funkce použít (`*pf`) (). Druhá škola zastává názor, že jelikož jméno funkce je ukazatel na funkci, ukazatel by měl fungovat jako její jméno; z toho důvodu byste měli `pf`() použít jako volání funkce. C++ zastává kompromisní stanovisko, že obě formy jsou správné, nebo alespoň mohou být povoleny, i když jsou jedna s druhou logicky neslučitelné. Předtím než příliš krutě odsoudíte tento kompromis uvažte, že schopnost zastávat názory, které logicky nepatří do jednoho celku, je charakteristickým znakem lidského myšlení.

Výpis programu 7.15 ukazuje, jak použít ukazatele v programu. Volá funkci `estimate()` dvakrát, jednou jí předává adresu funkce `betsy()`, jednou `pam()`. V prvním případě funkce `estimate()` používá `betsy()` k výpočtu nezbytného počtu hodin a ve druhém k tomuto výpočtu používá `pam()`. Návrh podporuje budoucí rozvoj programu. Když Ralph vyvine svůj vlastní algoritmus na odhad času, nemusí přepisovat `estimate()`. Místo toho pouze musí dodat svou vlastní funkci `ralph()` a zajistí, že má správnou signaturu a návratový typ. Samozřejmě přepsání funkce `estimate()` není obtížný úkol, ale stejný postup se používá u složitějších programových kódů. Metoda ukazatele na funkci také Ralphovi dovoluje modifikovat chování funkce `estimate()`, dokonce i když nemá přístup k jejímu zdrojovému kódu.

Výpis programu 7.15 fun_ptr.cpp

```
// fun_ptr.cpp - ukazatele na funkce
#include <iostream>
using namespace std;
double betsy(int);
double pam(int);

// druhý parametr je ukazatel na funkci typu double, která
// přijímá dva parametry typu int
void estimate(int lines, double (*pf)(int));

int main()
{
```

```
    int code;

    cout << "Kolik radku programoveho kodu potrebujes? ";
    cin >> code;
    cout << "Tady je odhad od Betsy:\n";
    estimate(code, betsy);
    cout << "Tady je odhad od Pam:\n";
    estimate(code, pam);
    return 0;
}
double betsy(int lns)
{
    return 0.05 * lns;
}

double pam(int lns)
{
    return 0.03 * lns + 0.0004 * lns * lns;
}

void estimate(int lines, double (*pf)(int))
{
    cout << lines << " radku zabere ";
    cout << (*pf)(lines) << " hodin\n";
}
```

Zde jsou ukázky dvou běhů programu:

```
Kolik radku programoveho kodu potrebujes? 30
Tady je odhad od Betsy:
30 radku zabere 1.5 hodin
Tady je odhad od Pam:
30 radku zabere 1.26 hodin

Kolik radku programoveho kodu potrebujes? 100
Tady je odhad od Betsy:
100 radku zabere 5 hodin
Tady je odhad od Pam:
100 radku zabere 7 hodin
```

Shrnutí

Funkce znamenají v C++ programové moduly. Abyste mohli funkci použít, musíte poskytnout definici a prototyp a musíte uplatnit funkční volání. Definice funkce je programový kód, který implementuje, co funkce dělá. Prototyp funkce popisuje její rozhraní: kolik a jaký druh hodnot se jí má předat a jaký druh návratového typu, je-li nějaký, se z ní má získat. Volání funkce způsobí, že jí program předá parametry a přesune jeho provádění na programový kód funkce.

Standardně se funkcím v C++ předávají parametry hodnotou. To znamená, že formální parametry v definici funkce jsou nové proměnné, které se inicializují hodnotami poskytnutými voláním funkce. Tedy funkce v C++ chrání integritu původních dat tím, že pracují s jejich kopiemi.

C++ zachází s parametrem jména pole jako s adresou jeho prvního prvku. Technicky je to stále předání parametru hodnotou, protože ukazatel je kopií původní adresy, ale funkce ho používá na přístup k obsahu původního pole. Když deklarujeme pro funkci formální parametry (a pouze tehdy), pak jsou dvě následující deklarace ekvivalentní:

```
jménoTypu arr[];
jménoTypu * arr;
```

Obě znamenají, že `arr` je ukazatel na `jménoTypu`. Když píšete programový kód funkce, můžete `arr` použít, jako by to bylo jméno pole na přístup k prvkům: `arr[i]`. Dokonce i když předáváte ukazatele, můžete ochránit integritu původních dat deklarováním formálního parametru, který je ukazatelem na typ `const`. Protože předání adresy pole nesděluje žádnou informaci o jeho velikosti, měli byste obvykle předat jeho velikost formou odděleného parametru.

C++ zobrazení řetězců umožňuje třemi způsoby: pole znaků, řetězcová konstanta a ukazatel na řetězec. Všechny jsou typu `char*` (ukazatel-na-`znak`), takže se funkci předávají jako parametr typu `char*`. C++ používá na ukončení řetězců prázdný znak (`\0`) a řetězcové funkce ho testují, aby určily konec libovolného řetězce, který zpracovávají.

C++ zachází se strukturami stejně, jako se základními typy, což znamená, že je můžete předávat hodnotou a používat je jako návratové typy funkcí. Avšak, je-li struktura veliká, bylo by účinnější předávat ukazatel na strukturu a nechat funkci pracovat s původními daty.

Funkce v C++ může být rekurzivní; to znamená, že programový kód určité funkce může zahrnovat volání funkce samotné.

Jméno funkce v C++ funguje jako její adresa. Použitím parametru funkce, který je ukazatelem na funkci, jí můžete předat jméno další funkce, chcete-li, aby ji první funkce vyvolala.

Opakovací otázky

1. Jaké jsou tři kroky při použití funkce?
2. Vytvořte prototypy funkcí, které vyhovují následujícím popisům:
   a) `igor()` nepřijímá žádné parametry, ani nenavrací hodnotu.
   b) `tofu()` přijímá parametr typu `int` a vrací hodnotu typu `float`
   c) `mpg()` přijímá dva parametry typu `double` a navrací `double`
   d) `summation()` přijímá jméno pole typu `long` a jeho velikost jako hodnotu a navrací hodnotu typu `long`
   e) `doctor()` přijímá řetězcový parametr (řetězec se nemá modifikovat) a navrací hodnotu typu `double`

f) `ofcourse()` přijímá jako parametr strukturu typu boss a nic nenavrací

g) `plot()` přijímá jako parametr ukazatel na strukturu `map` a navrací řetězec

3. Napište funkci, která přijímá dva parametry: jméno pole typu `int`, jeho velikost hodnotu typu `int`. Nechť funkce nastaví každý prvek pole hodnotou typu `int`.
4. Napište funkci, která přijímá jako parametry jméno a velikost pole typu `double` a navrací jeho největší hodnotu. Pamatujte si, že by funkce neměla měnit obsah pole.
5. Proč nemůžeme použít kvalifikátor `const` na parametry funkce, které patří mezi základní typy?
6. Výpis programu 7.7 používá na ukončení vstupního cyklu hodnotu záporného majetku. Předpokládejme namísto toho, že by na ukončení vstupního cyklu používala nenumerický vstup. Přepište funkci `fill_array()` tak, aby vyhovovala těmto novým cílům návrhu.
7. Jaké jsou tři tvary řetězců ve stylu C, které se mohou přijmout programem v C++?
8. Napište funkci, která má tento prototyp:

   ```
   int replace(char * str, char c1, char c2);
   ```

   Nechť funkce nahradí každý výskyt `c1` v řetězci `str` pomocí `c2` a nechť navrátí počet záměn, které provádí.
9. Co znamená výraz `*"pizza"`? a co třeba `"taco"[2]`?
10. C++ vám umožňuje předat strukturu hodnotou a nechá vás předat její adresu. Je-li `glitz` strukturní proměnná, jak byste ji předali hodnotou? Jak byste předali její adresu? Jaké jsou spojitosti mezi těmito dvěma přístupy?
11. Funkce `judge()` má návratovou hodnotu typu `int`. Jako parametr přijímá adresu funkce, která má jako parametr `char` ukazatel na `const` a která také navrací typ `int`. Napište prototyp funkce.

Programovací cvičení

1. Napište program, který vás opakovaně žádá o zavedení páru čísel, dokud alespoň jedno z nich není nula. Pro každý pár by měl program použít funkci na výpočet jejich harmonické střední hodnoty. Funkce by měla do `main()` vrátit odpověď, která oznámí výsledek. Harmonická střední hodnota je převrácená hodnota průměru převrácených hodnot a může se počítat následovně: harmonická střední hodnota = `2.0*x*y/(x+y)`.
2. Napište program, který vás požádá o zavedení až 10 výsledků golfu, které se ukládají do pole. Uživateli byste měli poskytnout prostředky na ukončení vstupu před zavedením 10. skóre. Program by měl zobrazit všechna skóre na jeden řádek a oznámit střední hodnotu skóre. Na zacházení se vstupem, zobrazením a výpočtem střední hodnoty použijte tři oddělené funkce, které zpracovávají pole.
3. Zde je šablona struktury:

   ```
   struct box
   {
   ```

```
        char maker[40];
        float height;
        float width;
        float length;
        float volume;
};
```

a) Napište funkci, které se předá struktura `box` hodnotou a která zobrazí hodnotu každého členu.

b) Napište funkci, které se předá adresa struktury `box` a jež nastaví člen `volume` na součin dalších tří dimenzí.

c) Napište jednoduchý program, který tyto dvě funkce používá.

4. Definujte rekurzivní funkci, která přijímá celočíselný parametr a navrací jeho faktoriál. Připomínáme, že 3 faktoriál, psáno `3!`, se rovná `3 x 2!` atd., kde `0!` se definuje jako `1`. Obecně `n!=n*(n-1)!`. Otestujte ji v programu, který používá cyklus a umožňuje uživateli zavést různé hodnoty, jejichž faktoriál se vypočítá.

5. Napište program, který používá následující funkce:

   `Fill_arr()` přijímá parametry jméno pole typu `double` a jeho velikost. Požádá uživatele o zadání hodnot typu `double`, které mají být uloženy do pole. Zadávání vstupu se zastaví, když je pole plné nebo když uživatel zadá nenumerickou hodnotu a navrátí skutečný počet vstupů.

   `Show_array()` přijímá jako parametry jméno pole typu `double` a jeho velikost a zobrazí jeho obsah.

   `Reverse_array()` přijímá jako parametry jméno pole typu `double` a jeho velikost a obrátí pořadí hodnot uložených do pole.

   Program by měl naplnit pole, ukázat je, zaměnit pořadí všech prvků kromě prvního a posledního a ukázat pole.

6. Toto cvičení poskytuje praxi v psaní funkcí, které zacházejí s poli a řetězci. Následuje kostra programu. Doplňte ji podle poskytnutého popisu funkcí.

```
#include <iostream>
using namespace std;

const int SLEN = 30;
struct student {
    char fullname[SLEN];
    char hobby[SLEN];
    int ooplevel;
};
// getinfo() má dva parametry: ukazatel na první prvek pole struktury
// student a int, které představuje počet prvků pole.
// Funkce si vyžádá a ukládá data o studentech.
// Ukončí se, když je pole plné
// nebo když narazí na prázdný řádek místo jména studenta.
// Funkce vrací skutečný počet naplněných prvků pole.
int getinfo(student pa[], int n);
```

```
// display1() přijímá jako parametr strukturu student
// a zobrazuje její obsah
void display1(student st);

// display2() přijímá jako parametr adresu struktury student
// a zobrazuje její obsah
void display2(const student * ps);

// display3() přijímá jako parametry adresu prvního prvku
// pole struktur student a počet prvků pole
// a zobrazuje obsah struktur
void display3(const student pa[], int n);

int main()
{
    cout << "Zadejte velikost tridy: ";
    int class_size;
    cin >> class_size;
    while (cin.get() != '\n')
        continue;

    student * ptr_stu = new student[class_size];
    int entered = getinfo(ptr_stu, class_size);
    for (int i = 0; i < entered; i++)
    {
        display1(ptr_stu[i]);
        display2(&ptr_stu[i]);
    }
    display3(ptr_stu, entered);
    delete [] ptr_stu;
    cout << "Hotovo\n";
    return 0;
}
```

7. Navrhněte funkci `calculate()`, která přijímá dvě hodnoty typu `double` a ukazatel na funkci, která přijímá dva parametry typu `double` a navrací `double`. Funkce `calculate()` by také měla být typu `double` a měla by navracet hodnotu, kterou funkce, na kterou se ukazuje, počítá pomocí parametrů. Předpokládejme, že máme tuto definici funkce `add()`:

```
double add(double x, double y)
{
    return x + y;
}
```

Potom zavolání funkce

```
double q = calculate(2.5, 10.4, add);
```

by mělo přimět `calculate()`, aby přijala hodnoty `2.5` a `10.4` pro funkci `add()` a potom navrátit návratovou hodnotu funkce `add()` (12.9).

Použijte tyto funkce v programu alespoň ještě s jednou dodatečnou funkcí, která má vlastnost `add()`. Program by měl použít cyklus, který by uživateli umožnil zadat dvojici čísel. Pro každou dvojici použijte `calculate()` na vyvolání `add()` a alespoň jedné další funkce. Jste-li odvážný, pokuste se vytvořit pole ukazatelů na funkce stylu `add()` a použijte cyklus na postupnou aplikaci `calculate()` na řadu funkcí pomocí těchto ukazatelů. Nápověda: Tady vidíte jak deklarovat takové pole tří ukazatelů:

```
double (*pf[3])(double, double);
```

Takové pole můžete inicializovat pomocí obvyklé syntaxe pro inicializaci pole a jméno funkce jako adresu.

KAPITOLA 8

Příběhy ve funkcích

Témata kapitoly:

- Vložené funkce
- Referenční proměnné
- Předávání parametrů funkcím odkazem
- Standardní parametry
- Přetěžování funkcí
- Funkční šablony
- Specializace ve funkčních šablonách
- Oddělená kompilace
- Paměťové třídy, rozsah platnosti a vazba
- Prostory jmen

Vybaveni poslední kapitolou nyní víte o funkcích v C++ dost, ale na pořadu je toho mnohem více. C++ poskytuje mnoho funkčních rysů, které se liší od dědictví z C. Tyto nové rysy zahrnují vložené funkce, předávání parametru odkazem, standardní hodnoty parametrů, překrývání funkcí (polymorfismus) a šablony funkcí. Tato kapitola pojednává o rozšířeních na funkce, které spadají do C++. Také zkoumá několikasouborové programy a varianty programových tříd C++, včetně jmen prostorů. Tato kapitola mnohem více, než jste se doposud učili, rozvíjí rysy, které se nalézají v C++ a nikoli v C, takže naznačuje váš první hlavní vpád do plusplusovitosti.

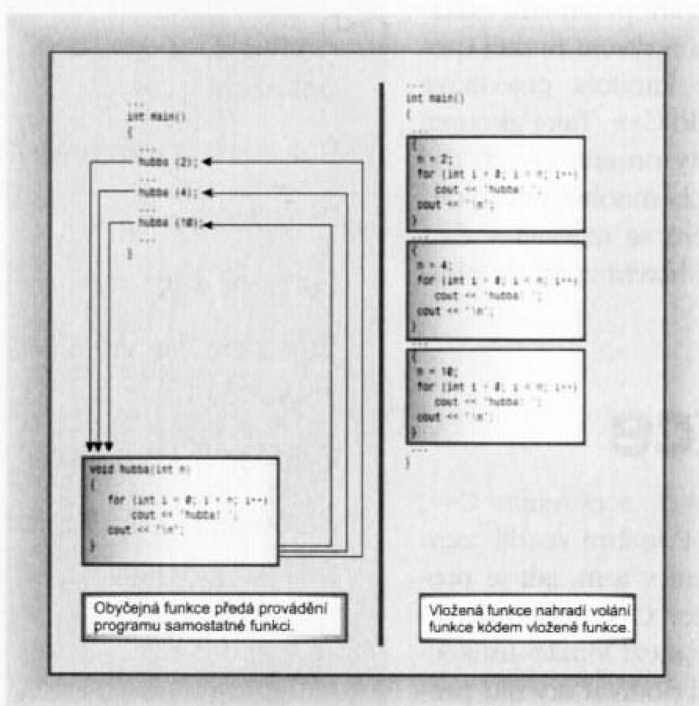
Vložené funkce

Začněme zkoumáním vložených funkcí, rozšířením C++, navržených pro zrychlení programů. Primární rozdíl mezi normálními a vloženými funkcemi není v tom, jak je programujete, ale v tom, jak je kompilátor C++ začleňuje do programu. Abyste porozuměli rozdílu mezi těmito funkcemi, musíte se z mnohem větší blízkosti podívat dovnitř programu, než jste to činili doposud. Udělejme to nyní.

Konečným produktem procesu kompilace je proveditelný program, který sestává z množiny instrukcí ve strojovém jazyce. Když startujete program, operační systém tyto instrukce zavádí do paměti počítače, takže každá instrukce má určitou paměťovou adresu. Počítač potom tyto instrukce krok za krokem provádí. Občas, napíšete-li cyklus nebo příkaz větvení, program bude přeskakovat instrukce, skákat zpět nebo dopředu na určitou adresu. Běžné volání funkce také od programu vyžaduje skákat na jinou adresu (adresu funkce) a potom zpět, když funkce skončí. Podívejme se na implementaci tohoto postupu trochu detailněji. Když program dosáhne instrukce volání funkce, uloží adresu paměti, která za ním bezprostředně následu-

je, nakopíruje parametry do zásobníku (blok paměti, který je pro tento účel rezervován), skočí na místo paměti, která označuje začátek umístění funkce, provede její programový kód (možná, že umístí do registru návratovou hodnotu) a potom skočí zpět na instrukci, jejíž adresa se uložila[1]. Skákání dozadu a dopředu a sledování kam skočit znamená, že se vynakládají režijní náklady, které se vztahují k uplynuté době.

Vložená funkce v C++ poskytuje alternativu. To je funkce, jejíž překompilovaný programový kód je „v řadě“ s dalším kódem programu. Tedy kompilátor nahrazuje volání funkce jejím odpovídajícím programovým kódem. Při vloženém kódu nemusí program skákat na jiné místo kvůli jeho provedení a potom skákat zpět. Vložené funkce tedy běží trochu rychleji než běžné, ale existuje trest ve formě plýtvání pamětí. Jestliže program volá vloženou funkci desetkrát, potom natahuje deset kopií vložené funkce do kódu. Viz obrázek 8.1.



Obrázek 8.1 Vložené funkce proti běžným funkcím

Vložení funkce byste měli používat selektivně. Zisk v rychlosti je obvykle minimální, pokud sama funkce není tak krátká, že čas, který je potřeba na její provedení, je srovnatelný s časem na odskok na funkci a zpět. V takovém případě je již funkce dostatečně rychlá, takže kdyby to byl pouze čas, který byste získali jako velkou část výhody, tak by musela být hlavním spotřebitelem v kritickém cyklu.

Abyste využili tyto vlastnosti, musíte udělat dvě věci:

- Opatřit definici funkce klíčovým slovem `inline`.
- Umístit její definici před všechny funkce, které ji volají.

[1] Je to jako opustit čtení nějakého textu pro zjištění, co říká poznámka pod čarou a potom po jejím ukončení návrat tam, kde jste v textu přestali číst.

Všimněte si, že musíte před nimi umístit celou definici (to znamená hlavičku funkce a celý její programový kód), ne pouze prototyp.

Kompilátor nemusí přijmout váš požadavek na vytvoření vložené funkce. Může se rozhodnout, že je funkce příliš velká nebo si všimne, že volá sama sebe (rekurze se u vložených funkcí nepovoluje) nebo tato vlastnost možná není implementována pro váš určitý kompilátor. Výpis programu 8.1 ukazuje techniku vkládání pomocí vložené funkce `square()`, která umocňuje na druhou svůj parametr. Všimněte si, že jsme umístili celou definici na jeden řádek. To se nevyžaduje, ale pokud se na ní nevejde, nemělo smysl o ní hovořit jako o vložené (doslova „řádkové") funkci.

Výpis programu 8.1 inline.cpp

```
// inline.cpp – použití inline funkce
#include <iostream>
using namespace std;
// inline funkce musí být definována před svým prvním použitím
inline double square(double x) { return x * x; }

int main()
{
    double a, b;
    double c = 13.0;

    a = square(5.0);
    b = square(4.5 + 7.5);    // může se předat výraz
    cout << "a = " << a << ", b = " << b << "\n";
    cout << "c = " << c;
    cout << ", c na druhou = " << square(c++) << "\n";
    cout << "Nyní c = " << c << "\n";
    return 0;
}
```

Zde je výstup:

```
a = 25, b = 144
c = 13, c na druhou = 169
Nyní c = 14
```

Výstup ukazuje, že se vložené funkci parametry předávají hodnotou, stejně, jako to dělají běžné funkce. Je-li parametr výrazem, jako například 4.5 + 7.5, funkci se předá hodnota výrazu, v tomto případě 12. To umožňuje, že je tento prostředek vkládání funkcí v C++ daleko kvalitnější oproti definicím maker v C. Všimněte si poznámky níže na téma Vložené funkce versus makra.

Vlastnosti vytváření prototypů v C++ jsou stále ještě ve hře, přestože program neposkytuje jednotlivé prototypy. To je proto, že celá definice, která se nachází před prvním použitím funkce, slouží jako prototyp. To znamená, že můžete použít `square()` s parametrem typu `int` nebo `long` a program automaticky přetypuje hodnotu na typ `double` před tím, než ji předá funkci.

Vložené funkce versus makra

Prostředek vkládání funkce je přídavkem C++. C používá na poskytnutí makra příkaz preprocesoru `#define`, což je nezralá implementace vloženého kódu. Zde je například makro na umocnění čísla na druhou:

```
#define SQUARE(X) X*X
```

Nepracuje pomocí předání parametrů, ale pomocí náhrady textu, kde X funguje jako symbolické návěští pro „parametr":

```
a = SQUARE(5.0); se nahradí pomocí a = 5.0*5.0;
b = SQUARE(4.5 + 7.5); se nahradí pomocí b = 4.5 + 7.5 * 4.5 + 7.5;
d = SQUARE(c++); se nahradí pomocí d = c++*c++;
```

Pouze první příklad pracuje správně. Záležitost můžete vylepšit pomocí liberálního použití závorek:

```
#define SQUARE(X) ((X)*(X))
```

Zůstává stále ještě problém, že se parametr nepředává makru hodnotou. Dokonce i s touto novou definicí, `SQUARE(c++)` inkrementuje `c` dvakrát, ale vložená funkce `square()` ve výpisu programu 8.1 `c` vyhodnotí, předá tuto hodnotu, aby se umocnila na druhou a potom jednou `c` inkrementuje.

Zde není úmyslem ukázat, jak psát makra v C. Spíše se předpokládá, že když jste používali makra v C na provádění služeb podobných funkcím, mohli byste uvažovat o jejich konverzi na vložené funkce v C++.

Referenční proměnné

C++ dodává do jazyka nový odvozený typ – referenční proměnnou. *Odkaz* je jméno pro předem definované proměnné, které funguje jako druhé jméno (alias) nebo alternativní jméno. Například, jestliže vytvoříte `twain` jako odkaz na proměnnou `clements`, můžete k reprezentaci proměnné střídavě použít jak `twain`, tak `clements`. Jaký užitek je z takového druhého jména? Má pomoci lidem, kterým vadí jejich volba proměnné? Možná, ale hlavní užitek odkazu je jeho použití jako formálního parametru ve funkci. Při použití odkazu jako parametru pracuje funkce s původními daty namísto s kopií. Odkazy poskytují vhodnou alternativu k ukazatelům na zpracování velkých struktur pomocí funkcí a jsou zásadní při návrhu tříd. Dříve než se podíváte, jak se odkazy s funkcemi používají, prozkoumejme nejprve základy jejich definování a použití. Mějte na paměti, že účelem následující diskuse je ukázat, jak odkazy pracují, ne jak se používají.

Vytvoření proměnné reference

Možná si vzpomenete, že C i C++ používají na indikaci adresy proměnné symbol `&`. C++ přiřazuje symbolu `&` dodatečný význam a tlačí ho do služeb na deklarování referencí. Například k vytvoření alternativního jména `rodents` pro proměnnou `rats` udělejte následující:

```
int rats;
int & rodents = rats;    // vytváří z rodents druhé jméno pro rats
```

V tomto kontextu není & adresovým operátorem. Místo toho slouží jako část identifikátoru typu. Právě tak jako `char *` znamená v deklaraci ukazatel-na-`char`, `int &` znamená odkaz-na-`int`. Deklarace odkazu vám umožňuje střídavě použít `rats` a `rodents`; obě se odkazují na stejnou hodnotu a stejnou lokaci paměti. Výpis programu 8.2 ukazuje pravdivost tohoto tvrzení.

Výpis programu 8.2 firstref.cpp

```
// firstref.cpp — definování a použití odkazu
#include <iostream>
using namespace std;
int main()
{
    int rats = 101;
    int & rodents = rats;    // rodents je odkaz
    cout << "krysy = " << rats;
    cout << ", hlodavci = " << rodents << "\n";
    rodents++;
    cout << "krysy = " << rats;
    cout << ", hlodavci = " << rodents << "\n";

    // některé implementace vyžadují přetypování následujících
    // adres na typ unsigned
    cout << "adresa krys = " << &rats;
    cout << ", adresa hlodavcu = " << &rodents << "\n";
return 0;
}
```

Všimněte si, že operátor & v příkazu

```
int & rodents = rats;
```

není adresovým operátorem, ale deklaruje, že `rodents` je typu `int &`, to jest odkazem na proměnnou `int`. Ale operátor v příkazu

```
cout<<", adresa hlodavcu = "<< &rodents << "\n";
```

je adresovým operátorem, kde `&rodents` představuje adresu proměnné, na kterou se `rodents` odkazuje. Zde je výstup programu:

```
krysy = 101, hlodavci = 101
krysy = 102, hlodavci = 102
adresa krys = 0065FE00, adresa hlodavcu = 0065FE00
```

Jak vidíte, obojí `rats` a `rodents` mají stejnou hodnotu a stejnou adresu. Zvýšení hodnoty `rodents` o 1 ovlivní obě proměnné. Přesněji, operace `rodents++` inkrementuje jedinou proměnnou, pro kterou máme dvě jména. (Pamatujte si, že i když tento příklad ukazuje, jak odkaz pracuje, nereprezentuje typické použití odkazu, které odpovídá funkčnímu parametru, zejména pro předání struktury nebo objektu. Na tato použití se podíváme docela brzy.)

Refrence jsou pro veterány, kteří přecházejí z C do C++, trochu matoucí, protože jsou utrápeni vzpomínkami na ukazatele, které se ještě ke všemu nějak liší. Například, můžete vytvořit jak referenci, tak ukazatel, které se odkazují na rats:

```
int rats = 101;
int & rodents = rats;   // rodents odkaz
int * prats = &rats;    // prats ukazatel
```

Potom byste mohli na střídačku s `rats` použít výrazy `rodents` a `*prats` a s `&rats` výrazy `&rodents` a `prats`. Z tohoto hlediska vypadá reference dost jako ukazatel v přestrojení, ve kterém se implicitně rozumí dereferenční operátor `*`. A ve skutečnosti je více méně odkazem. Ale mezi oběma zápisy existují rozdíly. Pro první je nezbytné odkaz inicializovat, když ho deklarujete; nemůžete ho deklarovat a později mu přiřadit hodnotu, což je způsob, který můžete použít pro ukazatel:

```
int rat;
int & rodent;
rodent = rat;    // Ne, toto nemůžete udělat.
```

Pamatujte:

Proměnnou typu reference byste měli inicializovat, když ji deklarujete.

Refrence je spíše jako `konstantní` ukazatel; musíte ji inicializovat, když ji vytvoříte a jednou, když zaručí svou oddanost určité proměnné, lpí na ní. Tedy

```
int & rodents = rats;
```

je v podstatě maskovaný zápis pro něco jako je toto:

```
int * const pr = &rats;
```

Zde hraje `rodents` stejnou úlohu jako `*pr`.

Výpis programu 8.3 ukazuje co se stane, když se pokusíte provést změnu loajality odkazu z proměnné `rats` na `bunnies`.

Výpis programu 8.3 secref.cpp

```
// secref.cpp – definování a použití odkazu
#include <iostream>
using namespace std;
int main()
{
    int rats = 101;
    int & rodents = rats;   // rodents je odkaz

    cout << "krysy = " << rats;
    cout << ", hlodavci = " << rodents << "\n";

    cout << "adresa krys = " << &rats;
    cout << ", adresa hlodavcu = " << &rodents << "\n";
```

```
    int bunnies = 50;
    rodents = bunnies;       // můžeme změnit odkaz?
    cout << "zajicci = " << bunnies;
    cout << ", krysy = " << rats;
    cout << ", hlodavci = " << rodents << "\n";

    cout << "adresa zajicku = " << &bunnies;
    cout << ", adresa hlodavcu = " << &rodents << "\n";
    return 0;
}
```

Zde je výstup z programu:

```
krysy = 101, hlodavci = 101
adresa krys = 0065FE00, adresa hlodavcu = 0065FE00
zajicci = 50, krysy = 50, hlodavci = 50
adresa zajicku = 0065FDF8, adresa hlodavcu = 0065FE00
```

Z počátku se `rodents` odkazuje na `rats`, ale potom se pokouší změnit odkaz na `bunnies`:

```
rodents = bunnies;
```

Na první pohled to vypadá, jako by toto úsilí bylo úspěšné, protože se hodnota 101 u `rodents` změnila na 50. Podrobnější sledování však ukazuje, že se i hodnota `rats` změnila na 50 a `rats` a `rodents` sdílejí stále stejnou adresu, která se liší od adresy `bunnies`. Protože `rodents` je alias pro `rats`, přiřazovací příkaz ve skutečnosti znamená následující:

```
rats = bunnies;
```

To znamená „přiřazení hodnoty proměnné `bunnies` proměnné `rats`". Zkrátka, odkaz můžete nastavit inicializací v deklaraci, nikoli přiřazením.

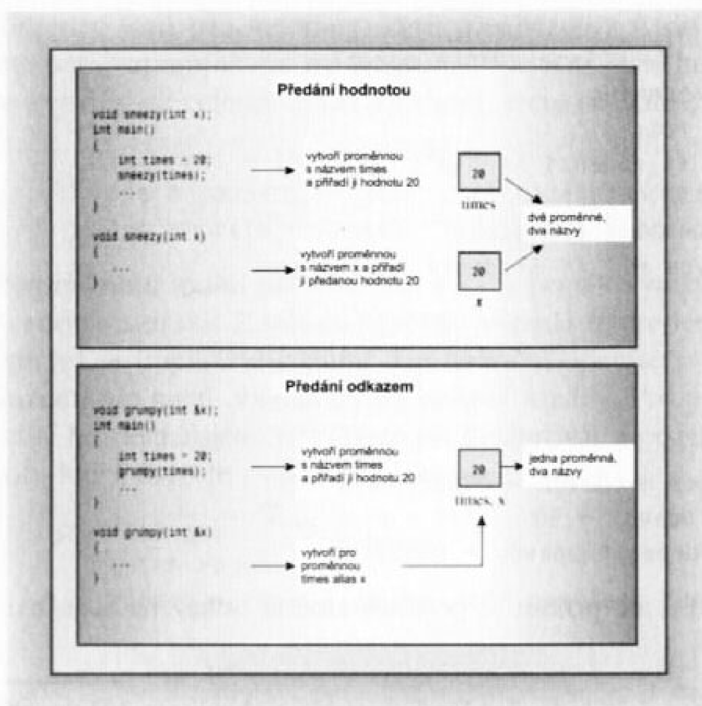
Předpokládejme, že se pokusíte provést následující:

```
int rats = 101;
int * pi = &rats;
int & rodents = *pi;
int bunnies = 50;
pi = &bunnies;
```

Inicializace `rodents` na `*pi` stanoví, že se `rodents` odkazuje na `rats`. Následná změna `pi` na ukazování na `bunnies` nemění skutečnost, že se `rodents` odkazuje na `rats`.

Reference jako funkční parametry

Nejčastěji se odkazy používají jako funkční parametry, vytvářejí ze jména proměnné ve funkci alias pro proměnnou ve volajícím programu. Tento postup předávání parametrů se nazývá *předávání odkazem*. Předávání odkazem umožňuje volané funkci přistupovat k proměnným volající funkce. Přidání této vlastnosti do C++ je odtržením od C, které předává pouze hodnotou. Připomínáme, že předání hodnotou má za následek, že volaná funkce pracuje s kopiemi hodnot volajícího programu. Viz obrázek 8.2. Samozřejmě vás C nechá obejít omezení předávání hodnotou použitím ukazatelů.



Obrázek 8.2 Předávání hodnotou a odkazem

Porovnejme použití referencí a ukazatelů na běžném počítačovém problému: záměna hodnot dvou proměnných. Zaměňovací funkce musí být schopna zaměnit hodnoty ve volajícím programu. To znamená, že obvyklý přístup předání proměnných hodnotou nepracuje, protože funkce skončí záměnou obsahů kopií původních proměnných místo proměnných samotných. Avšak když předáváte odkazem, funkce může pracovat s původními daty. Na přístup k původním datům můžete alternativně předat ukazatele. Výpis programu 8.4 ukazuje všechny tři metody, včetně té, která nepracuje, takže je můžete porovnat.

Výpis programu 8.4 swaps.cpp

```
// swaps.cpp — záměna pomocí odkazů a ukazatelů
#include <iostream>
using namespace std;
void swapr(int & a, int & b);   // a, b jsou aliasy pro proměnné typu int
void swapp(int * p, int * q);   // p, q jsou adresy proměnných typu int
void swapv(int a, int b);       // a, b jsou nové proměnné
int main()
{
    int wallet1 = 300;
    int wallet2 = 350;
    cout << "penezenka1 = $" << wallet1;
    cout << " penezenka2 = $" << wallet2 << "\n";

    cout << "Pouziti odkazu na zamenu obsahu:\n";
    swapr(wallet1, wallet2);        // předání proměnných
    cout << "penezenka1 = $" << wallet1;
    cout << " penezenka2 = $" << wallet2 << "\n";
```

```
    cout << "Pouziti ukazatelu na zamenu obsahu:\n";
    swapp(&wallet1, &wallet2); // pass addresses of variables
    cout << "penezenka1 = $" << wallet1;
    cout << " penezenka2 = $" << wallet2 << "\n";

    cout << "Pokus pouziti predani hodnotou:\n";
    swapv(wallet1, wallet2);     // předání hodnot proměnných
    cout << "penezenka1 = $" << wallet1;
    cout << " penezenka2 = $" << wallet2 << "\n";
    return 0;
}

void swapr(int & a, int & b)    // použití odkazů
{
    int temp;

    temp = a;         // použití a, b pro hodnoty proměnných
    a = b;
    b = temp;
}

void swapp(int * p, int * q)    // použití ukazatelů
{
    int temp;

    temp = *p;        // použití *p, *q pro hodnoty proměnných
    *p = *q;
    *q = temp;
}

void swapv(int a, int b)        // zkouška použití hodnot
{
    int temp;

    temp = a;        // použití a, b pro hodnoty proměnných
    a = b;
    b = temp;
}
```

Zde je výstup programu:

```
penezenka1 = $300 penezenka2 = $350          <- původní hodnoty
Pouziti odkazu na zamenu obsahu:
penezenka1 = $350 penezenka2 = $300          <- zaměněné hodnoty
Pouziti ukazatelu na zamenu obsahu:
penezenka1 = $300 penezenka2 = $350          <- hodnoty opět zaměněné
Pokus pouziti predani hodnotou:
penezenka1 = $300 penezenka2 = $350          <- záměna selhala
```

Jak jsme očekávali, metody s referencí a ukazatelem úspěšně zaměňují obsahy dvou peněženek, zatímco metoda předání hodnotou selhává.

Poznámky k programu

Za prvé si všimněte, jak se každá funkce volá:

```
swapr(wallet1, wallet2);            // předání proměnných
swapp(&wallet1, &wallet2);          // předání adres proměnných
swapv(wallet1, wallet2);            // předání hodnot proměnných
```

Předání odkazem (`swapr(wallet1, wallet2)`) a hodnotou (`swapv(wallet1, wallet2)`) vypadají identicky. Jediný způsob, podle kterého můžete říci, že `swapr()` předává odkazem je, že se podíváte na prototyp nebo definici funkce. Přítomnost adresního operátoru (`&`) určuje, kdy se funkci předává parametr adresou (`swapv(&wallet1, &wallet2)`). (Připomínáme, že typ deklarace `int *p` znamená, že `p` je ukazatel na `int`, a proto by měl být odpovídající parametr `p` adresou, jako například `&wallet1`.)

Dále porovnejme programový kód pro funkce `swapd()` (předání odkazem) a `swapv()` (předání hodnotou). Jediný zřejmý rozdíl mezi těmito dvěma funkcemi je, jak jsou deklarovány jejich parametry:

```
void swapr(int & a, int & b)
void swapv(int a, int b)
```

Vnitřní rozdíl je samozřejmě v tom, že ve `swapr()` slouží proměnné `a` a `b` jako druhá jména pro `wallet1 wallet2`, takže záměna `a` a `b` zamění `wallet1` a `wallet2`. Avšak ve `swapv()` proměnné `a` a `b` jsou nové proměnné, které kopírují hodnoty `wallet1` a `wallet2`, takže záměna `a` a `b` nemá žádný vliv na `wallet1` a `wallet2`. Nakonec porovnejme funkce `swapr()` (předání odkazem) a `swapp()` (předání ukazatelem). První rozdíl je v tom, jak jsou deklarovány parametry funkce:

```
void swapr(int & a, int & b)
void swapp(int * p, int * q)
```

Druhý rozdíl je, že verze ukazatele vyžaduje použití dereferenčního operátoru `*` při používání `p` a `q`.

Dříve jsme řekli, že byste měli při definování referenční proměnnou inicializovat. Parametry funkce typu odkaz můžete považovat za inicializované parametry, které byly předány funkčním voláním. To jest, funkční volání

```
swapr(wallet1, wallet2);
```

inicializuje formální parametr `a` na `wallet1` a formální parametr `b` na `wallet2`.

Vlastnosti a zvláštnosti referencí

Použití parametrů typu odkaz má několik fint, o kterých musíte vědět. Za prvé, uvažme výpis programu 8.5. K umocnění na třetí používá dvě funkce. Jedna přijímá parametr typu `double`, zatímco druhá odkaz na `double`. Skutečný programový kód umocnění na třetí je pro objasnění problému záměrně trochu neobvyklý.

Výpis programu 8.5 cubes.cpp

```
// cubes.cpp – běžné parametry a parametry typu odkaz
#include <iostream>
using namespace std;
double cube(double a);
double refcube(double &ra);
int main ()
{
    double x = 3.0;

    cout << cube(x);
    cout << " = treti mocnina ze " << x << "\n";
    cout << refcube(x);
    cout << " = treti mocnina z " << x << "\n";
    return 0;
}

double cube(double a)
{
    a *= a * a;
    return a;
}

double refcube(double &ra)
{
    ra *= ra * ra;
    return ra;
}
```

Zde je výstup:

```
27 = treti mocnina ze 3
27 = treti mocnina z 27
```

Všimněte si, že funkce `refcube()` modifikuje hodnotu `x` v `main()`, zatímco `cube()` ne, což nám připomíná, proč je předání hodnotou v normě. Proměnná `a` je vzhledem ke `cube()` lokální. Inicializuje se hodnotou `x`, ale změna `a` na ni nemá žádný vliv. Ale protože `ref-`

cube() používá parametr typu odkaz, změny, které provede s ra se skutečně provedou s x. Je-li vaším úmyslem, aby funkce používala předané informace bez modifikování a jestliže používáte odkaz, měli byste použít konstantní odkaz. Zde bychom například měli v prototypu a hlavičce funkce použít const:

```
double refcube(const double &ra);
```

Kdybychom to udělali, kompilátor by měl vygenerovat chybové hlášení, když by zjistil, že programový kód mění hodnotu ra.

Příležitostně, když budete potřebovat napsat funkci podle řádků tohoto příkladu, použijte raději předání hodnotou, než exotičtější předání odkazem. Parametry typu odkaz se stávají užitečné s velkými datovými jednotkami, jako například struktury a třídy, jak brzy uvidíte.

Funkce, kterým se předává hodnotou, jako například cube() ve výpisu programu 8.5, mohou použít mnoho druhů skutečných parametrů. Například všechna následující volání jsou platná:

```
double z = cube(x + 2.0);      // vyhodnotí x + 2.0, předá hodnotu
z = cube(8.0);                  // předá hodnotu 8.0
int k = 10;
z = cube(k);      // konvertuje hodnotu k na double, předá hodnotu
double yo[3] = { 2.2, 3.3, 4.4};
z = cube (yo[2]);               // předá hodnotu 4.4
```

Předpokládejme, že zkusíte podobné parametry pro funkce s referenčními parametry. Zdálo by se, že předání odkazu by mohlo více omezovat. Přece jenom, je-li ra alternativní jméno proměnné, pak by skutečný parametr měl být touto proměnnou. Něco jako

```
double z = refcube(x + 3.0);  // nemůže se zkompilovat
```

nevypadá, že má smysl, protože výraz x + 3.0 není proměnná. Například takovému výrazu nemůžete přiřadit hodnotu:

```
x + 3.0 = 5.0;  // nesmyslné
```

Co se stane, když se pokusíte zavolat funkci jako refcube(x + 3.0)? V současném C++ je to chyba a některé kompilátory vám to tudíž řeknou. Jiné vydají varování podle následujících řádků:

```
Warning: Temporary used for parameter 'ra' in call to refcube(double &)
Varování: V zavolání refcube(double &) se použil dočasný parametr 'ra'
```

Důvod mírnější odpovědi spočívá v tom, že C++ vám ve svých začátcích povoloval předat do referenční proměnné výraz. Co se stane: Protože x + 3.0 není proměnná typu double, program vytvoří dočasnou bezejmennou proměnnou a inicializuje ji hodnotou výrazu x + 3.0. Potom se ra stává odkazem na dočasnou proměnnou. Podívejme se na dočasné proměnné blíže a uvidíme, kdy se vytvářejí a kdy ne.

Dočasné proměnné, referenční parametry a konstanty

C++ může generovat dočasnou proměnnou, pokud skutečný parametr neodpovídá referenčnímu parametru. V současné době to C++ dovoluje pouze tehdy, pokud je referenční parametr konstantní odkaz. Podívejme se na příklady, kdy C++ generuje dočasné proměnné a uvidíme, proč má omezení na konstantní odkaz smysl.

Za prvé, kdy se vytváří dočasná proměnná? Za předpokladu, že je referenční parametr konstantní, kompilátor generuje dočasné proměnné ve dvou typech situací:

- skutečný parametr je správného typu, ale není Lvalue (l-hodnota)
- skutečný parametr je jedním ze špatných typů, ale typu, který se může na správný konvertovat.

Parametr, to jest *Lvalue*, je datový objekt, na který se dá odkazovat. Například proměnná, prvek pole, odkaz a dereferenční ukazatel jsou l-hodnoty. Mezi l-hodnoty nepatří literálové konstanty a výrazy s násobnými členy. Například předpokládejme, že předefinujeme funkci `refcube()` tak, že má následující konstantní referenční parametr:

```
double refcube(const double &ra)
{
    return ra * ra * ra;
}
```

Nyní uvažujme následující kód:

```
double side = 3.0;
double * pd = &side;
double & rd = side;
long edge = 5L;
double lens[4] = { 2.0, 5.0, 10.0, 12.0};
double c1 = refcube(side);             // ra je side
double c2 = refcube(lens[2]);          // ra je lens[2]
double c3 = refcube(rd);               // ra je rd je side
double c4 = refcube(*pd)               // ra je *pd je side
double c4 = refcube(edge);             // ra je dočasná proměnná
double c5 = refcube(7.0);              // ra je dočasná proměnná
double c6 = refcube(side + 10.0);      // ra je dočasná proměnná
```

Parametry `side`, `lens[2]`, `rd`, a `*pd` jsou pojmenované objekty typu `double`, proto je možné pro ně generovat odkaz a nepotřebuje se žádná dočasná proměnná. (Připomínáme, že prvek pole se chová jako proměnná stejného typu.) Avšak `edge`, ačkoli proměnná, je chybného typu. Reference na double se nemůže odkazovat na long. Parametry `7.0` a `side + 10.0` mají na druhé straně sice správný typ, ale nejsou to pojmenované objekty. V každém z těchto případů kompilátor generuje dočasnou anonymní proměnnou a zajistí, že se na ni `ra` odkazuje. Tyto dočasné proměnné mají platnost po dobu volání funkce a potom je kompilátor uvolní z paměti a vyhodí.

Tak proč je toto chování v pořádku pro konstantní odkazy a ne jinak? Připomínáme funkci `swapr()` z výpisu programu 8.4:

```
void swapr(int & a, int & b)    // použití odkazů
{
    int temp;

    temp = a;        // použití a, b pro hodnoty proměnných
    a = b;
    b = temp;
}
```

Co by se přihodilo, kdybychom měli mezi volnějšími pravidly dřívějšího C++ toto?

```
long a = 3, b = 5;
swapr(a, b);
```

Tady existuje nesoulad typů, takže kompilátor by měl vytvořit dvě dočasné proměnné typu `int`, inicializovat je na `3` a `5` a potom zaměnit obsahy dočasných proměnných a ponechat `a` a `b` nezměněné.

Zkrátka, je-li úmyslem funkce s referenčními parametry modifikovat předávané proměnné, situace, které vytvářejí dočasné proměnné, tento záměr maří. Řešením je zabránit v těchto situacích vytváření dočasných proměnných a to je právě to, co teď C++ dělá.

Nyní přemýšlejme o funkci `refcube()`. Jejím úmyslem je pouze použít předané hodnoty, nikoli je modifikovat, takže dočasné proměnné nezapřičiňují žádné poškození a vytvářejí mnohem obecnější funkci vzhledem k druhu parametrů, se kterými může zacházet. Tedy, jestliže deklarace stanoví, že je reference konstanta, C++ generuje dočasné proměnné, je-li to nezbytné. V podstatě funkce C++ s konstantním referenčním formálním parametrem a s neodpovídajícím skutečným parametrem napodobuje tradiční chování předávání hodnotou, zajišťuje, že se původní data nezmění a použije dočasnou proměnnou na úschovu hodnoty.

Pamatujte:

Jestliže parametr volání funkce není Lvalue a nebo neodpovídá korespondujícímu konstantnímu referenčnímu parametru, C++ vytváří anonymní promněnnou správného typu, přiřazuje jí hodnotu parametru funkčního volání a přinutí ho, aby se na ni odkazoval.

Použijte const, když můžete

Pro deklarování referenčních parametrů jako odkazů na konstantní data existují tři silné důvody:

- Použití `const` vás chrání před programovými chybami, které neúmyslně mění data.
- Použití `const` umožňuje funkci zpracovávat jak `konstantní`, tak `nekonstantní` skutečné parametry, zatímco funkce, která `const` v prototypu vynechá, může akceptovat pouze `konstantní` data.
- Použití `konstantního` odkazu dovoluje funkci vhodným způsobem generovat a používat dočasné proměnné.

Měli byste deklarovat formální referenční parametry jako `const`, kdykoli je to vhodné.

Použití referencí se strukturou

Reference pracují báječně se strukturami a třídami, což jsou uživatelsky definované typy v C++. Vskutku, odkazy byly primárně zavedeny kvůli použití s těmito typy, nikoli pro použití se základními vestavěnými typy.

Metoda používání odkazů se strukturami je stejná jako metoda používání odkazů na základní proměnné; pouze když deklarujete strukturní parametr, použijte referenční operátor `&`. Program ve výpisu programu 8.6 to přesně dělá. Ukazuje také zajímavou fintu, která přinutí funkci navrátit odkaz. To umožňuje použití vyvolání funkce jako parametru jiné funkce. Tedy, to platí pro funkce s návratovou hodnotou. Ale také to umožňuje přiřadit hodnotu volání funce, a to je možné pouze s návratovým typem, kterým je odkaz. Tyto body vysvětlíme po ukázání programového výstupu. Program má funkci `use()`, která zobrazuje dva členy struktury a inkrementuje třetí člen. Tedy třetí člen může sledovat, kolikrát se pomocí funkce `use()` zacházelo s určitou strukturou.

Výpis programu 8.6 strtref.cpp

```
// strtref.cpp - použití odkazů na struktury
#include <iostream>
using namespace std;
struct sysop
{
    char name[26];
    char quote[64];
    int used;
};

sysop & use(sysop & sysopref);  // funkce s návratovým typem odkaz
int main()
{
    // Poznámka: některé implementace vyžadují pro umožnění inicializace
    // klíčové slovo static v obou deklaracích struktur
    sysop looper =
    {
        "Rick \"Fortran\" Looper",
        "Jsem takovy chlapek.",
        0
    };

    use(looper);                 // looper je typ struktury sysop
    cout << looper.used << " pouziti\n";

    use (use(looper));           // use(looper) is type sysop
    cout << looper.used << " pouziti\n";

    sysop morf =
    {
        "Polly Morf",
        "Polly neni pocitacovy pirat.",
        0
    };
    use(looper) = morf;          // může se přiřadit funkci!
    cout << looper.name << " rika:\n" << looper.quote << '\n';
    return 0;
}
```

```
// funkce use() navrací odkaz, který jí byl předán
sysop & use(sysop & sysopref)
{
    cout << sysopref.name << " rika:\n";
    cout << sysopref.quote << "\n";
    sysopref.used++;
    return sysopref;
}
```

Zde je výstup:

```
Rick "Fortran" Looper rika:
Jsem takovy chlapek.
1 pouziti
Rick "Fortran" Looper rika:
Jsem takovy chlapek.
Rick "Fortran" Looper rika:
Jsem takovy chlapek.
3 pouziti
Rick "Fortran" Looper rika:
Jsem takovy chlapek.
Polly Morf rika:
Polly neni pocitacovy pirat.
```

Poznámky k programu

Program nás zavádí do tří nových oblastí. První je použití odkazu na strukturu, který ukazuje první volání funkce:

```
use(looper);
```

Předává funkci `use()` strukturu `looper` odkazem, přičemž dělá ze `sysopref` synonymum pro `looper`. Když funkce `use()` zobrazuje členy `name` a `quote` struktury `sysopref`, skutečně zobrazuje členy `looper`. Také, když inkrementuje `sysopref.used` na 1, skutečně inkrementuje `looper.used`, jak program ukazuje:

```
Rick "Fortran" Looper rika:
Jsem takovy chlapek.
1 pouziti
```

Druhá oblast je použití odkazu jako návratové hodnoty. Obvykle návratový mechanismus kopíruje návratovou hodnotu do oblasti dočasné paměti, do které potom sahá volající program. Navrácení odkazu však znamená, že volající program zpracovává návratovou hodnotu přímo, a ne její kopii. Typicky se odkaz odkazuje především na odkaz předaný funkci, takže volající funkce se skutečně přímo zaměstná jednou ze svých proměnných. Zde je například `sysopref` odkazem na `looper`, takže návratová hodnota je původní proměnnou `looper` v `main()`.

Protože `use()` navrací odkaz typu `sysop`, může se použít jako parametr pro jinou funkci, která očekává buď parametr `sysop` nebo odkaz-na-`sysop`, jako například samotná `use()`. Tedy následující volání funkce ve výpisu programu 8.6 jsou ve skutečnosti dvě volání s jednou návratovou hodnotou funkce, která slouží jako parametr pro druhou:

```
use(use(looper));
```

Vnitřní volání funkce tiskne členy `name` a `quote` a inkrementuje člen `used` na 2. Funkce navrací `sysopref` a redukuje se na toto:

```
use(sysopref);
```

Protože je `sysopref` odkazem na `looper`, je funkční volání ekvivalentní tomuto:

```
use(looper);
```

Tedy `use()` zobrazuje znovu dva řetězcové členy a inkrementuje člen `used` na `3`.

Pamatujte:

Funkce, jež navrací odkaz, je ve skutečnosti druhým jménem proměnné, na kterou se odkazuje.

Třetí oblast, kterou program zkoumá, je, že můžete přiřadit hodnotu funkci, jestliže má návratovou hodnotu typu odkaz.

```
use(looper) = morf;
```

Pro funkce, které nevracejí odkaz je toto přiřazení syntaktickou chybou, ale je v pořádku pro `use()`. Toto je postup operací. Za prvé se vyhodnotí `use()`. To znamená, že se jí `looper` předá odkazem. Jako obvykle, funkce zobrazí dva členy a inkrementuje člen `used` na 4. Potom vrátí odkaz. Protože se návratová hodnota odkazuje na `looper`, je poslední krok ekvivalentní následujícímu:

```
looper = morf;
```

C++ vám dovoluje přiřadit jednu strukturu druhé, takže tento příkaz kopíruje obsah struktury `morf` do `looper`, což je vidět při zobrazení `looper.name`, které produkuje jméno `morf` a nikoli `looper`. Zkrátka příkaz

```
use(looper) = morf;  // návratová hodnota odkaz na looper
```

je ekvivalentní následujícímu:

```
use(looper);
looper = morf;
```

Pamatujte:

Funkci v C++ můžete přiřadit hodnotu (včetně struktury nebo objektu třídy), jestliže navrací referenci na proměnnou, nebo obecněji, na datový objekt. V tomto případě se hodnota přiřadí proměnné, na kterou se odkazuje nebo datovému objektu.

Je to další vlastnost, která umožňuje určitým formám operátorů předefinování. Můžete ji například použít na předefinování indexu pole pro třídu pomocí operátoru `operator[]`, který definuje efektnější verzi pole.

Úvahy spojené s navrácením reference nebo ukazatele

Když funkce vrací referenci nebo ukazatel na datový objekt, pak by měl objekt po jejím ukončení zůstat. Nejjednodušším způsobem, jak toho dosáhnout, je umožnit návrat z funkce pomocí odkazu nebo ukazatele, který jí byl předán jako parametr. Tj. způsob, kdy se již reference nebo ukazatel odkazují na něco ve volajícím programu. Funkce use() ve výpisu programu 8.6 tento postup používá.

Druhá metoda spočívá v použití new na vytvoření nové paměti. Příklady, ve kterých new vytváří prostor pro řetězce a funkce navrací ukazatel na tento prostor, jste již viděli dříve. Zde se ukazuje, jak byste mohli dělat něco podobného s odkazem:

```
sysop & clone(sysop & sysopref)
{
    sysop * psysop = new sysop;
    *psysop = sysopref;     // kopírování informace
    return *psysop;         // návrat odkazu na kopii
}
```

První příkaz vytváří bezejmennou strukturu sysop. Ukazatel psysop ukazuje na strukturu, proto je *psysop struktura. Programový kód vypadá, že vrací strukturu, ale deklarace funkce indikuje, že funkce ve skutečnosti vrací odkaz na strukturu. Potom byste mohli použít funkci tímto způsobem:

```
sysop & jolly = clone(looper);
```

To vytváří z jolly odkaz na novou strukturu. Při takovém přístupu vzniká problém, který spočívá v tom, že byste na uvonění paměti alokované pomocí new, když ji dále nepotřebujete, měli použít delete. Volání funkce clone() skrývá volání new, což způsobuje snazší zapomenutí volání delete později. Šablona auto_ptr, o které se pojednává v kapitole 15, „Třída String a standardní knihovna šablon (Standard Template Library)", může pomoci automatizovat proces vymazání.

Následujícímu programovému kódu byste se měli vyhnout:

```
sysop & clone2(sysop & sysopref)
{
    sysop newguy;           // první krok k velké chybě
    newguy = sysopref;      // kopie informace
    return newguy;          // návrat odkazu na kopii
}
```

Při vrácení reference na dočasnou proměnnou (newguy), která přestane existovat, jakmile funkce skončí, to má neblahý následek. (Tato kapitola později pojednává o přetrvávání různých druhů proměnných v sekci o paměťových třídách.) Podobně byste se měli vyhnout navrácení ukazatelů na takové dočasné proměnné.

Kdy používat parametry typu odkaz

Pro použití referencí existují dva hlavní důvody:

- Umožnit vám změnu datového objektu ve volající funkci.
- Zrychlení programu předáním odkazu místo celého objektu.

Druhý důvod je nejdůležitější pro velké datové objekty, jako například struktury a objekty třídy. Tyto dva důvody jsou stejné, jaké by se daly použít pro ukazatelové parametry. To dává smysl, protože referenční parametry jsou ve skutečnosti pouze jiným rozhraním pro kód založený na ukazateli. Tak, kdy byste použili referenci? Kdy ukazatel? Kdy předání hodnotou? Tady je několik metodických pokynů:

Funkce používá předaný údaj bez jeho modifikace:

- Když je datový objekt malý, jako například vestavěný datový typ nebo malá struktura, předejte ho hodnotou.
- Když je datový objekt pole, použijte ukazatel, a protože je to vaše jediná volba, vytvořte ukazatel na `const`.
- Když je datový objekt rozsáhlá struktura, použijte pro zvýšení výkonnosti programu ukazatel const nebo referenci const. Ušetříte čas a prostor potřebný na kopírování struktury nebo navrženou třídu. Udělejte ukazatel nebo referenci konstantní.
- Když je datový objekt třída, použijte konstantní referenci. Sémantika návrhu třídy často vyžaduje použití reference, což je hlavní důvod, proč C++ zahrnul tento rys. Tedy standardní způsob předání objektového parametru třídy je odkazem.

Funkce modifikuje údaj ve volající funkci:

- Když je datový objekt vestavěný datový typ, použijte ukazatel. Když uvidíte programový kód jako `fixit(&x)`, kde x je typu int, je hned jasné, že má funkce v úmyslu modifikovat x.
- Když je datový objekt pole, použijte vaši jedinou volbu, ukazatel.
- Když je datový objekt struktura, použijte referenci nebo ukazatel.
- Když je datový objekt třída, použijte referenci.

Samozřejmě, jsou to pouze metodické pokyny a mohou existovat důvody pro provedení jiných voleb. Například `cin` používá reference pro základní typy, takže můžete použít `cin >> n` namísto `cin >> &n`.

Standardní parametry

Podívejme se na další téma z pytle nových triků C++ – standardní parametry. Standardní parametr je hodnota, která se automaticky používá, když vynecháte odpovídající skutečný parametr z funkčního volání. Například, když stanovíte `void wow(int n)` tak, že `n` má standardní hodnotu `1`, potom je funkční volání `wow()` stejné jako `wow(1)`. Umožňuje vám to větší pružnost v tom, jak využíváte funkci. Předpokládejme, že máte funkci s parametry řetězec a `n`, která se jmenuje `left()` a navrací prvních `n` znaků řetězce. Přesněji, funkce navrací ukazatel na nový řetězec, který sestává z vybrané části původního řetězce. Například volání `left("teorie", 3)` vytvoří nový řetězec „`the`" a navrátí ukazatel na něj. Nyní předpokládejte, že stanovíte pro druhý parametr standardní hodnotu 1. Volání `left("teorie", 3)` by mělo pracovat jako předtím s vaší volbou `3` přepisující standard. Ale volání `left("teorie")`, místo vzniku chyby, by mělo předpokládat, že druhý parame-

tr je 1 a mělo by vrátit ukazatel na řetězec „t“. Tento druh standardu je užitečný, jestliže váš program často potřebuje extrahovat jednoznakový řetězec, ale příležitostně potřebuje delší řetězce.

Jak můžete stanovit standardní hodnotu? Musíte použít funkční prototyp. Protože se kompilátor dívá na prototyp, aby zjistil, kolik parametrů funkce používá, funkční prototyp také musí program upozornit na možnost standardních parametrů. Například zde je prototyp, který vyhovuje popisu `left()`:

```
char * left(const char * str, int n = 1);
```

Chceme, aby funkce navrátila nový řetězec, takže jeho typ je `char *` nebo ukazatel-na-`char`. Chceme ponechat původní řetezec nezměněný, proto pro první parametr používáme kvalifikátor `const`. Chceme, aby `n` měl hodnotu 1, tak mu ji přiřadíme. Standardní hodnota parametru je inicializační hodnotou. Tudíž výše uvedený prototyp inicializuje `n` na hodnotu 1. Jestliže vynecháte samotný `n`, má hodnotu 1, ale když předáte parametr, nová hodnota přepíše 1.

Jestliže použijete funkci se seznamem parametrů, musíte standardní parametry přidávat zleva doprava. To jest pro určitý parametr nemůžete poskytnout standardní hodnotu, dokud také neposkytnete standardní hodnoty pro všechny parametry zprava:

```
int harpo(int n, int m = 4, int j = 5);    // v pořádku
int chico(int n, int m = 6, int j);        // není v pořádku
int groucho(int k = 1, int m = 2, int n = 3);    // v pořádku
```

Prototyp harpo() například dovoluje volání s jedním, dvěma nebo třemi parametry:

```
beeps = harpo(2);         // stejné jako harpo(2,4,5)
beeps = harpo(1,8);       // stejné jako harpo(1,8,5)
beeps = harpo (8,7,6);    // nebyly použity žádné implicitní parametry
```

Skutečné parametry jsou přiřazovány formálním parametrům zleva doprava; nemůžete je přeskočit. Tudíž následující se nepovoluje:

```
beeps = harpo(3, ,8);    // chybné, nenastavuje m na 4
```

Standardní parametry nejsou hlavním programovacím průlomem; spíše jsou výhodou. Až se dostanete k návrhu tříd, zjistíte, že mohou redukovat množství metod konstruktorů, které musíte definovat.

Výpis programu 8.7 standardní parametry využívá. Všimněte si, že pouze prototyp indikuje standard. Definice funkce je stejná, jako kdyby byla bez standardních parametrů.

Výpis programu 8.7 left.cpp

```
// left.cpp — funkce na zpracování řetezce s implicitním parametrem
#include <iostream>
using namespace std;
const int ArSize = 80;
char * left(const char * str, int n = 1);
int main()
{
    char sample[ArSize];
```

```
    cout << "Zadejte retezec:\n";
    cin.get(sample,ArSize);
    char *ps = left(sample, 4);
    cout << ps << "\n";
    delete [] ps;        // uvolnění starého řetězce
    ps = left(sample);
    cout << ps << "\n";
    delete [] ps;        // uvolnění nového řetězce
    return 0;
}
// Tato funkce vrací ukazatel na nový řetězec, který
// obsahuje prvních n znaků řetězce str.
char * left(const char * str, int n)
{
    if(n < 0)
        n = 0;
    char * p = new char[n+1];
    int i;
    for (i = 0; i < n && str[i]; i++)
        p[i] = str[i];  // kopírování znaků
    while (i <= n)
        p[i++] = '\0';  // zbytek se nastaví na '\0'
    return p;
}
```

Zde je ukázka běhu programu:

```
Zadejte retezec:
nastavajici
nast
n
```

Poznámky k programu

Program na vytvoření nového řetězce, který má uchovávat nové znaky, používá `new`. Jedna nepříjemná možnost je, že nespolupracující uživatel požádá o záporný počet znaků. V tomto případě funkce nastaví čítač znaků na nulu a případně vrátí prázdný řetězec. Jiná nepříjemná možnost je, že nezodpovědný uživatel vyžádá více znaků, než řetězec obsahuje. Funkce se tomu brání kombinovaným testem:

```
i < n && str[i];
```

Test `i < n` zastaví cyklus poté, co se zkopírovalo `n` znaků. Druhá část testu znamená, že výraz `str[i]` je programový kód znaku, který se má kopírovat. Když cyklus dosáhne prázdného znaku, programový kód je nula a cyklus končí. Poslední cyklus `while` zakončuje řetězec prázdným znakem a potom nastavuje zbytek alokovaného prostoru, je-li nějaký, na prázdné znaky.

Jiným přístupem k nastavení velikosti nového řetězce je nastavení `n` na menší z předané hodnoty nebo délky řetězce:

```
int len = strlen(str);
n = (n < len) ? n : len;    // menší z n a len
char * p = new char[n+1];
```

Toto zajišťuje, že `new` nealokuje na úschovu řetězce více prostoru než je potřeba. To může být užitečné, když provádíte volání jako `left("Ahoj!", 32767)`. První přístup zkopíruje „`Ahoj!`" do pole o velikosti `32767` znaků, přičemž nastaví všechny znaky až na první tři na prázdný znak. Druhý přístup zkopíruje „`Ahoj!`" do pole o čtyřech znacích. Ale přidání dalšího volání funkce (`strlen()`) zvyšuje velikost programu, zpomaluje proces a vyžaduje od vás, abyste si vzpoměli a zahrnuli hlavičkový soubor `cstring` (nebo `string.h`). Programátoři v C měli sklon zvolit rychlejší běh, kompaktnější programový kód a ponechat na programátorovi větší břemeno při správném používání funkcí. Avšak tradice v C++ dává větší váhu na spolehlivost. Přece jenom pomalejší program, který pracuje správně je lepší než rychlý, který pracuje nesprávně. Jestliže se čas spotřebovaný na volání jeví jako problém, můžete nechat `left()` určit menší hodnotu z `n` a délky řetězce přímo. Například následující cyklus končí, když `m` dosáhne `n` nebo konce řetězce, podle toho, co nastane dříve:

```
int m = 0;
while ( m <= n && str[m] != '\0')
      m++;
char * p = new char[m+1]:
// pro zbytek kódu použijeme m misto n
```

Polymorfismus funkcí (přetěžování funkcí)

Polymorfismus funkcí je elegantní přídavek C++ ke schopnostem C. Zatímco standardní parametry vám umožní volat stejnou funkci za použití měnícího se počtu parametrů, *polymorfismus funkcí*, neboli také *přetěžování funkcí*, vám umožní používat rozmanité funkce, které sdílejí jedno jméno. Slovo „polymorfismus" znamená mít mnoho tvarů, takže polymorfismus funkcí dovoluje funkci mít mnoho podob. Podobně výraz „přetěžování funkcí" znamená, že můžete připojit více než jednu funkci ke stejnému jménu, tudíž přetížit jméno. Oba výrazy nakonec znamenají totéž, ale my obvykle budeme používat výraz přetížení funkcí – to zní jako tvrdě pracovat. Přetížení funkcí můžete použít na návrh skupiny funkcí, které v podstatě provádějí totéž, ale používají různý seznam parametrů.

Přetížené funkce jsou podobné slovům, která mají více než jeden význam. Například (Miss Piggy can root at the ball park for the home team, or she can root in the soil of truffels – větu nepřekládám, jedná se o použití slova root v kontextu) kohoutek vyběhl na dvůr a zakokrhal nebo vodovodní kohoutek se povolil. Kontext (existuje naděje) vám říká, jaký význam slova `kohoutek` se právě zamýšlí. C++ používá podobně kontextu k rozhodnutí, jaká se zamýšlí verze přetížené funkce.

Klíčem k přetížení funkcí je seznam parametrů funkce, který se také nazývá *signatura funkce*. Pokud mají dvě funkce stejný počet parametrů stejného typu ve stejném pořadí, mají stejnou signaturu; na jménu proměnné nezáleží. C++ vám umožňuje definovat dvě funkce stejného jména za předpokladu, že funkce mají různé signatury. Signatura se může lišit v počtu parametrů nebo v jejich typu nebo v obojím. Například můžete definovat sadu funkcí `print()` s následujícími prototypy:

```
void print(const char * str, int width);  // #1
void print(double d, int width);           // #2
void print(long l, int width);             // #3
void print(int i, int width);              // #4
void print(const char *str);               // #5
```

Když potom použijete funkci `printf()`, kompilátor porovná vaše použití s prototypem, který má stejnou signaturu:

```
print("Livance", 15);          // použijte #1
print("Limonada");             // použijte #5
print(1999.0, 10);             // použijte #2
print(1999, 12);               // použijte #4
print(1999L, 15);              // použijte #3
```

Například `print("Livance", 15)` používá jako parametry řetězec a celé číslo, a to odpovídá prototypu #1.

Když používáte přetížené funkce, ujistěte se, že používáte ve funkčním volání správné parametry. Například uvažujte následující příkazy:

```
unsigned int year = 3210;
print(year, 6);             // nejednoznačné volání
```

Jakému prototypu zde volání `print()` odpovídá? Neodpovídá žádnému z nich! Neexistence odpovídajícího prototypu automaticky nevylučuje jednu z funkcí, protože se C++ pokusí použít standardní typy konverzí na vynucení shody. Řekněme, kdyby existoval pouze prototyp #2 funkce `print()`, volání `print(year, 6)` by konvertovalo hodnotu `year` na typ `double`. Ale ve výše uvedeném programovém kódu existují tři prototypy, které berou číslo jako první parametr a poskytují na konverzi proměnné `year` tři odlišné volby. Čelem k této nejednoznačné situaci, C++ zamítne volání funkce jako chybu.

Některé signatury, které se zdají být jedna od druhé odlišné, nemohou spolu existovat. Například uvažujte tyto dva prototypy:

```
double cube(double x);
double cube(double & x);
```

Možná, že si myslíte, že zde je to místo, kde byste mohli použít přetížení funkcí, protože se signatury funkcí zdají být odlišné. Ale podívejte se na věci z hlediska kompilátoru. Předpokládejte, že máte tento programový kód:

```
cout << cube(x);
```

Parametr x vyhovuje jak prototypu `double x`, tak prototypu `double &x`. Tudíž kompilátor nemá žádný způsob jak se dozvědět, kterou funkci použít. Proto, když kontroluje signaturu funkce, aby se vyhnul takovému zmatku, považuje odkaz na typ a typ samotný za stejnou signaturu.

Postup nalezení odpovídající funkce rozlišuje mezi proměnnými typu `const` a ne-`const`. Uvažujme následující prototypy:

```
void dribble(char * bits);                 // přetížené
void dribble (const char *cbits);         // přetížené
void dabble(char * bits);                  // nepřetížené
void drivel(const char * bits);           // nepřetížené
```

Dále vidíme, čemu by mohla odpovídat funkční volání:

```
const char p1[20] = "Jake je pocasi?";
char p2[20] = "Jak jdou obchody?";
dribble(p1);      // dribble(const char *);
dribble(p2);      // dribble(char *);
dabble(p1);       // žádná shoda
dabble(p2);       // dabble(char *);
drivel(p1);       // drivel(char *);
drivel(p2);       // drivel(char *);
```

Funkce `dribble()` má dva prototypy, jeden pro konstantní ukazatele a druhý pro běžné a kompilátor vybere jeden nebo druhý v závislosti na tom, zda skutečný parametr je nebo není `konstantou`. Funkce `dabble()` odpovídá pouze volání s nekonstantním parametrem, ale `drivel()` jak `konstantním`, tak `nekonstantním` parametrům. Důvodem rozdílu v chování mezi `drivel()` a `dabble()` je, že existuje platné přiřazení ne-`const` hodnoty proměnné `const`, ale ne naopak.

Uvědomme si, že je to signatura funkce a nikoli její typ, co umožňuje přetížení funkcí. Například následující dvě deklarace jsou nekompatibilní:

```
long gronk(int n, float m);        // stejná signatura,
double gronk(int n, float m);      // proto není povoleno
```

Proto vám C++ nedovolí přetížit tímto způsobem `gronk()`. Můžete mít různé návratové typy, ale pouze když jsou signatury také různé:

```
long gronk(int n, float m);          // odlišná signatura,
double gronk(float n, float m);      // proto je dovoleno
```

O vyhovujících funkcích budeme dále diskutovat, jakmile pojednáme později v této kapitole o šablonách.

Příklad přetížení

Již jsme vyvinuli funkci `left()`, která navrací ukazatel na prvních n znaků řetězce. Přidejme další funkci `left()`, která navrací prvních `n` číslic v celém čísle. Můžete ji například použít k prověření prvních tří číslic poštovního směrovacího čísla U.S., které se ukládá jako celé číslo, což je užitečný počin, chcete-li třídit podle území města.

Naprogramovat celočíselnou funkci je trochu obtížnější, než naprogramovat její řetězcovou verzi, protože nemáme výhodu, která spočívá v tom, že se každá číslice uloží do svého vlastního prvku pole. Jeden postup je, nejprve spočítat počet číslic v čísle. Dělení čísla deseti odřízne jednu číslici, takže dělení můžete použít na počítání. Přesněji, můžete to provádět pomocí cyklu, jako je tento:

```
unsigned digits = 1;
while (n /= 10)
    digits++;
```

Cyklus počítá, kolikrát můžete odstranit číslici z n, dokud žádná nezbývá. Připomínáme, že n /= 10 je zkratka pro n = n / 10. Je-li například n rovno 8, testovací podmínka přiřadí n hodnotu 8 / 10, neboli 0, protože je to celočíselné dělení. To ukončí cyklus a digits zůstává 1. Ale jestliže je n rovno 238, první test cyklu nastaví n na 238 / 10, neboli na 23. To není nula, takže cyklus zvyšuje digits na 2. Další cyklus nastaví n na 23 /10, neboli 2. Opět to není nula, takže digits vzroste na 3. Následující cyklus nastaví n na 2 / 10, neboli 0 a cyklus skončí a ponechá digits nastavenou na správnou hodnotu 3.

Nyní předpokládejme, že víte, že číslo má pět číslic a chcete navrátit první tři číslice. Tuto hodnotu dostanete dělením čísla deseti a odpovědi opět dělíte deseti. Každé dělení deseti odřízne další číslici z pravého konce. Prospočítání počtu číslic, které se mají odříznout odečtěte pouze počet právě ukázaných číslic od jejich celkového počtu. Například na ukázání čtyř číslic z čísla, které jich má devět, odřízněte pouze posledních pět číslic. Tento přístup můžete zakódovat následovně:

```
ct = digits - ct;
while (ct--)
    num /= 10;
return num;
```

Výpis programu 8.8 zahrnuje tyto programové kódy do nové funkce left(). Funkce obsahuje další doplňkový programový kód na zacházení se zvláštními případy, jako například na dotázání se na nulové číslice nebo na více číslic, než číslo vlastní. Protože se signatura nové left() liší od staré, můžeme použít a používáme obě funkce ve stejném programu.

Výpis programu 8.8 leftover.cpp

```
// leftover.cpp — přetížení funkce left()
#include <iostream>
using namespace std;
unsigned long left(unsigned long num, unsigned ct);
char * left(const char * str, int n = 1);

int main()
{
    char * trip = "Hawaii!!";    // testovací hodnota
    unsigned long n = 12345678; // testovací hodnota
    int i;
    char * temp;

    for (i = 1; i < 10; i++)
    {
        cout << left(n, i) << "\n";
        temp = left(trip,i);
        cout << temp << "\n";
        delete [] temp; // ukazuje na dočasnou paměť
    }
    return 0;
}
```

```
// tato funkce vrací ct prvních číslic z čísla num.
unsigned long left(unsigned long num, unsigned ct)
{
    unsigned digits = 1;
    unsigned long n = num;

    if (ct == 0 || num == 0)
        return 0;           // vrací 0, když už nejsou číslice
    while (n /= 10)
        digits++;
    if (digits > ct)
    {
        ct = digits - ct;
        while (ct--)
        num /= 10;
        return num;             // vrátí ct zleva číslic
    }
    else                        // jestliže ct >= počet číslic
        return num;         // vrací celé číslo
}

// Tato funkce vrací ukazatel na řetězec, který obsahuje
// prvních n naků v řetězci str.
char * left(const char * str, int n)
{
    if(n < 0)
        n = 0;
    char * p = new char[n+1];
    int i;
    for (i = 0; i < n && str[i]; i++)
        p[i] = str[i];  // kopírování znaků
    while (i <= n)
        p[i++] = '\0';  // nastaví zbytek řetězce na '\0'
    return p;
}
```

Zde je výstup z programu:

```
1
H
12
Ha
123
Haw
1234
Hawa
12345
Hawai
123456
Hawaii
1234567
```

```
Hawaii!
12345678
Hawaii!!
12345678
Hawaii!!
```

Kdy používat přetěžování funkcí

Možná se vám zdá, že přetěžování funkcí je fascinující, avšak nepřežeňte tento prostředek. Přetěžování funkcí byste měli rezervovat pro funkce, které v zásadě provádějí stejnou úlohu, ale s různými typy dat. Také byste mohli chtít ověřit, zda můžete dosáhnout stejného cíle se standardními parametry. Například byste mohli nahradit jedinou na řetězce orientovanou funkci `left` dvěma funkcemi, které se překrývají:

```
char * left(const char * str, unsigned n);   // dva parametry
char * left(const char * str);               // jeden parametr
```

Avšak použití jediné funkce se standardním parametrem je jednodušší. Namísto dvou se napíše pouze jediná a program vyžaduje paměť pouze pro jednu funkci místo pro dvě. Když se rozhodnete funkci modifikovat, existuje pouze jediná funkce, kterou musíte upravit. Avšak, vyžadujete-li různé typy parametrů, standardní parametry nejsou prospěšné, měli byste tudíž použít přetížení funkcí.

Šablony funkcí

Současné kompilátory C++ implementují jeden z novějších dodatků, *šablony funkcí*. Šablony funkcí jsou popisem rodových funkcí, to jest definují funkce na bázi všeobecných typů, na jejichž místo může být dosazen typ specifický, jako například `int`, `double` a podobně. Předáním typu jako parametu šabloně způsobíte, že kompilátor vygeneruje funkci tohoto určitého typu. Protože šablony vám umožňují programovat ve smyslu všeobecně použitelných namísto specifických typů, proces se občas nazývá *generické programování*. Protože jsou typy reprezentovány parametry, vlastnost šablony se občas vztahuje k *parametrizovaným typům*. Podívejme se, proč je taková vlastnost užitečná a jak pracuje.

Dříve jsme definovali funkci, která zaměňovala dvě hodnoty typu `int`. Předpokládejme, že místo toho chcete zaměnit dvě hodnoty typu `double`. Jeden způsob je, zduplikovat původní programový kód a nahradit každý typ `int` typem `double`. Když musíte zaměnit dvě hodnoty typu `char`, můžete opět použít stejný postup. Stále, když musíte provést tyto titěrné změny, je to plýtvání vaším cenným časem a vždy existuje pravděpodobnost vzniku chyby. Jakmile ručně provedete změny, měli byste prozkoumat typ `int`. Až uděláte globální prohledání a nahrazení, měli byste udělat něco jako překonvertování

```
int integer;
```

na toto:

```
double doubleeger;
```

Schopnost, že šablony funkce v C++ automatizují proces, vám šetří čas a poskytuje větší spolehlivost.

Šablony funkcí vám umožňují definovat funkce ve smyslu jistého libovolného typu. Například můžete zavést šablonu na záměnu takto:

```
template <class Any>
void Swap(Any &a, Any &b)
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

První řádek určuje, že jste vytvořili šablonu a že jste pojmenovali libovolný typ jménem `Any`. Klíčové slovo `template` a `class` (alternativně `typename`) jsou povinné stejně jako ostré závorky. Je na vás, jaké jméno typu zvolíte, pokud dodržujete obvyklá pravidla pro pojmenování v C++; mnoho programátorů používá jednoduchá jména jako například `T`. Zbytek programového kódu popisuje jednoduchý algoritmus na záměnu dvou hodnot typu `Any`. Šablona nevytváří žádné funkce. Namísto toho poskytuje kompilátoru směrnice o tom, jak funkci definovat. Chcete-li, aby funkce zaměnila dvě hodnoty typu `int`, potom kompilátor vytvoří funkci, která se řídí vzorem šablony a nahradí `Any` typem `int`. Podobně, jestliže potřebujete funkci na záměnu typů `double`, kompilátor se řídí šablonou a nahrazuje `Any` typem `double`.

Nedávno C++ dodal nové klíčové slovo `typename`, které můžete v této konkrétní souvislosti použít namísto klíčového slova `class`. Definici šablony tedy můžete napsat tímto způsobem:

```
template <typename Any>
void Swap(Any &a, Any &b)
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

Klíčové slovo `typename` trochu více ozřejmuje to, že parametr `Any` představuje typ; bohužel s použitím klíčového slova `class` již byly vyvinuty velké knihovny programového kódu. Dnešní kompilátory zacházejí stejným způsobem s těmito dvěma klíčovými slovy, jsou-li použita v této souvislosti.

Tip:

Používejte šablony, když potřebujete funkce, které aplikují stejný algoritmus na různé typy. Nezajímáte-li se o kompatibilitu směrem dozadu a smíříte se s psaním delších slov, používejte k deklarování typu parametrů raději klíčové slovo `typename` než class.

Abyste dali kompilátoru vědět, že potřebujete konkrétní formu funkce pro záměnu pouze ve vašem programu, použijte funkci `Swap()`. Kompilátor prověří typy parametrů, které používáte a potom vygeneruje odpovídající funkci. Výpis programu 8.9 ukazuje, jak to pracuje. Návrh programu dodržuje pro běžné funkce obvyklý vzor, prototyp šablony funkce na začátku programu a definice šablony funkce za `main()`.

Kompatibilita:

Zastaralé verze kompilátoru C++ možná šablony nepodporují. Novější verze akceptují klíčové slovo `typename` jako alternativu ke `class`. Verze 2.71 g++ vyžaduje, aby se definice šablony vyskytovala před jejím použitím.

Výpis programu 8.9 funtemp.cpp

```
// funtemp.cpp – použití šablony funkce
#include <iostream>
using namespace std;
// prototyp šablony funkce
template <class Any>  // nebo typename Any
void Swap(Any &a, Any &b);

int main()
{
    int i = 10;
    int j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Pouziti funkce zamenujici cisla typu int, ktera je generovana
kompilatorem:\n";
    Swap(i,j);  // generuje void Swap(int &, int &)
    cout << "Nyni i, j = " << i << ", " << j << ".\n";

    double x = 24.5;
    double y = 81.7;
    cout << "x, y = " << x << ", " << y << ".\n";
    cout << "Pouziti funkce zamenujici cisla typu double, ktera je genero-
vana kompilatorem:\n";
    Swap(x,y);  // generuje void Swap(double &, double &)
    cout << "Nyni x, y = " << x << ", " << y << ".\n";
    return 0;
}

// definice šablony funkce
template <class Any>  // nebo typename Any
void Swap(Any &a, Any &b)
{
    Any temp;   // temp, proměnná typu Any
    temp = a;
    a = b;
    b = temp;
}
```

První funkce Swap() má dva parametry typu int, takže kompilátor generuje její verzi typu int. To jest, nahrazuje každé použití Any pomocí int a produkuje definici, která vypadá takto:

```
void Swap(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Vy tento programový kód nevidíte, ale kompilátor ano. Druhá funkce má dva parametry typu double, takže kompilátor generuje její verzi typu double. To jest, nahrazuje Any pomocí double a generuje tento programový kód:

```
void Swap(double &a, double &b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

Zde je výstup programu; vidíte, že postup fungoval:

```
i, j = 10, 20.
Pouziti funkce zamenujici cisla typu int, ktera je generovana kompilato-
rem:
Nyni i, j = 20, 10.
x, y = 24.5, 81.7.
Pouziti funkce zamenujici cisla typu double, ktera je generovana kompila-
torem:
Nyni x, y = 81.7, 24.5.
```

Všimněte si, že šablony funkcí nedělají vaše proveditelné programy o nic kratší. Ve výpisu programu 8.9 se stále zaměstnáváte dvěma oddělenými definicemi funkcí, stejně jako byste každou definovali manuálně. A výsledný programový kód neobsahuje žádné šablony; obsahuje pouze skutečné funkce generované pro váš program. Výhody šablon jsou v tom, že vytvářejí generické definice násobných funkcí jednodušeji a spolehlivěji.

Přetížené šablony

Šablony používáte, když potřebujete funkce, které aplikují stejný algoritmus pro rozmanité typy, jako ve výpisu programu 8.8. Avšak mohlo by se stát, že všechny typy by nepoužívaly stejný výpočetní postup. Abyste vyhověli této možnosti, můžete přetypovat definice šablon, stejně tak jako jste přetypovali definice běžných funkcí. Jako u běžného přetypování, přetypované šablony potřebují odlišné signatury funkcí. Například výpis programu 8.10 dodává novou šablonu pro záměnu, která zaměňuje prvky pole. Původní šablona měla signaturu (Any &, Any &) a nová má signaturu (Any [], Any [],

int). Všimněte si, že poslední parametr má určitý typ (int) spíše než typ generický. Ne všechny parametry šablony musí být druhy vzorových parametrů.

Jestliže kompilátor v twotemps.cpp narazí na první použití funkce Swap(), všimne si, že má dva parametry typu int, a to odpovídá původní šabloně. Avšak druhé použití má dvě pole a hodnotu typu int, a to odpovídá nové šabloně.

Výpis programu 8.10 twotemps.cpp

```
// twotemps.cpp - použití přetížených šablon funkcí
#include <iostream>
using namespace std;
template <class Any>     // původní šablona
void Swap(Any &a, Any &b);

template <class Any>     // nová šablona
void Swap(Any *a, Any *b, int n);

void Show(int a[]);
const int Lim = 8;
int main()
{
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Pouziti funkce zamenujici cisla typu int, ktera je generovana
kompilatorem:\n";
    Swap(i,j);              // odpovídá původní šabloně
    cout << "Nyni i, j = " << i << ", " << j << ".\n";

    int d1[Lim] = {0,7,0,4,1,7,7,6};
    int d2[Lim] = {0,6,2,0,1,9,6,9};
    cout << "Puvodni pole:\n";
    Show(d1);
    Show(d2);
    Swap(d1,d2,Lim);        // odpovídá nové šabloně
    cout << "Zamenena pole:\n";
    Show(d1);
    Show(d2);

    return 0;
}

template <class Any>
void Swap(Any &a, Any &b)
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
template <class Any>
void Swap(Any a[], Any b[], int n)
{
    Any temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}

void Show(int a[])
{
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];
    cout << "\n";
}
```

Komapatibilita:

Zastaralé verze kompilátorů C++ možná nepodporují šablony. Nové verze možná akceptují klíčové slovo `typename` místo `class`. Starší verze C++ jsou náročnější na odpovídající typ a vyžadují následující programový kód pro vytvoření `const int Lim`, která vyhovuje požadavkům šablony na běžný typ `int`:

```
Swap(d1,d2, int (Lim));          // přetypování Lim na ne-const int
```

Verze 2.71 g++ vyžaduje, aby se definice šablon umisťovaly před `main()`.

Zde je výstup programu:

```
Pouziti funkce zamenujici cisla typu int, ktera je generovana kompilato-
rem:
Nyni i, j = 20, 10.
Puvodni pole:
07/04/1776
06/20/1969
Zamenena pole:
06/20/1969
07/04/1776
```

Explicitní specializace

Předpokládejme, že definujete takovouto strukturu:

```
struct job
{
```

```
        char name[40];
        double salary;
        int floor;
    };
```

Také předpokládáme, že chcete být schopni zaměnit obsah takových dvou struktur. Původní šablona používá k provedení záměny následující kód:

```
temp = a;
a = b;
b = temp;
```

Protože C++ vám dovolujeje přiřadit jednu strukturu druhé, funguje to bezvadně, dokonce i když je typ `Any` strukturou `job`. Předpokládejme však, že chcete zaměnit pouze členy `salary` a `floor`. To vyžaduje jiný programový kód, ale parametry pro `Swap()` by měly být stejné jako pro první případ (odkazy na dvě struktury `job`), takže na poskytnutí alternativního programového kódu nemůžete použít běžné přetížení šablon.

Můžete však dodat specializovanou definici, která se nazývá explicitní specializace a má požadovaný kód. Když kompilátor zjistí specializovanou definici, která přesně odpovídá funkčnímu volání, použije ji, aniž by se podíval na šablony.

Mechanismus specializace se měnil s rozvojem jazyka. Podíváme se na původní tvar, který byl podporován staršími kompilátory, dále na přechodný a potom na současný tvar.

Přístup podle první generace

Původně běžná deklarace funkce, která přesně odpovídala funkčnímu volání, překódovávala definici šablony. Například uvažujme následující fragment programu:

```
template <class Any>
void Swap(Any &a, Any &b);        // prototyp šablony
void Swap(int & n, int & m);      // běžný prototyp
int main()
{
    double u, v;
    ...
    Swap(u,v);  // použiti šablony
    int a, b;
    ...
    Swap(a,b);  // použiti void Swap(int &, int &)
```

Když kompilátor dosáhne na volání funkce `Swap(a, b)`, má výběr mezi generováním definice funkce pomocí šablony nebo funkce `Swap(int &, int &)`, která není šablonou. Původní vlastnost šablony žádá kompilátor, aby použil verzi, která není šablonou a nakládal s ní jako se specializací šablony.

Druhá generace

C++ (neoficální verze předběžného konceptu) žádá na chvíli o další úpravu. Ve výše uvedeném příkladu programového kódu kompilátor používal šablonu a ignoroval normální funkční prototyp a definici. Provedení této změny však neodstranilo potřebu explicitní specializace. C++ tedy na deklarování a definování explicitní specializace zavádí novou

syntaxi. Myšlenka spočívá v následování jména funkce lomenými závorkami, které obsahují typ specializace. Například prototyp specializace funkce `Swap()` by mohl vypadat takto:

```
void Swap<int>(int & a, int & b);    // specializace
```

Prototyp s odpovídajícími parametry se musí vyskytovat před prvním voláním funkce:

```
template <class Any>
void Swap(Any &a, Any &b);            // prototyp šablony
void Swap<int>(int & n, int & m);    // prototyp specializace
int main()
{
    double u, v;
    ...
    Swap(u,v);  // použití šablony
    int a, b;
    ...
    Swap(a,b);  // použije void Swap<int>(int &, int &)
}
void Swap<int>(int & n, int & m)    // definice specializace
{...}
```

Všimněme si, že výraz `<int>` se vyskytuje jak v prototypu, tak v definici funkce. Může se, ale nemusí, vyskytovat ve volání funkce.

```
Swap<int>(a, b);  // také možná volba
```

Třetí generace

Standard C++ ustanovil ještě jeden přístup:

- Normální funkce opět potlačuje šablonu, ale není považována za specializaci.
- Prototyp a definice explicitní specializace by měly mít prefix `template <>`.

Specializace potlačuje běžnou šablonu a funkce, která není šablonou, potlačuje obě.

Tedy verze, které nejsou šablonou, se vybírají před explicitními specializacemi a verzemi šablon a explicitní specializace před verzí, která se generuje ze šablony.

```
template <class Any>
void Swap(Any &a, Any &b);                          // prototyp šablony
template <> void Swap<int>(int & n, int & m);      // prototyp specializace
int main()
{
    double u, v;
    ...
    Swap(u,v);  // použije šablonu
    int a, b;
    ...
    Swap(a,b);  // použije void Swap<int>(int &, int &)
}
template <> void Swap<int>(int & n, int & m)     // definice specializace
{...}
```

Výraz <int> ve Swap<int> je volitelný, protože typy parametrů funkce indikují, že to je specializace pro int. Proto také může být prototyp napsán tímto způsobem:

```
template <> void Swap(int & n, int & m);   // jednodušší tvar
```

Hlavička šablony funkce může být také zjednodušena vynecháním části <int>.

Za chvilku se podíváme na důvody vystupňování spletitosti syntaxe, ale nejprve se podívejme na příklad:

Příklad

Výpis programu 8.11 ukazuje, jak explicitní specializace pracuje. Je to příprava k následování standardu C++, ale vy můžete vymazat nebo vsunout označení poznámek kvůli výběru jedné z dalších verzí.

Výpis programu 8.11 twoswap.cpp

```
// twoswap.cpp - specializace potlačuje šablonu
#include <iostream>
using namespace std;
template <class Any>
void Swap(Any &a, Any &b);

struct job
{
    char name[40];
    double salary;
    int floor;
};

// void Swap(job &j1, job &j2);        // prvni generace
// void Swap<job>(job &j1, job &j2); // druhá generace
template <> void Swap(job &j1, job &j2); // třetí generace
void Show(job &j);

int main()
{
    cout.precision(2);
    cout.setf(ios::fixed, ios::floatfield);
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Pouziti funkce zamenujici cisla typu int, ktera je generovana
kompilatorem:\n";
    Swap(i,j);    // generuje void Swap(int &, int &)
    cout << "Nyni i, j = " << i << ", " << j << ".\n";

    job sue = {"Susan Yaffee", 63000.60, 7};
    job sidney = {"Sidney Taffee", 66060.72, 9};
    cout << "Pred zamenou prace:\n";
    Show(sue);
    Show(sidney);
```

```
    Swap(sue, sidney); // používá void Swap(job &, job &)
    cout << "Po zamene prace:\n";
    Show(sue);
    Show(sidney);

    return 0;
}

template <class Any>
void Swap(Any &a, Any &b)    // obecná verze
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}

// zaměňuje pouze pole salary a floor struktury job

// void Swap(job &j1, job &j2)        // první generace
// void Swap<job>(job &j1, job &j2)  // druhá generace
template <> void Swap(job &j1, job &j2)  // třetí generace
{
    double t1;
    int t2;
    t1 = j1.salary;
    j1.salary = j2.salary;
    j2.salary = t1;
    t2 = j1.floor;
    j1.floor = j2.floor;
    j2.floor = t2;
}

void Show(job &j)
{
    cout << j.name << ": $" << j.salary
        << " na poschodi " << j.floor << "\n";
}
```

Kompatibilita:

Některé kompilátory rozpoznávají tvar `template <> void Swap()`, jiné `void Swap<job>` a jiné `void Swap()`.

Zde je výstup programu:

```
i, j = 10, 20.
Pouziti funkce zamenujici cisla typu int, ktera je generovana kompilato-
rem:
Nyni i, j = 20, 10.
Pred zamenou prace:
```

```
Susan Yaffee: $63000.60 na poschodi 7
Sidney Taffee: $66060.72 na poschodi 9
Po zamene prace:
Susan Yaffee: $66060.72 na poschodi 9
Sidney Taffee: $63000.60 na poschodi 7
```

Konkretizace a specializace

Mějme na paměti, že zařazení šablony funkce do kódu programu samo o sobě negeneruje funkční definici. Je to pouze plán na generování definice funkce. Jakmile kompilátor použije šablonu na generování definice funkce pro určitý typ, výsledek se nazývá *konkretizace* šablony. Například volání funkce `Swap(i, j)` způsobí, že kompilátor generuje konkretizaci `Swap()` a používá `int` jako typ. Šablona není definicí funkce, ale definicí funkce je určitá konkretizace, která `int` používá. Tento typ konkretizace se nazývá *implicitní konkretizace*, protože kompilátor odvozuje nutnost pro vytvoření definice tím, že si všimne, že program používá funkci `Swap()` s parametry typu int.

Původně byla implicitní konkretizace jediným způsobem, podle kterého kompilátor ze šablon generoval definice funkcí, ale nyní C++ povoluje *explicitní konkretizaci*. To znamená, že můžete dát kompilátoru pokyn na vytvoření určité konkretizace přímo, například `Swap<int>()`. Syntaxe má zaručit určitou variantu, kterou chcete, použitím označení <> na indikaci typu a předřazením klíčového slova `template`:

```
template Swap<int>(int, int);  // explicitni konkretizace
```

Kompilátor, který tento rys implementuje na základě shlédnutí této deklarace, použije na generování konkretizace šablonu `Swap()` používající typ `int`.

Explicitní konkretizace je v rozporu s explicitní specializací, která používá jednu nebo druhou z těchto ekvivalentních deklarací:

```
template <> Swap<int>(int, int);  // explicitni specializace
template <> Swap(int, int);       // explicitni specializace
```

Rozdíl je, že tyto deklarace znamenají „Nepoužívejte na generování definice funkce šablonu `Swap()`. Použijte místo toho pro typ `int` samostatnou specializovanou definici funkce explicitně." Tyto prototypy musí být spárovány se svými vlastními definicemi funkcí.

Upozornění:

Je chyba použít jak explicitní konkretizaci, tak explicitní specializaci na stejný typ(y) ve stejné programové jednotce.

Implicitní konkretizace, explicitní konkretizace a explicitní specializace se společně nazývají *specializace*. Co mají všechny společné je, že spíše představují definici funkce založenou na určitých typech, než definici, která je všeobecným popisem.

Přidání explicitní konkretizace vedlo k nové syntaxi použití prefixů `template` a `template <>` v deklaracích na rozlišení mezi explicitní konkretizací a explicitní specializací. Často náklady na více práce odpovídají více syntaktickým pravidlům.

Která funkce?

Co C++ s přetížením funkcí, šablonami funkcí a přetížením šablon funkcí potřebuje a má, je dobře definovaná strategie pro rozhodování, kterou definici funkce použít pro její volání, zvláště když jsou násobné parametry. Postup se nazývá *přetížené rozhodování*. Shrnutí detailů úplné strategie by zabralo menší kapitolu, tak si shrňme, jak postup funguje:

- Fáze 1: Vytvoříme seznam vhodných funkcí. Jsou to funkce a šablony funkcí se stejným jménem jako volaná funkce.
- Fáze 2: Ze seznamu kandidátů vytvoříme seznam životaschopných funkcí. Jsou to funkce se správným počtem parametrů, pro které existuje implicitní konverzní postup, jenž zahrnuje případ přesného souhladu každého typu skutečného parametru a odpovídajícího formálního parametru. Například funkční volání s parametrem typu float by mohlo konvertovat svou hodnotu na typ double, aby odpovídal typu double formálního parametru a šablona by mohla generovat konkretizaci pro float.
- Fáze 3: Určete, zda existuje nejživotaschopnější funkce. Jestliže ano, tuto funkci použijte, jinak je volání funkce chyba.

Uvažujte případ funkce pouze s jedním parametrem, například následující volání:

```
may('B');    // skutečný parametr je typu char
```

Nejprve kompilátor zachytí podezřelá místa, což jsou funkce a šablony funkcí, které mají jméno `may()`. Potom zjistí ty, které se mohou volat s jedním parametrem. Například toto bude pokusná sbírka:

```
void may(int);                              // #1
float may(float, float = 3);                // #2
void may(char);                             // #3
char * may(const char *);                   // #4
template<class T> void may(const T &);      // #5
template<class T> void may(T *);            // #6
```

Všimněte si, že se uvažují pouze signatury, nikoli návratové typy. Dva z těchto kandidátů (#4 a #6) však nevyhovují, protože se celočíselný typ nemůže konvertovat implicitně (to jest bez explicitního přetypování) na typ ukazatel. To ponechává čtyři životaschopné funkce, každá z nichž by mohla být použita, kdyby byla jedinou deklarovanou funkcí.

Dále musí kompilátor určit, která ze životaschopných funkcí je nejlepší. Bere v úvahu požadovanou konverzi na vytvoření volání funkce, jejíž parametr odpovídá vyhovujícímu parametru kandidáta. Obecně je řazení od nejlepšího po nejhoršího takové:

1. Přesná shoda.
2. Konverze s podporou (například automatické konverze `char` a `short` na `int`, nebo `float` na `double`).
3. Konverze standardním převodem (například `int` na `char`, nebo `long` na `double`).
4. Uživatelsky definované konverze, jako například v deklaracích tříd.

Například funkce #1 je lepší než funkce #2, protože `char` na `int` je povýšení (kapitola 3, „Práce s daty"), naopak `char` na `float` je standardní konverze (kapitola 3). Funkce #3 a #5

jsou obě lepší než #1 nebo #2, protože jsou přesnou shodou. Nyní vyvstává řada otázek. Co je přesná shoda a co se stane, když získáte dvě?

Přesné shody a nejlepší shody

C++, když směřuje k přesné shodě, podporuje několik „triviálních konverzí". Tabulka 8.1 je vyjmenovává, `Typ` má význam jistého libovolného typu. Například skutečný parametr `int` je přesnou shodou pro formální parametr `int &`. Všimněte si, že `Typ` může být něco jako `char &`, takže tato pravidla zahrnují pravidla konvertování `char &` na `const char &`. Vstup `Type (seznam parametrů)` znamená, že jméno funkce, které je skutečný parametr odpovídá ukazateli na funkci, který je formální parametr, pokud mají oba stejný návratový typ a seznam parametrů. (Vzpomeňte si na ukazatele na funkce z kapitoly 7, „Funkce – programové moduly v C++" a jak jim můžete předat jméno funkce jako parametr funkci, která očekává ukazatel na funkci.) O klíčovém slovu `volatile` pojednáme v této kapitole později.

Tabulka 8.1 Triviální konverze, které jsou pro přesnou shodu povoleny

Ze skutečného parametru	Na formální parametr
Typ	
Typ &	
Typ &	Typ
Typ []	* Typ
Typ (*seznam parametrů*)	Typ (*)(*seznam parametrů*)
Typ	const *Typ*
Typ	volatile *Typ*
Typ *	const
Typ *	
Typ *	volatile *Typ* *

Zkrátka a dobře předpokládejte, že máte následující funkční kód:

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(ink);
```

Potom by všechny následující prototypy měly být přesnou shodou:

```
void recycle(blot);              // #1  blot-na-blot
void recycle(const blot);        // #2  blot-na-(const blot)
void recycle(blot &);            // #3  blot-na-(blot &)
void recycle(const blot &);      // #4  blot-na-(const blot &)
```

Jak jste mohli očekávat, při hledání odpovídajících prototypů zjistíte, že kompilátor nemůže úplně dokončit rozpoznávací proces přetížení. Neexistuje nejlepší životaschopná funkce a kompilátor generuje chybovou zprávu, pravděpodobně za použití slov jako „nejednoznačný (ambiguous)".

Avšak někdy se může rozpoznání přetížení uskutečnit, dokonce i když přesně odpovídají dvě funkce. Za prvé, ukazatele a odkazy na nekonstantní data jsou přednostně přiřaze-

ny parametrům nekonstantního ukazatele a odkazu. To jest, pokud by v předchozí ukázce existovaly pouze funkce #3 a #4, byla by vybrána pouze funkce #3, protože ink nebyla deklarována jako konstanta. Bohužel toto omezení mezi const a ne-const se aplikuje pouze na data, na která se odkazuje pomocí ukazatelů a odkazů. To jest, kdyby byly dostupné pouze #1 a #2, obdrželi byste chybu nejednoznačnosti.

Další případ, kdy jedna přesná shoda je lepší než druhá nastane v okamžiku, když jedna funkce není šablonou funkce a druhá je. V tomto případě se považuje normální funkce za lepší než šablona včetně explicitních specializací.

Když se zaměstnáváte dvěma přesnými shodami, u kterých se stane, že jsou obě šablonami funkcí, šablona funkce, která je více specializovaná, je lepší funkcí. To například znamená, že se před funkcí generovanou implicitně ze vzoru šablony vybere explicitní specializace:

```
struct blot {int a; char b[10];};
template <class Type> void recycle (Type t); // šablona
template <> void recycle<blot> (blot & t);   // specializace pro blot
...
blot ink = {25, "spots"};
...
recycle(ink);  // použije specializaci
```

Výraz nejspecializovanější neimplikuje nezbytně explicitní specializaci; obecněji, indikuje, že se použije méně konverzí, když kompilátor odvodí typ, který se má použít. Například uvažujme dvě následující šablony:

```
template <class Type> void recycle (Type t);    #1
template <class Type> void recycle (Type * t);  #2
```

Předpokládejme, že program, který využívá tyto šablony obsahuje následující programový kód:

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(&ink);  // adresa struktury
```

Volání funkce recykle(&ink) vyhovuje šabloně #1 s Type, který se interpretuje jako blot *. Volání recykle(&ink) odpovídá také volání #2 s Type, který je tentokrát ink. Tyto kombinace do oblasti životaschopných funkcí vysílají dvě implicitní konkretizace recycle<blot>(blot *).

Z těchto dvou funkcí se recycle<blot *>(blot *) považuje za specializovanější, protože podstupuje méně konverzí, které se mají generovat. To jest šablona #2 již explicitně řekla funkci, že parametr byl ukazatel-na-Type, takže Type by mohl být přímo identifikován pomocí blot. Avšak šablona #1 měla Type jako parametr funkce, takže se Type musel interpretovat jako ukazatel-na-blot. To jest, v šabloně #2 se již specializoval jako ukazatel, odtud výraz „více specializovaný".

Pravidla nalezení specializovanější šablony se nazývají *pravidla částečného uspořádání* pro šablony funkcí. Podobně jako explicitní konkretizace jsou novými doplňky jazyka.

Zkrátka, rozpoznávací proces přetížení hledá funkci, která nejlépe odpovídá. Jestliže existuje pouze jedna, vybere se tato funkce. Existuje-li více než jedna, které jsou sice svázané, ale pouze jedna není šablonou funkce, vybere se tato. Existuje-li více než jeden kandidát, kteří jsou sice svázaní a všichni jsou šablonami funkcí, ale jedna je více specializovaná než zbytek, vybere se tato. Existují-li dvě nebo více stejně dobrých funkcí, které nejsou šablonami nebo existují-li dvě nebo více stejně dobrých funkcí, žádná z nichž není více specializovaná než zbytek, funkční volání je nejednoznačné a nastává chyba. Jestliže neexistuje žádné odpovídající volání, samozřejmě je to také chyba.

Funkce s více parametry

Tyto věci se skutečně zamotají, když se volání funkce s více parametry srovnává s prototypy s více parametry. Kompilátor se musí podívat na shody všech parametrů. Když může najít funkci, která je lepší než všechny ostatní životaschopné funkce, je to ta, která se vybere. Co se týká funkce, která je lepší než jiná, musí poskytnout alespoň tak dobrou shodu všech parametrů a lepší alespoň pro jeden.

Tato kniha nemá v úmyslu vyvolat proces shody pomocí komplexních příkladů. Pravidla existují, takže existuje přesně stanovený výsledek pro libovolnou možnou množinu prototypů funkcí a šablon.

Oddělená kompilace

C++, podobně jako C, vám dovoluje, dokonce vás podporuje, umísťovat jednotlivé funkce do programu v oddělených souborech. Jak popisuje kapitola 1, „Začínáme", soubory můžete zkompilovat odděleně a potom je sestavit do konečného proveditelného programu. (Kompilátor C++ typicky kompiluje programy a také spravuje sestavovací program.) Jestliže měníte pouze jeden soubor, můžete pouze tento jeden soubor překompilovat a potom ho sestavit s dříve zkompilovanou verzí dalších souborů. Tento prostředek umožňuje snadněji spravovat rozsáhlé programy. Kromě toho, většina prostředí C++ poskytuje na pomoc se správou doplňkové prostředky. Například systémy UNIX mají program `make`; sleduje, na kterých souborech program závisí a kdy se naposledy měnily. Když spustíte `make` a ten detekuje, že jste změnili od poslední kompilace jeden nebo více zdrojových souborů, pamatuje si správné pořadí potřebné na rekonstrukci programu. Prostředí Symantec C++, Turbo C++, Watcom C++, Microsoft Visual C++ a Metrowerks Code Warior poskytují podobné prostředky spolu s jejich menu Project.

Podívejme se na jednoduchý příklad. Místo zkoumání detailů kompilace, která závisí na implementaci, soustřeďme se na obecnější aspekty, jako například návrh.

Předpokládejme například, že se rozhodnete rozčlenit program ve výpisu programu 7.1 umístěním funkcí do oddělených souborů. Připomínáme, že tento programový výpis konvertuje pravoúhlé souřadnice na polární a potom zobrazuje výsledek. Nemůžete jednoduše odříznout původní soubor tečkovanou čárou na konci `main()`. Problém spočívá v tom, že `main()` a další dvě funkce všechny používají stejnou strukturu deklarací, takže musíte vložit deklarace do obou souborů. Jednoduché zapsání do každého souboru může vést k chybě. I když nakopírujete deklarace struktur správně, musíte si pamatovat, že musíte měnit obě sady deklarací, když později provedete změny. Zkrátka, rozšíření programu do několika souborů vytváří nové problémy.

Kdo chce více problémů? Vývojáři C a C++ nechtěli, takže poskytli prostředek #include, který s tímto druhem situace umí zacházet. Místo umístění struktur deklarací do každého souboru je můžete umístit do hlavičkového souboru a potom zahrnout tento hlavičkový soubor do každého souboru zdrojového kódu. Tímto způsobem, když modifikujete strukturu deklarace, tak ji můžete upravit pouze jednou v hlavičkovém souboru. Také do hlavičkového souboru můžete umístit prototypy funkcí. Původní program tedy můžete rozdělit do tří částí:

- Hlavičkový soubor, který obsahuje struktury deklarací a prototypů funkcí, jež tyto struktury používají.
- Soubor zdrojového kódu pro funkce, které se k těmto strukturám vztahují.
- Soubor zdrojového kódu, který obsahuje programový kód, jenž tyto funkce volá.

To je užitečná strategie organizace programu. Například, jestliže napíšete další program, který používá tyto stejné funkce, pouze začleníte tento hlavičkový soubor do projektu nebo seznamu `make`. Tato organizace také odráží přístup OOP. Jeden soubor, hlavičkový, obsahuje definici uživatelsky definovaných typů. Druhý soubor obsahuje funkční kód na manipulaci s uživatelsky definovanými typy. Společně tvoří balík, který můžete použít na varianty programů.

Nevkládejte definice funkcí nebo deklarace proměnných do hlavičkového souboru. Mohlo by to pracovat pokud jde o jednoduchý projekt, ale obvykle to vede k potížím. Například, kdybyste v hlavičkovém souboru měli definici funkce a potom ho zahrnuli do dvou dalších souborů, které by byly částí jednoduchého programu, zamotali byste se se dvěma definicemi stejné funkce v jediném programu, což je chyba. Tady jsou některé věci, které se společně nacházejí v hlavičkových souborech:

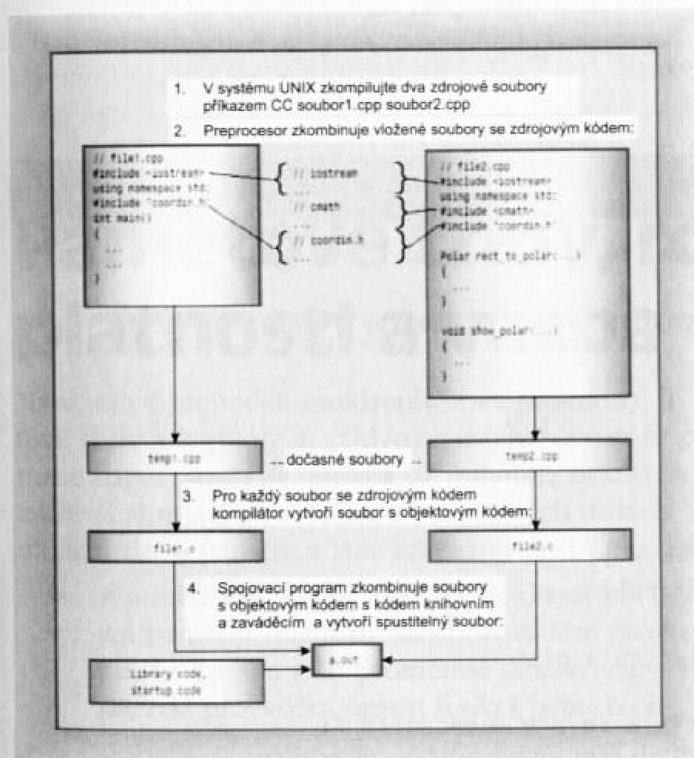
- Funkční prototypy.
- Symbolické konstanty definované pomocí direktivy #define nebo const.
- Deklarace struktur.
- Deklarace tříd.
- Deklarace šablon.
- Vložené funkce.

Je v pořádku vložit struktury deklarací do hlavičkového souboru, protože nevytvářejí proměnné; pouze říkají kompilátoru, jak má strukturu proměnné vytvořit, když ji deklarujete v souboru zdrojového kódu. Podobně, deklarace šablon nejsou programovým kódem, který se má kompilovat; jsou předpisem pro kompilátor jak má vygenerovat definice funkcí, které odpovídají funkčnímu volání zjištěnému ve zdrojovém kódu. Údaje deklarované pomocí funkce `const` a inline mají zvláštní vazební vlastnosti (brzy přijdou na pořad), které jim dovolují, aby se umístily do hlavičkových souborů, aniž by způsobily problémy.

Výpisy programů 8.12, 8.13 a 8.14 ukazují výsledek rozdělení výpisu programu 7.11 do jednotlivých částí. Všimněte si, že když vkládáme hlavičkový soubor, používáme „`coordin.h`" namísto `<coordin.h>`. Když se jméno souboru uzavírá do ostrých závorek, kompilátor se dívá do části systémového souboru hostitelského systému, který obsahuje standardní hlavičkové soubory. Ale když je jméno uzavřeno do dvojitých uvozovek, kompilátor se nej-

prve dívá do běžného pracovního adresáře (nebo někam jinam v závislosti na kompilátoru). Když tam hlavičkový soubor nenalezne, dívá se na standardní umístění. Takže když vkládáte vaše vlastní hlavičkové soubory, používejte znaky uvozovek, nikoli ostré závorky.

Obrázek 8.3 načrtává kroky pro sestavení tohoto programu dohromady na unixovém systému. Všimněte si, že kompilátoru CC pouze dodáte příkaz a další kroky následují automaticky. Symantec C++, Borland C++, Turbo C++, Metrowerks Code Warior, Watcom C++ a Microsoft Visual C++ procházejí v podstatě stejnými kroky, ale jak se ukázalo v kapitole 1, postup inicializujete různě použitím menu, které vám umožní vytvořit projekt a asociovat s ním soubory zdrojového kódu. Všimněte si, že pouze dodáváte do projektu soubory zdrojového kódu, nikoli hlavičkové soubory. To je proto, že direktiva #include hlavičkové soubory spravuje. Také nepoužívejte `#include` k zavedení souborů zdrojového kódu, protože to může vést k násobným deklaracím.



Obrázek 8.3 Kompilace několika souborových programů v C++ na unixovém systému

---

Upozornění:

V integrovaných vývojových prostředích nepřidávejte hlavičkové soubory do seznamu projektu a nepoužívejte `#include` k zavedení souborů zdrojového kódu v dalších souborech zdrojového kódu.

Výpis programu 8.12 coordin.h

```
// coordin.h – šablony struktur a prototypy funkcí
// structure templates
struct Polar
{
    double distance;    // vzdálenost od počátku
    double angle;       // směr od počátku
};
struct rect
{
    double x;           // horizontální vzdálenost od počátku
    double y;           // vertikální vzdálenost od počátku
};

// prototypy
Polar rect_to_polar(rect xypos);
void show_polar(Polar dapos);
```

Výpis programu 8.13 file1.cpp

```
// file1.cpp – příklad programu ze dvou souborů
#include <iostream>
#include "coordin.h" // šablony struktur, prototypy funkcí
using namespace std;
int main()
{
    rect rplace;
    Polar pplace;

    cout << "Zadejte hodnoty x a y: ";
    while (cin >> rplace.x >> rplace.y)  // úhledné použití cin
    {
        pplace = rect_to_polar(rplace);
        show_polar(pplace);
        cout << "Dalsi dve cisla (q pro ukonceni): ";
    }
    return 0;
}
```

Výpis programu 8.14 file2.cpp

```
// file2.cpp – obsahuje funkce volané v souboru file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h" // šablony struktur, prototypy funkcí
using namespace std;

// konvertuje pravoúhlé souřadnice na polární
```

```
Polar rect_to_polar(rect xypos)
{
    Polar answer;

    answer.distance =
        sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;      // vrací strukturu Polar
}

// ukazuje polární souřadnice, přičemž konvertuje úhel v radiánech na stupně
void show_polar (Polar dapos)
{
    const double Rad_to_deg = 57.29577951;

    cout << "vzdalenost = " << dapos.distance;
    cout << ", uhel = " << dapos.angle * Rad_to_deg;
    cout << " stupnu\n";
}
```

Paměťové třídy, rozsah platnosti a vazba

Nyní, když jste viděli multisouborové programy, je vhodná doba rozšířit diskusi o paměťové třídy z kapitoly 4, „Odvozené typy", protože paměťové třídy ovlivňují sdílené informace napříč soubory. Možná už uplynula nějaká doba, co jste naposledy četli kapitolu 4, takže si shrneme, co říkala o paměťových třídách. C++ používá tři odlišná schémata pro uložení dat v paměti, a tato schémata se liší tím, jak dlouho v ní uchovávají data:

- Automatické proměnné jsou deklarovány uvnitř definice funkce, která zahrnuje její pametry. Vznikají, když provádění programu vstoupí do funkce nebo do bloku, ve kterém jsou proměnné deklarovány. Paměť pro ně vyhrazená se uvolní, jakmile provádění opustí funkci nebo blok.
- Statické proměnné jsou definovány vně definice funkce nebo ještě pomocí klíčového slova `static`. Přetrvávají po celou dobu běhu programu.
- Dynamická paměť se alokuje pomocí operátoru `new` a přetrvává, dokud se neuvolní pomocí `delete` nebo dokud neskončí program, podle toho, co nastane dříve.

Nyní se dozvíme zbytek příběhu, včetně fascinujících podrobností o tom, kdy jsou proměnné různých typů v rozsahu platnosti (použitelné programem) a o vazbě, která určuje, jaká informace se sdílí napříč soubory.

Rozsah platnosti a vazba

Rozsah platnosti popisuje, jak široce je jméno v souboru viditelné (překladová jednotka). Například proměnná definovaná ve funkci se může používat v této funkci, ale ne v jiné, zatímco proměnná definovaná v souboru nad definicemi funkcí se může používat ve všech funkcích. *Vazba* popisuje, jak může být jméno sdíleno v jiných jednotkách. Jméno s *externí vazbou* může být sdíleno napříč soubory a jméno s *interní vazbou* může být sdíleno pouze funkcemi uvnitř jediného souboru. Jména automatických proměnných nemají žádnou vazbu, protože se nesdílejí.

Proměnná v C++ může mít jeden z několika rozsahů platnosti. Proměnná, která má *lokální rozsah platnosti* (také nazývaný *rozsah platnosti bloku*) je známa pouze uvnitř bloku, ve kterém je definovaná. Připomeňme si, blok je skupina příkazů uzavřená do složených závorek. Například tělo funkce je blok, ale můžete mít další vnořené bloky uvnitř jejího těla. Proměnná, která má *globální rozsah platnosti* (také nazývaný *rozsah platnosti souboru*) je známa v celém souboru od bodu, kde se definuje. Automatické proměnné mají lokální rozsah platnosti a statická proměnná může mít libovolný rozsah platnosti v závislosti na tom, jak se definuje. Jména použitá v *rozsahu platnosti prototypu funkce* jsou známa pouze uvnitř závorek, které uzavírají seznam parametrů. (Proto ve skutečnosti nevadí, co jsou nebo zda jsou dokonce přítomna.) Členy deklarované ve třídě mají *rozsah platnosti třídy* (viz kapitola 9, „Objekty a třídy"). Proměnné deklarované v prostoru jmen mají *rozsah platnosti prostoru jmen*. (Nyní, když byly do jazyka přidány prostory jmen, globální rozsah platnosti se stal zvláštním případem rozsahu platnosti prostoru jmen.)

Funkce v C++ mohou mít rozsah platnosti třídy nebo prostoru jmen včetně globálního rozsahu platnosti, ale nemohou mít lokální rozsah platnosti. (Protože funkce nemůže být definována uvnitř bloku, kdyby měla lokální rozsah platnosti, mohla by být známa pouze sama sobě, a tudíž by nemohla být volána jinou funkcí. Taková funkce by nemohla být funkcí.)

Podívejme se detailněji na rozsah platnosti a na vlastnosti vazby různých paměťových tříd v C++. Začneme zkoumáním situace před tím, než byly do mixu dodány prostory jmen a potom se podíváme, jak prostory jmen pohled na celou věc modifikují.

Automatické proměnné

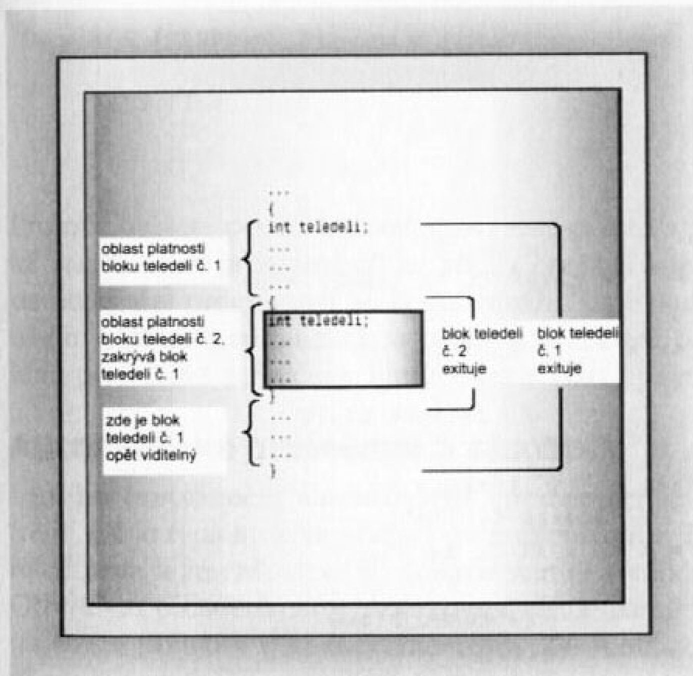
Funkční parametry a proměnné definované uvnitř funkce patří standardně do automatické třídy paměti. Tyto proměnné mají lokální viditelnost neboli rozsah platnosti. To jest, jesliže deklarujete proměnnou v `main()`, která se jmenuje `texas` a deklarujete jinou proměnnou se stejným jménem ve funkci, která se nazývá `oil()`, vytvořili jste dvě nezávislé proměnné, každá je známa pouze ve funkci, ve které je definována. Cokoli děláte s proměnnou `texas` ve funkci `oil()` nemá žádný vliv na `texas` v `main()` a naopak. Také se každá proměnná alokuje, když se začíná její funkce provádět a každá se ztratí ze života, když její funkce skončí.

Jestliže definujete proměnnou uvnitř bloku, její přetrvání a rozsah platnosti se omezuje na tento blok. Například předpokládejme, že definujete proměnnou na začátku `main()`, která se jmenuje `teledeli`. Nyní předpokládejme, že začnete uvnitř `main()` nový blok a definujete v tomto bloku novou proměnnou, která se nazývá `websight`. Potom `teledeli` je

viditelná jak ve vnějším tak ve vnitřním bloku, zatímco `websight` pouze ve vnitřním bloku a existuje pouze od bodu jeho definice do doby, kdy provádění programu mine konec tohoto bloku:

```
int main()
{
      int teledeli = 5;
      {
            int websight = -2;
            cout << websight << ' ' << teledeli << endl;
      } // websight zaniká
      cout << teledeli << endl;
      ...
}
```

Ale co když pojmenujete proměnnou ve vnitřním bloku `teledeli` namísto `websight`, takže máte dvě proměnné stejného jména, jednu ve vnějším a jednu ve vnitřním bloku? V tomto případě, když program provádí příkazy uvnitř bloku, interpretuje jméno `teledeli` tak, že znamená proměnnou lokálního bloku. Říkáme, že nová definice *skrývá* prvotní definici. Nová definice je v rozsahu platnosti a stará je dočasně mimo rozsah. Když program opouští blok, původní definice se vrací zpět do rozsahu platnosti. Viz obrázek 8.4.



Obrázek 8.4 Bloky a rozsah platnosti

Výpis programu 8.15 ukazuje, jak se automatické proměnné lokalizují vzhledem k funkci nebo bloku, který je obsahuje.

Výpis programu 8.15 auto.cpp

```
// auto.cpp - ilustrace rozsahu platnosti automatických proměnných
#include <iostream>
using namespace std;
void oil(int x);
int main()
{
    // Poznámka: některé implementace vyžadují přetypování
    // adres v programu na typ unsigned

    int texas = 31;
    int year = 1999;
    cout << "V main(), texas = " << texas << ", &texas =";
    cout << &texas << "\n";
    cout << "V main(), year = " << year << ", &year =";
    cout << &year << "\n";
    oil(texas);
    cout << "V main(), texas = " << texas << ", &texas =";
    cout << &texas << "\n";
    cout << "V main(), year = " << year << ", &year =";
    cout << &year << "\n";
    return 0;
}

void oil(int x)
{
    int texas = 5;

    cout << "V oil(), texas = " << texas << ", &texas =";
    cout << &texas << "\n";
    cout << "V oil(), x = " << x << ", &x =";
    cout << &x << "\n";
    {                                  // začátek bloku
        int texas = 113;
        cout << "V bloku, texas = " << texas;
        cout << ", &texas = " << &texas << "\n";
        cout << "V bloku, x = " << x << ", &x =";
        cout << &x << "\n";
    }                                  // konec bloku
    cout << "Za blokem texas = " << texas;
    cout << ", &texas = " << &texas << "\n";
}
```

Zde je výstup:

```
V main(), texas = 31, &texas =0065FE00
V main(), year = 1999, &year =0065FDFC
V oil(), texas = 5, &texas =0065FDEC
V oil(), x = 31, &x =0065FDF8
V bloku, texas = 113, &texas = 0065FDE8
```

```
V bloku, x = 31, &x =0065FDF8
Za blokem texas = 5, &texas = 0065FDEC
V main(), texas = 31, &texas =0065FE00
V main(), year = 1999, &year =0065FDFC
```

Všimněte si, že každá ze tří proměnných `texas` má svou vlastní odlišnou adresu a jak program používá v daném okamžiku v rozsahu platnosti pouze určitou proměnnou, takže přiřazení hodnoty 113 do `texas` ve vnitřním bloku v `oil()` nemá žádný vliv na další proměnné stejného jména.

Shrňme sled událostí. Když `main()` začíná, program alokuje prostor pro `texas` a `year` a tyto proměnné se dostávají do rozsahu platnosti. Když program vyvolá `oil()`, tyto proměnné zůstávají v paměti, ale z rozsahu platnosti vypadávají. Alokují se dvě nové proměnné `x` a `texas` a dostávají se do rozsahu platnosti. Když provádění programu dosáhne vnitřního bloku v `oil()`, nová `texas` vypadne z rozsahu platnosti, jelikož je nahrazena stejnou novější definicí. Avšak proměnná `x` zůstává v rozsahu platnosti, protože blok nedefinuje novou `x`. Když provádění ukončuje blok, paměť pro novější `texas` se uvolní a `texas` číslo 2 se dostává zpět do rozsahu platnosti. Když končí funkce `oil()`, `texas` a `x` zanikají a původní `texas` a `year` se vracejí zpět do rozsahu platnosti.

Mimochodem v C++ (a v C) k indikaci paměťové třídy explicitně můžete použít klíčové slovo `auto`:

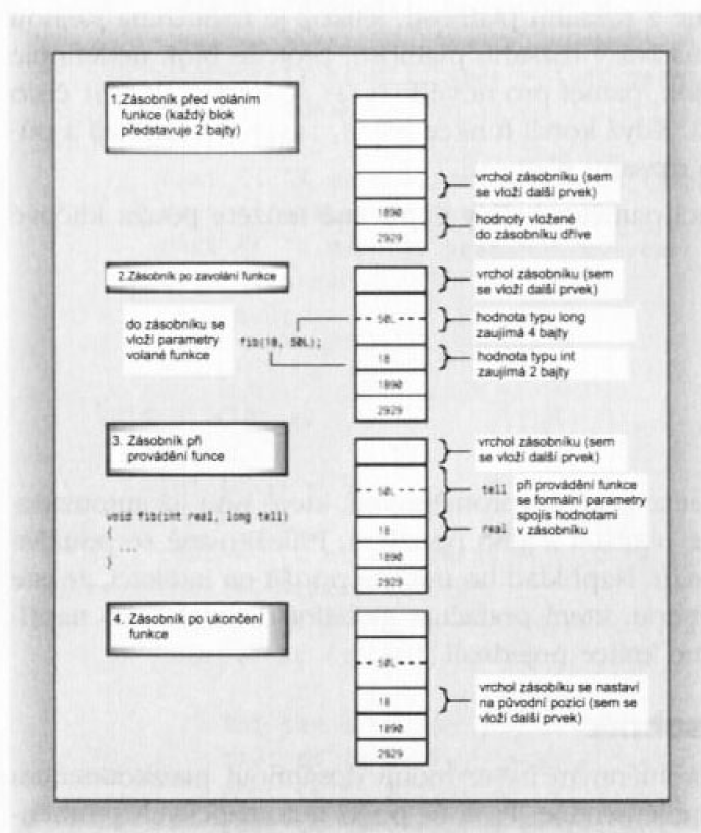
```
int froob(int n)
{
auto float ford;
...
}
```

Protože můžete použít klíčové slovo `auto` pouze s proměnnými, které jsou již automatické standardně, programátoři se zřídka obtěžují s jeho použitím. Příležitostně se používá na objasnění programového kódu čtenáři. Například ho můžete použít na indikaci, že jste účelově použili automatickou proměnnou, která potlačuje globální definici, jako například předchozí definice, o kterých jsme krátce pojednali.

Automatické proměnné a zásobník

Lepšího porozumění automatickým proměnným byste mohli dosáhnout prozkoumáním toho, jak je typický kompilátor C++ implementuje. Protože počet automatických proměnných roste a zmenšuje se, jak funkce startují a končí, musí je program za běhu spravovat. Obvyklým prostředkem je rezervování sekce paměti a zacházení s ní jako se zásobníkem na řízení přílivu a odlivu proměnných. Zásobník se nazývá proto, protože se nový údaj obrazně vrství nad starým (to jest na přilehlé lokaci, nikoli na stejné) a potom se ze zásobníku odsouvá, jakmile s ním program skončí. Standardní velikost zásobníku závisí na implementaci, ale kompilátor obvykle umožňuje její velikost měnit. Program sleduje zásobník pomocí dvou ukazatelů. Jeden ukazuje na patu zásobníku, kde se rezervovala paměť pro jeho začátek a jeden ukazuje na jeho vrchol, což je další volná paměťová lokace. Když se funkce vyvolá, její automatické proměnné se přidají do zásobníku a ukazatel na vrchol ukazuje na další dostupný volný prostor, který následuje za proměnnými. Když se funkce ukončí, ukazatel na vrchol se přenastaví na hodnotu, kterou měl předtím, než se funkce vyvolala a efektivně uvolní paměť, která by se mohla použít pro nové proměnné.

Zásobník je návrhem LIFO (last in-first out, poslední dovnitř, první ven), což znamená, že poslední přidaná proměnná do zásobníku jde první ven. Návrh zjednodušuje předávání parametru. Volání funkce umísťuje hodnoty svých parametrů na vrchol zásobníku a přenastaví ukazatel na vrchol. Volaná funkce k určení adresy každého parametru používá popis svých formálních parametrů. Například obrázek 8.5 ukazuje funkci `fib()`, která, když se volá, předává 2 bajty typu `int` a 4 bajty typu `long`. Tyto hodnoty jdou do zásobníku. Když se `fib()` začíná vykonávat, spojuje jména `real` a `tell` s těmito dvěma hodnotami. Když se `fib()` ukončuje, ukazatel na vrchol zásobníku se přemístí na svou dřívější pozici. Nové hodnoty se nevymažou, avšak nejsou dále označeny a prostor, který okupovaly bude použit dalším procesem, který do zásobníku umísťuje hodnoty. (Obrázek je trochu zjednodušený, protože volání funkcí může předávat dodatečnou informaci, jako například návratovou adresu.)



Obrázek 8.5 Předávání parametrů pomocí zásobníku

Možná jste si všimli, že když se přidávají nové proměnné, adresy se ve výpisu programu 8.15 snižují spíše než zvyšují. To je proto, že tento určitý kompilátor C++ implementuje zásobník shora dolů. To mění směr, kterým zásobník roste, ale uchovává si základní koncepci.

Proměnné typu `register`

C++, podobně C, podporují na deklarování automatických proměnných klíčové slovo `register`. Toto klíčové slovo je pokynem pro kompilátor, od kterého chcete, aby používal místo zásobníku k zacházení s určitými proměnnými registr CPU. Myšlenkou je, že CPU

může přistupovat k hodnotě jedním ze svých registrů mnohem rychleji, než může k paměti v zásobníku. Pro deklarování registrové proměnné předřaďte před typ klíčové slovo `register`:

```
register int count_fast; // požadavek na registrovou proměnnou
```

Pravděpodobně jste si všimli blíže určujících slov „pokyn" a „požadavek". Kompilátor nemusí požadavek akceptovat. Například registry již mohou být obsazeny nebo byste si mohli vyžádat typ, který se do registru nevejde. Běžné přesvědčení spočívá v tom, že kompilátory jsou často dostatečně chytré, nepotřebují pokyn. Když například píšete cyklus `for`, kompilátor by se mohl sám rozhodnout použít pro index cyklu registr.

Třída paměti static

Proměnné, které patří do paměťové třídy static, existují po celou dobu trvání programu; jsou méně pomíjející než automatické proměnné. Protože se počet statických proměnných nemění, když program běží, nepotřebujete na jejich spravování speciální zařízení jako zásobník. Místo toho alokujete pevný blok paměti na úschovu všech statických proměnných, ty přetrvávají tak dlouho, dokud se program vykonává. Také když nechcete explicitně inicializovat statickou proměnnou, kompilátor ji nastaví na nulu. Statická pole nebo struktury mají každý prvek nebo člen nastaven na nulu standardně.

Kompatibilita:

Klasický K&R C vám nedovoloval inicializovat automatická pole a struktury, ale povoloval vám inicializovat pole a struktury statické. ANSI C a C++ vám povolují inicializovat oba bloky, ale některé translátory C++ používají kompilátory C, které nejsou plně kompatibilní s ANSI C. Když používáte takovou implementaci, možná musíte pro inicializaci polí a struktur použít jednu ze tří variant statických tříd paměti.

C++, podobně C, poskytuje tři varianty statických proměnných: externí, statickou a externí statickou. Připadá-li vám to trochu matoucí, máte pravdu. Naneštěstí C++ používá slovo *static* ve dvou různých smyslech. Jeden, který se týká proměnné, jež přetrvává po dobu trvání programu. V tomto smyslu jsou všechny tři varianty statické. Druhý význam omezuje, do jaké míry je proměnná známa; ovlivňuje rozsah platnosti a vazbu. Externí proměnná je přístupná všem souborům v programu (globální rozsah platnosti, externí vazba); externí statická proměnná všem funkcím v jednom souboru (globální rozsah platnosti, interní vazba); a statická proměnná deklarovaná uvnitř bloku se omezuje na jediný blok (lokální rozsah, interní vazba). Tabulka 8.2 shrnuje vlastnosti paměťových tříd, jak se používaly v období před prostory jmen; již jste o tom byli trochu informováni, podrobněji se však na to podíváme nyní.

Tabulka 8.2 Paměťové třídy

Paměťová třída	Způsob vzniku	Rozsah platnosti	Vazba	Trvání
Automatický blok	Standardně pro parametry a proměnné deklarované uvnitř funkce	Lokální	Interní	Po dobu běhu funkce
Externí	Standardně pro proměnné deklarované vně funkce	Globální	Externí	Po dobu programu
Statická	Aplikací klíčového slova static na proměnnou deklarovanou uvnitř funkce	Lokální	Interní	Po dobu programu
Externí statická	Aplikací klíčového slova static na proměnnou deklarovanou vně funkce	Globální	Interní	Po dobu programu

Externí proměnné

Externí proměnné se nazývají externí proto, že jsou definovány vně jakékoli funkce a jsou k ní tudíž externí. Například by mohly být deklarovány nad funkcí `main()`. Externí proměnnou můžete použít v libovolné funkci, která následuje za definicí externích proměnných v souboru. Tedy externí proměnné se na rozdíl od automatických, které jsou lokální, také nazývají globální proměnné. Avšak, jestliže definujete automatickou proměnnou, která má stejné jméno jako externí proměnná, pak jediná proměnná, která spadá do rozsahu platnosti, když program provádí funkci, ve které je definována, je automatická proměnná. Výpis programu 8.16 tyto body vysvětluje. Také ukazuje, jak můžete použít k deklarování externí proměnné, která byla dříve deklarována, klíčové slovo `extern` a jak můžete v C++ pro přístup k jinak skryté proměnné použít operátor rozlišení rozsahu platnosti.

Výpis programu 8.16 external.cpp

```
// external.cpp - externí proměnné
#include <iostream>
using namespace std;
// externí proměnná
double warming = 0.3;

// prototypy funkce
void update(double dt);
void local();

int main()                        // používá globální proměnnou
{
    cout << "Globalni promenna warming je " << warming << " stupnu.\n";
    update(0.1);                  // vyvolání funkce na změnu proměnné warming
    cout << "Globalni promenna warming je " << warming << " stupnu.\n";
    local();                      // vyvolání funkce na lokální proměnnou war-
ming
```

```
    cout << "Globalni promenna warming je " << warming << " stupnu.\n";
    return 0;
}
void update(double dt)      // modifikuje globální proměnnou
{
    extern double warming;  // volitelná opětná deklarace
    warming += dt;
    cout << "Zmena globalni promenne warming na " << warming;
    cout << " stupnu.\n";
}

void local()                // používá lokální proměnnou
{
double warming = 0.8;   // nová proměnná skrývá externí proměnnou

    cout << "Lokalni promenna warming = " << warming << " stupnu.\n";
    // Přístup ke globální proměnné pomocí
    // operátoru na rozlišení platnosti
    cout << "Ale globalni promenna warming = " << ::warming;
    cout << " stupnu.\n";
}
```

Zde je výstup:

```
Globalni promenna warming je 0.3 stupnu.
Zmena globalni promenne warming na 0.4 stupnu.
Globalni promenna warming je 0.4 stupnu.
Lokalni promenna warming = 0.8 stupnu.
Ale globalni promenna warming = 0.4 stupnu.
Globalni promenna warming je 0.4 stupnu.
```

Poznámky k programu

Výstup programu ukazuje, že obě funkce `main()` a `update()` mohou přistupovat k externí proměnné `warming`. Všimněte si, že změna, kterou dělá `update()` s `warming` se dostaví při opětovném použití proměnné.

Funkce `update()` předeklaruje proměnnou `warming` pomocí klíčového slova `extern`. Toto klíčové slovo znamená „použijte proměnnou tohoto dříve externě definovaného jména". Protože je to totéž, co by `update()` mimochodem udělala, kdybyste vypustili celou deklaraci, je tato deklarace nadbytečná. Slouží ke zdokumentování toho, že je funkce navržena pro použití externí proměnné. Původní deklarace

```
double warming = 0.3;
```

se nazývá *definiční deklarací* nebo jednoduše *definicí*. Způsobí, že se má pro proměnnou alokovat paměť. Předeklarování

```
extern double warming;
```

se nazývá *referenční deklarací* nebo jednoduše *deklarací*. Nezpůsobí, že se má pro proměnnou alokovat paměť, protože se odkazuje na existující proměnnou. Klíčové slovo `extern` můžete použít pouze v deklaracích, které se odkazují na proměnné někde deklaro-

vané (nebo na funkce – o tom později). Také je nemůžete v referenční deklaraci inicializovat.

```
extern double warming = 0.5; // chybně
```

V deklaraci můžete proměnnou inicializovat pouze tehdy, jestliže ji deklarace alokuje, tedy, pouze v definiční deklaraci. Koneckonců, výraz inicializace znamená přiřazení hodnoty do paměťové lokace, když se přiděluje paměťové místo.

Funkce `local()` ukazuje, že když definujete lokální proměnnou, která má stejné jméno jako globální proměnná, lokální verze skrývá globální verzi. Například funkce `local()`, když zobrazuje svoji hodnotu, používá lokální definici `warming`.

C++ jde o krok dál za C nabídnutím operátoru rozlišení rozsahu platnosti (`::`). Když se tento operátor dá před jméno proměnné, znamená použití její globální verze. Tedy `local()` zobrazuje `warming` jako 0.8, ale `::warming` jako 0.4. Na tento operátor narazíte znovu v prostorech jmen a ve třídách.

Globální nebo lokální?

Nyní, když můžete používat globální nebo lokální proměnné, které byste měli použít? Za prvé, globální proměnné vypadají lákavě, protože k nim mají všechny funkce přístup a nemusíme se zatěžovat s předáváním parametrů. Za snadný přístup však platíte vysokou cenou – nespolehlivostí programu. Zkušenost s programováním ukázala, že čím lepší práci program udělá při izolaci dat před zbytečným přístupem, tím lepší práci udělá pro uchování integrity dat. Nejčastěji byste měli používat lokální proměnné a předávat spíše data funkcím podle zásady potřebných znalostí, než zpřístupňovat údaje šmahem pomocí globálních proměnných. Jak uvidíte, OOP jde v izolaci dat o krok dále.

Globální proměnné však mají své uplatnění. Například byste mohli mít blok dat, který se má používat několika funkcemi, jako pole jmen měsíců nebo atomových hmotností prvků. Externí třída paměti je zvláště vhodná pro reprezentaci konstantních dat, proto můžete použít klíčové slovo const na ochranu dat před změnou.

```
const char * const months[12] =
{
    "Leden", "Unor", "Brezen", "Duben", "Kveten",
    "Cerven", "Cervenec", "Srpen", "Zari", "Rijen",
    "Listopad", "Prosinec"
};
```

První const chrání před změnou řetězce, druhý zajišťuje, že každý ukazatel v poli zůstává ukazuje na stejný řetězec, na který ukazoval původně.

Modifikátor static (lokální proměnné)

Modifikátor `static` se může používat jak pro lokální, tak pro globální proměnné. Podíváme se nyní na lokální případ. Když ho použijete uvnitř bloku, `static` vytvoří lokální proměnnou, která má statickou paměťovou třídu. To znamená, že ačkoli je proměnná známa uvnitř bloku, existuje, i když je blok neaktivní. Tedy statická lokální proměnná ucho-

vává svou hodnotu mezi voláním funkcí. (Statické proměnné by byly užitečné pro reinkarnaci – mohli byste je použít k předání tajných čísel účtů pro švýcarskou banku, když se zase objevíte.) Také když inicializujete statickou proměnnou, program ji inicializuje jednou, když startuje. Následná volání funkce proměnnou znovu neinicializují, tak jak to dělají v případě automatických proměnných. Výpis programu 8.17 tyto body objasňuje.

Mimochodem program ukazuje způsob zacházení s řádkovým vstupem, který může přesáhnout velikost přijímacího pole. Metoda vstupu `cin.get(input,ArSize)`, připomínáme, čte až do konce řádku nebo do `ArSize - 1` znaků, podle toho, co nastane dříve. Znak nového řádku ponechává ve vstupní frontě. Tento program čte znak, který následuje za vstupním řádkem. Jestliže je to znak nového řádku, pak se přečetl celý řádek. Když není, na řádku zůstalo více znaků. Program potom používá na vyřazení jeho zbytku cyklus, ale vy můžete tento programový kód upravit na použití zbytku řádku pro další vstupní cyklus. Program také využívá skutečnost, že pokus o přečtení prázdného řádku pomocí `get(char *, int)` způsobí, že se `cin` nastaví na false.

Výpis programu 8.17 static.cpp

```
// static.cpp – použiti statické lokální proměnné
#include <iostream>
using namespace std;
// konstanty
const int ArSize = 10;

// prototyp funkce
void strcount(const char * str);

int main()
{
    char input[ArSize];
    char next;

    cout << "Zadejte radek textu:\n";
    cin.get(input, ArSize);
    while (cin && input[0])        // pouze cin nepracuje
    {
        cin.get(next);
        while (next != '\n')       // řetězec nepasoval!
            cin.get(next);
        strcount(input);
        cout << "Zadejte dalsi radek textu (prazdny radek na ukonceni):\n";
        cin.get(input, ArSize);
    }
    cout << "Sbohem\n";
    return 0;
}
void strcount(const char * str)
{
```

```
    static int total = 0;          // statická lokální proměnná
    int count = 0;                 // automatická lokální proměnná

    cout << "\"" << str <<"\" obsahuje ";
    while (*str++)                 // jděte na konec řetězce
        count++;
    total += count;
    cout << count << " znaku\n";
    cout << total << " znaku celkem\n";
}
```

Kompatibilita:

Některé starší kompilátory neimplementují požadavek, že když `cin.get(char *,int)` čte prázdný řádek, nastavuje chybový bit na příznak chyby, což způsobí, že se `cin` nastaví na false. V tomto případě můžete nahradit test

```
while (cin)
```

testem:

```
while (input[0])
```

Nebo pro test, který pracuje jak pro starší, tak pro novější implementace, udělejte toto:

```
while (cin && input[0])
```

Zde je výstup z programu

```
Zadejte radek textu:
krasne kalhoty
"krasne ka" obsahuje 9 znaku
9 znaku celkem
Zadejte dalsi radek textu (prazdny radek na ukonceni):
diky
"diky" obsahuje 4 znaku
13 znaku celkem
Zadejte dalsi radek textu (prazdny radek na ukonceni):
odlouceni je takova sladka bolest
"odlouceni" obsahuje 9 znaku
22 znaku celkem
Zadejte dalsi radek textu (prazdny radek na ukonceni):
ok
"ok" obsahuje 2 znaku
24 znaku celkem
Zadejte dalsi radek textu (prazdny radek na ukonceni):

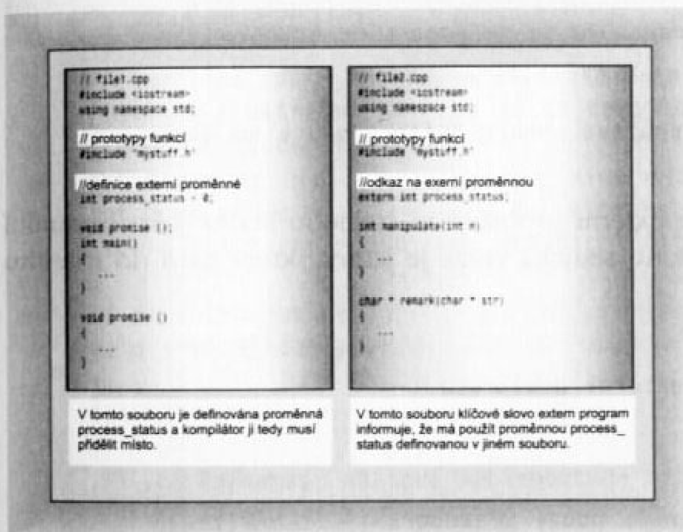
Sbohem
```

Všimněte si, že protože je velikost pole 10, program nepřečte více jak 9 znaků na řádku. Také si všimněte, že automatická proměnná `count` se nastaví na 0 pokaždé, když se funkce zavolá. Avšak statická proměnná `total` se nastaví na 0 jednou na začátku. Poté si

`total` udrží svou hodnotu mezi voláními funkce, takže je schopná udžovat průběžný součet.

Vazba a externí proměnné

Použití modifikátoru `static` na externí proměnnou nabývá významu v programech z několika souborů. V tomto kontextu je statická externí proměnná lokální vzhledem k souboru, který ji obsahuje. Ale běžná externí proměnná má externí vazbu, což znamená, že se může používat v různých souborech. Při externí vazbě může obsahovat externí definici proměnné pouze jediný soubor. Ostatní soubory, které chtějí tuto proměnnou používat, musí v referenční deklaraci použít klíčové slovo `extern`. Viz obrázek 8.6.



Obrázek 8.6 Definiční a referenční deklarace

Jestliže soubor neposkytuje externí deklaraci proměnné, nemůže použít externí proměnnou definovanou někde jinde:

```
// soubor1
int errors = 20;          // globální deklarace
...
_______________________
// soubor2
...                        // chybí deklarace extern int errors
void froobish()
{
     cout << errors;  // zhoubný pokus použít errors
...
```

Pokud se soubor pokouší definovat druhou externí proměnnou stejného jména, je to chyba:

```
// soubor1
int errors = 20;              // externí deklarace
...
_______________________
```

```
// soubor2
int errors;                // chybná deklarace
void froobish()
{
    cout << errors; // zhoubný pokus použití errors
..
```

Správný postup je použít ve druhém souboru klíčové slovo extern:

```
// soubor1
int errors = 20;           // externí deklarace
...
_______________________
// soubor2
extern int errors;         // odkazuje se na errors ze soubor1
void froobish()
{
     cout << errors;   // používá errors definovanou v soubor1
..
```

Ale, když soubor deklaruje statickou externí proměnnou stejného jména jako normální proměnná deklarovaná v jiném souboru, statická verze je jediná, která patří do rozsahu platnosti tohoto souboru:

```
// soubor1
int errors = 20;           // externí deklarace
...
_______________________
// soubor2
static int errors = 5;   // známa pouze v soubor2
void froobish()
{
     cout << errors;   // používá errors definovanou v soubor2
     ...
```

Pamatujte:

V programech z více souborů můžete definovat externí proměnnou pouze v jednom souboru. Všechny ostatní soubory, které tuto proměnnou používají, ji musí deklarovat s klíčovým slovem `extern`.

Běžné externí proměnné používejte ke sdílení dat mezi různými částmi několikasouborového programu. Statické externí proměnné používejte pro sdílení dat mezi funkcemi, které se nachází pouze v jednom souboru. (Prostory jmen pro to nabízejí alternativní metodu.) Také, jestliže vytvoříte statickou externí proměnnou, nemusíte se obávat o konflikt jejího jména s externími proměnnými, které se nalézají v ostatních souborech.

Výpisy programů 8.18 a 8.19 ukazují, jak C++ zachází s externími a externími statickými proměnnými. Výpis programu 8.18 (`twofile1.cpp`) definuje externí proměnné `tom` a `dick` a statickou externí proměnnou `harry`. Funkce `main()` v tomto souboru zobrazuje adresy tří proměnných a potom volá funkci `remote_access()`, která je definována ve

druhém souboru. Výpis programu 8.19 (`twofile2.cpp`) tento soubor ukazuje. Jako doplněk k definici `remote_access()` soubor používá klíčové slovo `extern` na sdílení proměnné `tom` s prvním souborem. Dále soubor definuje statickou proměnnou, která se jmenuje `dick`. Modifikátor `static` vytváří vzhledem k souboru z této proměnné lokální proměnnou a potlačuje globální definici. Potom soubor definuje externí proměnnou, která se jmenuje `harry`. Může to tak udělat, aniž by se dostal do konfliktu s proměnnou `harry` z prvního souboru, protože první `harry` má pouze interní vazbu. Potom funkce `remote_access()` zobrazuje adresu těchto tří proměnných, takže je můžete porovnat s adresami odpovídajících proměnných v prvním souboru. Nezapomeňte zkompilovat a sestavit dohromady oba soubory, abyste získali úplný program.

Výpis programu 8.18 twofile1.cpp

```
// twofile1.cpp - externí a statické externí proměnné
#include <iostream>       // má se kompilovat se souborem twofile2.cpp
using namespace std;
int tom = 3;              // definice externí proměnné
int dick = 30;            // definice externí proměnné
static int harry = 300;   // definice statické externí proměnné

// prototyp funkce
void remote_access();

int main()
{
    // Poznámka: některé implementace vyžadují, že přetypujete
    // adresu na typ unsigned

    cout << "main() podava zpravu o nasledujich adresach:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
    remote_access();
    return 0;
}
```

Výpis programu 8.19 twofile2.cpp

```
// twofile2.cpp - externí a statické externí proměnné
#include <iostream>
using namespace std;
extern int tom;           // tom definovaný kdekoli
static int dick = 10;     // skrývá externí proměnnou dick
int harry = 200;          // definice externí proměnné,
                          // nedochází ke konfliktu s proměnnou harry
                          // v twofile1

void remote_access()
{
    // Poznámka: některé implementace vyžadují, že přetypujete
```

```
        // adresu na typ unsigned

        cout << "remote_access() podava zpravu o nasledujich adresach:\n";
        cout << &tom << " = &tom, " << &dick << " = &dick, ";
        cout << &harry << " = &harry\n";
    }
```

Zde je výstup:

```
main() podava zpravu o nasledujich adresach:
0041C1C4 = &tom, 0041C1C8 = &dick, 0041C1CC = &harry
remote_access() podava zpravu o nasledujich adresach:
0041C1C4 = &tom, 0041C220 = &dick, 0041C224 = &harry
```

Jak můžete vidět, oba soubory používají stejnou proměnnou `tom`, ale různé proměnné `dick` a `harry`.

Kvalifikátory paměťové třídy: const, volatile a mutable

Určitá klíčová slova jazyka C++, která se nazývají kvalifikátory, poskytují dodatečnou informaci o paměti. Nejběžněji používaným je `const` a jeho účel jste již viděli. Signalizuje, že by jednou inicializovaná paměť neměla být programem měněna. Za chvíli se ke `const` vrátíme.

Klíčové slovo `volatile` naznačuje, že hodnota v paměťové lokaci může být změněna, i kdyby v programu obsah nic nemodifikovalo. Je to méně záhadné, než to zní. Například byste mohli mít ukazatel na hardwarovou lokaci, která obsahuje čas nebo informaci ze sériového portu. Zde mění obsah hardware, ne program. Smyslem tohoto klíčového slova je zlepšit optimalizační schopnosti kompilátorů. Například předpokládejme, že si kompilátor všimne, že program používá hodnotu určité proměnné uvnitř několika příkazů. Spíše než aby program hodnotu dvakrát vyhledal, kompilátor by ji měl ukrýt do registru. Tato optimalizace předpokládá, že se hodnota proměnné nezmění během dvojího použití. Když nenadeklarujete proměnnou jako `volatile`, potom se může kompilátor při provádění této optimalizace cítit svobodně. Jestliže deklarujete, že proměnná má být `volatile`, říkáte kompilátoru, že nemá tento druh optimalizace provádět.

C++ nedávno dodal nový kvalifikátor: `mutable`. Můžete ho použít k označení, že se určitý člen struktury (nebo třídy) může měnit, i když je určitá proměnná tohoto typu konstantou. Například uvažujte následující programový kód:

```
struct data
{
    char name[30];
    mutable int accesses;
    ...
};
const data veep = {"Claybourne Clodde", 0, ... };
strcpy(veep.name, "Joye Joux");    // není povoleno
veep.accesses++;                    // je povoleno
```

Kvalifikátor `const` u `veep` zabraňuje programu měnit jeho členy, ale kvalifikátor `mutable` u členu `accesses` chrání tento člen před omezením. Tato kniha nebude používat `volatile` nebo `mutable`, avšak na učení o `const` je toho ještě více.

Více o `const`

V C++ (ale ne v C) modifikátor `const` slabě mění standardní paměťové třídy. Zatímco globální proměnná má externí vazbu standardně, `konstantní` globální proměnná má standardně interní vazbu. To jest, C++ zachází s globální `konstantní` definicí jako se statickou globální definicí. Na klíčové slovo `const`, když se používá v externí definici, můžete nahlížet, jako by mu předcházelo klíčové slovo `static`.

```
const int fingers = 10;     // stejné jako static const int fingers;
int main(void)
{
   ...
```

Aby vám C++ učinil život snazším, změnil pravidla pro konstantní typy. Předpokládejte například, že máte sadu konstant, které byste chtěli umístit do hlavičkového souboru a že ho použijete v několika souborech ve stejném programu. Jakmile preprocesor zavede obsah hlavičkového souboru do každého zdrojového souboru, každý bude obsahovat takovéto definice:

```
const int fingers = 10;
const char * warning = "Wak!";
```

Kdyby měly `konstantní` deklarace externí vazbu jako normální proměnné, byla by to chyba, protože globální proměnnou můžete definovat v jednom souboru jen jednou. To jest, pouze jediný soubor může obsahovat předchozí deklaraci, zatímco ostatní soubory musí poskytnout referenční deklarace pomocí klíčového slova `extern`. Kromě toho, pouze deklarace bez klíčového slova `extern` mohou inicializovat hodnoty.

```
// mělo by se vyžadovat extern, kdyby konstanta měla externí vazbu
extern const int fingers;        // nemůže být inicializována
extern const char * warning;
```

Takže, pro jeden soubor potřebujete jednu skupinu definic a jinou pro další soubory. Ale, protože externě definovaná `konstantní` data mají interní vazbu, můžete použít stejné deklarace ve všech souborech.

Interní vazba také znamená, že každý soubor spíše získá svou vlastní sadu konstant, než že ji sdílí. Každá definice je vzhledem k souboru, který ji obsahuje, privátní. Tedy dobrý důvod umístit definice konstant do hlavičkového souboru. Tudíž, jestliže vložíte stejný hlavičkový soubor do dvou souborů se zdrojovým kódem, získají stejnou množinu konstant.

Jestliže z nějakého důvodu chcete vytvořit konstanty s externí vazbou, můžete na potlačení standardní interní vazby použít klíčové slovo `extern`:

```
extern const states = 50;   // externí vazba
```

Klíčové slovo `extern` musíte použít na deklaraci konstant ve všech souborech. To je rozdíl oproti normálním externím proměnným, ve kterých klíčové slovo `extern` nepoužívá-

te, když definujete proměnnou, ale extern používáte v ostatních souborech, které tuto proměnnou používají. Také můžete na rozdíl od běžných proměnných hodnotu ta vyžadují inicializaci.

Když deklarujete kvalifikátor const uvnitř funkčního bloku, má rozsah platnosti bloku, což znamená, že je konstanta použitelná pouze tehdy, když program provádí kód uvnitř tohoto bloku. To znamená, že můžete vytvářet konstanty definované spolu s funkcí nebo blokem a nemusíte se obávat konfliktu jmen s konstantami definovanými někde jinde.

Paměťové třídy a funkce

Funkce mají také paměťové třídy, ačkoli výběr je mnohem omezenější než pro proměnné. Programovací jazyk C++, podobně jako C, vám nedovoluje definovat funkci uvnitř jiné funkce, proto mají všechny funkce automaticky atribut statické paměťové třídy, což znamená, že existují po celou dobu běhu programu. Standardně jsou funkce externí, což znamená, že mají externí vazbu a mohou být sdíleny napříč soubory. Ve skutečnosti můžete použít v prototypu funkce klíčové slovo extern na označení, že je funkce definovaná v jiném souboru, ale toto je nepovinné. (Aby program nalezl funkci v jiném souboru, pak tento soubor musí být jedním ze souborů, který se má kompilovat jako část programu nebo knihovního souboru, jenž se prohlíží sestavovacím programem.) Můžete také použít klíčové slovo static, což dodá funkci interní vazbu, která omezuje použití na jediný soubor. Toto klíčové slovo byste měli použít na prototyp a na definici funkce:

```
static int private(double x);
...
static int private(double x)
{
    ...
}
```

To znamená, že je funkce známá pouze v tomto souboru. Také to znamená, že můžeme použít stejné jméno pro jinou funkci v jiném souboru. Stejně jako u proměnných, statická funkce potlačí externí definici v souboru, který obsahuje statickou deklaraci. Jestliže definujete svou vlastní funkci strlen() v jednom souboru mnohasouborového programu a deklarujete funkci s kvalifikátorem static, pak tento jediný soubor používá vaši definici funkce strlen(), zatímco ostatní používají knihovní verzi. Jestliže definujete svou vlastní funkci strlen() a nedeklarujete ji jako static, potom kompilátor používá vaši verzi ve všech souborech programu. Viz poznámka Kde C++ nalézá funkce.

Vložené funkce se chovají jinak než běžné. Standardně mají interní vazbu a odtud jsou vzhledem k souboru, který je obsahuje, lokální. Z tohoto důvodu je v pořádku umístit definice vložených funkcí do hlavičkového souboru.

Kde C++ nalézá funkce

Předpokládejme, že voláte funkci z určitého souboru v programu. Kde C++ hledá definici funkce? Jestliže prototyp funkce v tomto souboru indikuje, že je funkce statická, kompilátor se dívá po funkci pouze v tomto souboru. Jinak (také včetně sestavovacího programu) se dívá do všech souborů programu. Jestliže nalezne dvě definice, pošle vám chybové hlášení, protože

pro externí funkci můžete mít jen jednu definici. Jestliže selže při hledání definice v souborech, funkce se potom vyhledává v knihovnách. To má za následek, že jestliže definujete funkci, která má stejné jméno jako knihovní funkce, kompilátor raději použije vaši verzi než knihovní. (Avšak, jestliže použijete hlavičkový soubor, který deklaruje knihovní funkci a jestliže tento prototyp neodpovídá vaší verzi, nastanou problémy.) Některé kompilátory a sestavovací programy potřebují pro identifikaci, kterou knihovnu prohledávat, explicitní instrukce.

Jazyková vazba

Existuje další forma vazby, která se nazývá *jazyková vazba*. Za prvé trochu průpravy. Sestavovací program potřebuje různá symbolická jména pro každou jednotlivou funkci. V jazyce C bylo implementování jednoduché, protože pro jednu funkci mohlo být pouze jedno jméno. Takže pro vnitřní účely mohl kompilátor C přeložit jméno funkce v C jako například `spiff()` na `_spiff()`. Přístup jazyka C je označován jako *jazyková vazba C*. Jazyk C++ však může mít několik funkcí stejného jména, které se musí převést do jednotlivých symbolických jmen. Tudíž kompilátor C++ se oddává procesu generování různých symbolických jmen přetížených funkcí, který se nazývá ***komolení jmen***. Například by mohl konvertovat `spiff(int)` na, řekněme, `_spiff_i`, `spiff(double double)` na `_spiff_d_d`. Tento přístup jazyka C++ se nazývá *jazyková vazba C++*.

Když sestavovací program vyhledává funkci, aby odpovídala volání funkce v C++, používá jinou vyhledávací metodu, než jazyk C. Ale předpokládejme, že potřebujete v programu C++ použít předkompilovanou funkci z knihovny C! Například předpokládejme, že máte tento programový kód:

```
spiff(22); // chcete spiff(int) z knihovny C
```

Její symbolické jméno v souboru knihovny C je `_spiff`, ale pro náš hypotetický sestavovací program vyhledávací konvence C++ odpovídá hledání symbolického jména `_spiff_i`. Abyste se vyhnuli tomuto problému, můžete uplatnit k indikaci použitého protokolu, prototyp funkce:

```
extern "C" void spiff(int);   // použijte pro vyhledání jména protokol C
extern void spoff(int);       // použijte pro vyhledání jména protokol C++
extern "C++" void spaff(int); // použijte pro vyhledání jména protokol C++
```

První používá jazykovou vazbu C. Druhý a třetí jazykovou vazbu C++. Druhý to provádí standardně, třetí explicitně.

Paměťové třídy a dynamická alokace

Paměťové třídy popisují paměť, která je alokovaná pro proměnné (včetně polí a struktur) a funkce. Třídy se netýkají paměti alokované operátorem `new` v C++ (nebo u staršího C funkcí `malloc()`). Tento druh paměti nazýváme *dynamická paměť*. Jak jste viděli v kapitole 4, dynamická paměť se řídí operátory `new` a `delete`, nikoli pravidly rozsahu platnosti a vazby. Tudíž může být z jedné funkce alokována a z druhé uvolněna. Na rozdíl od automatické paměti, dynamická paměť není LIFO; pořadí alokace a uvolnění závisí na tom, kdy a jak se používají operátory `new` a `delete`. Prakticky kompilátor používá tři oddělené

oblasti paměti – jednu pro statické proměnné (může být dále dělena), jednu pro automatické proměnné a jednu pro dynamickou paměť.

Ačkoli se koncepty paměťových tříd netýkají dynamicky alokované paměti, týkají se ukazatelových proměnných používaných na sledování dynamické paměti. Například předpokládejte, že máte následující příkaz uvnitř funkce:

```
float * p_fees = new float [20];
```

80 bajtů (předpokládá se, že `float` jsou 4 bajty) alokovaných pomocí operátoru `new` zůstává v paměti, dokud je neuvolní operátor `delete`. Avšak ukazatel `p_fees` přestane existovat, když se funkce obsahující tuto deklaraci ukončí. Jestliže chcete mít těchto 80 bajtů alokované paměti přístupných pro jinou funkci, musíte předat nebo vrátit této funkci její adresu. Naproti tomu, uděláte-li stejnou deklaraci externě, potom bude ukazatel `p_fees` přístupný všem funkcím, které v souboru následují za touto deklarací. A použitím

```
extern float * p_fees;
```

vytvoříte stejný ukazatel, který je ve druhém souboru dostupný.

Kompatibilita:

Paměť alokovaná pomocí `new` se běžně uvolňuje, když program končí. Avšak není to vždy pravda. Například v DOSu požadavek na velký blok paměti za určitých okolností může mít za následek blok, který se automaticky nevymaže, když program končí.

Prostory jmen

Jméno v jazyce C++ se může odkazovat na proměnné, funkce, struktury, enumerátory, třídy a členy tříd a struktur. Když se programové projekty rozrostou, vzrůstá možnost konfliktu jmen. Když používáte knihovny tříd z více než jednoho zdroje, může dojít ke konfliktu jmen. Například, dvě knihovny by mohly obě definovat třídy, které se jmenují `List`, `Tree` a `Node`, avšak nekompatibilním způsobem. Mohli byste potřebovat třídu `List` z jedné knihovny, `Tree` z jiné a každá by mohla předpokládat svou vlastní verzi `Node`. Takové konflikty se nazývají problémy s prostory jmen.

Standard C++ poskytuje prostředky prostoru jmen, které zajišťují nad rozsahem platnosti jmen větší správu. V době psaní této knihy si tyto prostředky ještě razily cestu na trh a mohli byste ještě používat kompilátor, který je nepodporuje. Avšak jestliže současná verze vašeho kompilátoru nepodporuje prostory jmen, pravděpodobně další je podporovat bude.

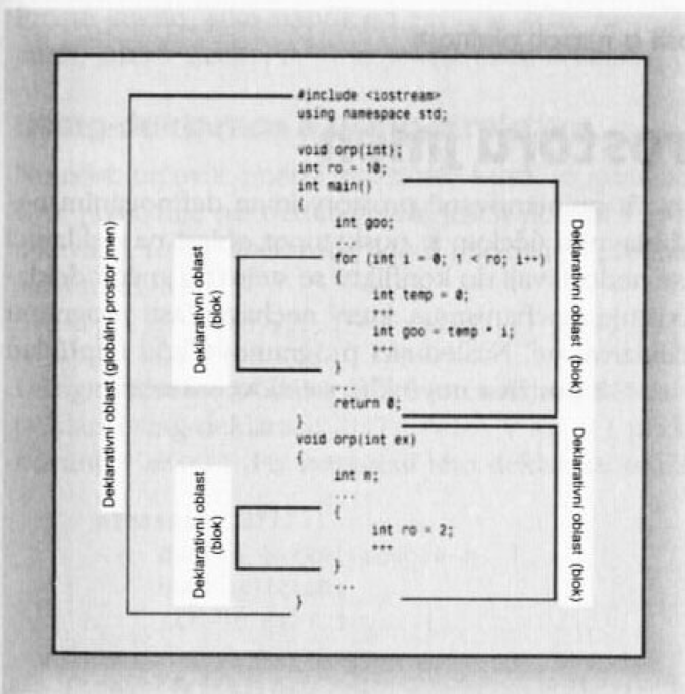
Tradiční prostory jmen v C++

Dříve než se podíváme na tyto nové prostředky, zopakujeme vlastnosti prostoru jmen, která již v C++ existují a zavedeme jistou terminologii. To pomůže vytvořit na prostory jmen názor, který vyhlíží důvěryhodněji.

První pojem je *deklarativní oblast*. Deklarativní oblast je oblast, ve které se mohou vytvářet deklarace. Například můžete deklarovat globální proměnnou vně jakékoli funkce. Deklarativní oblast této proměnné je soubor, ve kterém je deklarována. Pokud ji deklarujete uvnitř funkce, její deklarativní oblastí je nejvnitřnější blok, ve kterém je deklarovaná.

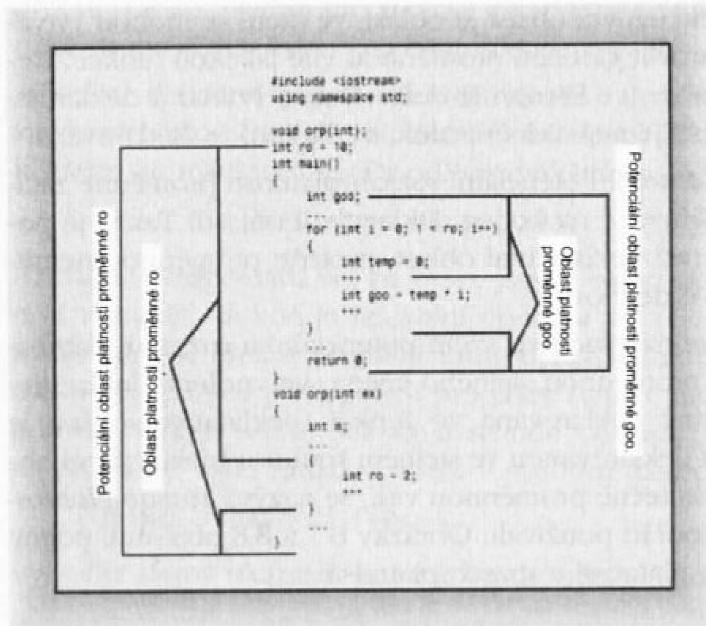
Druhý pojem je *potenciální rozsah platnosti*. Potenciální rozsah platnosti proměnné začíná v místě, kde je deklarovaná a rozšiřuje se na konec deklarativní oblasti. Takže je potenciální rozsah platnosti omezenější než deklarativní oblast, protože proměnnou nemůžete použít nad bodem, kde se poprvé definovala.

Proměnná by však nemusela být viditelná všude ve svém potenciálním rozsahu platnosti. Například může být překryta jinou proměnnou stejného jména ve vnořené deklarativní oblasti. Například lokální proměnná deklarovaná ve funkci (deklarativní oblast je funkce) překrývá globální proměnnou deklarovanou ve stejném souboru (deklarativní oblast je soubor). Část programu, který skutečně proměnnou vidí, se nazývá *rozsah platnosti*, což je způsob, kterým jsme pojem pořád používali. Obrázky 8.7 a 8.8 objasňují pojmy deklarativní oblast, potenciální rozsah platnosti a rozsah platnosti.



Obrázek 8.7 Deklarativní oblasti

Pravidla C++ o globálních a lokálních proměnných definují druh hierarchie prostoru jmen. Každá deklarativní oblast může deklarovat jména, která jsou nezávislá na jménech deklarovaných v jiných deklarativních oblastech. Lokální proměnná deklarovaná v jedné funkci se nedostává do rozporu s lokální proměnnou deklarovanou v jiné funkci.



Obrázek 8.8 Potenciální rozsah platnosti a rozsah platnosti

Nové vlastnosti prostoru jmen

Co nyní C++ dodává je schopnost vytvořit pojmenované prostory jmen definováním nového druhu deklarativní oblasti, jejímž hlavním účelem je poskytnout oblast na deklaraci jmen. Jména v jednom prostoru jmen se nedostávají do konfliktu se stejnými jmény deklarovanými v jiných prostorech jmen. Existuje mechanismus, který nechává části programu používat položky, které jsou v něm deklarované. Následující programový kód například k vytvoření dvou prostorů jmen `Jack` a `Jill` používá nové klíčové slovo `namespace`.

```
namespace Jack {
    double pail;
    void fetch();
    int pal;
    struct Well { ... };
}
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    int pal;
    struct Hill { ... };
}
```

Prostory jmen mohou být lokalizovány na globální úrovni nebo uvnitř jiného prostoru jmen, ale nesmějí být umístěny do bloku. Tudíž jméno deklarované v prostoru jmen má standardně externí vazbu (pokud se neodkazuje na konstantu nebo vloženou funkci).

Kromě uživatelsky definovaného prostoru jmen existuje ještě jeden – *globální prostor jmen.* Odpovídá deklarativní oblasti na úrovni souboru, takže co bývalo pojmenováno globálními proměnnými, se nyní popisuje jako část globálního prostoru jmen.

Jména v jednom libovolném prostoru jmen se nedostávají do konfliku se jmény v jiných prostorech jmen. Tudíž `fetch` v `Jack` může spoluexistovat s `fetch` v `Jill` a `Hill` v `Jill` může spoluexistovat s externí `Hill`. Pravidla řízení deklarací a definic v prostoru jmen jsou stejná jako pravidla pro globální deklarace a definice.

Prostory jmen jsou *otevřené*, to znamená, že k existujícím prostorům jmen můžete přidat nová jména. Například příkaz

```
namespace Jill {
    char * goose(const char *);
}
```

Nyní ovšem potřebujeme způsob pro přístup ke jménům v daném prostoru jmen. Nejjednodušší cesta je použít `::`, rozlišovací operátor platnosti na určení jména ve vztahu k jeho prostoru jmen:

```
Jack::pail = 12.34;      // použijte proměnnou
Jill::Hill mole;         // vytvořte objekt Queue
Jack::fetch();           // použijte funkci
```

Prosté jméno, jako například `pail`, je ***bliže neurčené jméno***, zatímco jméno s prostorem jmen, jako v `Jack::pail`, je ***jméno blíže určené***.

Using-deklarace a using-direktivy

Nutnost určovat jména pokaždé, když se používají, není přitažlivou vyhlídkou, a tak C++ poskytuje na zjednodušení jejich použití v prostoru jmen dva mechanismy. První je takzvaná *using-deklarace*, která spočívá v předřazení kvalifikovaného jména klíčovým slovem `using`:

```
using Jill::fetch;    // using-deklarace
```

Using-deklarace přidává určité jméno do deklarativní oblasti, ve které se vyskytuje. Například using-deklarace `Jill::fetch` v `main()` přidává `fetch` do deklarativní oblasti definované v `main()`. Po vytvoření této deklarace můžete místo `Jill::fetch` používat `fetch`.

```
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };
}
char fetch;
int main()
{
    using Jill::fetch;     // vloží fetch do lokálního prostoru jmen
/*  double fetch;          // Chyba! Již máme lokální fetch
    cin >> fetch;          // zapisuje hodnotu do Jill::fetch
    cin >> ::fetch;        // zapisuje hodnotu do globální fetch
    ...
}
```

Protože using-deklarace přidává jméno do lokální deklarativní oblasti, tento příklad znemožňuje vytvoření jiné lokální proměnné stejného jména `fetch`. Tedy jako jakákoli další lokální proměnná, `fetch` by měla potlačit globální proměnnou stejného jména. Umístění using-deklarace na externí úrověň přidává jméno do globálního prostoru jmen.

Druhý nový mechanismus je takzvaná *using-direktiva*. Sestává z předřazení jména prostoru jmen klíčovými slovy `using namespace`, zpřístupňuje všechna jména prostoru jmen bez použití rozlišovacího operátoru platnosti jmen.

```
using namespace Jack;  // zpřístupní všechna jména prostoru jmen Jack
```

Umístíme-li direktivu using na globální úrovni, zpřístupní jména prostoru jmen globálně. Viděli jste to mnohokrát v akci:

```
#include <iostream>     // umísťuje jména do prostoru jmen std
using namespace std;    // zpřístupňuje jména globálně
```

Jména můžeme zpřístupnit lokálně, když umístíme direktivu using do lokální deklarativní oblasti.

Použití using-direktivy k importování všech jmen masově *není* totéž, jako použití násobných using-deklarací. Je to téměř jako hromadná aplikace operátoru rozlišení platnosti jmen. Když použijete using-deklaraci, je to jako když se jméno deklaruje v místě using-deklarace. Jestliže je již určité jméno deklarované ve funkci, nemůžete pomocí using-deklarace importovat stejné jméno. Když však použijete using-direktivu, je to jako když jste deklarovali jména v menší deklarativní oblasti, která obsahuje jak using-deklaraci, tak samotný prostor jmen. Co se týče následujícího příkladu, mohl by to být globální prostor jmen. Můžete použít using-direktivu na importování jména, které je již deklarované ve funkci, ale lokální jméno překryje jméno prostoru jmen, stejně jako by je překryla globální proměnná stejného jména. Avšak ještě můžete použít operátor rozlišení rozsahu platnosti:

```
namespace Jill {
     double bucket(double n) { ... }
     double fetch;
     struct Hill { ... };
}
char fetch;                      // globální prostor jmen
int main()
{
     using namespace Jill;      // importuje jména prostoru jmen
     struct Hill Thrill;        // vytváří Jill::Hill
     double water = bucket(2); // používá Jill::bucket();
     double fetch;               // není chyba; skrývá Jill::fetch
     cin >> fetch;               // zapisuje hodnotu do lokální proměnné
fetch
     cin >> ::fetch;            // zapisuje hodnotu do globální proměnné
fetch
     cin >> Jill::fetch;         // zapisuje hodnotu do proměnné Jill::fetch
     ...
}
```

Zde se `Jill::fetch` umístí do lokálního prostoru jmen. Nemá lokální rozsah platnosti, takže nepotlačuje globální proměnnou `fetch`. Avšak obě proměnné `fetch` jsou přístupné, jestliže použijete operátor rozlišení rozsahu platnosti. Mohli byste chtít porovnat tento příklad s předcházejícím, který používal using-deklaraci.

Pamatujte:

Předpokládejme, že prostor jmen a deklarativní oblast definují obě stejné jméno. Pokud se pokusíte použít using-deklaraci na přenesení jména prostoru jmen do deklarativní oblasti, dostanou se obě jména do konfliktu a nastane chyba. Pokud použijete using-direktivu na přenesení jména prostoru jmen do deklarativní oblasti, lokální verze jména překryje verzi z prostoru jmen.

Obecně vzato, using-deklarace je bezpečnější na používání, protože přesně ukáže, jaká jména jste zpřístupnili. A pokud se jméno dostane do konfliktu s lokálním jménem, kompilátor vás upozorní. Using-direktiva přidá všechna jména, i ta, která možná nepotřebujeme. Vlastnost otevřenosti prostoru jmen znamená, že by mohl být jeho úplný seznam jmen rozšířen přes několik lokací, což znesnadní přesně se dozvědět, která jména jste dodali.

A co používaný přístup v příkladech této knihy?

```
#include <iostream>
using namespace std;
```

Za prvé, hlavičkový soubor `iostream` umísťuje vše do prostoru jmen `std`. Potom následující řádek exportuje všechno z tohoto prostoru jmen do globálního prostoru jmen. Tudíž tento přístup pouze reprodukuje dobu před prostory jmen. Hlavním rozumovým zdůvodněním tohoto přístupu je účelnost. Je snadné to udělat a jestliže váš systém nemá prostory jmen, můžete nahradit předchozí dva řádky původní formou:

```
#include <iostream.h>
```

Avšak naděje zastánců prostoru jmen je, že můžete být více vybíraví a používat buď operátor rozlišení nebo using-deklaraci. Tedy nepoužívejte toto:

```
using namespace std;  // vyhněte se, protože, je málo rozlišující
```

A místo toho udělejte toto:

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

Nebo ještě toto:

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

Další vlastnosti prostoru jmen

Deklarace prostoru jmen můžete do sebe vnořovat:

```
namespace elements
{
```

```
    namespace fire
    {
        int flame;
        ...
    }
    float water;
}
```

V tomto případě se odkazujeme na proměnnou `flame` prostřednictvím `elements::fire::flame`. Podobně můžete vnitřní jména zpřístupnit touto using-direktivou:

```
using namespace elements::fire;
```

Také můžete uvnitř prostorů jmen používat using-direktivy a using-deklarace:

```
namespace myth
{
    using Jill::fetch;
    using namespace elements;
}
```

Předpokládejme, že chcete přistoupit k `Jill::fetch`. Protože `Jill::fetch` je nyní částí prostoru jmen `myth`, kde se může `fetch` volat, můžeme k ní přistupovat touto cestou:

```
cin >> myth::fetch;
```

Samozřejmě, protože je také částí prostoru jmen `Jill`, můžeme ji stále volat pomocí `Jill::fetch`:

```
cout << Jill:fetch;  // zobrazí hodnotu zapsanou do myth::fetch
```

Nebo můžete ještě provést následující věc, která nezpůsobí žádný konflikt lokálních proměnných:

```
using namespace myth;
cin >> fetch;        // ve skutečnosti Jill::fetch
```

Nyní uvažujme o použití using-direktivy `myth` na prostor jmen. Using-direktiva je tranzitivní. Řekněme, že operace `op` je tranzitivní, jestliže `A op B` a `B op C` implikuje `A op C`. Například operátor > je tranzitivní. (To jest A větší než B a B větší než C implikuje A větší než C. Z tohoto kontextu vyplývá, že výraz

```
using namespace myth;
```

má za následek, že prostor jmen `elements` byl také dodán pomocí using-direktivy. To je totéž, jako:

```
using namespace myth;
using namespace elements;
```

Pro prostor jmen také můžete vytvořit alias. Například předpokládejme, že máte prostor jmen definovaný následovně:

```
namespace my_very_favorite_things { ... };
```

Z `mvtf` můžete pro `my_very_favorite_things` vytvořit alias následujícím příkazem:

```
namespace mvft = my_very_favorite_things;
```

Tento postup můžete použít na zjednodušení používání vnořených prostorů jmen:

```
namespace MEF = myth::elements::fire;
using MEF::flame;
```

Nepojmenované prostory jmen

Nepojmenovaný prostor jmen můžete vytvořit vynecháním jeho jména:

```
namespace          // nepojmenovaný prostor jmen
{
    int ice;
    int bandycoot;
}
```

Takto definovaný prostor jmen se chová, jako by byl následován using-direktivou; tj. jména deklarovaná v tomto prostoru jsou v potenciálním rozsahu platnosti až do konce deklarativní oblasti, která obsahuje nepojmenovaný prostor jmen. V tomto ohledu se jména chovají jako globální proměnné. Avšak protože prostor jmen nemá žádné jméno, nemůžete na zpřístupnění jména kdekoli explicitně použít using-direktivu nebo using-deklaraci. Především nemůžete tato jména z nepojmenovaného prostoru jmen použít v jiném souboru než v tom, který obsahuje deklaraci prostoru jmen. Což poskytuje alternativu použití statických externích proměnných.

Prostory jmen a budoucnost

Jakmile se programátoři seznámí s prostory jmen, objeví se obecná ustálená programová spojení. Především tu je velká naděje, že použití globálního prostoru jmen se bude vytrácet a že se knihovny tříd budou navrhovat pomocí mechanismů prostorů jmen. Vskutku, běžný C++ již volá po umístění funkcí standardní knihovny do prostoru jmen, který se nazývá `std`.

Jak bylo poznamenáno dříve, změny jmen hlavičkových souborů tyto změny odrážejí. Starší styl hlavičkových souborů, jako například `iostream.h`, nepoužívá prostory jmen, ale novější hlavičkový soubor `iostream` by měl používat prostor jmen `std`.

Shrnutí

C++ rozšířil schopnosti funkcí C. Použitím klíčového slova `inline` v definici funkce a umístěním této definice před její první volání doporučujete kompilátoru C++, aby vytvořil vloženou funkci. Tedy místo toho, aby program skákal do oddělené sekce programového kódu kvůli provedení funkce, nahrazuje její volání odpovídajícím vloženým programovým kódem. Prostředek vkládání by se měl používat pouze tehdy, když je funkce krátká.

Referenční proměnná je druhem přestrojeného ukazatele, který vám umožní vytvořit alias (druhé jméno) proměnné. Referenční proměnné se primárně používají jako parametry funkcí, které zpracovávají struktury a objekty tříd.

Prototypy v C++ vám umožňují definovat pro parametry standardní hodnoty. Když volání funkce vynechá odpovídající parametr, program použije standardní hodnotu. Když funkce zahrne hodnotu parametru, program použije tuto hodnotu namísto standardní. Standardní parametry se mohou poskytovat zprava doleva ze seznamu parametrů. Tudíž, když poskytnete standardní hodnotu určitému parametru, musíte také poskytnout standardní hodnoty pro jeho parametry po jeho pravé straně.

Signatura funkce je její seznam parametrů. Můžete definovat dvě funkce, které mají stejné jméno za předpokladu, že mají různou signaturu. To se nazývá polymorfismus funkcí, neboli přetížení funkcí. Běžně přetěžujete funkce, abyste poskytli v podstatě stejné služby různým typům dat.

Šablony funkcí automatizují proces přetěžování funkcí. Definujete funkci, která používá obecně použitelný typ a určitý algoritmus a kompilátor generuje odpovídající definice funkcí pro určité typy parametrů, které používáte v programu.

C++ vás při návrhu programu povzbuzuje v používání paralelních souborů. Efektivní organizační strategie je použití hlavičkového souboru pro definování uživatelských typů funkčních prototypů funkcí k manipulaci s uživatelskými typy. Použijte samostatný soubor zdrojového kódu na definice funkcí. Hlavičkový a zdrojový soubor společně definují a implementují uživatelsky definované typy a určují, jak se mohou používat. Potom `main()` a další funkce, které je používají, mohou vstoupit do třetího souboru.

Paměťové třídy v C++ definují, jak dlouho proměnné zůstávají v paměti a jaké části programu k nim mají přístup (rozsah platnosti a vazba). Automatické proměnné se definují uvnitř bloku, jako například tělo funkce nebo blok uvnitř těla. Existují a jsou známy pouze tehdy, když program provádí příkazy v bloku, který obsahuje definice.

Statické proměnné existují během trvání programu. Proměnná definovaná vně nějaké funkce má externí paměťovou třídu. Je známa všem funkcím v souboru, který následuje za její definicí (rozsah platnosti souboru) a zpřístupňuje se dalším souborům v programu (externí vazba). V dalším souboru, který takovou proměnnou používá, ji musíte deklarovat pomocí klíčového slova `extern`. Proměnná definovaná vně nějaké funkce, ale blíže určená klíčovým slovem `static`, má rozsah platnosti souboru a nezpřístupňuje se dalším souborům (interní vazba). Proměnná definovaná vně bloku, ale blíže určená klíčovým slovem `static` je lokální vzhledem k tomuto bloku, ale její hodnota zůstává po dobu trvání programu.

Funkce v C++ mají standardně externí paměťovou třídu, takže se mohou sdílet mezi soubory. Ale vložené funkce a funkce, které jsou blíže určeny klíčovým slovem `static`, mají interní vazbu a omezují se na definiční soubor.

Prostory jmen vám umožňují pojmenovávat oblasti, ve kterých můžete deklarovat identifikátory. Úmyslem je omezovat konflikty jmen, zvláště ve velkých programech, které používají programové kódy od různých dodavatelů. Identifikátory v prostoru jmen se mohou zpřístupnit použitím operátoru rozlišení rozsahu platnosti, použitím using-deklarace nebo using-direktivy.

Opakovací otázky

1. Jaké druhy funkcí jsou dobrými kandidáty pro vložené funkce?
2. Předpokládejte, že funkce `song()` má tento prototyp:

   `void song(char * name, int times);`

   a) Jak byste upravili prototyp, aby standardní hodnota pro `times` byla 1?

   b) Jaké změny byste provedli v definici funkce?

   c) Můžete poskytnout standardní hodnotu „`0, Muj Tati`" pro `name`?
3. Napište přetíženou verzi funkce `iquote()`, která zobrazuje svůj parametr uzavřený do znaků dvojitých uvozovek. Napište tři verze: jednu pro parametr `int`, druhou pro `double` a třetí pro `řetezec`.
4. Zde je šablona struktury:

```
struct box
{
    char maker[40];
    float height;
    float width;
    float length;
    float volume;
};
```

   a) Napište funkci, která se odkazuje na strukturu `box` jako na její formální parametr a zobrazuje hodnotu každého členu.

   b) Napište funkci, která se odkazuje na strukturu `box` jako na její formální parametr a nastavuje člen `volume` na součin ostatních tří dimenzí.
5. Tady je několik požadovaných výsledků. Určete, zda může být každý uskutečněn pomocí standardních parametrů, přetížením funkcí, obojím nebo ničím. Poskytněte vhodné prototypy.

   a) `mass(density, volume)` navrací hmotnost objektu, který má hustotu `density` a objem `volume`, kdežto `mass(density)` navrací hmotnost, která má hustotu `density` a objem 1.0 kubický metr.

   b) `repeat(10, "Jsem OK")` zobrazuje naznačený řetězec desetkrát, kdežto `repeat("Jste to ale hlupak")` zobrazuje naznačený řetězec pětkrát.

   c) `average(3,6)` navrací průměr typu `int` dvou parametrů typu `int`, kdežto `average(3.0, 6.0)` navrací průměr typu `double` dvou hodnot typu `double`.

   d) `mangle („Tesi me, ze se s vami setkavam")` navrací znak `T` nebo ukazatel na řetězec „`Tesi me, ze se s vami setkavam`" v závislosti na tom, zda přiřadíte návratovou hodnotu do proměnné typu `char` nebo do `char *`.

   e) `average(3, 6)` navrací průměr typu `int` dvou parametrů `int`, když se zavolal jeden soubor, `double` dvou parametrů typu `int`, když se ve stejném programu zavolal jiný.
6. Napište šablonu funkce, která navrací větší ze dvou parametrů.

7. Je dána šablona z opakovací otázky 6 a struktura `box` z opakovací otázky 4, poskytněte specializaci šablony, která vezme dva parametry typu box a navrací ten, který má větší člen `volume`.
8. Jakou paměťovou třídu použijete pro následující situace?
   a) `homer` je formální parametr funkce.
   b) Proměnná `secret` má být sdílena dvěma soubory.
   c) Proměnná `topsecret` se má sdílet funkcemi v jednom souboru a skrývat před jinými.
   d) `beencancelled` sleduje, kolikrát byla volána funkce, která ji obsahuje.
9. Pojednejte o rozdílech mezi using-deklarací a using-direktivou.

Programovací cvičení

1. Napište funkci, která normálně přijímá jeden parametr, adresu řetězce, a jednou ho vytiskne. Avšak poskytuje-li se druhý parametr typu `int` a je nenulový, funkce tiskne řetězec tolikrát, kolikrát se funkce v tomto bodě zavolala. (Všimněte si, že počet kolikrát se řetězec tiskne, není roven hodnotě druhého parametru; je roven počtu krát, kolikrát se funkce volala.) Ano, je to hloupá funkce, ale přinutí vás použít některé postupy probrané v této kapitole. Funkci použijte v jednoduchém programu, který ukazuje, jak pracuje.
2. Struktura `CandyBar` obsahuje tři členy. První člen obsahuje značku cukrové tyčinky. Druhý její váhu (která může mít desetinnou část) a třetí počet kalorií (celé číslo). Napište program, který používá funkci, jež má jako parametr odkaz na `CandyBar`, ukazatel-na-`char`, `double` a `int` a poslední tři hodnoty používá k nastavení odpovídajících členů struktury. Poslední tři parametry by měly mít standardní hodnoty „Millenium Much", 2.85 a 350. Také by měl program používat funkci, která přijímá odkaz na `CandyBar` a zobratuje obsah struktury. Použijete `const`, kde je to vhodné.
3. Níže je kostra programu. Zkompletujte ji použitím popsaných funkcí a prototypů. Všimněte si, že by měly existovat dvě funkce `show()`, každá používá standardní parametry. Použijte parametry s kvalifikátorem `const`, kde je to vhodné. Všimněte si, že `set()` by měla používat operátor `new` na alokování dostatečného prostoru na úschovu navrženého řetězce. Postupy zde použité jsou podobné těm, které používají a implementují třídy. (Měli byste v závislosti na vašem kompilátoru změnit jména hlavičkových souborů a vymazat using-direktivu.)

```
#include <iostream>
using namespace std;
#include <cstring>        // kvůli strlen(), strcpy()
struct stringy {
    char * str;          // ukazuje na řetězec
    int ct;              // délka řetězce (bez '\0')
    };
```

```
// prototypy pro set(), show(), a show() patří sem
int main()
{
    stringy beany;
    char testing[] = "Reality isn't what it used to be.";

    set(beany, testing);    // první parametr je odkaz,
                // alokuje prostor na úschovu kopie testing,
                // nastavuje člen str struktury beany, který má ukazovat
                // na nový blok, kopíruje testing do nového bloku
                // a nastavuje člen ct struktury beany
    show(beany);      // tiskne člen  string jednou
    show(beany, 2);   // tiskne člen  string dvakrát
    testing[0] = 'D';
    testing[1] = 'u';
    show(testing);    // tiskne řetězec testing jednou
    show(testing, 3); // tiskne řetězec testing třikrát
    show("Done!");
    return 0;
}
```

4. Zde je hlavičkový soubor:

```
// golf.h – pro pe8-3.cpp

const int Len = 40;
struct golf
{
    char fullname[Len];
    int handicap;
};

//  funkce se domáhá u uživatele o jméno a handicap
//  a nastavuje členy g g na zavedené hodnoty
//  navrací 1, když se zavede jméno, 0, když je jméno prázdný řetězec
int setgolf(golf & g);

//  funkce nastavuje strukturu golf kvůli poskytnutí jména a handicap
//  použitím hodnot předaných jako parametry funkci
void setgolf(golf & g, const char * name, int hc);

// funkce přenastavuje handicap na novou hodnotu
void handicap(golf & g, int hc);

// funkce zobrazuje obsah struktury golf
void showgolf(const golf & g);
```

Dejte dohromady několikasouborový program založený na této hlavičce. Jeden soubor, který se jmenuje `golf.cpp`, by měl poskytnout vhodné definice funkcí, které vyhovují prototypům v hlavičkovém souboru. Druhý by měl obsahovat `main()` a demonstrovat všechny vlastnosti prototypovaných funkcí. Například cyklus by měl požádat vstup o pole struktur typu `golf` a ukončit se, když je pole plné nebo

uživatel zavede prázdný řetězec jména hráče golfu. Funkce `main()` by pouze měla používat pro přístup ke strukturám golf prototypované funkce.

5. Napište šablonu funkce `max5()`, která přijímá jako své parametry pole o pěti položkách typu `T` a vrací jeho největší prvek. (Protože je velikost pevná, může být obtížné zakódovat ji do cyklu, než ji předat jako parametr.) Otestujte ji v programu, který používá funkci s polem o pěti hodnotách typu `int` a `double`.
6. Napište šablonu funkce `maxn()`, která přijímá jako své parametry pole položek typu T a celé číslo, které představuje počet prvků pole a která navrací jeho největší prvek. Otestujte to v programu, který používá šablonu funkce s polem o šesti hodnotách typu `int` a čtyřech typu `double`. Program by měl také zahrnovat specializaci, která přijímá pole ukazatelů-na-`char` jako parametr a počet ukazatelů jako druhý parametr a která navrací adresu největšího řetězce. Jestliže existuje více řetězců, které mají největší délku, funkce navrací adresu prvého, který je nejdelší. Otestujte specializaci pomocí pole o pěti řetězcových ukazatelích.

KAPITOLA 9

Objekty a třídy

Témata kapitoly:

- Procedurální a objektově orientované programování
- Koncept třídy
- Jak definovat a implementovat třídu
- Veřejný a soukromý přístup ke třídě
- Datové položky třídy
- Metody třídy (členské funkce třídy)
- Vytváření a používání objektů třídy
- Konstruktory a destruktory třídy
- `Konstantní` členské funkce
- Ukazatel `this`
- Vytváření polí objektů
- Rozsah platnosti třídy
- Abstraktní datové typy (ADT)

Objektově orientované programování (OOP) je určitý koncepční přístup návrhu programů a jazyk C++ rozšiřuje jazyk C o vlastnosti, které jeho použití usnadňují. Mezi nejdůležitější vlastnosti OOP patří:

- Abstrakce
- Zapouzdření a ukrytí dat
- Polymorfismus
- Dědičnost
- Znovupoužitelný kód

Třída je nejdůležitějším rozšířením C++, které uvedené vlastnosti umožňuje implementovat a spojit dohromady. V této kapitole začneme třídy zkoumat. Vysvětlíme pojmy abstrakce, zapouzdření, ukrývání dat a ukážeme, jak třída tyto rysy implementuje. Dozvíte se, jak třídu definovat, o její soukromé a veřejné části, a jak se vytváří metody pracující s daty třídy. Také vás tato kapitola seznámí s koncepcí konstruktorů a destruktorů, což jsou speciální metody třídy určené pro vytváření a rušení objektů náležejících třídě. Nakonec se seznámíte s ukazatelem `this`, důležitou součástí programování pomocí tříd. Následující kapitoly rozšiřují témata o překrývání operátorů (další druh polymorfismu) a dědičnost, což je základ znovupoužitelného kódu.

Procedurální a objektově orientované programování

Ačkoli jsme hledisko OOP prozkoumali již velice brzy, většinou jsme se drželi velice blízko ke standardnímu procedurálnímu přístupu jazyků, jako jsou C, Pascal nebo BASIC. Pojďme se podívat na příklad, který objasní, jak se pohled OOP liší od přístupu procedurálního programování.

Jako nejnovější člen baseballového týmu Genre Giants jste byl požádán o vedení statistiky o týmu. Přirozeně budete chtít využít svůj počítač. Kdybyste byl procedurální programátor, mohl byste uvažovat asi tímto způsobem:

Takže chci mít možnost vkládat jméno, počet stání na pálce, počet odpalů, průměrný odpal (pro ty, kteří se nezajímají o baseball nebo softball, průměrný odpal je počet odpalů vydělený počtem stání na pálce. Odpal je ukončen, když hráč oběhne všechny mety nebo je vyřazen protihráčem) a ostatní údaje o každém hráči. Počkat, počítač by mi měl usnadňovat práci, proto ho nechám některé hodnoty vypočítat, například průměrný odpal. Také chci, aby program vypisoval výsledky. Jak to mám zorganizovat? Myslím, že bych měl začít hned a použít funkce. Ano, vytvořím funkci `main()`, která bude volat další funkce pro vstup dat, samotný výpočet hodnot a poté funkci pro výpis výsledků. A co když získám data z další hry? Nechci všechno vkládat znova. Dobrá přidám funkci pro aktualizaci statistiky. Také asi budu potřebovat nějakou funkci pro nabídku k výběru mezi vkládáním dat, výpočtem, aktualizací a výpisem dat. Jak bych měl data reprezentovat? Mohl bych použít pole řetězců pro jména hráčů, další pole pro počet stání na pálce a ještě jedno pole pro počet úspěšných odpalů atd. Ne, to je hloupost. Mohu vytvořit strukturu, obsahující všechny informace o jednom hráči a poté použít pole těchto struktur pro reprezentaci celého týmu.

Zkrátka a dobře, nejdříve se soustředíte na procedury, které chcete použít, a poté uvažujete o reprezentaci dat. (Poznámka: Abyste nemuseli mít program spuštěn po celou herní sezónu, budete asi chtít data také ukládat do souboru a poté ze souboru načítat. Ale protože jsme zatím o souborech nehovořili, budeme tuto komplikaci nyní ignorovat.)

Nyní se podívejme, jak se váš pohled změní, jestliže budete používat objektový přístup. Začnete uvažovat nejdříve o datech. Navíc nebudete uvažovat pouze o reprezentaci samotných dat, ale budete také brát v potaz jejich plánované použití:

Úvaha

Tak se podíváme, o čem vlastně sbírám data. Samozřejmě o basebalovém hráči. Takže budu chtít objekt reprezentující celého hráče, nikoli pouze počet odpalů nebo průměrný odpal. Ano, toto bude můj základní kámen, objekt obsahující jméno a statistiky hráče. Budu potřebovat nějaké metody pro manipulaci s tímto objektem. Asi budu potřebovat metodu pro zadání základních informací o objektu. Objekt by měl některé informace vypočítat sám – přidám metody provádějící tyto výpočty. Program by měl tyto výpočty provádět automaticky, aniž by uživatel musel o jejich provedení požádat. Také budu potřebovat metody pro aktualizaci a výpis dat. Takže uživatel dostane tři možnosti interakce: inicializaci, aktualizaci a výpis dat. To je uživatelské rozhraní.

Stručně řečeno, soustředíte se na objekt tak, jak jej vnímá uživatel, přičemž uvažujete o datech potřebných pro reprezentaci objektu a o metodách vystihujících interakci uživatele s objektem. Po vytvoření popisu tohoto rozhraní se přesunete k problému jeho implementace a uložení dat. Nakonec vytvoříte program využívající nový návrh.

Abstrakce a třídy

Život je velmi složitý a s touto složitostí se vyrovnáváme pomocí zjednodušujících abstrakcí. Vy jste kolekcí více než jedné miliardy atomů. Někteří studenti psychologie by řekli, že vaše vědomí je kolekcí částečně autonomních agentů. Ale je mnohem jednodušší o sobě uvažovat jako o jednotce. U počítačů je abstrakce základním krokem k reprezentaci dat pomocí jejich interakce s uživatelem. To znamená, že abstrahujete základní vlastnosti problému a pomocí nich nastíníte řešení. V příkladu o baseballovém týmu rozhraní popisovalo, jak uživatel inicializuje, aktualizuje a zobrazuje data. Od abstrakce je to jen krůček k uživatelem definovanému datovému typu, což je v C++ třída, reprezentující uvedené rozhraní.

Co je typ

Zamysleme se trochu více nad tím, co tvoří typ. Například kdo to je vědátor? Pokud se přikloníte k oblíbenému stereotypu, můžete takového vědátora posuzovat podle jeho vzhledu – tlusté brýle s rámečkem, kapsa plná tužek atp. Po krátké úvaze můžete dojít k závěru, že vědátor se lépe definuje podle svého chování. Například podle toho, jak jedná v nepříjemných situacích. My jsme v podobné situaci, jestliže dovolíte trochu přehnanou analogii, i s procedurálním jazykem C. Nejdříve máte tendenci přemýšlet o datech podle jejich vzhledu – jak budou uložena v paměti. Typ `char` je například jeden bajt v paměti a typ `double` často zabírá bajtů osm. Ale jednoduchá úvaha nás dovede k závěru, že datový typ je také určován operacemi, které je s ním možné provádět. Například typ `int` můžete použít na jakékoli aritmetické operace. Můžete sčítat, odčítat, násobit a dělit celá čísla. Také můžete použít operátor modulo (%) pro získání zbytku po dělení.

Na druhé straně vezměme v úvahu ukazatele. Ukazatel bude v paměti často zabírat stejnou velikost jako typ `int`. Dokonce může být interně uložen jako celé číslo. Ale ukazatel neposkytuje stejné operace jako celá čísla. Například nemůžete vynásobit dva ukazatele. Taková operace nemá smysl, proto není v C++ implementována. Jestliže tedy nadefinujete proměnnou typu `int` nebo typu ukazatel na reálné číslo, nealokujete pouze místo v paměti, ale také určujete, které operace bude možné s proměnnou provádět. Stručně řečeno, specifikací základního typu určujete tyto dvě věci:

- Velikost paměti potřebné pro uložení obsahu proměnné.
- Množinu operací nebo metod, které je možné pomocí datového objektu provádět.

V případě vestavěných typů jsou tyto informace obsaženy již v kompilátoru. Ale jestliže v C++ definujete uživatelský typ, musíte tyto informace určit sami. Výměnou získáte možnost vytvářet nové datové typy odpovídající požadavkům reálného světa.

Třída

Třída je nástroj, kterým jazyk C++ umožňuje převést abstrakci do uživatelem definovaného typu. Kombinuje data a metody pro manipulaci s těmito daty do elegantního celku. Podívejme se na třídu reprezentující kapitálové vklady.

Nejdříve si musíme rozmyslet, jak budeme vklady reprezentovat. Mohli bychom vzít jednu akcii podílu jako základ a vytvořit třídu reprezentující jednu akcii. To by však v případě sta akcií mělo za následek potřebu sta objektů reprezentujících akcii, což není praktické. Namísto toho vezměme za základ aktuální podíl konkrétní osoby v daném vkladu. Počet vlastněných akcií bude součástí objektu. Při realistickém pohledu na věc bude potřeba pro daňové účely zaznamenat informace, jako jsou pořizovací cena a datum nákupu. Také by se mělo například počítat s dělením podílu. To vypadá pro začátek už trochu náročně, takže vezmeme v úvahu zidealizovaný stav zjednodušující pohled na věc. Konkrétně se omezíme na tyto operace:

- Získání podílu ve firmě.
- Nákup více akcií v rámci jednoho podílu.
- Prodání podílu.
- Aktualizace hodnoty akcie.
- Zobrazení informace o vkladech.

Tento seznam můžeme využít k definici veřejného rozhraní naší třídy, přičemž další operace ponecháváme jako cvičení pro zainteresované. Pro podporu tohoto rozhraní potřebujeme některá data ukládat. Opět použijeme zjednodušený pohled. Například nebudeme brát v úvahu standardní praxi v USA, kde je hodnota podílu násobkem osmi dolarů. Budeme ukládat následující informace:

- Název podniku.
- Počet akcií.
- Cenu jedné akcie.
- Celkovou hodnotu všech akcií.

V dalším kroku nadefinujeme třídu. Obecně má specifikace třídy dvě části:

- *Deklaraci třídy*, která popisuje datové složky a veřejné rozhraní.
- *Definici metod třídy*, která obsahuje implementaci jednotlivých metod třídy.

Zhruba řečeno, deklarace třídy obsahuje popis třídy, zatímco definice metod obsahují implementační detaily.

Výpis 9.1 obsahuje předběžnou deklaraci třídy `Stock`. (Jako pomůcku pro identifikaci tříd používáme častou, ale nikoli univerzální, konvenci pojmenovávání tříd.) Zajisté si všimnete podobnosti s deklarací struktury až na několik částí obsahujících deklaraci metod a částí pro veřejný a soukromý přístup k datům. Brzy tuto deklaraci vylepšíme (proto ji nepoužívejte jako definitivní model), ale nejdříve se podívejme, jak tato definice funguje.

Výpis 9.1. stocks.cpp

```
// začátek souboru stocks.cpp
class Stock // deklarace třídy
{
private:
        char company[30];
        int shares;
        double share_val;
        double total_val;
        void set_tot() { total_val = shares * share_val; }
public:
        void acquire(const char * co, int n, double pr);
        void buy(int num, double price);
        void sell(int num, double price);
        void update(double price);
        void show();
};   // všimněte si středníku na konci deklarace
```

Bližší pohled na detaily třídy získáte později, nejdříve prozkoumáme obecnější vlastnosti. Na začátku klíčové slovo jazyka C++ class identifikuje následující kód jako deklaraci třídy. Nový typ je identifikován jménem `Stock`. Tato deklarace nám dovoluje definovat proměnné typu Stock nazývané objekty nebo instance. Každý objekt reprezentuje jeden podíl. Například deklarace

```
Stock sally;
Stock solly;
```

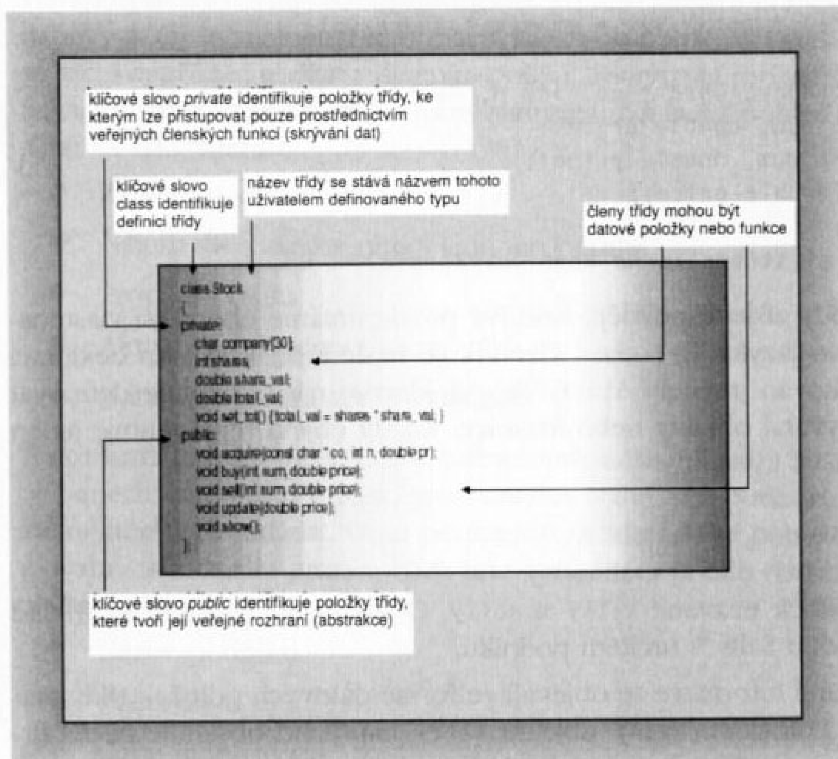
vytvoří dva objekty typu Stock nazvané `sally` a `solly`. Objekt sally by mohl například obsahovat informace o podílu Sally v určitém podniku.

Dále si všimněte, že ukládané informace se objevují ve formě datových položek třídy, například `company` a `shares`. Položka `company` objektu `sally` například obsahuje název firmy, položka share obsahuje počet vlastněných akcií, položka `share_val` udává cenu jedné akcie a položka `total_val` obsahuje celkovou cenu všech akcií, které Sally vlastní. Podobně i námi zvolené operace se objevují jako členské funkce třídy, například funkce `sell()` a `update()`. Členské funkce třídy se také nazývají metody třídy. Členská funkce může být definována přímo v deklaraci třídy, jako například funkce `set_tot()`, nebo může být reprezentována prototypem funkce tak jako ostatní funkce této třídy. Úplná definice metod přichází později, ale prototypy stačí k popisu funkčního rozhraní třídy. Spojení dat a metod do jednoho celku je největším přínosem třídy. Díky návrhu je vytvoření objektu typu `Stock` zároveň vytvořením pravidel pro jeho používání.

Již dříve jste viděli, že třídy `istream` a `ostream` mají členské funkce jako `get()` nebo `getline()`. Prototypy funkcí v definici třídy `Stock` ukazují, jak se členské funkce vytvářejí. Hlavičkový soubor `iostream` například obsahuje v deklaraci třídy `istream` prototyp funkce `getline()`.

Novinkou jsou také klíčová slova `public` a `private`. Tato návěští popisují přístupová práva k položkám třídy. Jakýkoli program používající objekt dané třídy může přímo přistupovat k položkám z části public. K soukromým datům třídy může přistupovat pouze pomocí ve-

řejných členských funkcí (nebo, jak uvidíte v kapitole 10 „Práce s třídami," pomocí spřátelené funkce). Například jediným způsobem, jak změnit položku `shares` třídy `Stock`, je použití některé veřejné členské funkce. Tudíž veřejné členské funkce se chovají jako prostředníci mezi programem a privátní částí objektu, nabízejí rozhraní mezi programem a objektem. Izolace dat od přímého přístupu z programu se nazývá skrývání dat. (C++ nabízí třetí klíčové slovo pro omezení přístupu k položkám třídy, protected, o kterém budeme hovořit v kapitole 12 „Dědičnost tříd.") Prohlédněte si obrázek 9.1. Zatímco ukrývání dat může být bezohledné v rámci obchodu s akciemi, v programování se jedná o dobrý zvyk, protože zachovává integritu dat.



Obrázek 9.1. Třída Stock

Návrh třídy se snaží oddělit veřejné rozhraní od implementačních detailů. Veřejné rozhraní reprezentuje abstraktní část návrhu třídy. Sloučení implementačních detailů a jejich oddělení od abstrakce se nazývá *zapouzdření*. *Skrývání dat* (uložení dat v soukromé části třídy) je příkladem zapouzdření, stejně jako oddělení funkčních detailů do privátní části, jak to dělá třída `Stock` v případě metody `set_tot()`. Jiným příkladem zapouzdření je běžný postup oddělení deklarace třídy a definice jejích metod do oddělených souborů.

OOP a jazyk C++

Objektově orientované programování je styl programování, který lze do jisté míry použít u jakéhokoli jazyka. Zajisté můžete mnoho idejí OOP přenést do jazyka C. Například kapitola 8 „Příběhy ve funkcích" nabízí příklad (výpisy 8.12, 13, 14), ve kterém hlavičkový soubor obsahuje definici struktury společně s prototypy funkcí, pracujících s touto strukturou. Funkce `main()` jednoduše definuje proměnné tohoto typu a používá uvedené funkce pro práci s těmito proměnnými; funkce `main()` nepřistupuje přímo k položkám struktury. V podstatě tento příklad definuje abstraktní datový typ a ukládá definici struktury společně s prototypy funkcí do hlavičkového souboru, čímž skrývá implementaci dat. Nicméně jazyk C++ obsahuje vlastnosti navržené speciálně pro implementaci přístupu OOP, takže vám umožňuje jít o několik kroků dále než jazyk C. Za prvé, uložení reprezentace dat a prototypů funkcí do jedné deklarace třídy namísto do souboru sjednocuje popis třídy tím, že všechny informace jsou uloženy do jedné deklarace třídy. Za druhé, přesunutí dat do privátní oblasti třídy zajistí, že přístup k nim bude možný pouze pomocí autorizovaných funkcí. Jestliže v našem příkladu z jazyka C bude funkce `main()` přistupovat přímo k položkám struktury, překročí sice pravidla OOP, ale neporuší žádná pravidla jazyka C. Ale pokus o přímý přístup například k položce `shares` objektu `Stock` porušuje pravidla jazyka C++ a kompilátor tento pokus odhalí.

Všimněte si, že skrývání dat nejenže vám brání v přímém přístupu k položkám třídy, ale také vás zprošťuje potřeby znát reprezentaci dat. Například metoda `show()` zobrazí celkovou hodnotu podílu. Tato hodnota může být uložena v rámci objektu, jak je tomu v našem příkladu, nebo může být vypočítána až v době použití. Z pohledu uživatele v tom není žádný rozdíl. Vše co potřebujete vědět je funkce jednotlivých metod třídy, to znamená, že potřebujete znát počet a typ parametrů funkce a její návratovou hodnotu. Princip spočívá v oddělení implementace od návrhu rozhraní. Jestliže později najdete lepší způsob reprezentace dat nebo implementace metod, můžete tyto detaily změnit bez nutnosti změny rozhraní třídy, což velmi zjednodušuje údržbu programu.

Veřejné nebo privátní?

Položky třídy, ať už se jedná o datové položky nebo funkce, můžete uvést buďto v části veřejné nebo privátní. Ale protože jedním z hlavních pravidel OOP je skrývání dat, jsou datové položky třídy téměř výlučně uváděny v privátní části třídy. Členské funkce tvořící rozhraní třídy se zapisují ve veřejné části třídy, jinak byste je nemohli volat z vnějšku třídy. Jak ukazuje deklarace třídy `Stock`, můžete také členské funkce uvést v privátní části. Takové funkce nemůžete volat přímo z programu, ale mohou je volat veřejné metody. Běžně se privátní funkce používají pro správu implementačních detailů, které nejsou součástí veřejného rozhraní.

Klíčové slovo `private` nemusíte v deklaraci třídy používat explicitně, protože se jedná o implicitní modifikátor přístupu k položkám objektu:

```
class World
{
    float mass;        // implicitně private
    char name[20];     // implicitně private
public:
```

```
    void tellall(void);
    ...
};
```

My však klíčové slovo `private` explicitně zapisovat budeme, abychom zdůraznili koncept skrývání dat.

Třídy a struktury

Popisy tříd se v mnohém podobají deklaracím struktur, obsahují však navíc členské funkce a návěští `public` a `private`, určující viditelnost. Jazyk C++ vlastně rozšiřuje struktury o stejné vlastnosti, jako mají třídy. Jediným rozdílem je, že implicitním modifikátorem přístupu je v případě struktury přístup `public`, zatímco u třídy je to přístup `private`. Programátoři v jazyku C++ většinou používají třídy k implementaci popisů tříd, zatímco omezující struktury používají k reprezentaci čistě datových objektů nebo někdy i k reprezentaci třídy bez privátních složek.

Implementace členských funkcí třídy

Zatím nám stále zbývá dodělat druhou část specifikace třídy: musíte doplnit kód členských funkcí, které jsou v deklaraci reprezentovány prototypy funkcí. Definice členských funkcí jsou velmi podobné definicím běžných funkcí. Obsahují hlavičku a tělo funkce. Mohou mít návratový typ a parametry. Ale mají také dvě zvláštní charakteristiky:

- Při definici členské funkce používáte operátor rozsahu platnosti (`::`) k určení třídy, do které funkce patří.
- Metody třídy mohou přistupovat k položkám v `privátní` části třídy.

Podívejme se nyní na tyto dva body.

Za prvé, v hlavičce členské funkce použijete operátor rozsahu platnosti (::) k určení třídy, do které funkce patří. Například hlavička členské funkce `update()` by mohla vypadat takto:

```
void Stock::update(double price)
```

Tato notace znamená, že definujeme funkci `update()` , kterážto je členskou funkcí třídy `Stock`. Nejedná se pouze o identifikaci funkce `update()` jako funkce členské, ale také to znamená, že stejný název metody můžeme použít v rámci jiné třídy. Například definice metody update() třídy `Buffoon` by měla tuto hlavičku:

```
void Buffoon::update()
```

Operátor rozsahu platnosti tedy určuje, ke které třídě metoda patří. Říkáme, že funkce `update()` má *třídní rozsah*. Další metody třídy `Stock` mohou v případě nutnosti použít metodu `update()` bez uvedení operátoru rozlišení. Je to možné proto, že obě metody patří do stejné třídy a metoda `update()` je v dosahu. Použití metody `update()` mimo deklaraci třídy nebo definice metod však vyžaduje zvláštní opatření, ke kterým se brzy dostaneme.

Jeden způsob, jak je možné na názvy metod pohlížet je ten, že úplný název metody třídy obsahuje název třídy. Říkáme, že název `Stock::update()` je ***kvalifikovaný název*** meto-

dy. Jednoduchý název `update()` je zkrácenou verzí kvalifikovaného názvu, který může být použit pouze v rozsahu třídy.

Další speciální vlastností členských funkcí je možnost přístupu k privátním složkám třídy. Například metoda `show()` by mohla obsahovat takovýto kód:

```
cout << "Company: " << company
        << " Shares: " << shares << '\n'
       << " Share Price: $" << share_val
       << " Total Worth: $" << total_val   << '\n';
```

Zde jsou `company`, `shares` a další privátními datovými položkami třídy `Stock`. Jestliže se pokusíte přistoupit k privátním položkám z jiné než členské funkce třídy, kompilátor tuto chybu odhalí již při překladu. (I když spřátelené funkce popsané v kapitole 10 představují výjimku z tohoto pravidla.)

Jestliže budeme mít na mysli tyto dvě zvláštnosti, můžeme implementovat metody třídy tak, jak je ukázáno ve výpisu 9.2. Tyto definice mohou být uloženy v odděleném souboru, nebo mohou být uloženy společně s deklarací třídy. Nyní v začátcích budeme předpokládat, že jsou definice uloženy společně s deklarací třídy v jednom souboru. Toto je nejjednodušší, i když ne nejlepší metoda, jak zpřístupnit deklaraci třídy definicím metod. (Nejlepším způsobem, jak uvidíme později, je použití hlavičkového souboru pro deklaraci třídy a zdrojového souboru pro definici metod třídy.)

Výpis 9.2. stocks.cpp

```
//další část stocks.cpp – implementace metod třídy
  #include <iostream>
  using namespace std;
  #include <cstdlib> // –nebo stdlib.h–pro exit()
  #include <cstring> // –nebo string.h–pro strncpy()
  void Stock::acquire(const char * co, int n, double pr)
  {
    strncpy(company, co, 29); // ořezání co v případě potřeby
    company[29] = '\0';
    shares = n;
    share_val = pr;
    set_tot();
  }
  void Stock::buy(int num, double price)
  {
    shares += num;
    share_val = price;
    set_tot();
  }
  void Stock::sell(int num, double price)
  {
    if (num > shares)
    {
        cerr << "Nemuzete prodat vic nez
          mate!\n";
```

```
            exit(1);
        }
        shares -= num;
        share_val = price;
        set_tot();
    }
    void Stock::update(double price)
    {
        share_val = price;
        set_tot();
    }
    void Stock::show()
    {
            cout << "Spolecnost: " << company << " Pocet akcii" << shares
                                  << " Cena akcie: Kc" << share_val
                                  << " Celkova cena: Kc" << total_val <<
'\n';
    }
```

Poznámky ke členským funkcím

Funkce `acquire()` obstarává prvotní vytvoření podílu v daném podniku, zatímco funkce `buy()` a `sell()` řídí přidávání a odebírání z existujícího vlastnictví. Jestliže se uživatel pokusí prodat více akcií než vlastní, funkce `sell()` zavolá funkci `exit()` a ukončí běh programu. (Výjimky, popsané v kapitole 14, „Přátelé, výjimky a další", umožňují pružnější reakci na chyby.) Čtyři další členské funkce mění hodnotu položky `total_val`. Spíše než zapisování výpočtu do každé z těchto funkcí bylo zvoleno volání funkce `set_tot()`. Protože tato funkce se týká implementace třídy a není součástí veřejného rozhraní, je uvedena v privátní části třídy. Pokud by výpočet byl delší, ušetřili byste si tímto zbytečné psaní a rozsah kódu. Zde však hlavní přínos spočívá v tom, že voláním funkce (namísto zapisováním výpočtu) zajistíte, že bude proveden vždy stejný výpočet. Také v případě potřeby úpravy výpočtu (což není v našem příkladu pravděpodobné) budete muset upravit kód pouze na jednom místě.

Metoda `acquire()` používá funkci `strncpy()` k vytvoření kopie řetězce. Jestliže jste již zapomněli, volání funkce `strncpy(s2, s1, n)` zkopíruje nejvýše `n` znaků řetězce `s1` do řetězce `s2`. Pokud řetězec `s1` obsahuje méně než `n` znaků, funkce `strncpy()` doplní řetězec `s2` nulovými znaky až do délky `n`. To znamená, že volání `strncpy(firstname,"Tim",6)` zkopíruje znaky `T`, `i` a `m` do položky `firstname` a přidá tři nulové znaky, aby se dosáhlo délky šesti znaků. Ale jestliže je řetězec `s1` delší než `n`, pak nebudou přidány žádné nulové znaky. To znamená, že v případě volání `strncpy(firstname,"Priscilla",4)` budou zkopírovány pouze znaky `P`, `r`, `i` a `s`, což vytvoří pole znaků, které ale nebude tvořit řetězec, protože neobsahuje na konci nulový znak. Proto funkce `acquire()` přidává nulový znak na konec znakového pole, aby bylo jisté, že se jedná o řetězec.

Objekt cerr a funkce exit()

Objekt `cerr` stejně jako objekt `cout` jsou instancemi třídy `ostream`. Rozdíl je v tom, že přesměrování výstupu programu ovlivní objekt `cout`, ale nikoli objekt `cerr`. Objekt `cerr` je používán pro chybové zprávy. Tudíž jestliže přesměrujete výstup programu do souboru a nastane nějaká chyba, bude chybová zpráva stále zobrazována na obrazovce. Funkce `exit()` ukončí program. Většinou se používá nenulový parametr k indikaci abnormálního ukončení programu a nulová hodnota při normálním ukončení. Nicméně pro maximální kompatibilitu s normou ANSI můžete jako návratové hodnoty použít konstanty `EXIT_SUCCESS` a `EXIT_FAILURE`. Zavolání funkce `exit()` z funkce `main()` má stejný efekt jako příkaz `return` v rámci funkce `main()`, ale na rozdíl od příkazu `return` funkce `exit()` ukončí program bez ohledu na funkci, ze které je volána. Hlavičkový soubor `cstdlib` (dříve `stdlib.h`) obsahuje prototyp funkce a definuje přenositelné návratové hodnoty.

Vložené metody

Funkce s definicí uvedenou v rámci deklarace třídy se automaticky stává vloženou funkcí. Funkce `Stock::set_tot()` je tedy vloženou funkcí. Deklarace tříd využívají vložené funkce většinou pro krátké funkce, což je případ funkce `set_tot()`.

Pokud chcete, můžete definovat vloženou funkci i mimo deklaraci třídy. Abyste tak učinili, použijte kvalifikátor `inline` v rámci implementace třídy:

```
class Stock
{
private:
    ...
    void set_tot(); // definice je uvedena zvlášť
public:
    ...
};
inline void Stock::set_tot() // použití inline v definici
{
    total_val = shares * share_val;
}
```

Protože vložené funkce mají vnitřní vazbu, jsou známy pouze v souboru, v němž jsou deklarovány. Nejjednodušším způsobem jak zajistit, aby vložené definice byly přístupné všem souborům v programu, je vložení definice do stejného hlavičkového souboru, ve kterém je odpovídající třída definována. (Některé systémy pro vývoj programů obsahující inteligentní sestavovací program umožňují uvedení vložených definic v separátním souboru.)

Mimochodem, podle přepisovacího pravidla je uvedení definice v rámci deklarace třídy ekvivalentní uvedení prototypu v deklaraci třídy a poté zapsání definice vložené funkce ihned za deklarací třídy. To znamená, že původní definice funkce `set_tot()` je ekvivalentní s právě uvedenou definicí.

Který objekt?

Nyní přikročíme k jednomu z nejdůležitějších aspektů používání objektů: jak aplikovat metodu na daný objekt. Následující kód

```
shares += num;
```

používá položku `shares` objektu. Ale kterého objektu? To je dobrá otázka! K jejímu zodpovězení se nejdříve zamyslíme nad tím, jak se nějaký objekt vytvoří. Nejjednodušší cestou je definice proměnných typu dané třídy:

```
Stock kate, joe;
```

Tímto způsobem vytvoříte dva objekty třídy `Stock`, jeden se jmenuje `kate` a druhý `joe`.

Dále uvažujte, jakým způsobem použít nějakou členskou funkci jednoho z těchto objektů. Odpovědí je, stejně jako u struktur a jejich položek, použití operátoru pro přístup ke složkám:

```
kate.show();  // vyvolání členské funkce objektu kate
joe.show();   // vyvolání členské funkce objektu joe
```

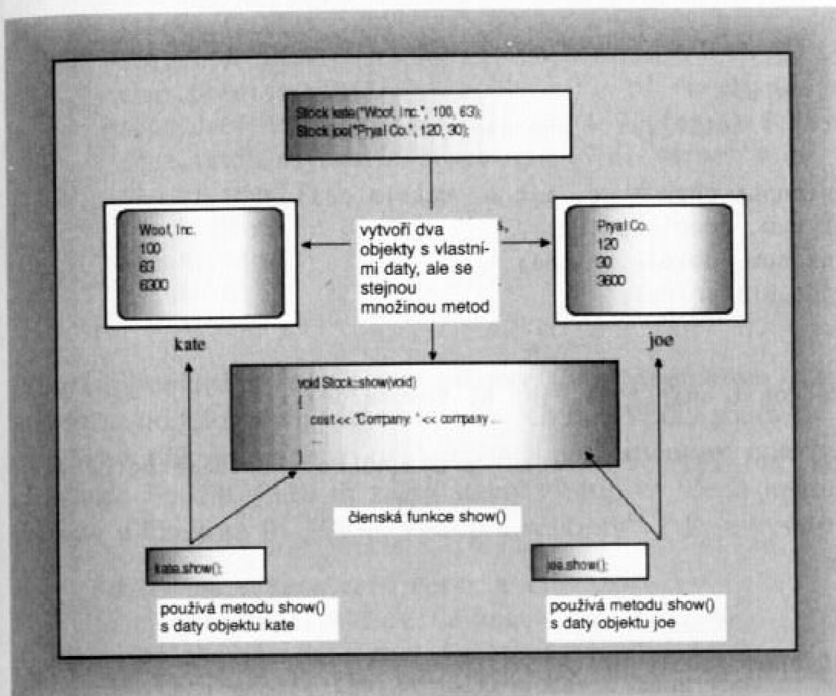
První řádek volá metodu `show()` objektu `kate`. To znamená, že metoda vyhodnotí odkaz na `shares` jako `kate.shares` a odkaz na položku `share_val` jako `kate.share_val`. Podobně volání `joe.show()` způsobí, že metoda `show()` interpretuje odkazy `shares` a `share_val` jako `joe.shares` a `joe.share_val`.

Pamatujte

Když zavoláte členskou funkci objektu, používá tato funkce datové položky objektu, který členskou funkci vyvolal.

Podobně volání `kate.sell()` použije metodu `set_tot()`, jako kdyby se jednalo o volání `kate.set_tot()`, což způsobí, že data budou brána z objektu `kate`.

Každý nově vytvořený objekt obsahuje místo pro uložení svých vnitřních proměnných, členů třídy. Ale všechny objekty náležející ke stejné třídě sdílí stejnou množinu metod a každá metoda existuje pouze v jedné kopii. Předpokládejme například, že `kate` a `joe` jsou objekty třídy `Stock`. Potom `kate.shares` zabírá část paměti a `joe.shares` zabírá jinou část paměti. Ale obě volání `kate.show()` a `joe.show()` volají stejnou metodu, to znamená, že provádí stejný úsek kódu. Pouze aplikují stejné metody na jiná data. Volání členské funkce je v některých OO jazycích nazýváno *zaslání zprávy*. Tedy zaslání stejné zprávy dvěma odlišným objektům spustí stejnou metodu, ta je ale aplikována na dva různé objekty. (Viz obrázek 9.2.)



Obrázek 9.2. Objekty, data a členské metody

Použití třídy

Nyní jste viděli, jak definovat třídu a její metody. Dalším krokem je napsání programu, který vytvoří a bude používat objekty nějaké třídy. Cílem jazyka C++ je maximálně přiblížit používání tříd k používání základních typů, jako jsou `int` a `char`. Objekt dané třídy můžete vytvořit buď pomocí deklarace proměnné daného typu nebo pomocí operátoru `new` pro alokaci nové instance třídy. Objekty můžete předávat jako parametry, vracet v návratových hodnotách funkcí nebo přiřazovat jeden objekt druhému. C++ nabízí prostředky pro inicializaci objektu, učí objekty `cout` a `cin` tyto objekty rozeznávat a dokonce nabízí prostředky pro automatickou konverzi typů objektů náležejících k podobným třídám. Bude chvíli trvat, než budete moci všechno toto dělat. Ale začněme od jednodušších vlastností. Již jste viděli, jak se deklaruje objekt a jak se volá jeho členská funkce. Výpis 9.3 kombinuje tyto techniky s deklarací třídy a definicí členských funkcí k vytvoření kompletního programu. Program vytvoří objekt třídy `Stock` pojmenovaný `stock1`. Program je jednoduchý, ale testuje všechny vlastnosti vložené do naší třídy.

Výpis 9.3. úplný program stock.cpp

```
#include <iostream>
  using namespace std;
  #include <cstdlib> // -nebo stdlib.h-pro exit()
  #include <cstring> // -nebo string.h-pro strncpy()
class Stock
{
  private:
      char company[30];
```

```
        int shares;
        double share_val;
        double total_val;
        void set_tot() { total_val = shares * share_val; }
    public:
        void acquire(const char * co, int n, double pr);
        void buy(int num, double price);
        void sell(int num, double price);
        void update(double price);
        void show();
};
void Stock::acquire(const char * co, int n, double pr)
{
    strncpy(company, co, 29); // zkrácení proměnné co v případě potřeby
    company[29] = '\0';
    shares = n;
    share_val = pr;
    set_tot();
}
void Stock::buy(int num, double price)
{
    shares += num;
    share_val = price;
    set_tot();
}
void Stock::sell(int num, double price)
{
  if (num > shares)
  {
    cerr << "Nemuzete prodat vic nez mate!\n";
    exit(1);
  }
  shares -= num;
  share_val = price;
  set_tot();
}
void Stock::update(double price)
{
        share_val = price;
        set_tot();
}
void Stock::show()
{
            cout << "Spolecnost: " << company
                                    << " Pocet akcii: " << shares << '\n'
                                    << " Cena akcie: Kc" << share_val
                                    << " Celkova cena: Kc" << total_val <<
'\n';
}
int main()
{
```

```
    Stock stock1;
    stock1.acquire("NanoSmart", 20, 12.50);
    cout.precision(2);                     // formát #.##
    cout.setf(ios_base::fixed);            // formát #.##
    cout.setf(ios_base::showpoint);        // formát #.##
    stock1.show();
    stock1.buy(15, 18.25);
    stock1.show();
    return 0;
}
```

Program používá tři formátovací příkazy. Cíleným efektem je zobrazení dvou číslic za desetinným oddělovačem včetně koncových nul. Podle současné praxe jsou potřeba pouze první dva příkazy, zatímco některé starší implementace potřebují první a třetí formátovací příkaz. Použití všech tří zajistí stejný výstup ve všech implementacích jazyka. Více se dozvíte v kapitole 16 „Vstup, výstup a soubory". Zde je výstup tohoto programu:

```
Spolecnost: NanoSmart Pocet akcii: 20
  Cena akcie: $12.50 Celkova cena: $250.00
Spolecnost: NanoSmart Pocet akcii: 35
  Cena akcie: $18.25 Celkova cena: $638.75
```

Všimněte si, že funkce `main()` slouží pouze k otestování třídy `Stock`. Jestliže naše třída funguje dle předpokladů, můžeme nyní třídu `Stock` použít jako uživatelem definovaný typ v jiných programech. Kritickým bodem při používání nového datového typu je pochopení členských funkcí. Implementačními detaily jste se zabývat nemuseli. Přečtěte si následující poznámku, týkající se modelu klient-server.

Model klient/server

OO programátoři často hovoří o návrhu programu v rámci modelu klient-server. Podle této koncepce je *klientem* program používající třídu. Deklarace třídy společně s definicí jejích metod tvoří *server*, který je zdrojem pro programy, jež jej potřebují. Klient používá server pouze pomocí veřejného rozhraní. To znamená, že jedinou starostí klienta a potažmo jedinou starostí programátora je znalost tohoto rozhraní. V odpovědnosti serveru a tedy zodpovědností návrháře serveru je zajistit, aby server spolehlivě a přesně prováděl akce v součinnosti se svým rozhraním. Toto programátorům dovoluje vylepšovat klienta nebo server nezávisle na sobě, aniž by změny v serveru měly nepředvídatelné dopady na chování klienta.

Dosavadní poznatky

Prvním krokem při specifikaci návrhu třídy je deklarace třídy. Deklarace třídy je podobná deklaraci struktury a může obsahovat datové položky a členské funkce. Deklarace obsahuje privátní část, přičemž k položkám uvedeným v této části mohou přistupovat pouze členské funkce. Deklarace má také veřejnou část a funkce zde definované mohou být používány jakýmkoli programem využívajícím danou třídu. Běžně jsou datové položky uváděny v privátní části a členské funkce ve veřejné části. Takže typická deklarace třídy vypadá takto:

```
class názevTřídy
  {
  private:
      deklarace datových položek
  public:
      prototypy členských funkcí
  };
```

Obsah veřejné části tvoří abstraktní část návrhu, veřejné rozhraní. Zapouzdření dat v privátní části chrání integritu dat a nazývá se *skrývání dat*. Třída je tedy způsob, jakým jazyk C++ umožňuje implementovat cíle OOP jako jsou zapouzdření, skrývání dat a abstrakce.

Dalším krokem při specifikaci třídy je implementace členských funkcí. V rámci deklarace třídy můžete použít namísto prototypu funkce kompletní definici funkce, ale běžnou praxí, s výjimkou velmi krátkých funkcí, je zapsání definic členských funkcí zvlášť. V takovém případě potřebujete operátor rozsahu platnosti k určení třídy, do které funkce patří. Předpokládejme například, že třída `Bozo` má metodu se jménem `retort()`, která vrací ukazatel na typ `char`. Potom bude hlavička funkce vypadat následovně:

```
char * Bozo::retort()
```

Jinými slovy, funkce `retort()` není pouze funkcí vracející ukazatel na `char`, ale je to funkce daného typu náležející do třídy `Bozo`. Úplný neboli kvalifikovaný název metody je `Bozo::retort()`. Název `retort()` je na druhé straně zkratkou úplného názvu a může být použit pouze v některých situacích, jako je kód metod dané třídy. Dalším způsobem, jak popsat tuto situaci, je říci, že název `retort` má platnost pouze v rámci své třídy, takže je potřeba použít operátor rozsahu platnosti ke kvalifikaci názvu vně deklarace třídy a metody třídy.

K vytvoření objektu, konkrétní instance třídy, použijte název třídy, jako by se jednalo o název typu:

```
Bozo bozetta;
```

Toto funguje, protože třída je uživatelem definovaný typ.

Členská funkce nebo metoda třídy může být vyvolána pouze pomocí objektu této třídy. Můžete tak učinit pomocí operátoru přístupu k položkám:

```
cout << bozetta.retort();
```

Tento příkaz vyvolá metodu `retort()` a kdykoli se bude metoda odvolávat na některou datovou položku, funkce použije hodnotu uloženou v rámci objektu bozetta.

Konstruktory a destruktory třídy

Je toho ještě více, co je třeba udělat na třídě Stock. Existují jisté standardní funkce nazývané *konstruktory* a *destruktory*, které byste měli u třídy definovat. Pojďme se podívat, proč jsou tyto funkce potřeba, a jak je vytvořit.

Jedním z cílů jazyka C++ je připodobnit používání tříd používání běžných datových typů. Nemůžete však objekt třídy Stock inicializovat stejným způsobem jako obyčejný typ int nebo struck:

```
int year = 2001                                            // v pořádku
  struct thing
  {
      char * pn;
      int m;
  };
  thing amabob = {"wodget", -23};                          // v pořádku
  Stock hot = {"Sukie's Autos, Inc.", 200, 50.25};      // NELZE!
```

Důvodem, proč nemůžete inicializovat objekt třídy Stock tímto způsobem je fakt, že datové položky objektu jsou definovány v privátní části třídy, což znamená, že program k nim nemůže přistupovat přímo. Jak jste již viděli, jediným způsobem jak přistupovat k datovým položkám, je pomocí členských funkcí. Tudíž budete potřebovat vytvořit odpovídající členskou funkci, pokud máte při inicializaci objektu uspět. (Bylo by možné objekt inicializovat výše uvedeným způsobem, jestliže by datové položky byly součástí veřejné sekce třídy. To by ale odporovalo jednomu z hlavních důvodů používání tříd, a to je skrývání dat.)

Obecně je nejlépe objekty inicializovat ihned po jejich vytvoření. Uvažujte například následující kód:

```
Stock gift;
gift.buy(10, 24.75);
```

Při aktuální implementaci třídy Stock nemá položka company v objektu gift žádnou hodnotu. Návrh třídy předpokládá, že dříve než bude zavolána jakákoli jiná metoda, zavolá uživatel metodu acquire(). Ale neexistuje způsob, jak to zajistit. Jedním ze způsobů řešení je automatická inicializace objektu při jeho vytvoření. K dosažení tohoto cíle nabízí C++ speciální členské funkce nazývané *konstruktory*, které mají za úkol inicializovat datové položky při vytváření objektu. Přesněji řečeno, C++ dodává název takové funkce a syntaxi pro její volání. Název je shodný s názvem třídy. Například konstruktorem třídy Stock by mohla být členská funkce Stock(). Prototyp konstruktoru a jeho hlavička mají jednu zajímavou vlastnost – ačkoli konstruktor nemá žádnou návratovou hodnotu, není jeho návratový kód definován jako void. Konstruktor vlastně nemá žádný typ.

Deklarace a definice konstruktorů

Pojďme vytvořit konstruktor třídy Stock. Protože třída Stock obsahuje tři hodnoty viditelné z vnějšího světa, musíte dát konstruktoru tři parametry. (Čtvrtá hodnota, total_val, je vypočítána z hodnot položek shares a share_val, takže ji není potřeba uvádět.) Pravděpodobně budete chtít nastavit pouze položku company a ostatní položky nastavit na nulu. Toho dosáhnete pomocí implicitních hodnot parametrů (viz kapitola 8). Takže prototyp by vypadal asi takto:

```
// prototyp konstruktoru s implicitními hodnotami pro některé parametry
  Stock(const char * co, int n = 0, double pr = 0.0);
```

Prvním parametrem je ukazatel na řetězec znaků, používaný k inicializaci znakového pole company v rámci třídy. Parametry pr a n slouží k nastavení hodnot položek shares a share_val. Všimněte si, že není uveden žádný návratový typ. Prototyp konstruktoru je uveden ve veřejné části třídy.

Dále je zde jedna možná definice konstruktoru:

```
// definice konstruktoru
  Stock::Stock(const char * co, int n, double pr)
  {
      strncpy(company, co, 29);
      company[29] = '\0';
      shares = n;
      share_val = pr;
      set_tot();
  }
```

Jedná se o stejný kód jako u funkce acquire(). Rozdíl spočívá v tom, že program automaticky spustí konstruktor při deklaraci objektu.

Upozornění

Začátečníci často používají názvy datových položek třídy jako parametry konstruktoru:

```
Stock::Stock(const char * company, int shares, double
    share_val) // nelze!
 {
 ...
 }
```

To je špatně. Parametry konstruktoru nereprezentují položky třídy, reprezentují hodnoty, které budou přiřazeny datovým položkám třídy. Proto musí mít odlišné názvy.

Použití konstruktoru

Jazyk C++ nabízí dva způsoby inicializace objektu pomocí konstruktoru. Prvním způsobem je explicitní volání konstruktoru:

```
Stock food = Stock("World Cabbage", 250, 1.25);
```

Tento příkaz nastaví položku company na hodnotu „World Cabbage", položku shares na hodnotu 250 atd.

Druhým způsobem je implicitní spuštění konstruktoru:

```
Stock garment("Furry Mason", 50, 2.5);
```

Tato kompaktnější forma volání je ekvivalentní s následujícím explicitním voláním:

```
Stock garment = Stock("Furry Mason", 50, 2.5);
```

C++ použije konstruktor kdykoli vytvoříte nový objekt nějaké třídy, dokonce i když použijete operátor new pro dynamickou alokaci paměti. Zde můžete vidět použití konstruktoru s operátorem new:

```
Stock *pstock = new Stock("Electroshock Games", 18, 19.0);
```

Tento příkaz vytvoří objekt třídy Stock, inicializuje jeho datové položky podle zadaných parametrů a přiřadí adresu vytvořeného objektu ukazateli pstock. V tomto případě objekt nemá název, ale můžete s ním pracovat pomocí ukazatele. Další diskusi týkající se použití ukazatelů na objekty odložíme až na kapitolu 10.

Konstruktory jsou používány odlišně od ostatních metod třídy. Objekt běžně používáte k vyvolání metody:

```
stock1.show(); // objekt stock1 vyvolá metodu show()
```

Není však možné použít objekt k vyvolání konstruktoru, protože dokud konstruktor nedokončí svou práci, žádný objekt neexistuje. Spíše než k vyvolávání pomocí objektu je konstruktor určen k jeho vytváření.

Implicitní konstruktor

Implicitní konstruktor se používá k inicializaci položek třídy v případě, že nepoužijete žádnou explicitní inicializaci. To znamená, že se jedná o konstruktor používaný v případech podobných tomu následujícímu:

```
Stock stock1; // použije se implicitní konstruktor
```

Ale výpis 9.1 již tuto konstrukci používal! Důvodem, proč tento příkaz funguje, je pravidlo, že C++ automaticky vytvoří implicitní konstruktor v případě, že jste žádný jiný konstruktor sami nedefinovali. Jedná se o bezparametrický konstruktor, který nedělá vůbec nic. V případě třídy Stock by vypadal takto:

```
Stock::Stock() { }
```

Tímto získáme objekt stock1 bez inicializovaných datových položek, přesně jako příkaz

```
int x;
```

vytvoří proměnnou x bez počáteční hodnoty.

Zvláštností implicitního konstruktoru je skutečnost, že je vytvořen kompilátorem pouze v případě, jestliže sami žádný konstruktor nevytvoříte. Poté, co definujete pro třídu nějaký konstruktor, přechází zodpovědnost za vytvoření implicitního konstruktoru z kompilátoru na vás. Jestliže vytvoříte nějaký konstruktor, jako například Stock(const char * co, int n, double pr), a nedodáte vlastní verzi bezparametrického konstruktoru, pak následující deklarace

```
Stock stock1; // s existujícím konstruktorem nelze
```

způsobí chybu. Důvodem takového chování je skutečnost, že zřejmě chcete zabránit vytváření neinicializovaných objektů. Na druhé straně budete někdy chtít mít možnost vytvářet objekty bez explicitní inicializace. V takovém případě musíte nadefinovat vlastní bezparametrický konstruktor. To je konstruktor, který nepoužívá žádné parametry. Můžete ho definovat dvěma způsoby. Jedním je uvedení implicitních hodnot u všech parametrů některého konstruktoru:

```
Stock(const char * co = "Error", int n = 0, double pr = 0.0);
```

Druhým způsobem je využití přetěžování funkcí a definování dalšího konstruktoru, který nebude mít žádné parametry:

```
Stock();
```

(V prvních verzích C++ bylo možné bezparametrický konstruktor vytvořit pouze druhým způsobem.)

Ve skutečnosti byste měli objekty inicializovat vždy, abyste se ujistili, že datové položky začínají s nějakými smysluplnými hodnotami. Bezparametrický konstruktor tedy většinou obsahuje inicializaci všech datových položek třídy. Zde je například ukázka toho, jak by mohl vypadat takový konstruktor pro třídu Stock:

```
Stock::Stock()
{
        strcpy(company, "no name");
        shares = 0;
        share_val = 0.0;
        total_val = 0.0;
}
```

Tip

Když navrhujete třídu, měli byste vždy dodat bezparametrický konstruktor, který inicializuje všechny datové položky třídy.

Po použití některé z uvedených metod (žádné parametry nebo implicitní hodnoty všech parametrů) k vytvoření bezparametrického konstruktoru můžete deklarovat proměnné třídy bez explicitní inicializace:

```
Stock first;                     // implicitně volá standardní konstruktor
Stock first = Stock();           // explicitní volání
Stock *prelief = new Stoc k;     // implicitní volání
```

Nenechte se však zmást formou volání explicitního konstruktoru:

```
Stock first("Concrete Conglomerate");   // volá konstruktor
Stock second();                          // deklaruje funkci
Stock third;                             // volá implicitní konstruktor
```

První deklarace volá explicitní konstruktor, to znamená ten, který přijímá parametry. Druhý řádek deklaruje funkci `second()` vracející objekt třídy `Stock`. Jestliže voláte implicitní konstruktor, nepoužívejte závorky.

Destruktory

Když použijete k vytvoření objektu konstruktor, přebírá program zodpovědnost za sledování tohoto objektu až do doby, kdy platnost tohoto objektu skončí. V tuto chvíli program automaticky zavolá speciální členskou funkci nesoucí hrozivý název *destruktor*. Destruktor by měl uklidit všechny zbytky, takže vlastně slouží konstruktivně. Například jestliže váš konstruktor používá operátor `new` k alokaci paměti, destruktor by měl použít operátor `delete` k jejímu uvolnění. Konstruktor třídy `Stock` nic takového jako je operátor `new` nepoužívá, takže by ani nebylo potřeba destruktor vytvářet. Ale je dobré přesto destruktor vytvořit, kdyby byl potřeba v příštích verzích třídy.

Podobně jako konstruktor i destruktor má speciální název: název třídy, kterému předchází znak vlnovka (~). Konstruktor třídy `Stock` se tedy jmenuje `~Stock()`. Stejně jako konstruktor ani destruktor nevrací hodnotu a nemá žádný deklarovaný typ. Narozdíl od konstruktoru destruktor nesmí mít žádné parametry. Destruktor třídy `Stock` musí mít tedy tento prototyp:

```
~Stock();
```

Protože destruktor třídy Stock nemá žádné povinnosti, můžeme ho nadefinovat jako prázdnou funkci:

```
Stock::~Stock()
{
}
```

Abychom však viděli, kdy je destruktor volán, zapíšeme ho následovně:

```
Stock::~Stock()     // destruktor třídy
{
      cout << "Nashledanou, " << company <<   "!\n";
}
```

Kdy by měl být destruktor volán? Toto je na rozhodnutí kompilátoru, program nesmí volat destruktor explicitně. Jestliže vytvoříte statický objekt, jeho destruktor bude zavolán automaticky při ukončení programu. Pokud vytvoříte automatický objekt paměťové třídy, což jsme dělali, bude její destruktor zavolán při ukončení bloku programu, ve kterém je objekt definován. Jestliže je objekt vytvořen pomocí operátoru `new`, je uložen na haldě neboli ve volné paměti a jeho destruktor bude zavolán při jeho zrušení pomocí operátoru `delete`. A konečně program může vytvářet dočasné objekty pro provedení nějaké operace. V takovém případě program automaticky zavolá destruktor po skončení používání objektu.

Protože destruktor je volán automaticky při zrušení objektu, musí existovat. Jestliže sami žádný nenadefinujete, vytvoří kompilátor implicitní destruktor, který nic neprovádí.

Vylepšení třídy Stock

Dalším krokem je začlenění konstruktorů a destruktorů do třídy a definic metod. Tentokrát se budeme řídit praxí jazyka C++ a rozdělíme program do samostatných souborů. Deklaraci třídy umístíme do hlavičkového souboru se jménem `stock1.h`. (Jak napovídá název, počítáme s budoucími změnami.) Definice metod třídy budou uloženy v souboru

stock1.cpp. Hlavičkový soubor, obsahující deklaraci třídy, a zdrojový soubor, obsahující definice metod, by měly mít stejný základní název, abyste věděli, které soubory patří k sobě. Použití různých souborů pro deklaraci třídy a pro členské funkce odděluje abstraktní definici rozhraní (deklaraci třídy) od implementačních detailů (definic členských funkcí). Mohli byste například distribuovat deklaraci třídy jako textový hlavičkový soubor, zatímco definice funkcí jako zkompilovaný program. Nakonec uložíme program používající tyto zdroje do třetího souboru s názvem usestock1.cpp.

Hlavičkový soubor

Výpis 9.4 ukazuje obsah hlavičkového souboru. K původní deklaraci třídy jsou přidány prototypy konstruktoru a destruktoru. Třída se také zbavila funkce acquire(), která není již potřeba, protože máme konstruktory.

Výpis 9.4. stock1.h

```
// stock1.h
  #ifndef _STOCK1_H_
  #define _STOCK1_H_
  class Stock
  {
  private:
       char company[30];
       int shares;
       double share_val;
       double total_val;
       void set_tot() { total_val = shares * share_val; }
  public:
       Stock();         // bezparametrický konstruktor
       Stock(const char * co, int n = 0, double pr = 0.0);
       ~Stock();        // upovídaný destruktor
       void buy(int num, double price);
       void sell(int num, double price);
       void update(double price);
       void show();
  };
  #endif
```

Správa hlavičkových souborů

Do souboru byste měli daný hlavičkový soubor vkládat pouze jednou. To může vypadat jako jednoduchá záležitost, ale je možné vložit hlavičkový soubor několikrát, aniž byste o tom věděli. Například můžete vložit hlavičkový soubor, který vkládá další hlavičkový soubor. V C++ existuje standardní postup pro zamezení několikanásobného vložení hlavičkového souboru. Je založen na direktivě preprocesoru #ifndef (zkratka pro *if not defined*). Segment následujícího kódu

```
#ifndef _STOCK1_H_
...
#endif
```

znamená, že příkazy mezi direktivami #ifndef a #endif budou vloženy pouze tehdy, pokud již nebyl definován symbol `_STOCK1_H_` pomocí direktivy preprocesoru `#define`.

Normálně použijete příkaz #define k vytváření symbolických konstant, jako třeba zde:

```
#define MAXIMUM 4096
```

Použití direktivy `#define` s názvem však k definici tohoto jména stačí, jako v tomto případě:

```
#define _STOCK1_H_
```

Technika použitá ve výpisu 9.4. spočívá v obalení obsahu souboru direktivou `#ifndef`:

```
#ifndef _STOCK1_H_
#define _STOCK1_H_
// místo pro obsah souboru
#endif
```

Když kompilátor narazí na tento soubor poprvé, neměl by být symbol `_STOCK1_H_` definován. (Vybrali jsme název podle vloženého souboru a proložili několika podtržítky, aby se snížila pravděpodobnost, že vytvořený název bude definován někde jinde.) V takovém případě kompilátor vezme v úvahu příkazy uzavřené mezi direktivami `#ifndef` a `#endif`, což chceme. Při zpracování kompilátor přečte řádek definující název `_STOCK1_H_`. Jestliže pak ve stejném souboru opět narazí na vložení souboru stock1.h, všimne si, že název `_STOCK1_H_` je již definován a skočí na řádek následující za direktivou `#endif`. Všimněte si, že tato metoda nezamezí opětovnému vložení souboru. Namísto toho způsobí ignorování všech vložení s výjimkou toho prvního. Tuto metodu používá většina standardních hlavičkových souborů jazyků C a C++.

Implementační soubor

Výpis 9.5 obsahuje implementaci metod. Obsahuje hlavičkový soubor `stock1.h` za účelem získání deklarace třídy. (Vzpomeňte si, že uzavření jména souboru do uvozovek namísto znaků < a > způsobí, že kompilátor hledá soubor na stejném místě, kde jsou uloženy zdrojové soubory.) Výpis také obsahuje systémové soubory `iostream` a `cstring`, protože tyto metody používají třídy `cin` a `cout` a metodu `strncpy()`. Před tyto metody jsou přidány definice metod konstruktoru a destruktoru.

Výpis 9.5. stock1.cpp

```
// stock1.cpp               // metody třídy Stock
  #include <iostream>
  #include <cstdlib>          // (nebo stdlib.h) pro exit()
  #include <cstring>          // (nebo string.h) pro strncpy()
  using namespace std;
  #include "stock1.h"
  // konstruktory
  Stock::Stock()              // bezparametrický konstruktor
  {
      strcpy(company, "spolecnost");
      shares = 0;
```

```
    share_val = 0.0;
    total_val = 0.0;
}
Stock::Stock(const char * co, int n, double pr)
{
    strncpy(company, co, 29);
    company[29] = '\0';
    shares = n;
    share_val = pr;
    set_tot();
}
// destruktor
Stock::~Stock()            // upovídaný destruktor třídy
{
    cout << "Nashledanou, " << company <<   "!\n";
}
//ostatní metody
void Stock::buy(int num, double price)
{
    shares += num;
    share_val = price;
    set_tot();
}
void Stock::sell(int num, double price)
{
    if (num > shares)
    {
        cerr << "Nemuzete prodat vic nez mate!\n";
        exit(1);
    }
    shares -= num;
    share_val = price;
    set_tot();
}
void Stock::update(double price)
{
    share_val = price;
    set_tot();
}
void Stock::show()
{
    cout << "Spolecnost: " << company
         << " Pocet akcii: " << shares << '\n'
         << " Cena akcie: Kc" << share_val
         << " Celkova cena: Kc" << total_val << '\n';
}
```

Kompatibilita:

Namísto jmen `cstdlib` a `cstring` můžete použít názvy `stdlib.h` a `string.h`.

Klientský soubor

Výpis 9.6 představuje krátký program pro otestování nových metod. Podobně jako soubor `stock1.cpp` obsahuje také soubor `stock1.h` za účelem získání deklarace třídy. Program demonstruje použití konstruktorů a destruktorů. Také používá stejné formátovací příkazy uvedené již ve výpisu 9.3. Ke zkompilování celého programu použijte techniku pro vícesouborové programy popsanou v kapitolách 1 a 8.

Výpis 9.6. usestock1.cpp

```
// usestok1.cpp - použití třídy Stock
#include <iostream>
using namespace std;
#include "stock1.h"
int main()
{
// vytvoření nových objektů pomocí konstruktorů
    Stock stock1("NanoSmart", 12, 20.0);             // syntaxe 1
    Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // syntaxe 2
    cout.precision(2);                               // formát #.##
    cout.setf(ios::fixed, ios::floatfield);          // formát #.##
    cout.setf(ios::showpoint);                       // formát #.##
    stock1.show();
    stock2.show();
    stock2 = stock1;                                 // přiřazení objektu
// použití konstruktoru k novému nastavení objektu
    stock1 = Stock("Nifty Foods", 10, 50.0);         // dočasný objekt
    cout << "Po zmenach na burze:\n";
    stock1.show();
    stock2.show();
    return 0;
}
```

Kompatibilita:

Namísto třídy `ios::base` byste mohli použít starší třídu `ios::`.

Zde je výstup programu:

```
Spolecnost: NanoSmart Pocet akcii: 12
  Cena akcie: $20.00 Celkova cena: $240.00
Spolecnost: Boffo Objects Pocet akcii: 2
  Cena akcie: $2.00 Celkova cena: $4.00
Nashledanou, Nifty Foods!
 Po zmenach na burze:
Spolecnost: Nifty Foods Pocet akcii: 10
  Cena akcie: $50.00 Celkova cena: $500.00
Spolecnost: NanoSmart Pocet akcii: 12
  Cena akcie: $20.00 Celkova cena: $240.00
Nashledanou, NanoSmart!
Nashledanou, Nifty Foods!
```

Poznámky k programu

Příkaz

```
Stock stock1("NanoSmart", 12, 20.0);
```

vytvoří objekt třídy `Stock` nazvaný `stock1` a inicializuje jeho datové položky na uvedené hodnoty. Příkaz

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
```

používá druhou variantu vytvoření a inicializace objektu pojmenovaného `stock2`.

Konstruktor můžete použít nejen pro inicializaci nově vytvořeného objektu. Program například obsahuje ve funkci main() tento příkaz:

```
stock1 = Stock("Nifty Foods", 10, 50.0);
```

Objekt `stock1` již existuje. Namísto inicializace objektu `stock1` tedy tento příkaz přiřadí položkám objektu nové hodnoty. Konstruktor nejdříve vytvoří nový dočasný objekt a potom jeho obsah zkopíruje do objektu `stock1`.

Příkaz

```
stock2 = stock1;    // přiřazení objektu
```

ukazuje možnost přiřazení jednoho objektu druhému, jestliže jsou stejného typu. Stejně jako u přiřazení struktury i přiřazení objektu třídy implicitně kopíruje datové položky jednoho objektu do druhého. V tomto případě je původní obsah objektu `stock2` přepsán.

Pamatujte:

Při přiřazení objektů stejné třídy jazyk C++ implicitně zkopíruje obsah datových položek zdrojového objektu do odpovídajících položek cílového objektu.

Všimněte si, že výstup programu obsahuje text `Nashledanou, Nifty Foods!` ještě před vypsáním nového obsahu objektu `stock1`. Později na konci program vypisuje zprávy `Nashledanou, NanoSmart!` a `Nashledanou, Nifty Foods!`. Kde se tato rozloučení vzala? Vzpomeňte si, že destruktor obsahuje příkaz pro výstup, abyste viděli, kdy je volán. (Toto je výuková pomůcka, nikoli běžná součást návrhu!) Poslední dvě rozloučení se objeví na konci funkce `main()`, kdy jsou rušeny dva lokální objekty `stock1` a `stock2`. Protože tyto automatické proměnné jsou uloženy v zásobníku, je nejdříve zrušen naposledy vytvořený objekt a první vytvořený objekt je smazán jako poslední. (Všimněte si, že hlášení „`NanoSmart`" obsahoval původně objekt stock1, ale později bylo přesunuto do objektu `stock2`.)

A co první loučení Nifty Foods? Vzpomeňte si, že když program pomocí konstruktoru přiřazuje objektu `stock1` hodnoty Nifty Foods, vytvoří nejdříve dočasný bezejmenný objekt s uvedenými hodnotami. Potom se tyto hodnoty zkopírují do objektu stock1 a nakonec, až skončí platnost tohoto dočasného objektu, zavolá na něj program funkci destruktoru. První hlášení Nifty Foods patří dočasnému objektu, zatímco druhé objektu `stock1`. Jazyk C++ neuvádí, kdy je přechodný objekt zlikvidován. Tato implementace (Microsoft Visual C++ 5.0) jej zruší ihned, jakmile není potřeba, zatímco Turbo C++ 2.0 zruší dočasný objekt až na konci funkce.

Tato malá epizoda ukazuje, že mezi následujícími dvěma příkazy je zásadní rozdíl:

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
stock1 = Stock("Nifty Foods", 10, 50.0); // dočasný objekt
```

První příkaz je inicializace – vytvoří objekt s uvedenými hodnotami. Druhý příkaz je přiřazení. Vytvoří dočasný objekt, který poté zkopíruje do již existujícího objektu. Toto je samozřejmě méně efektivní než inicializace. (Kompilátor však může implementovat výše uvedenou formu inicializace objektu `stock2` jako vytvoření pomocného objektu a zkopírování jeho obsahu do objektu `stock2`.)

Tip:

Jestliže můžete nastavit hodnoty objektu jak inicializací tak i přiřazením, volte raději inicializaci. Většinou je efektivnější.

Výstup metody `show()` na konci programu demonstruje, že fungovalo jak přiřazení objektu, tak i jeho opětovné nastavení a přiřazení pomocí konstruktoru.

Konstantní členské funkce

Uvažujte následující kousek kódu:

```
const Stock land = Stock("Kludgehorn Properties");
land.show();
```

Při současné verzi jazyka C++ by se měl kompilátor zastavit u druhého řádku. Proč? Protože kód metody `show()` negarantuje, že nezmění stav volaného objektu. Ten je definován jako konstanta a neměl by tedy být měněn. Tento druh problému jste již řešili dříve, když jste parametr funkce deklarovali jako `konstantní` referenci nebo ukazatel na `konstantu`. Zde však narážíme na syntaktický problém: metoda `show()` žádné parametry nemá. Místo toho získá používaný objekt implicitně voláním metody. Je potřeba nějaká nová syntaxe, která zaručí, že funkce nezmění volající objekt. Jazyk C++ řeší tento problém uvedením klíčového slova const za závorkami funkce. To znamená, že deklarace funkce `show()` by měla vypadat takto:

```
void show() const;                 // slibuje, že volající objekt nezmění
```

Podobně i začátek definice funkce by měl vypadat takto:

```
void stock::show() const           // slibuje, že volající objekt nezmění
```

Funkce třídy deklarované a definované tímto způsobem se nazývají metody `konstantního` objektu. Stejně jako byste měli pokud možno používat `konstantní` reference a ukazatele na místě formálních parametrů funkcí, měli byste také metody třídy deklarovat jako `konstantní`, pokud nemění volající objekt. Od této chvíle se budeme držet této zásady.

Opakování konstruktorů a destruktorů

Nyní, když jsme prošli několik příkladů použití konstruktorů a destruktorů, si možná přejete trochu odpočinku a možnost vstřebat uvedené informace. Abychom vám pomohli, nabízíme zde přehled o těchto metodách.

Konstruktor je speciální členská funkce třídy, která je volána vždy při vytvoření objektu této třídy. Má stejný název jako daná třída, ale díky zázraku přetěžování funkcí je možné mít více než jeden konstruktor se stejným názvem za předpokladu, že každý má svou vlastní signaturu nebo seznam parametrů. Konstruktor nemá deklarován žádný typ. Obvykle se používá k inicializaci položek objektu třídy. Inicializace by se měla shodovat se seznamem parametrů konstruktoru. Předpokládejme například, že konstruktor třídy `Bozo` má následující prototyp:

```
Bozo(char * fname, char * lname);  // prototyp konstruktoru
```

Potom byste jej použili k inicializaci nových objektů takto:

```
Bozo bozetta = bozo("Bozetta", "Biggens");          // primární zápis
Bozo fufu("Fufu", "O'Dweeb");                       // zkrácený zápis
Bozo *pc = new Bozo("Popo", "Le Peu");              // dynamický objekt
```

Jestliže má konstruktor pouze jeden parametr, je vyvolán, jakmile inicializujete objekt na hodnotu shodného typu, jako je typ parametru konstruktoru. Předpokládejme například, že máme konstruktor s tímto prototypem:

```
Bozo(int age);
```

V tom případě můžete k inicializaci objektu použít jakýkoli z následujících zápisů:

```
Bozo dribble = bozo(44);  // primární zápis
Bozo roon(66);            // sekundární zápis
Bozo tubby = 32;          // speciální zápis konstruktoru s jedním
                             parametrem
```

Třetí příklad je pro vás vlastně nový – není to opakování, nyní však je vhodná chvíle povědět vám o něm. V kapitole 10 je zmíněn způsob, jak tomuto způsobu zamezit.

Pamatujte

Konstruktor s jedním parametrem umožňuje při inicializaci nového objektu použít syntaxi přiřazení:

```
Názevtřídy objekt = hodnota;
```

Implicitní konstruktor nemá žádné parametry a použije se, jestliže vytvoříte objekt a neinicializujete ho explicitně. Když neuvedete žádný konstruktor, kompilátor za vás vytvoří implicitní konstruktor. Jinak musíte dodat svůj vlastní bezparametrický konstruktor. Nemusí mít žádné parametry nebo může mít u všech svých parametrů uvedeny implicitní hodnoty:

```
Bozo();            // prototyp implicitního konstruktoru
Bistro(const char * s = "Chez Zero");// implicitní konstruktor třídy Bistro
```

Pro neinicializované objekty použije program implicitní konstruktor:

```
Bozo bubi;          // implicitní použití
Bozo *pb;           // implicitní použití
```

Podobně jako program při vytváření objektu spustí konstruktor, při jeho rušení spustí destruktor. Třída může mít pouze jeden destruktor. Nemá žádný návratový typ, dokonce ani typ void, nemá žádné parametry a jeho název začíná znakem vlnovka. Destruktor třídy Bozo má například tento prototyp:

```
~Bozo();  // destruktor třídy
```

Destruktory tříd jsou potřeba v případě, že konstruktory používají operátor new.

Poznáváme objekty: ukazatel this

Třída Stock ještě stále není dokončena. Až potud každá členská funkce třídy pracovala pouze s jedním objektem, což byl objekt, který ji vyvolal. Ovšem někdy může nastat případ, že metoda potřebuje pracovat se dvěma objekty, a tehdy nastupuje zvláštní ukazatel jazyka C++ nazývaný this. Pojďme se podívat, z čeho může taková potřeba vyvstat.

Ačkoli třída Stock zobrazuje nějaká data, chybí jí analytická schopnost. Například při pohledu na výstup metody show() můžete říci, který z vašich podílů má nejvyšší hodnotu, ale program toto říci nemůže, protože nemá přímý přístup k položce total_val. Nejpřímější cestou, jak program informovat o uložených datech, je zavedení metod vracejících hodnoty. Běžným způsobem je použití vložené funkce:

```
class Stock
  {
  private:
      ...
      double total_val;
      ...
  public:
      double total() const { return total_val; }
      ...
  };
```

Pokud jde o přímý přístup programu, má taková definice ten účinek, že položka total_val je určena pouze pro čtení.

Přidáním této funkce do deklarace třídy umožníte programu zjišťovat hodnoty jednotlivých podílů a určit tak ten s největší hodnotou. Zvolíme však odlišný přístup, hlavně kvůli tomu, abyste se naučili používat ukazatel this. Tento způsob spočívá v definici členské funkce, která se podívá na oba objekty třídy Stock a vrátí referenci na objekt s větší hodnotou. Při pokusu o implementaci tohoto přístupu vyvstanou některé zajímavé otázky, a na ty se nyní podíváme.

Za prvé, jakým způsobem zavedete členskou funkci, která bude oba objekty porovnávat? Předpokládejme, že se rozhodnete metodu pojmenovat topval(). Potom volání stock1.topval() má přístup k datům objektu stock1, zatímco volání stock2.topval() přistupuje k datům v objektu stock2. Jestliže chcete, aby metoda oba objekty porovnala, mu-

síte druhý objekt dodat jako parametr. Z důvodu efektivity budete parametr předávat jako referenci. To znamená, že metoda bude používat parametr typu `const Stock &`.

Za druhé, jakým způsobem sdělíte výsledek metody zpět volajícímu programu? Nejpřímější cestou je, aby metoda vrátila referenci na objekt s větší celkovou hodnotou. Porovnávací metoda by tedy měla mít tento prototyp:

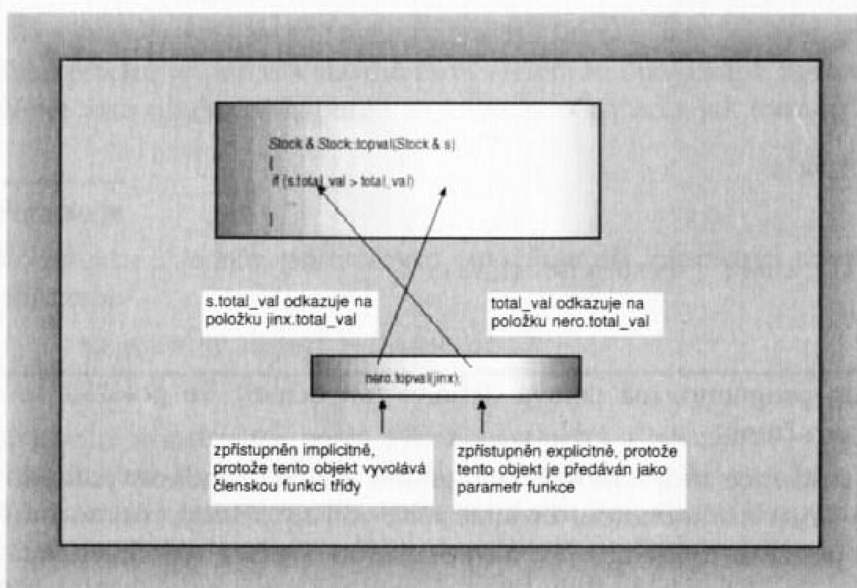
```
const Stock & topval(const Stock & s) const;
```

Tato funkce přistupuje k jednomu objektu implicitně a k druhému explicitně a vrací referenci na jeden z těchto dvou objektů. Modifikátor `const` v závorkách stanoví, že funkce nebude měnit objekt s explicitním přístupem, zatímco modifikátor `const` za závorkami udává, že funkce nebude měnit objekt s implicitním přístupem. Protože funkce vrací referenci na jeden ze dvou `konstantních` objektů, musí být konstantní reference také návratovou hodnotou.

Předpokládejme, že chcete porovnat objekty `stock1` a `stock2` třídy `Stock` a přiřadit ten s větší celkovou hodnotou objektu `top`. Můžete použít jeden ze dvou následujících příkazů:

```
top = stock1.topval(stock2);
top = stock2.topval(stock1);
```

První zápis přistupuje k objektu `stock1` implicitně a k objektu `stock2` explicitně, zatímco druhý tak činí obráceně. Podívejte se na obrázek 9.3. V obou případech metoda porovná dva objekty a vrátí referenci na objekt s vyšší celkovou hodnotou.



Obrázek 9.3. Získání přístupu ke dvěma objektům pomocí členské funkce

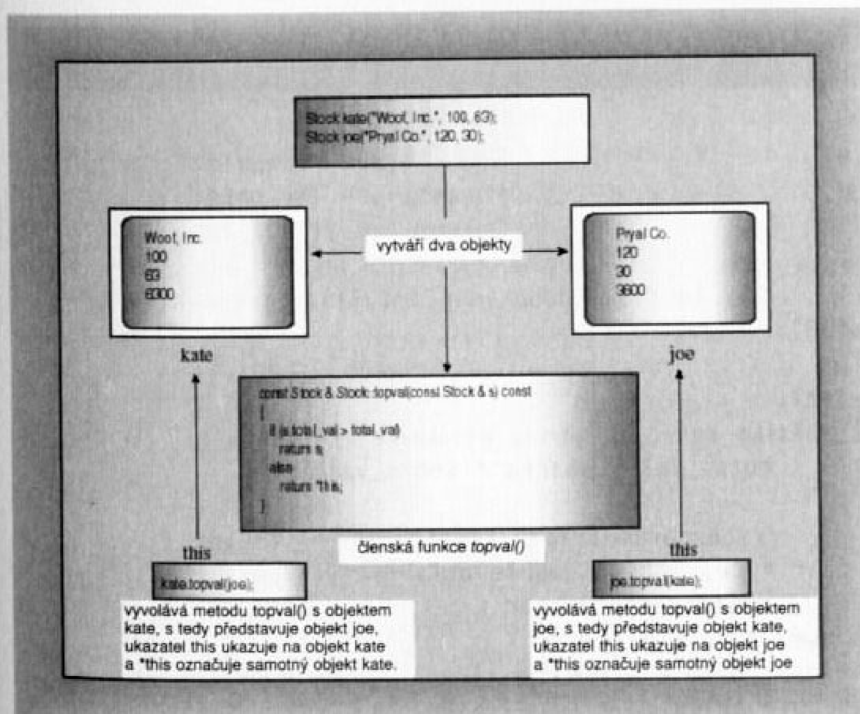
V praxi je taková syntaxe trochu zavádějící. Mnohem jasnější by bylo, kdybyste mohli nějakým způsobem využít k porovnání obou objektů relační operátor >. Toho můžete dosáhnout pomocí přetěžování operátorů, které bude probráno v kapitole 10.

Ještě nám zbývá implementace metody `topval()`. Tady vzniká malý problém. Zde je část implementace, která ho osvětlí:

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;          // objekt jako parametr
    else
        return ?????;      // volající objekt
}
```

Zde proměnná `s.total_val` představuje celkovou hodnotu objektu předaného jako parametr a proměnná `total_val` je celkovou hodnotou objektu, kterému je zpráva zaslána. Jestliže je hodnota `s.total_val` větší než `total_val`, vrátí funkce objekt s. Jinak bude vrácen objekt, který metodu vyvolal. (V terminologii OOP se mluví o objektu, kterému byla zaslána zpráva `topval`.) Problémem je, jak pojmenovat tento objekt? Jestliže voláte funkci `stock1.topval(stock2)`, představuje s referenci objektu `stock2` (to znamená alias objektu stock2), alias objektu `stock1` však neexistuje.

Tento problém řeší jazyk C++ speciálním ukazatelem pojmenovaným `this`. Ukazatel `this` ukazuje na objekt, který vyvolal nějakou členskou funkci. (V podstatě je metodě předán jako skrytý parametr.) Volání funkce `stock1.topval(stock2)` tedy nastaví ukazatel `this` na adresu objektu stock1 a zpřístupní tento ukazatel metodě `topval()`. Podobně volání funkce `stock2.topval(stock1)` nastaví ukazatel `this` na adresu objektu `stock2`. Obecně lze říci, že všechny metody třídy mají ukazatel this nastaven na adresu objektu, který danou metodu vyvolal. Proměnná `top_val` vlastně představuje v metodě `top_val` jen zkrácený zápis příkazu `this->total_val`. (Vzpomeňte si z kapitoly 4, že operátor -> používáte pro přístup ke složkám struktury pomocí ukazatele. Totéž platí o složkách třídy.) (Viz obrázek 9.4.)



Obrázek 9.4. Ukazatel this ukazuje na volající objekt

Ukazatel this

Každá členská funkce včetně konstruktorů a destruktorů má ukazatel this. Speciální vlastností tohoto ukazatele this je, že ukazuje na volající objekt. Jestliže se metoda potřebuje odkázat na volající objekt jako celek, může použít výraz *this . Použijete-li za závorkami s parametry kvalifikátor const , určíte, že je ukazatel this konstantní. V tom případě ho nebudete moci použít ke změně hodnoty objektu.

Vy však nechcete vrátit ukazatel this, protože ten obsahuje adresu objektu. Chcete vrátit objekt samotný, a to symbolizuje výraz *this. (Vzpomeňte, že použitím operátoru dereference * na ukazatel získáte hodnotu, na kterou tento ukazatel ukazuje.) Nyní můžete dokončit definici metody pomocí výrazu *this jako aliasu volajícího objektu.

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;           // objekt jako parametr
    else
        return *this;       // volajici objekt
}
```

Skutečnost, že návratový typ je reference, znamená, že vrácený objekt je samotný volající objekt, a nikoli jeho kopie předaná návratovým mechanismem. Výpis 9.7 ukazuje nový hlavičkový soubor.

Výpis 9.7. stock2.h

```
// stock2.h
#ifndef _STOCK2_H_
#define _STOCK2_H_
class Stock
{
private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();            // bezparametrický konstruktor
    Stock(const char * co, int n, double pr);
    ~Stock() {}    // prázdný destruktor
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
};
#endif
```

Výpis 9.8 představuje soubor s upravenými metodami třídy. Obsahuje také novou metodu `topval()`. Teď, když jste již viděli, jak metoda destruktoru funguje, nahradíme ji tichou verzí.

Výpis 9.8. stock2.cpp

```
// stock2.cpp      // metody třídy Stock
#include <iostream>
using namespace std;
#include <cstdlib>     // pro exit()
#include <cstring> // pro strcpy()
#include "stock2.h"
// konstruktory
Stock::Stock()
{
    strcpy(company, "spolecnost");
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}
Stock::Stock(const char * co, int n, double pr)
{
    strcpy(company, co);
    shares = n;
    share_val = pr;
    set_tot();
}
void Stock::buy(int num, double price)
{
    shares += num;
    share_val = price;
    set_tot();
}
void Stock::sell(int num, double price)
{
    if (num > shares)
    {
        cerr << "Nemuzete prodat vic nez mate!\n";
        exit(1);
    }
    shares -= num;
    share_val = price;
    set_tot();
}
void Stock::update(double price)
{
    share_val = price;
    set_tot();
}
void Stock::show() const
```

```
{
    cout << "Spolecnost: " << company
         << " Pocet akcii: " << shares << '\n'
         << " Cena akcie: $" << share_val
         << " Celkova cena: $" << total_val << '\n';
}
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;
    else
        return *this;
}
```

Samozřejmě chceme vidět, zda ukazatel `this` funguje. Přirozeným místem pro použití této nové metody je program používající pole objektů, což nás přivádí k dalšímu tématu.

Pole objektů

Často potřebujete vytvořit několik objektů stejné třídy. Tak tomu bylo dosud v příkladech s třídou `Stock`. Můžete vytvořit několik samostatných proměnných objektu jako ve výše uvedených příkladech, ale rozumnější je vytvořit pole objektů. To může znít jako velký krok do neznáma, ve skutečnosti však pole objektů deklarujete stejným způsobem jako pole kteréhokoli standardního typu:

```
Stock mystuff[4]; // vytvoří pole 4 objektů třídy Stock
```

Vzpomeňte si, že program volá implicitní konstruktor třídy vždy, když vytváří objekty neinicializované explicitně. Tato deklarace vyžaduje, aby třída buď nedefinovala explicitně žádný konstruktor a v tom případě se použije prázdný implicitní konstruktor, nebo aby byl jako v tomto případě definován explicitní konstruktor.

Každý prvek pole – `mystuff[0]`, `mystuff[1]` atd. – je objektem třídy `Stock`, a proto ho mohou použít metody této třídy:

```
mystuff[0].update();        // aplikuje metodu update() na první prvek
mystuff[3].show();          // aplikuje metodu show() na čtvrtý prvek
Stock tops = mystuff[2].topval(mystuff[1]);
                                // porovná třetí a druhý prvek
```

Prvky pole můžete také inicializovat pomocí konstruktoru. V takovém případě musíte konstruktor volat pro každý jednotlivý prvek:

```
const int STKS = 4;
Stock stocks[STKS] = {
    Stock("NanoSmart", 12.5, 20),
    Stock("Boffo Objects", 200, 2.0),
    Stock("Monolithic Obelisks", 130, 3.25),
    Stock("Fleep Enterprises", 60, 6.5)
    };
```

V tomto kódu se pro inicializaci pole používá standardní zápis: čárkami oddělený seznam hodnot uzavřený ve složených závorkách. V tomto případě představuje volání metody konstruktoru každou jednotlivou hodnotu. Jestliže má třída více než jeden konstruktor, můžete pro různé prvky použít různé konstruktory:

```
const int STKS = 10;
Stock stocks[STKS] = {
    Stock("NanoSmart", 12.5, 20),
    Stock(),
    Stock("Monolithic Obelisks", 130, 3.25),
};
```

V tomto případě jsou prvky `stocks[0]` a `stocks[2]` inicializovány konstruktorem `Stock(const char * co, int n, double pr)`, zatímco prvek `stocks[1]` konstruktorem `Stock()`. Protože taková deklarace inicializuje pouze část pole, je zbývajících sedm prvků inicializováno implicitním konstruktorem.

Schéma inicializace pole objektů vytváří prvky pole nejdříve pomocí implicitního konstruktoru. Potom konstruktory ve složených závorkách vytvoří dočasné objekty, jejichž obsah je zkopírován do vektoru. Jestliže tedy chcete vytvořit pole objektů třídy, musí mít daná třída implicitní konstruktor.

Upozornění

Jestliže chcete vytvořit pole objektů třídy, musí mít daná třída implicitní konstruktor.

Výpis 9.9. ukazuje použití těchto zásad na krátkém programu, který inicializuje pole o čtyřech prvcích, zobrazuje jejich obsah a pomocí testu najde prvek s největší hodnotou. Protože metoda `topval()` porovnává současně pouze dva objekty, prohledává program celé pole pomocí smyčky `for`. Použijte hlavičkový soubor z výpisu 9.7 a metody z výpisu 9.8.

Výpis 9.9. usestock2.cpp

```
// usestok2.cpp - použití třídy Stock
#include <iostream>
using namespace std;
#include "stock2.h"
const int STKS = 4;
int main()
{
// vytvoří pole inicializovaných objektů
    Stock stocks[STKS] = {
        Stock("NanoSmart", 12, 20.0),
        Stock("Boffo Objects", 200, 2.0),
        Stock("Monolithic Obelisks", 130, 3.25),
        Stock("Fleep Enterprises", 60, 6.5)
        };
    cout.precision(2);                          // formát #.##
    cout.setf(ios::fixed, ios::floatfield); // formát #.##
```

```
    cout.setf(ios::showpoint);                    // formát #.##
    cout << "Vlastnictvi akcii:\n";
    int st;
    for (st = 0; st < STKS; st++)
        stocks[st].show();
    Stock top = stocks[0];
    for (st = 1; st < STKS; st++)
        top = top.topval(stocks[st]);
    cout << "\nNejcennejsi akcie:\n";
    top.show();
    return 0;
}
```

Kompatibilita:

Namísto hlavičkových souborů `cstdlib` a `cstring` můžete použít soubory `stdlib.h` a `string.h`. Také byste mohli místo třídy `ios::base` použít starší třídu `ios::`.

Zde je výstup programu:

```
Vlastnictvi akcii:
Spolecnost: NanoSmart Pocet akcii: 12
  Cena akcie: $20.00 Celkova cena: $240.00
Spolecnost: Boffo Objects Pocet akcii: 200
  Cena akcie: $2.00 Celkova cena: $400.00
Spolecnost: Monolithic Obelisks Pocet akcii: 130
  Cena akcie: $3.25 Celkova cena: $422.50
Spolecnost: Fleep Enterprises Pocet akcii: 60
  Cena akcie: $6.50 Celkova cena: $390.00
Nejcennejsi akcie:
Spolecnost: Monolithic Obelisks Pocet akcii: 130
  Cena akcie: $3.25 Celkova cena: $422.50
```

Stojí za povšimnutí, že většinu práce zabere návrh třídy. Jakmile je hotov, je samotné psaní programu dosti jednoduché.

Díky znalosti ukazatele `this` snáze pochopíte, jak jazyk C++ funguje. Například překladač `cfront` konvertuje programy napsané v C++ do jazyka C. Aby bylo možné pracovat s definicemi metod, stačí překonvertovat následující metodu napsanou v C++

```
void Stock::show() const
{
    cout << "Spolecnost: " << company
         << " Pocet akcii: " << shares << '\n'
         << " Cena akcie: $" << share_val
         << " Celkova cena: $" << total_val << '\n';
}
```

do následující definice jazyka C:

```
void show(const Stock * this)
{
    cout << "Spolecnost: " << this->company
         << "Pocet akcii: " << this->shares << '\n'
         << "Cena akcie: $" << this->share_val
         << " Celkova cena: $" << this->total_val << '\n';
}
```

To znamená, že překladač překonvertuje kvalifikátor `Stock::` na parametr funkce, kterým je ukazatel na `Stock`, a potom tento ukazatel použije pro přístup k položkám třídy. Podobně kompilátor překonvertuje volání funkce

```
top.show();
```

na tvar

```
show(&top);
```

Zde je ukazateli `this` přiřazena adresa volajícího objektu. (Ve skutečnosti může být vše složitější.)

Rozsah platnosti třídy

Kapitola 8 rozebírá globální proměnné (platí v celém souboru) a lokální proměnné (s platností v bloku). Vzpomeňte si, že globální proměnnou můžete použít kdekoli v souboru, který obsahuje její definici, zatímco lokální proměnnou můžete použít pouze v bloku, který obsahuje její definici. Také funkce mají globální rozsah, ale nikdy nemají rozsah lokální. Třídy v C++ zavádějí nový typ rozsahu: *třídní rozsah*. Ten se týká položek definovaných uvnitř třídy, jako jsou datové položky a členské funkce. Položky, mající rozsah platnosti třídy, jsou viditelné uvnitř této třídy, ale ne zvenčí. Můžete tedy použít stejné názvy položek v různých třídách, aniž by došlo ke střetu. Položka `shares` třídy `Stock` je rozdílná proměnná od proměnné `shares` ve třídě `JobRide`. Třídní rozsah také znamená, že zvenčí nemůžete přistupovat k položkám třídy přímo. To platí i pro veřejné členské funkce. To znamená, že chcete-li vyvolat veřejnou členskou funkci, musíte použít objekt.

```
Stock sleeper("Exclusive Ore", 100, 0.25); // vytvoření objektu
sleeper.show();        // vyvolání členské funkce pomocí objektu show();
// chyba – metodu nelze volat přímo
```

Podobně je potřeba použít při definici členské funkce rozlišovací operátor:

```
void Stock::update(double price)
{
    ...
}
```

Stručně řečeno, uvnitř deklarace třídy nebo definice členské funkce můžete použít jednoduchý název (nekvalifikovaný název), jako když metoda `sell()` volá členskou funkci `set_tot()`. V ostatních případech musíte v závislosti na kontextu použít operátor pro přímý přístup ke složkám objektu (.) nebo operátor nepřímého přístupu ke složkám objektu (->), případně operátor rozlišení (::).

Někdy by bylo pěkné mít symbolické konstanty třídního rozsahu. Deklarace třídy Stock například používala k určení velikosti pole pro proměnnou company třicetiznakový literál. Protože je konstanta pro všechny objekty stejná, bylo by dobré vytvořit jedinou konstantu, sdílenou všemi objekty. Možná byste považovali za řešení následující kód:

```
class Stock
{
private:
    const int Len = 30;  // deklarace konstanty?
    char company[Len];
    ...
```

Ale toto fungovat nebude, protože deklarace třídy objekt popisuje, ale nevytváří. To znamená, že dokud objekt nevytvoříte, neexistuje místo pro uložení hodnoty. Existuje však několik způsobů, jak dosáhnout v podstatě stejného žádaného efektu.

Za prvé, můžete v rámci třídy deklarovat výčet. Výčet uvedený uvnitř třídy má třídní rozsah, takže můžete pomocí výčtů vytvářet symbolické konstanty typu integer s třídním rozsahem. Deklaraci třídy Stock můžete tedy začít takto:

```
class Stock
{
private:
    enum {Len = 30}; // konstanta platná pouze uvnitř třídy
    char company[Len];
    ...
```

Všimněte si, že deklarace výčtu zde nevytvoří datovou položku třídy. To znamená, že ne každý objekt v sobě obsahuje výčet. Proměnná Len je pouze symbolický název, který kompilátor nahradí 30, až na něj narazí v třídním rozsahu kódu.

Protože zde je výčet používán pouze pro vytvoření symbolické konstanty a ne s úmyslem vytvářet proměnné tohoto typu, nemusíte ho pojmenovávat. V mnoha implementacích dělá něco podobného ve své veřejné části třída ios_base. To je zdroj identifikátorů typu ios_base::fixed. Zde je fixed položka výčtu definovaného v třídě ios_base.

Zcela nedávno zavedl jazyk C++ druhý způsob definice konstanty uvnitř třídy – pomocí klíčového slova static:

```
class Stock
{
private:
    static const int Len = 30;  // deklarace konstanty!
    char company[Len];
    ...
```

Tento kód vytvoří konstantu pojmenovanou Len uloženou spolu s ostatními statickými proměnnými a nikoli s objektem. Existuje tedy pouze jedna konstanta Len, sdílená všemi objekty třídy Stock. Kapitola 11, „Třídy a dynamické přidělování paměti", se dívá na statické položky třídy podrobněji. Pomocí této techniky lze deklarovat pouze statické konstanty s celočíselnými a výčtovými hodnotami. Uložit konstantu typu double tímto způsobem nelze.

Abstraktní datový typ

Třída Stock je dosti konkrétní. Často však programátoři definují třídy, reprezentující obecnější koncepty. Třídy jsou například velmi vhodné pro implementaci toho, co informatici nazývají *abstraktní datový typ* nebo zkráceně ADT. Jak název napovídá, ADT popisuje datový typ pouze obecně a nezavádí do jazyka nebo implementace detaily. Uvažujte například zásobník. Zásobník je způsob ukládání dat, kdy data jsou přidávána na vrchol zásobníku nebo mazána pouze z vrcholu zásobníku. Programy v C++ například pomocí zásobníku řídí automatické proměnné. Když se vytvoří nové automatické proměnné, jsou přidány na vrchol zásobníku. Jakmile jejich platnost skončí, jsou ze zásobníku odstraněny.

Popišme si vlastnosti zásobníku obecně, abstraktně. Za prvé, zásobník obsahuje několik položek. (Tato vlastnost z něj činí *kontejner*, čili ještě obecnější abstrakci.) Dále je zásobník charakterizován pomocí operací, které je možné na něm provádět. Můžete provádět následující operace:

- vytvořit prázdný zásobník,
- přidat položku na vrchol zásobníku (operace *push*),
- odebrat položku z vrcholu zásobníku (operace *pop*),
- zjistit, zda je zásobník plný,
- zjistit, zda je zásobník prázdný.

Tento popis můžete srovnat s deklarací třídy, ve které veřejné členské funkce poskytují rozhraní, představující operace se zásobníkem, a soukromé datové položky se starají o ukládání dat. Koncept třídy lze pěkně přirovnat k přístupu ADT.

V privátní části je potřeba rozhodnout, jak budou data uchovávána. Například je možné použít obyčejné pole, dynamicky alokované pole nebo nějakou pokročilejší datovou strukturu, jako je spojový seznam. Veřejné rozhraní by však mělo přesnou reprezentaci skrýt. Namísto toho by mělo být rozhraní vyjádřeno obecnými termíny jako je vytvoření zásobníku, vložení položky atd. Výpis 9.10 ukazuje jeden možný přístup. Předpokládá implementaci typu bool. Pokud v systému tento typ není, můžete místo něho použít typ int a hodnoty 0 (false) a 1 (true).

Výpis 9.10. stack.h

```
// stack.h - definice třídy pro zásobník ADT
#ifndef _STACK_H_
#define _STACK_H_
typedef unsigned long Item;
class Stack
{
 private:
    enum {MAX = 10};    // specifická konstanta třídy
    Item items[MAX];    // místo pro uložení položek
    int top;            // index položky na vrcholu zásobníku
 public:
    Stack();
```

```
        bool isempty() const;
        bool isfull() const;
        // push() vrací false, jestliže je zásobník již plný, jinak vrací true
        bool push(const Item & item);   // přidá položku do zásobníku
        // pop() vrací false, jestliže je zásobník již prázdný, jinak vrací
true
        bool pop(Item & item);          // odebere horní položku ze zásobníku
    };
    #endif
```

Kompatibilita:

Jestliže váš systém neimplementuje typ bool, můžete místo něj použít typ `int` a hodnoty `0` (`false`) a 1 (`true`). Také je možné, že váš systém podporuje starší nestandardní zápis jako `boolean` nebo `Boolean`.

V tomto případě je v soukromé části vidět, že zásobník je implementován pomocí pole, ale veřejná část tuto skutečnost neodhaluje. Můžete tedy toto pole nahradit třeba dynamicky alokovaným polem, aniž byste změnili rozhraní třídy. To znamená, že ke změně implementace zásobníku nemusíte přepisovat programy používající tento zásobník. Stačí znovu zkompilovat kód pro zásobník a sestavit s již existujícím programem.

Rozhraní je trochu redundantní v tom, že metody `pop` a `push` nejsou typu `void`, ale vracejí informace o stavu zásobníku (plný nebo prázdný). Díky tomu má programátor několik možností, jak ošetřit přeplnění nebo vyprázdnění zásobníku. Před úpravou zásobníku může zjistit jeho stav pomocí funkcí `isempty()` a `isfull()` nebo určit úspěšnost operace pomocí návratových hodnot funkcí `pop()` a `push()`.

Třída nedefinuje zásobník jako nějaký konkrétní typ, ale spíše ho popisuje jako obecný typ `Item`. V tomto případě vytvoří hlavičkový soubor typ `Item` pomocí klíčového slova `typedef` stejně jako typ `unsigned long`. Jestliže chcete například zásobník pro typ double nebo pro strukturu, změníte pouze deklaraci `typedef` a deklaraci třídy a definice metod necháte beze změny. Šablony třídy (kapitola 13, „Znovupoužitelný kód v jazyce C++") poskytují mnohem silnější metodu pro oddělení typu uložených dat od návrhu třídy.

Dále budeme implementovat metody třídy. Výpis 9.11 ukazuje jednu z možností.

Výpis 9.11. stack.cpp

```
// stack.cpp — členské funkce třídy Stack
  #include "stack.h"
  Stack::Stack()  // vytvoření prázdného zásobníku
  {
      top = 0;
  }
  bool Stack::isempty() const
  {
      return top == 0;
  }
  bool Stack::isfull() const
```

```
{
    return top == MAX;
}
bool Stack::push(const Item & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}
bool Stack::pop(Item & item)
{
    if (top > 0)
    {
        item = items[-top];
        return true;
    }
    else
        return false;
}
```

Bezparametrický konstruktor zaručuje, že všechny zásobníky jsou vytvořeny jako prázdné. Kód metod `pop()` a `push()` zaručuje správnou manipulaci s vrcholem zásobníku. Díky těmto zárukám je objektově orientované programování mnohem spolehlivější. Předpokládejme, že bychom místo toho vytvořili pole reprezentující zásobník a nezávislou proměnnou reprezentující index vrcholu zásobníku. V tom případě je na vaší zodpovědnosti, abyste sladili kód vždy při vytvoření nového zásobníku. Bez ochrany, kterou nabízí privátní datové položky, vždy existuje možnost chyby v programu, díky níž dojde k neúmyslné změně dat.

Nyní zásobník otestujeme. Výpis 9.12 modeluje život úředníka, zpracovávajícího objednávky z vrcholu koše způsobem LIFO (last in – first out).

Výpis 9.12. stacker.cpp

```
// stacker.cpp - test třídy Stack
#include <iostream>
using namespace std;
#include <cctype> // nebo ctype.h
#include "stack.h"
int main()
{
    Stack st; // vytvoří prázdný zásobník
    char c;
    unsigned long po;
    cout << "Pro pridani objednavky zadejte \"P\",\n"
         << "pro zpracovani \"Z\" a pro ukonceni \"K\".\n";
    while (cin >> c && toupper(c) != 'K')
    {
```

```
            while (cin.get() != '\n')
                continue;
            if (!isalpha(c))
        {
                cout << '\a';
                continue;
        }
        switch(c)
        {
             case 'P':
             case 'p': cout << "Zadejte cislo objednavky: ";
                       cin >> po;
                       if (st.isfull())
                          cout << "zasobnik je jiz plny\n";
                       else
                           st.push(po);
                       break;
            case 'Z':
            case 'z': if (st.isempty())
                           cout << "zasobnik je jiz prazdny\n";
                       else {
                           st.pop(po);
                           cout << "Objednavka c." << po << " byla zpraco-
vana\n";
                       }
                       break;
        }
        cout << "Pro pridani objednavky zadejte \"P\",\n"
            << "pro zpracovani \"Z\" a pro ukonceni \"K\".\n";
      }
      cout << "Nashledanou\n";
      return 0;
  }
```

Malý cyklus while, s jehož pomocí se zbavujeme zbytku řádku, zde není potřeba, ale bude se hodit v kapitole 13, až budeme tento program upravovat. Zde je příklad běhu programu:

```
Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
P
Zadejte cislo objednavky: 17885
 Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
Z
Objednavka c. 17885 byla zpracovana
 Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
P
Zadejte cislo objednavky: 17965
 Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
```

```
P
Zadejte cislo objednavky: 18002
 Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
Z
 Objednavka c.18002 byla zpracovana
 Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
Z
Objednavka c. 17965 byla zpracovana
 Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
Z
 zasobnik je jiz prazdny
 Pro pridani objednavky zadejte "P",
 pro zpracovani "Z" a pro ukonceni "K".
K
Nashledanou
```

Shrnutí

Objektově orientované programování klade důraz na způsob, jakým program reprezentuje data. Prvním krokem k řešení problému je v OOP popis dat z hlediska rozhraní s programem a určení způsobu jejich použití. Dále je třeba vytvořit třídu, která toto rozhraní implementuje. Soukromé datové položky ukládají informace, zatímco veřejné členské funkce (také nazývané metody) poskytují k těmto datům jediný přístup. Třída kombinuje data a metody do jednoho celku a soukromý aspekt slouží ke skrývání dat.

Deklarace třídy se obvykle rozděluje na dvě části do dvou samostatných souborů. Vlastní deklarace třídy společně s metodami představovanými funkčními prototypy je obsažena v hlavičkovém souboru. Zdrojový kód, definující členské funkce, je obsažen v souboru s metodami. Tento přístup odděluje popis rozhraní od implementačních podrobností. V zásadě potřebujete znát pouze veřejné rozhraní třídy, abyste ji mohli použít. Samozřejmě se můžete podívat i na implementaci (pokud nebyl kód dodán pouze ve zkompilovaném tvaru), ale váš program by se neměl spoléhat na implementační podrobnosti, jako například že určitá hodnota je uložena v proměnné typu `int`. Dokud program s třídou komunikují pouze pomocí metod definujících toto, můžete bez obav obě části zlepšovat, aniž byste se museli obávat nepředvídaných vzájemných působení.

Třída je uživatelem definovaný typ a objekt je instancí třídy. To znamená, že objekt je proměnnou tohoto typu nebo ekvivalentem proměnné, jako například paměť přidělená operátorem `new` konkrétní třídě. Jazyk C++ se snaží uživatelem definované typy co nejvíce přiblížit standardním typům, takže můžete deklarovat objekty, ukazatele na objekty a pole objektů. Objekty můžete předávat jako parametry, vracet je jako návratovou hodnotu funkce a přiřadit je jinému objektu stejného typu. Jestliže vytvoříte metodu konstruktoru, můžete objekty při vytváření inicializovat. Pokud dodáte metodu destruktoru, program ji spustí, jakmile skončí platnost objektu.

Každý objekt uchovává vlastní kopie části dat deklarace třídy, metody třídy však objekty sdílí. Jestliže `mr_object` je název nějakého konkrétního objektu a `try_me()` je název členské funkce, pak členskou funkci vyvoláte pomocí operátoru přístupu ke složkám objektu: `mr_object.try_me()`. V terminologii OOP se tomuto volání funkce říká zaslání zprávy `try_me` objektu `mr_object`. Všechny odkazy na datové položky třídy v metodě `try_me()` se potom vztahují na datové položky objektu `mr_object`. Podobně volání `i_object.try_me()` získá přístup k datovým položkám objektu `i_object`.

Jestliže chcete, aby členská funkce pracovala s více jak jedním objektem, můžete předat této metodě další objekty jako parametry. Pokud se metoda potřebuje explicitně odkázat na objekt, který ji vyvolal, může použít ukazatel `this`. Ukazatel `this` je nastaven na adresu volajícího objektu, takže výraz `*this` představuje alias samotného objektu.

Třídy lze dobře přirovnat k popisu abstraktních datových typů (ADT). Veřejné rozhraní členské funkce poskytuje služby popsané v ADT, zatímco soukromá část třídy a kód metod poskytují implementaci, která je klientům třídy skryta.

Opakovací otázky

1. Co je třída?
2. Jak třída dosáhne abstrakce, zapouzdření a skrývání dat?
3. Jaký je vztah mezi objektem a třídou?
4. Čím se liší členské funkce (kromě toho, že jsou funkcemi) od datových položek třídy?
5. Definujte třídu představující bankovní účet. Datové položky by měly obsahovat jméno vkladatele, číslo účtu (použijte řetězec) a zůstatek. Členské funkce by měly umožňovat:
   - Vytvořit objekt a inicializovat ho.
   - Přiřadit datovým složkám počáteční hodnoty.
   - Zobrazit jméno vkladatele, číslo účtu a zůstatek na účtu.
   - Uložit částku peněz zadanou jako parametr.
   - Vybrat částku peněz zadanou jako parametr.

   Vytvořte pouze deklaraci třídy, implementace metod nejsou potřeba. (Příležitost napsat implementaci budete mít v programovacím cvičení č. 1.)
6. Kdy jsou volány konstruktory třídy a kdy destruktory?
7. Vytvořte kód konstruktoru pro třídu bankovního účtu z otázky číslo 5.
8. Co je implicitní konstruktor a jaké jsou jeho výhody?
9. Upravte třídu `Stock` (verzi obsaženou v souboru `stock2.h`) tak, aby měla členské funkce vracející hodnoty jednotlivých datových položek. Poznámka: Metoda vracející název firmy by neměla poskytovat nástroj na změnu pole. To znamená, že nemůže jednoduše vracet ukazatel na typ `char`. Mohla by vracet `konstantní` ukazatel nebo ukazatel na kopii pole vytvořeného pomocí operátoru `new`.
10. Co jsou výrazy `this` a `*this`?

Programovací cvičení

1. Vytvořte definice metod třídy popsané v otázce číslo 5 a napište krátký program, ilustrující možnosti této třídy.
2. Udělejte cvičení číslo 4 z kapitoly 8, ale nahraďte odpovídající kód deklarací třídy `golf`. Počáteční hodnoty vytvořte pomocí konstruktoru s vhodnými parametry.
3. Uvažujte následující deklaraci struktury:

```
struct customer {
        char fullname[35];
        double payment;
};
```

Napište program, který přidává a odebírá struktury uživatele ze zásobníku představovaného deklarací třídy. Kdykoli je zákazník odebrán, přidejte jeho platbu k mezisoučtu a tento vypište.

4. Zde je deklarace třídy:

```
class Move
{
  private:
      double x;
      double y;
  public:
      Move(double a = 0, double b = 0);  // nastaví x, y na a, b
      showmove() const;                   // zobrazí aktuální hodnoty
x a y
      Move add(const Move & m) const;
// funkce přidá proměnnou x objektu m do proměnné x volajícího objektu a
// získá novou proměnnou x, přidá proměnnou y objektu m do proměnné y
// volajícího objektu a získá novou proměnnou y, vytvoří nový objekt move
// inicializovaný na nové hodnoty x a y a objekt vrátí
     reset(double a = 0, double b = 0); // nastaví x,y na a, b
};
```

Přidejte definice členských funkcí a program používající tuto třídu.

5. Váš pes má následující vlastnosti:

- Data
  - Má jméno, jehož délka nepřesahuje 19 znaků.
  - Pes má index spokojenosti (IS), který je reprezentován celým číslem.
- Vlastnosti
  - Nový pes začíná s určeným jménem a IS hodnoty 50.
  - Hodnota IS se může změnit.
  - Pes může vypsat své jméno a IS.
  - Implicitní jméno psa je „Puňťa“.

- Vytvořte deklaraci třídy Pes (datové členy a prototypy členských funkcí) reprezentující psa. Napište definice členských funkcí a vytvořte krátký program, který demonstruje všechny rysy třídy Pes.

6. Jednoduchý seznam můžeme popsat následovně:
   - Jednoduchý seznam může obsahovat 0 a více položek konkrétního typu.
   - Můžete vytvořit prázdný seznam.
   - Můžete přidat položku do seznamu.
   - Můžete zjistit, zda je seznam prázdný.
   - Můžete zjistit, zda je seznam plný.
   - Můžete si prohlédnout každou položku v seznamu a provést s ní nějakou činnost.

   Jak vidíte, jedná se skutečně o jednoduchý seznam, který neumožňuje například vkládání nebo mazání. Hlavní význam takového seznamu je poskytnout zjednodušený programový projekt. V takovém případě vytvořte třídu odpovídající tomuto popisu. Seznam můžete implementovat jako pole nebo, pokud znáte datové typy, jako spojový seznam. Veřejné rozhraní by to však ovlivnit nemělo. To znamená, že by nemělo mít indexy polí, ukazatele na uzly, apod. Mělo by být vyjádřeno v obecných koncepcích vytvoření seznamu, přidání položky do seznamu atd. Obvykle se k prohlížení prvků a provedení nějaké činnosti použije funkce, která má jako parametr ukazatel na funkci:

```
void visit(void (*pf)(Item &));
```

   Zde `pf` ukazuje na funkci (ne na členskou funkci), která má jako parametr referenci na parametr `Item`, kde `Item` je typ položek v seznamu. Funkce `visit()` použije tuto funkci na všechny položky seznamu.

   Měli byste také napsat krátký program používající tento návrh.

Práce s třídami

KAPITOLA 10

Témata kapitoly:

- Přetěžování operátorů
- Spřátelené funkce
- Přetížení operátoru << pro výstup
- Stavové položky
- Generování náhodných hodnot pomocí funkce `rand()` – Automatické konverze a přetypování tříd
- Funkce pro konverze tříd

Třídy v C++ se vyznačují mnoha vlastnostmi, jsou komplexní a výkonné. V kapitole 9 jste se naučili definovat a používat jednoduchou třídu a tím jste se vydali na cestu k objektově orientovanému programování. Viděli jste, jak třída definuje datový typ definováním typu dat, která budou použita k reprezentaci objektu a také definováním operací, které bude možné s daty pomocí členských funkcí provádět. Také jste se dozvěděli o dvou speciálních členských funkcích, konstruktoru a destruktoru, které řídí vytváření a rušení objektů podle specifikace třídy. Tato kapitola vás posune o několik kroků dále – spíše než všeobecné principy budete zkoumat vlastnosti tříd a soustředíte se na techniky jejich návrhu. Některé zde popisované vlastnosti vám budou pravděpodobně připadat jasné, některé trochu tajemnější. Abyste tyto nové vlastnosti co nejlépe pochopili, měli byste uvedené příklady zkoušet a experimentovat s nimi. Co se stane, když v této funkci použiji běžný parametr namísto reference? Co se bude dít, jestliže něco v destruktoru vynechám? Nebojte se dělat chyby; obvykle se více naučíte odstraňováním chyb než když něco budete dělat správně, ale bezmyšlenkovitě. (Nepředpokládejte ovšem, že záplava chyb nevyhnutelně vede k hlubokým znalostem.) Na konci budete odměněni tím, že hlouběji porozumíte fungování jazyka C++ a tomu, co vám může dát.

V této kapitole začneme přetěžováním operátorů, které vám umožní používat standardní operátory C++ typu = a + u objektů tříd. Potom rozebereme přátele, což je mechanismus C++, umožňující nečlenským funkcím přistupovat k privátním datům. Nakonec se podíváme, jakým způsobem instruovat jazyk C++, aby prováděl automatickou konverzi mezi třídami. V průběhu této a následující kapitoly mnohem více oceníte roli, kterou hrají konstruktory a destruktory. Také uvidíte několik fází, kterými můžete projít při vývoji a vylepšování návrhu třídy.

Učení se jazyku C++ s sebou přináší jednu potíž (minimálně do doby, než do něho dostatečně proniknete) – je zde strašně mnoho věcí k zapamatování. A je nesmyslné očekávat, že si vše zapamatujete, dokud nezískáte dostatek zkušeností, o které se opřete. V tomto ohledu je učení jazyka C++ podobné učení se s rozsáhlým textovým nebo tabulkovým procesorem. Žádná funkce není odstrašující, ale v praxi většina lidí skutečně dobře zná pouze pravidelně používané funkce, jako jsou vyhledávání textu nebo nastavení kurzívy. Možná si vzpomenete, že jste někde četli, jak vygenerovat alternativní znaky nebo jak vytvořit obsah, ale tyto znalosti pravděpodobně nebudou součástí vaší každodenní praxe, dokud se nedostanete do situace, že tyto funkce budete často potřebovat. Asi nejlepším způsobem, jak vstřebat bohatou látku této kapitoly, je začít zahrnovat některé z těchto vlastností do svých vlastních programů. Jakmile díky zkušenostem těmto vlastnostem více porozumíte a oceníte je, začněte přidávat další. Jak doporučil Bjarne Stroustrup, tvůrce jazyka C++, na konferenci profesionálních programátorů v C++: „Dostávejte se do jazyka pozvolna. Zbavte se pocitu, že musíte používat všechny vlastnosti a nesnažte se je všechny používat hned první den."

Přetěžování operátorů

Pojďme se podívat na techniku, která zpříjemní operace s objekty. *Přetěžování operátorů* je dalším, příkladem polymorfismu v jazyce C++. V kapitole 8 jste viděli, jakým způsobem C++ umožňuje definovat několik funkcí se stejným názvem, ale s různou signaturou (seznamem parametrů). To bylo přetěžování funkcí neboli funkční polymorfismus. Účelem této techniky je umožnit použití funkce se stejným názvem pro stejnou základní operaci, i když pro odlišné datové typy. (Představte si, jak by čeština vypadala hrozně, kdybyste museli používat vždy jiné sloveso pro jiný objekt – zvedni_lvr pro zvednutí levé ruky, ale zvedni_prn pro zvednutí pravé nohy.) Přetěžování operátorů rozšiřuje koncept přetěžování i na operátory a umožňuje přiřadit jim více významů. Ve skutečnosti je v C++ (i v C) již mnoho operátorů přetíženo. Pokud například operátor * použijete na adresu, získáte hodnotu, která je na ní uložena. Použijete-li však operátor * na dvě čísla, získáte výsledek těchto hodnot. Pomocí počtu a typu operandů jazyk C++ určí, jaká činnost se má provést.

C++ umožňuje rozšířit přetěžování operátorů i na typy definované uživatelem, můžete například pomocí operátoru + sečíst dva objekty. Kompilátor opět pomocí počtu a typu operandů určí, která definice sčítání se má použít. Díky přetížení operátorů může kód často vypadat přirozeněji. Běžným výpočetním úkolem je například sečtení dvou polí. Většinou to skončí podobně jako následující smyčka for:

```
for (int i = 0; i < 20; i++)
        evening[i] = sam[i] + janet[i]; // sčítání prvku po prvku
```

Ale v C++ můžete nadefinovat třídu reprezentující pole a přetěžující operátor +. Můžete tedy napsat:

```
evening = sam + janet;        // sečte dva objekty reprezentující pole
```

Přesně toto uděláme v kapitole 12. (Proč ne teď? Protože musíte také přetížit operátor [], a to je trošku komplikovanější než přetížení operátoru +.) Tento jednoduchý zápis sčítání skrývá mechanismus a zdůrazňuje to, co je podstatné, a to je další cíl OOP.

Operátor přetížíte pomocí speciálního tvaru funkce. Takové funkci se říká *funkce operátoru*. Funkce operátoru má tento tvar:

```
operatorop(seznam parametrů)
```

kde *op* je symbol operátoru, který chcete přetížit. To znamená, že funkce `operator+()` přetíží operátor + (*op* je +) a funkce `operator*()` přetíží operátor * (*op* je *). *Op* ovšem musí být platný operátor jazyka C++, nemůžete si vymyslet nový symbol. Nemůžete mít například funkci `operator@()` , protože C++ žádný operátor @ nemá. Ale funkce `operator[]()` by již přetížila operátor [], protože ten představuje index pole. Předpokládejme například, že máte třídu `Salesperson`, a definujete pro ni členskou funkci `operator+()` tak, aby přetížený operátor přičítal údaje o prodeji jednoho objektu třídy k jinému objektu. Jestliže jsou tedy `district2`, sid a sara všechno objekty třídy `Salesperson`, můžete napsat tuto rovnici:

```
district2 = sid + sara;
```

Kompilátor pozná, že operandy patří třídě `Salesperson` a nahradí operátor odpovídající funkcí operátoru:

```
district2 = sid.operator+(sara);
```

Tato funkce potom použije objekt `sid` implicitně (protože vyvolal tuto metodu) a objekt `sara` explicitně (protože je předán jako parametr), k provedení součtu, který potom vrátí. Příjemné na celé věci je, že místo dlouhého zápisu funkce můžete použít zápis s operátorem +.

C++ stanoví na přetěžování operátorů jistá omezení, ta však pochopíte snadněji, až uvidíte, jak přetěžování funguje. Proto nejdříve vytvoříme několik příkladů pro objasnění a potom se budeme zabývat omezeními.

Zpracování času

Jestliže jste na popisu pracovali dopoledne 2 hodiny 35 minut a odpoledne 2 hodiny 40 minut, jak dlouho jste pracovali celkem? Uvádíme příklad, kde koncept sčítání dává smysl, ale sčítané jednotky (směs hodin a minut) neodpovídají vestavěným typům. V kapitole 7 jste podobný případ řešili definováním struktury `travel_time` a funkcí `sum()`, která takové struktury sčítala. Nyní můžeme problém zevšeobecnit na třídu `Time` s metodou pro sčítání. Začneme obyčejnou metodou a potom se podíváme, jak se převede na přetížený operátor. Na výpisu 10.1 je deklarace třídy.

Výpis 10.1

```
mytime0.h
//mytime0.h - Třída Time před použitím operátoru přetížení
#ifndef _MYTIME0_H_
#define _MYTIME0_H_
#include <iostream>
using namespace std;
class Time
```

```
{
private:
   int hours;
   int minutes;
public:
   Time();
   Time(int h, int m = 0);
   void AddMin(int m);
   void AddHr(int h);
   void Reset(int h = 0, int m = 0);
   Time Sum(const Time & t) const;
   void Show() const;
};
#endif
```

Třída má metody pro úpravu a vynulování času, zobrazení hodnot času a sečtení dvou časových hodnot. Definice těchto metod jsou ve výpisu 10.2. Všimněte si, jak metody AddMin() a Sum() pomocí dělení celých čísel a operátoru modulo upravují hodnoty minutes (minuty) a hours (hodiny), když celkový počet minut přesáhne hodnotu 59.

Výpis 10.2

```
mytime0.cpp
// mytime0.cpp - implementace metod třídy Time
#include "mytime0.h"
Time::Time()
{
   hours = minutes = 0;
}
Time::Time(int h, int m)
{
   hours = h;
   minutes = m;
}
void Time::AddMin(int m)
{
   minutes += m;
   hours += minutes / 60;
   minutes %= 60;
}
void Time::AddHr(int h)
{
   hours += h;
}
void Time::Reset(int h, int m)
{
   hours = h;
   minutes = m;
}
Time Time::Sum(const Time & t) const
{
```

```
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t. hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}
void Time::Show() const
{
    cout << hours << " hodin, " << minutes << " minut";
    cout <<'\n';
}
```

Zamyslete se nad kódem funkce `Sum()`. Všimněte si, že jejím parametrem je reference, ale návratovým typem reference není. V parametru je reference z důvodu efektivity. Kdyby byl objekt `Time` předán hodnotou, program by dal stejné výsledky. Obvykle je však rychlejší a z hlediska paměti vhodnější předat pouze referenci.

Návratovou hodnotou však reference být nemůže. Důvodem je, že funkce vytvoří nový objekt `Time (sum)` představující součet dvou objektů `Time`. Vrácením objektu program vytvoří jeho kopii, kterou může použít volající funkce. Kdyby však návratovým typem byla reference na `Time`, odkazovala by na objekt `sum`. Objekt `sum` je však lokální proměnná a je při ukončení funkce zničen, takže reference by byla referencí na neexistující objekt. Použití objektu `Time` jako návratového typu znamená, že program vytvoří kopii objektu `sum` před jeho zničením a volající funkce získá tuto kopii.

Upozornění

Jako návratovou hodnotu nepoužívejte referenci na lokální proměnnou ani jiný dočasný objekt.

Ve výpisu 10.3 je test části, ve které třída provádí sčítání.

Výpis 10.3 usetime0.cpp

```
// usetime0.cpp - použití prvního návrhu třídy Time
// společná kompilace souborů usetime0.cpp a mytime0.cpp
#include <iostream>
#include "mytime0.h"
using namespace std;
int main()
{
    Time A;
    Time B(5, 40);
    Time C(2, 55);
    A.Show();
    B.Show();
    C.Show();
    A = B.Sum(C);
    A.Show();
    return 0;
}
```

Zde je výstup:

```
0 hodin, 0 minut
5 hodin, 40 minut
2 hodin, 55 minut
8 hodin, 35 minut
```

Přidání operátoru sčítání

Upravit třídu `Time` tak, aby používala přetížený operátor sčítání, je jednoduché. Stačí změnit název metody `Sum()` na podivně vypadající název `operator+()`. Správně – jen přidáte symbol operátoru (v tomto případě +) za název `operator` a výsledek použijete jako název metody. Toto je jediné místo v identifikačním názvu, kde můžete použít znak, kterým není písmeno, číslice ani podtržítko. Tato malá změna je zachycena na výpisech 10.4 a 10.5.

Výpis 10.4 mytime1.h

```
//mytime1.h - Třída Time po použití operátoru přetížení
#ifndef _MYTIME1_H_
#define _MYTIME1_H_
#include <iostream>
using namespace std;
class Time
{
private:
  int hours;
  int minutes;
public:
  Time();
  Time(int h, int m = 0);
  void AddMin(int m);
  void AddHr(int h);
  void Reset(int h = 0, int m = 0);
  Time operator+(const Time & t) const;
  void Show() const;
};
#endif
```

Výpis 10.5 mytime1.cpp

```
// mytime1.cpp - implementace metod třidy Time
#include "mytime1.h"
Time::Time()
{
  hours = minutes = 0;
}
Time::Time(int h, int m)
{
```

```
    hours = h;
    minutes = m;
}
void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr(int h)
{
    hours += h;
}
void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}
Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t. hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}
void Time::Show() const
{
    cout << hours << " hodin, " << minutes << " minut";
    cout <<'\n';
}
```

Stejně jako metoda `Sum()` je i metoda `operator+()` vyvolána objektem třídy `Time`, má objekt `Time` jako druhý parametr a vrací objekt `Time`. Můžete ji tedy vyvolat pomocí stejné syntaxe jako metodu `Sum()`:

```
A = B.operator+(C);          // zápis pomocí funkce
```

Díky pojmenování metody `operator+()` však můžete použít zápis pomocí operátoru:

```
A = B + C;                   // zápis pomocí operátoru
```

Oběma zápisy vyvoláte metodu operator+(). Všimněte si, že v zápisu pomocí operátoru je objekt vlevo od operátoru (v tomto případě B) objekt volající, zatímco objekt vpravo (v tomto případě C) je objekt předaný jako parametr. Tato vlastnost je zachycena na výpisu 10.6.

Výpis 10.6 usetime1.cpp

```
// usetime1.cpp - použití druhého návrhu třídy Time
// společná kompilace souborů usetime1.cpp a mytime1.cpp
#include <iostream>
```

```
#include "mytime1.h"
using namespace std;
int main()
{
    Time A;
    Time B(5, 40);
    Time C(2, 55);
    A.Show();
    B.Show();
    C.Show();
    A = B.operator+(C);     // zápis pomocí funkce
    A.Show();
    B = A + C;                     // zápis pomocí operátoru
    B.Show();
    return 0;
}
```

Zde je výstup:

```
0 hodin, 0 minut
5 hodin, 40 minut
2 hodin, 55 minut
8 hodin, 35 minut
11 hodin, 30 minut
```

Stručně řečeno, díky názvu `operator+()` můžete na vyvolání této funkce použít buď zápis pomocí funkce nebo operátoru. Pomocí operandů kompilátor zjistí, co má dělat:

```
int a, b, c;
Time A, B, C;
c = a + b;          // použije sčítání celých čísel
C = A + B;          // použije sčítání definované pro objekty třídy Time
```

Omezení přetížení

Většinu operátorů v jazyce C++ (viz tabulku 10.1) lze přetížit stejným způsobem. Přetížené operátory nemusí nutně být členskými funkcemi (až na několik výjimek). Alespoň jeden z operandů však musí být uživatelem definovaný typ. Podívejme se podrobněji na omezení, která jazyk C++ zavádí na uživatelem definované přetěžování operátorů:

1. Přetěžovaný operátor musí mít alespoň jeden operand, kterým je uživatelem definovaný typ. To vám znemožní přetížit operátory standardních typů. Nemůžete tedy předefinovat operátor mínus (-) tak, aby místo rozdílu dvou hodnot typu `double` dal jejich součet. Díky tomuto omezení je zachována logika programu, ačkoli třeba brání tvůrčímu účetnictví.
2. Nemůžete použít operátor způsobem, který porušuje syntaktická pravidla původního operátoru, například přetížit operátor modulo (%) na použití s jediným operandem:

```
int x;
Time shiva;
```

```
% x;          // nelze použít s operátorem modulo
% shiva;  // nelze použít jako přetížený operátor
```

Podobně nemůžete změnit prioritu operátoru. Jestliže tedy přetížíte operátor sčítání, abyste mohli sečíst dvě třídy, bude mít nový operátor stejnou prioritu jako obyčejné sčítání.

3. Nelze vytvářet nové symboly operátorů. Nemůžete například označit umocňování definováním funkce `operator**()`.
4. Nelze přetížit následující operátory:

sizeof	operátor `sizeof`
`.`	operátor přístupu ke členu
`.*`	operátor ukazatel na člen
`::`	operátor rozlišení
`?:`	operátor podmínky
`typeid`	operátor RTTI
`const_cast`	operátor přetypování
`dynamic_cast`	operátor přetypování
`reinterpret_cast`	operátor přetypování
`static_cast`	operátor přetypování

Zbývající operátory v tabulce 10.1 přetížit lze.

5. Většinu operátorů z tabulky 10.1 lze přetížit buď pomocí členských nebo nečlenských funkcí. Následující operátory však můžete přetížit pouze pomocí členských funkcí:

`=`	operátor přiřazení
`()`	operátor volání funkce
`[]`	operátor indexování
`->`	operátor přístupu ke členům pomocí ukazatele

Poznámka

Neprobrali jsme a ani nebudeme probírat každý operátor uvedený v seznamu omezení nebo v tabulce 10.1. Operátory neprobírané v hlavní části knihy však najdete v dodatku E.

Tabulka 10.1 Operátory, které lze přetížit

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>,~=</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td><td>-=</td><td>*=</td></tr>
<tr><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td><td><<</td><td>>></td><td>>>=</td></tr>
<tr><td><<=</td><td>==</td><td>!=</td><td><=</td><td>>=</td><td>&&</td><td>||</td><td>++</td></tr>
<tr><td>—</td><td>,</td><td>->*</td><td>-></td><td>()</td><td>[]</td><td>new</td><td>delete</td></tr>
<tr><td>new []</td><td>belete []</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Kromě těchto formálních omezení byste měli při přetěžování operátorů zachovávat rozumnou zdrženlivost. Nepřetěžujte například operátor * tak, aby zaměňoval datové po-

ložky dvou objektů třídy `Time`. Nic v zápisu nenapoví, co operátor udělal, lepší tedy bude definovat metodu třídy s vysvětlujícím názvem `Swap()`.

Další přetížené operátory

Ve třídě `Time` mají smysl i některé další operace. Můžete například odečítat jeden čas od druhého nebo ho násobit nějakým faktorem. Tím se nabízí přetypování operátorů odečítání a násobení. Technika je stejná jako u operátoru sčítání – vytvoříte metody `operator()` a `operator*()`. Přidejte tedy do deklarace třídy následující prototypy:

```
Time operator-(const Time & t) const;
Time operator*(double mult) const;
```

Potom do implementačního souboru přidejte následující definice metod:

```
Time Time::operator-( const Time & t) const
{
  Time diff;
  int tot1, tot2;
  tot1 = t.minutes + 60 * t.hours;
  tot2 = minutes + 60 * hours;
  diff.minutes = (tot2 - tot1) % 60;
  diff.hours = (tot2 - tot1) / 60;
  return diff;
}
Time Time::operator*(double mult) const
{
  Time result;
  long totalminutes = hours * mult * 60 + minutes * mult;
  result.hours = totalminutes / 60;
  result.minutes = totalminutes % 60;
  return result;
}
```

Po provedených změnách můžete tyto nové definice vyzkoušet pomocí kódu z výpisu 10.7. (Předpokladem je, že v upravených souborech třídy změníte verzi `mytime1` na `mytime2`.

```
Výpis 10.7 usetime2.cpp
// usetime2.cpp - použití třetího návrhu třídy Time
// společná kompilace souborů usetime2.cpp a mytime2.cpp
#include <iostream>
#include "mytime2.h"
using namespace std;
int main()
{
  Time A;
  Time B(5, 40);
  Time C(2, 55);
  A.Show();
  B.Show();
  C.Show();
  A = B + C;                    // operator+()
```

```
    A.Show();
    A = B - C;                    // operator-()
    A.Show();
    A = B * 2.75;     // operator*()
    A.Show();
    return 0;
}
```

Zde je výstup:

```
0 hodin, 0 minut
5 hodin, 40 minut
2 hodin, 55 minut
8 hodin, 35 minut
2 hodin, 45 minut
15 hodin, 35 minut
```

Úvod k přátelům

Jak jste již viděli, jazyk C++ řídí přístup k soukromým částem objektu třídy. Obvykle jediný možný přístup představují veřejné metody třídy, ale v některých případech je toto omezení příliš přísné a z hlediska určitých programovacích problémů se nehodí. Pro takové případy nabízí jazyk C++ jinou formu přístupu – pomocí `přítele`. Přátelé jsou trojího druhu:

- Spřátelené funkce
- Spřátelené třídy
- Spřátelené členské funkce

Tím, že z funkce učiníte přítele třídy, dáte jí stejná přístupová práva, jako má členská funkce dané třídy. Nyní se podíváme na spřátelené funkce, a zbývající dva druhy si necháme až do kapitoly 14.

Dříve než si ukážeme, jak se spřátelené funkce vytváří, podívejme se, k čemu je můžete potřebovat. Často vzniká potřeba spřátelených funkcí při přetěžování binárního operátoru (se dvěma parametry) třídy. Přesně taková situace vzniká při násobení objektu `Time` reálným číslem, takže si tento případ prostudujeme.

V příkladě s třídou `Time` se přetížený operátor násobení liší od ostatních dvou přetížených operátorů v tom, že kombinuje dva různé typy. To znamená, že oba operátory sčítání a odčítání kombinují dvě hodnoty třídy `Time`, ale operátor násobení kombinuje hodnotu objektu `Time` s hodnotou typu `double` a tím je použití tohoto operátoru omezeno. Nezapomeňte, že levý operand je volající objekt. To znamená, že se příkaz

```
A = B * 2.75;
```

přeloží na následující volání členské funkce:

```
A = B.operator*(2.75);
```

Co ale následující příkaz?

```
A = 2.75 * B;      // neodpovídá členské funkci
```

Teoreticky by `2.75 * B` mělo být stejné jako `B * 2.75`, první výraz však neodpovídá členské funkci, protože `2.75` není objektem typu `Time`. Nezapomeňte, že levý operand je volající objekt a `2.75` objektem není. Kompilátor tedy nemůže tento výraz nahradit voláním členské funkce.

Jednou z možností, jak se tomuto problému vyhnout, je říci každému (a zapamatovat si), že lze psát pouze `B * 2.75`, ale nikdy `2.75 * B`. Takové řešení představuje kamarádský přístup programátora a pamatuje na uživatele, ale s OOP nemá nic společného.

Existuje však další možnost – použít nečlenskou funkci. (Pamatujte, že většinu operátorů lze přetížit pomocí členských nebo nečlenských funkcí.) Nečlenská funkce není vyvolána objektem a všechny hodnoty, které používá, včetně objektů, jsou explicitními parametry. Kompilátor by tedy mohl přirovnat výraz

```
A = 2.75 * B;      // neodpovídá členské funkci
```

k následujícímu volání nečlenské funkce:

```
A = operator*(2.75, B);
```

Funkce by měla takový prototyp:

```
Time operator*(double m, const Time & t);
```

U nečlenské funkce s přetíženým operátorem odpovídá levý operand operátoru prvnímu parametru funkce operator a pravý operand odpovídá druhému parametru.

Použití nečlenské funkce sice vyřeší problém s požadovaným pořadím operandů (nejdříve `double`, potom `Time`), vznikne však problém nový: nečlenská funkce nemůže přímo přistupovat k soukromým datům ve třídě. Alespoň o obyčejných nečlenských funkcích to platí. Existuje však zvláštní kategorie nečlenské funkce zvaná *spřátelená funkce*, která k soukromým členům třídy přístup má.

Vytváření spřátelených funkcí

Prvním krokem při vytváření spřátelené funkce je umístění prototypu do deklarace třídy a přidání klíčového slova `friend` před deklaraci:

```
friend Time operator*(double m, const Time & t); // bude v deklaraci třídy
```

Důsledkem takového prototypu je, že

- ačkoli je funkce `operator*()` deklarována v deklaraci třídy, není to členská funkce,
- ačkoli funkce `operator*()` není členskou funkcí, má stejná přístupová práva jako členská funkce.

Druhým krokem je napsání definice funkce. Protože se nejedná o členskou funkci, nepoužijete kvalifikátor `Time::` a v definici také nebude klíčové slovo `friend`:

```
Time operator*(double m, const Time & t)
{
  Time result;
  long totalminutes = t.hours * mult * 60 + t.minutes * mult;
```

```
    result.hours = totalminutes / 60;
    result.minutes = totalminutes % 60;
    return result;
}
```

Po takové deklaraci a definici se příkaz

```
A = 2.75 * B;
```

přeloží na

```
A = operator*(2.75, B);
```

a vyvolá právě definovanou nečlenskou spřátelenou funkci.

Stručně řečeno, spřátelená funkce třídy je nečlenská funkce, která má stejná přístupová práva jako funkce členská.

Jsou spřátelené funkce nevěrné zásadám OOP?

Na první pohled se může zdát, že co se týká skrývání dat, porušují spřátelené funkce zásady OOP, protože jejich mechanismus umožňuje nečlenským funkcím přistupovat k soukromým datům. To je příliš zúžený pohled. Místo toho považujte spřátelené funkce za součást rozšířeného rozhraní třídy. Z koncepčního hlediska je například násobení čísla `double` hodnotou `Time` víceméně stejné jako násobení hodnoty `Time` číslem `double`. Skutečnost, že v prvním případě je nutná spřátelená funkce, zatímco v druhém stačí členská funkce, vyplývá ze syntaxe jazyka C++, nikoli z rozdílu pojmů. Použijete-li současně spřátelenou funkci i metodu třídy, můžete vyjádřit obě operace pomocí téhož uživatelského rozhraní. Pamatujte také, že pouze deklarace třídy může rozhodnout o tom, které funkce budou spřátelené, takže deklarace třídy stále určuje, které funkce získají přístup k soukromým datům. Stručně řečeno, metody třídy a spřátelené funkce jsou pouze dvěma rozdílnými mechanismy pro vyjádření rozhraní třídy.

Výše uvedenou spřátelenou funkci můžete napsat jako členskou, změníte-li definici následujícím způsobem:

```
Time operator*(double m, const Time & t)
{
    return t * m;   // použití členské funkce
}
```

Původní verze přistupovala k proměnným `t.minutes` a `t.hours` explicitně, takže musela být použita spřátelená funkce. Tato verze používá objekt `t` třídy `Time` pouze jako celek a zpracování soukromých hodnot přenechává členské funkci, takže nemusí být spřátelenou funkcí. Přesto by však nebylo špatné udělat i z této verze spřátelenou funkci. Jednak zapojuje funkci jako součást oficiálního rozhraní, a za druhé, pokud později zjistíte, že potřebujete přímý přístup k soukromým datům, bude stačit změnit definici funkce a ne prototyp třídy.

Tip

Chcete-li přetížit operátor třídy a použít ho jako první operand u netřídního členu, můžete obrátit pořadí operandů pomocí spřátelené funkce.

Obecný druh přítele: přetížení operátoru <<

Jednou z nejužitečnějších vlastností tříd je možnost přetížení operátoru <<, takže ho můžete použít společně s objektem `cout` ke zobrazení obsahu nějakého objektu. V určitém směru je toto přetížení trochu komplikovanější, než co jsme viděli v předchozích příkladech, proto ho vytvoříme ve dvou krocích namísto v jednom.

Předpokládejme, že `trip` je objekt třídy `Time`. Pro zobrazení hodnot třídy `Time` používáme metodu `Show()`. Nebylo by však příjemné, kdybyste mohli napsat následující příkaz?

```
cout << trip; // pozná objekt cout třídu Time?
```

Ano můžete, protože << je jedním z operátorů jazyka C++, které lze přetížit. Ve skutečnosti je již tento operátor silně přetížen. V úplně základním tvaru představuje jeden z operátorů bitového posunu jazyků C a C++; posouvá bity doleva (viz příloha E). Třída `ostream` ho však přetěžuje a dělá z něho nástroj pro výstup. Vzpomeňte si, že `cout` je objektem třídy `ostream` a je dost inteligentní, aby rozeznal všechny základní typy jazyka C++. Je tomu tak proto, že deklarace třídy `ostream` obsahuje definice přetíženého `operátoru` << pro všechny základní typy. To znamená, že jedna definice používá parametr typu int, další typ double atd. Jeden způsob, jak naučit objekt `cout` rozeznat třídu Time, je přidat definice nových funkcí operátoru do deklarace třídy `ostream`. Dělat změny do souboru `iostream` a pohrávat si se standardním rozhraním však není dobrý nápad. Lepší způsob je naučit třídu `Time` používat objekt `cout` pomocí její deklarace.

První verze přetížení operátoru <<

Máte-li naučit třídu `Time` používat objekt `cout`, musíte použít spřátelenou funkci. Proč? Protože příkaz

```
cout << trip;
```

používá dva objekty a prvním je objekt třídy `ostream (cout)`. Jestliže přetížíte operátor `<<` pomocí členské funkce třídy `Time`, aby objekt třídy `Time` byl prvním parametrem, jak tomu bylo při přetěžování operátoru `*` pomocí členské funkce. To znamená, že byste museli použít operátor `<<` tímto způsobem:

```
trip << cout;  // pokud by operator<<() představoval členskou funkci třídy
               // Time
```

To by bylo zavádějící. Ale pomocí spřátelené funkce můžete operátor přetížit takto:

```
void operator<<(ostream & os, const Time & t)
{
      os << t.hours << " hodin, " << t.minutes << " minut";
}
```

Taková definice umožní pomocí příkazu

```
cout << trip;
```

vypsat data v následujícím formátu:

```
4 hodin, 23 minut
```

Přítel ano či ne?

Nová deklarace třídy `Time` učiní z funkce `operator<<()` spřátelenou funkci této třídy. Tato funkce, ačkoli není nepřátelská vůči třídě `ostream`, není její spřátelenou funkcí. Funkce `operator<<()` má jako parametry objekty tříd `ostream` a `Time`, takže by se mohlo zdát, že je spřátelená s oběma třídami. Pokud se však podíváte na její kód, všimnete si, že má přístup k jednotlivým členům objektu třídy `Time`, ale objekt třídy `ostream` používá pouze jako celek. Vzhledem k tomu, že k soukromým členům třídy `Time` přistupuje přímo, musí být spřátelenou funkcí této třídy. Nemůže však být spřátelenou funkcí třídy `ostream`, protože k soukromým členům objektu této třídy přímo nepřistupuje. To je příjemné, protože to znamená, že se definicí třídy `ostream` nemusíte zabývat.

Všimněte si, že nová definice funkce `operator<<()` používá jako první parametr referenci `os` na třídu `ostream`. Normálně bude `os` odkazovat na objekt `cout` tak jako ve výrazu `cout << trip`. Tento operátor byste však mohli použít i u jiných objektů třídy `ostream` a v takovém případě bude `os` odkazovat na tyto objekty. (Cože? Vy neznáte žádné jiné objekty třídy `ostream`? Nezapomeňte na objekt `cerr` uvedený v kapitole 9. V kapitole 16 se také naučíte, jak vytvářet nové objekty, které budou řídit výstup do souborů, a tyto objekty budou využívat metody třídy `ostream`. Potom budete moci pomocí definice funkce `operator<<()` vypisovat data třídy `Time` jak do souborů, tak i na obrazovku.) Navíc volání `cout << trip` by mělo používat samotný objekt `cout`, ne jeho kopii, proto funkce předává objekt jako referenci a ne jako hodnotu. Výraz `cout << trip` tedy způsobí, že `os` bude aliasem objektu `cout`, zatímco výraz `cerr << trip` učiní z `os` alias objektu `cerr`. Objekt třídy `Time` může být předán hodnotou i odkazem, protože hodnoty objektu budou funkci přístupné v obou případech. Opět je však třeba říci, že předání odkazem používá méně paměti a je rychlejší než předání hodnotou.

Druhá verze přetížení operátoru <<

Právě předvedená implementace má jeden problém. Příkazy jako

```
cout << trip;
```

fungují dobře, ale naše implementace nedovoluje kombinovat předefinovaný operátor << s operátory, které normálně používá objekt cout:

```
cout << "Trip time: " << trip << "(Tuesday)\n"; // nefunguje
```

Máte-li pochopit, proč takový příkaz nefunguje a co je třeba udělat, aby fungoval, musíte nejdříve vědět trochu více o fungování objektu cout. Uvažujte následující příkazy:

```
int x = 5;
int y = 8;
cout << x << y;
```

Jazyk C++ čte příkaz výstupu zleva doprava a pokládá jej za rovnocenný následujícímu příkazu:

```
(cout << x) << y;
```

Operátor <<, tak jak je definován v třídě `iostream`, přijímá jako parametr objekt třídy `ostream` vlevo od sebe. Výraz cout << x tento požadavek samozřejmě splňuje, protože `cout` je objektem třídy `ostream`. Příkaz výstupu ale také vyžaduje, aby i celý výraz (`cout << x`) byl typ objektu třídy `ostream`, protože tento výraz je nalevo od výrazu << y. Třída `ostream` tedy implementuje funkci `operator<<()` tak, že vrací objekt třídy ostream. Konkrétně vrací objekt volající, v tomto případě objekt `cout`. Výraz (`cout << x`) je sám objektem třídy ostream a může být použit nalevo od operátoru <<.

Stejně můžete postupovat u spřátelené funkce. Pouze ji upravte tak, aby vracela referenci na objekt třídy ostream:

```
ostream & operator<<(ostream & os, const Vector & v)
{
    os << t.hours << " hodin, " << t.minutes << " minut";
    return os;
}
```

Všimněte si, že návratovým typem je reference na třídu `ostream`. To znamená, že funkce vrátí referenci na objekt třídy `ostream`. Protože program předá funkci nejdříve referenci na objekt, výsledkem je, že návratovou hodnotou funkce je právě objekt, který jí byl předán. To znamená, že příkaz

```
cout << trip;
```

se změní na volání funkce:

```
operator<<(cout, trip);
```

A toto volání vrátí objekt cout. Nyní tedy následující příkaz fungovat bude:

```
cout << "Cas vyletu: " << trip << "(utery)\n"; // funguje
```

Abychom viděli, jak tento příkaz funguje, rozebereme ho odděleně po krocích. Nejdříve příkaz

```
cout << " Cas vyletu: "
```

vyvolá konkrétní definici operátoru `<<` třídy `ostream`, která zobrazí řetězec a vrátí objekt třídy `ostream`. Výraz `cout << " Cas vyletu: "` tedy zobrazí řetězec a potom je nahrazen svou návratovou hodnotou, objektem `cout`. Tím se původní příkaz zkrátí na:

```
cout << trip << "(utery)\n";
```

Dále program pomocí deklarace operátoru << třídy `Time` zobrazí hodnoty proměnné trip a opět vrátí objekt cout. Tím se příkaz zkrátí na:

```
cout << "(utery)\n";
```

Nakonec program použije definici operátoru << třídy `ostream` pro řetězce a zobrazí konečný řetězec.

Pamatujte

Chcete-li přetížit operátor << tak, aby zobrazil objekt třídy *c_name*, použijte spřátelenou funkci s definicí v tomto tvaru:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
os << ... ; // zobrazí obsah objektu
        return os;
}
```

Ve výpisu 10.8 obsahuje upravená definice třídy dvě spřátelené funkce `operator*()` a `operator<<()`. První implementuje jako funkci vloženou, protože její kód je krátký. (Pokud je definice současně prototypem jako v tomto případě, použijete prefix `friend`.)

Pamatujte

Klíčové slovo `friend` použijete pouze u prototypu v deklaraci třídy. V definici funkce je použijete pouze v případě, že je tato definice současně prototypem.

Výpis 10.8 mytime3.h

```
//mytime3.h - Třída Time se spřátelenými funkcemi
#ifndef _MYTIME3_H_
#define _MYTIME3_H_
#include <iostream>
using namespace std;
class Time
{
private:
   int hours;
   int minutes;
public:
   Time();
   Time(int h, int m = 0);
   void AddMin(int m);
   void AddHr(int h);
   void Reset(int h = 0, int m = 0);
   Time operator+(const Time & t) const;
   Time operator-(const Time & t) const;
   Time operator*(double mult) const;
   friend Time operator*(double m, const Time & t)
                  { return t * m; }              // definice vložené
        funkce
   friend ostream & operator<<(ostream & os, const Time & t);
};
#endif
```

Výpis 10.9 mytime3.cpp

```
// mytime3.cpp - implementace metod třídy Time
#include "mytime3.h"
Time::Time()
{
  hours = minutes = 0;
}
Time::Time(int h, int m)
{
  hours = h;
  minutes = m;
}
void Time::AddMin(int m)
{
  minutes += m;
  hours += minutes / 60;
  minutes %= 60;
}
void Time::AddHr(int h)
{
  hours += h;
}
void Time::Reset(int h, int m)
{
  hours = h;
  minutes = m;
}
Time Time::operator+(const Time & t) const
{
  Time sum;
  sum.minutes = minutes + t.minutes;
  sum.hours = hours + t. hours + sum.minutes / 60;
  sum.minutes %= 60;
  return sum;
}
Time Time::operator-( const Time & t) const
{
  Time diff;
  int tot1, tot2;
  tot1 = t.minutes + 60 * t.hours;
  tot2 = minutes + 60 * hours;
  diff.minutes = (tot2 - tot1) % 60;
  diff.hours = (tot2 - tot1) / 60;
  return diff;
}
Time Time::operator*(double mult) const
{
  Time result;
  long totalminutes = hours * mult * 60 + minutes * mult;
  result.hours = totalminutes / 60;
```

```
    result.minutes = totalminutes % 60;
    return result;
}
ostream & operator<<(ostream & os, const Time & t)
{
    os << t.hours << "hodin, " << t.minutes << " minut";
    return os;
}
```

Výpis 10.10 usetime3.cpp

```
// usetime3.cpp - použití čtvrtého návrhu třídy Time
// společná kompilace souborů usetime3.cpp a mytime3.cpp
#include <iostream>
#include "mytime3.h"
using namespace std;
int main()
{
    Time A;
    Time B(5, 40);
    Time C(2, 55);
    cout << A << "; " << B << ": " << C << endl;
    A = B + C;                    // operator+()
    cout << A << endl;
    A = B * 2.75;            // operator*()
    cout << A << endl;
    cout << 10 * B << endl;
    return 0;
}
```

Zde je výstup:

```
0 hodin, 0 minut: 5 hodin, 40 minut: 2 hodin, 55 minut
8 hodin, 35 minut
15 hodin, 35 minut
56 hodin, 40 minut
```

Přetížené operátory: členské funkce a nečlenské funkce

U mnoha operátorů můžete při implementaci jejich přetížení volit mezi členskými a nečlenskými funkcemi. Ve verzi s nečlenskou funkcí se běžně používá spřátelená funkce, která tedy má přímý přístup k soukromým datům třídy. Uvažujte například operátor sčítání třídy `Time`. V deklaraci třídy je následující prototyp:

```
Time operator+(const Time & t) const;     // verze s členskou funkcí
```

Třída by místo toho musela použít následující prototyp:

```
friend Time operator+(const Time & t1, const Time & t2);
// verze s nečlenskou funkcí
```

Operátor sčítání potřebuje dva operandy. Ve verzi s členskou funkcí je jeden předán implicitně pomocí ukazatele this a druhý explicitně jako parametr funkce. Ve verzi se spřátelenou funkcí jsou oba operandy předány jako parametry.

Pamatujte

Ve funkcích přetěžujících stejný operátor potřebuje verze s nečlenskou funkcí o jeden parametr více než verze s funkcí členskou.

Oba tyto prototypy srovnávají výraz `B + C`, kde `B` i `C` jsou objekty typu `Time`. To znamená, že kompilátor může příkaz

```
A = B + C;
```

převést na kterýkoli z následujících příkazů:

```
A = B.operator+(C);      // členská funkce
A = operator+(B, C);     // nečlenská funkce
```

Pamatujte, že při definování daného operátoru musíte vybrat jeden či druhý tvar, ale nikoli oba. Vzhledem k tomu, že oba tvary jsou rovny stejnému výrazu, je definice obou tvarů považována za chybu z důvodu nejednoznačnosti.

Který tvar je tedy lépe použít? Jak již bylo dříve řečeno, u některých operátorů lze použít pouze členskou funkci. Jinak v tom velký rozdíl není. Na základě návrhu třídy je někdy výhodnější verze s nečlenskou funkcí, zvláště pokud jste ve třídě definovali konverze typů. Příklad uvidíte později v této kapitole.

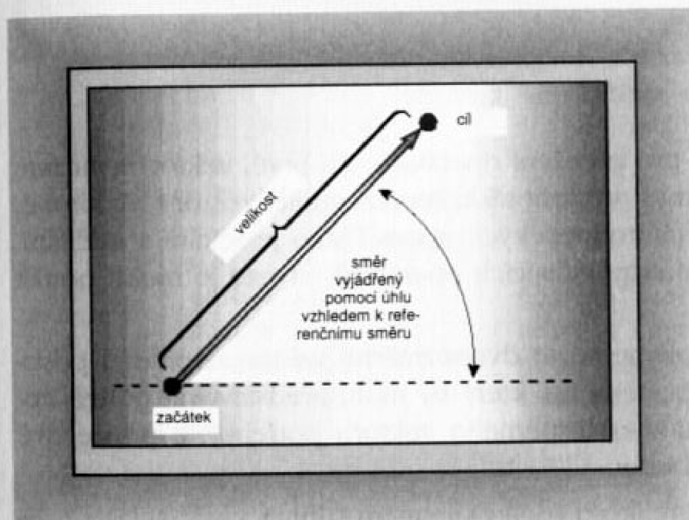
Další přetěžování – třída Vector

Podívejme se na další návrh třídy používající přetížení a spřátelené funkce – na třídu představující vektory. Tato třída bude ilustrovat další aspekty návrhu třídy, jako například začlenění dvou rozdílných způsobů popisujících stejnou věc do třídy. I když vás vektory nezajímají, budete moci mnohé nové techniky využít v jiných kontextech. *Vektor*, termín používaný v technice a fyzice, je veličina mající velikost i směr. Jestliže například na něco tlačíte, závisí účinek na síle a směru tlaku. Tlak jedním směrem může zabránit pádu kymácející se vázy, zatímco tlak opačným směrem může její zkázu urychlit. Chcete-li úplně popsat pohyb auta, měli byste uvést rychlost (velikost) i směr; argumentovat dálniční hlídce tím, že jste jel nižší než povolenou rychlostí bude mít malou váhu, jestliže jste jel v protisměru. (Imunologové a počítačoví odborníci používají termín vektor asi v jiném významu; zatím je ignorujte, alespoň do kapitoly 15, která se zabývá verzí používanou v počítačové vědě.) V následující poznámce se o vektorech dozvíte více, ale jejich úplné pochopení není z hlediska aspektů jazyka C++ v následujících příkladech nutné.

Vektory

Jste včelou dělnicí a objevili jste skrýš skvělého nektaru. Spěcháte zpět do úlu a oznámíte, že jste 120 metrů odsud našli nektar. „To je málo informací" zabzučí ostatní včely. Musíš nám také říct směr!" Vy odpovíte „Je to 30 stupňů severně od slunce." S informacemi o vzdálenosti (velikost) a směru spěchají ostatní včely na ono sladké místo. Včely znají vektory.

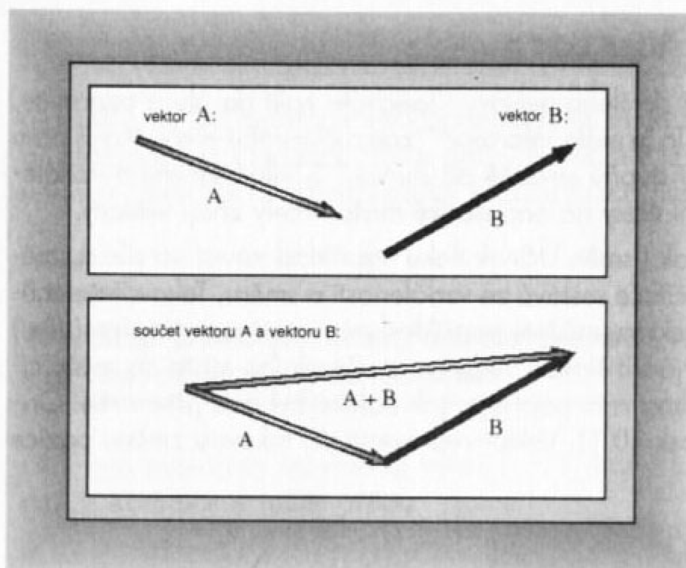
Mnoho veličin obsahuje jak velikost, tak i směr. Účinek tlaku například závisí na síle a směru. Přesunutí objektu na obrazovce počítače sestává ze vzdálenosti a směru. Takové věci můžete popsat pomocí vektorů. Pomocí vektoru můžete například popsat přesunutí (přemístění) objektu na obrazovce – můžete ho vyjádřit šipkou vedenou z původního místa do místa cílového. Délka vektoru udává jeho velikost a ta popisuje, jak daleko byl bod přemístěn. Orientace šipky popisuje směr (viz obrázek 10.1). Vektor reprezentující takovou změnu pozice se nazývá *vektor posunutí*.



Obrázek 10.1. Popis posunutí pomocí vektoru

Nyní jste Lhanappa, velký lovec mamutů. Zvědové hlásí stádo mamutů 14,1 kilometrů na severozápad. Ale protože vane jihovýchodní vítr, nechcete se přiblížit z jihovýchodu. Proto jdete 10 kilometrů na západ, potom 10 kilometrů na sever a blížíte se ke stádu z jihu. Víte, že tyto dva vektory posunutí vás dovedou na stejné místo jako jeden vektor dlouhý 14,1 kilometru ukazující na severozápad. Lhanappa, velký lovec mamutů, také umí sčítat vektory.

Součet dvou vektorů lze snadno interpretovat geometricky. Nejdříve nakreslíte jeden vektor, potom nakreslíte druhý vektor s počátkem na konci šipky prvního vektoru a nakonec nakreslíte vektor z počátečního bodu prvního vektoru do koncového bodu druhého vektoru. Tento třetí vektor představuje součet prvních dvou (viz obrázek 10.2). Všimněte si, že délka součtu může být menší než součet jednotlivých délek.

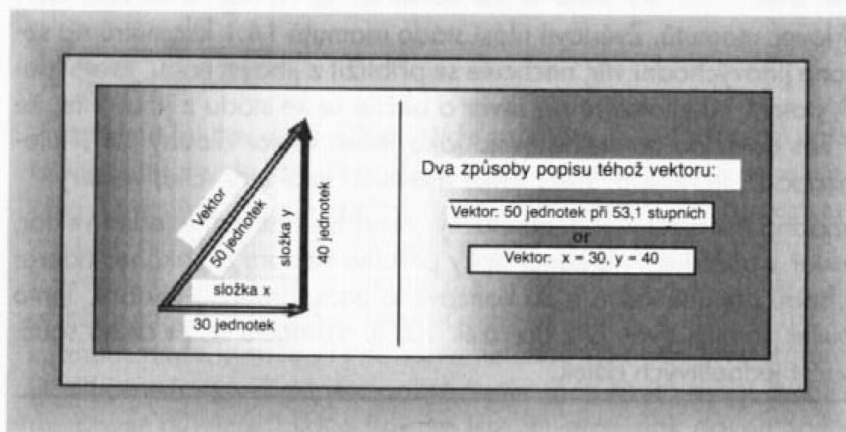


Obrázek 10.2. Sčítání dvou vektorů

Vektory jsou přirozeným kandidátem pro přetížení operátorů. Za prvé, vektor nemůžete znázornit jediným číslem a má tedy smysl vytvořit třídu reprezentující vektory. Za druhé, práce s vektory je obdobou obyčejných aritmetických operací jako je sčítání a odčítání. Tato paralela přímo vybízí k přetížení odpovídajících operátorů, abyste je mohli použít s vektory.

Z důvodu jednoduchosti budeme implementovat dvojrozměrný vektor, například posunutí obrazovky, namísto vektoru trojrozměrného, který by mohl představovat pohyb helikoptéry nebo gymnasty. K vyjádření dvojrozměrného vektoru potřebujete pouze dvě čísla a můžete si vybrat, která dvě to budou:

- můžete ho popsat pomocí jeho velikosti (délky) a směru (úhlu).
- můžete ho vyjádřit pomocí jeho složek x a y.



Obrázek 10.3. Složky x a y u vektoru

Složky tvoří horizontální vektor (složka x) a vertikální vektor (složka y). Jejich součet udává konečný vektor. Pohyb můžete například popsat jako posunutí bodu o 30 jednotek

doprava a 40 jednotek nahoru (viz obrázek 10.3). Tento pohyb umístí bod na stejné místo jako posunutí o 50 jednotek při úhlu 53,1 stupňů od horizontu. Vektor o velikosti 50 a s úhlem 53,1 stupňů je tedy roven vektoru s horizontální složkou velikosti 30 jednotek a vertikální složkou velikosti 40 jednotek. Při přemísťování vektorů je důležité kde začínáte a kde končíte, kudy se tam dostanete není důležité. Tato volba reprezentace je v podstatě totéž, čím jsme se zabývali v kapitole 7 u programu provádějícího konverzi mezi pravoúhlým a polárním systémem souřadnic.

Někdy je vhodnější jeden tvar, jindy zase druhý. Můžete tedy do popisu třídy začlenit obě reprezentace. Přečtěte si poznámku o vícenásobných reprezentacích a třídách, která následuje v dalším textu. Třídu rovněž navrhneme tak, aby při změně jedné reprezentace vektoru automaticky aktualizovala reprezentaci druhou. Možnost zabudovat takovou inteligenci do objektu je dalším kladem jazyka C++. Výpis 10.11 představuje deklaraci třídy.

Výpis 10.11. vector.h

```
// vector.h - Třída Vector s operátorem << a režimem stavu (mode)
#ifndef _VECTOR_H_
#define _VECTOR_H_
class Vector
{
private:
    double x;           // hodnota horizontální souřadnice
    double y;           // hodnota vertikální souřadnice
    double mag;         // délka vektoru
    double ang;         // směr vektoru
  char mode;            // 'r' = pravoúhlý (rectangular), 'p' = polární
                        //(polar)
// soukromé metody pro nastavení hodnot
    void set_mag();
    void set_ang();
    void set_x();
    void set_y();
public:
    Vector();
    Vector(double n1, double n2, char form = 'r');
    void set(double n1, double n2, char form = 'r');
    ~Vector();
    double xval() const {return x;}     // vrací hodnotu x
    double yval() const {return y;}     // vrací hodnotu y
    double magval() const {return mag;}// vrací velikost
    double angval() const {return ang;}// vrací úhel
    void polar_mode();                  // nastaví režim na 'p'
    void rect_mode();                          // nastaví režim na 'r'
// funkce pro přetížení operátorů
    Vector operator+(const Vector & b) const;
    Vector operator-(const Vector & b) const;
    Vector operator-() const;
    Vector operator*(double n) const;
// spřátelené funkce
```

```
    friend Vector operator*(double n, const Vector & a);
    friend ostream& operator<<(ostream& os, const Vector & v);
};
#endif
```

Všimněte si, že čtyři funkce vracející hodnoty složek jsou definovány v deklaraci třídy. To z nich automaticky činí funkce vložené. Skutečnost, že jsou tak krátké, z nich dělá skvělé kandidáty pro vkládání. Žádná z nich by neměla měnit data objektu, proto jsou deklarovány pomocí modifikátoru `const`. Jak si možná vzpomínáte z kapitoly 9, jedná se o syntaxi deklarace funkce, která objekt, k němuž má implicitní přístup, nezmění.

Ve výpisu 10.12 jsou všechny metody a spřátelené funkce deklarované ve výpisu 10.11. Všimněte si, jak konstruktory funkcí a všechny funkce `set()` nastavují pravoúhlé i polární reprezentace vektoru. Obě sady hodnot jsou tedy v případě potřeby ihned k dispozici bez dalšího výpočtu. Jak bylo také uvedeno v kapitolách 4 a 7, zabudované matematické funkce jazyka C++ pracují s úhly v radiánech, takže tyto funkce zabudovaly do metod převod na stupně a ze stupňů. Implementace takových věcí, jako převod z polárních souřadnic na pravoúhlé nebo převod radiánů na stupně, před uživatelem skrývá. Uživateli stačí vědět, že třída pracuje s úhly ve stupních a že vektor zpřístupňuje ve dvou rovnocenných reprezentacích.

Tato návrhářská rozhodnutí se řídí tradicemi OOP, kde se rozhraní třídy soustředí na podstatné věci (abstraktní model) a detaily skrývá. Při používání třídy `Vector` můžete tedy přemýšlet o všeobecných vlastnostech vektoru, jako že mohou reprezentovat přemístění a že můžete dva vektory sečíst. Ať již vyjadřujete vektor zápisem složek nebo velikostí, zápis směru je druhořadý, neboť pro nastavení a zobrazení hodnot vektoru můžete použít ten tvar, který je momentálně výhodnější. Na některé vlastnosti se podíváme za chvíli podrobněji.

Kompatibilita:

Některé systémy mohou stále používat hlavičkový soubor `math.h` místo `cmath`. Některé systémy C++ rovněž automaticky neprohledávají knihovnu matematických funkcí, například v některých systémech UNIX musíte zadat

```
$ CC source_file(s) -lm
```

Volba `-lm` říká linkeru, že má prohledat knihovnu matematických funkcí. Když tedy nakonec pomocí třídy `Vector` programy zkompilujte a dostanete zprávu o nedefinovaných externích proměnných, zkuste volbu `-lm` nebo zkontrolujte, zda systém nemá nějaký podobný požadavek.

Výpis 10.6. vector10.12.cpp

```
// vector.cpp - metody třídy Vector
#include <iostream>
#include <cmath>
using namespace std;
#include "vector.h"
```

```
const double Rad_to_deg = 57.2957795130823;
// privátní metody
// vypočítá velikost podle složek x a y
void Vector::set_mag()
{
    mag = sqrt(x * x + y * y);
}
void Vector::set_ang()
{
    if (x == 0.0 && y == 0.0)
      ang = 0.0;
    else
      ang = atan2(y, x);
}
// nastaví položku x podle polárních souřadnic
void Vector::set_x()
{
    x = mag * cos(ang);
}
// nastaví položku y podle polárních souřadnic
void Vector::set_y()
{
    y = mag * sin(ang);
}
// veřejné metody
Vector::Vector()          // implicitní konstruktor
{
    x = y = mag = ang = 0.0;
    mode = 'r';
}
// jestliže form je 'r', vytvoří vektor z pravoúhlých souřadnic
// (implicitně), jinak z polárních souřadnic, pokud form je 'p'
Vector::Vector(double n1, double n2, char form)
{
    mode = form;
    if (form == 'r')
    {
         x = n1;
         y = n2;
         set_mag();
         set_ang();
    }
    else if (form == 'p')
    {
         mag = n1;
         ang = n2 / Rad_to_deg;
         set_x();
         set_y();
    }
    else
    {
```

```
            cout << " Nespravny 3. parametr pro metodu Vector() - -";
            cout << "vektor nastaven na hodnotu 0\n";
            x = y = mag = ang = 0.0;
            mode = 'r';
        }
}
// jestliže form je 'r', vytvoří vektor z pravoúhlých souřadnic
// (implicitně), jinak z polárních souřadnic, pokud form je 'p'
void Vector:: set(double n1, double n2, char form)
{
    mode = form;
    if (form == 'r')
    {
        x = n1;
        y = n2;
        set_mag();
        set_ang();
    }
    else if (form == 'p')
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        cout << "Nespravny 3. parametr pro metodu Vector() - -";
        cout << "vektor nastaven na hodnotu 0\n";
        x = y = mag = ang = 0.0;
        mode = 'r';
    }
}
Vector::~Vector()              // destruktor
{
}
void Vector::polar_mode()  // nastaví režim polárních souřadnic
{
     mode = 'p';
}
void Vector::rect_mode() // nastaví mód pro použití pravoúhlých souřadnic
{
    mode = 'r';
}
// přetěžování operátorů
// sčítání dvou vektorů
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}
// odečítání vektoru b od vektoru a
```

```
Vector Vector::operator-(const Vector & b) const
{
    return Vector(x - b.x, y - b.y);
}
// změna znaménka vektoru
Vector Vector::operator-() const
{
    return Vector(-x, -y);
}
// násobení vektoru číslem n
Vector Vector::operator*(double n) const
{
    return Vector(n * x, n * y);:
}
// spřátelené metody
// násobení čísla n vektorem a
Vector operator*(double n, const Vector & a)
{
    return a * n;
}
// jestliže mode je 'r', zobrazí pravoúhlé souřadnice,
// jinak, pokud mode je 'p', zobrazí souřadnice polární
ostream& operator<<(ostream & os, const Vector  & v)
{
    if (v.mode == 'r')
        os << "(x,y) = (" << v.x << ", " << v.y << ")";
    else if (v.mode == 'p')
    {
        os << "(m,a) = (" << v.mag << ", "
            << v.ang * Rad_to_deg << ")";
    }
    else
        os << "Neplatny rezim objektu Vector";
    return os;
}
```

Použití stavové položky

Třída ukládá pravoúhlé i polární souřadnice vektoru. Pomocí členské proměnné `mode` řídí, který typ konstruktoru, metody `set()` a přetížené funkce `operator<<()` se použije – 'r' představuje pravoúhlý režim (je implicitní) a 'p' režim polární. Takový člen se nazývá *stavová položka*, protože popisuje stav, ve kterém se objekt nachází. Abyste zjistili, co to znamená, podívejte se na kód konstruktoru:

```
Vector::Vector(double n1, double n2, char form)
{
    mode = form;
     if (form == 'r')
    {
        x = n1;
```

```
        y = n2;
        set_mag();
        set_ang();
    }
    else if (form == 'p')
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        cout << " Nespravny 3. parametr pro metodu Vector() - -";
        cout << "vetor nastaven na hodnotu 0\n";
        x = y = mag = ang = 0.0;
        mode = 'r';
    }
}
```

Jestliže je třetí parametr 'r' nebo je vynechán (prototyp přiřadí implicitní hodnotu `'r'`), interpretují se vstupy jako pravoúhlé souřadnice, zatímco hodnota `'p'` způsobí, že budou interpretovány jako souřadnice polární:

```
Vector folly(3.0, 4.0);          // nastaví x = 3, y = 4
Vector foolery(20.0, 30.0);      // nastaví mag = 20, ang = 30
```

Všimněte si, že dodáte-li hodnoty proměnných x a y, nastaví konstruktor hodnoty velikosti a úhlu pomocí soukromých metod `set_mag()` a `set_ang()`, zatímco dodáte-li hodnoty pro velikost a úhel, nastaví hodnoty x a y pomocí soukromých metod `set_x()` a `set_y()`. Také si všimněte, že konstruktor pošle varovnou zprávu a stav nastaví na `'r'`, jestliže zadáte něco jiného než `'r'`nebo `'p'`.

Podobným způsobem funkce `operator<<()` určí pomocí proměnné mode, jakým způsobem má hodnoty zobrazit:

```
// jestliže mode je 'r', zobrazí pravoúhlé souřadnice,
// jinak, pokud mode je 'p', zobrazí souřadnice polární
ostream& operator<<(ostream & os, const Vector   & v)
{
    if (v.mode == 'r')
        os << "(x,y) = (" << v.x << ", " << v.y << ")";
    else if (v.mode == 'p')
    {
        os << "(m,a) = (" << v.mag << ", "
            << v.ang * Rad_to_deg << ")";
    }
    else
        os << "Neplatny rezim objektu Vector";
    return os;
}
```

Různé metody umožňující nastavit proměnnou mode pečlivě hlídají, aby platné hodnoty byly pouze 'p' nebo 'r', takže na poslední else v této funkci nikdy nedojde. Přesto však se taková kontrola často hodí – může pomoci zachytit jinou nejasnou chybu v programu.

Vícenásobné reprezentace a třídy

Případy veličin s různými ale rovnocennými reprezentacemi jsou běžné. Spotřebu benzínu můžete například měřit v mílích na galon, jak je tomu v USA, nebo v litrech na 100 kilometrů jako v Evropě. Číslo můžete vyjádřit formou řetězce nebo čísla a inteligenci pomocí IQ nebo v kilotupcích. Třídy přijdou velmi vhod, jestliže potřebujete do jediného objektu zahrnout různé aspekty a reprezentace nějaké entity. Za prvé, můžete do jednoho objektu uložit více reprezentací a, za druhé, můžete napsat funkce třídy, které po přiřazení hodnot jedné reprezentaci přiřadí automaticky hodnoty ostatním reprezentacím. Například metoda set_by_polar() třídy Vector nastavuje položky mag a ang na hodnoty parametrů funkce, ale zároveň nastavuje položky x a y. Díky interním konverzím můžete o veličině uvažovat podle její základní povahy a ne podle reprezentace.

Další přetěžování

Sčítání dvou vektorů pomocí souřadnic x, y je velmi jednoduché. Stačí přidat dvě složky x a získáte výslednou složku x a součet dvou složek y dá výslednou složku y. Tento popis vás může zlákat k použití následujícího kódu:

```
Vector Vector::operator+(const Vector & b) const
{
   Vector sum;
   sum.x = x + b.x;
   sum.y = y + b.y;
   return sum;
}
```

Takový kód by stačil, kdyby objekt ukládal pouze složky x a y. Tato verze však bohužel nedokáže nastavit hodnoty polárních souřadnic. Náprava by byla možná přidáním dalšího kódu:

```
Vector Vector::operator+(const Vector & b) const
{
   Vector sum;
   sum.x = x + b.x;
   sum.y = y + b.y;
   sum.set_ang(sum.x, sum.y);
   sum.set_mag(sum.x, sum.y);
   return sum;
}
```

Tuto práci však snadněji a spolehlivěji provede konstruktor:

```
Vector Vector::operator+(const Vector & b) const
{
```

```
    return Vector(x + b.x, y + b.y);        // vrátí vytvořený vektor
}
```

Tento kód předá konstruktoru `Vector` dvě nové hodnoty složek x a y. Konstruktor potom pomocí těchto hodnot vytvoří nový bezejmenný objekt a funkce vrátí kopii tohoto objektu. Tímto způsobem zaručíte, že se nový objekt vytvoří podle standardních pravidel stanovených v konstruktoru.

Tip

Jestliže má nějaká metoda vytvořit nový objekt, ověřte si, zda tuto práci nemůže udělat konstruktor. Tím si nejen ušetříte starosti, ale budete mít zajištěno, že se nový objekt vytvoří správně.

Násobení

Vizuálně představuje násobení vektoru určitým číslem prodloužení nebo zkrácení tohoto vektoru o hodnotu tohoto činitele. Vynásobíte-li tedy vektor číslem 3, získáte sice vektor třikrát tak dlouhý, ale ukazovat bude stále stejným směrem. Přeložit tento obraz na způsob, kterým reprezentuje vektor třída, je snadné. V systému polárních souřadnic vynásobíte velikost a úhel necháte, zatímco v systému pravoúhlých souřadnic provedete násobení vektoru číslem tak, že vynásobíte daným číslem obě složky x a y. To znamená, že pokud má vektor hodnoty složek 5 a 12 a vynásobíte ho číslem 3, budou hodnoty složek 15 a 36. A to právě dělá přetížený operátor násobení:

```
Vector operator*(double n) const
{
    return Vector(n * x, n * y);
}
```

Stejně jako v případě přetíženého operátoru sčítání vytvoří kód správný objekt `Vector` ze složek x a y. V tomto případě násobí hodnotu typu `Vector` hodnotou typu `double`. Stejně jako v příkladě s třídou `Time` můžeme i pro násobení použít vloženou spřátelenou funkci.

```
Vector operator*(double n, const Vector & a)      // spřátelená funkce
{
    return a * n;              // převede násobení double * Vector na Vector *
                               // double
}
```

Další vylepšení: přetížení přetíženého operátoru

V obyčejném jazyce C++ má operátor '-' již dva významy. Za prvé, při použití se dvěma operandy představuje operátor odčítání. Operátor odčítání se označuje *binární operátor*, protože má přesně dva operandy. Za druhé, při použití s jedním operandem, například `-x`, představuje znaménko mínus. Tomuto tvaru se říká *unární operátor*, což znamená, že má přesně jeden operand. Obě operace – odčítání a změna znaménka – mají smysl i u vektorů, proto má obě i třída `Vector`.

Chcete-li odečíst vektor B od vektoru A, odečtete pouze složky, takže definice přetížení odčítání je dosti podobná definici sčítání:

```
Vector operator-(const Vector & b) const;          // prototyp
Vector Vector::operator-(const Vector & b) const  // definice
{
        return Vector(x - b.x, y - b.y); // vrátí vytvořený vektor
}
```

Zde je správné pořadí operandů důležité. Uvažujte následující příkaz:

```
diff = v1 - v2;
```

Tento příkaz se převede na volání členské funkce:

```
diff = v1.operator-(v2);
```

Tento příkaz znamená, že vektor předaný jako explicitní parametr bude odečten od implicitního vektoru, takže bychom měli použít `x - b.x`, a ne `b.x - x`.

Dále uvažujte unární operátor mínus, který přijímá pouze jeden operand. Použijete-li tento operátor na běžné číslo, například -x, změníte znaménko hodnoty. Použijete-li tedy tento operátor na vektor, změníte znaménka všech jeho složek. Přesněji řečeno, funkce by měla vrátit nový vektor, který bude mít opačné znaménko než vektor původní. (V systému polárních souřadnic ponechá negace velikost nezměněnou, ale obrátí směr. Tuto operaci intuitivně zvládá mnoho politiků s minimálním matematickým vzděláním.) Zde je prototyp a definice pro přetížení negace:

```
Vector operator-() const;
```

Vector Vector::operator-() const

```
{
        return Vector (-x, -y);
}
```

Všimněte si, že funkce `operator-()` má nyní dvě oddělené definice. To je v pořádku, protože každá definice má jinou signaturu. Můžete definovat binární i unární verzi operátoru -, protože jazyk C++ obě tyto verze poskytuje. Operátor mající pouze binární tvar, například dělení (/), lze přetížit pouze jako binární operátor.

Pamatujte

Protože přetěžování operátorů je implementováno pomocí funkcí, můžete stejný operátor přetížit mnohokrát, jestliže má každá funkce operátoru odlišnou signaturu a stejný počet operandů jako odpovídající zabudovaný operátor jazyka C++.

Komentář k implementaci

Tato implementace ukládá pravoúhlé i polární souřadnice vektoru v objektu. Veřejné rozhraní však na této skutečnosti nezávisí. Rozhraní pouze požaduje, aby bylo možné zobrazit obě reprezentace a aby byly vráceny jednotlivé hodnoty. Vnitřní implementace

by mohla být úplně jiná. Objekt by mohl například ukládat pouze složky x a y. Potom by třeba metoda magval(), vracející velikost vektoru, mohla vypočítat velikost podle hodnot x a y a nemusela by hledat hodnoty uložené v objektu. Takový přístup změní implementaci, ale uživatelské rozhraní zůstane nezměněno. Toto oddělení rozhraní od implementace je jedním z cílů OOP. Umožní vám doladit implementaci, aniž byste měnili kód v programech používajících tuto třídu.

Obě tyto implementace mají své výhody a nevýhody. Uložení dat znamená, že objekt zabírá více paměti a že kód musí při každé změně objektu třídy Vector aktualizovat pravoúhlé a polární reprezentace opatrně. Vyhledání dat je však rychlejší. Jestliže aplikace musí často přistupovat k oběma reprezentacím vektoru, je implementace použitá v uvedeném příkladě výhodnější. Pokud jsou polární data potřeba jen někdy, je lepší druhá implementace. V jednom programu byste mohli použít první implementaci, v jiném druhou, a uživatelské rozhraní přesto může zůstat stejné pro obě.

Použití třídy Vector na problém náhodné chůze

Ve výpisu 10.13 je krátký program používající naši upravenou třídu. Simuluje proslulý problém Opilcovy chůze. Avšak nyní, kdy je opilý považován spíše za někoho s vážným zdravotním problémem než za zdroj pobavení, se úloha nazývá Problém náhodné chůze. Myšlenka spočívá v tom, že někoho postavíte ke sloupu lampy. Subjekt se začne pohybovat, ale směr každého kroku se vzhledem k předchozímu kroku náhodně mění. Jednou z možností, jak tento problém formulovat je otázka: kolik kroků bude náhodný chodec potřebovat, aby ušel tak 15 metrů od lampy? Z hlediska vektorů se jedná o sčítání skupiny náhodně orientovaných vektorů, dokud součet nepřesáhne hodnotu 15 metrů.

V programu na výpisu 10.13 si můžete zvolit cílovou vzdálenost, kterou bude třeba ujít, a délku chodcova kroku. Program uchovává mezisoučet reprezentující pozici po každém kroku (reprezentována jako vektor) a hlásí počet kroků potřebných k dosažení cílové vzdálenosti společně s pozicí chodce (v obou formátech). Jak uvidíte, chodec postupuje naprosto neefektivně. Cesta z tisíce kroků, z nichž každý je dlouhý 60 cm, dovede chodce pouze do vzdálenosti 15 metrů od výchozího bodu. Program vydělí čistou ušlou vzdálenost (v tomto případě 15 metrů) počtem kroků a získá míru chodcovy neefektivnosti. Zásluhou všech náhodných směrů je tento průměr mnohem menší než délka jediného kroku. Náhodný směr program vybírá pomocí funkcí rand(), srand() a time() ze standardní knihovny, které jsou popsány v poznámkách k programu. Nezapomeňte zkompilovat kód z výpisu 10.13 společně s kódem na výpisu 10.12.

Výpis 10.13. randwalk.cpp

```
// randwalk.cpp - použití třídy Vector
// kompilovat společně s vector.cpp
#include <iostream>
#include <cstdlib>            // prototypy funkcí rand(), srand()
#include <ctime>              // prototyp funkce time()
using namespace std;
```

```
#include "vector.h"
int main()
{
    srand(time(0));          // nastaví generátor náhodných čísel
    double direction;
    Vector step;
    Vector result(0.0, 0.0);
    unsigned long steps = 0;
    double target;
    double dstep;
    cout << "Zadejte cilovou vzdalenost (k pro ukonceni): ";
    while (cin >> target)
    {
        cout << "Zadejte delku kroku: ";
        if (!(cin >> dstep))
            break;
        while (result.magval() < target)
        {
            direction = rand() % 360;
            step.set(dstep, direction, 'p');
            result = result + step;
            steps++;
        }
        cout << "Po " << steps << " krocich se subjekt "
            << "nachazi na nasledujicim pozici:\n";
        cout << result << "\n";
        result.polar_mode();
        cout << " nebo\n" << result << "\n";
        cout << "Prumerna vzdalenost na jeden krok = "
            << result.magval()/steps << "\n";
        steps = 0;
        result.set(0.0, 0.0);
        cout << "Zadejte cilovou vzdalenost (k pro ukonceni): ";
    }
    cout << "Nashledanou!\n";
    return 0;
}
```

Kompatibilita:

Místo hlavičkového souboru `stdlib.h` byste mohli zadat `cstdlib` a místo hlavičkového souboru `time.h` soubor `ctime`.

Příklad běhu programu:

```
Zadejte cilovou vzdalenost (k pro ukonceni): 15
Zadejte delku kroku: 60
Po 253 krocich se subjekt nachází na nasledujici pozici:
(x,y) = (46.1512, 20.4902)
 nebo
```

```
(m,a) = (50.495, 23.9402)
Prumerna vzdalenost na jeden krok = 0.399174
Zadejte cilovou vzdalenost (k pro ukonceni): 15
Zadejte delku kroku: 60
Po 951 krocich se subjekt nachází na nasledujici pozici:
(x,y) = (-21.9577, 45.3019)
 nebo
(m,a) = (50.3429, 115.8593)
Prumerna vzdalenost na jeden krok = 0.1058724
Zadejte cilovou vzdalenost (k pro ukonceni): 15
Zadejte delku kroku: 30
Po 1716 krocich se subjekt nachází na nasledujici pozici:
(x,y) = (40.0164, 31.1244)
 nebo
(m,a) = (50.6956, 37.8755)
Prumerna vzdalenost na jeden krok = 0.0590858
Zadejte cilovou vzdalenost (k pro ukonceni): k
```

Díky náhodné povaze procesu vznikají případ od případu dosti rozdílné výsledky, přestože počáteční podmínky jsou stejné. Zmenšíte-li však délku kroku na polovinu, počet kroků potřebných ke zdolání stanovené vzdálenosti vzroste v průměru čtyřikrát. Podle teorie pravděpodobnosti je v průměru počet kroků (*N*) délky *s* potřebných ke zdolání čisté vzdálenosti *D* stanoven následující rovnicí:

$$N = (D/s)^2$$

Toto je pouze průměrná hodnota, jednotlivé výsledky se budou měnit případ od případu. Například tisíc případů chůze na vzdálenost 15 metrů při délce kroku 60 centimetrů vedlo k průměrné hodnotě 636 kroků (což se blíží průměrné teoretické hodnotě 625), ale jednotlivé hodnoty se pohybovaly v rozmezí od 91 až po 3951. Podobně tisíc pokusů o zdolání 50 metrů při délce kroku 30 centimetrů dalo průměrnou hodnotu 2557 kroků (blíží se teoretickému průměru 2500), s rozmezím od 345 do 10 882 kroků. Při náhodné chůzi tedy dělejte sebevědomě dlouhé kroky. Stále sice nebudete kontrolovat směr k cíli, ale dostanete se alespoň dále.

Poznámky k programu

Nejdříve se budeme bavit o náhodných číslech. Standardní knihovna ANSI C, která je také obsažena v C++, obsahuje funkci `rand()`, která vrací náhodné celé číslo v rozmezí od 0 po určitou hodnotu závislou na implementaci. Náš program získává hodnotu úhlu v rozmezí `0-359` pomocí operátoru modulo. Funkce `rand()` použije na počáteční nastavenou hodnotu určitý algoritmus a získá náhodnou hodnotu. Tato hodnota se použije jako počáteční při dalším volání funkce atd. Čísla jsou ve skutečnosti pseudonáhodná, neboť deset po sobě jdoucích volání vrátí většinou stejnou množinu náhodných čísel. (Přesné hodnoty závisí na implementaci.) Funkce `srand()` však dovoluje implicitně nastavenou hodnotu přepsat a začít tak jinou sekvenci náhodných čísel. Tento program používá k nastavení počáteční hodnoty návratovou hodnotu funkce `time(0)`. Funkce `time(0)` vrátí aktuální kalendářní čas, který je často implementován jako počet sekund od nějakého konkrétního data. (Obecněji se dá říci, že funkce `time()` přijímá jako parametr adresu proměnné typu

`time_t` a vloží čas do této proměnné a zároveň jej vrátí. Díky použití parametru s adresou 0 nevznikne potřeba jinak nepotřebné proměnné `time_t`.) Příkaz

```
srand(time(0));
```

tedy při každém spuštění programu nastaví jinou počáteční hodnotu. Díky tomu vypadá náhodný výstup ještě náhodněji. Hlavičkový soubor `cstdlib` (u dřívějších verzí `stdlib.h`) obsahuje prototypy funkcí `srand()` a `rand()`, zatímco `ctime` (dříve `time.h`) obsahuje prototyp funkce `time()`.

Program zaznamenává postup chodcovy chůze pomocí vektoru `result`. V každém cyklu ve vnitřní smyčce nastaví vektor `step` novým směrem a přičte jej k vektoru `result`. Pokud velikost vektoru `result` překročí cílovou vzdálenost, smyčka skončí.

Díky nastavení režimu vektoru program zobrazí konečnou pozici v pravoúhlých i v polárních souřadnicích.

Mimochodem, příkaz

```
result = result + step;
```

nastaví vektor `result` do režimu `'r'` bez ohledu na počáteční nastavení vektorů `result` a `step`. Zde je vysvětlení. Za prvé, funkce operátoru sčítání vytvoří a vrátí nový vektor, obsahující součet obou parametrů. Funkce vytvoří tento vektor pomocí implicitního konstruktoru, který vytváří vektory v režimu `'r'`. Vektor přiřazený vektoru `result` je tedy v režimu `'r'`. Standardně přiřazení přiřazuje každou členskou proměnnou zvlášť, takže hodnota `'r'` je přiřazena do `result.mode`. Pokud byste chtěli upřednostnit nějaké jiné chování, například aby si vektor `result` zachoval původní režim, můžete provést implicitní přiřazení tak, že nadefinujete operátor přiřazení třídy. Příští kapitola takové příklady obsahuje.

Automatické konverze a přetypování tříd

Dalším tématem v nabídce tříd je typová konverze. Podíváme se, jak C++ provádí převod do a z uživatelem definovaného typu. Nejdříve si zopakujme, jak C++ provádí konverze vestavěných typů. Jestliže provádíte přiřazení hodnoty standardního typu proměnné jiného standardního typu, C++ tuto hodnotu automaticky převede na stejný typ jako má přijímací proměnná za předpokladu, že oba typy jsou kompatibilní. Všechny následující příkazy generují číselnou konverzi typů:

```
long count = 8;     // převede hodnotu 8 typu int na typ long
double time = 11;   // převede hodnotu 11 typu int na typ double
int side = 3.33;    // převede hodnotu 3.33 typu double  na hodnotu 3
                    // typu int
```

Tato přiřazení fungují, protože C++ pozná, že všechny tyto rozdílné číselné typy reprezentují v základě totéž – číslo – a protože C++ obsahuje vestavěná pravidla pro provádění konverzí. Ale vzpomeňte si (četli jste o tom v kapitole 3), že při těchto konverzích může dojít ke ztrátě přesnosti. Přiřadíte-li například hodnotu 3.33 proměnné typu int, dostanete hodnotu 3 a část 0.33 ztratíte.

Jazyk C++ neprovádí automaticky konverzi z typů, které nejsou kompatibilní. Například příkaz

```
int * p = 10; // kolize typů
```

skončí chybou, protože na levé straně je ukazatel, zatímco na pravé straně je číslo. A přestože vnitřně počítač může reprezentovat adresu jako celé číslo, jsou celá čísla a ukazatele koncepčně zcela odlišná. Určitě byste například neumocňovali ukazatel. Pokud však automatická konverze nefunguje, můžete použít přetypování:

```
int * p = (int *) 10; // ok, p a (int *) 10 jsou ukazatele
```

Tento příkaz nastaví ukazatel na adresu 10 tím, že přetypuje číslo deset na ukazatel na celé číslo (tedy int *).

Je možné definovat třídu, která bude mít úzký vztah k některému základnímu typu nebo jiné třídě, a bude mít smysl provádět konverzi mezi takovými typy. V takovém případě můžete C++ sdělit, jak provádět takové konverze automaticky, případně pomocí přetypování. Abychom ukázali, jak to funguje, pozměníme program pounds-to-stone z kapitoly 3 do podoby třídy. Nejdříve navrhneme odpovídající typ. V zásadě reprezentujeme pouze jednu věc (váhu), ale dvěma způsoby (v librách a v kamenech). Třída skýtá dobrou příležitost začlenit obě tyto reprezentace do jednoho celku. Tudíž má smysl vložit obě reprezentace váhy do jedné třídy a vytvořit metody třídy vyjadřující váhu v různých formách. Ve výpisu 10.14 je hlavičkový soubor třídy.

Výpis 10.14. stonewt.h

```
// stonewt.h - definice třídy Stonewt
#ifndef _STONEWT_H_
#define _STONEWT_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};        // počet liber na jeden kámen
    int stone;                       // váha v celých kamenech
    double pds_left;                // zbytek váhy v librách
    double pounds;                   // celková váha v librách
public:
    Stonewt(double lbs);            // konstruktor typu double pro libry
    Stonewt(int stn, double lbs); // konstruktor pro kameny, libry
    Stonewt();                       // standardní konstruktor
    ~Stonewt();
    void show_lbs() const;          // zobrazení váhy v librách
    void show_stn() const;          // zobrazení váhy v kamenech
};
#endif
```

Všimněte si, že třída má tři konstruktory. Ty umožňují inicializovat objekt třídy `Stonewt` počtem liber nebo kombinací kamenů a liber v čísle s pohyblivou řádovou čárkou. Můžete také vytvořit objekt bez inicializace.

Třída má také dvě zobrazovací funkce. Jedna zobrazí váhu v librách a druhá v kamenech a librách. Na výpisu 10.15. je implementace metod této třídy. Všimněte si, že všechny konstruktory přiřadí hodnoty všem třem soukromým položkám. Při vytvoření objektu třídy Stonewt se tedy nastaví obě reprezentace váhy.

Jak již bylo zmíněno v kapitole 9, výčty poskytují vhodný způsob pro definice konstant uvnitř třídy za předpokladu, že se jedná o celá čísla.

Výpis 10.15. stonewt.cpp

```
#include <iostream>
using namespace std;
#include "stonewt.h"
// vytvoření objektu Stonewt z hodnoty typu double
Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;     // celočíselné dělení
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}
// vytvoření objektu Stonewt z hodnot stone, lbs
Stonewt::Stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds = stn * Lbs_per_stn +lbs;
}
Stonewt::Stonewt()          // bezparametrický konstruktor, wt = 0
{
    stone = pounds = pds_left = 0;
}
Stonewt::~Stonewt()         // destruktor
{
}
// zobrazí váhu v kamenech
void Stonewt::show_stn() const
{
    cout << stone << " kamenu, " <<   pds_left << " liber\n";
}
// zobrazí váhu v librách
void Stonewt::show_lbs() const
{
    cout << pounds << " liber\n";
}
```

Protože objekt Stonewt reprezentuje jedinou váhu, je rozumné najít způsob, jak převést celé nebo reálné číslo na objekt třídy Stonewt. A my jsme tak již učinili! V jazyce C++ se každý konstruktor s jedním parametrem chová jako funkce pro převod z typu parametru na typ třídy. Pak tedy konstruktor

```
Stonewt(double lbs);  // šablona pro konverzi typu double na typ Stonewt
```

slouží jako instrukce pro konverzi typu double na hodnotu typu Stonewt. To znamená, že můžete napsat následující kód:

```
Stonewt myCat;       // vytvoří objekt třídy Stonewt
myCat = 19.6;        // provede konverzi pomocí konstruktoru Stonewt(double)
```

Program vytvoří pomocí konstruktoru Stonewt(double) dočasný objekt třídy Stonewt s počáteční hodnotou 19.6. Přiřazení jednotlivých položek potom zkopíruje obsah dočasného objektu do objektu myCat. Tento proces se nazývá implicitní konverze, protože se tak děje automaticky bez potřeby explicitního přetypování.

Jako konverzní funkce lze použít pouze konstruktor s jedním parametrem. Konstruktor

```
Stonewt(int stn, double lbs);
```

má parametry dva a nelze ho tedy použít ke konverzi typů.

Existence konstruktoru jako funkce pro automatickou konverzi vypadá jako příjemná vlastnost. Ale jak programátoři získávají s jazykem C++ více zkušeností, zjistí, že automatický převod není vždy žádoucí, protože může vést k neočekávaným konverzím. Poslední implementace C++ tedy obsahují nové klíčové slovo explicit, které tento automatický aspekt vypne. Můžete tedy konstruktor deklarovat takto:

```
explicit Stonewt(double lbs);  // není povolena implicitní konverze
```

Ve výše uvedeném případě jsou tedy implicitní konverze vypnuty, explicitní jsou ale povoleny stále – tedy konverze používající explicitní přetypování:

```
Stonewt myCat;              // vytvoření objektu třídy Stonewt
myCat = 19.6;               // není povoleno, pokud je konstruktor
                            // Stonewt(double) deklarován jako explicit
mycat = Stonewt(19.6);      // ok, explicitní konverze
mycat = (Stonewt) 19.6;     // ok, starší forma explicitního přetypování
```

Pamatujte

V C++ konstruktor s jedním parametrem definuje konverzi typů z typu parametru na typ třídy. Jestliže je konstruktor blíže určen klíčovým slovem explicit, můžete ho použít pouze pro explicitní konverze. Bez tohoto slova ho lze použít i pro implicitní konverze.

Kdy kompilátor použije funkci stonewt(double)? Jestliže v deklaraci konstruktoru použijete klíčové slovo explicit, použije se konstruktor Stonewt(double) pouze pro explicitní přetypování. Jinak ho lze použít při implicitních konverzích v následujících případech:

- Při inicializaci objektu třídy Stonewt na hodnotu typu double.
- Při přiřazení hodnoty typu double objektu třídy Stonewt.
- Při předávání hodnoty typu double funkci očekávající parametr typu Stonewt.
- Jestliže funkce má podle deklarace vrátit hodnotu typu Stonewt a snaží se vrátit hodnotu typu double.
- Pokud je v předchozích případech použit vestavěný typ, který lze převést na typ double, aniž by došlo k nejednoznačnostem.

Podívejme se na poslední bod podrobněji. Proces kontroly typů parametrů, poskytovaný prototypem funkce umožní konstruktoru `Stonewt(double)` fungovat jako konvertor i pro ostatní číselné typy. To znamená, že oba následující příkazy budou fungovat, přičemž nejdříve dojde k převodu z typu `int` na typ `double` a až poté se použije náš konstruktor.

```
Stonewt Jumbo(7000);  // konverze z int do double pomocí Stonewt(double)
Jumbo = 7300;         // konverze z int do double Stonewt(double)
```

Tento dvojstupňový převod však funguje pouze tehdy, pokud nemůže dojít k dvojznačnostem. To znamená, že jestliže by byl v třídě definován také konstruktor `Stonewt(long)`, kompilátor by tyto příkazy odmítl a pravděpodobně by vypsal zprávu, že typ `int` lze převést jak na typ `long`, tak na typ `double`. Volání je nejednoznačné.

Ve výpisu 10.16. se pomocí konstruktorů tříd inicializují některé objekty třídy `Stonewt` a provádí se typová konverze. Nezapomeňte společně s výpisem 10.16 zkompilovat také výpis 10.15.

Výpis 10.16. stone.cpp

```
// stone.cpp — uživatelem definované konverze
  // zkompilujte společně s stonewt.cpp
  #include <iostream>
  using namespace std;
  #include "stonewt.h"
  void display(Stonewt st, int n);
  int main()
  {
      Stonewt pavarotti = 260;   // použití konstruktoru k inicializaci
      Stonewt wolfe((double)285.7);// stejné jako Stonewt wolfe = 285.7;
      Stonewt taft(21, 8);
    cout << "Tenor vazil ";
    pavarotti.show_stn();
    cout << "Detektiv vazil ";
    wolfe.show_stn();
    cout << "Prezident vazil ";
    taft.show_lbs();
    pavarotti = double(265.8);      // použití konstruktoru ke konverzi
    taft = 325;                       // stejné jako taft = Stonewt(325);
    cout << "Po veceri tenor vazil ";
    pavarotti.show_stn();
    cout << "Po veceri prezident vazil ";
    taft.show_lbs();
    display(taft, 2);
    cout << "Zapasnik vazil jeste vic.\n";
    display(422, 2);
    cout << "Zadny kamen nezbyl\n";
    return 0;
  }
    void display(Stonewt st, int n)
  {
      for (int i = 0; i < n; i++)
```

```
        {
            cout << "Uf! ";
            st.show_stn();
        }
    }
```

Zde je výstup:

```
Tenor vazil 18 kamenu, 8 liber
Detektiv vazil 20 kamenu, 5.7 liber
Prezident vazil 302 liber
Po veceri tenor vazil 18 kamenu, 13.8 liber
Po veceri prezident vazil 325 liber
Uf! 23 kamenu, 3 liber
Uf! 23 kamenu, 3 liber
Zapasnik vazil ještě vic.
Uf! 30 kamenu, 2 liber
Uf! 30 kamenu, 2 liber
Zadny kamen nezbyl
```

Poznámky k programu

Za prvé si všimněte, že pokud konstruktor má jeden parametr, můžete pro objekt třídy použít následující formu inicializace:

```
// syntaxe pro inicializaci objektu
// pomocí konstruktoru s jedním parametrem
Stonewt pavarotti = 260;
```

Tento příkaz je ekvivalentní ostatním dvěma zápisům, které jsme použili:

```
// standardní syntaktické formy inicializace objektů třídy
Stonewt pavarotti(260);
Stonewt pavarotti = Stonewt(260);
```

Poslední dvě formy lze použít i u konstruktorů s více parametry.

Dále si všimněte následujících dvou přiřazení z výpisu 10.16:

```
pavarotti = 265.8;
taft = 325;
```

První přiřazení používá konstruktor s parametrem typu `double` pro převedení čísla 265.8 na typ `Stonewt`. Nastaví položku pounds objektu `pavarotti` na hodnotu 265.8. Protože se tak děje pomocí konstruktoru, přiřazení také nastaví položky třídy `stone` a `pds_left`. Podobně i druhé přiřazení převede hodnotu typu int na typ `double` a potom použije konstruktor `Stonewt(double)` k nastavení všech tří hodnot položek.

Nakonec si všimněte tohoto volání funkce:

```
display(422, 2);    // převede 422 na typ double a poté na typ Stonewt
```

Prototyp funkce `display()` udává, že její první parametr by měl být objekt třídy `Stonewt`. Když kompilátor narazí na parametr typu `int`, hledá konstruktor `Stonewt(int)`, aby mohl pomocí něho převést typ `int` na požadovaný typ `Stonewt`. Když se mu to nepodaří, hle-

dá konstruktor s nějakým jiným vestavěným typem, na který může být typ int převeden. Konstruktor `Stonewt(double)` těmto požadavkům vyhovuje. Kompilátor tedy převede celé číslo na typ double a potom převede výsledek pomocí konstruktoru `Stonewt(double)` na objekt třídy `Stonewt`.

Konverzní funkce

Ve výpisu 10.16 se provádí konverze čísla na objekt třídy `Stonewt`. Je možná opačná konverze? Čili je možné převést objekt třídy `Stonewt` na hodnotu typu `double` jako v následujícím příkladu?

```
Stonewt wolfe(285.7);
double host = wolfe;  // ?? je to možné ??
```

Odpověď je ano, ale nikoli pomocí konstruktorů. Konstruktory lze použít pouze pro převod jiného typu na typ třídy. Chcete-li opačný postup, musíte použít speciální formát funkce operátoru C++ nazývaný *konverzní funkce*.

Konverzní funkce se podobají přetypování na uživatelem definovaný typ a můžete je použít stejným způsobem, jako byste použili běžné přetypování. Jestliže například nadefinujete konverzní funkci z typu `Stonewt` na typ `double`, můžete použít následující konverze:

```
Stonewt wolfe(285.7);
double host = double (wolfe);    // 1. syntaxe
double thinker = (double) wolfe; // 2. syntaxe
```

Nebo můžete nechat rozhodnutí na kompilátoru:

```
Stonewt wells(20, 3);
double star = wells;  // implicitní použití konverzní funkce
```

jakmile si počítač všimne, že na pravé straně je typ `Stonewt` a na levé typ double, podívá se, zda jste definovali konverzní funkci odpovídající danému popisu.

Jak tedy vytvoříte konverzní funkci? Pro konverzi na typ *názevTypu* použijte konverzní funkci v tomto formátu:

```
operator názevTypu();
```

Všimněte si následujících bodů:

- Konverzní funkce musí být metodou třídy.
- U konverzní funkce nesmí být uveden typ návratové hodnoty.
- Konverzní funkce nesmí mít žádné parametry.

Například funkce pro převod na typ double by měla tento prototyp:

```
operator double();
```

Část *názevTypu* udává konverzi typ, na který se má převádět, proto není potřeba žádný návratový typ. Skutečnost, že funkce je metodou třídy znamená, že musí být vyvolána pomocí nějakého objektu dané třídy a tento objekt funkci sdělí, kterou hodnotu má konvertovat. Funkce tedy nepotřebuje žádné parametry.

Chcete-li tedy přidat funkce, provádějící konverze objektů `stone_wt` na typy `int` a `double`, budete muset do deklarace třídy přidat následující prototypy:

```
operator int();
operator double();
```

Na výpisu 10.17 je upravená deklarace třídy.

Výpis 10.17. stonewt1.h

```
// stonewt1.h — upravená definice třídy Stonewt
#ifndef _STONEWT1_H_
#define _STONEWT1_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};        // počet liber na kámen
int stone;                          // počet celých kamenů
    double pds_left;                // zbytek váhy v librách
    double pounds;                  // celková váha v librách
public:
    Stonewt(double lbs);            // konstruktor pro libry
    Stonewt(int stn, double lbs);   // konstruktor pro kameny
    Stonewt();                      // implicitní konstruktor
    ~Stonewt();
    void show_lbs() const;          // zobrazení váhy v librách
    void show_stn() const;          // zobrazení váhy v kamenech
// konverzní funkce
    operator int() const;
    operator double() const;
};
#endif
```

Dále výpis 10.18 obsahuje definice těchto dvou konverzních funkcí. Tyto definice by měly být přidány do souboru obsahujícího členské funkce třídy. Všimněte si, že každá funkce vrací požadovanou hodnotu i přesto, že návratový typ není deklarován. Také si všimněte, že definice konverze na typ int hodnoty neořezává, ale zaokrouhlí je na nejbližší celé číslo. Pokud například položka pounds má hodnotu 114.4, pak součet `pounds` + 0.5 je 114.9 a přetypování `int(114.9)` dá hodnotu 114. Ale jestliže položka pounds obsahuje hodnotu 114.5, pak pounds + `0.5` dá hodnotu 115.1 a `int(115.1)` je 115.

Výpis 10.18. stonewt1.cpp

```
// stonewt1.cpp — metody třídy Stonewt + konverzní funkce  #include
<iostream>
using namespace std;
#include "stonewt1.h"
// následují předcházející definice
// konverzní funkce
Stonewt::operator int() const
```

```
{
    return int (pounds + 0.5);
}
Stonewt::operator double()const
{
    return pounds;
}
```

Ve výpisu 10.19 probíhá testování nových konverzních funkcí. Přiřazení využívá implicitní konverze, zatímco poslední příkaz cout využívá explicitního přetypování. Nezapomeňte zkompilovat výpis 10.19 společně s výpisem 10.18.

Výpis 10.19. stone1.cpp

```
// stone1.cpp -- uživatelem definované konverzní funkce
// potřeba zkompilovat společně s  stonewt1.cpp
#include <iostream>
using namespace std;
#include "stonewt1.h"
int main()
{
    Stonewt poppins(9, 2.8);     // 9 kamenů, 2.8 liber
    double p_wt = poppins;       // implicitní konverze
        cout << "Konverze na typ double => ";
    cout << "Poppins: " << p_wt << " liber.\n";
    cout << "Konverze na typ int => ";
        cout << "Poppins: " << int (poppins) << " liber.\n";
  return 0;
}
```

Zde je výstup tohoto programu, který zobrazuje výsledek převodu objektu třídy Stonewt na typy double a int:

```
Konverze na typ double => Poppins: 128.8 liber.
Konverze na typ int => Poppins: 129 liber.
```

Použití automatické typové konverze

Poslední příklad použil výraz `int(poppins)` společně s objektem `cout`. Předpokládejme, že bychom explicitní přetypování vynechali:

```
cout << "Poppins: " << poppins << " liber.\n";
```

Použije program implicitní konverzi, jako v následujícím příkazu?

```
double p_wt = poppins;
```

Odpověď je ne. V případě proměnné `p_wt` kontext naznačuje, že by se proměnná `poppins` měla převést na typ `double`. Ale v příkladě s objektem cout nic nenaznačuje, zda by měla proběhnout konverze na typ `int` nebo typ `double`. Vzhledem k nedostatku informací by kompilátor namítal, že jste použili nejednoznačnou konverzi. Nic v příkazu nenaznačuje, jaký typ se má použít.

Je zajímavé, že pokud by třída měla definovanou pouze konverzní funkci pro převod na typ `double`, kompilátor by tento příkaz přijal. Pokud je možná pouze jediná konverze, nejednoznačnost odpadá.

Podobná situace může nastat u přiřazení. Při současných deklaracích třídy kompilátor odmítne následující příkaz jako nejednoznačný:

```
long gone = poppins;  // nejednoznačné
```

V C++ je možné přiřadit proměnné typu `long` jak hodnoty typu `int`, tak i hodnoty typu `double`. Kompilátor tedy může legitimně použít obě konverzní funkce, ale nechce mít zodpovědnost za výběr. Jestliže však jednu z těchto dvou konverzních funkcí vyloučíte, příkaz přijme. Předpokládejme například, že vynecháte definici pro typ `double`. V tom případě kompilátor převede objekt `poppins` na typ `int` pomocí konverze na typ `int`. Potom před přiřazením proměnné `gone` převede hodnotu typu `int` na typ `long`.

Jestliže je ve třídě definováno dvě a více konverzí, můžete ještě pomocí explicitního přetypování určit, která konverzní funkce se má použít. Použít lze oba zápisy přetypování:

```
long gone = (double) poppins; // konverze na double
long gone = int (poppins);    // konverze na int
```

První příkaz převede váhu `poppins` na hodnotu typu `double` a potom přiřazení převede hodnotu typu `double` na typ `long`. Podobně druhý příkaz převede poppins nejdříve na typ `int` a potom na typ `long`.

Stejně jako konverzní konstruktory i konverzní funkce mohou být pochybným požehnáním. Problém u funkcí provádějících automatické, implicitní konverze spočívá v tom, že mohou provést konverze, když je neočekáváte. Předpokládejme například, že jste napsali následující nedbalý kód:

```
int ar[20];
...
Stonewt temp(14, 4);
...
int Temp;
...
cout << ar[temp] <<"!\n"; // použito temp namísto Temp
```

Normálně byste očekávali, že kompilátor odchytí takovou chybu, jako je použití objektu místo celého čísla jako indexu pole. Ale třída `Stonewt` definuje funkci `operator int()`, takže objekt `temp` bude zkonvertován na hodnotu 200 typu `int` a ta bude použita jako index pole. Mravním ponaučením je, že často je lepší použít explicitní konverze a vyloučit možnost konverzí implicitních. Klíčové slovo `explicit` u konverzních funkcí nefunguje. Stačí však nahradit konverzní funkci funkcí nekonverzní, která provede stejný úkol. Musí být ovšem volána explicitně. Můžete tedy nahradit kód

```
Stonewt::operator int() { return int (pounds + 0.5); }
```

kódem

```
int Stonewt::Stone_to_Int() { return int (pounds + 0.5); }
```

a znemožnit příkaz

```
int plb = poppins;
```

pokud ale opravdu potřebujete konverzi, použijte následující příkaz:

```
int plb = poppins.Stone_to_Int();
```

Upozornění

Používejte implicitní konverzní funkce opatrně. Často je lepším řešením funkce, kterou lze vyvolat pouze explicitně.

V jazyce C++ tedy existují pro třídy následující konverze typů:

- Konstruktor třídy s jediným parametrem slouží jako instrukce pro převod hodnoty z typu parametru na typ třídy. Například konstruktor třídy `Stonewt` s parametrem typu `int` je automaticky vyvolán při přiřazení hodnoty typu `int` objektu třídy `Stonewt`. Použijete-li však v deklaraci konstruktoru klíčové slovo `explicit`, vyloučíte implicitní konverze a povoleny budou pouze konverze explicitní.
- Speciální členská funkce třídy operátoru zvaná konverzní funkce slouží jako instrukce pro konverzi objektu třídy na nějaký jiný typ. Konverzní funkce je členskou funkcí, nemá deklarován návratový typ, nemá parametry a má název `operator` *názevTypu()*, kde *názevTypu* je typ, na který bude objekt převeden. Tato konverzní funkce je vyvolána automaticky, když přiřadíte objekt třídy proměnné daného typu nebo na tento typ použijete operátor přetypování.

Konverze a přátelé

Přidejme třídě `Stonewt` sčítání. Jak jsme se zmínili u třídy `Time`, můžete sčítání přetížit buď pomocí členské funkce nebo pomocí spřátelené funkce. (Pro jednoduchost předpokládejme, že nejsou definovány žádné konverzní funkce tvaru `operator double()`.) Sčítání můžete implementovat pomocí následující členské funkce:

```
Stonewt Stonewt::operator+(const Stonewt & st) const
{
    double pds = pounds + st.pounds;
    Stonewt sum(pds);
    return sum;
}
```

Můžete také použít spřátelenou funkci:

```
Stonewt operator+(const Stonewt & st1, const Stonewt & st2)
{
    double pds = st1.pounds + st2.pounds;
    Stonewt sum(pds);
    return sum;
}
```

Obě formy vám dovolí následující příkazy:

```
Stonewt jennySt(9, 12);
Stonewt bennySt(12, 8);
```

```
Stonewt total;
total = jennySt + bennySt;
```

Navíc díky konstruktoru Stonewt(double) jsou možné následující příkazy:

```
Stonewt jennySt(9, 12);
double kennyD = 176.0;
Stonewt total;
total = jennySt + kennyD;
```

Ale pouze spřátelené funkce umožní toto:

```
Stonewt jennySt(9, 12);
double pennyD = 146.0;
Stonewt total;
total = pennyD + jennySt;
```

Abychom viděli proč, převedeme každé sčítání do odpovídajícího volání funkce. Nejdříve se z příkazu

```
total = jennySt + bennySt;
```

stane

```
total = jennySt.operator+(bennySt);  // členská funkce
```

nebo

```
total = operator+(jennySt, bennySt); // spřátelená funkce
```

V obou případech skutečné parametry typově odpovídají formálním parametrům. Také členská funkce je vyvolána pomocí objektu Stonewt, jak je požadováno.

Dále se z příkazu

```
total = jennySt + kennyD;
```

stane

```
total = jennySt.operator+(kennyD);  // členská funkce
```

nebo

```
total = operator+(jennySt, kennyD); // spřátelená funkce
```

Opět je členská funkce vyvolána pomocí objektu `Stonewt`, jak je požadováno. Tentokrát je v obou příkladech jeden parametr typu `double`, což při převodu parametru na objekt Stonewt vyvolá konstruktor `Stonewt(double)`.

Kdybychom měli nadefinovánu členskou funkci `operator double()`, došlo by zde k chybě, protože byla možná i jiná interpretace. Namísto konverze proměnné `kennyD` na typ `double` a sčítání s objektem třídy `Stonewt` by kompilátor mohl převést proměnnou jennySt na typ `double` a provést sčítání typů `double`. Příliš mnoho konverzních funkcí vytváří nejednoznačnosti.

Nakonec se z příkazu

```
total = pennyD + jennySt;
```

stane

```
total = operator+(pennyD, jennySt); // spřátelená funkce
```

Zde jsou oba parametry typu `double`, což při jejich převodu na objekty `Stonewt` vyvolá konstruktor `Stonewt(double)`. Členskou funkci však vyvolat nelze.

```
total = pennyD.operator+(jennySt);   // nemá smysl
```

Důvodem je, že členskou funkci může vyvolat pouze objekt. C++ se nepokusí převést proměnnou `pennyD` na objekt třídy `Stonewt`. Konverze se provádí u parametrů členských funkcí, nikoli u objektů, které členské funkce vyvolávají.

Ponaučením je, že definujete-li sčítání jako spřátelenou funkci, program se snadněji vypořádá s automatickými typovými konverzemi. Důvodem je, že z obou operandů se stanou parametry funkce a do hry vstoupí prototypy funkcí obou operandů.

Možnost výběru

Za předpokladu, že chcete sčítat hodnoty typu `double` s objekty třídy `Stonewt`, máte několik možností. První, kterou jsme právě načrtli, je definovat funkci `operator+(const Stonewt &, const Stonewt &)` jako funkci spřátelenou a použít pro převod parametrů typu `double` na typ `Stonewt` konstruktor `Stonewt(double)`.

Druhou možností je další přetížení operátoru sčítání pomocí funkcí, které explicitně používají jeden parametr typu `double`:

```
Stonewt operator+(double x); // členská funkce
friend Stonewt operator+(double x, Stonewt & s);
```

Takto příkaz

```
total = jennySt + kennyD; // Stonewt + double
```

přesně odpovídá členské funkci `operator+(double x)` a příkaz

```
total = pennyD + jennySt; // double + Stonewt
```

přesně odpovídá spřátelené funkci `operator+(double x, Stonewt &s)`. Dříve jsme něco podobného udělali s násobením u třídy `Vector`.

Každá volba má své výhody. První vede ke kratšímu programu, protože definujete méně funkcí. To také znamená méně práce pro vás a méně možností chyb. Nevýhodou je potřeba větší časové a paměťové režie při vyvolání konverzního konstruktoru při každé konverzi. Druhá možnost je přesným opakem té první. Vyžaduje delší program a více práce na vaší straně, ale program běží rychleji.

Jestliže váš program ve velké míře používá sčítání hodnot typu `double` s objekty třídy `Stonewt`, může se vyplatit přetížit sčítání přímo pro takové případy. Pokud ale program tyto operace používá zřídka, je jednodušší spoléhat se na automatické konverze, nebo, pokud chcete být opatrnější, na konverze explicitní.

Shrnutí

Tato kapitola pokrývá mnoho důležitých aspektů definování a používání tříd. Některá látka vám možná bude připadat nejasná až do doby, než ji pochopíte díky vlastním zkušenostem. Zatím si obsah kapitoly zrekapitulujeme.

Jediný možný přístup k soukromým položkám třídy bývá běžně pomocí metod třídy. Jazyk C++ toto omezení díky spřátelené funkci zmírňuje. Chcete-li z funkce učinit funkci spřátelenou, deklarujte ji v deklaraci třídy a deklaraci uveďte klíčovým slovem `friend`.

C++ rozšiřuje přetěžování operátorů, neboť umožňuje definovat funkce operátorů popisující vztah jednotlivých operátorů ke konkrétní třídě. Funkce operátoru může být členskou funkcí nebo spřátelenou funkcí. (Členskou funkcí třídy může být jen několik málo operátorů.) C++ umožňuje vyvolat funkci operátoru buď vyvoláním samotné funkce operátoru nebo pomocí přetíženého operátoru obvyklou syntaxí. Funkce operátoru pro operátor *op* má tuto formu:

```
operatorop(seznam parametrů)
```

Seznam parametrů reprezentuje operandy daného operátoru. Jestliže je funkce operátoru členskou funkcí, je prvním operandem volající objekt a není součástí seznamu parametrů. Přetížili jsme například sčítání definicí členské funkce `operator+()` pro třídu `Vector`. Jestliže jsou `up`, `right` a `result` tři vektory, můžete vyvolat sčítání vektorů pomocí kteréhokoli z následujících příkazů:

```
result = up.operator+(right);
result = up + right;
```

Pokud jde o druhou verzi, skutečnost že operandy up a right jsou typem `Vector` říká C++, aby použil definici sčítání z této třídy.

Jestliže je funkce operátoru členskou funkcí, je prvním operandem objekt vyvolávající funkci. V předchozích příkazech je například volajícím objektem objekt up. Pokud chcete funkci operátoru definovat tak, aby prvním operandem nebyl objekt třídy, musíte použít spřátelenou funkci. Potom můžete uvést operandy v definici funkce v libovolném pořadí.

Jedním z nejběžnějších úkolů u přetěžování operátorů je definice operátoru << tak, aby mohl být použit společně s objektem `cout` ke zobrazení obsahu objektu. Má-li být objekt třídy `ostream` prvním operandem, definujte funkci operátoru jako spřátelenou. Aby bylo možné zřetězit tento předefinovaný operátor se sebou samým, musí být návratovým typem reference na `ostream &`. Zde je obecný tvar vyhovující těmto požadavkům:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
        os << … ; // zobrazí obsah objektu
        return os;
}
```

Pokud však má třída metody, vracející hodnoty datových položek, které chcete zobrazit, můžete místo přímého přístupu pomocí funkce `operator<<()` použít tyto metody. V takovém případě funkce nemusí (a neměla by) být funkcí spřátelenou.

Jazyk C++ umožňuje vytvořit konverze mezi třídami a ostatními typy. Za prvé, každý konstruktor vlastnící pouze jeden parametr se chová jako konverzní funkce a převádí hodnoty z typu parametru na typ třídy. Tento konstruktor vyvolá C++ automaticky, přiřadíte-li objektu hodnotu typu parametru. Předpokládejme například, že máte třídu `String` s konstruktorem s jediným parametrem typu `char *`. Jestliže je v tom případě `bean` objektem třídy `String`, můžete použít následující příkaz:

```
bean = "pinto";  // převede typ char * na typ String
```

Jestliže však před deklarací konstruktoru uvedete klíčové slovo `explicit`, potom lze konstruktor používat pouze pro explicitní konverze:

```
bean = String("pinto");  // převede typ char * na typ String explicitně
```

Chcete-li konverzi z třídy na jiný typ, musíte nadefinovat konverzní funkci obsahující instrukce o způsobu konverze. Konverzní funkce musí být členskou funkcí. Pokud je cílem konverze na typ *názevTypu*, měla by mít tento prototyp:

```
operator názevTypu();
```

Všimněte si, že nesmí mít deklarován žádný návratový typ, nesmí mít žádné parametry a musí vracet výslednou hodnotu konverze (i když nemá deklarován návratový typ). Například funkce pro konverzi objektu třídy `Vector` na číslo typu `double` by měla tento tvar:

```
Vector::operator double()
{
        ...
        return double_hodnota;
}
```

Zkušenosti ukazují, že často je lepší se na takové implicitní konverzní funkce nespoléhat.

Jak jste si asi všimli, třídy vyžadují do detailů mnohem více péče a pozornosti než obyčejné struktury ve stylu jazyka C. Na oplátku toho pro vás mnohem více udělají.

Opakovací otázky

1. Použijte členskou funkci pro přetížení operátoru násobení třídy `Stonewt`. Operátor umožní násobit členská data číslem typu `double`. Nezapomeňte, že bude třeba převod pro reprezentaci v kamenech a librách. Dvojnásobek 10 kamenů a 8 liber je 21 kamenů a 2 libry.
2. Jaký je rozdíl mezi spřátelenou funkcí a členskou funkcí?
3. Musí být nečlenská funkce spřátelenou funkcí, aby mohla přistupovat k položkám třídy?
4. Použijte spřátelenou funkci pro přetížení operátoru násobení třídy `Stonewt`. Operátor umožní násobit hodnotu typu `double` hodnotou typu `Stonewt`.
5. Které operátory nelze přetížit?
6. Jaké omezení platí pro přetěžování následujících operátorů? = () [] ->
7. Definujte konverzní funkci pro třídu `Vector`, která převede `Vektor` na hodnotu typu `double`, reprezentující jeho velikost.

Programovací cvičení

1. Upravte program z výpisu 10.13 tak, aby nevypisoval výsledky jednoho pokusu s určitou kombinací cíle a délky kroku, ale aby vypisoval největší, nejmenší a průměrný počet kroků pro *N* pokusů, kde *N* je celé číslo zadané uživatelem.
2. Přepište třídu `Stonewt` tak, aby obsahovala stavovou položku určující, zda bude objekt reprezentován v kamenech, v librách celým číslem nebo v librách číslem s plovoucí řádovou čárkou. Definujte přetížený operátor <<, který nahradí metody `show_stn()` a `show_lbs()`. Definujte přetížené operátory sčítání, odčítání a násobení tak, aby uměli sčítat, odečítat a násobit hodnoty typu `Stonewt`. Vyzkoušejte třídu pomocí krátkého programu.
3. Přepište třídu `Stonewt` tak, aby přetěžovala relační operátory. Napište program, který bude deklarovat pole šesti objektů `Stonewt` a inicializuje první tři objekty v deklaraci pole. Potom by měl pomocí smyčky načíst hodnoty pro nastavení zbývajících tří prvků pole. Dále by měl vypsat nejmenší a největší prvek, a počet prvků, jejichž hodnota je větší nebo rovna 11 kamenům.
4. Komplexní číslo má dvě části: reálnou a imaginární. Jedním způsobem, jak napsat imaginární číslo, je (3.0, 4.0i). Zde je 3.0 reálnou částí a 4.0i částí imaginární. Předpokládejme, že platí `a = (A,Bi)` a `c = (C,Di)`. Zde jsou některé operace s komplexními čísly:

   - Sčítání: a + c = (A + C, (B + D)i)
   - Odčítání: a – c = (A – C, (B – D)i)
   - Násobení: a * c = (A * C – B*D, (A*D + B*C)i)
   - Násobení: (x je reálné číslo): x * c = (x*C,x*Di)
   - Konjugace: ~a = (A, – Bi)

   Definujte komplexní třídu tak, aby následující program dával správné výsledky. Nezapomeňte, že musíte přetížit operátory << a >>. Mnoho systémů obsahuje podporu komplexních čísel v hlavičkovém souboru `complex.h`, proto použijte název `complex0.h`, aby nedošlo ke kolizi. Použijte modifikátor `const` kdekoli je to možné.

```
#include <iostream>
using namespace std;
#include "complex0.h"      // zamezí konfliktu s complex.h
int main()
  {
      complex a(3.0, 4.0);  // inicializace na hodnotu (3,4i)
      complex c;
      cout << "Zadejte komplexni cislo (k pro ukonceni):\n";
      while (cin >> c)
        {
          cout << "c is " << c << '\n';
          cout << "komplexni sdruzene cislo je " << ~c << '\n';
          cout << "a + c je " << a + c << '\n';
          cout << "a - c je" << a - c << '\n';
          cout << "a * c je" << a * c << '\n';
```

```
        cout << "2 * c je" << 2 * c << '\n';
        cout << " Zadejte komplexni cislo (k pro ukonceni):\n";
    }
    cout << "Hotovo!\n";
    return 0;
}
```

Zde je ukázka běhu programu. Všimněte si, že díky přetížení výraz cin >> c nyní požádá o reálnou a imaginární část:

```
Zadejte komplexni cislo (k pro ukonceni):
realne: 10
imaginarni: 12
c je (10,12i)
komplexni sdruzene cislo (10,-12i)
a + c je (13,16i)
a - c je (13,16i)
a * c je (-18,76i)
2 * c je (20,24i)
 Zadejte komplexni cislo (k pro ukonceni):
realne: k
Hotovo!
```

Třídy a dynamické přidělování paměti

KAPITOLA 11

Témata kapitoly:

- Dynamické přidělování paměti položkám třídy
- Implicitní a explicitní kopírovací konstruktory
- Implicitní a explicitní přetížené operátory přiřazení
- Jak postupovat při použití operátoru new v konstruktoru
- Používání `statických` položek třídy – Používání ukazatelů na objekty
- Implementace fronty ADT

V této kapitole se podíváme na způsoby používání operátorů `new` a `delete` ve třídách a na způsoby řešení některých záludných problémů, které mohou vzniknout při dynamickém přidělování paměti. Může to vypadat jako stručný seznam témat, tato témata však ovlivňují návrh konstruktorů a destruktorů a přetěžování operátorů.

Podívejme se na konkrétní příklad, jak může C++ zvýšit zatížení paměti. Předpokládejme, že chcete vytvořit třídu s položkou reprezentující něčí příjmení. Nejjednodušší je použít pro příjmení pole znaků. Ale toto řešení má jisté nevýhody. Použijete třeba pole o 14 znacích, ale pak narazíte na jméno Bartholomew Smeadsbury-Craftho-vingham. Nebo pro jistotu použijete pole o 40 znacích. Ale jestliže potom vytvoříte pole s 2 000 takovými objekty, budete plýtvat pamětí, protože některá pole budou zaplněna pouze částečně. (V takové chvíli zatěžujeme paměť počítače.) Ale existuje jiné řešení.

Často je mnohem lepší rozhodovat o věcech, jako je množství použité paměti, až při běhu programu a ne při kompilaci. V jazyce C++ se při ukládání jména obvykle použije v konstruktoru třídy operátor `new` a správné množství paměti se přidělí za běhu programu. Použití operátoru `new` v konstruktoru však přinese několik nových problémů, pokud zapomenete provést několik dalších kroků, jako jsou rozšíření destruktoru třídy, sladění všech konstruktorů s novým destruktorem a napsání dalších metod třídy, které zabezpečí správnou inicializaci a přiřazení. (V této kapitole budou všechny tyto kroky samozřejmě vysvětleny.) Pokud se C++ právě učíte, uděláte asi lépe, když ze začát-

ku zůstanete sice u horšího, ale jednoduchého řešení s polem znaků. Teprve až bude navržená třída správně fungovat, můžete se vrátit na pole OOP a rozšířit deklaraci třídy o operátor `new`. Stručně řečeno, dostávejte se do C++ postupně.

Dynamická paměť a třídy

Co byste chtěli k snídani, obědu a večeři na příští měsíc? Kolik litrů mléka k večeři třetí den? Kolik rozinek v müsli k snídani patnáctý den? Pokud jste jako většina lidí, asi některá tato rozhodnutí odložíte až na skutečnou dobu jídla. Částí strategie C++ je podobný přístup k přidělování paměti – programu je ponechána možnost rozhodovat o paměti až za běhu a ne při kompilaci. Při tomto způsobu může spotřeba paměti záviset na potřebách programu a ne na sadě pravidel pro třídu pevně stanovených. Jak si vzpomínáte, používá C++ pro správu dynamické paměti operátory `new` a `delete`. Naneštěstí použití těchto operátorů ve třídách může v programování způsobit některé nové problémy. Jak uvidíte, destruktory mohou být nutností, ne jen jakousi ozdobou. A někdy budete muset zajistit správný chod programů přetížením operátoru přiřazení. Na tyto věci se nyní podíváme.

Příklad na zopakování a statické položky tříd

Nějakou chvíli jsme už operátory `new` a `delete` nepoužívali, zopakujme si je tedy pomocí krátkého programu. Přitom zavedeme novou paměťovou třídu: statickou položku třídy. K tomu nám poslouží třída `String`. (Jazyk C++ nyní nabízí knihovnu tříd pro práci s řetězci podporovanou hlavičkovým souborem `string` a zabývat se jí bude kapitola 15. Mezitím nás skromná třída `String` v této kapitole nechá proniknout do podstaty takové třídy.) Objekt třídy `String` bude obsahovat ukazatel na řetězec a hodnotu představující délku tohoto řetězce. Třídu `String` použijeme především proto, abychom objasnili fungování operátorů `new` a `delete` a `statických` položek třídy. Z tohoto důvodu budou konstruktory i destruktory při zavolání zobrazovat zprávy, takže budete moci sledovat jejich činnost. Abychom rozhraní třídy zjednodušili, vypustíme některé užitečné členské a spřátelené funkce jako jsou přetížené operátory + a >> a konverzní funkce. (Ale netruchlete! Opakovací otázky z této kapitoly vám dají příležitost tyto užitečné obslužné funkce přidat.) Ve výpisu 11.1 je deklarace třídy. Soubor jsme pojmenovali `string1.h`, abychom se vyhnuli konfliktu se souborem `string.h` ze standardní knihovny. (Poslední implementace jazyka C++ obsahují hlavičkový soubor `string.h`, který podporuje funkce jazyka C jako `strcpy()` pro práci s řetězci ve stylu jazyka C, hlavičkový soubor `cstring` založený na hlavičkovém souboru `string.h` a hlavičkový soubor `string` podporující třídy `string` jazyka C++.)

Výpis 11.1. strng1.h

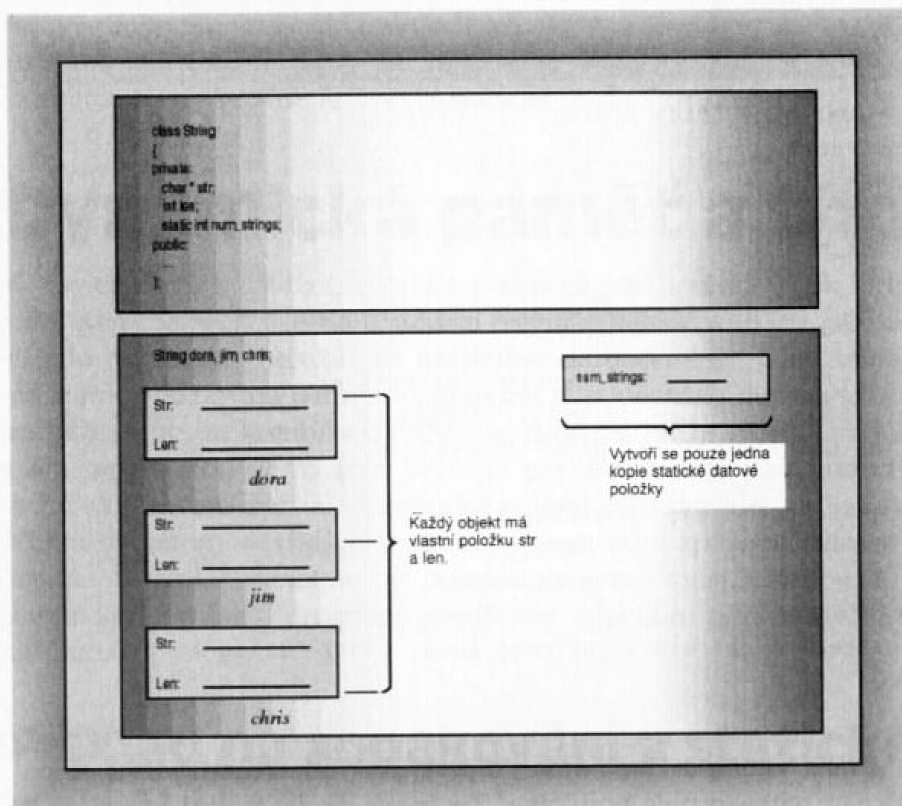
```
// strng1.h - definice třídy string
#include <iostream>
using namespace std;
#ifndef _STRNG1_H_
#define _STRNG1_H_
class String
{
private:
    char * str;                 // ukazatel na řetězec
    int len;                    // délka řetězce
    static int num_strings;  // počet objektů
public:
    String(const char * s);  // konstruktor
    String();                   // implicitní konstruktor
    ~String();                  // destruktor
// spřátelená funkce
    friend ostream & operator<<(ostream & os, const String & st);
};
#endif
```

V této deklaraci byste si měli všimnout dvou míst. Za prvé, pro reprezentaci jména se používá ukazatel na typ `char` namísto pole typu `char`. To znamená, že v deklaraci třídy se místo pro uložení samotného řetězce nepřiděluje. Místo toho se řetězci místo přidělí v konstruktorech pomocí operátoru `new`. Díky tomuto uspořádání nebude deklarace třídy omezena předdefinovanou velikostí řetězce.

Za druhé, v definici je deklarována položka `num_strings` jako `statická` paměťová třída. Statická položka třídy má speciální vlastnost: program vytvoří pouze jednu kopii této proměnné bez ohledu na počet vytvořených objektů. To znamená, že statickou položku sdílí všechny objekty dané třídy, podobně jako telefonní číslo mohou sdílet všichni členové rodiny. Jestliže vytvoříte například deset objektů třídy `String`, pak bude existovat 10 položek `str` a 10 položek `len`, ale pouze jedna sdílená položka `num_strings` (viz obrázek 11.1). Toto se hodí pro data, která by měla být ve třídě soukromá, ale měla by mít stejnou hodnotu pro všechny objekty třídy. Položka `num_strings` má například zaznamenávat počet vytvořených objektů.

Statickou položku `num_strings` jsme použili jako vhodný prostředek pro ilustraci statických položek jako nástroje pro upozornění na potenciální programovací problémy. Třída `String` obecně takovou položku nepotřebuje. Pokud byste takový prostředek potřebovali, bylo by lepší definovat obecnější třídu `String` a potom tuto vlastnost přidat odvozené třídě pomocí dědičnosti (kapitola 12). (Dědičnost umožňuje vytvořit novou třídu rozšiřující třídu již existující.)

Podívejme se na implementaci metod třídy ve výpisu 11.2. Zde uvidíte způsob použití těchto dvou prostředků (ukazatele a statické položky).



Obrázek 11.1. Statická datová položka

Výpis 11.2. strngl.cpp

```
// strngl.cpp - metody třídy String
#include <iostream>
#include <cstring>                          // někdy string.h
#include "strngl.h"
using namespace std;
// inicializace statické položky třídy
int String::num_strings = 0;
// metody třídy
String::String(const char * s)      // vytvoří třídu String z řetězce ja-
zyka C
{
    len = strlen(s);                    // nastaví velikost
    str = new char[len + 1];           // přidělí paměť
    strcpy(str, s);                     // inicializuje ukazatel
    num_strings++;                      // nastaví čítač objektů
    cout << num_strings << ": \"" << str
         << "\" objekt vytvoren\n"; // pro vaši informaci (PVI)
}
String::String()                        // implicitní konstruktor
{
    len = 4;
    str = new char[4];
```

```
        strcpy(str, "C++");                    // implicitni řetězec
    num_strings++;
        cout << num_strings << ": \"" << str
             << "\" implicitni objekt vytvoren\n";   // PVI
    }
    String::~String()                          // nutný destruktor
    {
        cout << "\"" << str << "\" objekt zrusen, ";   // PVI
        --num_strings;                         // nutné
        cout << num_strings << " zbyvaji\n";   // PVI
        delete [] str;                         // nutné
    }
    ostream & operator<<(ostream & os, const String & st)
    {
        os << st.str;
        return os;
    }
```

Nejdříve si ve výpisu 11.2 všimněte následujícího příkazu:

```
int String::num_strings = 0;
```

Tento příkaz nastaví počáteční hodnotu statické položky na hodnotu 0. Všimněte si, že statickou členskou proměnnou nemůžete inicializovat v deklaraci třídy. Deklarace totiž způsob přidělení paměti pouze popisuje, ale přidělení neprovádí. Paměť tímto způsobem přidělíte a inicializujete při vytvoření objektu. V případě statické položky třídy inicializujete statickou položku odděleně mimo deklaraci třídy. Je totiž uložena odděleně a netvoří součást objektu. Všimněte si, že inicializační příkaz obsahuje název typu a operátor rozlišení.

```
int String::num_strings = 0;
```

Tato inicializace je uložena v souboru metod, nikoli v deklaračním souboru třídy. Deklarace třídy se totiž nachází v hlavičkovém souboru a program může obsahovat hlavičkový soubor v několika dalších souborech. To by vedlo k vytvoření více kopií inicializačního příkazu, což je chyba.

Výjimku (viz kapitola 9) ze zákazu inicializace statické položky uvnitř deklarace třídy představuje `konstantni` statická položka celočíselného nebo výčtového typu.

Pamatujte

Statická datová položka je deklarována v deklaraci třídy a inicializována v souboru s metodami třídy. Operátor rozlišení slouží k určení třídy, do které statická položka patří. Jestliže však je statická položka konstantou celočíselného nebo výčtového typu, lze ji inicializovat v samotné deklaraci třídy.

Dále si všimněte, že každý konstruktor obsahuje výraz `num_strings++`. Díky tomu je zajištěno, že se sdílená proměnná `num_strings` zvětší o jedničku při každém vytvoření nového objektu a eviduje tak celkový počet objektů třídy `String`. Také si všimněte, že destruktor obsahuje výraz `--num_strings`. Třída `String` eviduje také zrušené objekty a položka `num_strings` tedy obsahuje aktuální hodnotu počtu objektů.

Statické členské funkce třídy

Členskou funkci je také možné deklarovat jako statickou. (Jestliže je funkce definována odděleně, klíčové slovo `static` bude pouze v deklaraci, v definici ne.) Toto má dva důležité důsledky. Za prvé, statická funkce nemusí být vyvolána pomocí objektu; vlastně ani nemá ukazatel `this`, se kterým by spolupracovala. Jestliže je statická členská funkce deklarována ve veřejné části, může být vyvolána pomocí názvu třídy a operátoru rozlišení. Předpokládejme například, že třída `Person` má statickou členskou funkci `HowMany()` s následujícím prototypem a definicí v deklaraci třídy:

```
static int HowMany() { return num_strings; }
```

Potom by mohla být vyvolána následujícím způsobem:

```
int count = String::HowMany();    // vyvolání statické členské funkce
```

Za druhé, protože statická členská funkce není spojena s žádným konkrétním objektem, může používat pouze statické datové položky. Statická metoda `HowMany()` by například mohla přistupovat ke statické položce `num_strings`, ale ne k položkám `str` nebo `len`. Podobně lze pomocí statické členské funkce nastavit příznak třídy, který bude řídit způsob zobrazení dat uložených v objektech.

Nyní se podívejme na první konstruktor, který inicializuje objekt třídy `String` pomocí běžného řetězce jazyka C:

```
String::String(const char * s)          // vytvoří objekt třídy String
                                        // z řetězce jazyka C
   {
       len = strlen(s);                 // nastaví délku
        str = new char[len + 1];        // přidělí paměť
        strcpy(str, s);                 // inicializuje ukazatel
        num_strings++;                  // nastaví čítač objektů
        cout << num_strings << ": \"" << str
             << "\" objekt vytvoren\n";  // PVI
   }
```

Vzpomeňte si, že položka `str` je pouze ukazatel, takže konstruktor musí přidělit paměť pro uložení řetězce. Při inicializaci objektu můžete konstruktoru předat ukazatel na řetězec:

```
String boston("Boston");
```

Konstruktor potom musí pro řetězec přidělit dostatek paměti a řetězec na toto místo zkopírovat. Projděme si tento proces krok za krokem.

Nejdříve funkce pomocí funkce `strlen()` vypočte délku řetězce a inicializuje položku `len`. Dále pomocí operátoru `new` přidělí řetězci dostatek místa a ukazateli str přiřadí adresu této nové paměti. (Vzpomeňte si, že funkce `strlen()` vrací délku řetězce bez ukončovacího nulového znaku, konstruktor tedy zvětší hodnotu proměnné len o jedničku, aby se vytvořilo místo i pro nulový znak.)

Dále konstruktor pomocí funkce `strcpy()` zkopíruje předaný řetězec do nově přidělené paměti. Potom aktualizuje čítač objektů. Nakonec, abychom mohli snáze sledovat činnost,

vypíše konstruktor aktuální počet objektů a řetězec uložený do objektu. Tato vlastnost se bude hodit později, až třídě String úmyslně přivodíme problémy.

Abyste tomuto postupu rozuměli, musíte si uvědomit, že řetězec není uložen v objektu. Je uložen odděleně na haldě a objekt pouze ukládá informace o jeho místě.

Všimněte si, že nepoužíváte následující příkaz:

```
str = s; // toto nikam nevede
```

Tímto příkazem pouze uložíte adresu, ale řetězec nezkopírujete. Implicitní konstruktor se chová podobně s tím rozdílem, že obsahuje implicitní řetězec „C++". To nejdůležitější, co jsme pro práci se třídami do příkladu přidali, obsahuje destruktor:

```
String::~String()                         // nutný destruktor
  {
      cout << "\"" << str << "\" objekt zrusen. ";    // PVI
      --num_strings;                      // nutné
      cout << num_strings << " zbyvaji\n"; // PVI
      delete [] str;                      // nutné  }
```

Destruktor nejdříve informuje o svém zavolání. Tato část je informativní, není však podstatná. Ovšem příkaz `delete` je životně důležitý. Vzpomeňte si, že ukazatel `str` ukazuje na paměť přidělenou operátorem `new`. Když platnost objektu třídy `String` skončí, přestane existovat i ukazatel `str`. Ale paměť, na kterou `str` ukazoval, zůstane přidělena, dokud ji neuvolníte příkazem `delete`. Zrušením objektu se uvolní paměť, kterou objekt obsadil, ale paměť, na kterou ukazovaly ukazatele, které byly položkami objektů, se automaticky neuvolní. Z tohoto důvodu musíte použít destruktor. Příkazem `delete` v destruktoru zajistíte, že paměť přidělená v konstruktoru operátorem `new` bude uvolněna, jakmile skončí platnost objektů.

Pamatujte

Kdykoli v konstruktoru přidělíte paměť pomocí operátoru `new`, měli byste v odpovídajícím destruktoru pomocí operátoru `delete` tuto paměť uvolnit. Jestliže použijete příkaz `new []`, musíte také použít příkaz `delete []`.

Výpis 11.3 ilustruje, kdy a jak konstruktory a destruktory třídy `String` fungují. Nezapomeňte zkompilovat výpis 11.3 společně s výpisem 11.2.

Výpis 11.3. vegnews.cpp

```
// vegnews.cpp - použití operátorů new a delete ve třídách
// zkompilovat společně s strngl.cpp
#include <iostream>
using namespace std;
#include "strngl.h"
String sports("Spinach Leaves Bowl for Dollars");
                            // externí objekt sports
void callme1();              // vytvoří lokální objekt
String * callme2();          // vytvoří dynamický objekt
```

```
int main()
{
    cout << "Zacatek main()\n";
    String headlines[2] =   // lokální pole objektů
    {
        String("Celery Stalks at Midnight"),
        String("Lettuce Prey")
    };
    cout << headlines[0] << "\n";
    cout << headlines[1] << "\n";
    callme1();
    cout << "Telo main()\n";
    String *pr = callme2(); // nastaví ukazatel na objekt
    cout << sports << "\n";
    cout << *pr << "\n";     // vyvolá metodu třídy
    delete pr;               // zruší objekt
    cout << "Konec main()\n";
    return 0;
}
void callme1()
{
    cout << "Zacatek callme1()\n";
    String grub;               // lokální objekt
    cout << grub << "\n";
    cout << "Konec callme1()\n";
}
String * callme2()
{
    cout << "Zacatek callme2()\n";
    String *pveg = new String("Cabbage Heads Home");
            // použití konstruktoru u dynamického objektu
    cout << *pveg << "\n";
    cout << "Konec callme2()\n";
    return pveg;               // platnost proměnné pveg končí, objekt
přetrvává
}
```

Kompatibilita:

V tomto prvním konceptu návrhu třídy `String` je několik úmyslných závad, které se v této fázi u mnoha překladačů neprojeví. Některé překladače však tyto závady ovlivní, takže následující výstup nedostanete. Při opravě třídy `String` v další části budou problémy tříd probrány a napraveny. Některé implementace také mohou závěrečný destruktor volat až po ukončení výstupu, takže zpráva nemusí být vidět.

Zde je výstup programu:

```
1: "Spinach Leaves Bowl for Dollars" objekt vytvoren
Zacatek main()
2: "Celery Stalks at Midnight" objekt vytvoren
3: "Lettuce Prey" objekt vytvoren
```

```
Celery Stalks at Midnight
Lettuce Prey
Zacatek callme1()
4: "C++" implicitni objekt vytvoren
C++
Konec callme1()
"C++"objekt zrusen, 3 zbyvaji
Telo main()
Zacatek callme2()
4: "Cabbage Heads Home" objekt vytvoren
Cabbage Heads Home
Konec callme2()
Spinach Leaves Bowl for Dollars
Cabbage Heads Home
"Cabbage Heads Home"objekt zrusen, 3 zbyvaji
Konec main()
"Lettuce Prey"objekt zrusen, 2 zbyvaji
"Celery Stalks at Midnight"objekt zrusen, 1 zbyvaji
"Spinach Leaves Bowl for Dollars"objekt zrusen, 0 zbyvaji
```

Poznámky k programu

Ujistěte se, že sledu událostí v tomto vzorovém programu rozumíte. Nyní si je projdeme. Objekt `sports` je externí proměnná, je tedy vytvořen předtím, než se spustí funkce `main()`. Dalšími vytvořenými objekty jsou dva prvky pole `headlines`. Program zavolá konstruktor dvakrát, při každém volání inicializuje jeden prvek pole. Každý prvek je objekt třídy, takže volání

```
cout << headlines[0] << "\n";
cout << headlines[1] << "\n";
```

vyvolá spřátelenou metodu `operator<<()` pro oba objekty `headlines[0]` a `headlines[1]`.

Dále program zavolá funkci `callme1()`. Tato funkce vytvoří pomocí implicitního konstruktoru lokální objekt `grub`. Implicitní konstruktor inicializuje položku `str` na hodnotu „C++". Platnost objektu končí s ukončením funkce `callme1()`, jak je vidět z těchto řádků výstupu:

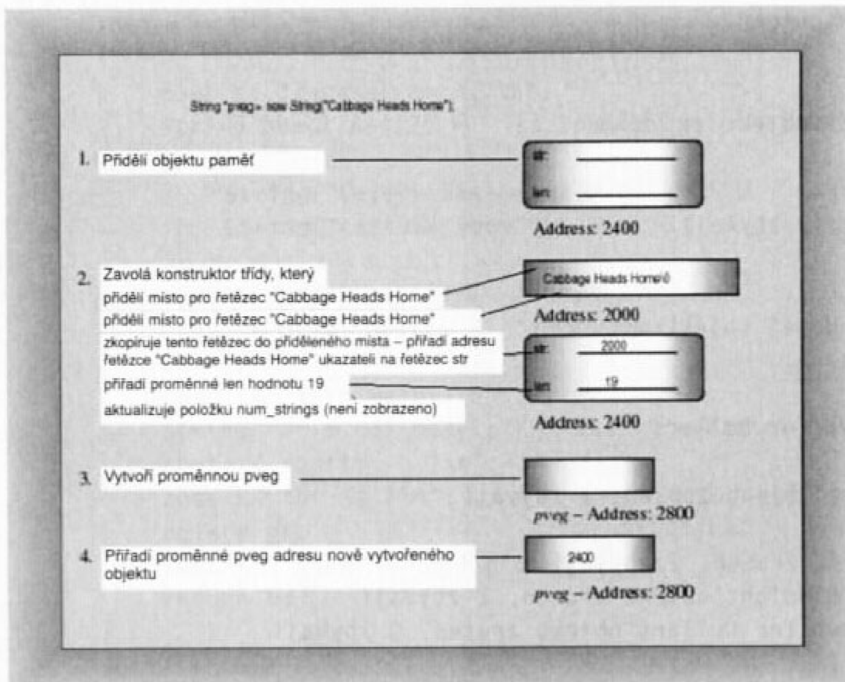
```
Zacatek callme1()
4: "C++" implicitni objekt vytvoren
C++
Konec callme1()
"C++"objekt zrusen, 3 zbyvaji
```

Kromě vypsání zprávy na rozloučenou destruktor také uvolní paměť, která obsahovala řetězec „`C++`".

Nyní přicházíme k nejtěžší části příkladu. Program zavolá funkci `callme2()` a ta vytvoří a inicializuje pomocí operátoru `new` objekt třídy `String`:

```
String *pveg = new String("Cabbage Heads Home");
```

Funkce přiřadí ukazateli pveg adresu tohoto nového objektu. Protože předá operátoru new řetězcový parametr, program inicializuje objekt zavoláním odpovídajícího konstruktoru. Na obrázku 11.2 je shrnutí tohoto příkazu:



Obrázek 11.2 Vytvoření objektu pomocí operátoru new

Všimněte si, že protože `pveg` je ukazatel na objekt, výraz `*pveg` představuje objekt. To znamená, že výraz `*pveg` můžete použít stejným způsobem jako deklarovaný objekt:

```
cout << *pveg << "\n";
```

To způsobí, že funkce vypíše řetězec „`Cabbage Heads Home`".

Potom funkce skončí a automaticky uvolní paměť používanou jejími proměnnými. To znamená, že uvolní paměť, která obsahovala ukazatel `pveg`. Ale protože funkce `callme2()` nepoužívá příkaz delete `pveg`, paměť obsahující objekt, na který ukazoval ukazatel `pveg`, je stále přidělena. Všimněte si, že destruktor při ukončení funkce `callme2()` nevypíše žádnou zprávu; to znamená, že objekt stále existuje – řetězec „`Cabbage Heads Home`" žije dál! Protože však platnost proměnné `pveg` skončila, nemůže již program pomocí ní k tomuto objektu přistupovat. Program však vrátí hodnotu proměnné `pveg` volajícímu programu a přiřadí ji ukazateli `pveg`. Stručně řečeno: nejdříve proměnná `pveg` ukazovala na objekt třídy `String`, potom platnost této proměnné skončila, ale mezitím program nastavil ukazatel pr na objekt třídy `String`. Nyní tedy program může přistupovat k dynamickému objektu pomocí ukazatele `pr`. A přesně to dělá po prvním zobrazení obsahu objektu `sports`.

```
Zacatek callme2()
4: "Cabbage Heads Home" objekt vytvoren
Cabbage Heads Home              používá ukazatel pveg ve funkci callme2()
Konec callme2()
Spinach Leaves Bowl for Dollars
Cabbage Heads Home              používá ukazatel pr ve funkci main()
```

Nyní program přejde ke smutným úkolům – k rušení zbývajících objektů. Protože objekt `cabbage` vytvořil pomocí operátoru `new`, může ho odstranit pomocí operátoru `delete`:

```
delete pr;
```

Nezapomeňte, že tento příkaz uvolní paměť, na kterou ukazatel `pr` ukazuje, ale ne samotnou proměnnou `pr`. Zrušení objektu naopak aktivuje destruktor třídy, který potom zruší paměť obsazenou řetězcem „`Cabbage Heads Home`":

```
"Cabbage Heads Home" objekt zrusen, 3 zbyvaji
```

Existence zbylých objektů se řídí pravidly pro platnost proměnných. Dva prvky pole jsou automatické proměnné, takže jejich existence skončí provedením bloku, ve kterém jsou definovány. V tomto případě je tímto blokem tělo funkce `main()`, takže tyto dva objekty jsou uvolněny při ukončení funkce `main()`. A konečně platnost externích objektů skončí při ukončení celého programu:

```
Konec main()
"Celery Stalks at Midnight" object deleted, 2 left
"Lettuce Prey" object deleted, 1 left
"Spinach Leaves Bowl for Dollars" object deleted, 0 left
```

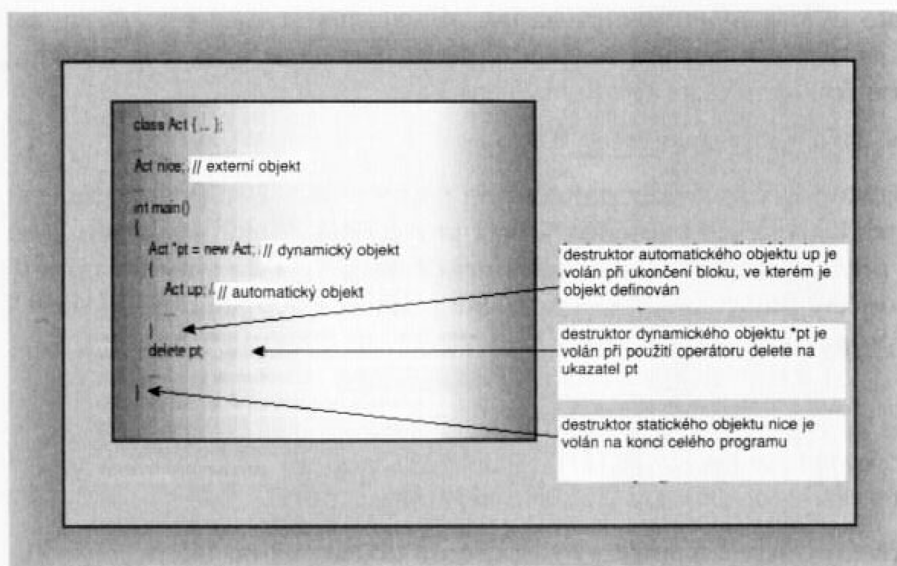
(Jak bylo zmíněno dříve, některé kompilátory ruší externí objekty až po ukončení výstupu, takže by nezobrazily poslední řádek.)

Znovu operátory new a delete

Všimněte si, že program používá operátory `new` a `delete` na dvou úrovních. Za prvé, pomocí operátoru `new` přiděluje místo v paměti pro řetězce každého vytvořeného objektu. Toto se děje v konstruktorech, takže destruktor tuto paměť uvolní pomocí operátoru `delete`. Protože každý řetězec je pole znaků, používá destruktor operátor `delete` s hranatými závorkami. Paměť pro uložení obsahu řetězců je tedy automaticky uvolněna při zrušení objektu. Za druhé, pomocí operátoru `new` program ve funkci `callme2()` přidělí paměť pro celý objekt. Nepřidělí místo pro název řetězce, ale pro objekt, to znamená, pro ukazatel `str`, obsahující adresu řetězce a pro položku `len`. (Nepřidělí místo pro položku `num_strings`, protože se jedná o statickou položku uloženou odděleně od objektů.) Při vytvoření objektu je naopak zavolán konstruktor, který přidělí místo pro uložení řetězce a přiřadí adresu řetězce ukazateli `str`. Jakmile program práci s objektem skončil, zrušil ho pomocí operátoru `delete`. Tento objekt je jediný, proto program používá operátor `delete` bez hranatých závorek. Tímto operátorem opět uvolní pouze místo obsazené proměnnými `str` a `len`. Neuvolní paměť obsazenou řetězcem, na který ukazuje ukazatel `str`, ale o tento poslední úkol se postará destruktor.

Vyzdvihněme opět situace, kdy jsou volány destruktory (viz také obrázek 11.3).

1. Jestliže objekt je automatická proměnná, pak destruktor tohoto objektu bude vyvolán, jakmile program opustí blok, ve kterém je tento objekt definován. Destruktor pro proměnné `headlines[0]` a `headlines[1]` je tedy vyvolán, jakmile program opustí funkci `main()` a destruktor pro objekt `grub` je vyvolán při opuštění funkce `callme1()`.
2. Pokud je objekt statická proměnná (externí, statická, externí statická nebo z nějakého jmenného prostoru), jeho destruktor je vyvolán při ukončení programu. To je případ objektu `sports`.
3. Jestliže je objekt vytvořen pomocí operátoru `new`, jeho destruktor je vyvolán pouze při explicitním použití operátoru `delete` na daný objekt, což je případ objektu vytvořeného ve funkci `callme2()` a zrušeného ve funkci `main()`.



Obrázek 11.3. Kdy se volají destruktory

Způsob, kdy program v jedné funkci objekt vytváří a v jiné ho ruší, je potenciálním zdrojem problémů, protože počítá s tím, že programátor nezapomene objekt zrušit. Uvažujte například následující ošklivou variantu:

```
String * ps;
for (int i = 0; i < 100; i++)
{
    ps = callme2();
    cout << *ps << "\n";
}
delete ps;
```

Tento kód vytvoří sto různých objektů. V každém cyklu smyčky je ukazatel `ps` nastaven na adresu posledního vytvořeného objektu, čímž ztrácí záznam o umístění předchozího objektu. Nakonec kód zruší pouze naposledy vytvořený objekt. Zbývajících 99 bude zabírat paměť, do které program nemá přístup. Takové těžkopádné programování se nazývá *únik paměti*. Správně navržené destruktory zamezí únikům vnitřní paměti v objektu při jeho zrušení, stále ale existují objekty vytvořené explicitně pomocí operátoru `new`, a ty musíte explicitně zrušit. V tomto případě by měl být příkaz `delete` umístěn v cyklu `for`.

Potíže s třídou String

Tato definice třídy `String` není úplná. Z důvodu stručnosti samozřejmě neimplementuje mnoho užitečných metod jako přetížení operátorů <, == a >, které usnadňují porovnávání řetězců. Má však mnohem podstatnější závady. Jako důkaz uvažujte tento jednoduchý program (výpis 11.4), používající aktuální implementaci třídy `String`.

Výpis 11.4. problem1.cpp

```
// problem1.cpp - používá funkci s parametrem typu String
// zkompilovat s strng1.cpp
#include <iostream>
using namespace std;
#include "strng1.h"
void showit(String s, int n);
int main()
{
    String motto("Home Sweet Home");
    showit(motto, 3);
    return 0;
}
void showit(String s, int n)    // zobrazí řetězec s n-krát
{
    for (int i = 0; i < n; i++)
        cout << s << "\n";
}
```

Kompatibilita:

Protože tento a následující dva příklady demonstrují návrh s vadami, bude se výstup lišit v závislosti na kompilátoru. Uvedené příklady vznikly s použitím kompilátoru Borland C++ 3.1.

V tomto příkladu je objekt třídy `String` předán funkci `showit()`, která potom řetězec zobrazí tolikrát, kolikrát udává druhý parametr. Zde je výstup v jednom systému:

```
1: "Home Sweet Home" objekt vytvoren
Home Sweet Home
Home Sweet Home
Home Sweet Home
"Home Sweet Home" objekt zrusen, 0 zbyvaji
"Home Sweet Home" objekt zrusen, -1 zbyvaji
Null pointer assignment
```

Všimněte si několika podivných rysů. Za prvé, podle výpisu programu se vytvoří pouze jeden objekt, ale zrušeny jsou objekty dva a celkový počet objektů v paměti zůstane -1. (Ve vaší implementaci může poslední výstupní příkaz místo řetězce „`Home Sweet Home`" vypsat nějaký blábol.) Potom je zde tajemná zpráva o přiřazení prázdného ukazatele. (To, zda dostanete tuto zprávu nebo jinou a nebo vůbec žádnou, závisí na kompilátoru. Každopádně je zde ale skrytý problém.)

Zde je další jednoduchý program (výpis 11.5), který zadrhává:

Výpis 11.5. problem2.cpp

```
// problem2.cpp — inicializuje jeden řetězec jiným
// zkompilovat s strngl.cpp
#include <iostream>
using namespace std;
#include "strngl.h"
int main()
{
    String motto("Home Sweet Home");
    String ditto(motto);      // inicializuje objekt ditto
                              //objektem motto
    cout << motto << "\n";
    cout << ditto << "\n";
    return 0;
}
```

Program zkouší udělat něco, co zřejmě deklarace třídy neošetřuje: inicializovat objekt třídy `String` pomocí jiného objektu. Jak však ukazuje následující výstup, pokus se vydařil. Program se chová stejně podivně jako v předchozím příkladě:

```
1: "Home Sweet Home" objekt vytvoren
Home Sweet Home
Home Sweet Home
"Home Sweet Home" objekt zrusen, 0 zbyvaji
"Home Sweet Home"objekt zrusen, -1 zbyvaji
Null pointer assignment
```

Implicitní členské funkce

Oba tyto příklady mají společné to, že vyvolávají implicitní členské funkce definované automaticky a chování těchto funkcí neodpovídá tomuto konkrétnímu návrhu třídy. Jazyk C++ vytváří automaticky následující členské funkce:

- Implicitní konstruktor, jestliže žádný nedefinujete.
- Kopírovací konstruktor, pokud nějaký nedefinujete.
- Operátor přiřazení, jestliže žádný nedefinujete.
- Implicitní destruktor, jestliže žádný nedefinujete.
- Operátor získání adresy, pokud žádný nedefinujete.

Implicitní operátor získání adresy vrací adresu volajícího objektu (tedy hodnotu ukazatele `this`). To našim záměrům vyhovuje a touto členskou funkcí se již zabývat nebudeme. Implicitní destruktor nedělá nic a taky se jím nebudeme zabývat, pouze upozorníme, že daná třída za něj vytvořila náhradu. Ale ostatní si zaslouží další rozbor.

Implicitní konstruktor

Jestliže žádný konstruktor nevytvoříte, C++ automaticky vytvoří konstruktor implicitní. Předpokládejme například, že nadefinujete třídu `Klunk` a vynecháte konstruktory. Kompilátor v tom případě dodá následující implicitní konstruktor:

```
Klunk::Klunk() { } // standardní implicitní konstruktor
```

To znamená, že kompilátor dodá konstruktor bez parametrů, který nic nedělá. Je však potřeba, protože při vytvoření objektu je vždy vyvolán konstruktor.

```
Klunk lunk; // vyvolá implicitní konstruktor
```

Implicitní konstruktor vytvoří objekt `lunk` jako obyčejnou automatickou proměnnou. To znamená, že její hodnota při inicializaci nebude známá.

Jakmile nějaký konstruktor nadefinujete, nebude se již C++ s definováním implicitního konstruktoru obtěžovat. Jestliže budete chtít vytvářet objekty bez explicitní inicializace nebo budete-li chtít vytvářet pole objektů, budete muset implicitní konstruktor definovat explicitně. To je konstruktor bez parametrů, ale můžete jím nastavit konkrétní hodnoty:

```
Klunk::Klunk() // explicitní standardní konstruktor
{
    klunk_ct = 0;
    ...
}
```

I konstruktor s parametry může být implicitním konstruktorem, jestliže všechny jeho parametry obsahují implicitní hodnoty. Třída `Klunk` by například mohla mít následující vložený konstruktor:

```
Klunk(int n = 0) { klunk_ct = n; }
```

Implicitní konstruktor však můžete mít pouze jeden. Nemůžete tedy napsat:

```
Klunk() { klunk_ct = 0 }
Klunk(int n = 0) { klunk_ct = n; }  // nejednoznačné
```

Kopírovací konstruktor

Kopírovací konstruktor se používá při kopírování objektu do nově vytvořeného objektu. To znamená při inicializaci objektu, ne při obyčejném přiřazení. Kopírovací konstruktor třídy má tento prototyp:

```
Název_třídy(const Název_třídy &);
```

Všimněte si, že má jako parametr konstantní referenci na objekt třídy. Kopírovací konstruktor třídy `String` by například měl tento prototyp:

```
String(const String &);
```

O kopírovacím konstruktoru musíte vědět dvě věci: kdy se používá a co dělá.

Kdy se kopírovací konstruktor používá

Kopírovací konstruktor je vyvolán vždy, když je nový objekt vytvořen a inicializován pomocí existujícího objektu stejného typu. To nastává v několika situacích. Nejzřejmější je

explicitní inicializace objektu pomocí již existujícího objektu. Pokud je například motto objektem třídy String, pak následující čtyři deklarace vyvolají kopírovací konstruktor:

```
String ditto(motto);                    // volá String(const String &)
String metoo = motto;                   // volá String(const String &)
String also = String(motto);            // volá String(const String &)
String * pstring = new String(motto);   // volá String(const String &)
```

V závislosti na implementaci mohou prostřední dvě deklarace použít kopírovací konstruktor přímo k vytvoření objektů metoo a also, nebo ho mohou použít k vytvoření dočasných objektů, jejichž obsah je potom přiřazen objektům metoo a also. Poslední příklad inicializuje anonymní objekt pomocí objektu motto a adresu nového objektu přiřadí ukazateli pstring.

Méně zřejmé je, že kompilátor použije kopírovací konstruktor kdykoli program vytváří kopie nějakého objektu. Konkrétně se jedná o situace, kdy funkce předává objekt hodnotou nebo když objekt vrací. Pamatujte, že předávání hodnotou znamená vytvoření kopie původní proměnné. Kompilátor také použije kopírovací konstruktor, když vytváří dočasné objekty. Může například vygenerovat dočasný objekt třídy Vector pro mezisoučty při sčítání tří vektorů. Různé kompilátory generují dočasné soubory v různých okamžicích, ale při předávání objektů hodnotou a při jejich vracení volají všechny kopírovací konstruktor. Konkrétně volání funkce ve výpisu 11.4 vyvolalo kopírovací konstruktor:

```
showit(motto, 3); // vytvoří a předá kopii objektu motto
```

Program pomocí kopírovacího konstruktoru inicializuje proměnnou st, formální parametr typu String funkce showit().

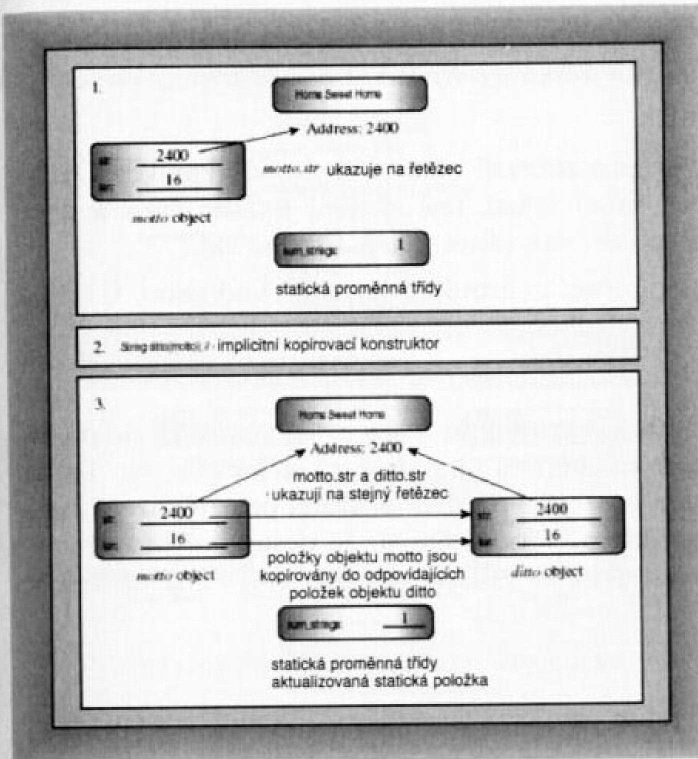
Skutečnost, že předávání objektu hodnotou vyvolá kopírovací konstruktor, je dobrým důvodem, proč raději použít předávání odkazem. Ušetří se čas potřebný pro vyvolání konstruktoru a místo pro uložení nového objektu.

Co kopírovací konstruktor dělá

Implicitní kopírovací konstruktor kopíruje postupně jednotlivé nestatické položky objektu. Každá položka je kopírována hodnotou. Ve výpisu 11.5 to vypadá následovně:

```
ditto.str = motto.str;
ditto.len = motto.len;
```

Jestliže je položka sama objektem nějaké třídy, použije se ke kopírování kopírovací konstruktor této třídy. Statické položky, jako například num_strings, to neovlivní, protože patří třídě jako celku a ne jednotlivým objektům. Činnost implicitního kopírovacího konstruktoru ilustruje obrázek 11.4.



Obrázek 11.4. Kopírování položek objektu

Kde jsme udělali chybu

Nyní jsme v situaci, kdy musíme pochopit tři podivnosti ve výpisech 11.4 a 11.5. První podivností je, že program zobrazil jeden vytvořený objekt, ale dva objekty zrušené. Vysvětlení je takové, že každý program vytvořil objekty dva, přičemž ten druhý byl vytvořen pomocí kopírovacího konstruktoru. Implicitní kopírovací konstruktor svou činnost nehlásí, takže vytvoření neoznámil. Tato podivnost v komunikaci má pouze kosmetickou povahu a spolehlivost programu neovlivňuje.

Druhou podivností je, že každý program nahlásil -1 zbylých objektů. Vysvětlení pro toto je, že implicitní konstruktor neovlivňuje statické položky. Kopírovací konstruktor tedy neaktualizoval čítač `num_strings`. Destruktor však aktualizaci čítače provádí a je vyvolán při každém rušení objektu bez ohledu na způsob jeho vytvoření. Tato podivnost problém představuje, protože to znamená, že program nepočítá objekty přesně. Řešením je vytvoření explicitního kopírovacího konstruktoru, který bude čítač aktualizovat:

```
String::String(const String & s)
{
    num_strings++;
    ...// zde je zbylý kód
}
```

Tip

Jestliže máte ve třídě statickou položku, která při vytvoření nového objektu změní svou hodnotu, vytvořte explicitní kopírovací konstruktor, který se postará o evidenci.

Třetí podivnost je nejzáludnější a nejnebezpečnější. Příznakem (u překladače Borland C++ 3.1) byla následující zpráva, která se objevila po ukončení programu:

```
Null pointer assignment
```

Microsoft Visual C++ 5.0 (v ladicím režimu) zobrazil okno s chybovou zprávou Debug Assertion `_CtrlsValidHeapPointer(pUserData)` failed. Jiné systémy mohou dát jiné zprávy, případně vůbec žádné. V programech se však přesto ukrývá stejné zlo.

Příčinou je skutečnost, že implicitní kopírovací konstruktor kopíruje hodnotou. Uvažujte například výpis 11.5. Vzpomeňte si, že výsledkem je:

```
ditto.str = motto.str;
```

Tento příkaz nezkopíruje řetězec: zkopíruje ukazatel do řetězce. To znamená, že po inicializaci objektu `ditto` objektem motto získáte dva ukazatele na stejný řetězec. To nepředstavuje problém, jestliže funkce `operator<<()` používá ukazatel pro zobrazení řetězce. Problém nastane při zavolání destruktoru. Vzpomeňte si, že destruktor třídy `String` uvolňuje paměť, na kterou ukazuje ukazatel `str`. Zrušení objektu `ditto` má následující účinek:

```
delete [] ditto.str;  // zruší řetězec, na který ukazuje ditto.str
```

Tento příkaz uvolní paměť obsazenou řetězcem „`Home Sweet Home`". Účinek zrušení objektu `motto` je tento:

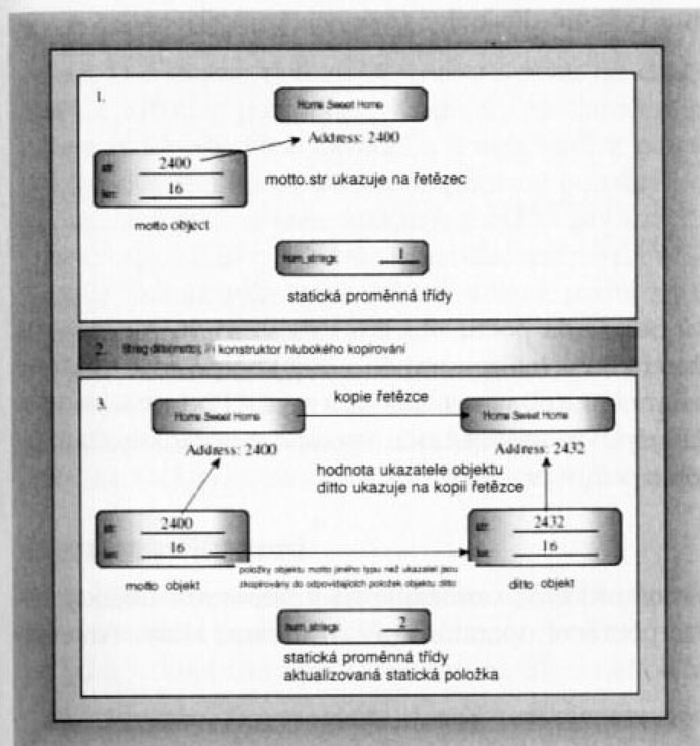
```
delete [] motto.str;  // nepředvídatelný účinek
```

Zde `motto.str` ukazuje na místo v paměti, které již bylo uvolněno, a to vede k nepředvídatelnému a pravděpodobně zhoubnému chování. V našem případě program vypsal varování o nulovém ukazateli, což je obvykle známkou špatné správy paměti.

Lékem je provedení *hluboké kopie*. To znamená, že místo pouhého kopírování adresy řetězce by kopírovací konstruktor měl řetězec duplikovat a položce `str` přiřadit adresu duplikátu. Takto bude mít každý objekt svůj řetězec a nebude odkazovat na řetězec druhého objektu. A každé volání destruktoru uvolní jiný řetězec a nebude se opakovaně snažit uvolnit řetězec již uvolněný. Zde je způsob, jak napsat takový kopírovací konstruktor třídy `String`:

```
String::String(const String & st)
   {
        num_strings++;              // aktualizace statické položky
        len = st.len;               // stejná délka
        str = new char [len + 1];   // alokace místa
        strcpy(str, st.str);        // kopírování řetězce na jiné místo
        cout << num_strings << ": \"" << str
             << "\" objekt vytvoren\n"; // PVI
   }
```

Definice kopírovacího konstruktoru je nutná z toho důvodu, že některé inicializované položky třídy jsou ukazateli na data a nikoli samotnými daty. Hluboké kopírování ilustruje obrázek 11.5.



Obrázek 11.5. Hluboké kopírování

Upozornění

Jestliže třída obsahuje ukazatele inicializované pomocí operátoru `new`, pak byste měli definovat kopírovací konstruktor, který zkopíruje data, na která tyto ukazatele ukazují a ne ukazatele samotné. Tomuto způsobu se říká hluboké kopírování.

Brzy již vyzkoušíme nový kopírovací konstruktor, ale nejdříve se podíváme ještě na jeden problém, který demonstruje krátký, ale chybný program ve výpisu 11.6. Program přiřazuje jeden objekt druhému.

Výpis 11.6. problem3.cpp

```
// problem3.cpp - přiřazuje jeden objekt druhému
// zkompilovat společně s strngl.cpp
#include <iostream>
using namespace std;
#include "strngl.h"
int main()
{
    String motto("Home Sweet Home");
    String ditto;          // implicitní konstruktor
    ditto = motto;         // přiřazení objektu
    cout << motto << "\n";
    cout << ditto << "\n";
    return 0;
}
```

Zde je výsledek běhu tohoto programu (přesné důsledky závisí na kompilátoru):

```
1: "Home Sweet Home" objekt vytvoren
2: "C++" objekt vytvoren
Home Sweet Home
Home Sweet Home
"Home Sweet Home" objekt zrusen, 1 zbyvaji
Sweet Home"objekt zrusen, 0 zbyvaji
Null pointer assignment
```

Zde alespoň počet zrušených objektů odpovídá počtu objektů vytvořených. Ale vypadá to, že jedno zrušení objektu způsobilo chybu v řetězci druhého objektu, protože zrušený řetězec je jiný. A další zpráva o nulovém ukazateli naruší náš klid mysli. Podrobnosti výstupu opět závisí na kompilátoru, ale i když váš výstup bude vypadat v pořádku, nachází se v programu skrytý, potenciálně nebezpečný problém.

Operátor přiřazení

Stejně jako ANSI C umožňuje přiřazovat struktury, umožňuje C++ přiřazovat objekty třídy. Děje se tak pomocí automatického přetížení operátoru přiřazení dané třídy. Prototyp vypadá následovně:

```
název_třídy & název_třídy::operator=(const název_třídy &);
```

Znamená to, že parametrem i návratovým typem je reference na objekt třídy. Zde je například prototyp pro třídu `String`:

```
String & String::operator=(const String &);
```

Kdy se použije operátor přiřazení

Přetížený operátor přiřazení se použije, jestliže přiřadíte objekt jinému, již existujícímu objektu:

```
string motto("Home Sweet Home");
string ditto;
ditto = motto; // použije přetížený operátor přiřazení
```

Při inicializaci objektu není jeho použití nutné:

```
string metoo = ditto; // použije se kopírovací konstruktor
```

Zde je `metoo` nově vytvořený objekt inicializovaný hodnotami položek objektu `ditto`, proto se použije kopírovací konstruktor. Jak však bylo zmíněno dříve, implementace mají možnost zpracovat tento příkaz ve dvou krocích: pomocí kopírovacího konstruktoru vytvořit dočasný objekt a potom pomocí přiřazení zkopírovat do nového objektu hodnoty. To znamená, že inicializace vždy vyvolá kopírovací konstruktor a formáty používající operátor = mohou také vyvolat operátor přiřazení.

Co dělá operátor přiřazení

Podobně jako kopírovací konstruktor, také implicitní implementace operátoru přiřazení provádí kopírování po položkách. Jestliže je položka sama objektem nějaké třídy, program provede kopírování této konkrétní položky pomocí operátoru přiřazení definovaného pro tuto třídu. Statické položky nejsou ovlivněny.

Kde jsme udělali chybu

Výpis 11.6 ukázal dvě zvláštnosti. Za prvé, při druhém volání destruktoru byl zobrazený řetězec záhadně pozměněn. Za druhé, při skončení program vypsal zprávu o nulovém přiřazení ukazatele. Obojí svědčí o špatné správě paměti. Problém je stejný, jako jsme viděli u kopírovacího konstruktoru: kopírovaní položky po položce zkopíruje hodnoty ukazatelů namísto dat, na která ukazují. Když je zavolán destruktor objektu `ditto`, zruší řetězec „`Home Sweet Home`", a když je zavolán destruktor objektu `motto`, snaží se zrušit již zrušený řetězec. Jak již bylo řečeno dříve, účinek snahy zrušit již zrušená data je nepředvídatelný. Vymazání paměti může také změnit její obsah. Ve výpisech 11.4 a 11.5 první příkaz `delete` řetězec nezměnil, protože při druhém volání destruktoru byl řetězec zobrazen správně. Ve výpisu 11.6 však první příkaz `delete` řetězec změnil. Jak někteří rádi poukazují, jestliže účinek některé operace není definován, kompilátor si může dělat co chce, například zobrazit Deklaraci nezávislosti nebo zbavit pevný disk nevzhledných souborů.

Oprava přiřazení

Řešit problémy zapříčiněné nevyhovujícím implicitním operátorem přiřazení znamená definovat vlastní operátor přiřazení, který provede hlubokou kopii. Implementace je podobná jako u kopírovacího konstruktoru, ale s několika rozdíly.

1. Protože cílový objekt již může odkazovat na dříve přiřazená data, měla by funkce pomocí příkazu `delete []` předchozí závazky uvolnit.
2. Funkce by měla objekt ochránit před přiřazením sobě samému; v opačném případě by mohlo výše popsané uvolnění paměti smazat obsah objektu ještě před přiřazením.
3. Funkce vrací referenci na volající objekt.

Díky vrácení objektu může funkce emulovat způsob řetězení přiřazení obyčejných vestavěných typů. To znamená, jestliže `A`, `B` a `C` jsou objekty třídy `String`, můžete zapsat následující příkaz:

```
A = B = C;
```

Ve funkčním zápisu to značí následující:

```
A.operator=(B.operator=(C));
```

Návratová hodnota výrazu `B.operator=(C)` se tedy stane parametrem funkce `A.operator=()`. Protože návratová hodnota je reference na objekt třídy `String`, je typ parametru správný.

Zde byste mohli napsat operátor přiřazení pro třídu String:

```
String & String::operator=(const String & st)
{
        if (this == &st)              // objekt přiřazený sám sobě
            return *this;             // hotovo
        delete [] str;                // uvolní starý řetězec
        len = st.len;
        str = new char [len + 1];    // získá místo pro nový řetězec
        strcpy(str, st.str);         // zkopíruje řetězec
        return *this;                 // vrátí referenci na volající objekt
}
```

Nejdříve kód zkontroluje, zda objekt není přiřazen sám sobě. Zjistí, zda adresa na pravé straně přiřazení (`&s`) je stejná jako adresa přijímajícího objektu (`this`). Pokud ano, vrátí hodnotu `*this` a skončí. Možná si z kapitoly 10 vzpomínáte, že operátor přiřazení patří do skupiny operátorů, které mohou být přetíženy pouze pomocí členské funkce.

V ostatních případech funkce pokračuje uvolněním paměti, na kterou ukazoval ukazatel `str`. Důvodem je, že krátce na to bude ukazateli `str` přiřazena adresa nového řetězce. Kdybyste nejdříve nepoužili operátor `delete`, předchozí řetězec by zůstal v paměti. Protože program již nemá ukazatel na původní řetězec, bylo by to plýtvání pamětí.

Dále program pokračuje jako kopírovací konstruktor, přidělí dostatek místa pro nový řetězec a potom zkopíruje řetězec z pravého objektu do nového místa.

Na konci vrátí výraz `*this` a skončí.

Přiřazení nevytváří nový objekt, nemusíte tedy hodnotu statické položky `num_strings` upravovat.

Nová, vylepšená třída String

Nyní, když jsme o trochu moudřejší, přepracujeme třídu `String`. Nejdříve přidáme kopírovací konstruktor a operátor přiřazení, které jsme právě probrali, aby třída správně pracovala s pamětí, používanou objekty třídy. Když již víme, kdy se objekty vytváří a ruší, můžeme konstruktory a destruktory umlčet, takže při používání již nebudou nic hlásit. Jestliže již implicitní konstruktor nebude sledovat konstruktory při práci, zjednodušíme ho tak, aby místo řetězce s hodnotou „`C++`" vytvořil prázdný řetězec. Protože již také víme jak fungují statické položky třídy, můžeme odstranit prostředek na počítání objektů.

Dále třídě přidáme několik vlastností. Užitečná třída `String` by měla zahrnout veškerou funkčnost řetězcových funkcí standardní knihovny `cstring`, my ale přidáme jen tolik, abychom ukázali způsob jejich vytvoření. (Nezapomínejte, že tato třída `String` je pouze ilustrativní příklad a že standardní třída řetězců v C++ je mnohem obsažnější.) Konkrétně přidáme následující metody:

```
int length () const { return len; }
friend bool operator>(const String &st1, const String &st2);
friend bool operator<(const String &st, const String &st2);
friend bool operator==(const String &st, const String &st2);
friend operator>>(istream & is, String & st);
```

První nová metoda vrátí délku uloženého řetězce a další tři umožní porovnávat řetězce. Funkce `operator>()` například vrátí hodnotu `true`, jestliže první řetězec bude v abecedním pořadí za druhým řetězcem (přesněji řečeno, v porovnávací posloupnosti počítače). Nejjednodušší implementace funkcí na porovnání řetězců spočívá ve využití standardní funkce `strcmp()`, která vrací zápornou hodnotu, jestliže první parametr je abecedně před druhým, nulu, jsou-li řetězce shodné, a kladnou hodnotu, pokud první řetězec následuje abecedně za druhým řetězcem. Funkci `strcmp()` je možné využít například takto:

```
bool operator>(const String &st1, const String &st2)
{
    if (strcmp(st1.str, st2.str) > 0)
```

```
            return true;
        else
            return false;
    }
```

Jestliže z porovnávacích funkcí učiníme spřátelené, usnadníme porovnání mezi objekty třídy `String` a standardními řetězci jazyka C. Předpokládejme například, že `answer` je objekt třídy `String` a že máte následující kód:

```
if ("love" == answer)
```

Tento výraz se převede na následující:

```
if (operator==("love", answer))
```

Kompilátor potom pomocí jednoho z konstruktorů převede kód na tento tvar:

```
if (operator==(String("love"), answer))
```

A toto již odpovídá prototypu operátoru rovnosti.

Nový standardní konstruktor zasluhuje pozornost. Bude vypadat následovně:

```
    String::String()
    {
        len = 0;
        str = new char[1];
        str[0] = '\0';          // implicitní řetězec
    }
```

Možná se divíte, proč kód obsahuje tento příkaz:

```
str = new char[1];
```

a ne příkaz následující:

```
str = new char;
```

Oba tvary přidělí stejné množství paměti. Rozdíl je v tom, že první tvar je kompatibilní s destruktorem třídy, zatímco druhý ne. Vzpomeňte si, že destruktor obsahuje tento kód:

```
delete [] str;
```

Použití příkazu `delete` s hranatými závorkami je kompatibilní s ukazateli inicializovanými pomocí operátoru `new` s hranatými závorkami a s nulovým ukazatelem. Účinek na ukazatele inicializované jiným způsobem není definován.

Než se podíváme na nové výpisy, uvažujme ještě jednu věc. Předpokládejme, že chcete zkopírovat obyčejný řetězec do objektu třídy `String`. Předpokládejme například, že jste přečetli řetězec pomocí funkce `getline()` a chcete jej uložit do objektu třídy `String`. Metody třídy umožňují napsat následující kód:

```
    String name;
    char temp[40];
    cin.getline(temp, 40);
    name = temp; // použití konstruktoru k převodu typu
```

Takové řešení by však nebylo při častém používání uspokojivé. Abychom věděli proč, zrekapitulujme si, jak poslední příkaz funguje:

1. Program pomocí konstruktoru `String(const char *)` vytvoří dočasný objekt třídy `String`, který bude obsahovat kopii řetězce uloženého v poli `temp`. Vzpomeňte si (kapitola 10), že konstruktor s jedním parametrem slouží jako konverzní funkce.
2. Program pomocí funkce `String & String::operator=(const String &)` zkopíruje informace z dočasného objektu do objektu name.
3. Program zavolá destruktor `~String()` a zruší dočasný objekt.

Vyšší účinnosti jednoduše dosáhnete, přetížíte-li operátor přiřazení tak, aby pracoval s obyčejnými řetězci. Tím dojde k odstranění zbytečných kroků pro vytvoření a zrušení dočasného objektu. Zde je jeden možný způsob implementace:

```
String & String::operator=(const char * s)
{
      delete [] str;
      len = strlen(s);
      str = new char[len + 1];
      strcpy(str, s);
      return *this;
}
```

Jako obvykle budete muset uvolnit paměť spravovanou ukazatelem str a přidělit dostatek paměti pro nový řetězec.

Na výpisu 11.7 je opravená deklarace třídy.

Výpis 11.7. strng2.h

```
// strng2.h - definice třídy String
#ifndef _STRNG2_H_
#define _STRNG2_H_
#include <iostream>
using namespace std;
class String
{
private:
    char * str;                          // ukazatel na řetězec
    int len;                             // délka řetězce
public:
    String(const char * s);            // konstruktor
    String();                            // implicitní konstruktor
    String(const String & st);
    ~String();                           // destruktor
    int length() const { return len; }
// přetížené operátory
    String & operator=(const String & st); // operátor přiřazení
    String & operator=(const char * s);    // operátor přiřazení č. 2
// spřátelené funkce
    friend bool operator>(const String &st1, const String &st2);
    friend bool operator<(const String &st, const String &st2);
    friend bool operator==(const String &st, const String &st2);
    friend ostream & operator<<(ostream & os, const String & st);
```

```
    friend istream & operator>>(istream & is, String & st);
};
#endif
```

Kompatibilita:

Možná máte kompilátor, který neimplementuje typ `bool`. V tom případě můžete místo typu `bool` použít typ `int` s hodnotami `0` místo `false` a `1` místo `true`.

Na dalším výpisu 11.8 jsou upravené definice metod.

Výpis 11.8. strng2.cpp

```
// strng2.cpp - metody třídy String
#include <iostream>
#include <cstring>
using namespace std;
#include "strng2.h"
// metody třídy
String::String(const char * s)  // vytvoření objektu třídy String
                                //jazyka C z řetězce
{
    len = strlen(s);
    str = new char[len + 1];    // přidělí paměť
    strcpy(str, s);             // inicializuje ukazatel
}
String::String()                // implicitní konstruktor
{
    len = 0;
    str = new char[1];
    str[0] = '\0';              // implicitní řetězec
}
String::String(const String & st)// kopírovací konstruktor
{
    len = st.len;
    str = new char[len + 1];
    strcpy(str, st.str);
}
String::~String()                 // destruktor
{
    delete [] str;                // nutné
}

    // přiřazení jednoho objektu třídy String druhému
String & String::operator=(const String & st)
{
    if (this == &st)
        return *this;
    delete [] str;
```

```
    len = st.len;
    str = new char[len + 1];
    strcpy(str, st.str);
    return *this;
}

    // přiřazení řetězce jazyka C objektu třídy String
String & String::operator=(const char * s)
{
    delete [] str;
    len = strlen(s);
    str = new char[len + 1];
    strcpy(str, s);
    return *this;
}

    // true jestliže st1 následuje v porovnávací posloupnosti za st2
bool operator>(const String &st1, const String &st2)
{
    if (strcmp(st1.str, st2.str) > 0)
        return true;
    else
        return false;
}

    // true jestliže st1 předchází v porovnávací posloupnosti st2
bool operator<(const String &st1, const String &st2)
{
    if (strcmp(st1.str, st2.str) < 0)
        return true;
    else
        return false;
}
// přátelé
    // true jestliže st1 je roven st2
bool operator==(const String &st1, const String &st2)
{
    if (strcmp(st1.str, st2.str) == 0)
        return true;
    else
        return false;
}
    // zobrazení řetězce
ostream & operator<<(ostream & os, const String & st)
{
    os << st.str;
    return os;
}
    // rychlý a nečistý vstup řetězce
istream & operator>>(istream & is, String & st)
{
    char temp[80];
```

```
    is.get(temp, 80);
    if (is)
        st = temp;
    while (is && is.get() != '\n')
        continue;
    return is;
}
```

Přetížený operátor >> poskytuje jednoduchý způsob, jak načíst řetězec z klávesnice do objektu třídy `String`. Není však odolný vůči chybám, protože předpokládá vstup maximálně 79 znaků. Pamatujte, že hodnota objektu `istream` v podmínce `if` se vyhodnotí jako false v případě chyby při vstupu, jako je například znak konec souboru, nebo pokud funkce `get(char *, int)` načte prázdný řádek.

Pojďme vyzkoušet naši třídu pomocí krátkého programu, který umožňuje zadat několik řetězců. Program nechá uživatele vložit několik pořekadel, uloží je do objektů třídy `String`, zobrazí, a oznámí, který řetězec je nejkratší a který je abecedně první. Program je ve výpisu 11.9.

Výpis 11.9. sayings1.cpp

```
// sayings1.cpp - použití rozšířené třídy String
// zkompilovat společně s strng2.cpp
#include <iostream>
using namespace std;
#include "strng2.h"
const ArSize = 10;
const MaxLen = 81;
int main()
{
    String name;
    cout <<"Dobry den,jak se jmenujete?\n>> ";
    cin >> name;
    cout << name << ", zadejte prosim maximalne " << ArSize
        << " kratkych rceni <prazdny radek pro ukonceni >:\n";
    String sayings[ArSize];        // pole objektů
    char temp[MaxLen];              // paměť pro dočasný řetězec
    int i;
    for (i = 0; i < ArSize; i++)
    {
        cout << i+1 << ": ";
        cin.get(temp, MaxLen);
        while (cin && cin.get() != '\n')
            continue;
        if (!cin || temp[0] == '\0')  // prázdný řádek?
            break;                   // i není zvýšeno o 1
        else
            sayings[i] = temp;    // přetížené přiřazení
    }
    int total = i;                  // celkový počet přečtených řádků
    cout << "Zadali jste tato rceni: \n";
```

```
        for (i = 0; i < total; i++)
            cout << sayings[i] << "\n";
        int shortest = 0;
        int first = 0;
        for (i = 1; i < total; i++)
        {
            if (sayings[i].length() <  sayings[shortest].length())
                shortest = i;
            if (sayings[i] < sayings[first])
                first = i;
        }
    cout << "Nejkratsi rceni:\n" << sayings[shortest] << "\n";
    cout << "Prvni rceni podle abecedy:\n" << sayings[first] << "\n";
    return 0;
}
```

Kompatibilita:

Starší verze funkce `get(char *, int)` nevrátí hodnotu `false` při načtení prázdného řádku. U těchto verzí však bude při zadání prázdného řádku první znak řetězce nulový. Tento příklad používá následující kód:

```
if (!cin || temp[0] == '\0')      // prázdný řádek?
    break;                        // i neni zvýšeno o 1
```

Jestliže implementace odpovídá aktuálnímu standardu, pak první test příkazu `if` odhalí načtení prázdného řádku, zatímco druhý test odhalí prázdný řádek u starších implementací.

Program požádá uživatele o zadání maximálně deseti pořekadel. Každé pořekadlo je načteno do dočasného pole znaků a potom zkopírováno do objektu třídy `String`. Pokud uživatel zadá prázdný řádek, příkaz `break` ukončí vstupní smyčku. Program potvrdí vstup a potom pomocí členských funkcí `length()` a `operator<()` najde nejkratší řetězec a první řetězec podle abecedy. Zde je ukázka výstupu programu:

```
  Dobry den jak se jmenujete?
  >> Misty Gutz
  Misty Gutz, zadejte prosim maximalne 10 kratkych rceni <prazdny radek
pro ukonceni >:
  1: poslouchat na slovo
  2: zahodit flintu do zita
  3: spat jako dudek
  4: sejde z oci, sejde z mysli
  5: pilny jako vcelicka
  6: mit na ruzich ustlano
  7:
  Zadali jste tato rceni: poslouchat na slovo  zahodit flintu do zita
spat jako dudek  sejde z oci, sejde z mysli  pilny jako vcelicka  mit na
ruzich ustlano  Nejkratsi rceni: spat jako dudek  Prvni rceni podle abece-
dy:
  mit na ruzich ustlano
```

Kdy se v konstruktoru používá operátor new

Již jste si všimli, že při inicializaci ukazatelů objektu pomocí operátoru new musíte být mimořádně opatrní. Konkrétně byste měli postupovat takto:

- Jestliže v konstruktoru inicializujete pomocí operátoru new ukazatel, měli byste v destruktoru použít operátor delete.
- Použití operátorů new a delete by měla být kompatibilní. Používejte new společně s delete a new [] společně s delete [].
- Pokud existuje více konstruktorů, měli byste ve všech použít stejnou formu operátoru new – buď všechny s hranatými závorkami nebo všechny bez závorek. Destruktor existuje pouze jeden, takže všechny konstruktory s ním musí být kompatibilní. Je však povoleno v jednom konstruktoru inicializovat ukazatel pomocí operátoru new a v jiném ukazatel nastavit na nulovou hodnotu (NULL nebo 0), protože operace delete s nulovým ukazatelem je možná.

NULL nebo 0?

Nulový ukazatel může být vyjádřen jako 0 nebo jako NULL, což je symbolická konstanta definovaná v některých hlavičkových souborech jako 0. Programátoři v jazyce C používají místo 0 často výraz NULL jako vizuální připomnění, že hodnota je hodnotou ukazatele, stejně jako používají výraz '\0' místo 0 pro prázdný znak jako vizuální připomnění, že touto hodnotou je znak. Tradice jazyka C++ však upřednostňuje používání hodnoty 0 namísto ekvivalentu NULL.

- Měli byste definovat kopírovací konstruktor, který inicializuje jeden objekt pomocí druhého a provádí přitom hlubokou kopii. Konstruktor by se měl podobat tomuto příkladu:

```
String::String(const String & st)
{
 num_strings++;        // v případě potřeby aktualizuje
                       // statickou proměnnou
                        len = st.len;     // nastaví stejnou délku
     str = new char [len + 1];   // přidělí místo
     strcpy(str, st.str);        // zkopíruje řetězce na nové místo
}
```

- Kopírovací konstruktor by měl především přidělit místo pro kopírovaná data a měl by zkopírovat samotná data, ne pouze jejich adresy. Také by měl aktualizovat statické členské proměnné, jejichž hodnotu bude proces ovlivňovat. Měli byste definovat operátor přiřazení, který pomocí hluboké kopie zkopíruje jeden objekt do druhého. Metoda třídy se bude většinou podobat tomuto příkladu:

```
String & String::operator=(const String & st)
{
     if (this == &st)          // objekt přiřazený sám sobě
          return *this;        // konec
     delete [] str;            // uvolní starý řetězec
```

```
        len = st.len;
        str = new char [len + 1]; // získá místo pro nový řetězec
        strcpy(str, st.str);      // zkopíruje řetězec
        return *this;             // vrátí referenci na volající objekt
    }
```

- Metoda by měla zkontrolovat, zda objekt není přiřazen sám sobě, měla by uvolnit paměť, na kterou předtím ukazoval ukazatel, měla by zkopírovat data, ne pouze jejich adresy a měla by vrátit referenci na volající objekt.

Následující výňatek ukazuje dva chybné příklady a jeden příklad použití správného konstruktoru:

```
String::String()
{
  str = "default string";        // chybí operátor new []
  len = strlen(str);
}
String::String(const char * s)
{
  len = strlen(s);
  str = new char;                // chybí []
  strcpy(str, s);                // není přiděleno místo
}
String::String(const String & st)
{
  len = st.len;
  str = new char[len + 1];       // v pořádku, přidělí místo
  strcpy(str, st.str);           // v pořádku, zkopíruje hodnotu
}
```

První konstruktor neinicializuje ukazatel str pomocí operátoru new. Destruktor volaný pro implicitní objekt použije na ukazatel str operátor delete. Výsledek je nepředvídatelný, ale pravděpodobně bude špatný. Správné by bylo použít něco z následujícího:

```
String::String()
{
  len = 0;
  str = new char[1];       // použití new s []
  str[0] = '\0';
}
String::String()
{
  len = 0;
  str = NULL;         // nebo str = 0;
}
String::String()
{
  static const char * s = "C++";   // inicializováno právě jednou
  len = strlen(s);
  str = new char[len + 1];         // použití new společně s []
  strcpy(str, s);
}
```

Druhý konstruktor v původním výňatku použije operátor new, ale nepožádá o dostatečné množství paměti; operátor new tedy vrátí blok, kde bude místo pouze pro jeden znak. Pokus zkopírovat na toto místo delší řetězec způsobí problémy s pamětí. Také použití operátoru new bez hranatých závorek je v rozporu se správným tvarem ostatních konstruktorů.

Třetí konstruktor je v pořádku.

Nakonec, zde je destruktor, který s předcházejícími konstruktory nebude správně fungovat:

```
String::~String()
{
    delete str;   // mělo by být delete [] str;
}
```

Destruktor nesprávně používá operátor delete. Protože konstruktory požadují pole znaků, destruktor by měl rušit pole.

Používání ukazatelů na objekty

Programy v C++ používají velmi často ukazatele na objekty, pojďme si je tedy trochu procvičit. Program ve výpisu 11.9 pomocí hodnot indexů pole zaznamenal nejkratší řetězec a první řetězec podle abecedy. Jiný přístup spočívá v použití ukazatelů na aktuální vedoucí těchto kategorií. Tento přístup je implementován ve výpisu 11.10, kde jsou použity dva ukazatele na objekty třídy String. Na počátku ukazatel shortest ukazuje na první objekt pole. Pokaždé, když program narazí na objekt s kratším řetězcem, nastaví ukazatel shortest na tento objekt. Podobně ukazatel first eviduje první řetězec v abecedním pořadí. Všimněte si, že tyto dva ukazatele nevytváří nové objekty, pouze ukazují na existující objekty. Není tedy třeba pro ně přidělovat další paměť pomocí operátoru new.

Z důvodu různorodosti program používá ukazatel, který zaznamená nově vytvořený objekt:

```
String * favorite = new String(sayings[choice]);
```

Zde ukazatel favorite představuje jediný přístup k bezejmennému objektu vytvořenému operátorem new. Tato konkrétní syntaxe znamená inicializaci nového objektu třídy String pomocí objektu sayings[choice]. Dojde k vyvolání kopírovacího konstruktoru, protože typ parametru kopírovacího konstruktoru (const String &) odpovídá inicializační hodnotě (sayings[choice]). Náhodnou hodnotu program vybírá pomocí funkcí srand(), rand() a time().

Inicializace objektu pomocí operátoru new

Obecně platí, že pokud *název_třídy* je třída a *hodnota* je typu *název_typu*, pak příkaz

```
název_třídy * pclass = new název_třídy(hodnota);
```

vyvolá konstruktor

```
název_třídy(název_typu);
```

Mohou proběhnout i nějaké triviální konverze:

název_třídy(const *název_typu* &);

Provedou se také obvyklé konverze vyvolané porovnáním prototypu, jako například převod z typu int na typ double, pokud nedojde k nejednoznačnosti. Inicializace ve formátu

název_třídy *ptr = new *název_třídy*;

vyvolá implicitní konstruktor.

Výpis 11.10. sayings2.cpp

```
// sayings2.cpp - použití ukazatelů na objekty
// zkompilovat společně s strng2.cpp
#include <iostream>
using namespace std;
#include <cstdlib>              // (nebo stdlib.h) pro rand(), srand()
#include <ctime>                // (nebo time.h) pro time()
#include "strng2.h"
const ArSize = 10;
const MaxLen = 81;
int main()
{
  String name;
  cout <<"Dobry den, jak se jmenujete?\n>> ";
  cin >> name;
  cout << name << ", zadejte prosim maximalne " << ArSize
    << " kratkych rceni <prazdny radek pro ukonceni >:\n";
  String sayings[ArSize];
  char temp[MaxLen];                   // paměť pro dočasný řetězec
  int i;
  for (i = 0; i < ArSize; i++)
  {
    cout << i+1 << ": ";
    cin.get(temp, MaxLen);
    while (cin && cin.get() != '\n')
      continue;
    if (!cin || temp[0] == '\0') // prázdný řádek?
      break;                       // i není zvýšeno o 1
    else
      sayings[i] = temp;           // přetížené přiřazení
  }
  int total = i;                     // celkový počet přečtených řádků
  cout << "Zadali jste tato rceni:\n";
  for (i = 0; i < total; i++)
    cout << sayings[i] << "\n";
// použití ukazatelů k evidenci nejkratšího a prvního řetězce
  String * shortest = &sayings[0];   // inicializace prvním objektem
  String * first = &sayings[0];
  for (i = 1; i < total; i++)
  {
```

```
        if (sayings[i].length() < shortest->length())
            shortest = &sayings[i];
        if (sayings[i] < *first)        // porovnání hodnot
            first = &sayings[i];        // přiřadí adresu
    }
    cout << "Nejkratsi rceni:\n" << * shortest << "\n";
    cout << "Prvni rceni podle abecedy:\n" << * first << "\n";
    srand(time(0));
    int choice = rand() % total;       // náhodný výběr indexu
  // použiti operátoru new k vytvoření objektu třídy String a jeho inicia-
lizace
    String * favorite = new String(sayings[choice]);
    cout << "Moje oblibene rceni:\n" << *favorite << "\n";
    delete favorite;
    return 0;
}
```

Kompatibilita:

Starší implementace mohou vyžadovat hlavičkové soubory `stdlib.h` namísto `cstdlib` a `time.h` místo `ctime`.

Ukázka běhu programu:

```
Dobry den, jak se jmenujete? >> Kirt Rood
Kirt Rood, zadejte prosim maximalne 10 kratkych rceni <prazdny radek pro
ukonceni >:
1: v nouzi poznas pritele
2: jedna vlastovka jaro nedela
3: jak si usteles, tak si lehnes
4: dvakrat mer, jednou rez
5: lepe pozde nez nikdy
6: darovanemu koni na zuby nehled
7:
Zadali jste tato rceni:
v nouzi poznas pritele
jedna vlastovka jaro nedela
jak si usteles, tak si lehnes
dvakrat mer, jednou rez
lepe pozde nez nikdy
darovanemu koni na zuby nehled
Nejkratsi rceni:
lepe pozde nez nikdy
Prvni rceni podle abecedy:
darovanemu koni na zuby nehled
Moje oblibene rceni:
jak si usteles, tak si lehnes
```

Za povšimnutí stojí několik myšlenek, týkajících se používání ukazatelů na objekty. (Viz také obrázek 11.6)

Obrázek 11.6. Ukazatele na objekty

- Deklarujete ukazatel na objekt obvyklou notací:

```
String * glamour;
```

- Ukazatel můžete inicializovat pomocí již existujícího objektu:

```
String * first = &sayings[0];
```

- Ukazatel můžete inicializovat pomocí operátoru `new`. Tímto způsobem se vytvoří nový objekt:

```
String * favorite = new String(sayings[choice]);
```

- Použití operátoru `new` společně s třídou vyvolá odpovídající konstruktor třídy a inicializuje nově vytvořený objekt:

```
// vyvolá implicitní konstruktor
String * gleep = new String;

// vyvolá konstruktor String(const char *)
String * glop = new String("my my my");

// vyvolá konstruktor String(const String &)
String * favorite = new String(sayings[choice]);
```

- Pomocí operátoru -> získáte přístup k metodě třídy přes ukazatel:

```
if (sayings[i].length() < shortest->length())
```

- Pomocí operátoru dereference (*) u ukazatele na objekt získáte hodnotu objektu:

```
if (sayings[i] < *first)   // porovná hodnoty objektů
    first = &sayings[i];   // přiřadí adresu objektu
```

Opakování technik

Dosud jste poznali několik programovacích technik, řešících různé problémy s třídami, a možná máte problém s jejich zapamatováním. Proto je zrekapitulujeme a uvedeme, kdy se používají.

Přetížení operátoru <<

Chcete-li předefinovat operátor << tak, abyste ho mohli použít spolu s objektem `cout` pro zobrazení obsahu objektu, definujte spřátelenou funkci následujícího tvaru:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
  os << … ; // zobrazí obsahu objektu
  return os;
}
```

Zde *c_name* zastupuje název třídy. Pokud třída obsahuje veřejné metody vracející požadovaný obsah, můžete tyto metody použít ve funkci operátoru a obejít se bez spřáteleného stavu.

Konverzní funkce

Jestliže chcete převést jednoduchou hodnotu na typ třídy, vytvořte konstruktor třídy s následujícím prototypem:

```
c_name(type_name value);
```

Zde `c_name` zastupuje název třídy a `type_name` typ, který chcete konvertovat.

Chcete-li konvertovat typ třídy na nějaký jiný typ, vytvořte členskou funkci s následujícím prototypem:

```
operator type_name();
```

Ačkoli funkce nemá deklarován návratový typ, měla by vracet hodnotu požadovaného typu.

Pamatujte, že konverzní funkce musíte používat opatrně. Pomocí klíčového slova `explicit` v deklaraci konstruktoru zabráníte implicitním konverzím.

Třídy s konstruktory používajícími operátor new

U tříd používajících pro přidělení paměti operátor `new` musíte v návrhu dbát na některá bezpečnostní opatření. (Před chvílí jsme je již shrnuli, ale je velmi důležité si tato pravidla zapamatovat, především proto, že kompilátor je nezná a nemůže vaše chyby odchytit.)

1. Každá položka ukazující na paměť přidělenou operátorem `new` by měla být v destruktoru třídy použita s operátorem `delete`. Tím dojde k uvolnění přidělené paměti.
2. Jestliže destruktor pro uvolnění paměti použije operátor `delete` na ukazatel, který je členem třídy, pak by všechny konstruktory této třídy měly tento ukazatel inicializovat buď pomocí operátoru `new` nebo nastavením na nulu.

3. V konstruktorech byste měli používat buď tvar `new []` nebo `new`, ale nemíchat obojí. Pokud jste použili `new []`, měli byste v destruktoru použít `delete []`, jestliže jste v konstruktorech použili `new`, použijte v destruktoru `delete`.
4. Nekopírujte ukazatel do již existující paměti. Chcete-li přidělit novou paměť, definujte kopírovací konstruktor. Tímto způsobem umožníte, aby program inicializoval jeden objekt třídy jiným. Prototyp konstruktoru by měl mít následující tvar:

   `název_třídy(const název_třídy &)`

5. Měli byste definovat členskou funkci třídy přetěžující operátor přiřazení, v následujícím tvaru (zde `c_pointer` je položka třídy `c_name` typu ukazatel na `type_name`):

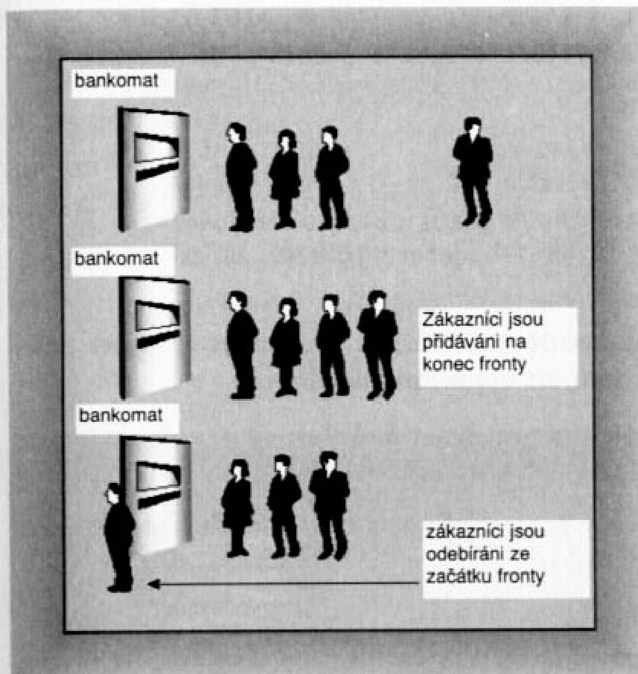
```
c_name & c_name::operator=(const c_name & cn)
{
    if (this == & cn_)
        return *this:    // konec, pokud objekt přiřazuje sám sobě
    delete c_pointer:
    c_pointer = new type_name[size]:
    // zkopíruje data, na která ukazuje cn.c_pointer na
    // místo, kam ukazuje c_pointer
    ...
    return *this:
}
```

Simulace fronty

Nové vědomosti o třídách použijeme na programátorský problém. Banka Bank of Heather chce instalovat bankovní automat v odchodním domě Food Heap. Vedení supermarketu má obavy, že fronty u automatu budou narušovat chod obchodu a chce stanovit omezení na počet lidí ve frontě u automatu. Lidé z banky chtějí odhad doby čekání ve frontě. Vaším úkolem je připravit program simulující tuto situaci, aby vedení banky zjistilo možné účinky automatu.

Problém lze nejpřirozeněji vyjádřit pomocí fronty zákazníků. Fronta je abstraktní datový typ (ADT), který obsahuje seřazenou posloupnost prvků. Nové položky jsou přidávány na konec fronty, odebírány mohou být ze začátku. Fronta se trochu podobá zásobníku s tím rozdílem, že přidávání i odebírání probíhá u zásobníku na stejném konci. Zásobník tedy představuje strukturu typu LIFO (last-in-first-out, poslední dovnitř, první ven), zatímco fronta strukturu typu FIFO (first-in-first-out, první dovnitř, první ven). Koncepčně je fronta něco jako stání u pokladny nebo u bankovního automatu, takže se ideálně hodí pro tento úkol. Jednou částí vašeho projektu tedy bude definovat třídu `Queue`.

Položkami fronty budou zákazníci. Zástupce banky vám sdělí, že průměrně třetina zákazníků bude odbavena do minuty, další třetina do dvou minut a poslední třetina do tří minut. Zákazníci navíc přicházejí v náhodných intervalech, ale průměrný počet zákazníků za hodinu je víceméně neměnný. V dalších dvou částech projektu budete muset navrhnout třídu zastupující zákazníky a sestavit program simulující vzájemný vztah zákazníků a fronty (viz obrázek 11.7).



Obrázek 11.7. Fronta

Třída Queue

Prvním úkolem je návrh třídy `Queue`. Nejdříve si uděláme seznam potřebných vlastností takové fronty:

- Fronta obsahuje seřazenou posloupnost položek.
- Fronta může obsahovat omezený počet položek. – Mělo by být možné vytvořit prázdnou frontu.
- Mělo by být možné zjistit, zda je fronta prázdná. – Mělo by být možné zjistit, zda je fronta plná.
- Měla by existovat možnost přidat položku na konec fronty.
- Měla by existovat možnost odebrat položku ze začátku fronty.
- Mělo by být možné zjistit počet prvků ve frontě.

Jako obvykle při návrhu třídy budete muset vytvořit veřejné rozhraní a soukromou implementaci.

Rozhraní

Podle vlastností fronty se nabízí pro třídu `Queue` následující veřejné rozhraní:

```
class Queue
{
  enum {Q_SIZE = 10};
private:
// privátní část bude vytvořena později
```

```
public:
    Queue(int qs = Q_SIZE);  // vytvoří frontu s omezením qs
    ~Queue();
    bool isempty() const;
    bool isfull() const;
    int queuecount() const;
    bool enqueue(const Item &item);  // přidá položku na konec
    bool dequeue(Item &item);        // odebere položku ze začátku
};
```

Konstruktor vytvoří prázdnou frontu. Standardně může fronta obsahovat až deset položek, tento počet však lze přepsat parametrem při explicitní inicializaci.

```
Queue line1;          // fronta pro deset položek
Queue line2(20);      // fronta pro dvacet položek
```

Při použití fronty můžete pomocí deklarace `typedef` definovat typ `Item`. (V kapitole 13 se naučíte, jak lze místo toho použít šablony tříd.)

Implementace

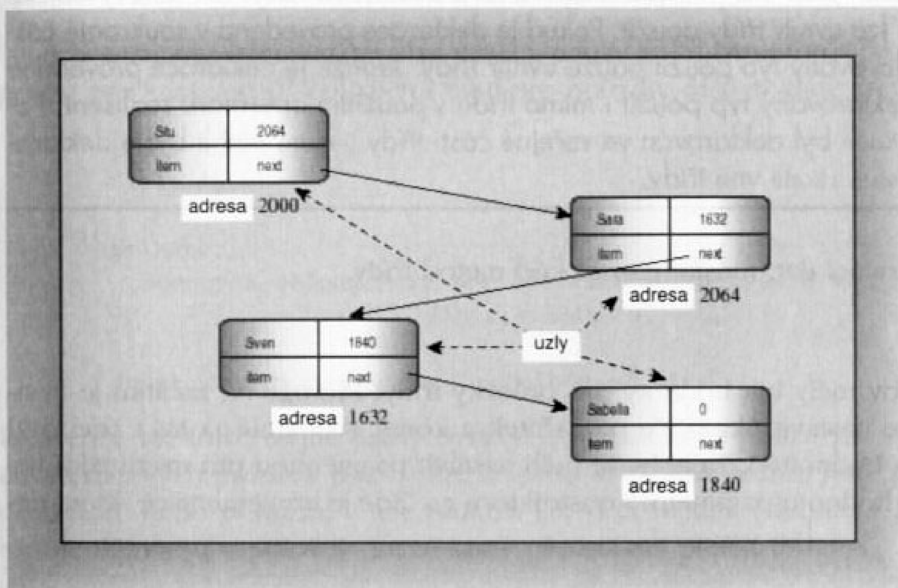
Nyní pojďme rozhraní implementovat. Nejdříve se musíte rozhodnout, jak budou data fronty reprezentována. Jednou možností je vytvořit dynamické pole s požadovaným počtem prvků pomocí operátoru `new`. Pole se však pro operace s frontou příliš nehodí. Například po odebrání položky ze začátku pole byste museli posunout všechny zbývající prvky o jednu pozici blíže k začátku fronty. Nebo budete muset udělat něco složitějšího, například pracovat s polem cyklicky. Požadavky fronty však dobře splňuje *spojový seznam*. Spojový seznam tvoří posloupnost *uzlů*. Každý uzel obsahuje informace uložené v seznamu a navíc ukazatel na další uzel v seznamu. V naší frontě bude každá datová část hodnotou typu `Item`. Uzel můžete vyjádřit strukturou:

```
struct Node
{
    Item item;             // data uložená v uzlu
    struct Node * next;    // ukazatel na další uzel
};
```

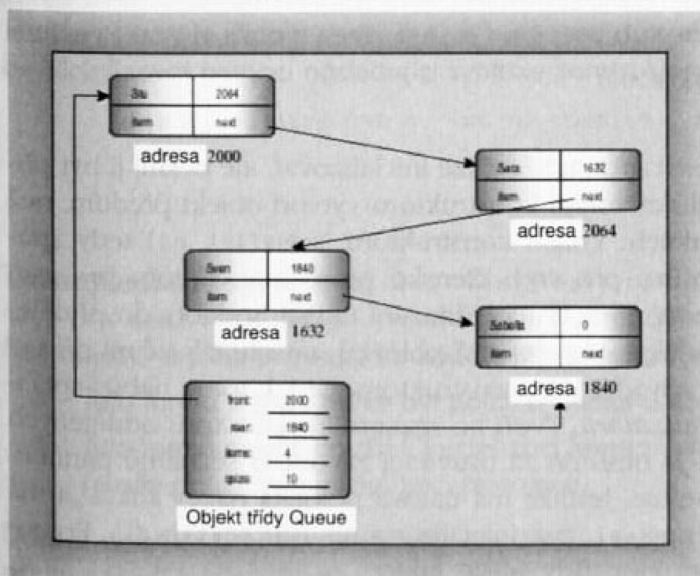
Spojový seznam ilustruje obrázek 11.8. Tato konkrétní forma se nazývá jednocestný seznam, protože každý uzel má pouze jeden ukazatel na další uzel. Jestliže máte adresu prvního uzlu, můžete najít ukazatele všech následujících uzlů v seznamu. Většinou je ukazatel v posledním uzlu seznamu nastaven na `NULL` (nebo `0`), čímž označuje, že další uzly už nejsou. Abyste mohli se spojovým seznamem pracovat, musíte znát adresu prvního uzlu. Začátek seznamu lze nastavit pomocí ukazatele třídy `Queue`. V podstatě je to jediná informace, kterou potřebujete, protože díky zřetězení uzlů najdete jakýkoli jiný uzel. Protože však ve frontě je nová položka přidána vždy na konec fronty, bude vhodné mít také ukazatel na poslední uzel (viz obrázek 11.9). Kromě toho můžete datové položky použít pro evidenci maximálního počtu položek ve frontě a pro evidenci aktuálního počtu. Soukromá část deklarace třídy může tedy vypadat takto:

```
class Queue
{
// definice rozsahu třídy
```

```
    // Node je vnořená struktura s platností uvnitř třídy
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};
private:
    Node * front;        // ukazatel na začátek fronty
    Node * rear;         // ukazatel na konec fronty
    int items;           // aktuální počet položek ve frontě
    const int qsize;     // maximální počet položek ve frontě
    ...
public:
//...
};
```



Obrázek 11.8. Spojový seznam



Obrázek 11.9. Objekt třídy Queue

V deklaraci se používá nový rys jazyka C++: schopnost vnořit strukturu nebo třídu do jiné třídy. Jestliže umístíte deklaraci struktury `Node` do třídy `Queue`, bude mít třídní rozsah. To znamená, že typ `Node` můžete použít pro deklaraci položek třídy i jako typ v metodách třídy, platnost je však omezena na danou třídu. Nemusíte se tedy obávat konfliktu s nějakou globální deklarací typu `Node` nebo s typem `Node` deklarovaným v jiné třídě. Ne všechny překladače podporují vnořené struktury a třídy. Pokud k nim patří i váš kompilátor, budete muset definovat strukturu `Node` globálně s platností pro celý soubor.

Vnořené struktury a třídy

Jsou-li struktura, třída nebo výčet deklarovány uvnitř deklarace třídy, říkáme, že jsou ve třídě vnořené. Jejich rozsah platnosti je uvnitř třídy. Taková deklarace nevytvoří datový objekt. Určuje však typ, který lze uvnitř třídy použít. Pokud je deklarace provedena v soukromé části třídy, potom lze deklarovaný typ použít pouze uvnitř třídy. Jestliže je deklarace provedena ve veřejné části, lze deklarovaný typ použít i mimo třídu s použitím operátoru rozlišení. Pokud by například typ `Node` byl deklarován ve veřejné části třídy `Queue`, mohli byste deklarovat proměnné typu `Queue::Node` vně třídy.

Jakmile určíte reprezentaci dat, musíte napsat kód metod třídy.

Metody třídy

V konstruktoru třídy by měly být hodnoty pro položky třídy. Protože na začátku je fronta prázdná, měli byste nastavit ukazatele na začátek a konec seznamu na `NULL` (nebo `0`) a proměnnou `item` na hodnotu `0`. Také byste měli nastavit proměnnou pro maximální velikost fronty `qsize` na hodnotu parametru konstruktoru `qs`. Zde je implementace, která nefunguje:

```
Queue::Queue(int qs)
{
    front = rear = NULL;
    items = 0;
    qsize = qs;  // nepřijatelné!
}
```

Problém je, že proměnná `qsize` je `konstanta`, takže ji lze inicializovat, ale nesmí jí být přiřazena hodnota. Z koncepčního hlediska volání konstruktoru vytvoří objekt předtím, než je proveden kód ve složených závorkách. Volání konstruktoru `Queue(int qs)` tedy způsobí, že program nejdříve přidělí místo pro čtyři členské proměnné. Potom program vstoupí mezi složené závorky a pomocí normálního přiřazení umístí hodnoty do přiděleného místa. Jestliže tedy chcete inicializovat konstantní položku, musíte tak učinit při vytvoření objektu, dříve než se začne provádět tělo konstruktoru. C++ k tomu nabízí speciální syntaxi. Nazývá se *seznam inicializátorů*. Tvoří ho seznam inicializátorů oddělených čárkou, kterým předchází dvojtečka. Je umístěn za uzavírací závorkou seznamu parametrů a před otevírací závorkou těla funkce. Jestliže má datová položka název `mdata` a má být inicializována na hodnotu proměnné `val`, pak inicializátor má tvar `mdata(val)`. Pomocí této notace je možné napsat konstruktor třídy `Queue` takto:

```
Queue::Queue(int qs) : qsize(qs)  // inicializace qsize na qs
{
  front = rear = NULL;
  items = 0;
}
```

Obecně řečeno, počáteční hodnotu lze získat z konstant i parametrů konstruktoru. Tato technika není omezena pouze na inicializaci konstant; konstruktor třídy `Queue` lze napsat také tímto způsobem:

```
Queue::Queue(int qs) : qsize(qs), front(NULL), rear(NULL), items(0)
{
}
```

Syntaxi seznamu inicializátorů lze použít pouze u konstruktorů. Jak jste viděli, musíte ji použít pro `konstantní` položky a také pro položky deklarované jako reference:

```
class Agency {...};
class Agent
{
private:
  Agency & belong;  // pro inicializaci je nutné použít seznam
                    //inicializátorů    ...
};
Agent::Agent(Agency & a) : belong(a) {...}
```

Reference lze totiž, stejně jako `konstanty`, inicializovat pouze při vytváření. U jednoduchých datových položek jako `front` a `items` je celkem jedno, jestli použijete seznam inicializátorů nebo přiřazení v těle funkce. Jak však uvidíte v kapitole 13, seznam inicializátorů je účinnější u položek, které samy představují objekt třídy.

Syntaxe seznamu inicializátorů

Jestliže `Classy` je třída a `mem1`, `mem2` a `mem3` jsou datové položky třídy, pak lze datové položky inicializovat pomocí následující syntaxe konstruktoru třídy:

```
Classy::Classy(int n, int m) :mem1(n), mem2(0), mem3(n*m + 2)
{
//...
}
```

Tímto způsobem je proměnná `mem1 inicializována na hodnotu parametru n, mem2 na hodnotu 0 a mem3 na hodnotu výrazu n*m+2`. Koncepčně tyto inicializace proběhnou při vytvoření objektu a před provedením kódu v těle konstruktoru. Zapamatujte si následující:

- Tato forma zápisu může být použita pouze u konstruktorů.
- Tuto formu musíte použít k inicializaci konstantních nestatických položek. – Tuto formu musíte použít pro inicializaci reference.

Datové položky jsou inicializovány v pořadí, v jakém se nacházejí v deklaraci třídy, nikoli podle pořadí inicializátorů.

Upozornění

Syntaxi seznamu inicializátorů nemůžete použít u jiných metod třídy než u konstruktorů.

Způsob inicializace použitý v seznamu inicializátorů lze použít i jinde. Pokud budete chtít, můžete následující kód

```
int games = 162;
double talk = 2.71828;
```

zaměnit za

```
int games(162);
double talk(2.71828);
```

Díky tomu vypadá inicializace vestavěných typů stejně jako inicializace objektů.

Kód funkcí `isempty()`, `isfull()`, a `queuecount()` je jednoduchý. Jestliže má položka `items` hodnotu 0, fronta je prázdná. Pokud se její hodnota rovná hodnotě položky `qsize`, potom je fronta plná. Vrácená hodnota položky `items` odpoví na otázku, kolik položek je ve frontě. Kód ukážeme později v hlavičkovém souboru.

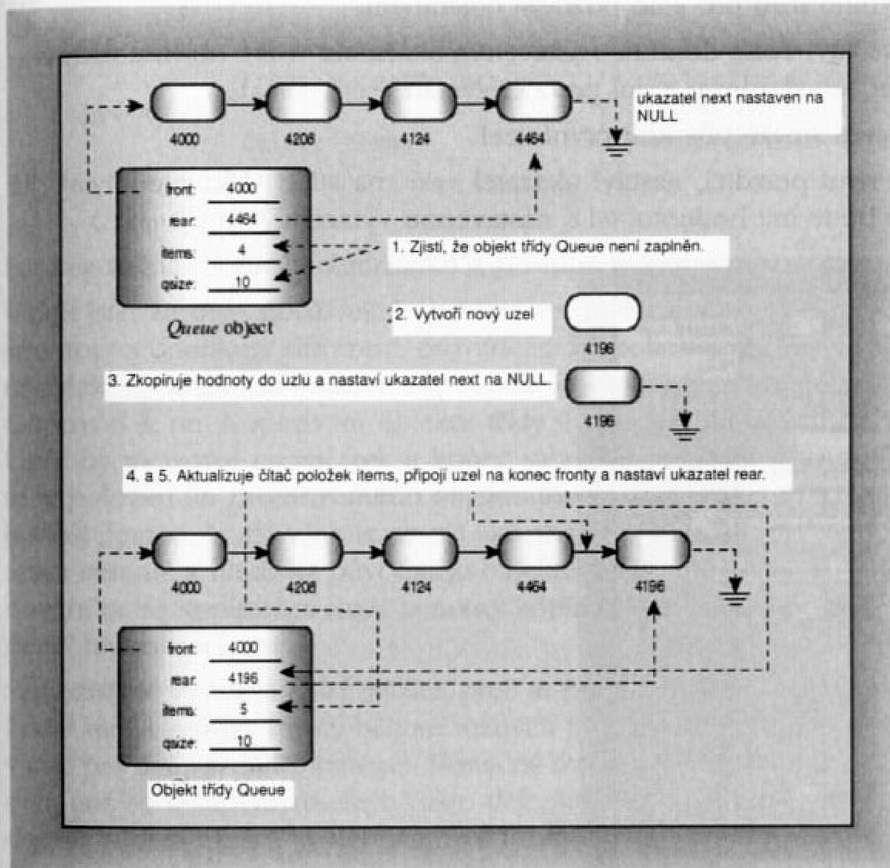
Přidání položky na konec fronty je složitější. Zde je jeden možný přístup:

```
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;    // vytvoří uzel
    if (add == NULL)
        return false;         // návrat při nedostatku paměti
    add->item = item;         // nastaví ukazatele uzlu
    add->next = NULL;
    items++;
    if (front == NULL)        // pokud je fronta prázdná,
        front = add;          // umístí položku na začátek
    else
        rear->next = add;     // jinak na konec fronty
    rear = add;               // nastaví ukazatel rear na nový uzel
    return true;
}
```

Zde je shrnutí fází, kterými metoda prochází (viz také obrázek 11.10):

1. Pokud je již fronta zaplněna, ukončí běh.
2. Vytvoří nový uzel. Pokud se jí to nepodaří, ukončí běh (například při nedostatku paměti).
3. Nastaví do uzlu odpovídající hodnoty. V tomto případě zkopíruje hodnotu položky `Items` do datové části uzlu a nastaví ukazatel next na `NULL`. Tím je uzel připraven na roli poslední položky fronty.
4. Zvýší hodnotu čítače položek (`items`) o jedničku.

5. Připojí uzel na konec fronty. Tento proces se skládá ze dvou částí. První je svázání uzlu s ostatními uzly seznamu. To se děje nastavením ukazatele `next` aktuálně posledního uzlu na nový poslední uzel. Druhou částí je nastavení ukazatele `rear` třídy `Queue` na nový uzel, takže fronta má k poslednímu uzlu přímý přístup. Jestliže je fronta prázdná, musíte také nastavit ukazatel front na nový uzel. (Pokud existuje pouze jeden uzel, je zároveň prvním i posledním uzlem.)



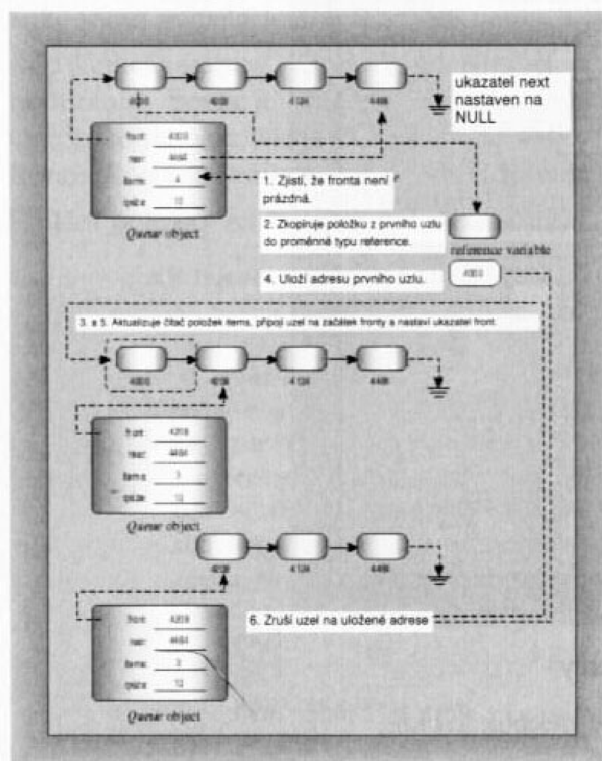
Obrázek 11.10 Přidání položky na konec fronty

Odebrání položky ze začátku fronty má také několik kroků:

```
bool Queue::dequeue(Item & item)
{
    if (front == NULL)
        return false;
    item = front->item;     // nastaví item na první položku fronty
    items--;
    Node * temp = front;    // uloží adresu první položky
    front = front->next;    // nastaví ukazatel front na další položku
    delete temp;             // zruší původní první položku
    if (items == 0)
        rear = NULL;
    return true;
}
```

Zde je shrnutí fází, kterými metoda prochází (viz také obrázek 11.11):

1. Jestliže je fronta již prázdná, ukončení běhu.
2. První položku fronty poskytne volající funkci. Toho dosáhne zkopírováním datové části aktuálního uzlu `front` do proměnné typu reference, která je předána metodě.
3. Sníží hodnotu čítače položek (`items`) o jedničku.
4. Uloží adresu prvního uzlu pro jeho pozdější odstranění.
5. Odstraní uzel z fronty. Toho dosáhne nastavením ukazatele `front` objektu třídy `Queue` na další uzel, jehož adresu získá pomocí výrazu `front->next`.
6. Kvůli šetření paměti smaže původní první uzel.
7. Pokud je fronta nyní prázdná, nastaví ukazatel `rear` na `NULL`. (Ukazatel `front` již v tomto případě bude mít hodnotu `NULL` nastavenou výrazem `front->next`.)



Obrázek 11.11. Odebrání položky ze začátku fronty

Čtvrtý krok je nutný, protože v pátém kroku dojde ke smazání paměti, kde se nacházel předchozí první prvek.

Ostatní metody třídy

Potřebujete nějaké další metody? V konstruktoru třídy se nepoužívá operátor `new`, takže na první pohled se může zdát, že se nemusíte znepokojovat zvláštními požadavky tříd, které operátor `new` používají. První pohled samozřejmě klame, protože při přidávání prvků do fronty se nové uzly vytvářejí pomocí operátoru `new`. Je pravda, že metoda `dequeue()` provádí úklid rušením uzlů, ale nemáme žádnou záruku, že fronta bude prázdná, když

skončí její platnost. Třída tedy vyžaduje explicitní destruktor, který zruší všechny zbývající uzly. Zde je jeho implementace:

```
Queue::~Queue()
{
     Node * temp;
     while (front != NULL)          // dokud fronta není prázdná
     {
          temp = front;             // uloží adresu první položky
          front = front->next;      // nastaví ukazatel na následující po-
  ložku
          delete temp;              // zruší předchozí první položku
     }
}
```

Funkce začne na začátku seznamu a postupně zruší všechny uzly.

Viděli jste, že třídy používající operátor `new` většinou vyžadují explicitní kopírovací konstruktory a operátory přiřazení, provádějící hlubokou kopii. Jedná se o stejný případ? První otázkou, na kterou musíme odpovědět je, zda kopírování po položkách je tím pravým. Odpověď je ne. Kopírování objektu třídy `Queue` po položkách by vytvořilo nový objekt, který by ukazoval na začátek a konec stejného seznamu jako původní objekt. Přidáte-li tedy položku do zkopírovaného objektu `Queue`, změníte tento sdílený spojový seznam. To je dost špatné. Horší však je, že se aktualizuje pouze zkopírovaný ukazatel rear, čímž je tento seznam z hlediska původního objektu podstatně narušen. Je tedy jasné, že pro klonování nebo kopírování front je nutný kopírovací konstruktor a operátor přiřazení provádějící hlubokou kopii.

Samozřejmě vyvstává další otázka, proč je potřeba frontu kopírovat? Možná budete potřebovat uložit snímky fronty během různých fází simulace. Nebo budete chtít vytvořit stejný vstup pro dvě rozdílné strategie. Skutečně může být užitečné mít operace, které frontu rozdělí, což se v supermarketech často děje, když je například otevřena další pokladna. Podobně budete možná potřebovat dvě fronty spojit nebo nějakou frontu zkrátit.

My ale nic z toho v této simulaci potřebovat nebudeme. Nemůžete tyto věci jednoduše ignorovat a použít metody, které již máte? Samozřejmě že můžete. Někdy v budoucnu však možná budete frontu potřebovat znovu, a tentokrát s možností kopírování. A třeba zapomenete, že jste pro kopírování nevytvořili patřičný kód. Programy půjdou zkompilovat a poběží, přinesou však záhadné výsledky a havarují. Takže je asi lepší vytvořit kopírovací konstruktor a operátor přiřazení, i když je nyní nebudete potřebovat.

Naštěstí existuje rafinovaný způsob, jak se vyhnout práci navíc, přitom se ochránit před háváriemi programu v budoucnosti. Myšlenka spočívá v definování požadovaných metod v soukromé části jako metod fiktivních:

```
class Queue
{
private:
   Queue(const Queue & q) : qsize(0) { }
   Queue & operator=(const Queue & q) { return *this;}
//...
};
```

Taková definice má dva účinky. Za prvé, dojde k přepsání implicitních metod, které by jinak byly automaticky vygenerovány, a za druhé, protože jsou tyto metody soukromé, nemohou být použity mimo třídu. Jestliže jsou tedy proměnné `nip` a tuck objekty třídy `Queue`, pak kompilátor neumožní zkompilovat následující kód:

```
Queue snick(nip);      // nepovoleno
tuck = nip;            // nepovoleno
```

V budoucnu tedy nebudete muset čelit záhadnému chování funkcí, ale obdržíte místo toho od kompilátoru srozumitelnější chybovou zprávu, oznamující, že tyto funkce nejsou k dispozici. Tento trik je užitečný také v případě, když definujete třídu, jejíž objekty by neměly být kopírovány.

Existují ještě nějaké další efekty, které stojí za povšimnutí? Ano. Vzpomeňte si, že kopírovací konstruktor je vyvolán, jestliže je objekt předáván (nebo vracen) hodnotou. Tento problém však nevznikne, pokud se budete řídit osvědčenou praxí a předávat objekty jako reference. Kopírovací konstruktor je také používán při vytváření jiných dočasných objektů. V definici třídy `Queue` ale chybí operace, které by vedly k dočasným objektům, například přetížený operátor sčítání.

Třída Customer

Dále musíte navrhnout třídu reprezentující zákazníka. Obecně má zákazník bankovního automatu hodně vlastností, jako je jméno, číslo účtu a stav účtu. Pro simulaci však potřebujete pouze vlastnosti, které budou zaznamenávat připojení zákazníka k frontě a čas potřebný k vyřízení transakce. Když simulace vytvoří nového zákazníka, program vytvoří nový objekt zákazníka, uloží do něj čas příchodu a náhodně vygenerovanou hodnotu času transakce. Když zákazník dojde na začátek fronty, program zaznamená čas a odečte od něho čas, kdy si zákazník do fronty stoupnul. Tím získá délku doby čekání. Zde je jedna z možných definicí a implementací třídy `Customer`:

```
class Customer
{
private:
    long arrive;          // čas příchodu
    int processtime;      // doba odbavení zákazníka
public:
    Customer() { arrive = processtime = 0; }
    void set(long when);
    long when() const { return arrive; }
    int ptime() const { return processtime; }
};
void Customer::set(long when)
{
    processtime = rand() % 3 + 1;
    arrive = when;
}
```

Implicitní konstruktor vytvoří prázdného zákazníka. Členská funkce `set()` nastaví do parametru čas příchodu a pro dobu odbavení vybere náhodně hodnotu od 1 do 3.

Ve výpisu 11.11 je společná deklarace tříd `Queue` a `Customer` a výpis 11.12 obsahuje definici metod.

Výpis 11.11. queue.h

```
// queue.h – rozhraní třídy Queue
#ifndef _QUEUE_H_
#define _QUEUE_H_
// Tato fronta bude obsahovat položky třídy Customer
class Customer
{
private:
    long arrive;        // čas příchodu zákazníka
    int processtime;    // doba odbavení zákazníka
public:
    Customer() { arrive = processtime = 0; }
    void set(long when);
    long when() const { return arrive; }
    int ptime() const { return processtime; }
};
typedef Customer Item;
class Queue
{
// definice s platností uvnitř třídy
    // Node definice vnořené struktury existující pouze uvnitř třídy
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};
private:
    Node * front;       // ukazatel na začátek fronty
    Node * rear;        // ukazatel na konec fronty
    int items;          // aktuální počet položek fronty
    const int qsize;    // maximální počet položek fronty
    // preemptivní definice nutné pro zamezení veřejného kopírování
  Queue(const Queue & q) : qsize(0) { }
    Queue & operator=(const Queue & q) { return *this;}
public:
    Queue(int qs = Q_SIZE); // vytvoří frontu s limitem qs
    ~Queue();
    bool isempty() const;
    bool isfull() const;
    int queuecount() const;
    bool enqueue(const Item &item); // přidá položku na konec
    bool dequeue(Item &item);       // odebere položku ze začátku
};
#endif
```

Výpis 11.12. queue.cpp

```
// queue.cpp – metody třídy Customer a Queue
#include "queue.h"
#include <cstdlib>          // (nebo stdlib.h) pro rand()
// metody třídy Queue
Queue::Queue(int qs) : qsize(qs)
{
    front = rear = NULL;
    items = 0;
}
Queue::~Queue()
{
    Node * temp;
    while (front != NULL)           // dokud fronta není prázdná
    {
        temp = front;               // uloží adresu první položky
        front = front->next;        // nastaví ukazatel na další položku
        delete temp;                // zruší původní první položku
    }
}
bool Queue::isempty() const
{
    return items == 0;
}
bool Queue::isfull() const
{
    return items == qsize;
}
int Queue::queuecount() const
{
    return items;
}
// přidání prvku do fronty
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;          // vytvoří uzel
    if (add == NULL)
        return false;               // návrat při chybě
    add->item = item;               // nastaví ukazatele uzlu
    add->next = NULL;
    items++;
    if (front == NULL)              // pokud je fronta prázdná
        front = add;                // umístí položku na začátek
    else
        rear->next = add;           // jinak na konec
    rear = add;                     // nastaví ukazatel rear na nový uzel
    return true;
}
```

```
// Umístí první položku do proměnné item a odebere z fronty
bool Queue::dequeue(Item & item)
{
    if (front == NULL)
        return false;
    item = front->item;          // nastaví ukazatel na první položku
fronty
    items-;
    Node * temp = front;         // uloží adresu první položky
    front = front->next;         // nastaví ukazatel front na další po-
ložku
    delete temp;                 // zruší předchozí první položku
    if (items == 0)
        rear = NULL;
    return true;
}
// metoda třídy Customer
// když zákazník přijde, nastaví se
// čas příchodu dp proměnné when a do proměnné, představující
// dobu odbavení, se nastaví náhodná hodnota v rozsahu 1 - 3.
void Customer::set(long when)
{
    processtime = rand() % 3 + 1;
    arrive = when;
}
```

Kompatibilita:

Možná vlastníte kompilátor, který neimplementuje typ `bool`. V tom případě můžete místo typu `bool` použít typ `int` s hodnotou `0` pro false a s hodnotou `1` pro `true`. Možná budete také muset použít hlavičkový soubor `stdlib.h` místo novějšího `cstdlib`.

Simulace

Nyní máme potřebné nástroje pro simulaci bankovního automatu. Program umožní uživateli zadat tři hodnoty: maximální velikost fronty, počet simulovaných hodin a průměrný počet zákazníků za hodinu. Program použije smyčku, ve které každý cyklus bude představovat jednu minutu. Během každého cyklu program provede následující akce:

1. Určí, zda přišel nový zákazník. Pokud ano, přidá zákazníka na konec fronty, bude-li pro něho místo, jinak ho odmítne.
2. Pokud nebude žádný zákazník odbavován, vezme první osobu z fronty. Určí, jak dlouho tato osoba čeká a nastaví čítač `wait_time` na dobu odbavení, kterou bude nový zákazník potřebovat.
3. Jestliže je zákazník odbavován, odečte jednu minutu z čítače `wait_time`.
4. Sleduje různé hodnoty, jako například počet obsloužených zákazníků, počet odmítnutých zákazníků, celkový čas strávený čekáním ve frontě a celkovou délku fronty.

Po skončení simulačního cyklu program vypíše různé statistické informace.

Zajímavý je způsob, kterým program pozná příchod nového zákazníka. Předpokládejme, že za hodinu v průměru přijde deset zákazníků. To znamená v průměru jeden zákazník každých šest minut. Program vypočítá a uloží tuto hodnotu do proměnné `min_per_cust`. Je ovšem nereálné, že by se přesně každých šest minut objevil jeden zákazník. To, co doopravdy chceme (alespoň po většinu času), je nahodilý proces, ve kterém na jednoho zákazníka připadá v průměru šest minut. Pomocí následující funkce program v cyklu zjišťuje, zda přišel zákazník:

```
bool newcustomer(double x)
{
    return (rand() * x / RAND_MAX < 1);
}
```

Zde je způsob práce této funkce. Hodnota `RAND_MAX` je definována v hlavičkovém souboru cstdlib (dříve `stdlib.h`) a představuje největší hodnotu, kterou může funkce `rand()` vrátit. Předpokládejme, že `x`, průměrná doba mezi zákazníky, je 6. Potom hodnota výrazu `rand() * x / RAND_MAX` bude někde mezi 0 a 6. Konkrétně, v průměru v jednom případě ze šesti bude tato hodnota menší než 1. Je však možné, že výsledkem této funkce bude jednou rozdíl jedné minuty mezi dvěma zákazníky a jindy minut dvacet. Takové chování vede k těžkopádnosti, která často odlišuje skutečné procesy od pravidelnosti, při které každých 6 minut přichází přesně jeden zákazník. Tato konkrétní metoda selže, jestliže průměrná doba mezi zákazníky klesne pod jednu minutu, simulace však není určena pro takový scénář. Pokud byste se však potřebovali zabývat takovým případem, použili byste jemnější časové rozlišení, ve kterém by každý cyklus představoval třeba 10 sekund.

Kompatibilita:

Některé kompilátory nedefinují konstantu `RAND_MAX`. Pokud se v takové situaci ocitnete, můžete definovat hodnotu `RAND_MAX` sami pomocí direktivy `#define` nebo jako `const int`. Jestliže v dokumentaci nenajdete správnou hodnotu, zkuste největší možnou hodnotu typu `int`, která je určena konstantou `INT_MAX` v souboru `climits` nebo `limits.h`.

Výpis 11.13 obsahuje detaily simulace. Necháte-li simulaci běžet po dlouhý časový úsek, získáte dlouhodobé průměry, pokud ji necháte běžet krátce, získáte krátkodobé odchylky.

Výpis 11.13. bank.cpp

```
// bank.cpp -- použije rozhraní třídy Queue
#include <iostream>
using namespace std;
#include <cstdlib> // pro rand() a srand()
#include <ctime>   // pro time()
#include "queue.h"
const int MIN_PER_HR = 60;
bool newcustomer(double x);     // přišel nový zákazník?
int main()
```

```
{
// nastavení
    srand(time(0));                 // náhodná inicializace funkce rand()
    cout << "Studie Case: Bankomat Bank of Heather\n";
    cout << "Zadejte maximalni delku fronty: ";
    int qs;
    cin >> qs;
    Queue line(qs);                 // ve frontě může být maximálně qs lidí
    cout << "Zadejte pocet simulovanych hodin: ";
    int hours;                      // počet simulovaných hodin
    cin >> hours;
    // jeden cyklus simulace trvá jednu minutu
    long cyclelimit = MIN_PER_HR * hours; // počet cyklů
    cout << "Zadejte prumerny pocet zakazniku za hodinu: ";
    double perhour;                 // průměrný počet zákazníků za hodinu
    cin >> perhour;
    double min_per_cust;            // průměrná doba mezi příchodem zákazníků
    min_per_cust = MIN_PER_HR; // za hodinu;
    Item temp;                      // data nového zákazníka
    long turnaways = 0;             // počet odmítnutých zákazníků
    long customers = 0;             // počet zákazníků ve frontě
    long served = 0;                // počet obsloužených zákazníků
    long sum_line = 0;              // celková délka fronty
    int wait_time = 0;              // doba čekání na bankomat
    long line_wait = 0;             // celková doba čekání ve frontě
// běh simulace
    for (int cycle = 0; cycle < cyclelimit; cycle++)
    {
        if (newcustomer(min_per_cust)) // nový zákazník
        {
        if (line.isfull())
            turnaways++;
        else
        {
            customers++;
            temp.set(cycle);      // cycle = čas příchodu
            line.enqueue(temp); // přidá nového zákazníka do fronty
        }
      }
      if (wait_time <= 0 && !line.isempty())
      {
          line.dequeue (temp);        // vyřizuje dalšího zákazníka
          wait_time = temp.ptime(); // po wait_time minut
          line_wait += cycle - temp.when();
          served++;
      }
      if (wait_time > 0)
          wait_time—;
      sum_line += line.queuecount();
    }
// výpis výsledků
```

```
    if (customers > 0)
    {
        cout << "prijato zakazniku: " << customers << '\n';
        cout << "obslouzeno zakazniku: " << served << '\n';
        cout << "odmitnuto zakazniku: " << turnaways << '\n';
        cout << "prumerna delka fronty: ";
        cout.precision(2);
        cout.setf(ios_base::fixed, ios_base::floatfield);
        cout.setf(ios_base::showpoint);
        cout << (double) sum_line / cyclelimit << '\n';
        cout << "prumerna cekaci doba: "
             << (double) line_wait / served << " minut\n";
    }
    else
        cout << "Nejsou zadni zakaznici!\n";
    return 0;
}
// x = průměrná doba v minutách mezi dvěma zákazníky
// návratová hodnota je true, pokud se do této doby objeví nový zákazník
bool newcustomer(double x)
{
    return (rand() * x / RAND_MAX < 1);
}
```

Kompatibilita:

Možná vlastníte kompilátor, který neimplementuje typ `bool`. V takovém případě můžete místo typu `bool` použít typ `int` s hodnotou `0` pro `false` a s hodnotou `1` pro `true`. Možná budete muset použít hlavičkové soubory `stdlib.h` a `time.h` místo novějších `cstdlib` a `ctime`, a možná také sami definovat konstantu `RAND_MAX`.

Zde je několik spuštění s delším časovým intervalem:

```
Studie Case: Bankomat Bank of Heather
Zadejte maximalni delku fronty: 10
Zadejte pocet simulovanych hodin: 100
Zadejte prumerny pocet zakazniku za hodinu: 15
prijato zakazniku: 1485
obslouzeno zakazniku: 1485
odmitnuto zakazniku: 0
prumerna delka fronty: 0.15
prumerna cekaci doba: 0.63 minut
Studie Case: Bankomat Bank of Heather
Zadejte maximalni delku fronty: 10
Zadejte počet simulovanych hodin: 100
Zadejte prumerny počet zakazniku za hodinu: 30
prijato zakazniku: 2896
obslouzeno zakazniku: 2888
odmitnuto zakazniku: 101
prumerna delka fronty: 4.64
prumerna cekaci doba: 9.63 minut
```

```
Studie Case: Bankomat Bank of Heather
 Zadejte maximalni delku fronty: 20
 Zadejte počet simulovanych hodin: 100
 Zadejte prumerny počet zakazniku za hodinu: 30
 prijato zakazniku: 2943
 obslouzeno zakazniku: 2943
 odmitnuto zakazniku: 93
 prumerna delka fronty: 13.06
 prumerna cekaci doba: 26.63 minut
```

Všimněte si, že přechod od patnácti zákazníků za hodinu ke třiceti zákazníkům za hodinu nezdvojnásobí průměrnou čekací dobu, ale zvýší ji patnáctkrát.

Delší fronta situaci jen zhorší. Simulace však nepočítá se skutečností, že mnoho zákazníků otrávených dlouhým čekáním z fronty prostě odejde.

Zde je několik dalších spuštění. Tyto ilustrují krátkodobé odchylky, přestože průměrný počet zákazníků za jednu hodinu je zachován.

```
Studie Case: Bankomat Bank of Heather
 Zadejte maximalni delku fronty: 10
 Zadejte počet simulovanych hodin: 4
 Zadejte prumerny počet zakazniku za hodinu: 30
 prijato zakazniku: 114
 obslouzeno zakazniku: 110
 odmitnuto zakazniku: 0
 prumerna delka fronty: 2.15
 prumerna cekaci doba: 4.52 minut
Studie Case: Bankomat Bank of Heather
 Zadejte maximalni delku fronty: 10
 Zadejte počet simulovanych hodin: 4
 Zadejte prumerny počet zakazniku za hodinu: 30
 prijato zakazniku: 121
 obslouzeno zakazniku: 116
 odmitnuto zakazniku: 5
 prumerna delka fronty: 5.28
 prumerna cekaci doba: 10.72 minut
Studie Case: Bankomat Bank of Heather
 Zadejte maximalni delku fronty: 10
 Zadejte počet simulovanych hodin: 4
 Zadejte prumerny počet zakazniku za hodinu: 30
 prijato zakazniku: 112
 obslouzeno zakazniku: 109
 odmitnuto zakazniku: 0
 prumerna delka fronty: 2.41
 prumerna cekaci doba: 5.16 minut
```

Shrnutí

Tato kapitola pokrývá mnoho důležitých aspektů definování a používání tříd. Některé tyto aspekty představují záludné, dokonce obtížné, pojmy. Jestliže vám některý z nich připadá nesrozumitelný nebo neobyčejně složitý, nic si z toho nedělejte – takto působí na většinu začátečníků v jazyku C++. Často si pojmy jako kopírovací konstruktor skutečně uvědomíte, jakmile se v důsledku jejich ignorování dostanete do problémů. Dokud tedy neobohatíte svoje chápání zkušenostmi, může vám některá látka této kapitoly připadat nejasná. Zatím si tuto kapitolu shrneme.

V konstruktoru můžete pomocí operátoru `new` přidělit paměť pro data a potom adresu této paměti přiřadit položce třídy. Díky tomu může třída například pracovat s řetězci proměnné délky, aniž by bylo nutné určit velikost pole předem. Používání operátoru `new` v konstruktorech s sebou také přináší potenciální problémy, jakmile platnost objektu skončí. Jestliže jsou v objektu ukazatele na paměť přidělenou operátorem `new`, uvolnění paměti používané objektem neznamená automaticky uvolnění paměti, na kterou ukazují ukazatele objektu. Jestliže tedy v konstruktoru přidělíte paměť pomocí operátoru `new`, měli byste v destruktoru třídy tuto paměť uvolnit pomocí operátoru `delete`. Tímto způsobem se při zániku objektu automaticky zruší paměť, na kterou ukazuje ukazatel.

U objektů s ukazateli na paměť přidělenou operátorem `new` vznikají také problémy při inicializaci objektu pomocí jiného objektu nebo při přiřazení objektu jinému objektu. C++ implicitně provádí kopírování a přiřazení po položkách, což znamená, že položky inicializovaného nebo přiřazeného objektu jsou přesnými kopiemi položek původního objektu. Jestliže ukazatel původního objektu ukazuje na nějaký blok dat, ukazuje ukazatel zkopírovaného objektu na tentýž blok. Když program nakonec oba tyto objekty zruší, bude se destruktor třídy snažit zrušit stejný blok paměti dvakrát, což vede k chybě. Řešení spočívá v definování speciálního kopírovacího konstruktoru, který inicializaci předefinuje a přetíží operátor přiřazení. V každém případě by nová definice měla vytvořit duplikáty všech dat, na které ukazují ukazatele a nastavit ukazatele nového objektu na tyto kopie. Tímto způsobem budou oba objekty odkazovat na totožná data, která však budou uložena samostatně a nebudou se překrývat. Stejná úvaha se týká definice operátoru přiřazení. V každém případě je cílem vytvořit hlubokou kopii, to znamená zkopírovat skutečná data a ne pouze ukazatele na ně.

Jazyk C++ umožňuje umístit do třídy definice struktury, třídy nebo výčtu. Takovéto vnořené typy mají třídní rozsah, což znamená, že jsou lokálními proměnnými této třídy a nejsou v rozporu s jinými strukturami, třídami nebo výčty stejného názvu definovanými někde jinde.

V C++ také existuje speciální syntaxe pro konstruktory tříd sloužící k inicializaci datových položek. Tato syntaxe se skládá z dvojtečky následované seznamem inicializátorů oddělených čárkou. Seznam je umístěn za uzavírací závorkou se seznamem parametrů konstruktoru a před otevírací složenou závorkou těla konstruktoru. Každý inicializátor obsahuje název inicializované položky, za kterou v kulatých závorkách následuje inicializační hodnota. Z koncepčního hlediska tyto inicializace proběhnou při vytvoření objektu a před provedením příkazů v těle konstruktoru. Syntaxe vypadá takto:

```
queue(int qs) : qsize(qs), items(0), front(NULL), rear(NULL)   { }
```

Tato forma je závazná, jestliže je datová položka nestatickou konstantou nebo referencí. Jak jste si asi všimli, musíte detailům tříd věnovat mnohem více péče a pozornosti než strukturám jazyka C. Na oplátku pro vás udělají mnohem víc.

Opakovací otázky

1. Předpokládejme, že třída `String` má tyto soukromé položky:

```
class String {
private:
    char * str;       // ukazuje na řetězec vytvořený operátorem new
    int len;          // obsahuje délku řetězce
//...
};
```

a) Co je špatně na tomto implicitním konstruktoru?

```
String::String() {}
```

b) Co je špatně na tomto konstruktoru?

```
String::String(const char * s)
{
    str = s;
    len = strlen(s);
}
```

c) Co je špatně na tomto konstruktoru?

```
String::String(const char * s)
{
    strcpy(str, s);
    len = strlen(s);
}
```

2. Vyjmenujte tři problémy, které mohou vyvstat, jestliže definujete třídu, ve které je ukazatel inicializován pomocí operátoru `new`, a navrhněte jejich nápravu.
3. Které metody třídy generuje kompilátor automaticky, pokud je nedefinujete explicitně? Popište, jak se tyto implicitně generované funkce chovají.
4. Najděte a opravte chyby v následující deklaraci třídy:

```
class nifty
{
// data
    char personality[];
    int talents;
// metody
    nifty();
    nifty(char * s);
    ostream & operator<<(ostream & os, nifty & n);
}
nifty:nifty()
```

```
{
    personality = NULL;
    talents = 0;
}
nifty:nifty(char * s)
{
    personality = new char [strlen(s)];
    personality = s;
    talents = 0;
}
ostream & nifty:operator<<(ostream & os, nifty & n)
{
    os << n;
}
```

5. Uvažujte následující deklaraci třídy:

```
class Golfer
{
private:
    char * fullname;        // ukazatel na řetězec obsahující jméno
                            // hráče golfu
    int games;              // počet odehraných her
    int * scores;           // ukazatel na první prvek pole
                            // dosažených výsledků
public:
    Golfer();
    Golfer(const char * name, int g= 0);
     // vytvoří prázdné pole g prvků, jestliže g > 0
    Golfer(const Golfer & g);
    ~Golfer();
};
```

a) Které metody třídy budou vyvolány následujícími příkazy?

```
Golfer nancy;                               // #1
Golfer lulu("Little Lulu");                 // #2
Golfer roy("Roy Hobbs", 12);                // #3
Golfer * par = new Golfer;                  // #4
Golfer next = lulu;                         // #5
Golfer hazzard = "Weed Thwacker";           // #6
*par = nancy;                               // #7
nancy = "Nancy Putter";                     // #8
```

b) Je jasné, že má-li být třída užitečná, potřebuje několik dalších metod, ale které metody jsou potřeba, aby se zamezilo poškození dat?

Programovací cvičení

1. Uvažujte následující deklaraci třídy:

```
class Cow {
    char name[20];
    char * hobby;
    double weight;
public:
    Cow();
    Cow(const char * nm, const char * ho, double wt);
    Cow(const Cow c&);
    ~Cow();
    Cow operator=(const Cow & c);
    void ShowCow(); // zobrazí jméno krávy, její libůstky a váhu
};
```

Vytvořte implementaci této třídy a napište krátký program využívající všechny tyto členské funkce.

2. Rozšiřte deklaraci třídy `String` (to znamená, změňte strng2.h na string3.h) takto:

   a) Přetěžte operátor sčítání tak, aby bylo možno spojit dva řetězce do jednoho.

   b) Vytvořte členskou funkci `stringlow()` převádějící všechny alfabetické znaky v řetězci na malá písmena.

   c) Vytvořte členskou funkci `stringup()` převádějící všechny alfabetické znaky v řetězci na velká písmena.

   d) Vytvořte členskou funkci s parametrem typu `char` a vracející počet výskytů tohoto znaku v řetězci.

Ověřte svou práci pomocí následujícího programu:

```
// pe11_1.cpp
#include <iostream>
using namespace std;
#include "strng3.h"
int main()
{
    String s1(" and I am a C++ student.");
    String s2 = "Please enter your name: ";
    String s3;
    cout << s2;                  // přetížený operátor <<
    cin >> s3;                   // přetížený operátor >>
    s2 = "My name is " + s3;    // přetížené operátory =, +
    cout << s2 << ".\n";
    s2 = s2 + s1;
    s2.stringup();               // převod na velká písmena
    cout << "The string\n" << s2 << "\ncontains " << s2.has('A')
      << " 'A' characters in it.\n";
    s1 = "red";        // String(const char *),
                       //potom String & operator=(const String&)
    String rgb[3] = { String(s1), String("green"), String("blue")};
```

```
        cout << "Enter the name of a primary color for mixing light: ";
        String ans;
        bool success = false;
        while (cin >> ans)
        {
            ans.stringlow();          // převod na malá písmena
            for (int i = 0; i < 3; i++)
            {
                if (ans == rgb[i])  // přetížený operátor ==
                {
                    cout << "That's right!\n";
                    success = true;
                    break;
                }
            }
            if (success)
                break;
            else
                cout << "Try again!\n";
        }
        cout << "Bye\n";
        return 0;
    }
```

Výstup by měl vypadat asi takto:

```
Please enter your name: Fretta Farbo
My name is Fretta Farbo.
The string
MY NAME IS FRETTA FARBO AND I AM A C++ STUDENT.
contains 6 'A' characters in it.
Enter the name of a primary color for mixing light: yellow
Try again!
BLUE
That's right!
Bye
```

3. Přepište třídu `Stock`, popsanou ve výpisech 9.7 a 9.8 tak, aby pro ukládání názvů firem používala místo pevných polí dynamicky přidělenou paměť. Nahraďte také členskou funkci `show()` definicí přetížené funkce `operator()<<`. Novou definici vyzkoušejte v programu z výpisu 9.9.
4. Uvažujte následující variantu třídy `Stack`, definované ve výpisu 9.10:

```
// stack.h - deklarace třídy pro zásobník ADT
typedef unsigned long Item;
class Stack
{
private:
enum {MAX = 10};        // specifická konstanta třídy
Item * pitems;          // obsahuje položky zásobníku
int size;               // počet prvků v zásobníku
int top;                // index položky na vrcholu zásobníku
```

```
public:
Stack(int n = 10); // vytvoří zásobník s n prvky
Stack(const Stack & st);
~Stack();
bool isempty() const;
bool isfull() const;
// push() vrátí False, jestliže je zásobník již plný, jinak True
bool push(const Item & item);      // přidá položky do zásobníku
// pop() vrátí False, jestliže je zásobník již prázdný, jinak True
bool pop(Item & item);             // načte prvek z vrcholu zásobníku
                                   //do proměnné item
Stack operator=(const Stack & st);
};
```

Jak napovídají soukromé datové položky, používá tato třída pro uložení položek dynamicky vytvořené pole. Přepište metody podle této nové reprezentace a napište program demonstrující všechny tyto metody včetně kopírovacího konstruktoru a operátoru přiřazení.

5. Bank of Heather provedla výzkum, ukazující, že zákazníci u automatu nechtějí ve frontě čekat déle než jednu minutu. S použitím simulace z výpisu 11.13 zjistěte hodnotu počtu zákazníků za hodinu, která vede k průměrné čekací době v délce jedné minuty. (Pokus provádějte alespoň 100 hodin.)
6. Bank of Heather by ráda věděla, co se stane po přidání druhého bankovního automatu. Upravte simulaci pro dvě fronty. Předpokládejte, že je-li v první frontě méně lidí, zákazník se postaví do první fronty, v opačném případě se připojí ke druhé frontě. Opět zjistěte hodnotu počtu zákazníků za jednu hodinu, vedoucí k průměrné čekací době v délce jedné minuty. (Poznámka: Nejedná se o lineární problém, při kterém by dvojnásobný počet automatů odbavil za hodinu dvojnásobný počet zákazníků při maximální čekací době v délce jedné minuty.)

Dědičnost tříd

KAPITOLA 12

Témata kapitoly:

- Dědičnost je jako vztah
- Jak veřejně odvodit jednu třídu od druhé
- Chráněný přístup
- Seznamy inicializátorů v konstruktoru
- Přetypování na předka a na potomka
- Virtuální členské funkce
- Časná (statická) a pozdní (dynamická) vazba
- Čistě virtuální funkce
- Kdy a jak použít veřejné dědění

Jedním z hlavních cílů objektově orientovaného programování je vytvoření znovupoužitelného kódu. Když vyvíjíte nový projekt, zvláště projekt velký, je příjemné, můžete-li použít již ověřený kód a nevymýšlet jej znovu. Použití starého kódu šetří čas a vzhledem k tomu, že se jedná o kód již použitý a vyzkoušený, nebudou do programu zataženy chyby. A čím méně se musíte znepokojovat podrobnostmi, tím více se můžete soustředit na celkovou strategii programu.

Tradiční knihovny jazyka C nabízí znovupoužitelný kód prostřednictvím předdefinovaných a předkompilovaných funkcí, jako jsou například funkce `strlen()` a `srand()`, které můžete použít ve svých programech. Mnoho prodejců dodává specializované knihovny jazyka C funkcemi, které nejsou ve standardní knihovně jazyka C. Můžete si například koupit knihovny s funkcemi pro správu databází nebo pro ovládání obrazovky. Knihovny funkcí však mají jisté omezení. Pokud vám prodejce nedodá zdrojový kód funkcí knihovny (což většinou nedělá), nemůžete funkce rozšířit nebo upravit pro svoje konkrétní potřeby. Namísto toho musíte přizpůsobit program funkcím knihovny. Dokonce i když prodejce zdrojový kód dodá, riskujete při změnách nechtěnou úpravu v části funkce nebo změnu vztahů mezi funkcemi knihovny.

Třídy v C++ přináší vyšší úroveň znovupoužitelného kódu. Mnoho prodejců nyní nabízí knihovny tříd, obsahující jak deklarace tříd, tak i jejich implementace. Protože třída kombinuje reprezentaci dat s metodami třídy, nabízí celistvější balík než knihovna funkcí. Jedna třída může například poskytovat všechny zdroje pro správu dialogového okna. Často jsou knihovny tříd k dispozici ve zdrojovém kódu, takže je můžete upravit podle svých potřeb. V C++ ale pro rozšíření a úpravu tříd existuje lepší metoda, než je úprava kódu. Tato metoda, zvaná *dědičnost*, umožňuje odvozovat nové třídy od již existujících tříd, přičemž odvozená třída zdědí všechny vlastnosti, včetně metod exis-

tující třídy, které se říká *základní třída*. Stejně jako je obvykle jednodušší jmění zdědit než vytvořit od začátku, je i odvození třídy obvykle jednodušší než navržení třídy nové. Zde je několik věcí, kterých lze dosáhnout pomocí dědičnosti:

- Existující třídě můžete přidat další funkčnost. Máte-li například základní třídu reprezentující pole, můžete přidat aritmetické operace.
- Můžete přidat datové položky. Máte-li například základní třídu reprezentující řetězec, můžete do odvozené třídy přidat datovou položku reprezentující barvu, která se použije při zobrazení řetězce.
- Můžete upravit způsob chování metody třídy. Máte-li například třídu `Passanger` (cestující), reprezentující služby poskytované cestujícímu v letadle, můžete odvodit třídu `Upgrade`, která bude poskytovat vyšší úroveň služeb.

Stejného cíle byste samozřejmě mohli dosáhnout duplikováním a úpravou kódu původní třídy, ale díky mechanismu dědičnosti vám stačí pouze přidat nové vlastnosti. Třídu můžete odvodit, aniž byste museli mít přístup ke zdrojovému kódu. Jestliže tedy koupíte knihovnu tříd, která obsahuje pouze hlavičkové soubory a zkompilovaný kód metod třídy, můžete přesto z tříd knihovny odvozovat nové třídy. Na druhou stranu můžete dodávat své třídy ostatním a přitom utajit způsob implementace, ale vašim klientům stále ponecháváte možnost přidat těmto třídám vlastnosti.

Dědičnost je skvělý koncept a její základní implementace je docela jednoduchá. Má-li však dědičnost správně fungovat ve všech situacích, budete v ní muset provést nějaké úpravy. V této kapitole se podíváme jak na ty jednoduché, tak i na záludné aspekty dědičnosti.

Jednoduchá základní třída

V případech, kdy jedna třída dědí od druhé, se původní třídě říká základní a třídě dědící se říká odvozená. Máme-li tedy předvést dědičnost, potřebujeme nejdříve základní třídu. Naštěstí Pontoon National Bank potřebuje třídu, reprezentující základní kontrolní program. Ve výpisu 12.1 je hlavičkový soubor efektivní třídy `BankAccount`, vyhovující této potřebě. Obsahuje datové položky reprezentující jméno klienta, číslo účtu a zůstatek, a metody pro vytvoření účtu, přijímání vkladů, vydávání plateb a zobrazení dat o účtu. Pro reprezentaci opravdového bankovního účtu je tato třída nedostačující, pro naši imaginární banku a úvod do studia dědičnosti však její vlastnosti postačují.

Výpis 12.1. bankacct.h

```
// bankacct.h—jednoduchá třída BankAccount
#ifndef _BANKACCT_H_
#define _BANKACCT_H_
class BankAccount
{
private:
    enum {MAX = 35};
    char fullName[MAX];
```

```
    long acctNum;
    double balance;
public:
    BankAccount(const char *s = "Nullbody", long an = -1,
                double bal = 0.0);
    void Deposit(double amt);
    void Withdraw(double amt);
    double Balance() const;
    void ViewAcct() const;
};
#endif
```

Ve třídě je použita technika vytvoření třídní konstanty pomocí výčtu.

Následují metody třídy uvedené ve výpisu 12.2.

Výpis 12.2. bankacct.cpp

```
// bankacct.cpp—metody třídy BankAccount
#include <iostream>
using namespace std;
#include "bankacct.h"
#include <cstring>
BankAccount::BankAccount(const char *s, long an, double bal)
{
    strncpy(fullName, s, MAX - 1);
    fullName[MAX - 1] = '\0';
    acctNum = an;
    balance = bal;
}
void BankAccount::Deposit(double amt)
{
    balance += amt;
}
void BankAccount::Withdraw(double amt)
{
    if (amt <= balance)
        balance -= amt;
    else
        cout << "Pozadovana castka $" << amt
        << " prevysuje zustatek.\n"
        << "Vyber stornovan.\n";
}
double BankAccount::Balance() const
{
    return balance;
}
void BankAccount::ViewAcct() const
{
    // nastavi ###.## formát
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
```

```
        cout.setf(ios_base::showpoint);
        cout.precision(2);
        cout << "Klient: " << fullName << endl;
        cout << "Cislo uctu: " << acctNum << endl;
        cout << "Zustatek: $" << balance << endl;
        cout.setf(initialState); // obnoví původní formát
    }
```

Kompatibilita:

Jestliže váš kompilátor nepodporuje hlavičkový soubor `cstring`, použijte soubor `string.h`. Pokud kompilátor nepodporuje třídu `ios_base`, použijte třídu `ios`. V takovém případě použijte pro proměnnou `initialState` typ `long` namísto typu `ios_base::fmtflags`.

V implementaci je několik věcí, o kterých je třeba se zmínit. Za prvé, metoda `Withdraw()` zkontroluje, zda zůstatek na účtu stačí pro výběr. Pokud ne, není výběr povolen. Jedná se o příklad rozhraní s ochranou, která by chyběla, jestliže by bylo možné přistupovat k datovým položkám přímo.

Za druhé, metoda `ViewAcct()` pomocí formátovacích příkazů zobrazuje částky ve formátu $2356.32. Podobná nastavení používaly i dřívější příklady (podrobněji budou probrány v kapitole 16), ale tento navíc uchovává původní formátovací informace:

```
ios_base::fmtflags initialState =
    cout.setf(ios_base::fixed, ios_base::floatfield);
```

Zde `ios_base::fmtflags` je typ definovaný ve třídě `ios_base`. (Knihovny používající starší třídu `ios` používají typ `long`.) Volání metody `setf()` nastaví výstupní zobrazovací formát s pevnou desetinnou tečkou a vrátí nastavení formátovacího příznaku, který existoval před voláním metody. Díky tomu dokáže funkce po skončení obnovit původní nastavení:

```
cout.setf(initialState); // obnoví původní formát
```

Následující výpis 12.3 obsahuje program, který ilustruje vlastnosti třídy.

Výpis 12.3. usebank.cpp

```
// usebank.cpp
// zkompilovat společně s bankacct.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "bankacct.h"
int main()
{
    BankAccount Porky("Porcelot Pigg", 381299, 4000.00);
    Porky.ViewAcct();
    Porky.Deposit(5000.00);
    cout << "Novy zustatek: $" << Porky.Balance() << endl;
    Porky.Withdraw(8000.00);
```

```
    cout << "Novy zustatek: $" << Porky.Balance() << endl;
    Porky.Withdraw(1200.00);
    cout << "Novy zustatek: $" << Porky.Balance() << endl;
    return 0;
}
```

Kompatibilita:

Jestliže váš kompilátor nepodporuje hlavičkový soubor `cstring`, použijte soubor `string.h`.

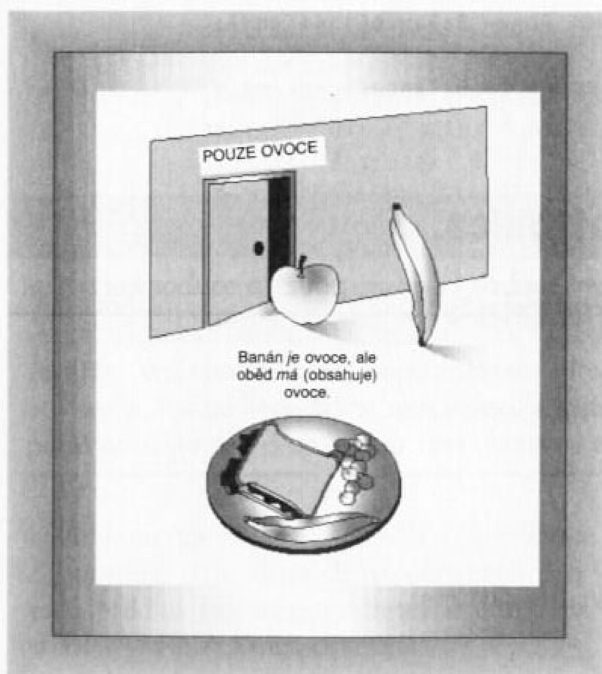
Zde je výstup programu:

```
  Klient: Porcelot Pigg
Cislo uctu: 381299
Zustatek: $4000.00
  Novy zustatek: $9000.00
Novy zustatek: $1000.00
  Pozadovana castka $1200.00 prevysuje zustatek.
  Vyber stornovan.
Novy zustatek: $1000.00
```

Dědičnost – vztah je

Nyní, když se třída `BankAccount` zařadila mezi fungující třídy, můžete od ní odvodit třídu novou. Předtím se však podíváme na základní model dědičnosti jazyka C++. Existují tři varianty dědičnosti: veřejná, chráněná a soukromá. Veřejná dědičnost je nejběžnější formou dědičnosti a modeluje vztah *je*. Takto je zkráceně řečeno, že objekt odvozené třídy by měl být také objektem základní třídy. Cokoli uděláte s objektem základní třídy, měli byste být schopni udělat i s objektem odvozené třídy. Předpokládejme například, že máte třídu `Fruit`. Ta by měla například ukládat informace o váze a kalorickém obsahu ovoce. Protože banán je konkrétním typem ovoce, můžete třídu `Banana` odvodit od třídy `Fruit`. Nová třída zdědí všechny datové položky původní třídy, takže objekt třídy `Banana` bude mít položky reprezentující váhu a kalorický obsah banánu. Do nové třídy `Banana` také můžete přidat položky speciálně pro banány, které neplatí pro ovoce obecně, jako je například hodnota Banana Institute Peel Index. Protože lze do odvozené třídy přidávat vlastnosti, je pravděpodobně přesnější popisovat tento vztah výrazem *je-druhu*, ale výraz *je* je obvyklý.

Abychom objasnili vztah *je*, podívejme se na některé příklady, které nespadají do tohoto vztahu. Veřejná dědičnost nemodeluje vztah *má*. Oběd může například obsahovat ovoce. Ale oběd obecně není ovoce. Ve snaze zahrnout ovoce do oběda byste tedy neměli odvozovat třídu `Lunch` od třídy `Fruit`. Chcete-li do oběda zahrnout ovoce, je správné uvažovat o vztahu *má*: oběd *má* (obsahuje) ovoce. Jak uvidíte v kapitole 13, tento případ nejsnáze představuje model, ve kterém je objekt třídy `Fruit` datovou položkou třídy `Lunch` (viz obrázek 12.1).



Obrázek 12.1. Vztah má a vztah je. FRUIT ONLY

Veřejná dědičnost nemodeluje vztah *je-jako*, to znamená, že nedělá přirovnání. Často se říká, že právníci jsou jako žraloci. Doslova ale není pravda, že právník je žralok. Žralok například žije pod vodou. Neměli byste tedy odvozovat třídu `Lawyer` (Právník) od třídy `Shark (Žralok)`. Dědičnost může základní třídě vlastnosti přidat, žádné ji však neodebere. V některých případech lze sdílené charakteristiky řešit navržením třídy, která tyto charakteristiky obsahuje a potom pomocí této třídy a vztahu *je* nebo *má* definovat příbuznou třídu.

Veřejná dědičnost nemodeluje vztah *je-implementován-jako*. Zásobník můžete například implementovat pomocí pole. Nebylo by však správné odvodit třídu `Stack` (zásobník) od třídy `Array` (pole). Zásobník není pole. Například indexování pole není vlastností zásobníku. Navíc zásobník může být implementován i jiným způsobem, například pomocí spojového seznamu. Správný přístup by spočíval v ukrytí implementace pole tím způsobem, že byste ze zásobníku udělali soukromou položku třídy `Array`.

Veřejná dědičnost nemodeluje vztah *používá*. Počítač například používá laserovou tiskárnu, ale nemá smysl odvozovat třídu `Printer` (tiskárna) od třídy `Computer` (počítač) a obráceně. Můžete však vytvořit spřátelené funkce nebo třídy pro komunikaci mezi objekty třídy `Printer` a objekty třídy `Computer`.

V C++ vám nic nebrání, abyste pomocí veřejné dědičnosti modelovali vztahy *má*, *je-implementován-jako* či *používá*. Takové praktiky však většinou vedou k programovým problémům. Držme se tedy vztahů *je*. Naštěstí se ukazuje, Pontoon National Bank také nabízí běžný účet Brass Plus. Tento účet má všechny vlastnosti běžného účtu Brass, ale navíc nabízí ochranu přečerpání účtu. To znamená, že jestliže vypíšete šek na větší částku (do určité výše), než je zůstatek, banka šek přijme s tím, že vám naúčtuje překročenou částku a připočte penále. Chtěli byste použít třídu `BankAccount`, ale ta novou vlastnost nepodporuje. Můžete však nadefinovat novou třídu, která zdědí všechny vlastnosti třídy

BankAccount a navíc bude mít potřebnou funkčnost. Tím, že místo psaní nové třídy vyjdete ze třídy BankAccount, zužitkujete práci vynaloženou na vývoj třídy BankAccount a využijete skutečnosti, že třída BankAccount je již vyzkoušena. Jinými slovy, použijete vyzkoušený kód. Výsledkem je méně práce a možná lepší produkt.

Obstojí nová třída (nazvěte ji Overdraft) v testu vztahu *je*? Samozřejmě ano. Vše co platí o objektu třídy BankAccount, bude platit i o objektu třídy Overdraft. Můžete tedy vkládat peníze nebo je vybírat a zobrazovat informace o účtu. Všimněte si, že vztah *je* není obecně oboustranný. Ovoce obecně není banán. A objekt třídy BankAccount nebude umět vše co objekt třídy Overdraft.

Deklarace odvozené třídy

Odvozená třída musí identifikovat třídu, ze které je odvozena. V C++ se název základní třídy vloží do deklarace odvozené třídy. Pokud odvozujete třídu Overdraft od třídy BankAccount, pak bude deklarace třídy začínat takto:

```
class Overdraft : public BankAccount
{
```

Dvojtečka označuje, že třída Overdraft je založena na třídě BankAccount. Tato konkrétní hlavička značí, že třída BankAccount je veřejnou základní třídou. Tomuto odvození se říká *veřejné odvození*. Objekt odvozené třídy obsahuje objekt třídy základní. Při použití veřejného odvození se veřejné položky základní třídy stanou veřejnými položkami odvozené třídy. Soukromé části základní třídy se stanou součástí odvozené třídy, ale přístup k nim bude možný pouze pomocí veřejných a chráněných metod základní třídy. (K chráněným metodám se dostaneme za chvíli.)

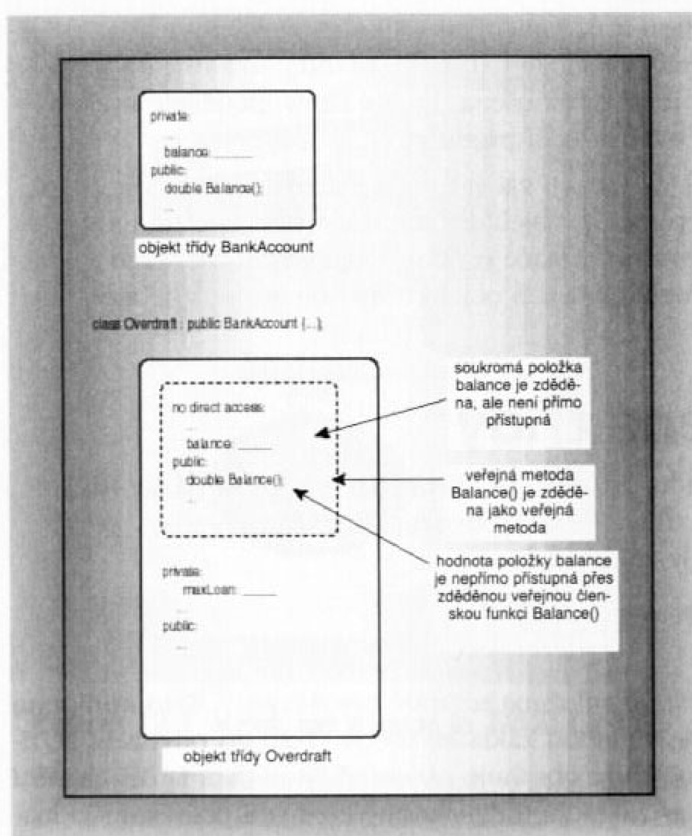
Funkce Deposit() se také například stane veřejnou funkcí třídy Overdraft. Položka balance třídy BankAccount se stane součástí objektu třídy Overdraft, ale přístup k ní bude možný pouze pomocí metod třídy BankAccount, jako je metoda Deposit() a konstruktory třídy. Stručně řečeno, třída Overdraft zdědí od základní třídy veřejné položky zároveň i přístup k nim. Nebudete je muset pro novou třídu předefinovat. Odvozená třída obsahuje také soukromé položky základní třídy, ale nemůže k nim přistupovat jinak než pomocí veřejných a chráněných metod základní třídy (viz obrázek 12.2).

C++ také podporuje chráněné a soukromé odvození:

```
class Computer : protected HardDisk
{…};
class House : private Study
{…};
```

Tyto formy probereme v kapitole 13. Nyní je hlavní, abyste věděli, že vynecháte-li klíčové slovo přístupu, C++ vytvoří soukromé odvození:

```
class House : Study // stejné jako private Study
{…};
```



Obrázek 12.2. Objekty základní a odvozené třídy

Jakmile třídu odvodíte, můžete k ní přidat nové položky. Vlastně musíte přidat nové konstruktory. Konstruktor totiž musí mít stejný název jako třída:

```
BankAccount bogey;    // potřebuje konstruktor BankAccount()
Overdraft orson;      // potřebuje konstruktor Overdraft()
```

Když vytvoříte objekt odvozené třídy, program nejdříve zavolá konstruktor základní třídy a konstruktor odvozené třídy. Toto je logické, protože konstruktor odvozené třídy může vycházet z datových položek základní třídy; objekt základní třídy tedy musí být vytvořen jako první. Je to něco, jako když se v budově postaví přízemí a potom se přidá první patro. Nově definované konstruktory by tedy neměly duplikovat práci konstruktorů základní třídy. Měly by ošetřovat jen to, co bylo přidáno a co odvozená třída potřebuje. Konstruktor by například mohl inicializovat nové datové položky. Obecně musí konstruktor odvozené třídy předat informace konstruktoru základní třídy. Brzy se podíváme na techniku, která se k tomu používá.

Nový destruktor explicitně přidat nemusíte, pokud nová třída nežádá uklidit něco, co neprovede destruktor základní třídy. Když platnost objektu skončí, program nejdříve zavolá případný destruktor odvozené třídy, potom destruktor třídy základní.

Pamatujte

Když vytváříte objekt odvozené třídy, program nejdříve zavolá konstruktor základní třídy a až potom konstruktor odvozené třídy. Když platnost objektu odvozené třídy skončí, program nejdříve zavolá destruktor odvozené třídy a potom destruktor třídy základní.

Obecně odvozená třída zdědí členské funkce třídy základní. Jestliže jsou členské funkce základní třídy veřejné nebo chráněné, mohou je objekty odvozené třídy vyvolat. Výjimku tvoří konstruktory a destruktory, ty ale mohou využít konstruktory a destruktory třídy základní, jak uvidíte v příkladech. Další neděděnou členskou funkcí je operátor přiřazení. Ten si zaslouží zvláštní pojednání, ke kterému se časem dostaneme. Nezapomeňte, že spřátelené funkce nejsou členskými funkcemi, a proto nejsou děděny.

Začátek deklarace odvozené třídy již znáte:

```
class Overdraft : public BankAccount
{
```

Než však deklaraci dokončíte, musíte přesně vědět, jaké vlastnosti má účet Brass Plus navíc. Diskuse s vlídným zástupcem banky Pontoon National Bank odhalila následující vlastnosti:

- Účet Brass Plus limituje částku, kterou banka zapůjčí pro krytí překročení účtu. Implicitní částka je 500 dolarů, ale u některých zákazníků může být z počátku použit jiný limit.
- Banka může zákazníkovi limit pro překročení změnit.
- Zapůjčená částka je zatížena úrokem. Standardní hodnota je 10 %, ale u některých zákazníků může být z počátku úrok jiný.
- Banka může zákazníkovi úrokovou sazbu změnit.
- Účet zaznamenává velikost dluhu (půjčky při překročení a úroky). Zákazník nemůže tuto částku splatit běžným vkladem nebo převodem z jiného účtu. Musí zaplatit v hotovosti vyčleněnému bankovnímu úředníkovi, který ho v případě nutnosti vyhledá. Po zaplacení dluhu je dlužná částka vynulována.

Poslední vlastnost není pro bankovnictví obvyklá, ale má dobré vedlejší efekty, které problém programování zjednodušují.

Tento seznam napovídá, že nová třída potřebuje datové položky, které budou obsahovat maximální hodnotu dluhu, úrokovou sazbu a aktuální výši dluhu. Musí mít konstruktory, které obsahují informace o účtu a také limit dlužné částky s implicitní hodnotou ve výši 500 dolarů a úrokovou sazbu s implicitní hodnotou 10 %. Měla by také mít metody pro nastavení limitu dluhu, úrokové sazby a aktuálního dluhu. Toto všechno musí být přidáno třídě `BankAccount` a deklarováno to bude v deklaraci třídy `Overdraft`.

Chování některých metod třídy `BankAccount` by se také mělo změnit. Konkrétně metody `Withdraw()` a `ViewAcct()` musí dělat více práce, než dělaly dříve. Veřejná dědičnost umožňuje předefinovat metody základní třídy, takže ve třídě `Overdraft` budete muset předefinovat metody `Withdraw()` a `ViewAcct()`.

Metoda `Deposit()` však funguje stejně pro obě třídy. Odvozená třída používá metodu základní třídy, pokud tato není předefinována. Takže pro použití metody `Deposit()` základní třídy nemusí třída `Overdraft` udělat vůbec nic.

Výpis 12.4 ukazuje třídu deklarovanou podle uvedených bodů. Všimněte si, že se základní třídou nemusíte dělat vůbec nic, abyste z ní mohli odvozovat nové třídy. Veškeré odvozování spočívá v definování nové třídy a jejich metod. Třídu tedy můžete odvodit, aniž byste museli mít přístup ke zdrojovému kódu základní třídy.

Výpis 12.4. overdrft.h

```
// overdrft.h — deklarace třídy Overdraft
#ifndef _OVERDRFT_H_
#define _OVERDRFT_H_
#include "bankacct.h"
class Overdraft : public BankAccount
{
private:
      double maxLoan;
      double rate;
      double owesBank;
public:
      Overdraft(const char *s = "Nullbody", long an = -1,
                double bal = 0.0, double ml = 500,
                double r = 0.10);
      Overdraft(const BankAccount & ba, double ml = 500, double
                r = 0.1);
      void ViewAcct()const;
      void Withdraw(double amt);
      void ResetMax(double m) { maxLoan = m; }
      void ResetRate(double r) { rate = r; };
      void ResetOwes() { owesBank = 0; }
};
#endif
```

Implementace odvozené třídy

Nyní prozkoumáme způsob implementace odvozené třídy a podíváme se na racionálnost některých metod. Začneme u konstruktorů. Nejdříve se zamysleme nad procesem tvorby objektu. Program nemůže vytvořit objekt třídy `Overdraft` dříve, než vytvoří objekt třídy `BankAccount`. Konstruktor základní třídy tedy musí být vyvolán dříve než program vstoupí do kódu konstruktoru odvozené třídy. Na druhé straně, konstruktor základní třídy nemůže být vyvolán předtím, než bude vyvolán konstruktor odvozené třídy, protože právě volání konstruktoru odvozené třídy programu říká, že je potřeba volat konstruktor základní třídy. Uvažujte následující konstruktor třídy `Overdraft`:

```
Overdraft(const char *s = "Nullbody", long an = -1,
          double bal = 0.0, double ml = 500,
          double r = 0.10);
```

Metoda má pět parametrů, přičemž tři z nich poskytují hodnoty pro položky třídy BankAccount a dva pro položky třídy `Overdraft`. Použít poslední dva parametry je dosti jednoduché:

```
// nekompletní verze
Overdraft::Overdraft(const char *s, long an, double bal, double
                     ml, double r)
{
    maxLoan = ml;
    owesBank = 0.0; // na počátku bez dluhu
    rate = r;
}
```

Ale co s částí `BankAccount`? Nejdříve uvažujte, co by se stalo, kdybyste použili tuto nekompletní verzi konstruktoru. Objekt základní třídy se vytvoří před přidáním odvozené části. V syntaxi tohoto konstruktoru to znamená, že objekt základní třídy se vytvoří dříve, než program začne provádět příkazy v těle konstruktoru. Protože explicitně není uveden žádný konstruktor, znamená to, že část tvořená základní třídou se vytvoří pomocí implicitního konstruktoru třídy `BankAccount`. Jinými slovy: pokud není řečeno jinak, konstruktor odvozené třídy volá implicitní konstruktor základní třídy před vstupem do těla konstruktoru odvozené třídy.

V tomto případě však implicitní konstruktor není tím pravým, protože místo požadovaných hodnot používá hodnoty implicitní. C++ nabízí speciální syntaxi pro určení volaného konstruktoru. Je to varianta syntaxe seznamu inicializátorů, který jste viděli v kapitole 11, s tím rozdílem, že namísto názvu položky použijete název třídy:

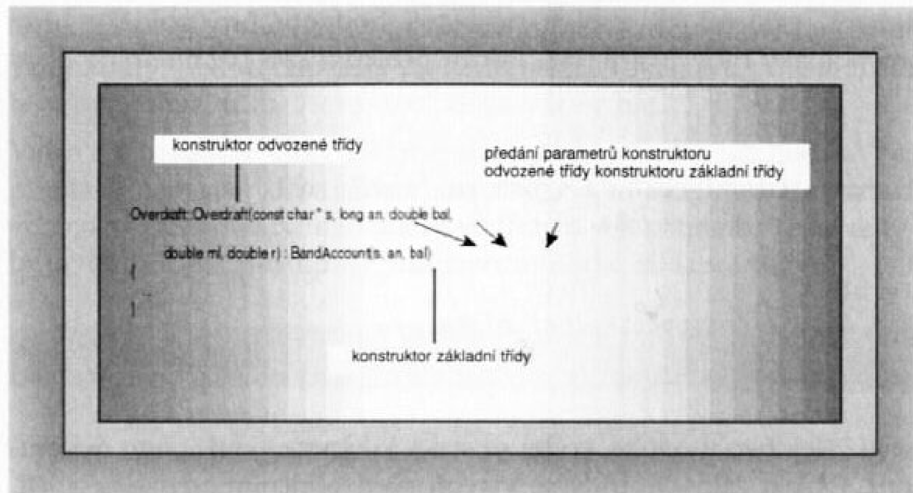
```
Overdraft::Overdraft(const char *s, long an, double bal, double
                     ml, double r) : BankAccount(s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
```

Zde část

```
: BankAccount(s, an, bal)
```

znamená „Zavolej konstruktor `BankAccount(const char *, double, double)` a vytvoř část objektu třídy `Overdraft`, která náleží základní třídě". Díky tomuto mechanismu jsou první tři parametry konstruktoru třídy `Overdraft` předány konstruktoru třídy `BankAccount`. Konstruktor třídy `BankAccount` tedy nastaví zděděné položky a tělo konstruktoru třídy Overdraft nastaví hodnoty nových položek (viz obrázek 12.3).

Stručně řečeno, konstruktor odvozené třídy vždy vyvolá konstruktor základní třídy, než začne provádět příkazy v těle konstruktoru odvozené třídy. Program použije standardní konstruktor základní třídy, pokud konstruktor neurčíte explicitně pomocí syntaxe seznamu inicializátorů. V takovém případě můžete použít parametry konstruktoru odvozené třídy jako parametry konstruktoru základní třídy.



Obrázek 12.3. Předávání parametrů konstruktoru základní třídy

Pamatujte

Konstruktory odvozené třídy jsou zodpovědné za inicializaci všech datových položek, které byly přidány k položkám zděděným. Konstruktory základní třídy jsou zodpovědné za inicializaci zděděných položek. Chcete-li označit, pomocí kterého konstruktoru základní třídy se má inicializovat, můžete použít syntaxi seznamu inicializátorů. Pokud tak neučiníte, použije se implicitní konstruktor základní třídy.

Seznamy inicializátorů

Konstruktor odvozené třídy může použít mechanismus inicializačního seznamu pro předání hodnot konstruktoru základní třídy.

```
derived::derived(typel x, type2 y) : base(x,y)
// seznam inicializátorů
{
        ...
}
```

Zde `derived` představuje odvozenou třídu, `base` základní třídu a proměnné `x` a `y` jsou používány konstruktorem základní třídy. Pokud budou konstruktoru předány například hodnoty 10 a 12, předá tento mechanismus hodnoty 10 a 12 konstruktoru základní třídy, ve kterém jsou definovány parametry těchto typů. S výjimkou virtuálních základních tříd (kapitola 14), může třída předat parametry zpět pouze nejbližší základní třídě. Tato třída však může pomocí stejného mechanismu vrátit informace své nejbližší základní třídě atd. Jestliže v seznamu inicializátorů konstruktor základní třídy neuvedete, použije program implicitní konstruktor základní třídy. Seznam inicializátorů lze použít pouze u konstruktorů.

V kapitole 11 jste poznali syntaxi pro inicializaci specifických položek třídy. Například konstruktor

```
Queue::Queue(int qs) : qsize(qs) // inicializace qsize na qs
{
    front = rear = NULL;
    items = 0;
}
```

inicializuje položku qsize objektu třídy Queue na hodnotu qs. Použitá syntaxe je variantou dříve uvedeného formátu. Rozdíl spočívá v tom, že pro inicializaci určité položky objektu použijete název položky třídy, zatímco pro inicializaci části objektu tvořené základní třídou použijete název základní třídy.

Inicializace objektů pomocí objektů

Podívejme se na druhý konstruktor třídy Overdraft:

```
Overdraft(const BankAccount & ba, double ml = 500, double r = 0.1);
```

Jeho záměrem je umožnit konverzi z účtu typu Brass na typ Brass Plus. Zde parametr ba poskytuje informace o starém účtu a zbylé parametry informace pro nové datové položky. Otázkou je, jak pomocí parametru BankAccount inicializovat část BankAccount. Protože se přitom vytvoří kopie objektu třídy BankAccount, měli byste použít kopírovací konstruktor:

```
Overdraft::Overdraft(const BankAccount & ba, double ml,double r)
        : BankAccount(ba)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
```

Je pravda, že deklarace třídy BankAccount explicitně kopii kopírovacího konstruktoru nedefinuje. Ale vzpomeňte si, že kompilátor vytvoří implicitní kopírovací konstruktor, pokud je potřeba a jestliže jste žádný nedefinovali. Implicitní kopírovací konstruktor provádí kopírování po položkách, což je v případě objektu třídy BankAccount dostačující.

Ostatní členské funkce

Třída Overdraft nedefinuje funkci Deposit(), což znamená, že objekt Overdraft použije funkci BankAccount::Deposit(). Totéž platí i pro funkci Balance(). Metodu ViewAcct() ale nová třída definuje a to znamená, že objekt třídy Overdraft bude používat funkci Overdraft::ViewAcct(). Podívejme se na její implementaci. Nejdříve něco, co nebude fungovat:

```
void Overdraft::ViewAcct() const   // NEFUNKČNÍ VERZE
{
    // nastaveni formátu ###.##
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);
    cout << "Klient: " << fullName << endl;          // nefunkční
    cout << "Cislo uctu: " << acctNum << endl;   // nefunkční
```

```
        cout << "Zustatek: $" << balance << endl;          // nefunkčni
        cout << "Maximalni vyse pujcky: $" << maxLoan << endl;
        cout << "Dluzna castka: $" << owesBank << endl;
        cout.setf(initialState);
    }
```

Měli byste se ujistit, že chápete, v čem je problém. Odvozená třída nemůže přímo přistupovat k soukromým datům a metodám základní třídy. Objekt třídy `Overdraft` sice obsahuje objekt třídy `BankAccount` s položkami `fullName`, `acctNum` a `balance`, ale nemůže k nim přistupovat podle názvu. Vtip je v tom, že veřejná část základní třídy definuje její rozhraní a zbytek programu včetně odvozených tříd by toto rozhraní měly používat. Třída `Overdraft` může například pomocí veřejného rozhraní třídy `BankAccount` přistupovat k datům této třídy. Metoda třídy `Overdraft` může například použít metodu `BankAccount::ViewAcct()`:

```
void Overdraft::ViewAcct() const
{
    // nastavení formátu ###.##
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);
    BankAccount::ViewAcct();   // zobrazí základní část
    cout << "Maximalni vyse pujcky: $" << maxLoan << endl;
    cout << "Dluzna castka: $" << owesBank << endl;
    cout.setf(initialState);
}
```

Nesmíte však zapomenout na operátor rozlišení. Kdybyste ho vynechali, volala by metoda `Overdraft::ViewAcct()` rekurzivně samu sebe:

```
void Overdraft::ViewAcct() const
{
...
    ViewAcct();     // NE! rekurzivní volání metody Overdraft::ViewAcct()
    BankAccount::ViewAcct();      // volá metodu ze základní třídy
...
}
```

Zopakujme si, kdy se které metody používají. Jestliže odvozená třída nepředefinuje metodu základní třídy, objekt odvozené třídy použije metodu základní třídy. Pokud ale odvozená třída tuto metodu předefinuje, použijí objekty odvozené třídy tuto novou definici. Říkáme, že definice odvozené třídy *přepisuje* definici třídy základní:

```
BankAccount bretta;
Overdraft ophelia;
bretta.Deposit(20);     // použije BankAccount::Deposit()
ophelia.Deposit(40);    // použije BankAccount::Deposit()
bretta.ViewAcct();      // použije BankAccount::ViewAcct()
ophelia.ViewAcct();     // použije Overdraft::ViewAcct()
```

Zbývá ještě jedna funkce, kterou je potřeba předefinovat. Nová verze funkce `Withdraw()` musí ošetřit ochranu přečerpání účtu. Pamatujte, že může využít verzi funkce `Withdraw()` `základní třídy, ale nemůže přímo přistupovat k položce` balance. Z toho vychází následující návrh:

```
void Overdraft::Withdraw(double amt)
{
    double bal = Balance();
    if (amt <= bal)
        BankAccount::Withdraw(amt);
    else if ( amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bankovni pujcka: $" << advance
            << endl;
        cout << "Financni poplatek: $" << advance *
            rate << endl;
        Deposit(advance);
        BankAccount::Withdraw(amt);
    }
    else
        cout << "Kreditni limit prekrocen. Transakce stornovana.
.\n";
}
```

Pokud zůstatek pokrývá částku, která se má vybrat, použijte verzi této funkce ze základní třídy. Jestliže je ochrana přečerpání potřebná a je tak malá, že ji zvládne účet Brass Plus, půjčte klientovi potřebnou částku, zaúčtujte mu ji společně s úrokem a proveďte výběr. Kompletní implementace třídy `Overdraft` je ve výpisu 12.5.

Výpis 12.5. overdrft.cpp

```
// overdrft.cpp — metody třídy Overdraft
#include <iostream>
using namespace std;
#include "overdrft.h"
Overdraft::Overdraft(const char *s, long an, double bal,
            double ml, double r) : BankAccount(s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
Overdraft::Overdraft(const BankAccount & ba, double ml,
            double r)
            : BankAccount(ba)  // použije implicitní kopírovací
                               // konstruktor
{
    maxLoan = ml;
    owesBank = 0.0;
```

```
        rate = r;
    }
    // předefinování metody ViewAcct()
    void Overdraft::ViewAcct() const
    {
        // nastavení formátu ###.##
        ios_base::fmtflags initialState =
            cout.setf(ios_base::fixed, ios_base::floatfield);
        cout.setf(ios_base::showpoint);
        cout.precision(2);
        BankAccount::ViewAcct();  // zobrazí základní části
        cout << "Maximalni vyse pujcky: $" << maxLoan << endl;
        cout << "Dluzna castka: $" << owesBank << endl;
        cout.setf(initialState);
    }
    // předefinování funkce Withdraw()
    void Overdraft::Withdraw(double amt)
    {
        // nastavení formátu ###.##
        ios_base::fmtflags initialState =
            cout.setf(ios_base::fixed, ios_base::floatfield);
        cout.setf(ios_base::showpoint);
        cout.precision(2);
        double bal = Balance();
        if (amt <= bal)
            BankAccount::Withdraw(amt);
        else if ( amt <= bal + maxLoan - owesBank)
        {
            double advance = amt - bal;
            owesBank += advance * (1.0 + rate);
            cout << "Bankovni pujcka: $" << advance << endl;
            cout << "Financni poplatek: $" << advance  * rate << endl;
            Deposit(advance);
            BankAccount::Withdraw(amt);
        }
        else
            cout << "Kreditni limit prekrocen. Transakce stornovana.\n";
        cout.setf(initialState);
    }
```

Dalším krokem je vyzkoušení odvozené třídy. K tomu slouží krátký program ve výpisu 12.6. Měli byste ho zkompilovat společně s `overdrft.cpp` a `bankacct.cpp`, protože odvozená třída používá definice základní třídy.

Výpis 12.6. useover.cpp

```
#include <iostream>
  using namespace std;
  #include "overdrft.h"
  int main()
  {
```

```
    BankAccount Porky("Porcelot Pigg", 381299, 4000.00);
// konverze na nový typ účtu
    Overdraft Porky2(Porky);
    Porky2.ViewAcct();
    cout << "Vlozeno $5000:\n";
    Porky2.Deposit(5000.00);
    cout << "Novy zustatek: $" <<
    Porky2.Balance() << "\n\n";
    cout << "Vybrano $8000:\n";
    Porky2.Withdraw(8000.00);
    cout << "Novy zustatek: $" <<
    Porky2.Balance() << "\n\n";
    cout << "Vybrano $1200:\n";
    Porky2.Withdraw(1200.00);
    Porky2.ViewAcct();
    cout << "\nVybrano $500:\n";
    Porky2.Withdraw(500.00);
    Porky2.ViewAcct();
    return 0;
}
```

Zde je výstup programu:

```
Klient: Porcelot Pigg
Cislo uctu: 381299
Zustatek: $4000.00
Maximalni vyse pujcky: $500.00
Dluzna castka: $0.00
Vlozeno $5000:
Novy zustatek: $9000.00
Vybrano $8000:
Novy zustatek: $1000.00
Vybrano $1200:
Bankovni pujcka: $200.00
Financni poplatek: $20.00
Klient: Porcelot Pigg
Cislo uctu: 381299
Zustatek: $0.00
Maximalni vyse pujcky: $500.00
Dluzna castka: $220.00
Vybrano $500:
Kreditni limit prekrocen. Transakce stornovana.
Klient: Porcelot Pigg
Cislo uctu: 381299
Zustatek: $0.00
Maximalni vyse pujcky: $500.00
Dluzna castka: $220.00
```

Poznámky k programu

Prozkoumejme některé části programu podrobněji. Nejdříve příkaz

```
Overdraft Porky2(Porky);
```

vytvoří objekt třídy Overdraft a inicializuje jeho základní část BankAccount pomocí hodnot uložených v objektu Porky. Nové položky maxLoan a rate jsou inicializovány pomocí implicitních hodnot ($500 a 10%).

Všechny příkazy

```
Porky2.ViewAcct();
```

vyvolají metodu Overdraft::ViewAcct(), která zase pomocí explicitně vyvolané metody BankAccount::ViewAcct() zobrazí položky základní třídy.

Příkaz

```
Porky2.Deposit(5000.00);
```

vyvolá metodu BankAccount::Deposit() a přidá na účet 5000 dolarů.

Příkaz

```
Porky2.Withdraw(1200.00);
```

vyvolá metodu Overdraft::Withdraw(), která uživateli půjčí 200 dolarů, zaúčtuje dlužnou částku 220 dolarů a potom funkce BankAccount::Withdraw() provede samotný výběr.

Řízení přístupu – chráněný režim

Prozatím příklady tříd používaly pro řízení přístupu ke svým položkám klíčová slova public a private. Existuje ještě jedna kategorie přístupu, označovaná klíčovým slovem protected. Způsob přístupu protected je podobný způsobu private v tom, že z okolního světa lze k položkám třídy v části protected přistupovat pouze pomocí členů třídy. Rozdíl mezi nimi se projeví pouze u tříd odvozených přímo od základní třídy. Metody odvozené třídy mohou k chráněným položkám základní třídy přistupovat přímo, nemohou však přímo přistupovat k jejím soukromým položkám. Položky z kategorie chráněných se tedy vůči okolnímu světu chovají jako položky soukromé, ale vůči odvozeným třídám se chovají jako položky veřejné.

Předpokládejme například, že třída BankAccount deklaruje položku balance jako protected:

```
class BankAccount
{
protected:
     double balance;
...
};
```

Potom by třída Overdraft mohla k položce balance přistupovat přímo bez pomoci metod třídy BankAccount. Jádro metody Withdraw() by mohlo být napsáno například takto:

```
void Overdraft::Withdraw(double amt)
{
     if (amt <= balance)     // přímý přístup k balance
          balance -= amt;
     else if ( amt <= balance + maxLoan - owesBank)
     {
```

```
            double advance = amt - balance;
            owesBank += advance * (1.0 + rate);
            cout << "Bankovni pujcka: $" << advance << endl;
            cout << "Financni poplatek: $" << advance * rate << endl;
            Deposit(advance);
            balance -= amt;
        }
        else
            cout << "Kreditni limit prekrocen. Transakce stornovana.\n";
    }
```

Použití chráněných datových položek může zjednodušit psaní kódu, ale má to jistý nedostatek v návrhu. Vyjdeme-li například z předchozího příkladu, pak jestliže byla proměnná `balance` chráněná, mohli byste napsat tento kód:

```
void Overdraft::Reset(double amt)
{
    balance = amt;
}
```

Třída `BankAccount` byla navržena tak, že jediným prostředkem pro změnu hodnoty proměnné balance byly metody `Withdraw()` a `Deposit()`. Ale metoda `Reset()` vlastně z proměnné balance činí vůči objektům třídy `Overdraft` proměnnou veřejnou a ignoruje například zabezpečení nalezená v metodě `Withdraw()`.

Upozornění

U datových položek dávejte přednost soukromému řízení přístupu před chráněným a odvozeným třídám zpřístupněte data základní třídy pomocí metod této základní třídy.

Chráněné řízení přístupu může být docela užitečné u členských funkcí, neboť odvozeným třídám dává přístup k vnitřním funkcím, které nejsou dostupné veřejně.

Vztah je, reference a ukazatele

Jeden způsob, jak veřejná dědičnost modeluje vztah *je*, vychází z používání referencí a ukazatelů na objekty, způsob zacházení s referencemi a ukazateli. Normálně v C++ nemůžete přiřadit adresu jednoho typu ukazateli na jiný typ. Také není možné, aby reference na jeden typ odkazovala na typ jiný:

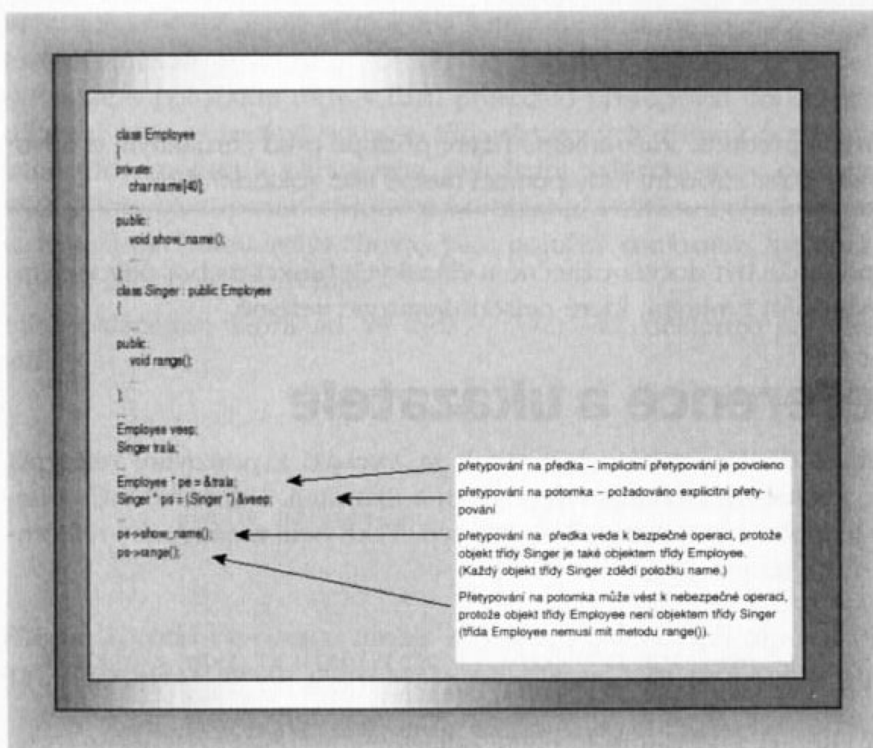
```
double x = 2.5;
int * pi = &x;    // neplatné přiřazení, neodpovídají si typy ukazatelů
long & rl = x;    // neplatné přiřazení, neodpovídají si typy referencí
```

Ale reference nebo ukazatel na základní třídu může odkazovat na objekt odvozené třídy, aniž by bylo potřeba použít explicitní přetypování. Například následující inicializace jsou přípustné:

```
Overdraft dilly ("Annie Dill", 493222, 2000);
BankAccount * pb = &dilly;  // ok
BankAccount & rb = dilly;   // ok
```

Konverze reference nebo ukazatele odvozené třídy na referenci nebo ukazatel základní třídy se nazývá *přetypování na předka* a u veřejné dědičnosti je vždy povolena bez nutnosti explicitního přetypování. Toto pravidlo je součástí vyjádření vztahu *je*. Objekt třídy `Overdraft` je také objektem třídy `BankAccount`, protože zdědí všechny datové položky a členské funkce objektu třídy `BankAccount`. To znamená, že vše co můžete udělat s objektem třídy `BankAccount`, můžete udělat i s objektem třídy `Overdraft`. Takže funkce sestrojená pro práci s referencí třídy `BankAccount` může provádět stejnou činnost v objektu třídy `Overdraft`, aniž byste se museli obávat nějakých problémů. Stejná myšlenka se uplatní, předáte-li jako parametr funkce ukazatel na objekt.

Opačný proces, konverze ukazatele nebo reference základní třídy na ukazatel nebo referenci odvozené třídy, se nazývá *přetypování na potomka* a není povoleno bez použití explicitního přetypování. Důvodem pro toto omezení je skutečnost, že vztah *je* není obecně oboustranný. Do odvozené třídy by mohly být přidány nové datové položky a členské funkce třídy používající tyto položky by u základní třídy nefungovaly. Předpokládejme například, že odvodíte třídu `Singer` (zpěvák) od třídy `Employee` (zaměstnanec), přičemž přidáme datovou položku reprezentující hlasové rozpětí zpěváka a členskou funkci `range()`, vracející hodnotu hlasového rozpětí. Nemělo by smysl použít metodu `range()` na objekt třídy `Employee`. Ale kdyby bylo povoleno implicitní přetypování na potomka, mohli byste ukazatel na objekt třídy `Singer` omylem nastavit na adresu objektu třídy `Employee` a s jeho pomocí vyvolat metodu `range()` (viz obrázek 12.4).



Obrázek 12.4. Přetypování na předka a přetypování na potomka

Virtuální členské funkce

Podívejme se podrobněji na metody, které jsou vyvolány, jestliže odvozená třída předefinuje metodu základní třídy. Dříve jste se dozvěděli, že rozhodující pro to, která metoda se použije, je typ volajícího objektu:

```
BankAccount bretta;  // objekt základní třídy
Overdraft ophelia;   // objekt odvozené třídy
bretta.ViewAcct();   // použije se BankAccount::ViewAcct()
ophelia.ViewAcct();  // použije se Overdraft::ViewAcct()
```

Předpokládejme však, že metodu vyvoláte pomocí ukazatele:

```
BankAccount * bp = &bretta;  // ukazuje na objekt třídy BankAccount
bp->ViewAcct();              // použije se BankAccount::ViewAcct()
bp = &ophelia;               // ukazatel třídy BankAccount na
                             // objekt třídy Overdraft
bp->ViewAcct();              // která verze se použije?
```

Pokud se bude kompilátor řídit typem ukazatele, vyvolá poslední příkaz metodu `BankAccount::ViewAcct()`. Ale jestliže se bude řídit typem objektu, na který ukazatel ukazuje, vyvolá metodu `Overdraft::ViewAcct()`. Co si tedy kompilátor vybere?

Implicitně vybírá C++ funkci podle typu ukazatele nebo reference a typ objektu ignoruje. Takže ve výše uvedeném případě by program použil metodu `BankAccount::ViewAcct()`. Pro takové chování existuje dobrý důvod – často totiž kompilátor typ nezná. Uvažujte například následující kód:

```
cout << "Chcete-li Brass Account zadejte 1, chcete-li Brass Plus
    Account, zadejte 2: ";
int kind;
cin >> kind;
BankAccount * bp;
if (kind == 1)
    bp = new BankAccount;
else if (kind == 2)
    bp = new Overdraft;
bp->ViewAcct();
```

V době kompilace kompilátor nemůže vědět, která volba (1 nebo 2) bude za běhu programu provedena, nemůže tedy vědět na jaký typ objektu bude ukazatel `bp` ukazovat. Takže jedinou možností v době kompilace je porovnat metody třídy s typem ukazatele nebo reference. Tato strategie se nazývá *časná vazba* nebo také *statická vazba*. Termín vazba označuje, že volaná funkce je spojena s konkrétní definicí funkce. Zde například musí kompilátor svázat volání funkce `bp->ViewAcct()` s jednou z metod `ViewAcct()`. (V jazyku C existuje pouze jedna funkce s daným názvem, takže volba je zřejmá. Ale v C++ může díky přetěžování a předefinování členských funkcí existovat více funkcí odpovídajících danému názvu.)

```
// statická vazba
bp = &ophelia;   // ukazatel třídy BankAccount na objekt třídy Overdraft
bp->ViewAcct();  // použije se BankAccount::ViewAcct()
```

Použití metody `BankAccount::ViewAcct()` s objektem třídy Overdraft žádnou škodu nenapáchá, pouze se nezobrazí všechna dostupná data. Bylo by tedy pěkné, kdyby volání `bp->ViewAcct()` mohlo zaznamenat typ objektu místo typu ukazatele a vyvolat místo toho metodu `Overdraft::ViewAcct()`. Jazyk C++ pro dosažení tohoto cíle nabízí druhou strategii nazývanou *pozdní vazba* nebo také *dynamická vazba*. Při použití této strategie kompilátor nerozhoduje, kterou metodu třídy použije. Místo toho přesune zodpovědnost na program, který potom učiní rozhodnutí za běhu vždy, když skutečně provede volání funkce. Použijete-li tuto strategii, bude program vybírat metodu podle typu objektu, na který reference či ukazatel odkazují:

```
// dynamická vazba
BankAccount * bp = &bretta;  // ukazuje na objekt třídy BankAccount
bp->ViewAcct ();              // použije se BankAccount::ViewAcct()
 bp = &ophelia;               // ukazatel třídy BankAccount na objekt
                              //třídy Overdraft
bp->ViewAcct();               // použije se Overdraft::ViewAcct()
```

Ve většině případů je dynamická vazba dobrou pomůckou.

Tato diskuse by ve vás měla vyvolat několik otázek:

- Jak aktivovat dynamickou vazbu?
- Proč existují dva druhy vazby?
- Když je dynamická vazba tak dobrá, proč není používána standardně?
- Jak funguje?

V další části se podíváme na odpovědi k těmto otázkám.

Aktivace dynamické vazby

Dynamickou vazbu můžete zapnout *pouze* pro členské funkce. Dosáhnete toho přidáním klíčového slova `virtual` před prototyp funkce v deklaraci základní třídy. Takové metodě se potom říká *virtuální metoda*. Jestliže následně předefinujete funkci v odvozené třídě, program pomocí dynamické vazby určí, která definice se má použít. Jakmile metodu učiníte virtuální, zůstane virtuální pro všechny třídy odvozené od tříd základních, pro všechny třídy odvozené od odvozených tříd atd. Pro danou metodu musíte klíčové slovo `virtual` použít pouze jednou, a to v základní třídě, v níž je poprvé definována.

Pamatujte

Virtuální členské funkce se vytvoří přidáním klíčového slova `virtual` před prototyp. Programy v C++ používají dynamickou či pozdní vazbu pro virtuální metody a statickou či časnou vazbu statickou pro metody nevirtuální. U virtuálních funkcí je pro vyvolání dané metody ukazatelem nebo referencí určující typ objektu, na který ukazatel či reference odkazují.

Výpis 12.7 ukazuje deklaraci třídy `BankAccount`, v níž došlo ke změně některých funkcí na funkce virtuální. Výpis je identický s výpisem 12.4 s tím rozdílem, že bylo dvakrát vloženo klíčové slovo `virtual` – jednou v deklaraci funkce `Deposit()` a jednou v deklaraci

funkce `ViewAcct()`. Mimo těchto dvou změn v deklaracích nemusíte dělat nic dalšího. Ostatní tři soubory podporující tyto dvě třídy zůstávají beze změny. Musíte je však znovu zkompilovat s novým hlavičkovým souborem.

Výpis 12.7. bankvirt.h

```
// bankvirt.h - jednoduchá třída BankAccount s virtuálními funkcemi
#ifndef _BANKACCT_H_
#define _BANKACCT_H_
class BankAccount
{
private:
     enum {MAX = 35};
     char fullName[MAX];
     long acctNum;
     double balance;
public:
     BankAccount(const char *s = "Nullbody", long an = -1,
                 double bal = 0.0);
     void Deposit(double amt);
     virtual void Withdraw(double amt);  // virtuální metoda
     double Balance() const;
     virtual void ViewAcct() const;      // virtuální metoda
};
#endif
```

Abyste viděli rozdíl mezi virtuálními a nevirtuálními funkcemi, spusťte program z výpisu 12.8 dvakrát. Nejdříve zkompilujte program s původní, nevirtuální verzí souboru `bankacct.h` (výpis 12.4). Podruhé zkompilujte program s použitím nové verze souboru `bankacct.h` (výpis 12.7). Program sám používá malé pole ukazatelů na objekty třídy `BankAccount`. Každý ukazatel v poli může podle rozhodnutí učiněného za běhu programu ukazovat buď na objekt třídy `BankAccount` nebo na objekt třídy `Overdraft`. Takové použití pole ukazatelů základní třídy je běžnou programovací technikou. Jestliže máte směsici objektů tříd `BankAccount` a `Overdraft`, nemůžete je vložit do stejného pole, protože každý prvek pole musí být stejného typu. Ale díky přetypování na předka můžete vytvořit pole ukazatelů na objekty základní třídy, které mohou ukazovat buď na objekty základní třídy nebo na objekty třídy odvozené. Můžete tedy pomocí jediného pole ukazatelů základní třídy spravovat směsici objektových typů.

Výpis 12.8. useover1.cpp

```
// useover1.cpp — test třídy Overdraft
  // zkompilovat s bankacct.cpp a overdrft.cpp
  #include <iostream>
  using namespace std;
  #include "overdrft.h"
  const int SIZE = 3;
  const int MAX = 35;
  inline void EatLine() {while (cin.get() != '\n') continue;}
```

```
int main()
{
    BankAccount * baps[SIZE];
    char name[MAX];
    long acctNum;
    double balance;
    int acctType;
    int i;
    for (i = 0; i < SIZE; i++)
    {
      cout << "Zadejte jmeno klienta: ";
      cin.get(name,MAX);
      EatLine();
      cout << "Zadejte cislo uctu klienta: ";
      cin >> acctNum;
      cout << "Zadejte pocatecni zustatek klienta: ";
      cin >> balance;
      cout << "Chcete-li Brass Account, zadejte 1, chcete-li Brass Plus"
            << "Account, zadejte 2: ";
      cin >> acctType;
      EatLine();
      if (acctType == 2)
            baps[i] = new Overdraft(name, acctNum, balance);
      else
      {
          baps[i] = new BankAccount(name, acctNum, balance);
          if (acctType != 1)
              cout << "Interpretovano jako 1.\n";
      }
    }
    for (i = 0; i < SIZE; i++)
    {
      baps[i]->ViewAcct();
      cout << endl;
    }
    cout << "Nashledanou!\n";
    return 0;
}
```

Nejdříve je zde ukázka běhu s použitím nevirtuálních funkcí:

```
Zadejte jmeno klienta: Rufus Overbeam
Zadejte cislo uctu klienta: 123984
Zadejte pocatecni zustatek klienta: 3000
Chcete-li Brass Account, zadejte 1, chcete-li Brass Plus Account,
zadejte 2: 2
Zadejte jmeno klienta: Lily Goldleaf
Zadejte cislo uctu klienta: 829302
Zadejte pocatecni zustatek klienta: 4000
Chcete-li Brass Account, zadejte 1, chcete-li Brass Plus Account,
zadejte 2: 1
Zadejte jmeno klienta: Harry Grub
```

```
Zadejte cislo uctu klienta: 111223
Zadejte pocatecni zustatek klienta: 22480
Chcete-li Brass Account, zadejte 1, chcete-li Brass Plus Account,
zadejte 2: 1
Klient: Rufus Overbeam
Cislo uctu: 123984
Zustatek: $3000.00

Klient: Lily Goldleaf
Cislo uctu: 829302
Zustatek: $4000.00

Klient: Harry Grub
Cislo uctu: 111223
Zustatek: $22480.00
```

Všimněte si, že i když Rufus Overhead má účet typu Brass Plus a je reprezentován objektem třídy `Overdraft`, použití ukazatele třídy `BankAccount` způsobí, že je použita metoda `BankAccount::ViewAcct()`. Tato metoda odpovídá typu ukazatele, platí statická vazba.

Dále je zde běh programu se stejnými vstupními daty, ale s použitím virtuální verze hlavičkového souboru (výpis 12.7).

```
Zadejte jmeno klienta: Rufus Overbeam
Zadejte cislo uctu klienta: 123984
Zadejte pocatecni zustatek klienta: 3000
Chcete-li Brass Account, zadejte 1, chcete-li Brass Plus Account,
zadejte 2: 2
Zadejte jmeno klienta: Lily Goldleaf
Zadejte cislo uctu klienta: 829302
Zadejte pocatecni zustatek klienta: 4000
Chcete-li Brass Account, zadejte 1, chcete-li Brass Plus Account,
zadejte 2: 1
Zadejte jmeno klienta: Harry Grub
Zadejte cislo uctu klienta: 111223
Zadejte pocatecni zustatek klienta: 22480
Chcete-li Brass Account, zadejte 1, chcete-li Brass Plus Account,
zadejte 2: 1
Klient: Rufus Overbeam
Cislo uctu: 123984
Zustatek: $3000.00
Maximum loan: $500.00
Owed to bank: $0.00
Klient: Lily Goldleaf
Cislo uctu: 829302
Zustatek: $4000.00
Klient: Harry Grub
Cislo uctu: 111223
Zustatek: $22480.00
```

Opět má Rufus Overhead účet typu Brass Plus a je reprezentován objektem třídy `Overdraft`, tentokrát ale program použije metodu `BankAccount::ViewAcct()`. Metoda odpovídá typu objektu; platí dynamická vazba.

Protože funkce `ViewAcct()` je virtuální, může příkaz

```
baps[i]->ViewAcct();
```

vyvolat někdy funkci `Overdraft::ViewAcct()` a jindy funkci `BankAccount::ViewAcct()`. Stejně jako u přetěžovacích funkcí, i zde se jedná o příklad polymorfismu. V závislosti na kontextu se může stejný kód odkazovat na různé třídy a různé funkce.

Proč dva druhy vazeb

Jestliže pozdní vazba umožňuje předefinovat metody třídy, zatímco statická vazba do toho jen částečně fušuje, proč mít vůbec statickou vazbu? Důvody jsou dva: efektivnost a koncepční model.

Nejdříve se podíváme na efektivnost. Aby program mohl provést rozhodnutí za běhu, musí nějak udržovat zaznamenávání typu objektů, na něž ukazatel či reference odkazují, a to vyžaduje určitou práci navíc. (Jednu metodu dynamické vazby popíšeme později.) Jestliže například navrhnete třídu, která nebude používána jako základní třída pro dědění, nepotřebujete pozdní vazbu. V takovém případě je logičtější použít vazbu statickou a získat trochu výkonu navíc. Z důvodu větší efektivity je statická vazba v C++ implicitní volbou. Stroustrup říká, že jednou z hlavních zásad jazyka C++ je neplatit (spotřebou paměti či dobou zpracování) za ty vlastnosti, které nepotřebujete. Použijte virtuální metody pouze tehdy, jestliže to vyžaduje návrh programu.

Dále vezměme v úvahu koncepční model. Když navrhujete třídu, mohou v ní být členské funkce, jejichž předefinování v odvozených třídách si nepřejete. Například funkci `BankAccount::Balance()` vracející zůstatek na účtu asi není vhodné předefinovat. Tím, že není virtuální, dosáhnete dvou věcí. Za prvé bude efektivnější. A za druhé oznamujete svůj záměr tuto metodu nepředefinovat. Z toho vychází následující čistě praktická zásada.

Tip

Jestliže bude metoda základní třídy v odvozené třídě předefinována, učiňte ji virtuální metodou. Pokud nemá být předefinována, vytvořte ji jako nevirtuální.

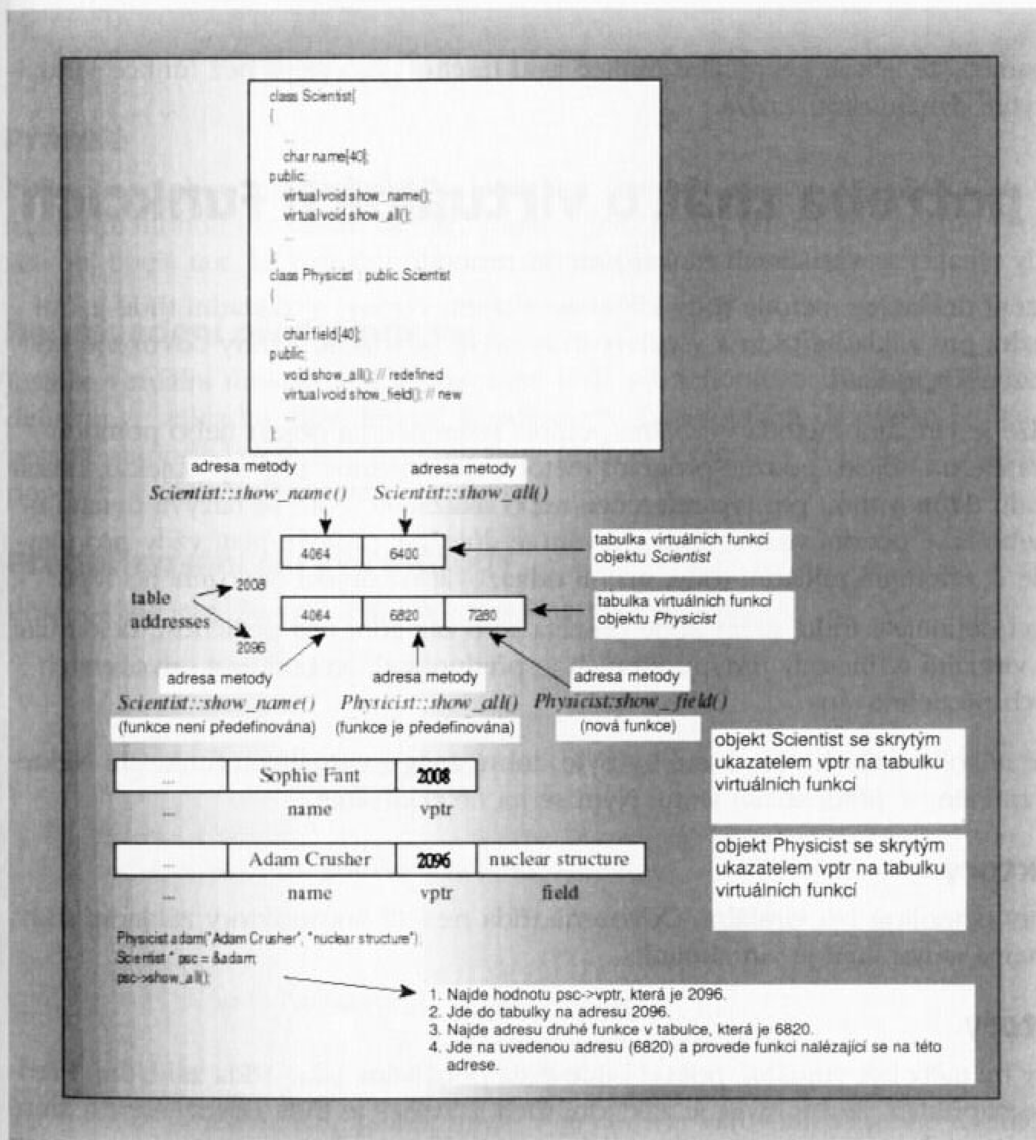
Samozřejmě při návrhu třídy není vždy zřejmé, do které kategorie daná metoda spadá. Podobně jako mnoho aspektů ve skutečném životě, ani návrh třídy není lineární proces.

Jak virtuální funkce pracují

C++ určuje, jak by se virtuální funkce měly chovat, implementaci však ponechává na autorovi kompilátoru. Virtuální funkce můžete používat, aniž byste znali metodu implementace, ale její poznání vám může pomoci konceptům lépe porozumět. Takže se na ni podívejme.

Kompilátory obvykle pracují s virtuálními funkcemi tak, že do každého objektu přidají skrytou položku. Skrytá položka obsahuje ukazatel na pole adres funkcí. Takové pole se většinou nazývá *tabulka*. Tabulka obsahuje adresy virtuálních funkcí, deklarovaných pro

objekty této třídy. Objekt základní třídy bude například obsahovat ukazatel na tabulku adres všech virtuálních funkcí této třídy. Objekt odvozené třídy bude obsahovat ukazatel na samostatnou tabulku adres. Jestliže odvozená třída nově definuje virtuální funkci, bude tabulka obsahovat adresu této nové funkce. Pokud odvozená třída virtuální funkci nepředefinuje, bude tabulka obsahovat adresu původní verze funkce. Jestliže odvozená třída definuje novou virtuální funkci, bude adresa této funkce přidána do tabulky (viz obrázek 12.5). Všimněte si, že ať pro třídu definujete jednu nebo deset virtuálních funkcí, přidáte objektu jen jednu položku s adresou; to, co se mění, je velikost tabulky.



Obrázek 12.5. Mechanismus virtuálních funkcí

Když zavoláte virtuální funkci, program se podívá na adresu tabulky uloženou v objektu a přejde do tabulky odpovídající adresám funkcí. Jestliže použijete první virtuální funkci z deklarace třídy, program použije první adresu funkce v poli a provede funkci s touto

adresou. Pokud použijete třetí virtuální funkci z deklarace třídy, program použije funkci, jejíž adresa je ve třetím prvku pole.

Stručně řečeno, používání virtuálních funkcí má následující skromné nároky na paměť a rychlost provedení:

- Velikost každého objektu se zvětší o velikost potřebnou pro uložení adresy.
- Pro každou třídu kompilátor vytvoří tabulku (pole) adres virtuálních funkcí.
- Každé volání funkce potřebuje jeden krok navíc. Tím je vyhledání adresy v tabulce.

Mějte na paměti, že ačkoli nevirtuální funkce jsou trochu výkonnější než funkce virtuální, neposkytují dynamickou vazbu.

Co je potřeba znát o virtuálních funkcích

Hlavní body týkající se virtuálních funkcí jsme již probrali:

- Uvození deklarace metody třídy klíčovým slovem `virtual` v základní třídě ji činí virtuální pro základní třídu a všechny třídy od ní odvozené a třídy odvozené od odvozených tříd atd.
- Jestliže je virtuální metoda vyvolána pomocí reference na objekt nebo pomocí ukazatele na objekt, použije program metodu definovanou pro typ objektu, nikoli metodu definovanou pro typ reference nebo ukazatele. Toto se nazývá dynamická nebo také pozdní vazba. Toto chování je důležité, protože platí vždy pro ukazatele či referenci základní třídy, mají-li odkazovat na objekt odvozeného typu.
- Pokud definujete třídu, která bude použita jako základní pro dědičnost, deklarujte jako virtuální ty metody třídy, u kterých je předpoklad, že budou v odvozených třídách předefinovány.

Je zde ještě několik dalších věcí, které by bylo dobré znát o virtuálních funkcích. Některé byly již zmíněny v předchozím textu. Nyní se na ně podíváme.

Konstruktory

Konstruktory nemohou být virtuální. Odvozená třída nedědí konstruktory základní třídy, takže ani nemá smysl činit je virtuálními.

Destruktory

Destruktory by měly být virtuální, pokud bude třída používána jako třída základní. Předpokládejme například, že `Employee` je základní třída a `Singer` je třída odvozená, do které je přidán ukazatel typu `char *`, ukazující na paměť vytvořenou pomocí operátoru `new`. Když potom platnost objektu `Singer` skončí, je důležité, aby byl zavolán destruktor `~Singer()`, který tuto paměť uvolní. Nyní uvažujte následující kód:

```
Employee * pe = new Singer;   // možné, protože Employee je základní
                              // třídou třídy Singer
...
delete pe;                    // ~Employee() nebo ~Singer()?
```

Jestliže se použije implicitní statická vazba, vyvolá příkaz delete destruktor `~Employee()`. Ten uvolní paměť, na kterou ukazují složky objektu `Singer` z části `Employee`, ale neuvolní paměť, na kterou ukazují nové položky třídy. Pokud jsou však destruktory virtuální, vyvolá stejný kód destruktor `~Singer()`, který uvolní paměť, na kterou ukazuje složka Singer a potom zavolá destruktor `~Employee()`, který uvolní paměť, na kterou ukazuje složka `Employee`.

Z toho vyplývá, že i když základní třída nevyžaduje služby explicitního destruktoru, neměli byste se spoléhat na implicitní destruktor a raději vytvořit virtuální destruktor, třeba i prázdný:

```
virtual ~BaseClass() { }
```

Přátelé

Přátelé nemohou být virtuálními funkcemi, protože nejsou členy třídy a virtuálními funkcemi mohou být pouze členské funkce. Jestliže tím vznikne při návrhu problém, můžete jej obejít tak, že virtuální funkce umístíte do spřátelené funkce.

Neprovedení předefinování

Jestliže virtuální funkce není v odvozené třídě předefinována, třída použije verzi funkce definice ze základní třídy. Pokud je odvozená třída součástí dlouhého řetězu odvození, použije naposledy předefinovanou verzi funkce. Výjimkou jsou skryté základní funkce popsané v následujícím textu.

Předefinování skrývá metody

Předpokládejme, že vytvoříte kód podobný následujícímu:

```
class Dwelling1
{
public:
      virtual void showperks(int a) const;
...
};
class Hovel : public Dwelling
{
{
public:
      void showperks();
...
};
```

To způsobí problém. Kompilátor může vygenerovat varování podobné tomuto:

```
Warning: Hovel::showperks(void) hides Dwelling::showperks(int)
```

Nebo možná varování ani nedostanete. Ale v obou případech má uvedený kód tyto důsledky:

```
Hovel trump;
trump.showperks();      // platné
trump.showperks(5);     // neplatné
```

V nové definici je funkce showperks definována bez parametrů. Místo vytvoření dvou přetížených verzí funkce skryje toto předefinování verzi základní třídy, ve které funkce přijímá parametr typu int. Stručně řečeno, předefinování zděděných metod není variantou přetížení. Jestliže předefinujete funkci v odvozené třídě, nedojde k přepsání deklarace základní třídy stejnou signaturou funkce, ale ke skrytí všech metod základní třídy se stejným názvem bez ohledu na signatury parametrů.

Z životní skutečnosti vyplývá několik praktických pravidel. Za prvé, pokud předefinujete zděděnou metodu, ujistěte se, že přesně odpovídá původnímu prototypu. Jedinou výjimkou je návratová hodnota typu reference nebo ukazatel na základní třídu, kterou lze nahradit referencí nebo ukazatelem na odvozenou třídu. (Tato výjimka je nová a ne všechny kompilátory ji rozeznají. A také pamatujte, že platí pouze pro návratové hodnoty, nikoli pro parametry.) Za druhé, pokud je deklarace základní třídy přetížena, předefinujte v odvozené třídě všechny verze ze základní třídy:

```
class Dwelling
{
public:
// tři přetížené metody showperks()
        virtual void showperks(int a) const;
        virtual void showperks(double x) const;
        virtual void showperks() const;
        ...
};
class Hovel : public Dwelling
{
public:
// tři předefinované metody showperks()
        void showperks(int a) const;
        void showperks(double x) const;
        void showperks() const;
        ...
};
```

Když předefinujete pouze jednu verzi, budou ostatní dvě skryté a objekty odvozené třídy je nebudou moci použít. Pokud není potřeba provést žádnou změnu, předefinovaná metoda může jednoduše zavolat verzi základní třídy.

Dědičnost a přiřazení

Jak jsme se již dříve zmínili, operátor přiřazení je jednou z členských funkcí, které nejsou děděny. Podívejme se na téma přiřazení a dědičnosti podrobněji. Nejdříve několik základních informací. Uvažujte následující kód:

```
BankAccount darf("Darfa Flemwit", 121234, 100);
BankAccount templ;
Overdraft bip("Bipp Fardbag", 212143, 200);
Overdraft temp2;
templ = darf;
temp2 = bip;
```

Přiřazení proměnným temp1 a temp2 jsou ekvivalentní následujícím voláním funkcí:

```
temp1.operator=(darf); // první volání
temp2.operator=(bip);  // druhé volání
```

Protože první volání je vyvoláno objektem třídy BankAccount, bude kompilátor hledat funkci BankAccount::operator=(). Jelikož i parametr je typu BankAccount, bude kompilátor hledat funkci s tímto parametrem. Oba požadavky jsou splněny díky implicitnímu operátoru přiřazení, který má tento prototyp:

```
BankAccount & BankAccount::operator=(const BankAccount &);
```

Podobně druhému volání odpovídá následující prototyp:

```
Overdraft & Overdraft::operator=(const Overdraft &);
```

Opět se jedná o funkci operator=(), která je pro třídu Overdraft generována implicitně. Má správnou signaturu pro přiřazení jednoho objektu třídy Overdraft druhému. Kdyby však třída Overdraft zdědila funkci operator=() z třídy BankAccount, byla by z hlediska práce s objekty třídy Overdraft signatura zděděné verze chybná, protože jejím parametrem je reference na objekt třídy BankAccount. Kompilátor tedy neumožní odvozené třídě operátor přiřazení zdědit, ale automaticky pro ni definuje nový.

Smíšené přiřazení

Pomocí operátorů přiřazení tedy můžete přiřadit objekt třídy BankAccount jinému objektu stejné třídy a jeden objekt třídy Overdraft jinému objektu této třídy. Ale co přiřazení objektu třídy BankAccount objektu třídy Overdraft a obráceně? Nejdříve se podívejme na přiřazení objektu odvozené třídy objektu třídy základní:

```
BankAccount temp;
Overdraft bip("Bipp Fardbag", 212143, 200);
temp = bip; // možné?
```

Odpovědí zde je, že takové přiřazení funguje, i když poněkud nepřímo. Kompilátor nejdříve převede tento příkaz do funkčního tvaru:

```
temp.operator=(bip);
```

V dalším kroku se kompilátor pokusí najít operátor přiřazení odpovídající funkčnímu volání. Protože volající objekt je třídy BankAccount a parametrem je objekt třídy Overdraft, bylo by ideální najít následující prototyp funkce:

```
BankAccount & BankAccount::operator=(const Overdraft &);
```

Všimněte si, že funkce musí být členem třídy BankAccount. Nejedná se o implicitní operátor přiřazení, není ani deklarována ve třídě BankAccount, žádný přesný prototyp tedy neexistuje. Kompilátor tedy zjišťuje, jestli existuje funkce operátoru, která bude fungovat po konverzi typu parametru. A taková funkce existuje:

```
BankAccount & BankAccount::operator=(const BankAccount &);
```

Vzpomeňte si, že reference na objekt základní třídy může odkazovat na objekt odvozené třídy (přetypování na předka) bez explicitního přetypování. Implicitní operátor přiřazení

základní třídy tedy přijme objekt odvozené třídy. Výsledkem je, že se zkopíruje pouze ta část objektu odvozené třídy, která patří třídě základní.

Pamatujte

Objekt odvozené třídy můžete přiřadit objektu třídy základní. Kompilátor použije operátor přiřazení základní třídy a zkopíruje pouze část, náležející základní třídě.

Dále zkusme přiřadit objekt základní třídy objektu odvozené třídy:

```
BankAccount darf("Darfa Flemwit", 121234, 100);
Overdraft temp;
temp = darf;  // možné?
```

Tentokrát zní odpověď „možná". Tento přiřazovací příkaz se přeloží do následujícího volání:

```
temp.operator=(darf);
```

Protože volajícím objektem je objekt třídy `Overdraft`, musí být odpovídající funkce přiřazení, pokud existuje, členem třídy `Overdraft` s parametrem třídy `BankAccount`:

```
Overdraft & Overdraft::operator=(const BankAccount &);
```

Taková funkce neexistuje, ale existuje implicitní operátor přiřazení:

```
Overdraft & Overdraft::operator=(const Overdraft &);
```

Lze ho použít? To by vyžadovalo, aby reference na objekt třídy `Overdraft` odkazovala na objekt třídy `BankAccount`. Reference na objekt odvozené třídy by odkazovala na objekt třídy základní. Běžně je takové přetypování na potomka povoleno jen u explicitního přetypování, takže normálně takové přiřazení povoleno není. V tomto případě však existuje konstruktor s jedním parametrem provádějící implicitní konverzní funkci z objektu třídy `BankAccount` na objekt třídy `Overdraft`:

```
Overdraft(const BankAccount & ba, double ml = 500, double r = 0.1);
```

V tomto konkrétním případě tedy program zavolá konstruktor `Overdraft(darf)` a vygeneruje dočasný objekt třídy `Overdraft`, který se použije jako parametr implicitní funkce `Overdraft operator=()`. Pokud by byl konstruktor deklarován jako explicitní, přiřazení by povoleno nebylo.

Pamatujte

Obecně není možné přiřadit objekt základní třídy objektu třídy odvozené. Jestliže však existuje konstruktor, v němž je definována konverze základní třídy na třídu odvozenou, bude takové přiřazení fungovat.

Přiřazení a dynamické přidělení paměti

Nyní se podíváme, jaký je vzájemný vztah dynamicky přidělené paměti s přiřazením a dědičností. Nejdříve upravíme třídu `BankAccount` tak, aby používala dynamicky přidělenou paměť. Výpis 12.9 obsahuje novou deklaraci třídy a ve výpisu 12.10 je její nová implementace. Jako obvykle přidání dynamicky přidělené paměti vyžaduje přidání explicitního destruktoru, kopírovacího konstruktoru a operátoru přiřazení.

Výpis 12.9. bankdyn.h

```
// bankdyn.h - jednoduchá třída BankAccountD s DAP
#ifndef _BANKACCT_H_
#define _BANKACCT_H_
class BankAccountD
{
private:
    char * fullName;
    long acctNum;
    double balance;
public:
    BankAccountD(const char *s = "Nullbody", long an = -1,
                    double bal = 0.0);
    BankAccountD(const BankAccountD & ba);
    virtual ~BankAccountD();
    void Deposit(double amt);
    virtual void Withdraw(double amt);  // virtuální metoda
    double Balance() const;
    virtual void ViewAcct() const;        // virtuální metoda
    BankAccountD & operator=(const BankAccountD & ba);
};
#endif
```

Výpis 12.10. bankdyn.cpp

```
// bankdyn.cpp - metody třídy BankAccountD
#include <iostream>
using namespace std;
#include "bankdyn.h"
#include <cstring>
BankAccountD::BankAccountD(const char *s, long an, double bal)
{
    fullName = new char[strlen(s) + 1];
    strcpy(fullName, s);
    acctNum = an;
    balance = bal;
}
BankAccountD::BankAccountD(const BankAccountD & ba)
{
    fullName = new char[strlen(ba.fullName) + 1];
```

```
    strcpy(fullName, ba.fullName);
    acctNum = ba.acctNum;
    balance = ba.balance;;
}
BankAccountD::~BankAccountD()
{
    delete [] fullName;
}
void BankAccountD::Deposit(double amt)
{
    balance += amt;
}
void BankAccountD::Withdraw(double amt)
{
    if (amt <= balance)
        balance -= amt;
    else
        cout << "Pozadovana castka $" << amt
             << " prevysuje zustatek.\n"
             << "Vyber stornovan.\n";
}
double BankAccountD::Balance() const
{
    return balance;
}
void BankAccountD::ViewAcct() const
{
    // nastaví formát ###.##
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);
    cout << "Klient: " << fullName << endl;
    cout << "Cislo uctu: " << acctNum << endl;
    cout << "Zustatek: $" << balance << endl;
    cout.setf(initialState); // obnoví původní formát
}
BankAccountD & BankAccountD::operator=(const BankAccountD & ba)
{
    if (this == &ba)
        return *this;
    delete [] fullName;
    fullName = new char[strlen(ba.fullName) + 1];
    strcpy(fullName, ba.fullName);
    acctNum = ba.acctNum;
    balance = ba.balance;
    return *this;
}
```

Případ první – odvozená třída nepoužívá operátor new

Předpokládejme, že odvodíte třídu Overdraft od třídy BankAccountD a ne od třídy BankAccount. Je nutné definovat explicitní destruktor, kopírovací konstruktor a operátor přiřazení pro třídu Overdraft? Za předpokladu, že třídě Overdraft nepřidáte žádné vlastnosti vyžadující tyto metody, pak odpověď zní ne.

Nejdříve uvažujte o potřebě destruktoru. Jestliže žádný nenadefinujete, kompilátor nadefinuje implicitní destruktor, který nic nedělá. Destruktor odvozené třídy ve skutečnosti vždy něco dělá; po provedení vlastního kódu volá destruktor základní třídy. Protože položky třídy Overdraft nevyžadují žádnou speciální činnost, implicitní destruktor stačí.

Dále uvažujte kopírovací konstruktor. Viděli jste, že implicitní kopírovací konstruktor provádí kopii po položkách, což se při použití dynamické paměti nehodí. Kopírování po položkách však vyhovuje třem novým položkám třídy Overdraft. Zbývá tedy pouze zděděný objekt třídy BankAccountD. Potřebujete vědět, že kopírování po položkách používá takový způsob kopírování, který je pro každý typ položky definován. Takže kopírování položky typu long do jiné položky typu long se děje pomocí obyčejného přiřazení. Ale kopírování položky třídy nebo položky zděděné třídy se provádí pomocí kopírovacího konstruktoru dané třídy. Implicitní kopírovací konstruktor třídy Overdraft tedy zkopíruje část BankAccountD objektu Overdraft pomocí explicitního kopírovacího konstruktoru třídy BankAccountD. Jestliže tedy implicitní kopírovací konstruktor vyhovuje novým položkám třídy Overdraft, vyhovuje také zděděnému objektu BankAccountD.

V podstatě stejná je i situace s operátorem přiřazení. Implicitní operátor přiřazení třídy automaticky použije operátor přiřazení základní třídy pro položky zděděné ze základní třídy. Takže je opět vše v pořádku.

Případ druhý – odvozená třída používá operátor new

Předpokládejme však, že odvozená třída používá operátor new. V tom případě musíte pro danou třídu definovat explicitní destruktor, kopírovací konstruktor a operátor přiřazení. Podívejme se, jak se to udělá. Předpokládejme, že bylo rozhodnuto, že majitelé účtu Brass Plus dostanou kódové jméno, takže bude potřeba přidat do třídy nový ukazatel. Výpis 12.11. obsahuje upravenou deklaraci třídy s vyznačením nových položek.

Výpis 12.11. overdyn2.h

```
// overdyn2.h - deklarace třídy OverdraftD s DAP
#ifndef _OVERDRFT_H_
#define _OVERDRFT_H_
#include "bankdyn.h"
class OverdraftD : public BankAccountD
{
private:
     double maxLoan;
     double rate;
     double owesBank;
     char * codeName;                    // nové
public:
     OverdraftD(const char * s = "Nullbody", const char *cn =
```

```
                    "cent", long an = -1, double bal = 0.0, double
                    ml = 500, double r = 0.10);
        OverdraftD(const BankAccountD & ba,const char * cn =
                    "cent", double ml = 500, double r = 0.1);
        OverdraftD(const OverdraftD & od);                   // nové
        ~OverdraftD();                                        // nové
        void ViewAcct()const;
        void Withdraw(double amt);
        void ResetMax(double m) { maxLoan = m; }
        void ResetRate(double r) { rate = r; };
        void ResetOwes() { owesBank = 0; }
        OverdraftD & operator=(const OverdraftD & od);       // nové
};
#endif
```

Aby bylo možné přidělit místo řetězci, musí být původní konstruktory obvyklým způsobem upraveny a do funkce `ViewAcct()` bude potřeba přidat kód pro zobrazení nové položky. Tyto detaily najdete ve výpisu 12.12. Zatím se soustředíme na nově přidané členské funkce.

Nejdříve kopírovací konstruktor. Lze ho napsat takto:

```
OverdraftD::OverdraftD(const OverdraftD & od) : BankAccountD(od)
{
    codeName = new char[strlen(od.codeName) + 1];
    strcpy(codeName, od.codeName);
    maxLoan = od.maxLoan;
    owesBank = od.owesBank;
    rate = od.rate;
}
```

Zodpovědnost za vytvoření části objektu `OverdraftD` leží na konstruktoru třídy `OverdraftD`. Tělo funkce přidělí paměť pro řetězec `codeName` obvyklým způsobem. Za vytvoření části objektu patřící základní třídě nesou zodpovědnost konstruktory základní třídy. V tomto případě metoda pro vyvolání kopírovacího konstruktoru základní třídy používá syntaxi seznamu inicializátorů. Všimněte si, že kopírovací konstruktor třídy `OverdraftD` předává kopírovacímu konstruktoru `BankAccountD` referenci na objekt třídy `OverdraftD`. Parametr kopírovacího konstruktoru základní třídy je deklarován jako typ `const BankAccountD &`. Protože však platí pravidlo implicitního přetypování na předka, může reference na objekt třídy `BankAccount` odkazovat na objekt třídy `OverdraftD`. Konkrétně bude odkazovat na část objektu třídy `Overdraft` patřící třídě `BankAccountD`.

Dále je zde destruktor. Ten musí spravovat pouze část třídy `OverdraftD`.

```
OverdraftD::~OverdraftD()
{
    delete [] codeName;
}
```

Pamatujte, že po jeho zavolání je automaticky vyvolán destruktor základní třídy, který uvolní paměť, na kterou ukazuje ukazatel `fullName`.

A nakonec je zde kopírovací konstruktor. Důležitým bodem, který byste měli mít na paměti je, že jestliže definujete operátor přiřazení pro odvozenou třídu, musí tento operátor zajistit, aby přiřazení fungovalo jak v části patřící odvozené třídě, tak i v části patřící třídě základní. Navíc pro část patřící základní třídě nemůžete použít syntaxi seznamu inicializátorů, neboť tu lze použít pouze u konstruktorů. Musíte tedy v těle funkce vyvolat operátor přiřazení základní třídy. K tomuto účelu se nejsnáze hodí zápis funkce pro operátor přiřazení:

```
OverdraftD & OverdraftD::operator=(const OverdraftD & od)
{
    if (this == &od)
        return *this;
    BankAccountD::operator=(od); // přiřazení části, patřící základní
                                 // třídě
    delete [] codeName;
    codeName = new char[strlen(od.codeName) + 1];
    strcpy(codeName, od.codeName);
    maxLoan = od.maxLoan;
    owesBank = od.owesBank;
    rate = od.rate;
    return *this;
}
```

Příkaz

```
BankAccountD::operator=(od);  // přiřazení části, patřící základní třídě
```

je zkratkou pro zápis

```
this->BankAccountD::operator=(od);// přiřazení části, patřící základní třídě
```

Jinými slovy, pro přiřazení parametru od objektu `*this` použijte verzi přiřazení třídy `BankAccountD`. Tímto způsobem zkopírujete část objektu `od`, náležející třídě `BankAccountD`, do části objektu `*this` náležejícího třídě `BankAccountD`. Úplnou implementaci obsahuje výpis 12.12.

Výpis 12.12. overdyn2.cpp

```
// overdyn2.cpp – metody třídy OverdraftD s DAP
#include <iostream>
using namespace std;
#include "overdyn2.h"
OverdraftD::OverdraftD(const char * s, const char * cn,
            long an, double bal,
            double ml, double r) : BankAccountD(s, an, bal)
{
    codeName = new char[strlen(cn) + 1];
    strcpy(codeName, cn);
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
```

```
OverdraftD::OverdraftD(const BankAccountD & ba, const char
            * cn, double ml, double r)
           : BankAccountD(ba)  // použije explicitní kopírovací
                               // konstruktor
{
    codeName = new char[strlen(cn) + 1];
    strcpy(codeName, cn);
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
OverdraftD::OverdraftD(const OverdraftD & od) : BankAccountD(od)
{
    codeName = new char[strlen(od.codeName) + 1];
    strcpy(codeName, od.codeName);
    maxLoan = od.maxLoan;
    owesBank = od.owesBank;
    rate = od.rate;
}
OverdraftD::~OverdraftD()
{
    delete [] codeName;
}
// předefinování funkce  ViewAcct()
void OverdraftD::ViewAcct() const
{
    // nastaví formát ###.##
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);
    BankAccountD::ViewAcct();  // zobrazí část, patřící základní třídě
    cout << "Nazev kodu: " << codeName << endl;
    cout << "Maximalni vyse pujcky: $" << maxLoan << endl;
    cout << "Dluzna castka: $" << owesBank << endl;
    cout.setf(initialState);
}
// předefinování funkce Withdraw()
void OverdraftD::Withdraw(double amt)
{
    // nastavení formátu ###.##
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);
    double bal = Balance();
    if (amt <= bal)
        BankAccountD::Withdraw(amt);
    else if ( amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
```

```
            owesBank += advance * (1.0 + rate);
            cout << "Bankovni pujcka: $" << advance << endl;
            cout << "Financni poplatek: $" << advance * rate << endl;
            Deposit(advance);
            BankAccountD::Withdraw(amt);
        }
        else
            cout << "Kreditni limit prekrocen. Transakce stornovana.\n";
        cout.setf(initialState);
    }
    OverdraftD & OverdraftD::operator=(const OverdraftD & od)
    {
        if (this == &od)
            return *this;
        BankAccountD::operator=(od);        // přiřazení části, patřící
                                            // základní třídě
        delete [] codeName;
        codeName = new char[strlen(od.codeName) + 1];
        strcpy(codeName, od.codeName);
        maxLoan = od.maxLoan;
        owesBank = od.owesBank;
        rate = od.rate;
        return *this;
    }
```

Výpis 12.13, který by měl být zkompilován spolu s výpisem 12.10 a 12.12, obsahuje program testující upravené třídy.

Výpis 12.13. usedyn2.cpp

```
// usedyn2.cpp - test třídy OverdraftD
// zkompilovat spolu s bankdyn.cpp a overdyn2.cpp
#include <iostream>
using namespace std;
#include "overdyn2.h"
int main()
{
    BankAccountD dolly("Dahlia Dahl", 453216, 6000);
    BankAccountD temp;
    temp = dolly;
    temp.ViewAcct();
    cout << endl;
    OverdraftD rolly("Roland Rayleigh", "Rocky", 223391, 8000);
    OverdraftD dup;
    dup = rolly;
    dup.ViewAcct();
    cout << endl;
    dup = dolly;
    dup.ViewAcct();
    cout << "Nashledanou!\n";
    return 0;
```

```
}
```

Zde je jeho výstup:

```
Klient: Dahlia Dahl
Cislo uctu: 453216
Zustatek: $6000.00
Klient: Roland Rayleigh
Cislo uctu: 223391
Zustatek: $8000.00
Nazev kodu: Rocky
Maximalni vyse pujcky: $500.00
Dluzna castka: $0.00
Klient: Dahlia Dahl
Cislo uctu: 453216
Zustatek: $6000.00
Nazev kodu: cent
Maximalni vyse pujcky: $500.00
Dluzna castka: $0.00
Nashledanou!
```

Jak vidíte, přiřazení fungovalo pro obě třídy, fungovalo dokonce přiřazení mezi těmito dvěma třídami, a to díky konstruktoru třídy `OverdraftD`, který se chová jako implicitní konverzní funkce z třídy `BankAccountD` na třídu `OverdraftD`. Všimněte si, že přiřazení objektu dolly objektu dup způsobilo, že položky objektu dup náležející třídě `OverdraftD` získaly implicitní hodnoty, což je přesně to, co kopírovací konstruktor činí:

```
OverdraftD(const BankAccountD & ba,const char * cn = "cent",
                double ml = 500, double r = 0.1);
```

Abstraktní základní třídy

Někdy nemusí být pravidla pro vztah *je* tak jednoduchá, jak by se mohlo zdát. Předpokládejme například, že vytváříte grafický program, který má mimo jiné zobrazovat kružnice a elipsy. Kružnice je speciální případ elipsy; její vedlejší a hlavní osa mají stejnou délku. Všechny kružnice jsou tedy elipsami a to svádí k odvození třídy `Circle` od třídy `Ellipse`. Ale při řešení některých detailů můžete narazit na problémy.

Abychom si to ukázali, uvažujte nejdříve, co by mohlo být součástí třídy `Ellipse`. Datové položky by mohly obsahovat souřadnice středu elipsy a délky obou poloos a orientaci úhlu, který svírá hlavní osa elipsy s horizontální osou souřadnicového systému. Třída by také mohla obsahovat metody pro posunutí elipsy, vrácení obsahu elipsy, otočení elipsy a změnu velikostí jednotlivých os:

```
class Ellipse
{
private:
    double x;      // x-ová souřadnice středu elipsy
    double y;      // y-ová souřadnice středu elipsy
    double a;      // hlavní poloosa
    double b;      // vedlejší poloosa
```

```
    double angle; // směrový úhel ve stupních
    ...
public:
    ...
    void Move(int nx, ny) { x = nx; y = ny; }
    virtual double Area() const { return 3.14159 * a * b; }
    virtual void Rotate(double nang) { angle = nang; }
    virtual void Scale(double sa, double sb)
        { a *= sa; b *= sb; }
    ...
};
```

Nyní předpokládejme odvození třídy Circle:

```
class Circle : public Ellipse
{
    ...
};
```

Ačkoli je kružnice elipsou, je odvození trochu nemotorné. Kružnice například potřebuje pouze jednu hodnotu, průměr, k vyjádření své velikosti a tvaru namísto hlavní (`a`) a vedlejší (`b`) poloosy. Konstruktory třídy `Circle` by se o to mohly postarat přiřazením stejné hodnoty položkám `a` a `b`, ale v tom případě budete mít zbytečnou reprezentaci stejné informace. Parametr `angle` a metoda `Rotate()` nemají u kružnice žádný smysl a metoda `Scale()`, tak jak je napsaná, může změnou přiřazení různých hodnot jednotlivým osám z kružnice vytvořit elipsu. Tyto věci můžete napravit pomocí různých triků, například začleněním předefinované metody `Rotate()` do soukromé části třídy `Circle`, což znemožní použít ji veřejně pro kruh, ale celkově je mnohem jednodušší definovat třídu `Circle` bez použití dědičnosti:

```
class Circle    // bez dědičnosti
{
private:
    double x;   // x-ová souřadnice středu elipsy
    double y;   // y-ová souřadnice středu elipsy
    double r;   // průměr
    ...
public:
    ...
    void Move(int nx, ny) { x = nx; y = ny; }
    double Area() const { return 3.14159 * r * r; }
    void Scale(double sr) { r *= sr; }
    ...
};
```

Nyní má třída pouze ty položky, které potřebuje. Ale ani toto řešení nevypadá přesvědčivě. Třídy Circle a Ellipse mají mnoho společného, ale separátní definice tento fakt ignoruje.

Existuje i jiné řešení. Spočívá v tom, že se z tříd `Ellipse` a `Circle` abstrahují společné vlastnosti a ty se vloží do *abstraktní základní třídy* (AZT). Od této abstraktní základní třídy následně odvodíte obě třídy `Circle` a `Ellipse`. Potom můžete například pomocí pole ukazatelů na základní třídu spravovat směsici objektů tříd `Circle` a `Ellipse` (to zname-

ná, použít polymorfní postup). V tomto případě mají obě třídy společné souřadnice středu, metodu `Move()`, která je pro obě třídy stejná, a metodu `Area()`, která funguje pro každou třídu odlišně. Metodu `Area()` nelze ani implementovat pro AZT, protože jí chybí potřebné datové položky. V C++ lze neimplementovanou funkci dodat pomocí *čistě virtuální funkce*. Čistě virtuální funkce má na konci své deklarace výraz =0:

```
class BaseEllipse // abstraktní základní třída
{
private:
    double x;  // x-ová souřadnice středu elipsy
    double y;  // y-ová souřadnice středu elipsy
    ...
private:
    BaseEllipse(double x0 = 0, double y0 = 0) : x(x0),y(y0) {}
    virtual ~BaseEllipse() {}
    void Move(int nx, ny) { x = nx; y = ny; }
    virtual Area() const = 0;  // čistě virtuální funkce
    ...
}
```

Jestliže třída obsahuje čistě virtuální funkci, pak nelze vytvořit objekt této třídy. Smyslem tříd s čistě virtuálními funkcemi je sloužit jako základní třídy. Aby třída byla opravdu abstraktní základní třídou, musí mít alespoň jednu čistě virtuální funkci.

Nyní můžete odvodit třídy `Ellipse` a `Circle` od třídy `BaseEllipse` a doplnit obě třídy o potřebné položky. Je potřeba připomenout, že třída `Circle` reprezentuje vždy kružnice, zatímco třída `Ellipse` reprezentuje elipsy, které mohou být také kružnicemi. Kružnici třídy `Ellipse` však lze změnit na elipsu, zatímco kružnice třídy `Circle` musí zůstat kružnicí.

Program používající tyto třídy bude moci vytvářet objekty tříd `Ellipse` a `Circle`, ale ne objekty třídy `BaseEllipse`. Protože objekty třídy `Ellipse` a `Circle` mají stejnou základní třídu, může být kolekce kružnic a elips spravována polem ukazatelů na třídu BaseEllipse.

Stručně řečeno, AZT popisuje rozhraní s alespoň jednou čistě virtuální funkcí a třídy odvozené od abstraktní základní třídy implementují pomocí běžných virtuálních funkcí rozhraní na základě vlastností konkrétní odvozené třídy.

Opakování návrhu tříd

Jazyk C++ lze použít na velké množství programovacích problémů a nemůžete návrh třídy redukovat na pevně danou posloupnost kroků. Existují však některá často používaná vodítka a nyní je vhodná chvíle projít si je, zopakovat a rozvést předcházející diskuse.

Členské funkce generované kompilátorem

Jak jsme si již řekli v kapitole 11, určité veřejné členské funkce kompilátor generuje automaticky. Tento fakt naznačuje, že se jedná o obzvlášť důležité funkce. Podívejme se na některé z nich znovu.

Implicitní konstruktor

Implicitní konstruktor je konstruktor bez parametrů neboli konstruktor, kde všechny parametry mají implicitní hodnoty. Jestliže nedefinujete žádný konstruktor, kompilátor definuje implicitní. Ten sice nic nedělá, ale musí existovat, abyste mohli provádět určitou činnost. Předpokládejme například, že `Star` je třída. Abyste mohli napsat následující výrazy, potřebujete implicitní konstruktor:

```
Star rigel;          // vytvoří objekt bez explicitní inicializace
Star pleiades[6];    // vytvoří pole objektů
```

Pokud napíšete konstruktor odvozené třídy a v seznamu inicializátorů nebude explicitně volat konstruktor základní třídy, použije kompilátor implicitní konstruktor základní třídy a s jeho pomocí vytvoří tu část nového objektu, která patří základní třídě.

Jestliže definujete jakýkoli konstruktor, kompilátor implicitní definovat nebude. V tom případě musíte v případě potřeby vytvořit implicitní konstruktor sami.

Jedním z důvodů existence konstruktorů je zajištění, aby objekty byly vždy správně inicializovány. Jestliže třída obsahuje ukazatele, měly by být určitě inicializovány. Má tedy význam dodat explicitně implicitní konstruktor inicializující všechny datové položky na smysluplné hodnoty.

Kopírovací konstruktor

Kopírovací konstruktor je konstruktor, který má jako parametr konstantní referenci na třídu stejného typu, jaký má parametr. Například kopírovací konstruktor třídy `Star` by měl tento prototyp:

```
Star(const Star &);
```

Kopírovací konstruktor třídy se použije v následujících situacích:

- Když je nový objekt inicializován pomocí objektu stejné třídy.
- Když je objekt předáván funkci hodnotou.
- Když funkce vrací objekt hodnotou.
- Když kompilátor generuje dočasný objekt.

Jestliže váš program nepoužívá kopírovací konstruktor (implicitně nebo explicitně), vytvoří kompilátor prototyp funkce, ale ne její definici. Jinak program definuje kopírovací konstruktor, který provádí kopírování po položkách. To znamená, že každá položka nového objektu je inicializována na hodnotu odpovídající položky z původního objektu.

V některých případech je kopírování po položkách nežádoucí. Například ukazatele inicializované pomocí operátoru `new` většinou vyžadují použití hluboké kopie, jako například

u třídy BankAccountD. Třída také může obsahovat statickou proměnnou, která musí být upravena. V takových případech musíte definovat vlastní kopírovací konstruktor.

Operátor přiřazení

Implicitní operátor přiřazení řeší přiřazení jednoho objektu jinému objektu stejné třídy. Nepleťte si přiřazení a inicializaci. Jestliže příkaz vytvoří nový objekt, použije inicializaci, pokud se ale změní hodnota existujícího objektu, jedná se o přiřazení:

```
Star sirius;
Star alpha = sirius;     // inicializace
Star dogstar;
dogstar = sirius;        // přiřazení
```

Jestliže musíte definovat kopírovací konstruktor explicitně, musíte ze stejných důvodů definovat explicitně operátor přiřazení. Prototyp operátoru přiřazení třídy Star by vypadal takto:

```
Star & Star::operator=(const Star &);
```

Všimněte si, že funkce operátoru přiřazení vrací referenci na objekt třídy Star. Třída BankAccountD představuje typický příklad explicitní funkce operátoru přiřazení.

Kompilátor negeneruje operátory přiřazení pro přiřazení jednoho typu objektu jinému. Předpokládejme, že chcete přiřadit řetězec objektu třídy Star. Jeden přístup spočívá v explicitní definici takového operátoru:

```
Star & Star::operator=(const char *) {...}
```

Druhý přístup je spolehnout se na konverzní funkci (viz následující část „Konverze"), která převede řetězec na objekt třídy Star, a potom použít funkci pro přiřazení objektů třídy Star. První přístup probíhá rychleji, ale vyžaduje více kódu. Použití konverzních funkcí může vést k neobvyklému chování kompilátoru.

Další metody třídy

Existuje ještě několik dalších bodů, na které byste měli při návrhu třídy pamatovat. Některé z nich jsou popsány v následujících částech.

Konstruktory

Konstruktory se od ostatních metod třídy liší tím, že vytváří nové objekty, zatímco ostatní metody jsou volány existujícími objekty.

Destruktory

Nezapomeňte definovat explicitní destruktor, který zruší veškerou paměť přidělenou operátorem new v konstruktorech třídy a postará se o jakékoli administrativní operace, které jsou potřeba pro zrušení objektu třídy. Jestliže bude třída sloužit jako základní třída, vytvořte destruktor jako virtuální.

Konverze

Každý konstruktor, který lze vyvolat právě s jedním parametrem, definuje konverzi z typu parametru na typ třídy. Uvažujte například následující prototypy konstruktorů třídy Star:

```
Star(const char *);                          // konvertuje char * na objekt
                                             // třídy Star
Star(const Spectral &, int members = 1);// konvertuje objekt třídy
                                             // Spectral na objekt třídy Star
```

Konverzní konstruktory se používají například při předání konvertibilního typu funkci, která má jako parametr definovánu třídu. Předpokládejme například následující příkazy:

```
Star north;
north = "polaris";
```

Druhý příkaz by vyvolal funkci `Star::operator=(const Star &)`, přičemž by pomocí konstruktoru `Star::Star(const char *)` vygeneroval objekt třídy `Star`, který se použije jako parametr funkce operátoru přiřazení. Předpokladem je, že jste nedefinovali operátor pro přiřazení ukazatele na `char` objektu třídy `Star`.

Pomocí klíčového slova `explicit` v prototypu jednoparametrového konstruktoru zamezíte implicitním konverzím, explicitní konverze jsou však možné:

```
class Star
{
...
public:
     explicit Star(const char *);
...
};
Star north;
north = "polaris";              // nepovoleno
north = Star ("polaris");       // povoleno
```

Chcete-li konverzi z objektu třídy na nějaký jiný typ, definujte konverzní funkci (kapitola 10). Konverzní funkce je členská funkce třídy bez parametrů a deklarovaného typu návratové hodnoty, která je nazvána podle typu, na který se bude převádět. Přestože nemá deklarován návratový typ, měla by vracet požadovanou převedenou hodnotu. Zde jsou nějaké příklady:

```
Star::Star double() {…}         // konverze objektu třídy star na typ double
Star::Star const char * () {…} // konverze na typ const char
```

Tyto funkce byste měli používat s rozvahou a pouze tehdy, pokud mají smysl. Také některé třídy s konverzními funkcemi mohou zvýšit pravděpodobnost nejednoznačného kódu. Předpokládejme například, že jste definovali konverzní funkci `double` pro třídu Vector z kapitoly 10 a předpokládejme následující kód:

```
vector ius(6.0, 0.0);
vector lux = ius + 20.2;     // nejednoznačné
```

Kompilátor by mohl objekt `ius` převést na typ `double` a použít sčítání těchto typů, nebo převést hodnotu 22.2 na objekt třídy `Vector` (pomocí jednoho z konstruktorů) a použít sčítání vektorů. Namísto toho kompilátor neudělá nic a informuje vás o nejednoznačnosti konstrukce.

Předávání objektu hodnotou a předávání reference

Obecně platí, že píšete-li funkci používající jako parametr objekt, měli byste předávat objekt spíše odkazem než hodnotou. Jedním z důvodů je efektivita. Předávání objektu hodnotou vyžaduje vytvoření dočasného objektu, což znamená volání kopírovacího konstruktoru a později volání destruktoru. Volání těchto funkcí zabere nějaký čas a kopírování velkého objektu může být mnohem pomalejší než předání reference. Jestliže funkce nebude objekt upravovat, deklarujte parametr jako konstantní referenci.

Dalším důvodem pro předávání objektů odkazem je to, že v případě dědičnosti používající virtuální funkce lze funkci, která má jako parametr definovanou referenci na objekt základní třídy, úspěšně používat u odvozených tříd, jak jste viděli dříve v této kapitole. Podívejte se také na pojednání o virtuálních metodách dále v této kapitole.

Vrácení objektu a vrácení reference

Některé metody třídy vracejí objekty. Pravděpodobně jste si již všimli, že některé tyto metody vracejí objekty přímo, zatímco jiné vracejí reference. V některých případech musí metoda vrátit objekt, ale pokud to není nutné, používejte místo toho referenci. Pojďme se na tento problém podívat blíže.

Za prvé, jediný rozdíl mezi vrácením objektu přímo a vrácením reference je v prototypu funkce a záhlaví:

```
Star nova1(const Star &);      // vrátí objekt třídy Star
Star & nova2(const Star &);    // vrátí referenci na objekt třídy Star
```

Dalším důvodem, proč raději vrátit referenci než objekt je to, že vrácení objektu vyžaduje vytvoření dočasné kopie vráceného objektu. To je kopie, kterou bude mít k dispozici volající program. Vrácení objektu tedy vyžaduje čas na vyvolání kopírovacího konstruktoru pro vytvoření kopie a vyvolání destruktoru, který ji zruší. Vrácení reference šetří čas i paměť. Přímé vrácení objektu je podobné předání parametru hodnotou: oba procesy vytvářejí dočasné kopie. Podobně i vrácení reference se podobá předání objektu odkazem: jak volající, tak i volaná funkce pracují se stejným objektem.

Vrátit referenci ale není vždy možné. Funkce by neměla vracet referenci na dočasný objekt vytvořený ve funkci, protože jakmile funkce skončí a objekt zmizí, nebude tato reference platná. V takovém případě by funkce měla vrátit objekt, čímž se vytvoří jeho kopie dostupná volajícímu programu.

Čistě praktická zásada říká, že pokud funkce vrací dočasný objekt v ní vytvořený, nemáte používat referenci. Následující metoda například vytvoří pomocí konstruktoru nový objekt a potom vrátí jeho kopii:

```
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}
```

Pokud funkce vrací objekt, který jí byl předán referencí nebo ukazatelem, vraťte ho odkazem. Následující funkce například vrací odkazem buď objekt, který ji vyvolal, nebo objekt předaný jako parametr:

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;            // objekt předaný jako parametr
    else
        return *this;        // volající objekt
}
```

Použití modifikátoru const

Buďte ostražití při možnosti použít modifikátor `const`. Jeho použitím garantujete, že metoda nezmění hodnotu svého parametru:

```
Star::Star(const char * s) {...}    // nezmění řetězec, na který ukazuje s
```

Pomocí modifikátoru const můžete garantovat, že metoda nezmění obsah objektu, který ji vyvolal:

```
void Star::show() const {...} // nezmění obsah volajícího objektu
```

Zde const značí const `Star * this`, kde `this` ukazuje na volající objekt.

Normálně může být funkce, vracející referenci, na levé straně přiřazovacího příkazu, což ve skutečnosti znamená, že můžete přiřadit hodnotu odkazovanému objektu. Pomocí modifikátoru `const` však můžete zajistit, aby hodnota vrácená referencí nebo ukazatelem nemohla být použita k úpravě dat v objektu:

```
const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;           // objekt předaný parametrem
    else
        return *this;       // volající objekt
}
```

Zde metoda vrátí referenci buď na objekt s nebo `this`. Protože `this` i s jsou deklarovány jako `const`, funkce je nemůže změnit, což znamená, že vrácená reference musí být také deklarována jako `const`.

Všimněte si, že jestliže je parametr funkce deklarován jako reference nebo ukazatel na const, může ho funkce předat jiné funkci pouze v tom případě, pokud i tato funkce zaručí jeho neměnnost.

Poznámky k veřejné dědičnosti

Přidání dědičnosti do programu přirozeně přináší více věcí, na které je potřeba myslet. Pojďme se na některé podívat.

Vztah je

Používejte vztah *je* jako vodítko. Jestliže vaše odvozená třída není nějakým konkrétním druhem základní třídy, pak veřejné odvození nepoužívejte. Neodvozujte například třídu `Brain` (mozek) od třídy `Programmer` (programátor). Pokud chcete vyjádřit víru, že programátor má mozek, použijte objekt třídy `Brain` jako položku objektu třídy `Programmer`.

V některých případech může být nejlepším postupem vytvoření abstraktní datové třídy, obsahující čistě virtuální funkce, a ostatní třídy odvozovat od této třídy.

Pamatujte, že vztah *je* vyjadřuje skutečnost, že ukazatel na objekt základní třídy může ukazovat na objekt odvozené třídy bez explicitního přetypování. Totéž se týká i odkazu na objekt základní třídy. Také si zapamatujte, že opačně to neplatí. Bez explicitního přetypování tedy nemůže ukazatel nebo reference na odvozenou třídu odkazovat na objekt třídy základní. V závislosti na deklaraci tříd může a nemusí mít takové explicitní přetypování (přetypování na potomka) smysl. (Můžete si opět prohlédnout obrázek 12.4.)

Co není součástí dědictví

Nedědí se konstruktory. Konstruktory odvozené třídy však pomocí seznamu inicializátorů volají konstruktory základních tříd, které vytvářejí tu část objektu odvozené třídy, která patří třídě základní. Jestliže konstruktor odvozené třídy nevolá pomocí seznamu inicializátorů explicitně konstruktor základní třídy, použije se implicitní konstruktor základní třídy. V řetězu dědičnosti může každá třída pomocí seznamu inicializátorů předat informace zpět své nejbližší základní třídě.

Destruktory se také nedědí. Při rušení objektu však program nejdříve zavolá destruktor odvozené třídy a potom destruktor třídy základní. Jestliže existuje implicitní destruktor základní třídy, vygeneruje kompilátor i implicitní destruktor odvozené třídy. Obecně řečeno, slouží-li třída jako základní, měl by být její destruktor virtuální.

Operátor přiřazení se také nedědí. Má však několik zajímavých vlastností, na které se nyní podíváme.

Operátor přiřazení

Kompilátor každou třídu automaticky vybaví operátorem přiřazení pro přiřazení jednoho objektu jinému objektu stejné třídy. Implicitní verze tohoto operátoru používá přiřazení po položkách, každé položce v cílovém objektu je přiřazena hodnota odpovídající položky ze zdrojového objektu. Jestliže však objekt náleží nějaké odvozené třídě, přiřadí kompilátor odvozenému objektu část patřící základní třídě pomocí operátoru přiřazení základní třídy. Jestliže jste pro základní třídy vytvořili explicitní operátor přiřazení, použije se tento operátor. Podobně jestliže třída obsahuje položku, kterou je objekt jiné třídy, použije se pro tuto položku operátor přiřazení oné třídy pro kopii dané položky.

Jak jste již několikrát viděli, jestliže konstruktory třídy inicializují ukazatele pomocí operátoru `new`, musíte vytvořit explicitní operátor přiřazení. Protože C++ používá pro základní část odvozených objektů operátor přiřazení základní třídy, nemusíte operátor přiřazení v odvozené třídě předefinovat, pokud ovšem nejsou přidány položky vyžadující zvláštní péči. Třída `BankAccountD` například definovala přiřazení explicitně, ale odvozená třída `Overdraft` používá implicitní operátor přiřazení vygenerovaný pro tuto třídu.

Předpokládejme ale, že odvozená třída používá operátor `new` a musíte vytvořit explicitní operátor přiřazení. Operátor musí sloužit všem položkám třídy, nejen těm novým. Třída `OverdraftD` ukazuje, jak to lze udělat:

```
OverdraftD & OverdraftD::operator=(const OverdraftD & od)
{
    if (this == &od)
        return *this;
    BankAccountD::operator=(od);  // přiřazení ze základní třídy
    delete [] codeName;
    codeName = new char[strlen(od.codeName) + 1];
    strcpy(codeName, od.codeName);
    maxLoan = od.maxLoan;
    owesBank = od.owesBank;
    rate = od.rate;
    return *this;
}
```

A co přiřazení objektu odvozené třídy objektu základní třídy? (Poznámka: Není to stejné jako inicializovat referenci na objekt základní třídy pomocí objektu odvozené třídy.)

```
BankAccountD blips;                                          // základní třída
OverdraftD snips("Ranele Posh", "Topper", 222333,3993);  // odvozená třída
blips = snips;          // přiřazení odvozeného objektu základnímu objektu
```

Který operátor přiřazení se použije? Vzpomeňte si, že přiřazovací příkaz se převede na metodu vyvolanou objektem na levé straně přiřazení:

```
blips.operator=(snips);
```

Zde je na levé straně objekt třídy `BankAccountD`, takže bude vyvolána funkce `BankAccountD::operator=(const BankAccountD &)`. Vztah *je* umožňuje, aby reference na třídu BankAccounD odkazovala na objekt odvozené třídy, jako je objekt `snips`. Operátor přiřazení se zabývá pouze položkami základní třídy, takže položka `codeName` a ostatní položky třídy `OverdraftD` objektu snips jsou v přiřazení ignorovány. Stručně řečeno, objekt odvozené třídy můžete přiřadit objektu třídy základní, ale přiřazení se bude týkat pouze položek základní třídy.

A co opačný případ? Můžete přiřadit objekt základní třídy objektu třídy odvozené?

```
BankAccount gp("Griff Parker", 21234, 1200);    // základní třída
Overdraft temp;                                  // odvozená třída
temp = gp;  // možné?
```

Zde by byl přiřazovací příkaz převeden na následující tvar:

```
temp.operator=(gp);
```

Objektem na levé straně je objekt třídy `Overdraft`, takže je vyvolána funkce `Overdraft::operator=(const Overdraft &)`. Reference na objekt odvozené třídy však nemůže automaticky odkazovat na objekt třídy základní, takže tento kód fungovat nebude, pokud ovšem neexistuje konverzní konstruktor:

```
Overdraft(const BankAccount &);
```

(Je možné, jako v tomto případě, že existuje konstruktor s více parametry, které mají implicitní hodnoty.) V tomto případě program pomocí tohoto konstruktoru vytvoří dočasný objekt třídy `Overdraft` z objektu `gb`. Tento dočasný objekt se potom použije jako parametr operátoru přiřazení.

Jinou možností je definovat operátor přiřazení pro přiřazení objektu základní třídy objektu třídy odvozené:

```
Overdraft & Overdraft::operator=(const BankAccount &) {…}
```

Zde typy parametru přesně odpovídají přiřazovacímu příkazu a konverze typů není potřeba.

Privátní metody a metody chráněné

Pamatujte, že chráněné položky se v odvozené třídě chovají jako položky veřejné, avšak pro okolní svět jako položky soukromé. Odvozená třída může k chráněným položkám základní třídy přistupovat přímo, ale k soukromým položkám pouze pomocí členských funkcí základní třídy. Vytvoříte-li tedy v základní třídě položky jako soukromé, bude program bezpečnější, zatímco s chráněnými položkami bude kód jednodušší a přístup rychlejší. Pokud jde o datové položky, Stroustrup zastává názor, že lepší je používat položky soukromé než chráněné, ale že chráněné metody jsou užitečné. (Bjarne Stroustrup, *The Design and Evolution of C++*. Reading, MA: Addison-Wesley Publishing Company, 1994.)

Virtuální metody

Při návrhu základní třídy se musíte rozhodnout, zda budou metody této třídy virtuální či nikoli. Pokud chcete, aby v odvozené třídě bylo možné nějakou metodu předefinovat, definujte ji v základní třídě jako virtuální. Umožníte tím pozdní neboli dynamickou vazbu. Jestliže nechcete, aby bylo možné metodu předefinovat, nedělejte ji virtuální. Nezabráníte tím, aby někdo tuto metodu předefinoval, ale každý by si to měl vysvětlit tak, že předefinování není žádoucí.

Povšimněte si, že nevhodný kód může dynamickou vazbu obejít. Uvažujte například dvě následující funkce:

```
void show(const BankAccount & rba)
{
    rba.ViewAcct();
    cout << endl;
}
void sloppy(BankAccount ba)
{
    ba.ViewAcct();
    cout << endl;
}
```

První předává objekt odkazem a druhá hodnotou.

Nyní předpokládejme, že je použijete s parametrem, kterým je objekt odvozené třídy:

```
Overdraft buzz("Buzz Parsec", 00001111, 4300);
show(buzz);
sloppy(buzz);
```

Vzhledem k tomu, že parametr `rba` bude po volání funkce `show()` referencí na objekt buzz třídy `Overdraft`, takže volaná funkce `rba.ViewAcct()` bude považována za verzi třídy `Overdraft`, což je správné. Ale ve funkci `sloppy()`, která předává objekt hodnotou, je ba objekt třídy `BankAccount` vytvořený konstruktorem `BankAccount(const BankAccount &)`. (Díky automatickému přetypování na předka může parametr konstruktoru odkazovat na objekt třídy `Overdraft`.) Ve funkci `sloppy()` je tedy volaná funkce `ba.ViewAcct()` verzí třídy `BankAccount`, takže zobrazena bude pouze část náležející třídě BankAccount.

Destruktory

Jak bylo zmíněno dříve, destruktor základní třídy by měl být virtuální. Budete-li potom rušit objekt odvozené třídy pomocí základního ukazatele nebo reference na objekt, použije program nejen destruktor základní třídy, ale ještě před ním zavolá destruktor třídy odvozené.

Shrnutí funkcí třídy

V jazyce C++ existuje mnoho variant funkcí tříd. Některé je možné dědit, jiné ne. Některé funkce operátorů mohou být jak členskou, tak i spřátelenou funkcí, zatímco jiné mohou být pouze členskými funkcemi. Tabulka 12.1 vycházející z podobné tabulky z *Annotated Reference Manual* (Komentovaná referenční příručka) tyto vlastnosti shrnuje. Zápis `op=` označuje operátory `+=`, `*=` atd. Všimněte si, že vlastnosti operátorů tvaru `op=` se nijak neliší od kategorie „ostatní operátory“. Důvodem pro oddělení operátorů `op=` je poukázat na skutečnost, že tyto operátory se chovají jinak než operátor =.

Tabulka 12.1. Vlastnosti členských funkcí.

Funkce	Děděná	Metoda nebo spřátelená funkce	Automaticky generovaná	Může být virtuální	Může mít typ návratové hodnoty
konstruktor	ne	metoda	ano	ne	ne
destruktor	ne	metoda	ano	ano	ne
=	ne	metoda	ano	ano	ano
&	ano	obojí	ano	ano	ano
konverze	ano	metoda	ne	ano	ne
()	ano	metoda	ne	ano	ano
[]	ano	metoda	ne	ano	ano
->	ano	metoda	ne	ano	ano
op=	ano	obojí	ne	ano	ano
new	ano	statická metoda	ne	ne	void *
delete	ano	statická metoda	ne	ne	void
ostatní operátory	ano	obojí	ne	ano	ano
ostatní metody	ano	metoda	ne	ano	ano
přátelé	ne	přítel	ne	ne	ano

Shrnutí

Díky dědičnosti můžete kód programu přizpůsobit konkrétním potřebám, jestliže definici nové třídy (odvozené) odvodíte od třídy existující (základní). Veřejná dědičnost modeluje vztah *je*, což znamená, že objekt odvozené třídy by také měl být určitým druhem objektu třídy základní. Jako součást modelu *je* dědí odvozená třída datové položky a většinu metod základní třídy. Nedědí však konstruktory, destruktor a operátor přiřazení. K veřejným a chráněným položkám základní třídy může odvozená třída přistupovat přímo a k soukromým pomocí veřejných a chráněných metod základní třídy. Potom můžete do odvozené třídy přidat nové datové položky a metody a použít ji jako základní pro další vývoj. Každá odvozená třída vyžaduje vlastní konstruktory. Když program vytváří objekt odvozené třídy, zavolá nejdříve konstruktor základní třídy a potom konstruktor třídy odvozené. Když program objekt ruší, zavolá nejdříve destruktor odvozené třídy a potom destruktor třídy základní.

Jestliže má třída být základní, můžete místo soukromých položek používat položky chráněné a umožnit tak odvozeným třídám přímý přístup k těmto položkám. Použití soukromých položek však obecně zúží prostor pro chyby v programu. Máte-li v úmyslu nějakou metodu základní třídy v odvozené třídě předefinovat, deklarujte ji pomocí klíčového slova `virtual` jako virtuální. To umožní, aby se s objekty, ke kterým se přistupuje pomocí ukazatelů nebo referencí, zacházelo podle jejich typu a nikoli podle typu ukazatele nebo reference. Zvláště destruktor základní třídy by měl být za normálních okolností virtuální.

V případě potřeby můžete definovat abstraktní základní třídu, která definuje rozhraní, ale nezabývá se implementací funkcí. Můžete například definovat abstraktní třídu `Shape` (tvar), od které budou odvozeny třídy představující určitý tvar, například `Circle` (kruh) nebo `Square` (čtverec). Abstraktní základní třída musí obsahovat alespoň jednu čistě virtuální metodu. Čistě virtuální metodu můžete deklarovat umístěním výrazu `=0` před ukončovací středník deklarace.

virtual double area() const = 0;

Čistě virtuální metody nedefinujete a také nemůžete vytvořit objekt třídy obsahující čistě virtuální metody. Čistě virtuální metody slouží pro definování společného rozhraní, které budou používat odvozené třídy.

Opakovací otázky

1. Co zdědí odvozená třída od třídy základní?
2. Co odvozená třída od základní třídy nezdědí?
3. Předpokládejme, že by návratový typ funkce `BankAccountD::operator=()` byl definován jako `BankAccountD` namísto `BankAccountD &`. Mělo by to nějaký účinek?
4. V jakém pořadí se volají konstruktory a destruktory třídy při vytváření a rušení objektu třídy odvozené?
5. Jestliže do odvozené třídy nepřidáte k datovým položkám základní třídy žádné nové datové položky, musí mít tato třída konstruktory?

6. Předpokládejme, že v základní i odvozené třídě je definována metoda se stejným názvem a objekt odvozené třídy tuto metodu vyvolá. Která to bude?
7. Kdy musíte v odvozené třídě definovat operátor přiřazení?
8. Můžete přiřadit adresu objektu odvozené třídy ukazateli na objekt třídy základní? A můžete přiřadit adresu objektu základní třídy ukazateli na objekt třídy odvozené?
9. Můžete přiřadit objekt odvozené třídy objektu třídy základní? Můžete přiřadit objekt základní třídy objektu třídy odvozené?
10. Předpokládejme, že definujete funkci, která má jako parametr referenci na objekt základní třídy. Proč může tato funkce mít jako parametr také objekt odvozené třídy?
11. Předpokládejme, že definujete funkci, která má jako parametr objekt základní třídy (to znamená, že funkce předává objekt základní třídy hodnotou). Proč může mít i tato funkce jako parametr objekt odvozené třídy?
12. Proč je většinou lepší předávat objekty odkazem než hodnotou?
13. Předpokládejme, že `Corporation` je základní třída a `PublicCorporation` je třída odvozená. Předpokládejme také, že v každé třídě je definována členská funkce `head()`, že `ph` je ukazatel na typ `Corporation` a je mu přiřazena adresa objektu třídy `PublicCorporation`. Jakým způsobem bude interpretováno volání `ph->head()`, jestliže je v základní třídě definována funkce `head()` jako
    a) běžná
    b) virtuální
14. Je něco špatně v následujícím kódu?

```
class Kitchen
{
private:
    double kit_sq_ft;
public:
    Kitchen() {kit_sq_ft = 0.0; }
    virtual double area() { return kit_sq_ft * kit_sq_ft; }
};
class House : public Kitchen
{
private:
    double all_sq_ft;
public:
    House() {all_sq_ft += kit_sq_ft;}
    double area(const char *s) { cout << s;
    return all_sq_ft; }
};
```

Cvičení

1. Nejdříve deklarujte následující třídu:

```
// základní třída
class Cd { // reprezentuje CD disk
private:
    char performers[50];
    char label[20];
    int selections;  // počet voleb
    double playtime; // hrací doba v minutách
public:
    Cd(char * s1, char * s2, int n, double x);
    Cd(const Cd & d);
    Cd();
    ~Cd();
    void Report() const; // vypíše všechna data na CD
    Cd & operator=(const Cd & d);
};
```

Vytvořte odvozenou třídu `Classic`, do které přidáte znakové pole pro řetězec, který bude identifikovat prvotní práci na disku. Pokud bude nutné, aby v základní třídě byly některé funkce virtuální, upravte deklaraci základní třídy. Vyzkoušejte výsledek pomocí následujícího programu:

```
#include <iostream>
using namespace std;
#include "classic.h"   // obsahuje #include cd.h
void Bravo(const Cd & disk);
int main()
{
    Cd c1("Beatles", "Capitol", 14, 35.5);
    Classic c2 = Classic("Piano Sonata in B flat,
            Fantasia in C", "Alfred Brendel", "Philips", 2,
            57.17);
    Cd *pcd = &c1;
    cout << "Using object directly:\n";
    c1.Report();  // použije metodu třídy Cd
    c2.Report();  // použije metodu třídy Classic
    cout << "Using type cd * pointer to objects:\n";
    pcd->report(); // použije pro objekt třídy Cd metodu třídy Cd
    pcd = &c2;
    pcd->Report(); // použije pro objekt třídy Classic metodu
                   // třídy Classic
    cout << "Calling a function with a Cd reference
        argument:\n";
    Bravo(c1);
    Bravo(c2);
    cout << "Testing assignment: ";
    Classic copy;
    copy = c2;
```

```
        copy.Report()
        return 0;
    }
    void bravo(Cd & disk)
    {
        disk.report();
    }
```

2. Zopakujte cvičení 1 s tím, že tentokrát pro různé řetězce těchto dvou tříd použijete místo polí pevné délky dynamicky přidělenou paměť.
3. Upravte třídy `BankAccount` a `Overdraft` tak, aby obě byly odvozeny od stejné abstraktní základní třídy. Výsledek ověřte pomocí programu podobného tomu z výpisu 12.8.
4. V nejmenované firmě mají sklad alkoholu. K jeho popisu vytvořil správce třídu `Port` s následující deklarací:

```
#include <iostream>
using namespace std;
class Port
{
private:
    char * brand;
    char style[20]; // tzn. zlatavá barva, rubínová barva, ročník
    int bottles;
public:
    Port(const char * br = "none", const char * st = "none",
        int b = 0);
  Port(const Port & p);         // kopírovací konstruktor
  virtual ~Port() { delete [] brand; }
  Port & operator=(const Port & p);
  Port & operator+=(int b);     // přidá b láhví
  Port & operator-=(int b);// odepíše b láhví, pokud existují
  int BottleCount() const { return bottles; }
  virtual void Show() const;
  friend ostream & operator<<(ostream & os, const Port & p);
  };
```

Metoda `Show()` prezentuje informace v následujícím tvaru:

```
Brand: Gallo
Kind: tawny
Bottles: 20
```

Funkce `operator<<()` vypisuje informace v následujícím formátu (bez znaku nového řádku na konci):

```
Gallo, tawny, 20
```

Správce dokončil definice metod třídy `Port` a potom odvodil následující třídu `VintagePort`. Potom však byl ze svého místa propuštěn, neboť omylem poslal jednu láhev Cockburnu ročník 1945 osobě připravující pokusnou omáčku na opékání masa.

```
class VintagePort : public Port // položka style má nutně hodnotu
                                // "vintage"
{
private:
    char * nickname; // například "The Noble" nebo "Old Velvet" atd.
    int year;        // ročník
public:
    VintagePort();
    VintagePort(const char * br, int b, const char * nn, int y);
    VintagePort(const VintagePort & vp);
    ~VintagePort() { delete [] nickname; }
    VintagePort & operator=(const VintagePort & vp);
    void Show() const;
    friend ostream & operator<<(ostream & os,
    const VintagePort & vp);
  };
```

Vaším úkolem je dokončení třídy `VintagePort`.

a) Prvním úkolem je vytvořit znovu definice metod třídy `Port`, protože dřívější správce své definice při propuštění zničil.

b) Dalším úkolem je vysvětlit, proč jsou některé metody předefinovány a jiné ne.

c) Třetím úkolem je vysvětlit, proč funkce `operator=()` a `operator<<()` nejsou virtuální.

d) Čtvrtým úkolem je vytvořit definice metod třídy `VintagePort`.

Znovupoužitelný kód v jazyce C++

KAPITOLA 13

Témata kapitoly:

- Třídy s položkami typu objekt
- Soukromá a chráněná dědičnost
- Vytváření šablon tříd
- Používání šablon tříd
- Specializace šablon
- Vícenásobná dědičnost
- Virtuální základní třídy

Jedním z hlavních cílů jazyka C++ je usnadnit znovupoužití kódu. Veřejná dědičnost je jedním ze způsobů, jak toho dosáhnout, ale ne jediným. V této kapitole budeme zkoumat další možnosti. Jedna z technik spočívá v používání položek třídy, které jsou samy objekty jiné třídy. Této technice se říká kompozice nebo vrstvení. Další volba je mezi dědičností soukromou a chráněnou. Kompozice, soukromá a chráněná dědičnost jsou běžně používány pro implementaci vztahů *má*, tedy vztahů, v nichž nová třída obsahuje objekt jiné třídy. Třída `Stereo` může například obsahovat objekt třídy `CdPlayer`. Vícenásobná dědičnost umožňuje vytvářet třídy dědící ze dvou nebo více základních tříd, a tím kombinovat jejich funkčnost.

V kapitole 9 byly uvedeny šablony funkcí. Nyní se podíváme na šablony tříd, což je další způsob opětovného použití kódu. Šablony tříd umožňují definovat třídu obecně. Potom můžete pomocí šablony vytvářet určité třídy definované pro určité typy. Mohli byste například definovat šablonu obecného zásobníku a potom s její pomocí vytvořit třídu reprezentující zásobník hodnot typu `int` a jinou třídu reprezentující zásobník hodnot typu `double`. Dokonce byste mohli vygenerovat třídu reprezentující zásobník zásobníků.

Třídy s objekty jako položkami

Začneme s třídami, které jako položky obsahují objekty tříd. Některé třídy, jako například třída `String` z kapitoly 11 nebo standardní třídy jazyka C++ a šablony z kapitoly 15, nabízejí vhodnou reprezentaci složek rozsáhlejších tříd. Nyní se podíváme na konkrétní příklad.

Co je student? Někdo zapsaný na škole? Někdo zabývající se zkoumáním? Uprchlík před drsnými požadavky reálného světa? Někdo, kdo je identifikován jménem a dosaženým počtem bodů ze zkoušek? Samozřejmě, poslední definice pro popis osobnosti naprosto nedostačuje, ale dobře se hodí pro jednoduchou reprezentaci v počítači. Pojďme tedy podle této definice vytvořit třídu `Student`.

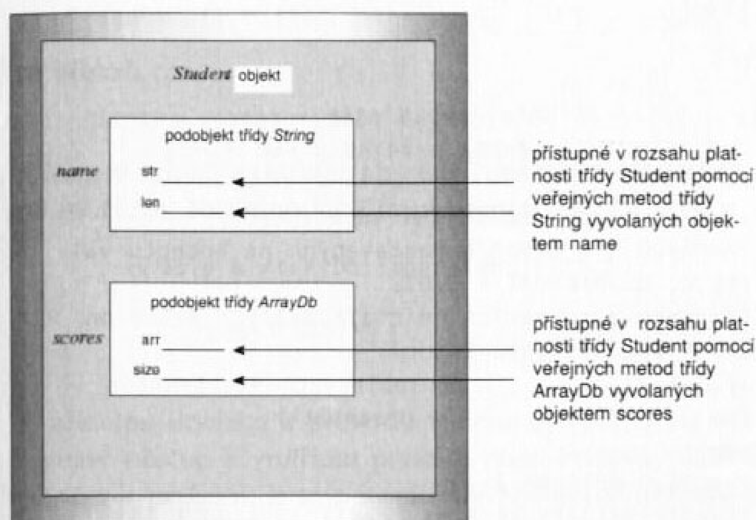
Zjednodušíte-li studenta na jméno a množinu výsledků zkoušek, nabízí se použití objektu třídy `String` (kapitola 11) pro jméno a objektu třídy reprezentující pole (přijde později), který bude obsahovat výsledky zkoušek (budeme předpokládat typ `double`). (Až se v kapitole 15 dozvíte o knihovních třídách, asi použijete třídy `string` a `vector`.) Možná jste v pokušení odvodit veřejně třídu `Student` od těchto dvou tříd. Jednalo by se o příklad použití vícenásobné veřejné dědičnosti, kterou jazyk C++ umožňuje, ale zde by se nehodila. Důvodem je, že vztah studenta k těmto třídám nezapadá do modelu *je*. Student není jméno. Student není pole výsledků zkoušek. Zde máme před sebou vztah *má*. Student má jméno a má pole výsledků zkoušek. Obvyklou technikou pro modelování vztahů *má* je v C++ kompozice. To znamená, vytvoření třídy složené z položek nebo obsahující položky, které jsou objekty jiné třídy. Deklaraci třídy `Student` můžeme začít například takto:

```
class Student
{
private:
    String name;      // objekt třídy String pro jméno
    ArrayDb scores;   // objekt třídy ArrayDb pro pole výsledků
    ...
};
```

Jako obvykle jsou datové položky třídy soukromé. To znamená, že členské funkce třídy `Student` mohou pomocí veřejných rozhraní tříd `String` a `ArrayDb` (pole čísel typu double) přistupovat k objektům `name` a `scores` a upravovat je, ale okolní svět tak učinit nemůže. Okolní svět může k položkám `name` a `scores` přistupovat přes veřejné rozhraní definované pro třídu `Student` (viz obrázek 13.1). Běžně se tento způsob popisuje tak, že třída `Student` získá implementaci svých objektů, ale nezdědí rozhraní. Pro uložení jména používá objekt třídy `Student` například implementaci třídy `String` namísto implementace položky name jako typu `char *` nebo `char name[26]`. Ale objekt třídy `Student` nemá vrozenou schopnost používat funkci `operator==()` třídy `String`.

Rozhraní a implementace

Při veřejné dědičnosti třída dědí rozhraní a pravděpodobně i implementaci. (Čistě virtuální funkce v základní třídě poskytují rozhraní bez implementace.) Získat rozhraní je úkolem vztahu *je*. U kompozice naopak třída získá implementaci bez rozhraní. Dědičnost bez rozhraní je úkolem vztahu *má*.



Obrázek 13.1. Kompozice

Skutečnost, že objekt třídy automaticky nezíská rozhraní obsaženého objektu, je pro vztah *má* výhodná. Bylo by například možné rozšířit třídu `String` o přetížený operátor +, sloužící ke spojení dvou řetězců, koncepčně však nemá smysl spojovat dva objekty třídy `Student`. To je jeden z důvodů, proč v tomto případě není použita veřejná dědičnost. Na druhé straně části rozhraní obsažené třídy mohou mít smysl v nové třídě. Například byste mohli použít metodu `operator<()` třídy `String` pro setřídění objektů třídy `Student` podle jména. Toho můžete dosáhnout definicí členské funkce `Student::Operator<()`, která bude interně používat funkci `String::Operator<()`. Přejděme k některým detailům.

Třída ArrayDb

Prvním detailem je vytvoření třídy `ArrayDb` tak, aby ji mohla použít třída `Student`. Bude hodně podobná třídě `String`, protože tato také představuje pole, v tomto případě pole znaků. Nejdříve si uveďme některé vlastnosti, které třída `ArrayDb` musí mít nebo které jsou žádoucí:

- Měla by umožňovat uložení několika hodnot typu `double`.
- Měla by poskytovat přímý přístup k jednotlivým položkám pomocí hranatých závorek s indexem.
- Musí umožnit přiřazení jednoho pole druhému.
- Třída bude provádět kontrolu mezí pole, aby bylo zajištěno, že indexy jsou platné.

První dvě vlastnosti jsou podstatou pole. Třetí vlastnost neplatí pro vestavěná pole, ale platí pro objekty tříd, takže vytvořením třídy s polem tuto vlastnost získáte. Poslední vlastnost pro vestavěná pole také neplatí, lze ji však přidat jako součást druhé vlastnosti.

V této chvíli se může návrh třídy z větší části podobat deklaraci třídy `String`. To znamená, že můžete napsat toto:

```
class ArrayDb
{
private:
    unsigned int size;          // počet prvků pole
    double * arr;               // adresa prvního prvku
public:
    ArrayDb();                  // implicitní konstruktor
    // vytvoří objekt ArrayDb o n prvcích nastavených na hodnotu val
    ArrayDb(unsigned int n, double val = 0.0);
    // vytvoří objekt ArrayDb o n prvcích inicializovaných polem pn
    ArrayDb(const double * pn, unsigned int n);
    ArrayDb(const ArrayDb & a);          // kopírovací konstruktor
    virtual ~ArrayDb();                  // destruktor
      // ostatní metody
    ArrayDb & operator=(const ArrayDb & a);
    friend ostream & operator<<(ostream & os, const ArrayDb & a);
};
```

Třída vytvoří pole požadované velikosti pomocí dynamicky přidělené paměti. Bude tedy také mít destruktor, kopírovací konstruktor a operátor přiřazení. Z praktických důvodů bude obsahovat několik dalších konstruktorů.

Hlavní novou vlastnost představuje přímý přístup pomocí zápisu pole. Předpokládejme, že provedete následující deklaraci:

```
ArrayDb scores(5, 20.0); // 5 prvků, každý  nastaven na hodnotu 20.0
```

Jestliže se objekt scores chová doopravdy jako pole, měli byste mít možnost psát následující příkazy:

```
double temp = scores[3];
scores[3] = 16.5;
```

Co tento přístup vyžaduje, aby veřejný výraz `scores[3]` odpovídal soukromé reprezentaci `scores.arr[3]`. Toho lze dosáhnout snadno, poněvadž `[]` je pouze další operátor jazyka C++, stejně jako <<, a může tedy být přetížen. Stejně jako << má dva operandy. Hlavní zvláštností je, že jeden z operandů je uveden v hranatých závorkách. Každopádně však výraz

```
scores[3]
```

je mapován jako následující volání přetížené funkce operátoru:

```
scores.operator[](3)
```

Zde je název funkce `operator[]`. Protože levým operandem je objekt třídy `ArrayDb` a pravým číslo typu int, odpovídala by členské funkci `ArrayDb::operator[](int)`.

Příkaz

```
temp = scores[2];
```

odpovídá následujícímu příkazu:

```
temp = scores.operator[](2);
```

To znamená, že funkce by měla vrátit hodnotu prvku `scores.attr[2]` nebo referenci na tento prvek pole. Avšak příkaz typu

```
scores[2] = 19.0;
```

se přeloží na

```
scores.operator[](2) = 19.0;
```

Přiřazení funkci vyžaduje, aby tato funkce vracela referenci, ne hodnotu, zbývá tedy jediná možnost. Nejjednodušší implementace by byla tato:

```
double & ArrayDb::operator[](int i)
{
    return arr[i];
}
```

V principu dochází k převodu výrazu `scores[i]` na výraz `scores.arr[i]`, což umožňuje veřejný přístup k vnitřním prvkům pole. Protože hlavním účelem pole je umožnit přímý přístup k prvkům, je toto konkrétní zpřístupnění soukromého prvku pro veřejné úpravy žádoucí. Jak uvidíte v další části, použití operátoru [] namísto povolení přímého přístupu pomocí veřejných datových položek poskytuje lepší ochranu.

Vylepšení operátoru []

Přetížený operátor [] umožňuje přístup k jednotlivým prvkům, ale pouze pomocí tohoto konkrétního operátoru. Tím vzniká možnost zabudovat některé bezpečnostní kontroly. Konkrétně metoda může kontrolovat, zda je plánovaný index v mezích pole. To znamená, že operátor můžete přepsat tímto způsobem:

```
double & ArrayDb::operator[](int i)
{
    // kontrola indexu před pokračováním
    if (i < 0 || i >= size)
    {
        cerr << "Prekroceny meze pole: "
             << i << " je neplatny index\n";
        exit(1);
    }
    return arr[i];
}
```

Program se tím zpomalí, protože při každém přístupu k prvku pole je potřeba vyhodnotit příkaz `if`. Ale zvýší se ochrana, program nebude moci vložit hodnotu do prvku s indexem 2000 u pole s pěti prvky.

Alternativa s modifikátorem const

Předchozí definice má jednu slabinu. Uvažujte následující kód:

```
const ArrayDb noChanges(5, 0.4);  // 5 prvků  nastavených na hodnotu 0.4
cout << noChanges[2];             // neplatné!
```

Problém je, že objekt `noChanges` je konstantní, ale funkce `operátor[]` neslibuje, že nebude hodnoty měnit a `nekonstantní` metodu s `konstantním` objektem vyvolat nemůžete. Naštěstí existuje jednoduché řešení. Můžete přetížit funkce, jejichž signatury jsou totožné až na to,

že jedna funkce používá konstantní referenci nebo ukazatel a druhá ne. To znamená, že třída potřebuje také následující funkci:

```
const double & ArrayDb::operator[](int i) const
{
    // kontrola před pokračováním
    if (i < 0 || i >= size)
    {
        cerr << "Prekroceny meze pole: "
            << i << " je neplatny index\n";
        exit(1);
    }
    return arr[i];
}
```

Všimněte si, že protože je vrácená reference `konstantní`, nemůžete tuto verzi použít pro přiřazení prvku pole. Takové chování je `konstantnímu` poli vlastní. Můžete pouze získat hodnotu v prvku uloženou. Příkaz

```
cout << noChanges[2];   // možné u konstantnich objektů
```

je tedy platný, zatímco následující příkaz je neplatný:

```
noChanges[2] = 300.4; // nepovoleno u konstantnich objektů
```

Pro `konstantní` objekty třídy `ArrayDb` kompilátor vybere konstantní verzi metody `operator[]()` a pro `nekonstantní` objekty použije druhou verzi.

Ve výpisu 13.1 je hlavičkový soubor třídy `ArrayDb`. Pro větší pohodlí třída obsahuje metodu `Average()` vracející průměr z hodnot prvků pole.

Výpis 13.1. arraydb.h

```
// arraydb.h — třída pro pole
#ifndef _ARRAYDB_H_
#define _ARRAYDB_H_
#include <iostream>
using namespace std;
class ArrayDb
{
private:
    unsigned int size;                          // počet prvků pole
    double * arr;                               // adresa prvniho prvku
public:
    ArrayDb();                                  // bezparametrický konstruktor
    // pole ArrayDb s n prvky, s hodnotou val
    explicit ArrayDb(unsigned int n, double val = 0.0);
    // pole ArrayDb s n prvky, inicializované polem pn
    ArrayDb(const double * pn, unsigned int n);
    ArrayDb(const ArrayDb & a);                     // kopirovaci konstruktor
    virtual ~ArrayDb();                             // destruktor
    unsigned int ArSize() const {return size;} //  vráti velikost pole
```

```
    double Average() const;                  // vrátí průměr
// přetížené operátory
    virtual double & operator[](int i);     // indexování pole
    virtual const double & operator[](int i) const; // indexování pole
    ArrayDb & operator=(const ArrayDb & a);
    friend ostream & operator<<(ostream & os, const ArrayDb & a);
};
#endif
```

Kompatibilita:

Starší implementace nepodporují klíčové slovo `explicit`.

Zajímavé je použití klíčového slova `explicit` u jednoho z konstruktorů:

```
explicit ArrayDb(unsigned int n, double val = 0.0);
```

Vzpomeňte si, že konstruktor, který může být vyvolán s jedním parametrem, slouží jako implicitní konverzní funkce z typu parametru na typ dané třídy. V tomto případě nepředstavuje první parametr hodnotu pole, ale počet prvků. Nemá tedy smysl, aby konstruktor sloužil jako konverzní funkce z typu int na typ `ArrayDb`. Použití klíčového slova `explicit` vypíná implicitní konverze. Pokud byste toto klíčové slovo vynechali, byl by možný následující kód:

```
ArrayDb doh(20,100); // pole 20 prvků nastavených na hodnotu 100
doh = 5;   // nastaví doh na pole 5 prvků s hodnotou 0
```

Zde nepozorný programátor zapsal doh namísto `doh[0]`. Kdyby konstruktor neobsahoval klíčové slovo `explicit`, hodnota 5 by byla pomocí konstruktoru `ArrayDb(5)` převedena na dočasný objekt třídy `ArrayDb`, přičemž jako druhý parametr by byla použita hodnota 0. Potom by přiřazení nahradilo původní objekt `doh` dočasným objektem. Při použití `explicit` odchytí kompilátor tento přiřazovací operátor jako chybu.

Výpis 13.2 obsahuje implementaci.

Výpis 13.2. arraydb.cpp

```
// arraydb.cpp — metody třídy ArrayDb
#include <iostream>
using namespace std;
#include <cstdlib> // prototyp funkce exit()
#include "arraydb.h"
// bezparamerický konstruktor
ArrayDb::ArrayDb()
{
    arr = NULL;
    size = 0;
}
// vytvoření pole s n prvky s nastavenými na hodnotou val
ArrayDb::ArrayDb(unsigned int n, double val)
{
```

```
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
        arr[i] = val;
}
// inicializace objektu pomocí normálního pole
ArrayDb::ArrayDb(const double *pn, unsigned int n)
{
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
        arr[i] = pn[i];
}
// inicializace objektu třídy ArrayDb jiným objektem stejné třídy
ArrayDb::ArrayDb(const ArrayDb & a)
{
    size = a.size;
    arr = new double[size];
    for (int i = 0; i < size; i++)
        arr[i] = a.arr[i];
}
ArrayDb::~ArrayDb()
{
    delete [] arr;
}
double ArrayDb::Average() const
{
    double sum = 0;
    int i;
    int lim = ArSize();
    for (i = 0; i < lim; i++)
        sum += arr[i];
    if (i > 0)
        return sum / i;
    else
    {
        cerr << "Zadne zaznamy v poli vysledku\n";
        return 0;
    }
}
// umožní uživateli přistupovat k prvkům pomocí indexu
// (přiřazení povoleno)
double & ArrayDb::operator[](int i)
{
    // kontrola před pokračováním
    if (i < 0 || i >= size)
    {
        cerr << "Prekroceny meze pole: "
            << i << " je neplatny index\n";
        exit(1);
    }
```

```
        return arr[i];
    }
    // umožní uživateli přistupovat k prvkům pomocí indexu
    // (přiřazení zakázáno)
    const double & ArrayDb::operator[](int i) const
    {
        // kontrola indexu před pokračováním
        if (i < 0 || i >= size)
        {
            cerr << "Prekroceny meze pole: "
               << i << " je neplatny index\n";
            exit(1);
        }
        return arr[i];
    }
    // definice operátoru přiřazení
    ArrayDb & ArrayDb::operator=(const ArrayDb & a)
    {
        if (this == &a)            // je-li objekt přiřazen sám sobě,
           return *this;           // žádná změna se neprovede
        delete [] arr;
        size = a.size;
        arr = new double[size];
        for (int i = 0; i < size; i++)
            arr[i] = a.arr[i];
        return *this;
    }
    // úsporný rychlý výstup, 5 hodnot na řádek
    ostream & operator<<(ostream & os, const ArrayDb & a)
    {
        int i;
        for (i = 0; i < a.size; i++)
        {
            os << a.arr[i] << " ";
            if (i % 5 == 4)
                os << "\n";
        }
        if (i % 5 != 0)
            os << "\n";
      return os;
    }
```

Příklad třídy Student

Nyní, když máme v rukou třídu `ArrayDb`, můžeme vytvořit deklaraci třídy `Student`. Samozřejmě by měla obsahovat konstruktory a alespoň několik funkcí pro rozhraní. Výsledek je obsažen ve výpisu 13.3, přičemž všechny konstruktory jsou definovány jako vložené.

Výpis 13.3. studentc.h

```
// studentc.h - definice třídy Student pomocí kompozice
#ifndef _STUDNTC_H_
#define _STUDENTC_H_
#include <iostream>
using namespace std;
#include "arraydb.h"
#include "strng2.h" // z kapitoly 11
class Student
{
private:
    String name;
    ArrayDb scores;
public:
    Student() : name("Null Student"), scores() {}
    Student(const String & s)
        : name(s), scores() {}
    Student(int n) : name("Nully"), scores(n) {}
    Student(const String & s, int n)
        : name(s), scores(n) {}
    Student(const String & s, const ArrayDb & a)
        : name(s), scores(a) {}
    Student(const char * str, const double * pd, int n)
        : name(str), scores(pd, n) {}
    ~Student() {}
    double & operator[](int i);
    const double & operator[](int i) const;
    double Average() const;
// přátelé
    friend ostream & operator<<(ostream & os, const Student & stu);
    friend istream & operator>>(istream & is, Student & stu);
};
#endif
```

Inicializace obsažených objektů

Všimněte si, že všechny konstruktory inicializují členské objekty `name` a `scores` pomocí známé syntaxe seznamu inicializátorů. V několika minulých případech konstruktory tímto způsobem inicializovaly položky vestavěných typů:

```
Queue::Queue(int qs) : qsize(qs) {...} // nastaví qsize na qs
```

Tento kód používá v seznamu název datové položky (`qsize`). Konstruktory z předchozích příkladů inicializovaly pomocí seznamu inicializátorů také část odvozeného objektu patřící základní třídě:

```
OverdraftD::OverdraftD(const OverdraftD & od) : BankAccountD(od) {...}
```

Pro zděděné objekty vyvolaly konstruktory určitý konstruktor základní třídy tím způsobem, že v seznamu inicializátorů byl uveden název třídy. Pro členské objekty používají

konstruktory název položky. Podívejme se například na poslední konstruktor ve výpisu 13.3:

```
Student(const char * str, const double * pd, int n)
        : name(str), scores(pd, n) {}
```

Protože inicializuje členské objekty a ne objekty zděděné, používá v inicializačním seznamu názvy členů a ne názvy tříd. Každá položka v inicializačním seznamu vyvolá odpovídající konstruktor. To znamená, že výraz `name(str)` vyvolá konstruktor `String(const char *)` a výraz `scores(pd, n)` vyvolá konstruktor `ArrayDb(const double *, int)`.

Co se stane, když nepoužijete syntaxi seznamu inicializátorů? Stejně jako u zděděných složek jazyk C++ vyžaduje, aby všechny členské objekty byly vytvořeny dříve, než je vytvořen zbytek objektu. Jestliže tedy inicializační seznam vynecháte, použije C++ implicitní konstruktory definované pro třídy s členskými objekty.

Použití rozhraní pro obsažený objekt

Rozhraní pro obsažený objekt není veřejné, ale lze je použít v metodách třídy. Zde je například způsob, jak můžete definovat funkci vracející průměr známek studenta:

```
double Student::Average() const
{
    return scores.Average();
}
```

Tímto způsobem je definována funkce, která může být vyvolána objektem třídy `Student`. Vnitřně používá funkci `ArrayDb::Average()`. Proměnná `scores` je objektem třídy `ArrayDb`, a může tedy volat členské funkce třídy `ArrayDb`.

Podobně můžete definovat spřátelenou funkci, používající verze operátoru << tříd `String` a `ArrayDb`:

```
ostream & operator<<(ostream & os, const Student & stu)
{
        os << "Vysledky studenta: " << stu.name << ":\n";
        os << stu.scores;
        return os;
}
```

Protože stu.name je objektem třídy `String`, vyvolá funkci `operator<<(ostream &, const String &)`. Podobně objekt třídy `ArrayDb` `stu.scores` vyvolá funkci `operator <<(ostream &, const ArrayDb &)`. Všimněte si, že nová funkce musí být přítelem třídy `Student`, aby mohla přistupovat k jejím položkám `scores` a `name`.

Výpis 13.4 představuje soubor s metodami třídy Student. Obsažené metody umožňují přistupovat k jednotlivým prvkům proměnné scores pomocí operátoru [].

Výpis 13.4. studentc.cpp

```
// studentc.cpp - třída Student s použitím kompozice
#include "studentc.h"
double Student::Average() const
```

```
{
    return scores.Average(); // použije ArrayDb::Average()
}
double & Student::operator[](int i)
{
    return scores[i];          // použije ArrayDb::operator[]()
}
const double & Student::operator[](int i) const
{
    return scores[i];
}
// přátelé
// použití verzí z tříd  String a ArrayDb
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Vysledky studenta: " << stu.name << ":\n";
    os << stu.scores;
    return os;
}
// použití verze z třídy String
istream & operator>>(istream & is, Student & stu)
{
    is >> stu.name;
    return is;
}
```

Zde nebylo nutné psát příliš nového kódu. Použití kompozice umožňuje využít již napsaný kód.

Použití nové třídy

Sestavme malý program testující naši novou třídu. Pro jednoduchost bude používat pole pouze s třemi objekty třídy `Student`, z nichž každý bude mít výsledky z pěti zkoušek. Také bude používat nenáročný cyklus pro vstup hodnot, které nebude ověřovat a neumožní vstupní proces zkrátit. Testovací program je ve výpisu 13.5. Zkompilujte jej společně se soubory `studentc.cpp`, `strng2.cpp`, a `arrayDb.cpp`.

Výpis 13.5. use_stuc.cpp

```
// use_stuc.cpp — použití kompozice
// zkompilovat s studentc.cpp, strng2.cpp, arraydb.cpp
#include <iostream>
using namespace std;
#include "studentc.h"
void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;
int main()
{
    Student ada[pupils] = {quizzes, quizzes, quizzes};
    int i;
```

```
    for (i = 0; i < pupils; i++)
        set(ada[i], quizzes);
    for (i = 0; i < pupils; i++)
    {
        cout << "\n" << ada[i];
        cout << "prumer: " << ada[i].Average() << "\n";
    }
    return 0;
}
void set(Student & sa, int n)
{
    cout << "Zadejte prosim jmeno studenta: ";
    cin >> sa;
    cout << "Zadejte prosim vysledky z " << n << " zkousek:\n";
    for (int i = 0; i < n; i++)
        cin >> sa[i];
    while (cin.get() != '\n')
        continue;
}
```

Zde je ukázka běhu programu:

```
Zadejte prosim jmeno studenta: Gil Bayts
Zadejte prosim vysledky z 5 zkousek:
92 94 96 93 95
Zadejte prosim jmeno studenta: Pat Roone
Zadejte prosim vysledky z 5 zkousek:
83 89 72 78 95
Zadejte prosim jmeno studenta: Fleur O'Day
Zadejte prosim vysledky z 5 zkousek:
92 89 96 78 64

Výsledky studenta: Gil Bayts:
92 94 96 93 95
prumer: 94

Výsledky studenta: Pat Roone:
83 89 72 78 95
prumer: 83.4

Výsledky studenta: Fleur O'Day
92 89 96 78 64
prumer: 83.8
```

Soukromá dědičnost

Pro implementaci vztahu *má* disponuje C++ i jinými prostředky – soukromou dědičností. Při soukromé dědičnosti se z veřejných a chráněných položek základní třídy stávají soukromé položky třídy odvozené. To znamená, že metody základní třídy se nestanou součástí veřejného rozhraní odvozené třídy. Mohou však být použity v metodách odvozené třídy.

Podívejme se na téma rozhraní podrobněji. Při veřejné dědičnosti se z veřejných metod základní třídy stávají veřejné metody třídy odvozené. To je součást vztahu *je*. Při soukromé dědičnosti se z veřejných metod základní třídy stávají soukromé metody třídy odvozené. Stručně řečeno, odvozená třída nedědí rozhraní třídy základní. Jak jste viděli u obsažených objektů, je absence takové dědičnosti součástí vztahu *má*.

Při soukromé dědičnosti třída dědí implementaci. To znamená, že jestliže třídu `Student` založíte na třídě `String`, zdědí třída `Student` část třídy `String`, kterou lze použít pro uložení řetězce. Navíc mohou metody třídy `Student` používat interně metody třídy `String` pro přístup k části třídy `String`.

Kompozice přidá do třídy objekt jako pojmenovaný členský objekt, zatímco soukromá dědičnost přidá objekt jako bezejmenný zděděný objekt. Pro označení objektu přidaného dědičností nebo kompozicí budeme používat termín *podobjekt*.

Soukromá dědičnost tedy poskytuje stejné vlastnosti jako kompozice: získá implementaci a nezíská rozhraní. Může být tedy také použita k implementaci vztahu *má*. Podívejme se, jak můžete navrhnout třídu `Student` pomocí soukromé dědičnosti.

Příklad třídy `Student` (nová verze)

Chcete-li použít soukromou dědičnost, použijte při definici třídy klíčové slovo `private` namísto klíčového slova `public`. (Modifikátor `private` je implicitní, takže jeho vynechání vede také k soukromé dědičnosti.) Třída `Student` by měla dědit ze dvou tříd, deklarace tedy bude obsahovat obě:

```
class Student : private String, private ArrayDb
{
public:
    ...
};
```

Použití více jak jedné základní třídy se nazývá *vícenásobná dědičnost*. Obecně řečeno, může vícenásobná dědičnost, zvláště vícenásobná veřejná dědičnost, vést k problémům, které musí být řešeny pomocí dodatečných syntaktických pravidel. O těchto záležitostech budeme v této kapitole mluvit později. V tomto konkrétním případě však vícenásobná dědičnost žádný problém nepůsobí.

Všimněte si, že nová třída nebude potřebovat soukromou část. To je dáno tím, že obě zděděné základní třídy již obsahují všechny potřebné datové položky. Kompozice poskytla jako položky dva explicitně pojmenované objekty. Soukromá dědičnost však poskytuje jako zděděné položky dva bezejmenné podobjekty. Toto je první z hlavních rozdílů mezi těmito dvěma přístupy.

Inicializace komponent základní třídy

Implicitní dědění komponent místo členských objektů bude mít vliv na kód, protože již nebudete moci objekty popsat pomocí názvů `name` a `scores`. Namísto toho se budete muset vrátit k technikám používaným u veřejné dědičnosti. Uvažujte například konstruktory. Kompozice používala tento konstruktor:

```
Student(const char * str, const double * pd, int n)
  : name(str), scores(pd, n) {}       // použije v kompozici názvy objektů
```

Nová verze použije pro zděděné třídy syntaxi inicializátorů, která identifikuje konstruktory podle názvu *třídy* a ne podle názvu *položky*:

```
Student(const char * str, const double * pd, int n)
    : String(str), ArrayDb(pd, n) {}    // použije v dědičnosti názvy tříd
```

To znamená, že seznam inicializátorů používá výrazy typu `String(str)` namísto `name(str)`. Toto je druhý hlavní rozdíl u těchto dvou přístupů.

Výpis 13.6. obsahuje novou deklaraci třídy. Jedinými změnami je vynechání explicitních názvů objektů a použití názvů tříd místo názvů položek u vložených konstruktorů.

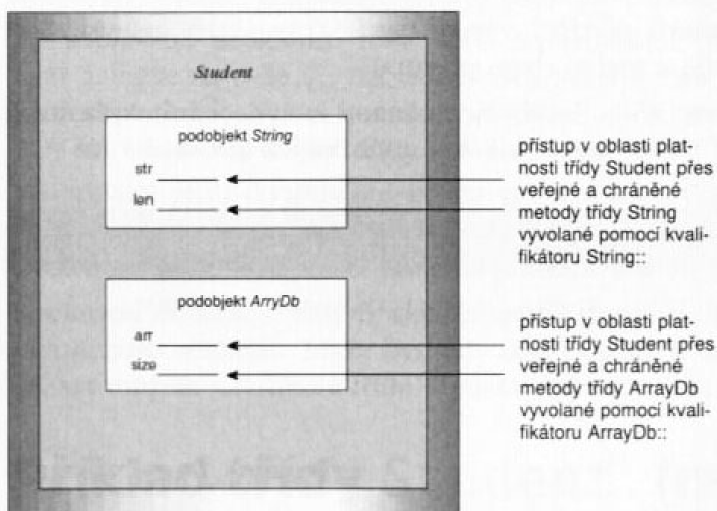
Výpis 13.6. studenti.h

```
// studenti.h – definice třídy Student pomoci soukromé dědičnosti
#ifndef _STUDNTI_H_
#define _STUDENTI_H_
#include <iostream>
using namespace std;
#include "arraydb.h"
#include "strng2.h"
class Student : private String, private ArrayDb
{
public:
    Student() : String("Null Student"), ArrayDb() {}
    Student(const String & s)
        : String(s), ArrayDb() {}
    Student(int n) : String("Nully"), ArrayDb(n) {}
    Student(const String & s, int n)
        : String(s), ArrayDb(n) {}
    Student(const String & s, const ArrayDb & a)
        : String(s), ArrayDb(a) {}
    Student(const char * str, const double * pd, int n)
        : String(str), ArrayDb(pd, n) {}
    ~Student() {}
    double & operator[](int i);
    const double & operator[](int i) const;
    double Average() const;
// přátelé
    friend ostream & operator<<(ostream & os, const Student & stu);
    friend istream & operator>>(istream & is, Student & stu);
};
#endif
```

Používání metod základních tříd

Soukromá dědičnost omezuje používání metod základních tříd pouze na metody odvozené třídy. Někdy však můžete chtít prostředky základní třídy zpřístupnit veřejně. Například deklarace třídy předpokládá možnost použít funkci `Average()`. Stejně jako u kompozice toho lze dosáhnout pomocí techniky s použitím funkce `ArrayDb::Average()` ve veřejné funkci `Student::Average()` (viz obrázek 13.2). Kompozice vyvolá metodu pomocí objektu:

```
double Student::Average() const
{
    return scores.Average(); // použije ArrayDb::Average()
}
```



Obrázek 13.2. Soukromá dědičnost

Zde ovšem soukromá dědičnost umožňuje vyvolat funkci základní třídy pomocí názvu třídy a operátoru rozlišení:

```
double Student::Average() const
{
    return ArrayDb::Average();
}
```

Vynechání kvalifikátoru `ArrayDb::` by kompilátor pochopil jako volání funkce `Student::Average()`, což by vedlo k naprosto nežádoucí definici rekurzivní funkce. Stručně řečeno, kompozice volá metodu pomocí názvů objektů, zatímco soukromá dědičnost používá název třídy a operátor rozlišení.

Tato technika explicitního označení názvu metody pomocí názvu třídy nefunguje u spřátelených funkcí, protože tyto nejsou součástí třídy. U základní třídy však můžete správné funkce vyvolat pomocí explicitního přetypování. Uvažujte například následující definici spřátelené funkce:

```
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Výsledky studenta: " << (const String &) stu << ":\n";
    os << (const ArrayDb &) stu;
    return os;
}
```

Jestliže plato je objektem třídy `Student`, pak příkaz

```
cout << plato;
```

vyvolá tuto funkci, přičemž `stu` bude referencí na objekt `plato` a `os` referencí na objekt cout. Přetypování

```
os << " Výsledky studenta: " << (const String &) stu << ":\n";
```

explicitně převede stu na referenci na typ objektu třídy `String` a to odpovídá funkci `operator<<(ostream &, const String &)`. Podobně přetypování

```
os << (const ArrayDb &) stu;
```

vyvolá funkci

```
operator<<(ostream &, const ArrayDb &).
```

Reference stu se automaticky nepřevede na referenci na typy `String` nebo `ArrayDb`. Základním důvodem je, že u soukromé dědičnosti nelze referenci nebo ukazateli na objekt základní třídy bez explicitního přetypování přiřadit referenci nebo ukazatel na objekt třídy odvozené.

I kdybyste však v tomto příkladu použili veřejnou dědičnost, museli byste použít explicitní přetypování. Jedním z důvodů je, že bez přetypování by kód

```
os << stu;
```

neodpovídal prototypu spřátelené funkce a vedl by k rekurzívnímu volání. Dalším důvodem je to, že třída používá vícenásobnou dědičnost a kompilátor by nepoznal, na kterou základní třídu by konverze měla být v tomto případě provedena, protože obě možné konverze odpovídají existujícím funkcím `operator <<()`.

Výpis 13.7 obsahuje všechny metody třídy kromě těch, které byly definovány jako vložené v deklaraci třídy.

Výpis 13.7. studenti.cpp

```
// studenti.cpp - třída Student používající soukromou dědičnost
#include "studenti.h"
double Student::Average() const
{
    return ArrayDb::Average();
}
double & Student::operator[](int i)
{
    return ArrayDb::operator[](i);
}
const double & Student::operator[](int i) const
{
    return ArrayDb::operator[](i);
}
// přátelé
ostream & operator<<(ostream & os, const Student & stu)
{
    os << " Výsledky studenta: " << (const String &) stu << ":\n";
    os << (const ArrayDb &) stu;
    return os;
```

```
}
istream & operator>>(istream & is, Student & stu)
{
    is >> (String &) stu;
    return is;
}
```

Použití upravené třídy Student

Opět nadešel čas vyzkoušet novou třídu. Všimněte si, že obě verze třídy Student mají zcela stejné veřejné rozhraní a můžete je tedy vyzkoušet pomocí stejného programu. Jediný rozdíl spočívá v tom, že musíte vložit soubor studenti.h místo souboru studentc.h a program musíte sestavit se souborem studenti.cpp namísto studentc.cpp. Program je obsažen ve výpisu 13.8. Zkompilujte jej společně se soubory studenti.cpp, strng2.cpp a arrayDb.cpp.

Výpis 13.8. use_stui.cpp

```
// use_stui.cpp - použití třídy se soukromou dědičností
// zkompilujte společně se studenti.cpp, strng2.cpp a arraydb.cpp
#include <iostream>
using namespace std;
#include "studenti.h"
void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;
int main()
{
    Student ada[pupils] = {quizzes, quizzes, quizzes};
    int i;
    for (i = 0; i < pupils; i++)
        set(ada[i], quizzes);
    for (i = 0; i < pupils; i++)
    {
        cout << "\n" << ada[i];
        cout << "prumer: " << ada[i].Average() << "\n";
    }
    return 0;
}
void set(Student & sa, int n)
{
   cout << "Zadejte prosim jmeno studenta: ";
   cin >> sa;
   cout << "Zadejte prosim vysledky z " << n << " zkousek:\n";
   for (int i = 0; i < n; i++)
       cin >> sa[i];
   while (cin.get() != '\n')
       continue;
}
```

Zde je ukázka běhu:

```
Zadejte prosim jmeno studenta: Gil Bayts
Zadejte prosim vysledky z 5 zkousek:
92 94 96 93 95
Zadejte prosim jmeno studenta: Pat Roone
Zadejte prosim vysledky z 5 zkousek:
83 89 72 78 95
Zadejte prosim jmeno studenta: Fleur O'Day
Zadejte prosim vysledky z 5 zkousek:
92 89 96 78 64

Vysledky studenta: Gil Bayts:
92 94 96 93 95
prumer: 94

Vysledky studenta: Pat Roone:
83 89 72 78 95
prumer: 83.4

Vysledky studenta: Fleur O'Day
92 89 96 78 64
prumer: 83.8
```

Stejný vstup použitý v předchozím příkladu vede ke stejnému výstupu.

Kompozice nebo soukromá dědičnost?

Jestliže je možné modelovat vztah *má* pomocí kompozice nebo pomocí soukromé dědičnosti, kterou z těchto možností zvolit? Většina programátorů v C++ dává přednost kompozici. Za prvé je jednodušší k pochopení. Když se podíváte na deklaraci třídy, vidíte explicitně pojmenované objekty reprezentující obsažené třídy a váš kód se může na tyto objekty odvolávat názvem. Při použití dědičnosti se tento vztah jeví více abstraktním. Za druhé, dědičnost může způsobit problémy, zvláště dědí-li třída od více základních tříd. Můžete se potýkat s takovými problémy, jako jsou různé základní třídy, které mají stejně pojmenované metody nebo sdílí společného předchůdce. Každopádně při použití kompozice je menší pravděpodobnost, že narazíte na problémy. Kompozice také umožňuje vložit více než jeden podobjekt stejné třídy. Jestliže třída potřebuje tři objekty třídy `String`, můžete pomocí kompozice deklarovat tři samostatné položky tříd `String`. Dědičnost vás ale omezuje na jediný objekt. (Bylo by obtížné jednotlivé objekty rozlišit, jestliže by byly bezejmenné.)

Soukromá dědičnost však nabízí vlastnosti, které kompozice poskytnout nemůže. Předpokládejme například, že třída má chráněné položky, kterými mohou být datové položky nebo členské funkce. Takové položky jsou přístupné odvozeným třídám, ale ne okolnímu světu. Jestliže vložíte takovou třídu pomocí kompozice do jiné třídy, bude nová třída součástí onoho okolního světa a ne odvozené třídy. Nebude tedy mít přístup k chráněným položkám. Pomocí dědičnosti však z nové třídy uděláte třídu odvozenou a bude tedy mít přístup k chráněným položkám.

Další situace, která volá po použití soukromé dědičnosti, je potřeba předefinovat virtuální funkce. Toto oprávnění má opět odvozená třída, ale ne třída kompozice. Při soukromé dědičnosti jsou předefinované funkce použitelné pouze v dané třídě, nikoli veřejně.

Tip

Kompozici používejte k modelování vztahu *má*. Soukromou dědičnost použijte, pokud nová třída potřebuje přistupovat k chráněným položkám původní třídy nebo potřebuje předefinovat virtuální funkce.

Chráněná dědičnost

Chráněná dědičnost je obměnou dědičnosti soukromé. Při uvádění základní třídy se používá klíčové slovo `protected`:

```
class Student : protected String, protected ArrayDb {...};
```

Při chráněné dědičnosti se z veřejných a chráněných položek základní třídy stanou chráněné položky třídy odvozené. Stejně jako při soukromé dědičnosti je rozhraní základní třídy přístupné odvozené třídě, ale ne okolnímu světu. Hlavní rozdíl mezi soukromou a chráněnou dědičností se projeví, odvodíte-li třídu od již odvozené třídy. Při soukromé dědičnosti již tato třída třetí generace nemůže vnitřně používat rozhraní základní třídy. Z veřejných metod základní třídy se totiž v odvozené třídě stanou soukromé a k soukromým položkám a metodám již z další úrovně odvození nelze přímo přistupovat. Při chráněné dědičnosti se z veřejných metod základní třídy stanou v druhé generaci chráněné a ty jsou vnitřně dostupné další generaci odvození.

Tabulka 13.1. shrnuje veřejnou, soukromou a chráněnou dědičnost. Termín *implicitní přetypování na předka* znamená, že můžete ukazateli nebo referenci na základní třídu přiřadit objekt odvozené třídy bez nutnosti použít explicitní přetypování.

Tabulka 13.1. Druhy dědičnosti

Vlastnost	Veřejná dědičnost	Chráněná dědičnost	Privátní dědičnost
Veřejné položky se stanou	veřejnými položkami odvozené třídy	chráněnými položkami odvozené třídy	soukromými položkami odvozené třídy
chráněné položky se stanou	chráněnými položkami odvozené třídy	chráněnými položkami odvozené třídy	soukromými položkami odvozené třídy
soukromé položky jsou	přístupné pouze pomocí rozhraní základní třídy	přístupné pouze pomocí rozhraní základní třídy	přístupné pouze pomocí rozhraní základní třídy
implicitní přetypování na předka	ano	ano (ale pouze v odvozené třídě)	ne

Změna přístupu pomocí klíčového slova using

Použijete-li chráněné nebo soukromé odvození, stanou se z veřejných položek základní třídy položky chráněné nebo soukromé. V posledním příkladu jste viděli, jak může odvozená třída zpřístupnit metodu základní třídy, použije-li tuto metodu ve své metodě:

```
double Student::Average() const
{
    return ArrayDb::Average();
}
```

Kromě zabalení volání funkce do jiné funkce existuje i alternativa, a tou je použití klíčového slova using v deklaraci (podobně jako u prostorů jmen). Tímto způsobem oznámíte, že určitý člen základní třídy může být použit odvozenou třídou, i když se jedná o soukromé odvození. Například v souboru `studenti.h` můžete vynechat deklaraci metody `Student::Average()` a nahradit ji následující deklarací s použitím using:

```
class Student : private String, private ArrayDb
{
public:
    using ArrayDb::Average;
    ...
};
```

Díky takové deklaraci bude metoda `ArrayDb::Average()` přístupná stejně, jako kdyby byla veřejnou metodou třídy `Student`. To znamená, že ze souboru studenti.cpp můžete vypustit definici metody `Student::Average()` a přesto metodu `Average()` používat jako předtím:

```
cout << "prumer: " << ada[i].Average() << "\n";
```

Rozdíl je v tom, že nyní je funkce `ArrayDb::Average()` volána přímo a ne zprostředkovaně voláním metody `Student::Average()`.

Všimněte si, že při deklaraci using se použije pouze název položky – žádné závorky, žádné funkční signatury funkce nebo návratové typy. Kdybyste například chtěli zpřístupnit metodu `ArrayDB::operator[]()` třídě `Student`, umístili byste následující deklaraci using do veřejné části deklarace této třídy:

```
using ArrayDb::operator[];
```

Tímto způsobem by byly zpřístupněny obě verze (`konstantní` i `nekonstantní`). Deklarace using funguje pouze u dědičnosti, u kompozice nefunguje.

Existuje i starší způsob, jak změnit deklaraci metod základní třídy v soukromé odvozené třídě, a tím je umístění názvu metody do veřejné části odvozené třídy. Zde je příklad, jak by to bylo možné udělat:

```
class Student : private String, private ArrayDb
{
public:
    ArrayDb::operator[];  //deklarace se změní na veřejnou,
                          // použije se pouze název
    ...
};
```

Vypadá to jako deklarace using bez použití klíčového slova using. Tento přístup je ale *zavrhován*, to znamená, že má být postupně odstraněn. Pokud tedy váš kompilátor podporuje deklaraci using a chcete-li zpřístupnit metodu základní třídy třídě odvozené, použijte tento způsob.

Šablony tříd

Dědičnost (veřejná, soukromá nebo chráněná) a kompozice nejsou vždy tou správnou odpovědí na požadavek znovupoužitelnosti kódu. Uvažujte například třídy `Stack` (kapitola 9), `Queue` (kapitola 10) nebo `ArrayDb` (tato kapitola). Všechny jsou příkladem kontejnerů, tříd navržených pro uložení jiných objektů nebo datových typů. Třída `Stack` například ukládala hodnoty typu `unsigned long`. Stejně jednoduše byste mohli definovat zásobník pro uložení hodnot typu `double` nebo objektů třídy `String`. Kód by byl identický až na typ uloženého objektu. Bylo by však příjemné, kdybyste místo psaní nových deklarací tříd mohli definovat zásobník obecným způsobem (tedy nezávisle na typu dat) a potom dodat určitý typ jako parametr třídy. Potom byste mohli pomocí stejného obecného kódu vytvářet zásobníky pro různé druhy hodnot. Poprvé jsme u třídy `Stack` tento požadavek v kapitole 9 řešili použitím klíčového slova `typedef`. Ale tento způsob má několik nevýhod. Za prvé, při každé změně typu musíte upravovat hlavičkový soubor. Za druhé, touto technikou můžete vytvořit v každém programu pouze jeden druh zásobníku. To znamená, že pomocí `typedef` nemůžete vytvořit současně dva různé typy a nemůžete tedy pomocí této metody definovat v jednom programu jeden zásobník pro typ `int` a jeden pro typ `String`.

Šablony tříd jazyka C++ poskytují lepší způsob generování obecných deklarací tříd. (C++ původně šablony nepodporoval a vzhledem k tomu, že se šablony od svého zavedení neustále vyvíjí, je možné, že váš kompilátor nebude podporovat všechny zde popsané vlastnosti.) Šablony nabízejí *parametrizované* typy, tedy možnost předat název typu jako parametr pro vytvoření třídy nebo funkce. Například předáním názvu typu `int` šabloně `Queue` vytvoří kompilátor třídu `Queue` pro frontu hodnot typu `int`.

Knihovna standardních šablon jazyka C++ (Standard Template Library, STL), která je částečně probírána v kapitole 15, nabízí výkonné a pružné implementace šablon několika kontejnerových tříd. V této kapitole prozkoumáme podstatně jednodušší návrhy.

Definice šablony třídy

Jako základ šablony použijeme třídu `Stack` z kapitoly 9. Zde je původní deklarace třídy:

```
typedef unsigned long Item;
class Stack
{
private:
    enum {MAX = 10};          // specifická konstanta třídy
    Item items[MAX];          // pole položek zásobníku
    int top;                  // index vrcholu zásobníku
public:
    Stack();
    bool isempty() const;
    bool isfull() const;
    // push() vrátí false, pokud je zásobník již plný, jinak true
    bool push(const Item & item); // přidání položky
    // pop() vrátí false, jestliže je zásobník již prázdný, jinak true
    bool pop(Item & item);          // odebere vrchní položku
};
```

Nyní zaměníme definici třídy `Stack` za definici šablony a členské funkce třídy `Stack` nahradíme členskými funkcemi šablony. Stejně jako u šablon funkcí i šabloně třídy předchází následující kód:

```
template <class Type>
```

Klíčovým slovem `template` kompilátoru říkáte, že budete definovat šablonu. Část v lomených závorkách je podobná seznamu parametrů funkce. Klíčové slovo `class` můžete zaměnit za název typu proměnné a slovo *Type* je název této proměnné.

Použití slova `class` na tomto místě neznamená, že *Type* musí být třídou, pouze znamená, že *Type* slouží jako specifikátor obecného typu, který bude nahrazen při použití šablony. Novější implementace umožňují v tomto kontextu použít méně zavádějící klíčové slovo `typename` namísto slova class:

```
template <typename Type> // novější možnost
```

Název obecného typu v místě *Type* si můžete zvolit; pravidla pro názvy jsou stejná jako pro ostatní identifikátory. Většinou se používá *T* nebo *Type*, my budeme používat to druhé. Při vyvolání šablony bude `Type` nahrazen konkrétní hodnotou typu, například `int` nebo `String`. V definici šablony používejte název obecného typu k identifikaci typu dat uložených v zásobníku. V případě třídy `Stack` by to znamenalo použití názvu *Type* všude tam, kde byl v původní deklaraci použit identifikátor `Item` vytvořený pomocí `typedef`. Například

```
Item items[MAX]; // obsahuje položky zásobníku
```

se změní na

```
Type items[MAX]; // obsahuje položky zásobníku
```

Podobně můžete nahradit metody původní třídy členskými funkcemi šablony. Každé hlavičce funkce bude předcházet stejné sdělení šablony:

```
template <class Type>
```

Opět nahraďte identifikátor `Item` názvem obecného typu `Type`. Ještě budete muset změnit kvalifikátor třídy `Stack::` na kvalifikátor šablony `Stack<Type>::`. Například:

```
bool Stack::push(const Item & item)
{
...
}
```

se změní na

```
template <class Type>        // nebo template <typename Type>
bool Stack<Type>::push(const Type & item)
{
...
}
```

Jestliže budete metodu definovat v deklaraci třídy (vložená definice), můžete klíčové slovo template a kvalifikátor třídy vynechat.

Ve výpisu 13.9. je kombinace šablon tříd a členských funkcí. Je důležité uvědomit si, že tyto šablony nejsou definicemi tříd ani členských funkcí. Spíše se jedná o instrukce pro kompilátor C++, jak vygenerovat definice tříd a členských funkcí. Konkrétní realizace šab-

lony, jako například třída zásobníku pro správu objektů třídy `String`, se nazývá *vytvoření instance* nebo *specializace*. Pokud nemáte kompilátor, který implementuje nové klíčové slovo `export`, nebude uložení členských funkcí šablon do odděleného souboru fungovat. Protože šablony nejsou funkce, nemohou být zkompilovány odděleně. Šablony musí být použity podle požadavků na konkrétní vytvoření jejich instancí. Nejjednodušším způsobem jak toho dosáhnout, je umístit všechny informace týkající se šablon do hlavičkového souboru a uložit tento hlavičkového do souboru, který bude šablony používat.

Výpis 13.9. stacktp.h

```
// stacktp.h – šablona zásobníku
template <class Type>
class Stack
{
private:
    enum {MAX = 10}; // specifická konstanta třídy
    Type items[MAX]; // obsahuje položky zásobníku
    int top;         // index vrchní položky zásobníku
public:
    Stack();
    bool isempty();
    bool isfull();
    bool push(const Type & item); // přidá položku do zásobníku
    bool pop(Type & item);        // odebere vrchní položku
};
template <class Type>
Stack<Type>::Stack()
{
    top = 0;
}
template <class Type>
bool Stack<Type>::isempty()
{
    return top == 0;
}
template <class Type>
bool Stack<Type>::isfull()
{
    return top == MAX;
}
template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
```

```
}
template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

Pokud váš kompilátor neimplementuje nové klíčové slovo `export`, můžete umístit definice metod šablon do samostatného souboru za předpokladu, že před každou definicí použijete klíčové slovo `export`:

```
export template <class Type>
Stack <Type>::Stack()
{
   top = 0;
}
```

V tom případě byste mohli použít stejnou konvenci používanou pro obyčejné třídy:

1. Umístěte deklaraci šablony třídy do hlavičkového souboru a pomocí direktivy `#include` zpřístupněte deklaraci programu.
2. Umístěte definice metod šablony třídy do zdrojového souboru programu a zpřístupněte definice programu například pomocí projektového souboru.

Použití šablony třídy

Pouhé vložení šablony do programu šablonu třídy nevygeneruje. Musíte požádat o instanci. K tomu je potřeba deklarovat objekt typu šablona třídy a nahradit název obecného typu konkrétním požadovaným typem. Zde je příklad vytvoření dvou zásobníků, jeden pro hodnoty typu `int` a druhý pro objekty třídy `String`:

```
Stack<int> kernels;       // vytvoří zásobník pro typ int
Stack<String> colonels;  // vytvoří zásobník objektů třídy String
```

Jakmile kompilátor uvidí tyto dvě deklarace, vytvoří podle šablony `Stack<Type>` dvě samostatné deklarace tříd a dvě samostatné sady členských funkcí. Deklarace `Stack<int>` nahradí `Type` typem `int`, zatímco deklarace `Stack<String>` nahradí `Type` typem `String`. Samozřejmě použité algoritmy musí být v souladu s danými typy. Třída Stack například předpokládá, že je možné přiřazovat jednu položku druhé. Tento předpoklad platí pro základní typy, struktury a třídy (pokud neučiníte operátor přiřazení soukromým), ale ne pro pole.

Obecné identifikátory typů, jako například `Type` v předešlém příkladu, se nazývají *typové parametry*, což znamená, že se chovají podobně jako proměnná, ale místo přiřazení číselné hodnoty jim přiřadíte typ. Ve výše uvedené deklaraci `kernels` má tedy `typový` parametr Type hodnotu `int`.

Všimněte si, že požadovaný typ musíte dodat explicitně. To je rozdíl oproti šablonám běžných funkcí, u kterých kompilátor pomocí typů parametrů pozná, jakou funkci má vygenerovat:

```
Template <class T>
void simple(T t) { cout << t << '\n';}
...
simple(2);     // vygeneruje funkci void simple(int)
simple("two") // vygeneruje funkci void simple(char *)
```

Ve výpisu 13.10. je upravený původní program pro test zásobníku (výpis 9.13), který místo hodnot typu `unsigned long` používá pro identifikaci objednávky řetězec. Protože používá naši třídu `String`, zkompilujte ho společně se souborem `strng2.cpp`.

Výpis 13.10. stacktem.cpp

```
// stacktem.cpp — test šablony pro zásobník
  // zkompilovat společně s strng2.cpp
  #include <iostream>
  using namespace std;
  #include <cctype>
  #include "stacktp.h"
  #include "strng2.h"
  int main()
  {
      Stack<String> st; // vytvoří prázdný zásobník
      char c;
      String po;
      cout << "Pro pridani objednavky zadejte \"P\",\n"
            << "pro zpracovani \"Z\" a pro ukonceni \"K\".\n";
      while (cin >> c && toupper(c) != 'K')
      {
          while (cin.get() != '\n')
              continue;
          if (!isalpha(c))
          {
              cout << '\a';
              continue;
          }
          switch(c)
          {
              case 'P':
              case 'p': cout << "Zadejte cislo objednavky: ";
                          cin >> po;
                          if (st.isfull())
                              cout << "zasobnik je jiz plny\n";
                          else
                              st.push(po);
                          break;
              case 'Z':
              case 'z': if (st.isempty())
                              cout << "zasobnik je jiz prazdny\n";
```

```
                else {
                    st.pop(po);
                    cout << "Objednavka c. " << po << " byla
zpracovana\n";
                    break;
                }
            }
            cout << "Pro pridani objednavky zadejte \"P\".\n"
                << "pro zpracovani \"Z\" a pro ukonceni \"K\".\n";
        }
        cout << "Nashledanou\n";
        return 0;
}
```

Kompatibilita:

Jestliže vaše implementace nenabízí soubor cctype, použijte starší hlavičkový soubor ctype.h.

Zde je ukázka běhu:

```
Pro pridani objednavky zadejte "P".
pro zpracovani "Z" a pro ukoceni "K".
P
Zadejte cislo objednavky: red911porsche
Pro pridani objednavky zadejte "P".
pro zpracovani "Z" a pro ukoceni "K".
P
Zadejte cislo objednavky: green328bmw
Pro pridani objednavky zadejte "P".
pro zpracovani "Z" a pro ukoceni "K".
P
Zadejte cislo objednavky: silver747boeing
Pro pridani objednavky zadejte "P".
pro zpracovani "Z" a pro ukoceni "K".
Z
Objednavka c. silver747boeing zpracovana
Pro pridani objednavky zadejte "P".
Pro zpracovani "Z" a pro ukoceni "K".
Z
 Objednavka c. green328bmw zpracovana
Pro pridani objednavky zadejte "P".
Pro zpracovani "Z" a pro ukoceni "K".
Z
 Objednavka c. red911porsche zpracovana
Pro pridani objednavky zadejte "P".
Pro zpracovani "Z" a pro ukoceni "K".
Z
zasobnik je jiz prazdny
Pro pridani objednavky zadejte "P".
Pro zpracovani "Z" a pro ukoceni "K".
K
Nashledanou
```

Bližší pohled na šablonu třídy

Jako typ šablony `Stack<Type>` můžete použít vestavěný typ nebo objekt třídy. A co ukazatel? Můžete ve výpisu 13.10 použít například ukazatel na typ `char` namísto objektu třídy `String`? Konec konců představují ukazatele vestavěný typ pro správu řetězců. Odpověď je, že zásobník ukazatelů vytvořit můžete, ale bez větších úprav programu nebude dobře fungovat. Kompilátor dokáže takovou třídu vytvořit, vaším úkolem však je zajistit, aby byla rozumně použita. Podívejme se, proč takový zásobník ukazatelů nebude dobře fungovat s programem z výpisu 13.10 a potom se podíváme na příklad, ve kterém je zásobník ukazatelů užitečný.

Nesprávné použití zásobníku ukazatelů

Nyní se v rychlosti podíváme na tři jednoduché, chybné pokusy upravit program z výpisu 13.10 pro použití zásobníku ukazatelů. Ponaučení z těchto pokusů je, že byste měli mít vždy na paměti návrh šablony a ne ji pouze slepě používat. Všechny tři začínají naprosto správným voláním šablony:

```
Stack<char *> st; // vytvoří zásobník ukazatelů na char
```

První verze potom nahradí deklaraci

```
String po;
```

deklarací ukazatele

```
char * po;
```

Myšlenkou je použít pro vstup z klávesnice ukazatele na typ `char` namísto objektu třídy `String`. Tento pokus ihned selže, protože pouhé vytvoření ukazatele nevytvoří místo pro vstupní řetězce.

Druhá verze nahradí deklaraci

```
String po;
```

deklarací

```
char po[40];
```

Touto deklarací přidělíte místo pro vstupní řetězec. Navíc `po` je ukazatel na typ `char`, může tedy být vložen do zásobníku. Ale pole je v rozporu s předpoklady, na kterých je vytvořena metoda `pop()`:

```
template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[-top];
        return true;
    }
    else
        return false;
}
```

Za prvé musí referenční proměnná `item` odkazovat na nějakou l-hodnotu, nikoli na název pole. Za druhé se v kódu předpokládá možnost přiřazení proměnné `item`. I kdyby proměnná `item` mohla odkazovat na pole, nemohli byste jí přiřadit název pole. Takže tento postup také selže.

Verze tři nahradí deklaraci

```
String po;
```

deklarací

```
char * po = new char[40];
```

Tato deklarace také přidělí místo pro vstupní řetězec. Navíc `po` je proměnná a je tedy kompatibilní s kódem metody `pop()`. Zde však narážíme na nejzásadnější problém. Existuje pouze jedna proměnná `po` a ta vždy ukazuje na stejné místo v paměti. Je sice pravda, že obsah paměti se změní při každém načtení nového řetězce, ale každá operace vložení vloží do zásobníku naprosto stejnou adresu. Když tedy ze zásobníku odeberete položku, vrátí se vám vždy stejná adresa, která vždy odkazuje do paměti na naposledy čtený řetězec. Konkrétně, zásobník neukládá nové řetězce tak, jak jsou průběžně načítány a jeho použití nemá smysl.

Správné použití zásobníku ukazatelů

Jeden z možných způsobů použití zásobníku ukazatelů je ve volajícím programu vytvářejícím pole ukazatelů, kde každý ukazatel ukazuje na jiný řetězec. Uložení těchto ukazatelů do zásobníku je v tomto případě rozumné, neboť každý ukazatel odkazuje na jiný řetězec. Všimněte si, že odpovědnost za vytváření oddělených ukazatelů má volající program, ne zásobník. Úkolem zásobníku je správa ukazatelů, nikoli jejich vytváření.

Předpokládejme například, že musíme simulovat následující situaci. Panu Novákovi kdosi doručil vozík se složkami. Jestliže je šuplík pana Nováka obsahující nezpracované složky prázdný, pak pan Novák vezme z vozíku horní složku a vloží ji do šuplíku pro nezpracované složky. Jestliže je šuplík plný, vezme z šuplíku vrchní složku, zpracuje ji a uloží do šuplíku pro zpracované složky. Pokud není šuplík ani prázdný ani plný, může pan Novák buď zpracovat vrchní nezpracovanou složku nebo vzít další z vozíku a vložit do šuplíku s nezpracovanými složkami. V okamžiku, který pan Novák tajně považuje tak trochu za ztřeštěné sebevyjádření, si hodí korunou, aby se mohl rozhodnout pro jeden z uvedených postupů. Prozkoumáme účinky jeho metody na původní pořadí složek.

Tuto situaci můžeme modelovat pomocí pole ukazatelů na řetězce reprezentující složky ve vozíku. Každý řetězec bude obsahovat jméno osoby, kterou složka popisuje. Pro reprezentaci šuplíku s nezpracovanými složkami můžeme použít zásobník a další pole ukazatelů pro reprezentaci šuplíku s vyřízenými složkami. Přidání složky do šuplíku s nevyřízenými složkami vyjadřuje vložení ukazatele z prvního pole do zásobníku, zatímco zpracování složky se vyjádří odebráním položky ze zásobníku a jejím přidáním do šuplíku s vyřízenými složkami.

Vzhledem k tomu, že je důležité prozkoumat všechny aspekty tohoto problému, bylo by užitečné, kdybychom mohli vyzkoušet zásobníky s různou velikostí. Ve výpisu 13.11 je šablona `Stack<Type>` trochu předefinovaná, aby konstruktor měl parametr určující velikost. K tomu je zapotřebí použít vnitřně dynamické pole, takže třída bude nyní potřebovat i destruktor, kopírovací konstruktor a operátor přiřazení. Několik metod v definici je vložených a díky tomu je kód kratší.

Výpis 13.11. stcktp1.h

```
// stcktp1.h – modifikovaná šablona Stack
template <class Type>
class Stack
{
private:
    enum {MAX = 10}; // specifická konstanta třídy
    int stacksize;
    Type * items;    // obsahuje položky zásobníku
    int top;         // index vrchní položky zásobníku
public:
    explicit Stack(int ss = MAX);
    Stack(const Stack & st);
    ~Stack() { delete [] items; }
    bool isempty() { return top == 0; }
    bool isfull() { return top == stacksize; }
    bool push(const Type & item); // přidá položku do zásobníku
    bool pop(Type & item);        // odebere vrchní položku
    Stack & operator=(const Stack & st);
};
template <class Type>
Stack<Type>::Stack(int ss) : stacksize(ss), top(0)
{
    items = new Type [stacksize];
}
template <class Type>
Stack<Type>::Stack(const Stack & st)
{
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
}
template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < stacksize)
    {
        items[top++] = item;
        return true;
    }
```

```
    else
        return false;
}
template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
template <class Type>
Stack<Type> & Stack<Type>::operator=(const Stack<Type> & st)
{
    if (this == &st)
        return *this;
    delete [] items;
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
    return *this;
}
```

Kompatibilita:

Některé implementace neznají klíčové slovo `explicit`.

Všimněte si, že v prototypu operátoru přiřazení je návratový typ deklarován jako reference na `Stack`, zatímco skutečná definice šablony funkce identifikuje typ jako `Stack<Type>`. První tvar je zkratkou druhého, ale lze ho použít pouze v oblasti platnosti třídy. Můžete tedy použít typ `Stack` v deklaraci šablony a v definicích šablon funkcí, ale vně třídy při identifikaci návratových typů a použití operátoru rozlišení musíte použít plný tvar `Stack<Type>`.

Program ve výpisu 13.12 používá novou šablonu zásobníku pro implementaci simulace práce pana Nováka. Pro generování náhodných čísel používá funkce `rand()`, `srand()` a `time()`stejným způsobem jako předchozí simulace. Zde náhodné generování hodnot 0 a 1 simuluje hození mincí.

Výpis 13.12. stkoptr1.cpp

```
// stkoptr1.cpp - test zásobníku ukazatelů
#include <iostream>
using namespace std;
#include <cstdlib>  // pro rand(), srand()
#include <ctime>    // pro time()
#include "stcktp1.h"
const int Stacksize = 4;
const int Num = 10;
int main()
{
    srand(time(0)); // nastaví rand()
    cout << "Zadejte prosim velikost zasobniku: ";
    int stacksize;
    cin >> stacksize;
    Stack<char *> st(stacksize); // vytvoří prázdný zásobník se čtyřmi
položkami
    char * in[Num] = {
            " 1: Hank Gilgamesh", " 2: Kiki Ishtar",
            " 3: Betty Rocker", " 4: Ian Flagranti",
            " 5: Wolfgang Kibble", " 6: Portia Koop",
            " 7: Joy Almondo", " 8: Xaverie Paprika",
            " 9: Juan Moore", "10: Misha Mache"
            };
    char * out[Num];
    int processed = 0;
    int nextin = 0;
    while (processed < Num)
    {
        if (st.isempty())
            st.push(in[nextin++]);
        else if (st.isfull())
            st.pop(out[processed++]);
        else if (rand() % 2 && nextin < Num) // 50% šance
            st.push(in[nextin++]);
        else
            st.pop(out[processed++]);
    }
    for (int i = 0; i < Num; i++)
        cout << out[i] << "\n";
    cout << "Nashledanou\n";
    return 0;
}
```

Kompatibilita:

Některé implementace vyžadují hlavičkové soubory `stdlib.h` a `time.h` namísto `cstdlib` a `ctime`.

Následují dva běhy programu. Všimněte si, že konečné řazení složek se může případ od případu trochu lišit, i když velikost zásobníku zůstane nezměněna:

```
Zadejte prosim velikost zasobniku: 5
    2: Kiki Ishtar
    1: Hank Gilgamesh
    3: Betty Rocker
    5: Wolfgang Kibble
    4: Ian Flagranti
    7: Joy Almondo
    9: Juan Moore
    8: Xaverie Paprika
    6: Portia Koop
   10: Misha Mache
   Nashledanou

Zadejte prosim velikost zasobniku: 5
    3: Betty Rocker
    5: Wolfgang Kibble
    6: Portia Koop
    4: Ian Flagranti
    8: Xaverie Paprika
    9: Juan Moore
   10: Misha Mache
    7: Joy Almondo
    2: Kiki Ishtar
    1: Hank Gilgamesh
   Nashledanou
```

Poznámky k programu

Řetězce samotné se nikdy nepřesouvají. Vložení řetězce do zásobníku vytvoří ve skutečnosti nový ukazatel na existující řetězec. To znamená, že vytvoří ukazatel, jehož hodnotou je adresa již existujícího řetězce. Odebrání řetězce ze zásobníku zkopíruje tuto hodnotu do pole `out`.

Jaký účinek má destruktor zásobníku na řetězce? Žádný. Konstruktor třídy vytvoří pomocí operátoru `new` pole pro uložení ukazatelů. Destruktor zruší toto pole, nikoli řetězce, na které prvky pole ukazovaly.

Příklad šablony pole a netypové parametry

Šablony jsou často používány pro třídy kontejnerů, protože myšlenka typových parametrů dobře ladí s potřebou aplikovat shodný způsob uložení na různé typy. Touha nabídnout třídám kontejnerů znovupoužitelný kód byla hlavní motivací pro zavedení šablon. Podívejme se tedy na další příklad odhalující další aspekty návrhu a používání šablon.

Začneme jednoduchou šablonou pro pole, která umožní zadat velikost pole. Jedna z technik, kterou použila poslední verze třídy `Stack`, spočívala v použití dynamického pole ve třídě a parametru konstruktoru pro určení počtu prvků pole. Jinou možností je určení velikosti běžného pole pomocí parametru šablony. Způsob provedení je ukázán ve výpisu 13.13.

Výpis 13.13. arraytp.h

```
//arraytp.h — šablona pole
#include <iostream>
using namespace std;
#include <cstdlib>
template <class T, int n>
class ArrayTP
{
private:
    T ar[n];
public:
    ArrayTP();
    explicit ArrayTP(const T & v);
    virtual T & operator[](int i);
    virtual const T & operator[](int i) const;
};
template <class T, int n>
ArrayTP<T,n>::ArrayTP()
{
    for (int i = 0; i < n; i++)
        ar[i] = 0;
}
template <class T, int n>
ArrayTP<T,n>::ArrayTP(const T & v)
{
    for (int i = 0; i < n; i++)
        ar[i] = v;
}
template <class T, int n>
T & ArrayTP<T,n>::operator[](int i)
{
    if (i < 0 || i >= n)
    {
        cerr << "Prekroceny meze pole: " << i
            << " se nachazi mimo rozsah\n";
        exit(1);
    }
    return ar[i];
}
template <class T, int n>
const T & ArrayTP<T,n>::operator[](int i) const
{
    if (i < 0 || i >= n)
```

```
        {
            cerr << "Error in array limits: " << i
                << " is out of range\n";
            exit(1);
        }
        return ar[i];
    }
```

Všimněte si hlavičky šablony:

```
template <class T, int n>
```

Klíčové slovo `class` identifikuje T jako typový parametr. Slovo `int` identifikuje proměnnou n typu `int`. Tento druhý typ parametru, který se nechová jako obecný název pro typ ale specifikuje konkrétní typ, se nazývá *netypový* nebo *výrazový* parametr. Předpokládejme následující deklaraci:

```
ArrayTP<double, 12> eggweights;
```

Podle této deklarace kompilátor definuje třídu nazvanou `ArrayTP<double,12>` a vytvoří její objekt `eggweights`. Při definování třídy kompilátor nahradí `T` za `double` a `n` za hodnotu `12`.

Netypové parametry mají některá omezení. Netypový parametr může být celočíselný typ, výčet, reference nebo ukazatel. Tedy `double m` je nepřípustné, ale `double & rm` nebo `double * pm` je již povoleno. V kódu šablony také nemůžete změnit hodnotu parametru nebo přiřadit jeho adresu. Takže v šabloně `ArrayTP` by výrazy `n++` nebo `&n` nebyly povoleny. Při vytváření instance šablony by hodnotou netypového parametru měl být konstantní výraz.

Tato metoda nastavování velikosti pole má oproti konstruktoru použitému ve třídě `Stack` jednu výhodu. Konstruktor používá paměť haldy spravovanou operátory `new` a `delete`, zatímco netypový parametr používá paměť zásobníku udržovanou pro automatické proměnné. Provádění programu je v tomto případě rychlejší, obzvláště pokud máte hodně malých polí.

Hlavní nevýhodou netypového parametru je, že pole si pro každou velikost generuje svou vlastní šablonu. To znamená, že deklarace

```
ArrayTP<double, 12> eggweights;
ArrayTP(double, 13> donuts;
```

vytvoří dvě oddělené deklarace třídy. Ale deklarace

```
Stack<int> eggs(12);
Stack<int> dunkers(13);
```

vygenerují pouze jednu deklaraci třídy a informace o velikosti jsou předány konstruktoru této třídy.

Dalším rozdílem je větší univerzálnost konstruktoru, protože velikost pole je uložena v položce třídy a není do definice pevně zakódovaná. Díky tomu je například možné definovat přiřazení pole jedné velikosti poli jiné velkosti nebo vytvořit třídu umožňující měnit velikost pole.

Použití šablony u skupiny tříd

Předpokládejme, že máte skupinu tříd. Konkrétně máte jednu základní třídu a několik tříd od ní odvozených. Dále předpokládejme několik objektů těchto tříd, které chcete do pole uložit. Jedním problémem je, že všechny prvky pole musí být stejného typu, ale objekt odvozené třídy se typově liší od objektu základní třídy nebo od objektu jiné odvozené třídy. Dokonce ani nemusí mít stejnou velikost jako objekt základní třídy, takže obelhání kompilátoru ve věci typu nefunguje. Stručně řečeno, pole představuje nástroj vhodný pro homogenní prvky (všechny prvky jsou stejného typu), ale skupina tříd představuje heterogenní kolekci objektů (objekty různých typů).

Právě v takových situacích přijde vhod vztah *je* a virtuální funkce. Jak jste viděli v programu z výpisu 12.8, můžete vytvořit pole ukazatelů na objekty základní třídy. Veřejná dědičnost znamená, že ukazateli na objekt základní třídy můžete přiřadit jakýkoli objekt třídy odvozené, takže takové pole může obsahovat adresy objektů různých typů. Velikosti objektů se mohou lišit, ale velikost všech ukazatelů je stejná. Navíc, pokud pomocí těchto ukazatelů vyvoláte metody třídy, virtuální funkce zajistí, aby ukazatel základní třídy na objekt odvozené třídy vyvolal metody objektu odvozené třídy.

Vyzkoušejme si to pomocí šablony `ArrayTP`. Nejdříve budete potřebovat skupinu tříd. Pro názornost použijeme jednoduché třídy. Ve výpisu 13.14 jsou definovány třídy `Worker`, `Waiter`, `Singer` a `Greeter`. Třída `Worker` obsahuje jméno a identifikační číslo. Do třídy `Waiter`, odvozené od třídy `Worker`, je přidána položka vyjadřující hodnocení elegance. Do třídy `Singer`, odvozené od třídy `Worker`, je přidán popis hlasového rozsahu. A do třídy `Greeter`, odvozené od třídy `Worker`, je přidáno hodnocení dobré nálady.

Výpis 13.14. worker.h

```
// worker.h - třídy pracujících
#include "strng2.h"
class Worker // abstraktní základní třída
{
private:
    String fullname;
    long id;
public:
    Worker() : fullname("no one"), id(0L) {}
    Worker(const String & s, long n)
            : fullname(s), id(n) {}
    virtual ~Worker() = 0; // čistě virtuální destruktor
    virtual void Set();
    virtual void Show() const;
};
class Waiter : public Worker
{
private:
    int panache;
public:
    Waiter() : Worker(), panache(0) {}
    Waiter(const String & s, long n, int p = 0)
```

```
                  : Worker(s, n), panache(p) {}
        Waiter(const Worker & wk, int p = 0)
                  : Worker(wk), panache(p) {}
        void Set();
        void Show() const;
};
class Singer : public Worker
{
protected:
        enum {jiny, alt, kontralt, sopran,
                              bas, bariton, tenor};
        enum {Vtypes = 7};
private:
        static char *pv[Vtypes]; // řetězce pro typy hlasu
        int voice;
public:
        Singer() : Worker(), voice(other) {}
        Singer(const String & s, long n, int v = other)
                  : Worker(s, n), voice(v) {}
        Singer(const Worker & wk, int v = other)
                  : Worker(wk), voice(v) {}
        void Set();
        void Show() const;
};
class Greeter : public Worker
{
private:
        int cheer;
public:
        Greeter() : Worker(), cheer(0) {}
        Greeter(const String & s, long n, int c = 0)
                  : Worker(s, n), cheer(c) {}
        Greeter(const Worker & wk, int c = 0)
                  : Worker(wk), cheer(c) {}
        void Set();
        void Show() const;
};
```

V tomto uspořádání patří každý k nějakému konkrétnímu typu pracovníka (číšník, zpěvák, zdravič), není obecným pracovníkem. Z třídy `Worker` lze tedy udělat abstraktní základní třídu. Vzpomeňte si, že to vyžaduje alespoň jednu čistě virtuální funkci. V této třídě se této role ujme destruktor. (Protože třída `Worker` je základní třídou, měla by mít virtuální destruktor i tehdy, pokud nebude nic dělat. Jestliže má být třída `Worker` zároveň abstraktní základní třídou, můžete z destruktoru učinit čistě virtuální funkci.)

Při vytváření čistě virtuálního destruktoru je jedna zvláštnost. Čistě virtuální funkce je obvykle implementována v odvozené třídě, ale destruktor musí být implementován ve své třídě. V případě čistě virtuálního destruktoru tedy musíte dodat implementaci základní třídy, což výpis 13.15 činí. Deklarace destruktoru jako čistě virtuální funkce však stačí k tomu, aby třída byla abstraktní a aby se zamezilo vytváření objektů třídy `Worker`.

Za povšimnutí dále stojí, že některé konstruktory odvozené třídy volají konstruktor `Worker (const Worker &)`, i když jste žádný takový konstruktor nedefinovali. Vzpomeňte si ale, že kompilátor vygeneruje tento konstruktor (kopírovací konstruktor), pokud tak neučiníme sami. Pro tento konkrétní popis implicitní konstruktor stačí.

Dále musíte definovat ty funkce, které již nemají vložené definice. Tyto informace jsou obsaženy ve výpisu 13.15. Inicializováno je rovněž pole ukazatelů na data, které má platnost v rozsahu třídy a je používáno třídou `Singer`. V zájmu stručnosti není v tomto příkladu příliš kontrolována platnost vstupních dat.

Výpis 13.15. worker.cpp

```
// worker.cpp — metody tříd pracovníků
#include "worker.h"
#include <iostream>
using namespace std;
// metody třídy Worker
    // musí implementovat destruktor, i když je čistě virtuální
Worker::~Worker() {}
void Worker::Set()
{
    cout << "Zadejte jmeno pracovnika: ";
    cin >> fullname;
    cout << "Zadejte ID pracovnika: ";
    cin >> id;
    while (cin.get() != '\n')
        continue;
}
void Worker::Show() const
{
    cout << "Jmeno: " << fullname << "\n";
    cout << "ID pracovnika: " << id << "\n";
}
// metody třídy Waiter
void Waiter::Set()
{
    Worker::Set();
    cout << "Zadejte rating mrstnosti cisnika: ";
    cin >> panache;
    while (cin.get() != '\n')
        continue;
}
void Waiter::Show() const
{
    cout << "Kategorie: cisnik\n";
    Worker::Show();
    cout << "Rating mrstnosti: " << panache << "\n";
}
// metody třídy Singer
char * Singer::pv[] = {"jiny", "alt", "kontralt",
```

```
        "sopran", "bas", "bariton", "tenor"};
void Singer::Set()
{
    Worker::Set();
    cout << "Zadejte cislo urcujici rozsah vokalu zpevaka:\n";
    int i;
    for (i = 0; i < Vtypes; i++)
    {
        cout << i << ": " << pv[i] << " ";
        if ( i % 4 == 3)
            cout << '\n';
    }
    if (i % 4 != 0)
        cout << '\n';
    cin >> voice;
    while (cin.get() != '\n')
        continue;
}
void Singer::Show() const
{
    cout << "Kategorie: zpevak\n";
    Worker::Show();
    cout << "Rozsah vokalu: " << pv[voice] << "\n";
}
// metody třídy Greeter
void Greeter::Set()
{
    Worker::Set();
    cout << "Zadejte rating zdravicova veseli: ";
    cin >> cheer;
    while (cin.get() != '\n')
        continue;
}
void Greeter::Show() const
{
    cout << "Kategorie: zdravic\n";
    Worker::Show();
    cout << "Rating veseli: " << cheer << "\n";
}
```

A nakonec výpis 13.16 obsahuje program pro ověření použití pole ukazatelů základní třídy na skupinu tříd. Představuje najímání prvních zaměstnanců kavárnou Lola.

Kompatibilita:

Pokud váš systém šablony nepodporuje, můžete na třídu `Worker` použít obyčejné pole ukazatelů. Sledujte pokyny ve výpisu programu.

Výpis 13.16. workarr.cpp

```
// workarr.cpp — array of workers
// zkompilovat společně s worker.cpp, strng2.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "worker.h"
#include "arraytp.h" // vynechejte, pokud nejsou implementovány šablony
const int SIZE = 5;
int main()
{
    ArrayTP<Worker *, SIZE> lolas;
    // pokud neexistují šablony, vynechejte předchozí řádek
    // a použijte následující
    // Worker * lolas[SIZE];
    cout << "Zadejte zamestnance kavarny Lola: \n";
    int ct;
    for (ct = 0; ct < SIZE; ct++)
    {
        char choice;
        cout << "Zadejte kategorii zamestnance:\n"
            << "d: zdravic c: cisnik z: zpevak "
            << "k: konec\n";
        cin >> choice;
        while (strchr("dczk", choice) == NULL)
        {
            cout << "Zadejte prosim d, c, z nebo k: ";
            cin >> choice;
        }
        if (choice == 'q')
            break;
        switch(choice)
        {
            case 'd': lolas[ct] = new Greeter;
                      break;
            case 'c': lolas[ct] = new Waiter;
                      break;
            case 'z': lolas[ct] = new Singer;
                      break;
        }
        cin.get();
        lolas[ct]->Set();
    }
    cout << "\nZde jsou vasi zamestnanci:\n";
    int i;
    for (i = 0; i < ct; i++)
    {
        cout << '\n';
        lolas[i]->Show();
    }
    for (i = 0; i < ct; i++)
```

```
    delete lolas[i];
  return 0;
}
```

Zde je ukázka běhu:

```
Zadejte zamestnance kavarny Lola:
Zadejte kategorii zamestnance:
d: zdravic c: cisnik z: zpevak k: konec
c
Zadejte jmeno pracovnika: Fran Godot
Zadejte ID pracovnika: 1004
Zadejte rating mrstnosti cisnika: 6
Zadejte kategorii zamestnance:
d: zdravic c: cisnik z: zpevak k: konec
z
Zadejte jmeno pracovnika: Igor Tunefree
Zadejte ID pracovnika: 1009
Zadejte rozsah vokalu zpevaka:
0: jiny 1: alt 2: kontralt 3: sopran
4: bas 5: bariton 6: tenor
4
Zadejte kategorii zamestnance:
d: zdravic c: cisnik z: zpevak k: konec
d
Zadejte jmeno pracovnika: Hap Gladfoote
Zadejte ID pracovnika: 1003
Zadejte rating veseli zdravice: 8
Zadejte kategorii zamestnance:
d: zdravic c: cisnik z: zpevak k: konec
c
Zadejte jmeno pracovnika: Walt Wiltcress
Zadejte ID pracovnika: 1022
Zadejte rating mrstnosti cisnika: 5
Zadejte kategorii zamestnance:
d: zdravic c: cisnik z: zpevak k: konec
k
Zde jsou vasi zamestnanci:
Kategorie: cisnik
Jmeno: Fran Godot
ID pracovnika: 1004
Rating mrstnosti: 6
Kategorie: zpevak
Jmeno: Igor Tunefree
ID pracovnika: 1009
Rozsah vokalu: bass
Kategorie: zdravic
Jmeno: Hap Gladfoote
ID pracovnika: 1003
Rating veseli: 8
Kategorie: cisnik
Jmeno: Walt Wiltcress
ID pracovnika: 1022
Rating mrstnosti: 5
```

Poznámky k programu

Standardní funkce jazyka ANSI C `strchr(const char * str, char ch)` hledá výskyt znaku `ch` v řetězci `str`. Pokud ho najde, vrátí adresu prvního výskytu, jestliže existuje, jinak vrátí nulový ukazatel. Kód

```
while (strchr("ewsq", choice) == NULL)
```

tedy nabízí snadný způsob kontroly správnosti zadaného znaku.

Příkaz `switch` přiřadí adresu nového objektu do jednoho z prvků pole ukazatelů na třídu `Worker`.

```
switch(choice)
{
    case 'd': lolas[ct] = new Greeter;
                 break;
    case 'c': lolas[ct] = new Waiter;
               break;
    case 'z': lolas[ct] = new Singer;
                 break;
}
```

V závislosti na uživatelově vstupu může ukazatel ukazovat na objekt třídy `Singer`, `Waiter` nebo `Greeter`. Protože ukazatel na objekt základní třídy může ukazovat na objekt kterékoli odvozené třídy, není potřeba žádné přetypování.

Protože metody `set()` a `show()` jsou definovány jako virtuální, bude při volání těchto funkcí pomocí ukazatele na třídu `Worker` vždy vyvolána funkce odpovídající objektu, na který ukazatel ukazuje:

```
lolas[ct]->Set();
...
lolas[i]->Show();
```

Tímto způsobem můžete pomocí homogenní kolekce ukazatelů spravovat heterogenní kolekci objektů. Toto je ukázka polymorfismu v akci: jedno funkční volání může v závislosti na kontextu aktivovat různé funkce.

Všimněte si následujícího kódu:

```
for (i = 0; i < ct; i++)
    delete lolas[i];
```

Paměť přidělená pomocí operátoru `new` by měla být uvolněna pomocí operátoru delete. To není práce třídy `ArrayTP`, neboť objekt `lolas` objekty skupiny tříd `Worker` nevytváří ani neruší. Pouze ukládá adresy objektů.

Univerzálnost šablon

Na šablony tříd můžete použít stejné techniky jako na běžné třídy. Šablony mohou sloužit jako základní třídy a mohou být také komponentami jiných tříd. Samy mohou být typem parametru pro jiné šablony. Šablonu pro zásobník můžete například implementovat pomocí šablony pole. Nebo můžete mít šablonu pro pole a pomocí ní vytvořit pole, jehož

prvky jsou zásobníky vytvořené podle šablony pro zásobník. Můžete tedy napsat následující řádky:

```
template <class T>
class Array
{
private:
    T entry;
    ...
};
template <class Type>
class GrowArray : public Array<Type> {...};    // dědičnost
template <class Tp>
class Stack
{
      Array<Tp> ar;  // použije Array<> jako komponentu
      ...
};
...
Array < Stack<int> > asi; // pole zásobníků hodnot typu int
```

V posledním příkazu musíte oddělit dva symboly > alespoň jedním prázdným znakem, aby nedošlo k záměně s operátorem >>.

Dalším příkladem univerzálnosti šablon je jejich rekurzivní použití. Dříve definovanou šablonu pro pole můžete například použít následujícím způsobem:

```
ArrayTP< ArrayTP<int,20>, 10> twodee;
```

Tímto příkazem vytvoříte pole twodee o deseti prvcích, které jsou samy polem o dvaceti prvcích typu `int`.

Můžete vytvořit šablony s více než jedním typovým parametrem. Předpokládejme například, že potřebujete třídu obsahující dva druhy hodnot. Můžete vytvořit šablonu `Pair` pro uložení dvou odlišných hodnot. (Shodou okolností nabízí standardní knihovna šablon podobnou šablonu nazvanou `pair`.) Jako příklad slouží krátký program ve výpisu 13.17.

Výpis 13.17. pairs.cpp

```
// pairs.cpp - definice a použití šablony Pair
#include <iostream>
using namespace std;
template <class T1, class T2>
class Pair
{
private:
    T1 a;
    T2 b;
public:
    T1 & first(const T1 & f);
    T2 & second(const T2 & s);
    T1 first() const { return a; }
    T2 second() const { return b; }
```

```
        Pair(const T1 & f, const T2 & s) : a(f), b(s) { }
    };
    template<class T1, class T2>
    T1 & Pair<T1,T2>::first(const T1 & f)
    {
        a = f;
        return a;
    }
    template<class T1, class T2>
    T2 & Pair<T1,T2>::second(const T2 & s)
    {
        b = s;
        return b;
    }
    int main()
    {
        Pair<char *, int> ratings[4] =
        {
            Pair<char *, int>("The Purple Duke", 5),
            Pair<char *, int>("Jake's Frisco Cafe", 4),
            Pair<char *, int>("Mont Souffle", 5),
            Pair<char *, int>("Gertie's Eats", 3)
        };
       int joints = sizeof(ratings) / sizeof (Pair<char *, int>);
       cout << "Rating:\t Jidelna\n";
        for (int i = 0; i < joints; i++)
            cout << ratings[i].second() << ":\t "
                 << ratings[i].first() << "\n";
       ratings[3].second(6);
        cout << "Moment! Opraveny rating:\n";
        cout << ratings[3].second() << ":\t "
             << ratings[3].first() << "\n";
        return 0;;
    }
```

Za povšimnutí stojí, že ve funkci `main()` musíte použít `Pair<char *,int>` pro vyvolání konstruktorů a jako parametru operátoru `sizeof`. Název třídy je totiž `Pair<char *,int>` a nikoli `Pair`. Rovněž název `Pair<String,ArrayDb>` by patřil zcela jiné třídě.

Zde je výstup programu:

```
Rating:  Jidelna
5:        The Purple Duke
4:        Jake's Frisco Cafe
5:        Mont Souffle
3:        Gertie's Eats
Moment! Opraveny rating:
6:        Gertie's Eats
```

Další novou vlastností šablony je možnost zadat pro typové parametry implicitní hodnoty:

```
template <class T1, class T2 = int> class Map {…};
```

Tento příkaz způsobí, že kompilátor použije pro třídu T2 typ int, pokud hodnotu T2 neuvedete:

```
Map<double, double> m1; // T1 je double, T2 je double
Map<double> m2;         // T1 je double, T2 je int
```

Standardní knihovna šablon (kapitola 15) tuto vlastnost často využívá, přičemž implicitním typem bývá třída.

Ačkoli pro typové parametry šablon můžete dodat implicitní hodnoty, nemůžete tak učinit v případě parametrů šablon funkcí. Implicitní hodnoty netypových parametrů však můžete dodat jak do šablon tříd, tak i do šablon funkcí.

Specializace šablon

Šablony tříd se podobají šablonám funkcí v tom, že můžete vytvořit implicitní instance, explicitní vytvoření instance a explicitní specializaci, což se souhrnně nazývá specializace. To znamená, že šablona popisuje třídu podle obecného typu, zatímco specializace je deklarace třídy vygenerovaná pomocí konkrétního typu.

Implicitní instance

Příklady, které jsme až doposud viděli, používaly implicitní instance. To znamená, že je v nich deklarován jeden nebo více objektů požadovaného typu a kompilátor vygeneruje specializovanou definici třídy podle receptu dodaného obecnou šablonou:

```
ArrayTb<int, 100> stuff; // implicitní instance
```

Kompilátor vygeneruje implicitní instanci až tehdy, když potřebuje objekt:

```
ArrayTb<double, 30> * pt;        // ukazatel, objekt zatím není potřeba
pt = new ArrayTb<double, 30>;    // nyní je objekt potřeba
```

Explicitní instance

Kompilátor vygeneruje explicitní instanci deklarace třídy, jestliže třídu deklarujete pomocí klíčového slova `template` a s uvedením požadovaných typů.

```
template ArrayTb<String, 100>; // vygeneruje třídu ArrayTB<String, 100>
```

Výraz ArrayTb<String, 100> znamená deklaraci třídy. V tomto případě kompilátor vytvoří definici třídy, i když dosud nebyl vytvořen nebo uveden žádný její objekt. Ale stejně jako u implicitní instance, i zde je obecná šablona použita jako vodítko pro vygenerování specializace.

Explicitní specializace

Explicitní specializace je definice určitých typů, které mají být použity místo obecné šablony. Někdy můžete potřebovat šablonu upravit, aby se chovala při vytvoření instance určitého typu odlišně; v takovém případě můžete vytvořit explicitní specializaci. Předpokládejme například, že jste definovali šablonu třídy reprezentující setříděné pole, jehož prvky jsou tříděny při jejich přidání do pole:

```
template <class T>
class SortedArray
{
    ...// detaily vynechány
};
```

Také předpokládejme, že šablona používá pro porovnání hodnot operátor >. Pro čísla bude taková šablona fungovat. Bude fungovat i v případě, že T bude představovat třídu, pokud jste definovali metodu T::operator(). Nebude však fungovat, jestliže T bude řetězec představovaný typem char *. Šablona sice fungovat bude, ale řetězce budou setříděny podle adresy a ne abecedně. Místo operátoru < musíte definovat třídu používající funkci strcmp(). V takovém případě můžete dodat explicitní specializaci šablony. Ta má formát šablony definované pro jeden konkrétní typ, ne pro typ obecný. Jestliže je kompilátor postaven před volbu mezi specializovanou šablonou a šablonou obecnou, použije specializovanou verzi.

Definice specializované šablony třídy má následující formát:

```
template <> class Classname<název-specializovaného-typu> { ... };
```

Starší kompilátory mohou rozeznat pouze starší formát, který nepoužívá template <>:

```
class Classname<jméno-specializovaného-typu> { ... };
```

Kdybyste chtěli dodat šablonu SortedArray specializovanou pro typ char* pomocí nové notace, použili byste následující kód:

```
template <> class SortedArray<char *>
{
    ...// detaily vynechány
};
```

Zde by implementace porovnávala hodnoty pole pomocí funkce strcmp() namísto operátoru >. Místo obecnější definice šablony budou požadavky na vytvoření šablony SortedArray pro typ char * používat tuto specializovanou definici:

```
SortedArray<int> scores;    // použití obecné definice
SortedArray<char *> dates;  // použití specializované definice
```

Částečné specializace

C++ také umožňuje částečné specializace, které částečně omezují obecnost šablony. Při částečné specializaci můžete například dodat určitý typ pro jeden z typových parametrů:

```
template <class T1, class T2> class Pair {...}; // obecná šablona
template <class T1> class Pair<T1, int> {...};  // specializovaná
                                                 // s T2 nastaveným na int
```

Lomené závorky za klíčovým slovem template deklarují typové parametry, které zatím nejsou specializovány. Druhá deklarace tedy nastavuje T2 na typ int, ale T1 ponechává otevřené. Všimněte si, že určení všech typů vede k prázdnému páru závorek a úplné explicitní specializaci:

```
template <> class Pair<int, int> {...}; // specializace
                                         // T1 a T2 nastaveny na int
```

Má-li kompilátor na výběr, použije nejvíce specializovanou šablonu:

```
Pair<double, double> p1; // použije obecnou šablonu Pair
Pair<double, int> p2;    // použije částečnou specializaci Pair<T1, int>
Pair<int, int> p3;       // použije explicitní specializaci Pair<int, int>
```

Dodáním speciální verze ukazatelů můžete částečně specializovat již existující šablonu:

```
template<class T> // obecná verze
class Feeb { ... };
template<class T*> // částečná ukazatelová specializace  class Feeb { ... };
                   // upravený kód
```

Jestliže dodáte neukazatelový typ, použije kompilátor obecnou verzi; pokud dodáte ukazatel, použije ukazatelovou specializaci:

```
Feeb<char> fb1;      // obecná šablona Feeb, T je char
Feeb<char *> fb2;    // specializace Feeb T*, T je char
```

Bez částečné specializace by druhá deklarace musela použít obecnou šablonu a interpretovat `T` jako typ `char *`. S částečnou specializací použije specializovanou šablonu a `T` je interpretováno jako char.

Částečná specializace také umožňuje vytvářet různá omezení. Můžete například vytvořit následující deklarace:

```
template <class T1, class T2, class T3> class Trio{...};
                                   // obecná šablona
template <class T1, class T2> class Trio<T1, T2, T2> {...};
                                   // specializace s  T3 nastaveným na  T2
template <class T1> class Trio<T1, T1*, T1*> {...};
                                   // specializace T3 a T2 nastaveným na  T1*
```

Při uvedených deklaracích by kompilátor činil tato rozhodnutí:

```
Trio<int, short, char *> t1;    // použití obecné šablony
Trio<int, short> t2;            // použití Trio<T1, T2, T2>
Trio<char, char *, char *> t3;  // použití Trio<T1, T1*, T1*>
```

Vícenásobná dědičnost

Vícenásobná dědičnost popisuje třídu, mající více než jednu základní třídu. Stejně jako u jednoduché dědičnosti i veřejná vícenásobná dědičnost by měla vyjadřovat vztah *je*. Jestliže budete mít například třídy `Waiter` a `Singer`, můžete od obou odvodit třídu `SingingWaiter`:

```
class SingingWaiter : public Waiter, public Singer {...};
```

Všimněte si, že každou základní třídu musíte blíže určit klíčovým slovem public. Pokud totiž není kvalifikátor uveden, předpokládá kompilátor soukromé odvození.

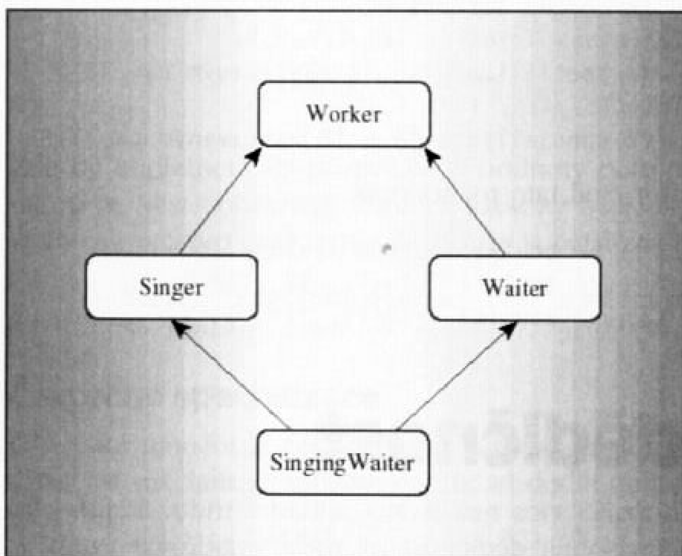
```
class SingingWaiter : public Waiter, Singer {...};
// Singer je soukromá základní třída
```

Jak jsme již v této kapitole probírali, může soukromá a chráněná dědičnost vyjádřit vztah *má*. Nyní se ale soustředíme na dědičnost veřejnou.

Vícenásobná dědičnost představuje pro programátora nové problémy. Dva hlavní problémy spočívají v dědění různých metod stejného názvu ze dvou základních tříd a dědění více instancí jedné třídy pomocí dvou nebo více příbuzných základních tříd. Řešení těchto problémů vyžaduje zavedení několika nových pravidel a syntaktických úprav. Použití vícenásobné dědičnosti tedy může být obtížnější a problémovější než jednoduchá dědičnost. Z tohoto důvodu mnoho členů komunity C++ vícenásobnou dědičnost důrazně odmítá; mnozí ji chtějí z jazyka odstranit. Jiní ji milují a tvrdí, že je velmi užitečná a u některých projektů nutná. A další radí používat ji opatrně a s mírou.

Prozkoumáme konkrétní příklad a zjistíme, jaké jsou problémy a jaká řešení. Použijeme pozměněný příklad s třídou `Worker` (výpisy 13.14 až 13.16). Příklad obsahuje třídy `Waiter` a `Singer` odvozené od abstraktní základní třídy `Worker`. Můžete tedy pomocí vícenásobné dědičnosti odvodit od tříd `Waiter` a `Singer` třídu `SingingWaiter` (viz obrázek 13.3). V tomto případě se základní třída `Worker` dědí dvakrát přes dvě samostatná odvození, což je situace, která u vícenásobné dědičnosti způsobuje nejvíce problémů. Aby byl příklad kratší, vynecháme nyní třídu `Greeter`. Předpokládejme tedy, že výpis 13.14 (`worker.h`) po smazání třídy `Greeter` tvoří základ pro vytvoření třídy `SingingWaiter`, která poslouží k ilustraci některých problémů. Konkrétně budete čelit následujícím otázkám:

- Kolik tříd Worker?
- Která metoda?



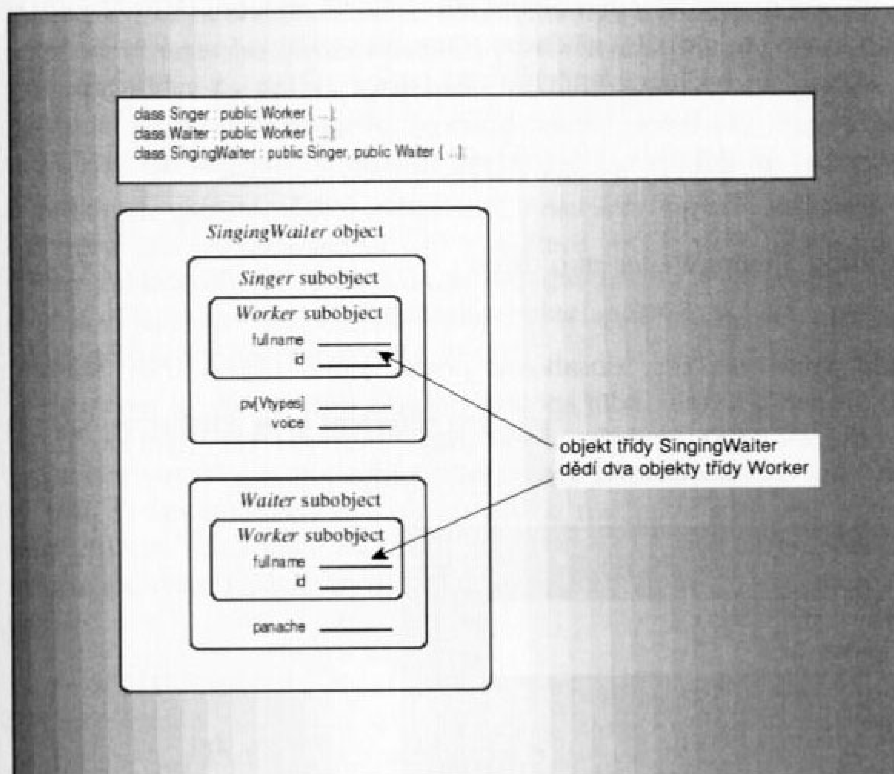
Obrázek 13.3. Vícenásobná dědičnost se sdíleným předkem

Kolik tříd Worker?

Předpokládejme, že začnete veřejným odvozením třídy `SingingWaiter` od tříd `Singer` a `Waiter`:

```
class SingingWaiter: public Singer, public Waiter {…};
```

Protože obě třídy `Singer` i `Waiter` zdědí komponentu `Worker`, bude mít třída `SingingWaiter` tyto dvě komponenty (viz obrázek 13.4).



Obrázek 13.4. Dědění dvou základních objektů

Jak jste mohli očekávat, způsobí toto odvození problémy. Běžně například můžete přiřadit adresu objektu odvozené třídy ukazateli na základní třídu, ale nyní toto přiřazení bude nejednoznačné:

```
SingingWaiter ed;
Worker * pw = &ed; // nejednoznačné
```

Normálně takový příklad nastaví ukazatel základní třídy na adresu objektu základní třídy v odvozeném objektu. Ale proměnná `ed` obsahuje dva objekty třídy `Worker`, a proto existují dvě adresy, ze kterých lze vybírat. Mohli byste objekt určit pomocí přetypování:

```
Worker * pw1 = (Waiter *) &ed; // objekt Worker ve třídě Waiter
Worker * pw2 = (Singer *) &ed; // objekt Worker ve třídě Singer
```

Toto určitě komplikuje techniku použití pole ukazatelů na základní třídu odkazujících na různé objekty (polymorfismus).

Existence dvou kopií objektu třídy `Worker` způsobí i další problémy. Skutečným problémem však je, proč byste vůbec měli mít dvě kopie objektu `Worker`? Zpívající číšník by měl mít, stejně jako jiný pracovník, pouze jedno jméno a jedno ID. Když byla do jazyka C++ přidána vícenásobná dědičnost, byla také přidána nová technika, *virtuální základní třída*, která takovou dědičnost umožňuje.

Virtuální základní třídy

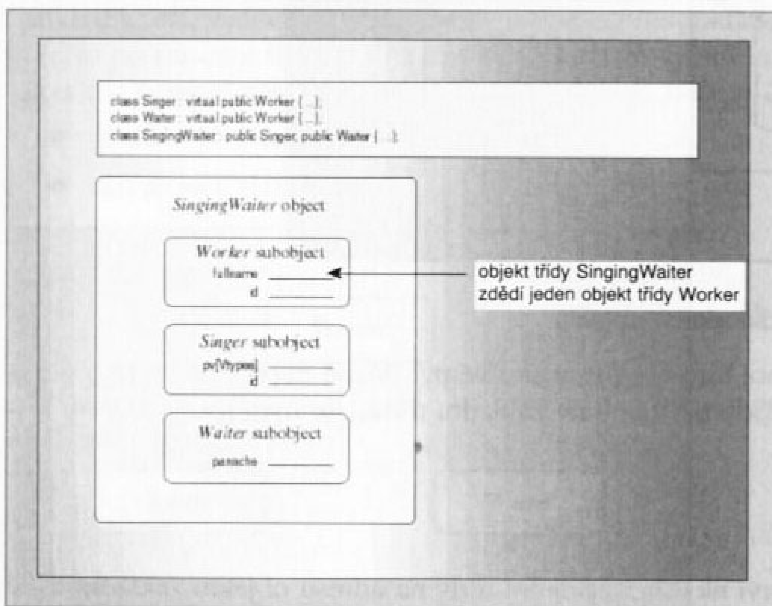
Virtuální základní třídy umožňují objektu odvozenému z více tříd, které samy sdílí společnou základní třídu, aby zdědil pouze jeden objekt této sdílené základní třídy. V našem příkladu byste udělali třídu `Worker` virtuální základní třídou pro třídy `Waiter` a `Singer` pomocí klíčového slova `virtual` v deklaracích těchto tříd (slova `virtual` a `public` mohou stát v libovolném pořadí):

```
class Singer : virtual public Worker {...};
class Waiter : public virtual Worker {...};
```

Potom byste definovali třídu SingingWaiter jako dříve:

```
class SingingWaiter: public Singer, public Waiter {...};
```

Nyní bude objekt třídy `SingingWaiter` obsahovat pouze jeden objekt třídy `Worker`. V podstatě objekty tříd `Singer` a `Waiter` sdílí společný objekt třídy `Worker` a nevytváří si své vlastní kopie (viz obrázek 13.5). Protože objekt třídy `SingingWaiter` nyní obsahuje pouze jeden podobjekt třídy Worker, můžete opět využít polymorfismu.



Obrázek 13.5. Dědičnost pomocí virtuální základní třídy

Podívejme se na některé možné otázky:

- Proč termín virtual?
- Proč se nezbavit deklarací virtuálních základních tříd a nestanovit virtuální chování normou vícenásobné dědičnosti?
- Existují nějaké záludnosti?

Za prvé, proč termín virtual? Koneckonců se nezdá, že mezi virtuálními funkcemi a virtuálními základními třídami existuje zřejmá souvislost. Ukazuje se, že ze strany komunity C++ existuje silný tlak proti zavádění nových klíčových slov. Bylo by nemilé, kdyby například nové klíčové slovo odpovídalo názvu nějaké důležité funkce nebo proměnné

v hlavním programu. Jazyk C++ tedy pouze znovu použil klíčové slovo `virtual` pro nový prostředek – tak trochu přetížení klíčového slova.

Dále, proč se nezbavit deklarací virtuálních základních tříd a nestanovit virtuální chování normou vícenásobné dědičnosti? Za prvé existují případy, kdy je potřeba mít více kopií základní třídy. Za druhé, vytvoříte-li základní třídu jako virtuální, bude program muset provádět nějakou práci navíc, a pokud takový prostředek nepotřebujete, neměli byste za něj platit. Za třetí existují důvody probírané v následujícím odstavci.

A nakonec, existují nějaké záludnosti? Ano. Aby virtuální základní třídy fungovaly, musí být upravena pravidla jazyka C++ a některé věci je potřeba zapsat odlišně. Použití virtuálních základních tříd si také může vyžádat změny v již existujícím kódu. Například přidáte-li třídu `SingingWaiter` do hierarchie `Worker`, musíte se vrátit k třídám `Singer` a `Waiter` a přidat jim klíčové slovo `virtual`.

Nová pravidla pro konstruktory

Existence virtuálních základních tříd vyžaduje zavedení nových pravidel pro konstruktory tříd. U nevirtuálních základních tříd je možné v seznamu inicializátorů uvést pouze konstruktory základních tříd nejbližší třídy. Tyto konstruktory však mohou zase předávat informace svým základním třídám. Můžete mít například následující uspořádání konstruktorů:

```
class A
{
    int a;
public:
    A(int n = 0) { a = n; }
    ...
};
class B: public A
{
    int b;
public:
    B(int m = 0, int n = 0) : A(n) : { b = m; }
    ...
};
class C : public B
{
    int c;
public:
    C(int q = 0, int m = 0, int n = 0) : B(m, n) { c = q; }
    ...
};
```

Konstruktor třídy C může vyvolat pouze konstruktory třídy `B` a konstruktor třídy `B` může vyvolat pouze konstruktory třídy `A`. Zde konstruktor třídy `C` použije hodnotu `q` a hodnoty `m` a n předá zpět konstruktoru třídy B. Konstruktor třídy `B` použije hodnotu `m` a hodnotu `n` předá konstruktoru třídy `A`.

Toto automatické předávání parametrů nefunguje, jestliže je třída `Worker` virtuální základní třídou. Uvažujte například následující možný konstruktor pro příklad vícenásobné dědičnosti:

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
                : Waiter(wk,p), Singer(wk,v) {}
```

Problém spočívá v tom, že automatické předání informací wk třídě Worker by proběhlo dvěma samostatnými cestami (Singer, Waiter). Aby nedocházelo k tomuto potenciálnímu konfliktu, neumožňuje C++ automatické předávání informací základní třídě přes nejbližší třídu, pokud je základní třída virtuální. Výše uvedený konstruktor tedy inicializuje položky panache a voice, ale informace v parametru wk se do podobjektu třídy Waiter nedostanou. Kompilátor však musí před vytvořením objektů odvozené třídy vytvořit komponentu objektu základní třídy; ve výše uvedeném případě se použije implicitní konstruktor třídy Worker.

Pokud chcete pro virtuální základní třídu použít něco jiného než implicitní konstruktor, musíte vyvolat konstruktor základní třídy explicitně. Konstruktor by měl tedy vypadat takto:

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
                : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
```

Zde je explicitně vyvolán konstruktor Worker(const Worker &). Všimněte si, že u virtuálních základních tříd je takové použití legální a často nutné, zatímco u nevirtuálních základních tříd je nelegální.

Upozornění

Jestliže má třída nepřímou virtuální základní třídu, měl by její konstruktor explicitně volat konstruktor této virtuální základní třídy, pokud implicitní konstruktor nepostačuje.

Která metoda?

Kromě změn zavedených pro konstruktory třídy bývá při vícenásobné dědičnosti často potřeba upravit kód. Uvažujte problém rozšíření metody Show() na třídu SingingWaiter. Protože objekt třídy SingingWaiter nemá nové datové položky, mohli byste si myslet, že postačí zděděné metody. To přináší první problém. Předpokládejme, že jste vynechali novou verzi metody Show() a snažíte se vyvolat zděděnou metodu Show() pomocí objektu třídy SingingWaiter:

```
SingingWaiter newhire("Elise Hawks", 2005, 6, sopran);
newhire.Show(); // nejednoznačné
```

Pokud byste nepředefinovali metodu při jednoduché dědičnosti, použila by se definice posledního předka. V tomto případě má funkci Show() každý z přímých předků, a proto je její volání nejednoznačné.

Upozornění

Vícenásobná dědičnost může vést k nejednoznačným funkčním voláním. Třída BadDude by například mohla zdědit dvě různé metody Draw() od tříd Gunslinger a PokerPlayer.

Svůj záměr můžete objasnit pomocí operátoru rozlišení:

```
SingingWaiter newhire("Elise Hawks", 2005, 6, sopran);
newhire.Singer::Show(); // použije verzi z třídy Singer
```

Ovšem lepším řešením je předefinovat metodu `Show()` pro třídu `SingingWaiter` a v ní určit, která metoda `Show()` se má použít. Kdybyste například chtěli, aby objekt třídy SingingWaiter používal verzi třídy `Singer`, napsali byste ji takto:

```
void SingingWaiter::Show()
{
    Singer::Show();
}
```

V případě jednoduché dědičnosti je způsob volání metody základní třídy z metody odvozené třídy zcela dostačující. Předpokládejme například existenci třídy `HeadWaiter` odvozené od třídy `Waiter`. Mohli byste použít posloupnost následujících definic, kde každá odvozená třída doplňuje informace zobrazené základní třídou:

```
void Worker::Show() const
{
    cout << "Jmeno: " << fullname << "\n";
    cout << "ID pracovnika: " << id << "\n";
}
void Waiter::Show() const
{
    Worker::Show();
    cout << "Rating mrstnosti: " << panache << "\n";
}
void HeadWaiter::Show() const
{
    Waiter::Show();
    cout << "Rating prezentace: " << presence << "\n";
}
```

Tento inkrementální postup však v případě třídy `SingingWaiter` fungovat nebude. Metoda

```
void SingingWaiter::Show()
{
    Singer::Show();
}
```

neuspěje, protože ignoruje komponentu třídy `Waiter`. Nápravu můžete provést, zavoláte-li také verzi třídy `Waiter`:

```
void SingingWaiter::Show()
{
      Singer::Show();
      Waiter::Show();
}
```

Tato funkce zobrazí jméno osoby a ID dvakrát, protože metoda `Worker::Show()` je volána jak z metody `Singer::Show()`, tak i z metody `Waiter::Show()`.

Jak to lze opravit? Jedním způsobem je použití modulárního přístupu namísto přístupu inkrementálního. To znamená vytvořit jednu metodu zobrazující pouze komponenty třídy Worker, druhou metodu zobrazující pouze komponenty třídy Waiter (ne komponenty obou tříd Waiter a Worker) a ještě další metodu zobrazující pouze komponenty třídy y. V takovém případě může metoda SingingWaiter.Show() tyto komponenty sloučit. Můžete například napsat:

```
void Worker::Data() const
{
    cout << "Jmeno: " << fullname << "\n";
    cout << "ID pracovnika: " << id << "\n";
}
void Waiter::Data() const
{
    cout << "Rating mrstnosti: " << panache << "\n";
}
void Singer::Data() const
{
    cout << "Rozsah vokalu: " << pv[voice] << "\n";
}
void SingingWaiter::Data() const
{
    Singer::Data();
    Waiter::Data();
}
void SingingWaiter::Show() const
{
    cout << "Kategorie: zpivajici cisnik\n";
    Worker::Data();
    Data();
}
```

Podobným způsobem by byly z odpovídajících komponent metod Data() vytvořeny ostatní metody Show().

Při tomto způsobu používají objekty metodu Show() veřejně. Naopak metody Data() by měly být pro třídy interní, pomocné metody pro veřejné rozhraní. Soukromé metody Data() by však zabránily použití metody Worker::Data() například ve třídě Waiter. Právě pro takové situace je užitečný chráněný přístup. Budou-li metody Data() chráněné, mohou být použity vnitřně všemi třídami v hierarchii a přitom zůstat skryty okolnímu světu.

Jiným způsobem řešení by bylo vytvořit všechny datové komponenty jako chráněné místo soukromých, ale pomocí chráněných metod namísto chráněných dat získáte pevnější kontrolu nad přístupem k datům.

Podobný problém představují metody Set(), které potřebují data pro nastavení hodnot objektů. Například metoda SingingWaiter::Set() by měla žádat objekt třídy Worker o informace pouze jednou a ne dvakrát. Je možné použít stejné řešení jako v předchozím případě. Vytvoříte chráněné metody Get(), které potřebují informace pouze pro jednu třídu a potom sestavíte metody Set(), které použijí metody Get() jako stavební kvádry.

Stručně řečeno, pro zavedení vícenásobné dědičnosti se sdíleným předkem je nutné zavést virtuální základní třídy, změnit pravidla pro inicializační seznamy konstruktorů a zřej-

mě i pozměnit kód tříd, pokud nebyly psány s ohledem na vícenásobnou dědičnost. Ve výpisu 13.18 jsou upravené deklarace tříd s těmito změnami a výpis 13.19 obsahuje implementaci.

Výpis 13.18. workermi.h

```
// workermi.h — třídy pro vícenásobnou dědičnost
#include "strng2.h"
class Worker   // abstraktní základní třída
{
private:
     String fullname;
     long id;
protected:
     virtual void Data() const;
     virtual void Get();
public:
     Worker() : fullname("no one"), id(0L) {}
     Worker(const String & s, long n)
              : fullname(s), id(n) {}
     virtual ~Worker() = 0; // čistě virtuální funkce
     virtual void Set() = 0;
     virtual void Show() const = 0;
};
class Waiter : virtual public Worker
{
private:
     int panache;
protected:
     void Data() const;
     void Get();
public:
     Waiter() : Worker(), panache(0) {}
     Waiter(const String & s, long n, int p = 0)
              : Worker(s, n), panache(p) {}
     Waiter(const Worker & wk, int p = 0)
              : Worker(wk), panache(p) {}
     void Set();
     void Show() const;
};
class Singer : virtual public Worker
{
protected:
enum {jiny, alt, kontralt, sopran,
                   bas, bariton, tenor};
     enum {Vtypes = 7};
     void Data() const;
     void Get();
private:
     static char *pv[Vtypes];    // řetězce pro typy hlasu
```

```
    int voice;
public:
    Singer() : Worker(), voice(other) {}
    Singer(const String & s, long n, int v = other)
            : Worker(s, n), voice(v) {}
    Singer(const Worker & wk, int v = other)
            : Worker(wk), voice(v) {}
    void Set();
    void Show() const;
};
// vícenásobná dědičnost
class SingingWaiter : public Singer, public Waiter
{
protected:
    void Data() const;
    void Get();
public:
    SingingWaiter() {}
    SingingWaiter(const String & s, long n, int p = 0,
                            int v = Singer::other)
            : Worker(s,n), Waiter(s, n, p), Singer(s, n, v) {}
    SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
            : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
    SingingWaiter(const Waiter & wt, int v = other)
            : Worker(wt),Waiter(wt), Singer(wt,v) {}
    SingingWaiter(const Singer & wt, int p = 0)
            : Worker(wt),Waiter(wt,p), Singer(wt) {}
    void Set();
    void Show() const;
};
```

Výpis 13.19. workermi.cpp

```
// workermi.cpp — metody třídy Worker pro vícenásobnou dědičnost
#include "workermi.h"
#include <iostream>
using namespace std;
// metody třídy Worker
Worker::~Worker() { }
// chráněné metody
void Worker::Data() const
{
    cout << "Jmeno: " << fullname << "\n";
    cout << "ID pracovnika: " << id << "\n";
}
void Worker::Get()
{
    cin >> fullname;
    cout << "Zadejte ID pracovnika: ";
    cin >> id;
```

```
    while (cin.get() != '\n')
        continue;
}
// metody třídy Waiter
void Waiter::Set()
{
    cout << "Zadejte jmeno cisnika: ";
    Worker::Get();
    Get();
}
void Waiter::Show() const
{
    cout << "Kategorie: cisnik\n";
    Worker::Data();
    Data();
}
// chráněné metody
void Waiter::Data() const
{
  cout << "Rating mrstnosti: " << panache << "\n";
}
void Waiter::Get()
{
    cout << "Zadejte rating mrstnosti cisnika: ";
    cin >> panache;
    while (cin.get() != '\n')
        continue;
}
// metody třídy Singer
char * Singer::pv[Singer::Vtypes] = {"jiny", "alt", "kontralt",
            "sopran", "bas", "bariton", "tenor"};
void Singer::Set()
{
    cout << "Zadejte jmeno zpevaka: ";
    Worker::Get();
    Get();
}
void Singer::Show() const
{
    cout << "Kategorie: zpevak\n";
    Worker::Data();
    Data();
}
// chráněné metody
void Singer::Data() const
{
    cout << "Rozsah vokalu: " << pv[voice] << "\n";
}
void Singer::Get()
{
    cout << "Zadejte cislo udavajici rozsah vokalu zpevaka:\n";
```

```
    int i;
    for (i = 0; i < Vtypes; i++)
    {
        cout << i << ": " << pv[i] << " ";
        if ( i % 4 == 3)
            cout << '\n';
    }
    if (i % 4 != 0)
        cout << '\n';
    cin >> voice;
    while (cin.get() != '\n')
        continue;
}
// metody třídy SingingWaiter
void SingingWaiter::Data() const
{
    Singer::Data();
    Waiter::Data();
}
void SingingWaiter::Get()
{
    Waiter::Get();
    Singer::Get();
}
void SingingWaiter::Set()
{
    cout << "Zadejte jmeno zpivajiciho cisnika: ";
    Worker::Get();
    Get();
}
void SingingWaiter::Show() const
{
    cout << "Kategorie: zpivajici cisnik\n";
    Worker::Data();
    Data();
}
```

Zvědavost samozřejmě žádá, abychom tyto třídy vyzkoušeli a ve výpisu 13.20, který vychází z výpisu 13.16, je kód k tomu určený. Všimněte si, že program využívá polymorfismu pomocí přiřazování adres různých druhů tříd ukazatelům základní třídy. Výpis zkompilujte společně se soubory `workermi.cpp` a `strng2.cpp`. Ujistěte se, že jste vložili hlavičkový soubor `arraytp.h`, obsahující deklaraci šablony pole.

Kompatibilita:

Jestliže váš systém šablony nepodporuje, můžete místo nich použít standardní pole. Musíte pouze provést malé změny podle pokynů ve výpisu.

Výpis 13.20. workmi.cpp

```
// workmi.cpp - vícenásobná dědičnost
// zkompilovat s workermi.cpp, strng2.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "workermi.h"
#include "arraytp.h"  // vynechte, jestliže systém šablony nepodporuje
const int SIZE = 5;
int main()
{
    ArrayTP<Worker *, SIZE> lolas;
    // jestliže systém šablony nepodporuje, vynechte předcházející řádek
    // a použijte následující Worker * lolas[SIZE];
    int ct;
    for (ct = 0; ct < SIZE; ct++)
    {
        char choice;
        cout << "Zadejte kategorii zamestnace:\n"
             << "c: cisnik z: zpevak "
             << "p: zpivajici cisnik k: konec\n";
        cin >> choice;
        while (strchr("czpk", choice) == NULL)
        {
            cout << "Zadejte prosim c, z, p nebok: ";
            cin >> choice;
        }
        if (choice == 'k')
            break;
        switch(choice)
        {
            case 'c': lolas[ct] = new Waiter;
                      break;
            case 'z': lolas[ct] = new Singer;
                      break;
            case 'p': lolas[ct] = new SingingWaiter;
                      break;
        }
        cin.get();
        lolas[ct]->Set();
    }
    cout << "\nZde jsou vasi zamestnanci:\n";
    int i;
    for (i = 0; i < ct; i++)
    {
        cout << '\n';
        lolas[i]->Show();
    }
    for (i = 0; i < ct; i++)
        delete lolas[i];
    return 0;
}
```

Zde je ukázka běhu:

```
    Zadejte kategorii zamestnance:
    c: cisnik z: zpevak p: zpivajici cisnik k: konec
    c
    Zadejte jmeno cisnika: Wally Snipeside
    Zadejte ID pracovnika: 1020
    Zadejte rating mrstnosti cisnika: 5
  Zadejte kategorii zamestnance:
    c: cisnik z: zpevak p: zpivajici cisnik k: konec
    p
    Zadejte jmeno zpivajiciho cisnika: Natasha Gargalova
    Zadejte ID pracovnika: 1021
    Zadejte rating mrstnosti cisnika: 6
    Zadejte cislo udavajici rozsah vokalu zpevaka:
    0: jiny 1: alt 2: kontralt 3: sopran
    4: bas 5: bariton 6: tenor
    3
  Zadejte kategorii zamestnance:
    c: cisnik z: zpevak p: zpivajici cisnik k: konec
    k

    Zde jsou vasi zamestnanci:

  Kategorie: cisnik
    Jmeno: Wally Snipeside
   ID pracovnika: 1020
    Panache rating: 5

  Kategorie: zpivajici cisnik
  Jmeno: Natasha Gargalova
   ID pracovnika: 1021
    Rozsah vokalu: sopran
    Rating mrstnosti: 6
```

Podívejme se ještě na několik dalších záležitostí, týkajících se vícenásobné dědičnosti.

Smíšené virtuální a nevirtuální základní třídy

Uvažujte opět případ odvozené třídy dědící základní třídu vícekrát. Pokud je základní třída virtuální, bude odvozená třída obsahovat jeden podobjekt základní třídy. Jestliže základní třída virtuální není, bude odvozená třída obsahovat více podobjektů. Co se stane v případě, jsou-li základní třídy virtuální i nevirtuální? Předpokládejme například, že třída `B` je virtuální základní třídou tříd `C` a `D` a nevirtuální základní třídou tříd `X` a `Y`. Dále předpokládejme, že třída `M` je odvozená od tříd `C`, `D`, `X` a `Y`. V tomto případě bude třída `M` obsahovat jeden podobjekt třídy `B` pro všechny virtuálně odvozené předky (to znamená třídy `C` a `D`) a samostatný podobjekt třídy `B` pro každého nevirtuálního předka (to znamená třídy `X` a `Y`). Celkem tedy bude obsahovat tři podobjekty třídy `B`. Jestliže třída dědí určitou základní třídu přes několik virtuálních a nevirtuálních cest, pak bude obsahovat jeden podobjekt základní třídy reprezentující všechny virtuální cesty a jeden podobjekt reprezentující každou nevirtuální cestu.

Virtuální základní třídy a dominance

Použití virtuálních základních tříd mění způsob, jakým jazyk C++ řeší nejednoznačnosti. U nevirtuálních základních tříd jsou pravidla jednoduchá. Jestliže třída dědí dvě nebo více položek (data nebo metody) stejného názvu od různých tříd, potom použití tohoto názvu bez určení třídy je nejednoznačné. Pokud jsou však použity virtuální základní třídy, takové použití může a nemusí být nejednoznačné. V takovém případě, jestliže jeden název *dominuje* nad ostatními, může být použit bez uvedení třídy.

Jak může jeden název položky dominovat nad druhým? Název v odvozené třídě dominuje nad stejným názvem v kterékoli třídě předka přímo či nepřímo. Uvažujte například následující definice:

```
class B
{
public:
        short q();
        ...
};
class C : virtual public B
{
public:
        long q();
        int omb()
        ...
};
class D : public C
{
       ...
};
class E : virtual public B
{
private:
       int omb();
       ...
};
class F: public D, public E
{
       ...
};
```

Zde definice metody `q()` ve třídě `C` dominuje nad metodou ve třídě `B`, protože třída `C` je odvozená od třídy `B`. Metody třídy `F` tedy mohou pomocí výrazu `q()` označit metodu `C::q()`. Na druhé straně ani jedna definice metody `omb()` není dominantní, protože třídy C a `E` nedědí jedna od druhé. Takže pokus použít výraz `omb()` bez kvalifikátoru ve třídě `F` bude nejednoznačný.

Pravidla virtuální nejednoznačnosti neberou ohled na pravidla přístupu. To znamená, že přestože je metoda `E::omb()` soukromá a není tedy přímo přístupná třídě `F`, použití `omb()` bude nejednoznačné. Podobně kdyby byla metoda `C::q()` soukromá, dominovala by metodě `D::q()`. V takovém případě byste mohli volat metodu `B::q()` ve třídě `F`, ale volání `q()` bez kvalifikátoru by odkazovalo na nepřístupnou metodu `C::q()`.

Přehled vícenásobné dědičnosti

Nejdříve si zopakujme vícenásobnou dědičnost bez virtuálních základních tříd. Tato forma nepotřebuje žádná nová pravidla. Jestliže však třída dědí dvě položky stejného názvu od různých tříd, musíte pro odlišení mezi těmito dvěma položkami použít v odvozené třídě kvalifikátor třídy. To znamená, že metody třídy `BadDude`, odvozené ze tříd `Gunslinger` a `PokerPlayer`, budou požívat volání `Gunslinger::draw()` a `PokerPlayer:: draw()`, aby byly odlišeny metody `draw()` zděděné od těchto dvou tříd. Jinak by si měl kompilátor stěžovat na nejednoznačné použití.

Jestliže třída dědí od nevirtuální základní třídy více cestami, pak zdědí jeden objekt základní třídy pro každou nevirtuální instanci základní třídy. V některých případech toto může být žádoucí, ale častěji představuje více instancí jedné základní třídy problém.

Dále se podíváme na vícenásobnou dědičnost s virtuálními základními třídami. Ze základní třídy se stane virtuální základní třída, jestliže v odvozené třídě použijete klíčové slovo `virtual` při označení dědění:

```
class marketing : public virtual reality { … };
```

Hlavní změnou, a také důvodem pro zavedení virtuálních základních tříd, je to, že třída dědící jednu nebo více instancí virtuální základní třídy zdědí pouze jeden objekt základní třídy. Implementace této vlastnosti vyžaduje určité změny:

- Odvozená třída s nepřímou virtuální základní třídou by měla pomocí svých konstruktorů vyvolat nepřímé konstruktory základní třídy přímo, což je zakázáno v případě nepřímých nevirtuálních základních tříd.
- Nejednoznačnost názvů řeší pravidlo dominance.

Jak vidíte, může vícenásobná dědičnost programování zkomplikovat. Většina komplikací však vzniká v situaci, kdy odvozená třída dědí od stejné základní třídy více než jednou cestou. Pokud se této situaci vyhnete, stačí dávat pozor na nutné kvalifikátory u zděděných názvů.

Shrnutí

Jazyk C++ nabízí několik způsobů psaní znovupoužitelného kódu. Veřejná dědičnost (kapitola 12) umožňuje modelovat vztah *je*, při kterém mohou odvozené třídy využívat kód tříd základních. Soukromá a chráněná dědičnost také umožňují využít kód základní třídy, ale v těchto případech je modelem vztah *má*. Při soukromé dědičnosti se z veřejných a chráněných položek základní třídy stanou soukromé položky třídy odvozené. V případě chráněné dědičnosti se z veřejných a chráněných položek stanou chráněné položky odvozené třídy. V obou případech se tedy veřejné rozhraní základní třídy stane vnitřním rozhraním odvozené třídy. Tento jev se někdy popisuje jako dědění implementace bez rozhraní, neboť objekt odvozené třídy nemůže explicitně použít rozhraní základní třídy. Nemůžete se tedy na objekt odvozené třídy dívat jako na druh objektu třídy základní. Z tohoto důvodu nemůže ukazatel nebo reference na základní třídu odkazovat na objekt třídy odvozené bez explicitního přetypování.

Kód třídy lze znovu použít také pro vytvoření třídy, jejíž položky jsou samy objekty. Tento způsob se nazývá kompozice nebo vrstvení a rovněž modeluje vztah *má*. Kompozice se snáze implementuje a používá než soukromá nebo chráněná dědičnost, a proto se jí obvykle dává přednost. Soukromá a chráněná dědičnost však nabízejí trochu větší možnosti. Díky dědičnosti může například odvozená třída přistupovat k chráněným položkám základní třídy a také lze předefinovat virtuální funkci zděděnou od základní třídy. Vzhledem k tomu, že kompozice není formou dědičnosti, nemáte tyto možnosti při znovupoužití kódu třídy k dispozici. Na druhou stranu kompozice je vhodnější při potřebě většího počtu objektů jedné třídy. Například objekt třídy `State` může obsahovat pole objektů třídy `County`.

Vícenásobná dědičnost umožňuje při návrhu třídy použít kód více než jedné třídy. Soukromá a chráněná vícenásobná dědičnost modelují vztah *má*, zatímco veřejná modeluje vztah *je*. Při vícenásobné dědičnosti mohou vzniknout problémy v důsledku více definovaných názvů a více zděděných základních tříd. Pomocí virtuálních základních tříd však zavedete nová pravidla pro psaní inicializačních seznamů pro konstruktory a pro řešení nejednoznačností.

Šablony tříd umožňují vytvořit obecný návrh třídy, ve které je nějaký typ, obvykle typ položky, reprezentován typovým parametrem. Typická šablona vypadá takto:

```
template <class T>
class Ic
{
      T v;
      ...
public:
      Ic(const T & val) : v(val) { }
      ...
};
```

Zde je T typovým parametrem a chová se jako zástupce skutečného typu určeného později. (Tímto parametrem může být jakýkoli platný název jazyka C++, ale názvy `T` nebo `Type` jsou nejběžnější.) V tomto kontextu můžete místo termínu `class` také použít termín `typename`:

```
template <typename T> // stejné jako template <class T>
class Rev {...} ;
```

Definice tříd (instance) jsou generovány při deklaraci objektu třídy s určením konkrétního typu. Například deklarace

```
class Ic<short> sic;    // implicitní instance
```

způsobí, že kompilátor vygeneruje deklaraci třídy, ve které je každý výskyt typového parametru T v šabloně nahrazen v deklaraci třídy skutečným parametrem typu short. V tomto případě je název třídy `Ic<short>`, nikoli `Ic`. `Ic<short>`, a nazývá se specializace šablony. Konkrétně jde o implicitní instanci.

Explicitní instance vzniká při deklarování specifické specializace pomocí klíčového slova `template`:

```
template class Ic<int>; // explicitní vytvoření instance
```

V této situaci vygeneruje kompilátor pomocí obecné šablony specializaci typu `int` `Ic<int>` i přesto, že dosud nebyly vytvořeny žádné objekty této třídy.

Můžete vytvořit také explicitní specializace, což jsou specializované deklarace tříd přepisující definici šablony. Stačí definovat třídu pomocí klíčového slova výraz `template <>`, potom uvést název třídy následovaný lomenými závorkami obsahujícími typ požadované specializace. Například specializaci třídy Ic pro ukazatele na typ char byste vytvořili takto:

```
template <> class Ic<char *>.
{
    char * str;
    ...
public:
    Ic(const char * s) : str(s) { }
    ...
};
```

V tom případě by deklarace tvaru

```
class Ic<char *> chic;
```

pro `chic` nepoužila obecnou šablonu, ale specializovanou definici.

V šabloně třídy můžete definovat více než jeden obecný typ a také můžete uvést netypové parametry:

```
template <class T, class TT, int n>
class Pals {…};
```

Deklarace

```
Pals<double, String, 6> mix;
```

by vygenerovala implicitní instanci, že typ `double` nahradí `T`, typ `String` nahradí `TT` a hodnota `6` nahradí `n`.

Šablony tříd mohou být specializovány jen částečně:

```
template <class T> Pals<T, T, 10> {…};
template <class T, class TT> Pals<T, TT, 100> {…};
template <class T, int n> Pals <T, T*, n> {…};
```

První deklarace vytvoří specializaci, kdy jsou oba typy stejné a parametr n má hodnotu `6`. Podobně druhá deklarace vytvoří specializaci pro parametr s n hodnotou `100` a třetí deklarace vytvoří specializaci, ve které druhý typ je ukazatel na první typ.

Cílem všech těchto metod je umožnit znovupoužití vyzkoušeného kódu bez nutnosti ručního kopírování. Tím se programování zjednodušuje a programy se stávají spolehlivějšími.

Opakovací otázky

1. Pro každou z následujících sad tříd rozhodněte, zda je lepší použít pro třídy ve druhém sloupci soukromé nebo veřejné odvození:

```
class Bear                  class PolarBear
class Kitchen               class Home
class Person                class Programmer
class Person                class HorseAndJockey
class Person,               class Driver
class Automobile
```

2. Předpokládejte následující definice:

```
class Frabjous {
private:
      char fab[20];
public:
      Frabjous(const char * s = "C++") : fab(s) { }
      virtual void tell() { cout << fab; }
};
class Gloam {
private:
      int glip;
      Frabjous fb;
public:
      Gloam(int g = 0, const char * s = "C++");
      Gloam(int g, const Frabjous & f);
      void tell();
};
```

Dodejte definice metod třídy `Gloam` za předpokladu, že metoda `tell()` třídy `Gloam` má zobrazit hodnoty položek `glip` a `fb`.

3. Předpokládejte následující definice:

```
class Frabjous {
private:
      char fab[20];
public:
      Frabjous(const char * s = "C++") : fab(s) { }
      virtual void tell() { cout << fab; }
};
class Gloam : private Frabjous{
private:
      int glip;
public:
      Gloam(int g = 0, const char * s = "C++");
      Gloam(int g, const Frabjous & f);
      void tell();
};
```

Dodejte definice metod třídy Gloam za předpokladu, že metoda tell() třídy Gloam má zobrazit hodnoty položek glip a fab.

4. Předpokládejte následující definici založenou na definici šablony Stack z výpisu 13.9 a třídy Worker z výpisu 13.18:

```
Stack<Worker *> sw;
```

Napište deklaraci třídy, která bude vygenerována. Stačí deklarace třídy bez vložených metod.

5. Pomocí definic šablon z této kapitoly definujte následující:
   - pole objektů třídy String
   - zásobník polí s prvky typu double
   - pole zásobníků obsahujících ukazatele na objekty třídy Worker

6. Popište rozdíl mezi virtuální a nevirtuální základní třídou.

Programovací cvičení

1. Třída Wine obsahuje objekt třídy String (kapitola 11), který obsahuje název vína, a objekt třídy ArrayDb (tato kapitola) obsahující počet láhví pro každý z několika po sobě jdoucích roků. Implementujte třídu Wine pomocí kompozice a otestujte ji pomocí jednoduchého programu. Program by se měl dotázat na název vína, velikost pole, první rok pole a počet lahví z každého roku. Pomocí těchto dat by měl vytvořit objekt třídy Wine a potom zobrazit informace uložené v tomto objektu.

2. Třída Wine obsahuje objekt třídy String (kapitola 11), který obsahuje název vína, a objekt třídy ArrayDb (tato kapitola) obsahující počet lahví pro každý z několika po sobě jdoucích roků. Implementujte třídu Wine pomocí soukromé dědičnosti a otestujte ji pomocí jednoduchého programu. Program by se měl dotázat na název vína, velikost pole, první rok pole a počet lahví z každého roku. Pomocí těchto dat by měl vytvořit objekt třídy Wine a potom zobrazit informace uložené v tomto objektu.

3. Definujte šablonu QueueTp simulující frontu. Otestujte ji vytvořením fronty ukazatelů na objekt třídy Worker (výpis 13.18) a dále ji použijte v programu, který bude podobný programu z výpisu 13.20.

4. Třída Person obsahuje křestní jméno a příjmení nějaké osoby. Kromě konstruktorů obsahuje metodu Show() zobrazující obě jména. Třída Gunslinger je virtuálně odvozená od třídy Person. Obsahuje metodu Draw(), která vrací hodnotu typu double reprezentující dobu, za kterou je pistolník schopen tasit zbraň. Také obsahuje položku typu int vyjadřující počet zářezů na pistolníkově zbrani. A nakonec má funkci Show() zobrazující všechny uvedené informace.

   Třída PokerPlayer je virtuálně odvozena od třídy Person. Obsahuje metodu Draw(), která vrací náhodné číslo v intervalu 1 až 52 vyjadřující hodnotu karty. (Mohli byste definovat třídu Card s hodnotami položek barvy a karty a vrátit hodnotu pro funkci Draw()). Třída PokerPlayer používá funkci Show(). Třída BadDude je veřejně

odvozena od tříd `Gunslinger` a `PokerPlayer`. Obsahuje metodu `Gdraw()` vracející dobu, ve které drsňák vytasí zbraň, a metodu `Cdraw()` vracející další vytaženou kartu. Také obsahuje odpovídající funkci `Show()`. Definujte všechny tyto třídy a metody společně s dalšími nezbytnými metodami (jako jsou metody pro nastavení hodnot objektů) a vyzkoušejte je v jednoduchém programu podobném tomu z výpisu 13.20.

5. Zde je několik deklarací tříd:

```
// emp.h — hlavičkový soubor třídy employee a jejich potomků
#include <cstring>
#include <iostream>
using namespace std;
const int SLEN = 20;
class employee
{
protected:
    char fname[SLEN];
    char lname[SLEN];
    char job[SLEN];
public:
    employee();
    employee(char * fn, char * ln, char * j);
    employee(const employee & e);
    virtual void ShowAll() const;
    virtual void SetAll(); // vyzve uživatele k zadání hodnot
    friend ostream & operator<<(ostream & os, const employee & e);
  };
class manager: virtual public employee
{
protected:
    int inchargeof;
public:
    manager();
    manager(char * fn, char * ln, char * j, int ico = 0);
    manager(const employee & e, int ico);
    manager(const manager & m);
    void ShowAll() const;
    void SetAll();
};
class fink: virtual public employee
{
protected:
    char reportsto[SLEN];
public:
    fink();
    fink(char * fn, char * ln, char * j, char * rpo);
    fink(const employee & e, char * rpo);
    fink(const fink & e);
    void ShowAll() const;
    void SetAll();
};
```

```
class highfink: public manager, public fink
{
public:
      highfink();
      highfink(char * fn, char * ln, char * j, char * rpo, int ico);
      highfink(const employee & e, char * rpo, int ico);
      highfink(const fink & f, int ico);
      highfink(const manager & m, char * rpo);
      highfink(const highfink & h);
      void ShowAll() const;
      void SetAll();
};
```

Všimněte si, že hierarchie tříd používá vícenásobnou dědičnost s virtuální základní třídou, pamatujte tedy na speciální pravidla pro inicializační seznamy konstruktorů v takovém případě. Všimněte si také, že datové položky jsou deklarovány jako chráněné a ne jako soukromé. Tím se zjednoduší kód pro některou z metod třídy `highfink`. (Všimněte si například, že metoda `highfink::ShowAll()` volá jen metody `fink::ShowAll()` a `manager::ShowAll()` a závěrem zavolá dvakrát metodu `ee::ShowAll()`.) Můžete však použít soukromá data a vytvořit další chráněné metody podle vzoru třídy `Worker` s vícenásobnou dědičností. Vytvořte implementace metod tříd a vyzkoušejte je v programu. Zde je minimální testovací program. Měli byste přidat alespoň jeden test členské funkce .

```
// useempl.cpp - použití tříd employee
#include <iostream>
using namespace std;
#include "emp.h"
int main()
{
      employee th("Trip", "Harris", "Thumper");
      cout << th << '\n';
      th.ShowAll();
      manager db("Debbie", "Bolt", "Twigger", 5);
      cout << db << '\n';
      db.ShowAll();
      cout << "Press a key for next batch of output:\n";
      cin.get();
      fink mo("Matt", "Oggs", "Oiler", "Debbie Bolt");
      cout << mo << '\n';
      mo.ShowAll();
      highfink hf(db, "Curly Kew");
      hf.ShowAll();
      cout << "Using an employee * pointer:\n";
      employee * tri[4] = { &th, &db, &mo, &hf };
      for (int i = 0; i < 4; i++)
         tri[i]->ShowAll();
      return 0;;
}
```

Kompatibilita:

Symantec C++ vyžaduje, aby prvky pole `tri` byly přiřazeny adresám objektů individuálně a ne inicializačním příkazem.

Proč není definován operátor přiřazení?

Proč jsou metody `showall()` a `setall()` virtuální?

Proč je třída `employee` virtuální základní třídou?

Proč nemá třída `highfink` žádnou datovou část?

Proč stačí pouze jedna verze funkce `operator<<()`?

Co by se stalo, kdyby kód na konci programu byl zaměněn následujícím kódem?

```
employee tri[4] = {th, db, mo, hf};
for (int i = 0; i < 4; i++)
        tri[i].showall();
```

Přátelé, výjimky a další

KAPITOLA 14

Témata kapitoly:

- Spřátelené třídy
- Metody spřátelených tříd
- Vnořené třídy
- Vyvolávání výjimek, pokusné a záchytné bloky
- Třídy výjimek
- RTTI (runtime type information)
- Operátory `dynamic_cast` a `typeid`
- Operátory `static_cast`, `const_cast` a `reinterpret_cast`

V této kapitole uzavřeme některá nedokončená témata jazyka C++ a potom se pustíme do nejnovějších dodatků jazyka C++. Mezi nedokončená témata patří spřátelené třídy, spřátelené členské funkce a vnořené třídy, což jsou třídy deklarované uvnitř jiné třídy. Mezi nejnovější dodatky, které budou probrány, patří výjimky, systém RTTI a vylepšená kontrola přetypování. Výjimky jazyka C++ poskytují mechanismus pro správu neobvyklých výskytů, které by způsobily ukončení programu. Systém RTTI (runtime type information) je mechanismus pro rozpoznání typu objektu. Nové operátory pro přetypování zvyšují bezpečnost přetypování. Poslední tři vlastnosti jsou v C++ celkem nové a ne všechny kompilátory je již podporují.

Přátelé

Několik příkladů v této knize používalo jako součást rozšířeného rozhraní třídy spřátelené funkce. Tyto funkce ale nejsou jediným druhem přátel, které třída může mít. Třída může být také přítelem. V takovém případě může jakákoli metoda spřátelené třídy přistupovat k soukromým a chráněným položkám původní třídy. Můžete být i trochu omezující a označit pouze některé členské funkce třídy za přátele jiné třídy. Ve třídě definujete, které funkce, členské funkce nebo třídy budou spřátelené; přátelství nemůže být nastaveno z vnějšku. Ačkoli tedy přátelé poskytují přístup zvenku k soukromým položkám třídy, nenarušují tím podstatu objektově orientovaného programování. Naopak, poskytují veřejnému rozhraní větší pružnost.

Spřátelené třídy

Kdy můžete z jedné třídy udělat přítele jiné třídy? Podívejme se na nějaký příklad. Předpokládejme, že musíte naprogramovat simulaci televizoru a jeho dálkového ovládání. Rozhodnete se definovat třídu `Tv` představující televizi a třídu `Remote` představující dálkové ovládání. Je jasné, že mezi těmito třídami by měl být nějaký vztah, ale jaký? Dálkové ovládání není televize a stejně je tomu i naopak, není tedy možné uplatnit vztah *je* veřejné dědičnosti. Ani jeden z těchto objektů není součástí toho druhého, není tedy možné uplatnit vztah *má* pomocí kompozice nebo soukromé či chráněné dědičnosti. Platí, že dálkové ovládání může změnit stav televizoru a proto se nabízí udělat z třídy `Remote` přítele třídy `Tv`.

Nejdříve nadefinujeme třídu `Tv`. Televizor můžete vyjádřit pomocí sady stavových položek, to znamená proměnných, popisujících různé vlastnosti televizoru. Zde jsou některé z možných stavů:

- Zapnutý-Vypnutý
- Nastavení programu
- Nastavení hlasitosti
- Režim ladění signálu z kabelu nebo z antény
- Vstup z TV nebo videorekordéru

Ladicí režim odráží skutečnost, že v USA je mezera mezi kanály, počínaje kanálem 14, rozdílná pro příjem z kabelu a pro příjem vysílání UHF. Výběr vstupu volí mezi TV, což může být buď signál z kabelu nebo z TV, a videorekordérem. Některé přístroje mohou nabízet více možností, ale pro naše účely tento seznam postačuje.

Televizor má také některé parametry, které nejsou proměnnými stavu. Televizory se liší například počtem přijímaných programů, a pro sledování této hodnoty můžete přidat nějakou položku,

Dále musíte vytvořit třídu s metodami umožňujícími tato nastavení změnit. Mnoho dnešních televizorů skrývá ovládací prvky za panelem, ale u většiny z nich je stále možné změnit program a další vlastnosti bez použití dálkového ovládání. Často se sice můžete posouvat po jednom programu, ale můžete zvolit libovolný program. Podobně obvykle existuje tlačítko pro snížení nebo zvýšení hlasitosti.

Dálkové ovládání by mělo duplikovat ovládací prvky umístěné na televizoru. Mnoho z jeho metod by mohlo být implementováno pomocí metod třídy `Tv`. Navíc dálkový ovladač většinou nabízí volbu libovolného programu. To znamená, že z programu 2 můžete přímo přejít na program 20, aniž byste museli projít všechny programy mezi nimi. Mnoho dálkových ovládání může také fungovat ve dvou režimech – jako ovladač televizoru a jako ovladač videorekordéru.

Všechny tyto úvahy vedou k definici uvedené ve výpisu 14.1. Definice obsahuje několik konstant definovaných jako výčet. Příkazem, který udělá z třídy `Remote` spřátelenou třídu, je tento:

```
friend class Remote;
```

Přítele můžete deklarovat ve veřejné, soukromé nebo chráněné části; na místě nezáleží. Protože třída `Remote` zmiňuje třídu `Tv`, musí kompilátor o třídě Tv vědět dříve, než začne

zpracovávat třídu Remote. Toho nejsnáze dosáhnete, jestliže budete třídu Tv definovat jako první. Jinou možností je předběžná deklarace, kterou probereme za chvíli.

Kompatibilita:

Jestliže váš kompilátor nepodporuje typ bool, použijte typ int a hodnoty 0, 1 namísto hodnot false a true.

Výpis 14.1. tv.h

```
// tv.h - třídy Tv a Remote
#ifndef _TV_H_
#define _TV_H_
class Tv
{
public:
    friend class Remote;  // třída Remote může přistupovat k soukromým
//částem třídy Tv
    enum State{Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
    enum {TV, VCR};
    Tv(State s = Off, int mc = 100) : state(s), volume(5),
        maxchannel(mc), channel(2), mode(Cable), input(TV) {}
    void onoff() {state = (state == On)? Off : On;}
    bool ison() const {return state == On;}
    bool volup();
    bool voldown();
    void chanup();
    void chandown();
    void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
    void set_input() {input = (input == TV)? VCR : TV;}
    void settings() const; // zobrazí nastavení
private:
    State state;        // zapnuto nebo vypnuto
    int volume;         // hlasitost
    int maxchannel;     // maximální počet programů
    int channel;        // aktuální nastavení programů
    int mode;           // normální signál nebo kabel
    int input;          // TV nebo VCR
};
class Remote
{
private:
    int mode;           // ovládání TV nebo VCR
public:
    Remote(int m = Tv::TV) : mode(m) {}
    bool volup(Tv & t) { return t.volup();}
    bool voldown(Tv & t) { return t.voldown();}
```

```
        void onoff(Tv & t) { t.onoff(); }
        void chanup(Tv & t) {t.chanup();}
        void chandown(Tv & t) {t.chandown();}
        void set_chan(Tv & t, int c) {t.channel = c;}
        void set_mode(Tv & t) {t.set_mode();}
        void set_input(Tv & t) {t.set_input();}
};
#endif
```

Většina metod třídy je definována jako třídy vložené. Všimněte si, že každá metoda třídy `Remote` s výjimkou konstruktoru má jako parametr referenci na objekt třídy `Tv`. To ukazuje, že dálkové ovládání je určeno pro konkrétní televizor. Výpis 14.2 obsahuje zbývající definice. Funkce pro nastavení hlasitosti mění velikost položky volume po jednotce, dokud zvuk nedosáhne svého minimálního nebo maximálního nastavení. Funkce pro výběr programu používají cyklické přetáčení a nejnižší nastavený program 1 následuje ihned po nejvýše nastaveném `maxchannel`.

Mnoho metod používá pro přepínání mezi oběma nastaveními podmíněný operátor:

```
void onoff() {state = (state == On)? Off : On;}
```

Za předpokladu, že hodnoty těchto dvou stavů jsou 0 a 1, lze změny stavu dosáhnout kompaktněji pomocí kombinace bitové – výlučného OR a operátoru přiřazení (`^=`), které jsou popsané v příloze E:

```
void onoff() {state ^= 1;}
```

Výpis 14.2. tv.cpp

```
// tv.cpp – metody třídy Tv
  //(metody třídy Remote jsou vložené)
  #include <iostream>
  using namespace std;
  #include "tv.h"

  bool Tv::volup()
  {
      if (volume < MaxVal)
      {
          volume++;
          return true;
      }
      else
          return false;
  }
  bool Tv::voldown()
  {
      if (volume > MinVal)
      {
        volume–;
        return true;
      }
```

```
    else
        return false;
}
void Tv::chanup()
{
    if (channel < maxchannel)
        channel++;
    else
        channel = 1;
}
void Tv::chandown()
{
    if (channel > 1)
        channel--;
    else
        channel = maxchannel;
}
void Tv::settings() const
{
    cout << "Televize je " << (state == Off? "vypnuta" : "zapnuta") <<
endl;
    if (state == On)
    {
        cout << "Nastaveni hlasitosti = " << volume << endl;
        cout << "Nastaveni programu = " << channel << endl;
        cout << "Rezim = "
            << (mode == Antenna? "antenni" : "kabelovy") << endl;
        cout << "Vstup = "
            << (input == TV? "TV" : "VCR") << endl;
    }
}
```

Výpis 14.3 obsahuje krátký program testující některé z uvedených vlastností. Stejný ovladač je použit k ovládání dvou televizorů.

Výpis 14.3. use_tv.cpp

```
//use_tv.cpp
#include <iostream>
using namespace std;
#include "tv.h"
int main()
{
    Tv s20;
    cout << "Pocatecni nastaveni 20\" televize:\n";
    s20.settings();
    s20.onoff();
    s20.chanup();
    cout << "\nUpravene nastaveni 20\" televize:\n";
    s20.settings();
    Remote grey;
```

```
        grey.set_chan(s20, 10);
        grey.volup(s20);
        grey.volup(s20);
        cout << "\n20\" nastaveni pro pouziti dalkoveho ovladace:\n";
        s20.settings();
        Tv s27(Tv::On);
        s27.set_mode();
        grey.set_chan(s27,28);
        cout << "\n27\" nastaveni:\n";
        s27.settings();
        return 0;
}
```

Zde je výstup programu:

```
Pocatecni nastaveni 20" televize:
TV is Off

Upravene nastaveni 20"televize:
Televize je zapnuta
Nastaveni hlasitosti = 5
Nastaveni programu = 3
Rezim = kabelovy
Vstup = TV

20" nastaveni pro pouziti dalkoveho ovladace:
Televize je zapnuta
Nastaveni hlasitosti = 7
Nastaveni programu = 10
Rezim = kabelovy
Vstup = TV

27" nastaveni:
Televize je zapnuta
Nastaveni hlasitosti = 5
Nastaveni programu = 28
Rezim = antenni
Vstup = TV
```

Hlavním smyslem tohoto příkladu je ukázat, že spřátelená třída je přirozeným idiomem vyjadřujícím některé vztahy. Bez nějaké formy přátelství byste museli vytvořit některé části třídy `Tv` jako veřejné nebo vytvořit nějakou těžkopádnou, velkou třídu obsahující jak televizor, tak i dálkové ovládání. Takové řešení však neodpovídá skutečnosti, že pomocí jednoho dálkového ovladače lze ovládat několik televizorů.

Spřátelené členské funkce

Podíváte-li se na kód posledního příkladu, můžete si všimnout, že většina metod třídy `Remote` je implementována pomocí veřejného rozhraní třídy `Tv`. To znamená, že tyto metody ve skutečnosti status přítele nepotřebují. Jedinou metodou třídy `Remote` přistupující přímo k soukromým položkám třídy `Tv` je metoda `Remote::set_chan()`, takže pouze tato me-

toda musí být metodou spřátelenou. Vytvořit z vybraných členů třídy přátele jiné třídy je možné, ale je to poněkud obtížnější. Musíte dávat pozor na pořadí různých deklarací a definic. Podívejme se proč.

Z metody `Remote::set_chan()` se stane přítel třídy `TV`, pokud ji deklarujete jako spřátelenou v deklaraci této třídy:

```
class Tv
{
  friend void Remote::set_chan(Tv & t, int c);
  ...
};
```

Aby však kompilátor mohl tento příkaz zpracovat, musí již znát definici třídy Remote. Jinak nebude vědět, že `Remote` je třída a že `set_chan()` je metoda této třídy. Nabízí se tedy umístit definici třídy Remote před definici třídy `Tv`. Ale skutečnost, že v metodách třídy `Remote` jsou uvedeny objekty třídy `Tv` znamená, že definice třídy `Tv` by měla předcházet definici třídy `Remote`. Částečným řešením této kruhové závislosti tohoto problému je použití *předběžné deklarace*. To znamená, že před definici třídy `Remote` vložíte

```
class Tv;        // předběžná deklarace
```

Tím vznikne následující uspořádání:

```
class Tv; // předběžná deklarace
class Remote { ... };
class Tv { ... };
```

Mohli byste místo něho použít toto uspořádání?

```
class Remote;     // předběžná deklarace
class Tv { ... };
class Remote { ... };
```

Odpověď zní ne. Jak jsme se zmínili dříve, důvodem je, že když kompilátor zjistí, že metoda třídy `Remote` je deklarována jako přítel v deklaraci třídy `Tv`, musí již znát obecnou deklaraci třídy `Remote` a zvláště pak metody `set_chan()`.

Zůstává další problém. Ve výpisu 14.1 obsahovala deklarace třídy `Remote` následující vložený kód:

```
void onoff(Tv & t) { t.onoff(); }
```

Protože je zde volána metoda třídy `Tv`, musí kompilátor v této chvíli již znát deklaraci třídy `Tv`, aby věděl, jaké metody obsahuje. Ale jak jste viděli, tato deklarace musí následovat za deklarací třídy `Remote`. Řešením tohoto problému je omezit deklaraci třídy `Remote` na *deklarace* metod a skutečné *definice* umístit za třídu `Tv`. Tento způsob vede k následujícímu pořadí:

```
class Tv;              // předběžná deklarace
class Remote { ... };  // pouze prototypy metod používající třídu Tv
class Tv { ... };
// zde budou definice metod třídy Remote
```

Prototypy vydají takto:

```
void onoff(Tv & t);
```

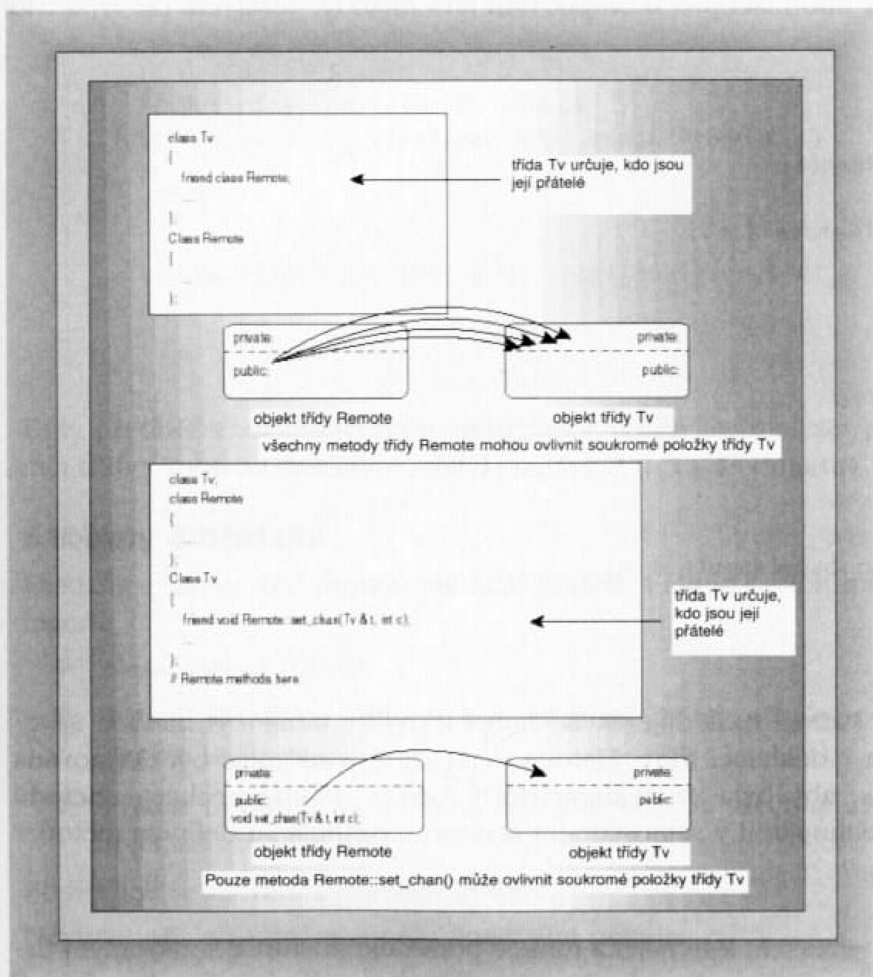
Vše co kompilátor při zpracování této deklarace potřebuje vědět je, že Tv představuje třídu, a tuto informaci dodá předběžná deklarace. V momentě, kdy kompilátor narazí na skutečné definice metod, bude již znát deklaraci třídy Tv a bude mít informace potřebné pro jejich zkompilování. Pomocí klíčového slova inline v definicích metod z nich můžete učinit metody vložené. Výpis 14.4 ukazuje upravený hlavičkový soubor.

Výpis 14.4. tvfm.h

```
// tvfm.h - třídy Tv a Remote s použitím spřátelené metody
#ifndef _TVFM_H_
#define _TVFM_H_
class Tv;           // předběžná deklarace
class Remote
{
public:
    enum State{Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
    enum {TV, VCR};
private:
    int mode;
public:
    Remote(int m = TV) : mode(m) {}
    bool volup(Tv & t);         // pouze prototyp
    bool voldown(Tv & t);
    void onoff(Tv & t) ;
    void chanup(Tv & t) ;
    void chandown(Tv & t) ;
    void set_mode(Tv & t) ;
    void set_input(Tv & t);
    void set_chan(Tv & t, int c);
};
class Tv
{
public:
    friend void Remote::set_chan(Tv & t, int c);
    enum State{Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
    enum {TV, VCR};
    Tv(State s = Off, int mc = 100) : state(s), volume(5),
        maxchannel(mc), channel(2), mode(Cable), input(TV) {}
    void onoff() {state = (state == On)? Off : On;}
    bool ison() const {return state == On;}
    bool volup();
    bool voldown();
    void chanup();
    void chandown();
    void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
    void set_input() {input = (input == TV)? VCR : TV;}
    void settings() const;
```

```
private:
        State state;
        int volume;
        int maxchannel;
        int channel;
        int mode;
        int input;
};
// metody třídy Remote jako vložené funkce
inline bool Remote::volup(Tv & t) { return t.volup();}
inline bool Remote::voldown(Tv & t) { return t.voldown();}
inline void Remote::onoff(Tv & t) { t.onoff(); }
inline void Remote::chanup(Tv & t) {t.chanup();}
inline void Remote::chandown(Tv & t) {t.chandown();}
inline void Remote::set_mode(Tv & t) {t.set_mode();}
inline void Remote::set_input(Tv & t) {t.set_input();}
inline void Remote::set_chan(Tv & t, int c) {t.channel = c;}
#endif
```

Tato verze se chová stejně jako ta původní. Rozdíl je v tom, že pouze jedna metoda třídy Remote je přítelem třídy `Tv` a ne všechny metody. Tento rozdíl zachycuje obrázek 14.1.



Obrázek 14.1. Spřátelené třídy a spřátelené metody

Mimochodem, chcete-li udělat z celé třídy Remote přítele, nemusíte provádět předběžnou deklaraci, protože příkaz friend sám identifikuje Remote jako třídu:

```
friend class Remote;
```

Jiné přátelské vztahy

Možné jsou i jiné kombinace přátel a tříd. Pojďme se ve zkratce na některé podívat. Předpokládejme, že pokrok v technologiích vyprodukuje interaktivní dálkové ovladače. Interaktivní dálkový ovladač například umožní zadat odpověď na otázku položenou v televizním pořadu a televize dokáže aktivovat bzučák na dálkovém ovladači, pokud odpověď byla špatná. Nebudeme si všímat možnosti televize programovat diváky pomocí takových prostředků a podíváme se pouze na aspekty při programování v jazyce C++. Nové nastavení by těžilo ze vzájemného přátelství, přičemž některé metody třídy Remote by mohly jako dříve ovlivnit objekt třídy Tv a některé metody třídy Tv by mohly ovlivnit objekt třídy Remote. Toho můžete dosáhnout, pokud z obou tříd vytvoříte přítele třídy druhé. To znamená, že třída Tv bude přítelem třídy Remote a třída Remote bude přítelem třídy Tv. Důležité je pamatovat si, že prototyp metody třídy TV používající objekt třídy Remote lze uvést *před* deklarací třídy Remote, ale její definici musíte uvést až *po* deklaraci třídy Remote, aby měl kompilátor dostatek informací pro zkompilování této metody. Sestavení by vypadalo následovně:

```
class Tv
{
friend class Remote;
public:
    void buzz(Remote & r);
    ...
};
class Remote
{
friend class Tv;
public:
    void Bool volup(Tv & t) { t.volup(); }
    ...
};
inline void Tv::buzz(Remote & r)
{
    ...
}
```

Protože deklarace třídy Remote následuje za deklarací třídy Tv, může být metoda Remote::volup() definována v deklaraci třídy. Metoda Tv::buzz() však musí být definována mimo deklaraci třídy Tv, aby byla za deklarací třídy Remote. Pokud nechcete metodu buzz() jako vloženou, definujte ji v samostatném souboru obsahujícím definice metod.

Sdílení přátelé

Další použití přátel je v situacích, kdy nějaká funkce potřebuje přístup k soukromým datům ve dvou samostatných třídách. Logicky by taková funkce měla být členskou funkcí

obou tříd, ale to je nemožné. Mohla by být členskou funkcí jedné třídy a přítelem druhé, ale občas je rozumnější učinit z ní přítele obou tříd. Předpokládejme například, že máte třídu `Probe` představující nějaké programovatelné měřicí zařízení a třídu `Analyzer` představující programovatelné analytické zařízení. Obě zařízení mají vnitřní hodiny a vy byste rádi měli možnost oboje tyto hodiny synchronizovat. Mohli byste vytvořit něco následujícího:

```
class Analyzer; // předběžná deklarace
class Probe
{
    friend void sync(Analyzer & a, const Probe & p); // synchronizace
a na p
    friend void sync(Probe & p, const Analyzer & a); // synchronizace
p na a
    ...
};
class Analyzer
{
    friend void sync(Analyzer & a, const Probe & p); // synchronizace
a na p
    friend void sync(Probe & p, const Analyzer & a); // synchronizace
p na a
    ...
};
// definice spřátelených funkcí
inline void sync(Analyzer & a, const Probe & p)
{
    ...
}
inline void sync(Probe & p, const Analyzer & a)
{
    ...
}
```

Díky předběžné deklaraci bude kompilátor vědět, že Analyzer je typ, až narazí v deklaraci třídy Probe na deklarace přátel.

Šablony a přátelé

Deklarace šablon tříd mohou mít také přátele. Přátele šablon můžeme rozdělit do tří kategorií:

- Nešablonoví přátelé.
- Přátelé vázaní na šablonu, což znamená, že typ přítele je určen typem třídy při vzniku její instance.
- Přátelé nevázaní na šablonu, což znamená, že všechny instance přítele jsou přáteli každé instance třídy.

Podívejme se na příklady všech tří typů.

Nejdříve příklad šablony s nešablonovými přáteli:

```
template <class Type>
```

```
class HasFriend
{
    friend void date(); // přítel všech instancí třídy HasFriend
    ...
};
```

Funkce `date()` bude přítelem všech možných instancí šablony. Bude například přítelem tříd `HasFriend<int>` a `HasFriend<String>`.

Předpokládejme, že chcete příteli předat parametr šablony třídy. Můžete změnit deklaraci přítele například takto?

```
friend void date(HasFriend &);  // možné?
```

Odpověď je ne. Důvodem je, že neexistuje žádný objekt třídy `HasFriend`. Existuje pouze několik konkrétních instancí, například `HasFriend<short>`. Chcete-li vytvořit parametr šablony třídy, musíte označit instanci. Můžete například napsat:

```
template <class Type>
class HasFriend
{
    friend void date(HasFriend<Type> &); // přítel vázaný na šablonu
...
};
```

Tento způsob by vyžadoval definici šablony funkce:

```
template <class A>
void date(A &) { ... };
```

Díky této kombinaci se kód `date<short>()` stane přítelem instance `HasFriend<short>` a podobně u dalších instancí. Každá instance třídy má vlastní odpovídající instanci spřátelené funkce. Toto je příklad *přátelství vázaného na šablonu*. Stejnou techniku lze použít při vytváření spřátelených tříd vázaných na šablonu.

Nyní předpokládejme, že deklaraci třídy poněkud pozměníte:

```
template <class Type>
class HasFriend
{
        friend void date(HasFriend<T> &); // přítel nevázaný na šablonu
        ...
};
```

Původní verze používala stejný název (`Type`) obecného typu v hlavičce třídy i v prototypu přítele. Tato verze používá pro prototyp jiný název obecného typu (`T`). Tento formát způsobí, že všechny instance třídy `HasFriend` budou mít všechny instance přítele `date()`.Tedy `date<int>()`, `date<double>()` a `date<HasFriend<String> >()` jsou všechno přátelé instance `HasFriend<char *>`. Toto je příklad *přátelství nevázaného na šabloně*. Stejnou techniku lze použít při vytváření spřátelených tříd nevázaných na šablonu.

Mohou být šablony přáteli nešablonových tříd? Nemohou, ale určité instance ano:

```
class Pal
{
        friend class HasFriend<long>; // ok
```

```
    friend class HasFriend;    // nepovoleno
};
```

Důvodem je, že názvy šablon bez instancí, například `HasFriend`, se mohou objevit pouze v šabloně. Běžný kód může použít pouze konkrétní instance, jako například `HasFriend<Complex>`.

Vnořené třídy

V jazyce C++ můžete deklaraci třídy vložit do jiné třídy. Třída deklarovaná v jiné třídě se nazývá *vnořená třída* a pomáhá vyhnout se nepořádku v názvech, bude-li mít nový typ platnost v rozsahu této třídy. Členské funkce třídy obsahující deklaraci mohou vytvářet a používat objekty vnořené třídy. Okolní svět může vnořenou třídu použít pouze v případě, že deklarace je ve veřejné části a s použitím operátoru rozlišení. (Starší verze C++ však vnořené třídy neumožňují nebo koncept implementují neúplně.)

Vnoření tříd není totéž co kompozice. Vzpomeňte si, že kompozice znamená existenci objektu jako položky jiné třídy. Vnoření třídy naopak položku nevytváří. Místo toho definuje typ, který je znám pouze lokálně třídě obsahující vloženou deklaraci třídy.

Obvyklými důvody pro vnoření třídy bývá snaha pomoci implementaci jiné třídy a vyhnout se konfliktům názvů. Příklad třídy `Queue` (kapitola 11, výpis 11.11) obsahoval skrytý případ vnořených tříd pomocí vnoření definice struktury:

```
class Queue
{
// definice platné v rozsahu třídy
    // Node je vnořená definice struktury a je v této třídě lokální
    struct Node {Item item; struct Node * next;};
    ...
};
```

Protože struktura je třída, jejíž položky jsou implicitně veřejné, je Node opravdu vnořenou třídou. Tato definice však nevyužívá možností tříd. Konkrétně postrádá explicitní konstruktor. Pojďme to nyní napravit.

Nejdříve zjistíme, kde jsou v příkladu třídy `Queue` vytvářeny objekty struktury `Node`. Prozkoumáním deklarace třídy (výpis 11.11) a definic metod (výpis 11.12) odhalíte, že jediným místem, kde se objekty struktury Node vytváří, je metoda `enqueue()`:

```
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;  // vytvoří uzel
    if (add == NULL)
        return false;           // při nedostatku paměti skončí
    add->item = item;           // nastaví ukazatele uzlu add->next = NULL;
    ...
}
```

V tomto kódu je vytvořen objekt struktury `Node`, a potom jsou jeho položkám explicitně přiřazeny hodnoty. Tento druh práce se lépe hodí pro konstruktor.

Nyní již víte, kde a jak konstruktor použít a můžete tedy vytvořit definici vhodného konstruktoru:

```
class Queue
{
// platnost v rozsahu třídy
// Node je vnořená definice třídy a je pro tuto třídu lokální
    class Node
    {
    public:
        Item item;
        Node * next;
        Node(const Item & i) : item(i), next(0) { }
    };
    ...
};
```

Tento konstruktor inicializuje položku uzlu `item` na hodnotu parametru i a ukazatel next nastaví na hodnotu `0`, což je jeden ze způsobů, jak v C++ zapsat prázdný ukazatel. (Při použití `NULL` by bylo nutné vložit soubor, ve kterém je `NULL` definován.) Protože ve všech uzlech vytvořených třídou `Queue` je ukazatel next zpočátku nastaven na nulu, je tento jediný konstruktor pro třídu dostačující.

Nyní přepíšeme funkci enqueue() pomocí konstruktoru:

```
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node(item); // vytvoří a inicializuje uzel
    if (add == 0)
        return false;            // při nedostatku paměti skonči
    ...
}
```

Díky konstruktoru je kód metody `enqueue()` trochu kratší a trochu bezpečnější, protože inicializace proběhne automaticky a programátor si nemusí přesně pamatovat, co by měl udělat.

V tomto příkladě byl konstruktor definován v deklaraci třídy. Předpokládejme, že byste ho chtěli definovat v souboru s definicemi metod. Definice musí odrážet skutečnost, že třída `Node` je definována v třídě `Queue`. Toho dosáhnete dvojím použitím operátoru rozlišení:

```
Queue::Node::Node(const Item & i) : item(i), next(0) { }
```

Vnořené třídy a přístup k nim

S vnořenými třídami se pojí dva druhy přístupu. Za prvé, místo deklarace vnořené třídy určuje její oblast platnosti; to znamená, že stanovuje části programu, ve kterých je možné vytvářet objekty dané třídy. Za druhé, jako u každé třídy poskytují veřejné, chráněné a soukromé části vnořené třídy řízení přístupu ke členům třídy. Místo a způsob použití

vnořené třídy závisí na rozsahu platnosti a řízení přístupu. Prozkoumejme tyto body podrobněji.

Rozsah platnosti

Jestliže je vnořená třída deklarována v soukromé části druhé třídy, pak je vidět pouze v této druhé třídě. To je například případ třídy `Node` vnořené do deklarace třídy `Queue` v posledním příkladu. (Může to vypadat, že třída `Node` byla definována před soukromou částí, ale nezapomeňte, že u tříd je soukromý přístup implicitní.) Členové třídy `Queue` tedy mohou používat objekty třídy `Node` a ukazatele na objekty této třídy, ale ostatní části programu nebudou o existenci třídy vědět. Pokud byste od třídy `Queue` odvodili nějakou třídu, nebyla by třída `Node` viditelná ani v této třídě, protože odvozená třída nemá přímý přístup k soukromým částem základní třídy.

Pokud je vnořená třída deklarována v chráněné části druhé třídy, je viditelná v této druhé třídě, ale neviditelná pro okolní svět. V tomto případě však bude odvozená třída o její existenci vědět a může přímo vytvářet objekty tohoto typu.

Jestliže je vnořená třída deklarována ve veřejné části druhé třídy, je dostupná této druhé třídě, třídám odvozeným od této druhé třídy a protože je veřejná, je také dostupná okolnímu světu. Ale protože její oblast platnosti je omezena na třídu, musí se v okolním světě použít s kvalifikátorem třídy. Předpokládejme například následující deklaraci:

```
class Team
{
public:
      class Coach { ... };
      ...
};
```

Nyní předpokládejme existenci nezaměstnaného trenéra, který nepatří žádnému týmu. Máte-li vytvořit objekt Coach vně třídy `Team`, můžete napsat toto:

```
Team::Coach forhire; // vytvoří objekt třídy Coach vně třídy Team
```

Stejný rozsah platnosti musíte vzít v úvahu při použití vnořených struktur a výčtů. Mnoho programátorů chce pomocí výčtů, umístěných ve veřejné části třídy vytvořit konstanty třídy, které by mohli používat jiní programátoři. Například mnoho implementací tříd definovaných na podporu proudu `iostream` nabízí pomocí této techniky různé formátovací možnosti, kterých jsme se dotkli již dříve a podrobněji je prozkoumáme v kapitole 16. V tabulce 14.1 jsou shrnuty rozsahy platností vnořených tříd, struktur a výčtů.

Tabulka 14.1. Rozsah platnosti vnořených tříd, struktur a výčtů

Deklarace ve vnořené třídě	Dostupnost vnořené třídě	Dostupnost třídám odvozeným od vnořené třídy	Dostupné okolnímu světu
Soukromá část	Ano	Ne	Ne
Chráněná část	Ano	Ano	Ne
Veřejná část	Ano	Ano	Ano, s kvalifikátorem třídy

Řízení přístupu

Jakmile je třída dostupná, přichází ke slovu kontrola přístupu. Přístup k vnořeným třídám se řídí stejnými pravidly jako přístup k běžným třídám. Deklarace třídy `Node` v deklaraci třídy `Queue` nezaručí třídě `Queue` žádná zvláštní přístupová privilegia k třídě `Node` a ani třídě `Node` nezaručí žádná zvláštní přístupová privilegia k třídě `Queue`. Objekt třídy `Queque` tedy může explicitně přistupovat pouze k veřejným položkám třídy `Node`. Z tohoto důvodu byly v příkladu s třídou `Queue` všechny položky třídy `Node` veřejné. Tím je narušena obvyklá praxe, podle které jsou datové položky soukromé, ale třída `Node` představuje vnitřní prostředek implementace třídy `Queue` a pro okolní svět není viditelná. Je totiž deklarována v soukromé části třídy `Queue`. Ačkoli tedy metody třídy `Queue` mohou přistupovat k položkám třídy `Node` přímo, klient používající třídu `Queue` tak činit nemůže.

Stručně řečeno, místo deklarace třídy určuje rozsah platnosti neboli viditelnost třídy. Je-li určitá třída v rozsahu platnosti, určují obvyklá pravidla řízení přístupu (veřejná, chráněná, soukromá, přítel) druh přístupu, který program získá k položkám vnořené třídy.

Vnořování do šablony

Viděli jste, že šablony se dobře hodí pro implementaci tříd kontejnerů, jako je třída `Queue`. Asi vás zajímá, zda existence vnořené třídy způsobí nějaké problémy při převedení definice třídy `Queue` na šablonu. Odpověď je ne. Ve výpisu 14.5 je ukázán způsob, jak takovou konverzi provést. Jak je u šablon běžné, hlavičkový soubor obsahuje šablonu třídy společně s metodou šablon funkcí.

Výpis 14.5. queuetp.h

```
// queuetp.h – šablona fronty s vnořenou třídou
template <class Item>
class QueueTP
{
private:
    enum {Q_SIZE = 10};
    // Node je definice vnořené třídy
    class Node
    {
    public:
        Item item;
        Node * next;
        Node(const Item & i):item(i), next(0){ }
    };
    Node * front;     // ukazatel na začátek fronty
    Node * rear;    // ukazatel na konec fronty
    int items;  // aktuální počet položek fronty
    const int qsize;  // maximální počet položek fronty
    QueueTP(const QueueTP & q) : qsize(0) {}
    QueueTP & operator=(const QueueTP & q) { return *this; }
public:
    QueueTP(int qs = Q_SIZE);
```

```
    ~QueueTP();
    bool isempty() const
    {
        return items == 0;
    }
    bool isfull() const
    {
        return items == qsize;
    }
    int queuecount() const
    {
        return items;
    }
    bool enqueue(const Item &item); // přidá položku na konec fronty
    bool dequeue(Item &item);       // odebere položku z fronty
};

// metody šablony QueueTP
template <class Item>
QueueTP<Item>::QueueTP(int qs) : qsize(qs)
{
    front = rear = 0;
    items = 0;
}

template <class Item>
QueueTP<Item>::~QueueTP()
{
    Node * temp;
    while (front != 0)   // dokud fronta není prázdná
    {
        temp = front;     // uloží adresu první položky
        front = front->next;// nastaví ukazatel na následující položku
        delete temp;    // zruší původní první položku
    }
}
// přidání položky na konec fronty
template <class Item>
bool QueueTP<Item>::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node(item);  // vytvoří uzel
    if (add == NULL)
        return false;     // konec v případě nedostatku paměti
    items++;
    if (front == 0)      // jestliže je fronta prázdná,
        front = add;     // přidá položku na začátek fronty
    else
        rear->next = add;  // jinak na konec fronty
    rear = add;        // ukazatel na konec fronty ukazuje na nový uzel
```

```
    return true;
}
// Uloží první položku do proměnné  item a odstraní ji z fronty
template <class Item>
bool QueueTP<Item>::dequeue(Item & item)
{
    if (front == 0)
        return false;
    item = front->item;  // nastaví item na první položku fronty
    items--;
    Node * temp = front;  // uloží adresu první položky
    front = front->next;  // nastaví začátek na následující položku
    delete temp;        // zruší původní první položku
    if (items == 0)
        rear = 0;
    return true;
}
```

Na této šabloně je zajímavé, že třída Node je definována pomocí obecného typu Item. Takže deklarace

```
QueueTp<double> dq;
```

vede ke třídě Node definované pro hodnoty typu double, zatímco deklarace

```
QueueTp<char> cq;
```

vede ke třídě `Node` definované pro hodnoty typu char. Tyto dvě třídy `Node` jsou definovány ve dvou samostatných třídách `QueueTP`, takže mezi nimi nedochází k žádnému konfliktu názvů. To znamená, že jeden uzel je typu `QueueTP<double>::Node` a druhý typu `QueueTP<char>::Node`.

Výpis 14.6 nabízí krátký program pro otestování této nové třídy. Vytváří frontu objektů třídy `String` a tudíž by měl být zkompilován spolu se souborem `strng2.cpp` (kapitola 11).

Výpis 14.6. nested.cpp

```
// nested.cpp - použití fronty s vnořenou třídou
// zkompilovat společně s strng2.cpp
#include <iostream>
using namespace std;
#include "strng2.h"
#include "queuetp.h"
int main()
{
    QueueTP<String> cs(5);
    String temp;
    while(!cs.isfull())
    {
      cout << "Zadejte prosim svoje jmeno. Obslouzeni budete "
           << "v poradi podle prichodu.\n"
              "Jmeno: ";
      cin >> temp;
```

```
        cs.enqueue(temp);
    }
    cout << "Fronta je zaplnena. Zpracovani zacina!\n";
    while (!cs.isempty())
    {
        cs.dequeue(temp);
        cout << "Nyni se zpracovava " << temp << "...\n";
    }
    return 0;
}
```

Zde je ukázka běhu:

```
Zadejte prosim svoje jmeno. Obslouzeni budete v poradi podle prichodu.
Jmeno: Kinsey Millhone
Zadejte prosim svoje jmeno. Obslouzeni budete v poradi podle prichodu.
Jmeno: Adam Dalgliesh
Zadejte prosim svoje jmeno. Obslouzeni budete v poradi podle prichodu.
Jmeno: Andrew Dalziel
Zadejte prosim svoje jmeno. Obslouzeni budete v poradi podle prichodu.
Jmeno: Kay Scarpetta
Zadejte prosim svoje jmeno. Obslouzeni budete v poradi podle prichodu.
Jmeno: Richard Jury
Fronta je zaplnena. Zpracovani zacina!
Nyni se zpracovava Kinsey Millhone...
Nyni se zpracovava Adam Dalgliesh...
Nyni se zpracovava Andrew Dalziel...
Nyni se zpracovava Kay Scarpetta...
Nyni se zpracovava Richard Jury...
```

Výjimky

Někdy program za běhu narazí na problémy, které znemožňují normálně pokračovat. Může se například snažit otevřít neexistující sobor nebo může požadovat více paměti, než je k dispozici, případně může narazit na nepřípustné hodnoty. Normálně se programátoři snaží takovým kalamitám předcházet. Výjimky jazyka C++ představují mocný a pružný nástroj pro zvládnutí těchto situací. Výjimky byly do jazyka C++ přidány nedávno a zatím ne všechny kompilátory je implementují.

Než začneme výjimky zkoumat, podíváme se na některé základní možnosti, které má programátor k dispozici. Jako příklad vezměme funkci počítající harmonický průměr dvou čísel. Harmonický průměr dvou čísel je definován jako obrácená hodnota průměru obrácených hodnot. To lze zjednodušit na následující výraz:

```
2.0 * x * y / (x + y)
```

Všimněte si, že pokud y představuje zápornou hodnotu x, vede tento vzorec k dělení nulou, což je dosti nežádoucí operace. Jedním z možných řešení je zavolat funkci `abort()` v případě, že hodnota jednoho parametru je zápornou hodnotou druhé hodnoty. Prototyp funkce `abort()` se nachází v hlavičkovém souboru cstdlib (nebo `stdlib.h`). Pokud je ta-

to funkce vyvolána při normální implementaci, pošle do standardního chybového proudu (stejný, který používá objekt cerr) zprávu typu „abnormal program termination" a ukončí program. Také vrátí operačnímu systému hodnotu, označující poruchu a která závisí na implementaci, případně tuto hodnotu vrátí nadřazenému procesu, pokud byl program spuštěn jiným programem. Na implementaci závisí, zda funkce abort() vyprázdní vyrovnávací paměti (oblasti paměti používané pro přenos dat mezi soubory). Můžete dát přednost funkci exit(), která vyrovnávací paměti vyprázdní, ale nezobrazí zprávu. Výpis 14.7 obsahuje krátký program používající funkci abort().

Výpis 14.7. error1.cpp

```
//error1.cpp – použití funkce abort()
#include <iostream>
using namespace std;
#include <cstdlib>
double hmean(double a, double b);
int main()
{
    double x, y, z;
    cout << "Zadejte dve cisla: ";
    while (cin >> x >> y)
    {
        z = hmean(x,y);
        cout << "Harmonicky prumer cisel " << x << " a " << y
            << " je " << z << "\n";
        cout << "Zadejte dalsi mnozinu cisel <k pro ukonceni>: ";
    }
    cout << "Nashledanou!\n";
    return 0;
}
double hmean(double a, double b)
{
    if (a == -b)
    {
        cout << "Neplatne parametry ve funkci hmean()\n";
        abort();
    }
    return 2.0 * a * b / (a + b);
}
```

Zde je ukázka běhu:

```
Zadejte dve cisla: 3 6
Harmonicky prumer cisel 3 a 6 je 4
Zadejte další mnozinu cisel <k pro ukonceni>: 10 -10
Neplatne parametry ve funkci hmean()
abnormal program termination
```

Všimněte si, že volání funkce abort() z funkce hmain() ukončí program přímo bez návratu do funkce main().

Ukončení programu by bylo možné zabránit, pokud by před voláním funkce `hmain()` byly kontrolovány hodnoty `x` a `y`. Avšak spoléhat se, že program takovou kontrolu umí provést, není příliš bezpečné.

Pružnější způsob než ukončení programu představuje označení problému pomocí návratové hodnoty funkce. Například členská funkce `get(void)` třídy `ostream` vrací obvykle ASCII-kód následujícího vstupního znaku, pokud však narazí na konec souboru, vrátí speciální hodnotu `EOF`. Tento způsob u funkce `hmean()` nefunguje. Jakákoli numerická hodnota by mohla být platnou návratovou hodnotou, takže pro označení problému žádná speciální hodnota neexistuje. V takové situaci můžete získat hodnotu volajícího programu pomocí parametru, kterým je ukazatel nebo reference a pomocí návratové hodnoty funkce stanovit její úspěšnost či neúspěšnost. Přetížené operátory >> skupiny tříd `istream` používají jednu z variant této techniky. Když volající program informujete o úspěšném či neúspěšném ukončení funkce, může namísto ukončení podniknout jiná opatření. Příklad tohoto postupu je ve výpisu 14.8. Funkce `hmain()` je předefinována na typ `bool` a její návratová hodnota označuje úspěch či neúspěch provedení. Pro získání výsledku je přidán třetí parametr.

Výpis 14.8. error2.cpp

```
#include <iostream>
  using namespace std;
  #include <cfloat> // (nebo float.h) pro DBL_MAX
  bool hmean(double a, double b, double * ans);
  int main()
  {
      double x, y, z;
      cout << "Zadejte dve cisla: ";
      while (cin >> x >> y)
      {
          if (hmean(x,y,&z))
              cout << "Harmonicky prumer cisel " << x << " a " << y
                   << " je " << z << "\n";
          else
              cout << "Jedna hodnota nesmi byt zapornou "
                   << "hodnotou druhe - opakujte.\n";
              cout << "Zadejte dalsi mnozinu cisel <k pro ukonceni>: ";
      }
      cout << "Nashledanou!\n";
      return 0;
  }
  bool hmean(double a, double b, double * ans)
  {
      if (a == -b)
      {
          *ans = DBL_MAX;
          return false;
      }
      else
      {
```

```
            *ans = 2.0 * a * b / (a + b);
            return true;
        }
    }
```

Zde je ukázka běhu:

```
Zadejte dve cisla: 3 6
Harmonicky prumer cisel 3 a 6 je 4
Zadejte další mnozinu cisel <k pro ukonceni>: 10 -10
Jedna hodnota nesmi byt zapornou hodnotou druhe - opakujte.
Zadejte další mnozinu cisel <k pro ukonceni>: 1 19
Harmonicky prumer cisel 1 a 19 je 1.9
Zadejte další mnozinu cisel <k pro ukonceni>: k
Nashledanou!
```

Poznámky k programu

Tento návrh programu obešel účinky špatných vstupních údajů a umožnil uživateli pokračovat v práci. Návrh se samozřejmě nespoléhá na uživatele, že bude kontrolovat návratovou hodnotu funkce, což ovšem programátoři ne vždy dělají. Aby byly výpisy ukázkových programů v této knize stručné, nekontroluje většina z nich například to, zda operátor `new` vrací nulový ukazatel nebo zda se objektu `cout` podařilo zpracovat výstup.

Jako třetí parametr můžete použít ukazatel nebo referenci. Mnoho programátorů upřednostňuje jako parametry vestavěných typů ukazatele, neboť je tak zřejmé, který parametr slouží k vrácení hodnoty.

Mechanismus výjimek

Nyní se podívejme jak můžete řešit problémy pomocí mechanismu výjimek. V C++ je výjimka reakcí na výjimečnou situaci vzniklou za běhu programu, jako například pokus o dělení nulou. Výjimky představují způsob, jak předat řízení z jedné části programu do jiné. Ošetření výjimky se skládá ze tří částí:

- Vyvolání výjimky
- Zachycení výjimky pomocí handleru
- Použití pokusného bloku (`try block`)

Výjimka je vyvolána při vzniku nějakého problému. Můžete například upravit funkci `hmean()` tak, aby místo volání funkce `abort()` vyvolala výjimku. Příkaz spouštějící výjimku je v podstatě skokem; to znamená, že programu řekne, aby skočil na příkazy na jiném místě. Vyvolání výjimky označuje klíčové slovo `throw`. Za ním následuje hodnota, například řetězec znaků nebo objekt, označující povahu výjimky.

Výjimku zachytíte pomocí ***handleru výjimky*** v místě programu, kde chcete problém řešit. Zachycení výjimky označuje klíčové slovo `catch`. Handler začíná klíčovým slovem `catch`, za kterým v kulatých závorkách následuje deklarace typu označující typ výjimky, na kterou bude handler reagovat. Za ním zase následuje blok programu, uzavřený ve složených závorkách, označující činnost, která se provede. Klíčové slovo `catch` slouží spo-

lečně s typem výjimky jako návěští určující místo v programu, na které by mělo provádění programu při vyvolání výjimky skočit. Handler výjimky se také nazývá záchytný blok (`catch block`).

Pokusný blok (`try block`) identifikuje blok kódu, pro který budou výjimky aktivovány. Samotný pokusný blok je označen klíčovým slovem `try`, za kterým následuje blok programu, uzavřený ve složených závorkách, pro který budou výjimky zaregistrovány.

Jak tyto tři prvky zapadají dohromady pochopíte nejsnáze, když se podíváte na krátký příklad, který nabízí výpis 14.9.

Výpis 14.9. error3.cpp

```
//error3.cpp
#include <iostream>
using namespace std;
double hmean(double a, double b);
int main()
{
    double x, y, z;
    cout << "Zadejte dve cisla: ";
    while (cin >> x >> y)
    {
        try {                   // začátek pokusného bloku
            z = hmean(x, y);
        }                       // konec pokusného bloku
        catch (char * s)        // začátek handleru výjimky
        {
            cout << s << "\n";
            cout << "Zadejte novou dvojici cisel: ";
            continue;
        }                           // konec handleru
        cout << "Harmonicky prumer cisel " << x << " a " << y
            << " je " << z << "\n";
        cout << "Zadejte dalsi mnozinu cisel <k pro ukonceni>: ";
    }
    cout << "Nashledanou!\n";
    return 0;
}
double hmean(double a, double b)
{
    if (a == -b)
        throw "Neplatne parametry ve funkci hmean(): a = -b neni povo-
leno";
    return 2.0 * a * b / (a + b);
}
```

Zde je ukázka běhu:

```
Zadejte dve cisla: 3 6
Harmonicky prumer cisel 3 a 6 je 4
Zadejte další mnozinu cisel <k pro ukonceni>: 10 -10
```

```
Neplatne parametry ve funkci hmean(): a = -b neni povoleno
Zadejte novou dvojici cisel: 1 19
Harmonicky prumer cisel 1 a 19 je 1.9
Zadejte další mnozinu cisel < k pro ukonceni>: k
Nashledanou!
```

Poznámky k programu

Pokusný blok vypadá takto:

```
try {                // začátek pokusného bloku
   z = hmean(x,y);
}                    // konec pokusného bloku
```

Jestliže některý z příkazů v tomto bloku vede k vyvolání výjimky, ošetří ji záchytné bloky následující za tímto blokem. Kdyby program volal funkci `hmean()` někde mimo tento (nebo nějaký jiný) pokusný blok, nebylo by možné výjimku ošetřit.

Vyvolání výjimky vypadá následovně:

```
if (a == -b)
   throw "Neplatne parametry ve funkci hmean(): a = -b neni povoleno";
```

V tomto případě je vyvolanou výjimkou řetězec „`Neplatne parametry ve funkci hmean(): a = -b neni povoleno`". Vyvolání výjimky se trochu podobá příkazu return v tom, že ukončuje provádění funkce. Nevrací však řízení volajícímu programu, ale způsobí, že program zpětně prochází sled aktuálně volaných funkcí až najde funkci, obsahující pokusný blok. Ve výpisu 14.9 je touto funkcí funkce volající. Brzy uvidíte příklad, kde se procházení týká více než jedné funkce. V uvedeném příkladě předává výjimka řízení programu zpět do funkce `main()`. Program hledá handler výjimky (za pokusným blokem) odpovídající typu vyvolané výjimky.

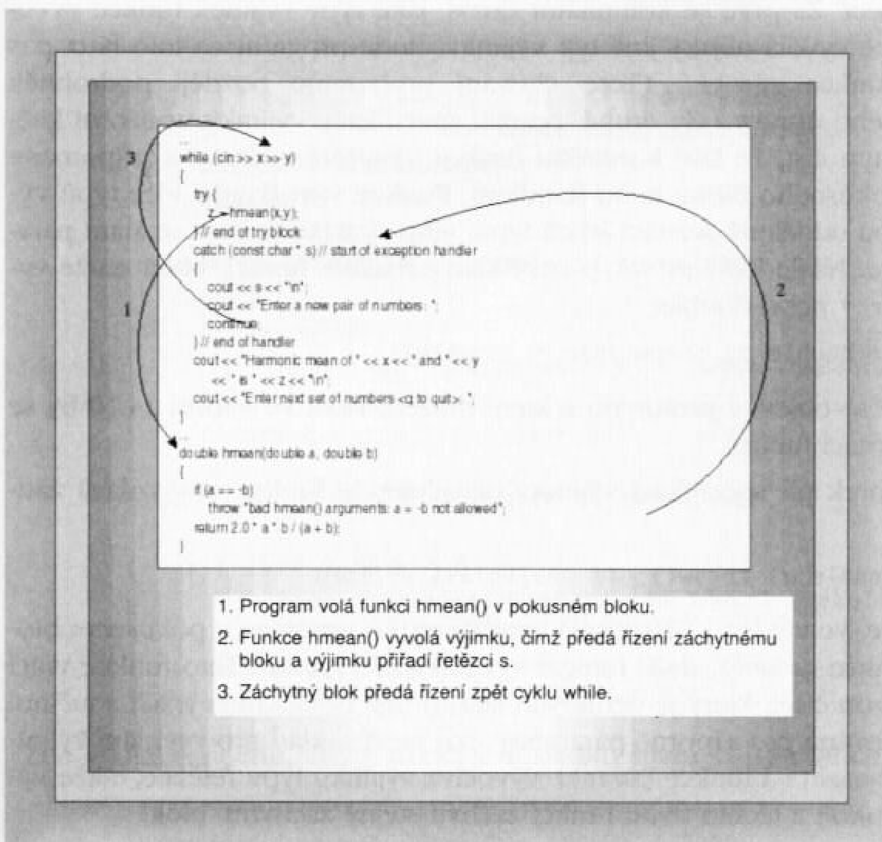
Handler neboli záchytný blok vypadá takto:

```
catch (char * s) // začátek handleru výjimky
{
     cout << s << "\n";
     cout << "Zadejte novou dvojici cisel: ";
     continue;
}                // konec handleru
```

Trochu to připomíná definici funkce, ale není to tak. Klíčové slovo `catch` identifikuje handler a `char * s` znamená, že tento handler porovnává vyvolanou výjimku, kterou je řetězec. Tato deklarace proměnné `s` se chová jako definice parametru funkce v tom, že této proměnné je přiřazena odpovídající vyvolaná výjimka. Jestliže výjimka odpovídá tomuto handleru, provede program kód ve složených závorkách.

Jestliže program provede příkazy v pokusném bloku bez vyvolání výjimky, přeskočí záchytné bloky a přejde na první příkaz za handlery. Když tedy byly v ukázkovém příkladu zpracovány hodnoty 3 a 6, přešlo provádění programu přímo na příkaz pro výstup, oznamující výsledek.

Projděme si události v ukázkovém běhu po předání hodnot 10 a –10 funkci hmean(). Test if způsobí ve funkci `hmean()` vyvolání výjimky. Tím se provádění funkce ukončí. Zpětným hledáním program určí, že funkce `hmean()` byla volána z pokusného bloku ve funkci `main()`. Potom hledá záchytný blok s typem odpovídajícím typu výjimky. Jediný existující záchytný blok má parametr typu `char *` a ten odpovídá. Po nalezení odpovídajícího typu přiřadí program proměnné s řetězec „`Neplatne parametry ve funkci hmean(): a = -b neni povoleno`". Následuje provedení kódu v handleru. Nejdříve je vytisknut řetězec s představující zachycenou výjimku. Potom jsou vytisknuty pokyny pro uživatele vyzývající k vložení nových dat. Nakonec je proveden příkaz `continue`, který způsobí, že program přeskočí zbytek smyčky `while` a skočí opět na její začátek. Skutečnost, že příkaz `ccontinue` vrátí program na začátek smyčky dokládá, že příkazy handleru jsou součástí smyčky a že řádek s příkazem catch se chová jako návěští řídící chod programu (viz obrázek 14.2).



Obrázek 14.2. Chod programu obsahujícího výjimky

Možná vás zajímá, co se stane, jestliže funkce vyvolá výjimku a pokusný blok nebo odpovídající handler neexistují. Implicitně program nakonec zavolá funkci `abort()`, ale toto chování můžete upravit. K tomuto tématu se vrátíme později.

Univerzálnost výjimek

Výjimky jazyka C++ nabízí univerzálnost, neboť pokusné bloky umožňují zvolit si kód, který budou výjimky kontrolovat a handlery umožňují určit, co se stane. Například ve vý-

pisu 14.9 byl pokusný blok umístěn ve smyčce, takže provádění programu po ošetření výjimky pokračovalo ve smyčce. Vložíte-li smyčku do pokusného bloku, může výjimka předat provádění programu mimo tuto smyčku a tak ji ukončit. To ilustruje výpis 14.10. Demonstruje také dva další body:

- Specifikací výjimky v definici funkce můžete označit druhy výjimek, které budou vyvolány.
- Záchytný blok může ošetřit více zdrojů výjimek.

Chcete-li pomocí prototypu funkce určit druhy vyvolávaných výjimek, přidejte specifikaci výjimky, která se skládá z klíčového slova `throw` následovaného seznamem typů výjimek oddělených čárkou a uzavřených v kulatých závorkách.:

```
double hmean(double a, double b) throw(char *);
```

Tím dosáhnete dvou věcí. Za prvé se kompilátor dozví, jaké typy výjimek funkce vyvolává. Jestliže pak funkce vyvolá nějaký jiný typ výjimky, bude program na toto faux pas reagovat zavoláním funkce `abort()`. (Toto chování probereme později podrobněji a ukážeme si způsob jeho úpravy.) Za druhé, použití specifikace výjimky upozorní každého, kdo bude prototyp číst, že tato konkrétní funkce vyvolává výjimky a připomene mu možnost použití pokusného bloku nebo handleru. Funkce vyvolávající více typů výjimek mohou mít čárkou oddělený seznam jejich typů; jeho syntaxe imituje seznam parametrů prototypu funkce. Následující prototyp například označuje funkci, která může vyvolat výjimky typu `char *` nebo double:

```
double multi_err(double z) throw(char *, double);
```

Stejné informace, které se objeví v prototypu a které můžete vidět ve výpisu 14.10 by se měly objevit také v definici funkce.

Použití prázdných závorek při specifikaci výjimky označuje, že funkce nevyvolává žádnou výjimku:

```
double simple(double z) throw(); // nevyvolává výjimku
```

Jak jsme se zmínili dříve, ve výpisu 14.10 je celá smyčka `while` umístěna v pokusném bloku. Přidána je také funkce `gmean()`, další funkce vyvolávající výjimku. Tato funkce vrací geometrický průměr dvou čísel, který je definován jako druhá odmocnina jejich součinu. Tato funkce není definována pro záporné parametry, což tvoří základ pro vyvolání výjimky. Stejně jako funkce `hmean()` i funkce `gmean()` vyvolává výjimku typu řetězec, takže výjimku vyvolanou kteroukoli z těchto dvou funkcí zachytí stejný záchytný blok.

Výpis 14.10. error4.cpp

```
//error4.cpp
  #include <iostream>
  using namespace std;
  #include <cmath>// nebo math.h, uživatelé systému Unix mohou potřebovat
                                                    //přepínač -lm
  double hmean(double a, double b) throw(char *);
  double gmean(double a, double b) throw(char *);
  int main()
```

```
{
    double x, y, z;
    cout << "Zadejte dve cisla: ";
    try {               // začátek pokusného bloku
        while (cin >> x >> y)
        {
            z = hmean(x, y);
            cout << "Harmonicky prumer cisel " << x << " a " << y
                 << " je " << z << "\n";
            cout << "Geometricky prumer cisel " << x << " a " << y
                 << " je " << gmean(x, y) << "\n";
            cout << "Zadejte dalsi mnozinu cisel <k pro ukonceni>: ";
        }
    }                   // konec pokusného bloku
    catch (char * s) // začátek záchytného bloku
    {
        cout << s << "\n";
        cout << "Bohuzel, dal se nedostanete. ";
    }                  // konec záchytného bloku
    cout << "Nashledanou!\n";
    return 0;
}
double hmean(double a, double b) throw(char *)
{
    if (a == -b)
        throw "Neplatne parametry ve funkci hmean(): a = -b neni
                                                        povoleno.";
    return 2.0 * a * b / (a + b);
}
double gmean(double a, double b) throw(char *)
{
    if (a < 0 || b < 0)
        throw "Neplatne parametry ve funkci gmean(): zaporne hodnoty "
                                                  „nejsou povoleny.";
    return sqrt(a * b);
}
```

Zde je ukázka běhu, který skončí v důsledku špatných vstupních dat pro funkci `hmean()`:

```
Zadejte dve cisla: 1 100
Harmonicky prumer cisel 1 a 100 je 1.9802
Geometricky prumer cisel 1 a 100 je 10
Zadejte další mnozinu cisel <k pro ukonceni>: 10 -10
Neplatne parametry ve funkci hmean(): a = -b neni povoleno.
Bohuzel, dal se nedostanete. Nashledanou!
```

Protože handler výjimky se nachází mimo cyklus, špatné vstupní údaje tento cyklus ukončí. Jakmile program ukončí provádění kódu v handleru, přejde na další řádek, který vytiskne řetězec „Nashledanou!".

Pro porovnání je zde ukázka běhu, který skončí v důsledku chybných vstupních údajů pro funkci `gmean()`:

```
Zadejte dve cisla: 1 100
```

```
Harmonicky prumer cisel 1 a 100 je 1.9802
Geometricky prumer cisel 1 a 100 je 10
Zadejte další mnozinu cisel <k pro ukonceni>: 3 -15
Harmonicky prumer cisel 3 a -15 je 7.5
Neplatne parametry ve funkci gmean(): zaporne hodnoty nejsou povoleny.
Bohuzel, dal se nedostanete. Nashledanou!
```

Zpráva odhalí, která výjimka byla ošetřena.

Vícenásobné pokusné bloky

Máte mnoho možností, jak pokusné bloky vytvářet. Mohli byste například ošetřit každé volání funkce zvlášť tím, že byste je umístili do vlastních pokusných bloků. Potom by bylo možné naprogramovat pro dvě možné výjimky dvě různé reakce, jak ukazuje následující kód:

```
while (cin gt;> x >> y)
{
try {                                    // pokusný blok č. 1
    z = hmean(x,y);
}                                         // konec pokusného bloku č. 1
catch (char * s)                         // začátek záchytného bloku č. 1
{
    cout << s << "\n";
    cout << "Zadejte novou dvojici cisel: ";
    continue;
}                                         // konec záchytného bloku  č. 1
cout << "Harmonicky prumer cisel " << x << " a " << y
     << " je " << z << "\n";
try {                                    // pokusný blok č. 2
    z = gmean(x,y);
}                                         // konec pokusného bloku č. 2
catch (char * s)                         // začátek záchytného bloku č. 2
{
    cout << s << "\n";
    cout << "Vstup dat ukoncen!\n";
    break;
}                                         // konec záchytného bloku č. 2
cout << "Zadejte další mnozinu cisel <k pro ukonceni>:    ";
}
```

Další možností je vnoření pokusných bloků, jak ukazuje další příklad:

```
try {                                    // vnější pokusný blok
  while (cin >> x >> y)
  {
  try {                                 // vnitřní pokusný blok
z = hmean(x,y);
  }                                      // konec vnitřního pokusného bloku
  catch (char * s)                      // vnitřní záchytný blok
  {
  cout << s << "\n";
cout << " Zadejte novou dvojici cisel: ";
```

```
    continue;
    }                               // konec vnitřního záchytného bloku
    cout << "Harmonicky prumer cisel " << x << " a " << y
         << " je " << z << "\n";
    cout << "Geometricky prumer " << x << " a " << y
         << " je " << gmean(x,y) << "\n";
    cout << "Zadejte další mnozinu cisel <k pro ukonceni>: ";
    }
}                                   // konec vnějšího pokusného bloku
catch (char * s)                    // vnější záchytný blok
{
    cout << s << "\n";
    cout << "Bohuzel, dal se nedostanete. ";
}                                   // konec vnějšího záchytného bloku
```

Zde je výjimka vyvolaná funkcí `hmean()` zachycena vnitřním handlerem výjimky, což umožňuje pokračování smyčky. Ale výjimka vyvolaná funkcí `gmean()` je zachycena handlerem vnějším, což vede k ukončení smyčky.

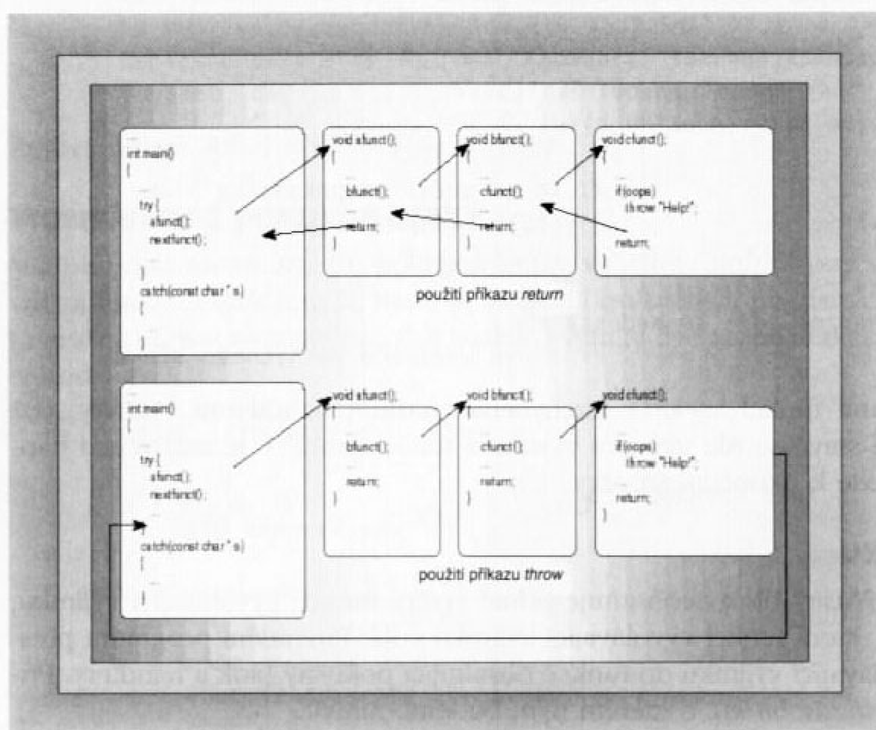
Uvolnění zásobníku

Předpokládejme, že pokusný blok neobsahuje přímé volání funkce vyvolávající výjimku, ale že obsahuje funkci, která funkci vyvolávající výjimku volá. Provádění programu přesto skočí z funkce vyvolávající výjimku do funkce obsahující pokusný blok a handlery. Přitom dochází k *uvolnění zásobníku*, o kterém nyní budeme mluvit.

Nejdříve se podíváme, jak C++ normálně ošetřuje volání funkce a návrat z ní. Při volání funkcí jsou informace běžně ukládány do zásobníku (kapitola 8). Konkrétně program uloží do zásobníku adresu instrukce (adresu návratu) volající funkce. Když volaná funkce skončí, rozhodne se program na základě této adresy, ve kterém místě bude provádění pokračovat. Při volání funkce jsou do zásobníku uloženy rovněž její parametry, kde se s nimi pracuje jako s automatickými proměnnými. Jestliže volaná funkce vytvoří nějaké nové automatické proměnné, budou také přidány do zásobníku. Když volaná funkce zavolá další funkci, budou informace o ní opět uloženy do zásobníku a tak dále. Když funkce skončí, přesune se provádění programu na adresu uloženou při jejím zavolání a uvolní místo na vrcholu zásobníku. Funkce se tedy normálně navrátí do funkce a každá funkce při svém ukončení uvolňuje automatické proměnné. Pokud je automatickou proměnnou objekt nějaké třídy, pak je zavolán případný konstruktor této třídy.

Nyní předpokládejme, že se funkce neukončí příkazem `return`, ale vyvoláním výjimky. Program opět uvolní paměť ze zásobníku. Nezastaví se však na první adrese návratu, ale bude pokračovat v uvolňování zásobníku, dokud nenarazí na adresu návratu, která se nachází v pokusném bloku (viz obrázek 14.3). Program potom nepokračuje prvním příkazem za voláním funkce, ale řízení je předáno handlerům na konci bloku. Tento proces se nazývá *uvolnění zásobníku*. Jednou z důležitých vlastností mechanismu výjimek je stejně jako u návratu z funkce vyvolání destruktorů pro všechny automatické objekty tříd v zásobníku. Návrat z funkce však zpracovává pouze objekty vložené do zásobníku touto funkcí, zatímco vyvolání výjimky zpracovává všechny objekty vložené do zásobníku celou řadou funkcí mezi pokusným blokem a vyvoláním výjimky. Bez uvolnění zásobníku

by vyvolání výjimky nespustilo destruktory automatických objektů uložených do zásobníku funkcemi, ležícími mezi pokusným blokem a funkcí, která výjimku vyvolala.



Obrázek 14.3. Příkaz throw a příkaz return

Výpis 14.11 obsahuje příklad uvolnění zásobníku. V tomto příkladu volá funkce `main()` funkci `details()` a ta zase volá funkci `hmean()`. Když funkce `hmean()` vyvolá výjimku, je řízení programu vráceno funkci `main()`, kde je výjimka zachycena. Během tohoto procesu jsou uvolněny automatické proměnné, představující parametry funkcí `hmean()` a `details()`.

Výpis 14.11. error5.cpp

```
//error5.cpp
#include <iostream>
using namespace std;
double hmean(double a, double b) throw(char *);
void details(double a, double b) throw(char *);
int main()
{
    double x, y;
    cout << "Zadejte dve cisla: ";
    try {
  while (cin >> x >> y)
      details(x, y);
    }
    catch (char * s)
```

```
    {
    cout << s << "\n";
    cout << "Bohuzel, dal se nedostanete. ";
      }
      cout << "Nashledanou!\n";
      return 0;
  }
  void details(double a, double b) throw(char *){
      cout << "Harmonicky prumer cisel " << a << " a " << b
    << " je " << hmean(a, b) << "\n";
      cout << "Zadejte dalsi mnozinu cisel <k pro ukonceni>: ";
  }
  double hmean(double a, double b) throw(char *)
  {
      if (a == -b)
    throw "Neplatne parametry ve funkci hmean(): a = -b neni povoleno.";
      return 2.0 * a * b / (a + b);
    }
```

Následuje ukázka běhu programu. Všimněte si, že po vyvolání výjimky program přeskočí přímo do funkce `main()` a funkce `details()` nedostane příležitost zobrazit text, který by se zobrazil při normálním ukončení funkce `hmean()`.

```
Zadejte dve cisla: 3 15
Harmonicky prumer cisel 3 a 15 je 5
Zadejte další mnozinu cisel <k pro ukonceni>: 20 -20
Neplatne parametry ve funkci hmean(): a = -b neni povoleno.
Bohuzel, dal se nedostanete. Nashledanou!
```

Další možnosti

Handler můžete nastavit tak, aby zachytil jakoukoli výjimku. Jestliže je také vnořen pokusný blok, můžete předat řízení z vnořeného handleru do handleru, obsahujícího tento pokusný blok.

Chcete-li zachytit jakoukoli výjimku, použijte jako typ výjimky výpustku:

```
catch (...) { // příkazy }
```

Pro předání kontroly nadřazenému pokusnému bloku použijte příkaz throw bez uvedení výjimky:

```
catch (char * s)
{
        cout << "Vyjimka zachycena ve vnitrnim cyklu.\n";
        throw; // předání výjimky nadřazenému pokusnému bloku
}
```

Tento příklad vypíše zprávu a potom předá řízení nadřízenému pokusnému bloku, kde program bude opět hledat handler odpovídající původní vyvolané výjimce.

Všimněte si, že existuje více způsobů, jak může jeden pokusný blok obsahovat blok jiný. Jeden způsob je vnoření jednoho bloku do druhého, jak jsme probírali dříve. Další možností je pokusný blok obsahující volání funkce, která vyvolá funkci obsahující pokusný

blok. V prvním případě by se řízení předalo vnějšímu pokusnému bloku, zatímco v druhém by program hledal další pokusný blok pomocí uvolnění zásobníku.

Výjimky a třídy

Výjimky neslouží pouze k ošetření chyb v běžných funkcích. Mohou být také součástí návrhu tříd. Výjimku by například mohl vyvolat konstruktor při neúspěšném volání operátoru `new`. Nebo může výjimku vyvolat přetížený operátor `[]` třídy reprezentující pole, jestliže se index ocitne mimo meze pole. Často je užitečné, jestliže výjimku mohou doprovázet nějaké informace, například hodnota neplatného indexu. V takovém případě by bylo možné použít například výjimku typu `int`, užitečnější je ale vyvolat výjimku představovanou objektem. Typ objektu pomůže identifikovat zdroj výjimky. V předchozím příkladě s funkcemi `hmean()` a `qmean()` se vyskytl problém, neboť obě procházely stejný typ výjimky (`char *`), a proto bylo obtížné vytvořit záchytné bloky rozlišující mezi těmito dvěma funkcemi. Pomocí objektů můžete navrhnout jiný typ objektu pro každou výjimku, kterou chcete zachytit. A objekt samotný může nést potřebné informace.

Tip

Máte-li funkci vyvolávající výjimku, definujte třídu výjimky, která se použije jako typ výjimky.

V praxi je běžné vyvolání výjimky pomocí objektu a jeho zachycení pomocí odkazu:

```
class problem {...};
...
void super()
{
    ...
    if (oh_no)
    {
        problem oops(); // vytvoří objekt
        throw oops ;    // vyvolá výjimku
    ...
}
...
try {
    super();
}
catch(problem & p)
{
...
}
```

Mimochodem, zatímco mechanismus výjimek se hodně podobá předávání parametrů, existuje i několik odlišností. Při vyvolání výjimky například kompilátor vždy vytvoří dočasnou kopii, takže v předcházejícím kódu by proměnná `p` neodkazovala na objekt `oops`, ale na jeho kopii. To je dobrá věc, protože po skončení funkce `super()` již objekt `oops` neexistuje. Často je jednodušší zkombinovat vytvoření objektu s vyvoláním výjimky:

```
throw oops();    // vytvoří objekt s vyvoláním výjimky
```

Jestliže se výjimka vztahuje na procesy ve třídě, bývá často užitečné, je-li typ výjimky definován jako vnořená třída. Při tomto způsobu jednak typ výjimky označí třídu vyvolávající výjimku, jednak je znemožněn konflikt názvů. Předpokládejme například, že máte třídu `ArrayDbE`, ve které je veřejně deklarována další třída `BadIndex`. Jestliže operátor [] najde špatnou hodnotu indexu, může vyvolat výjimku typu `BadIndex`. Handler takové výjimky by vypadal následovně:

```
catch (const ArrayDbE::BadIndex &) { …}
```

Díky kvalifikátoru `ArrayDbE::` poznáte, že BadIndex je deklarován ve třídě `ArrayDbE`. Čtenář je také informován, že tento handler je určen pro výjimky generované objekty třídy `ArrayDbE`. Název `BadIndex` dává čtenáři tušit povahu této výjimky. To zní dosti lákavě, pojďme tedy tuto myšlenku rozvinout. Konkrétně přidáme výjimky třídě `ArrayDb`, která byla prvně vytvořena v kapitole 13.

Do hlavičkového souboru přidejte výjimku třídy `BadIndex`, tedy třídy, definující objekty použité jako výjimky. Jak bylo naznačeno výše, výjimka se použije při neplatných hodnotách indexu a bude neplatnou hodnotu obsahovat. Všimněte si, že deklarace vnořené třídy tuto třídu pouze popisuje, ale nevytváří žádné objekty. Metody třídy vytvoří její objekty v případě vyvolání výjimek typu `BadIndex`. Také si všimněte, že vnořená třída je veřejná. Díky tomu mají záchytné bloky k tomuto typu přístup. Výpis 14.12 obsahuje nový hlavičkový soubor. Zbytek definice, až na změnu názvu třídy, je stejný jako definice třídy `ArrayDb` v kapitole 13 s tím, že prototypy metod mají kvalifikátor pro označení výjimek, které mohou vyvolat. Prototyp

```
virtual double & operator[](int i);
```

je tedy nahrazen prototypem

```
virtual double & operator[](int i) throw(BadIndex &);
```

a tak dále. (Stejně jako u běžných parametrů funkcí je při vyvolání výjimky obvykle lepší předávat místo objektů reference.)

Výpis 14.12. arraydbe.h

```
// arraydbe.h — definice třídy pro pole s použitím výjimek
#ifndef _ARRAYDBE_H_
#define _ARRAYDBE_H_
#include <iostream>
using namespace std;
class ArrayDbE
{
private:
    unsigned int size; // počet prvků pole
protected:
    double * arr;      // adresa prvního prvku
public:
    class BadIndex     // třída výjimky ošetřující špatný index
    {
    public:
```

```
        int badindex; // problematická hodnota indexu
        BadIndex(int i) : badindex(i) {}
    };
    ArrayDbE();                          // bezparametrický konstruktor
    // vytvoří pole ArrayDbE s n prvky nastavenými na hodnotu val
    ArrayDbE(unsigned int n, double val = 0.0);
    // vytvoří pole ArrayDbE s n prvky inicializovanými podle prvků pole pn
    ArrayDbE(const double * pn, unsigned int n);
    // kopírovací konstruktor
    ArrayDbE(const ArrayDbE & a);
    virtual ~ArrayDbE();                 // destruktor
    unsigned int ArSize() const;         // vrátí velikost pole
    double Average() const;              // vrátí průměr hodnot v poli
                                         // přetížené operátory
        // indexování pole, přiřazení
    virtual double & operator[](int i) throw(BadIndex &);
        // indexování pole
    virtual const double & operator[](int i) const throw(BadIndex &);
    ArrayDbE & operator=(const ArrayDbE & a);
    friend ostream & operator<<(ostream & os, const ArrayDbE & a);
};
#endif
```

Dále musíte dodat metody třídy. Jedná se o stejné metody jako v kapitole 12 s přidáním několika vyvolání výjimek. Vzhledem k tomu, že místo volání funkce `exit()` existuje přetížený `operátor[]` vyvolávající výjimku, není již třeba vkládat do programu hlavičkový soubor cstdlib. Výsledek těchto úprav je ve výpisu 14.13.

Výpis 14.13. arraydbe.cpp

```
// arraydbe.cpp — metody třídy ArrayDbE
#include <iostream>
using namespace std;
#include "arraydbe.h"
// bezparametrický konstruktor
ArrayDbE::ArrayDbE()
{
   arr = NULL;
   size = 0;
}
// vytvoří pole o n prvcích, každý je nastaven na hodnotou val
ArrayDbE::ArrayDbE(unsigned int n, double val)
{
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
       arr[i] = val;
}
// inicializuje objekty třídy ArrayDbE pomocí klasického pole
ArrayDbE::ArrayDbE(const double *pn, unsigned int n)
{
```

```
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
        arr[i] = pn[i];
}
// inicializuje objekt třídy ArrayDbE pomocí jiného objektu této třídy
ArrayDbE::ArrayDbE(const ArrayDbE & a)
{
    size = a.size;
    arr = new double[size];
    for (int i = 0; i < size; i++)
        arr[i] = a.arr[i];
}
ArrayDbE::~ArrayDbE()
{
    delete [] arr;
}
double ArrayDbE::Average() const
{
    double sum = 0;
    int i;
    int lim = ArSize();
    for (i = 0; i < lim; i++)
        sum += arr[i];
    if (i > 0)
        return sum / i;
    else
    {
        cerr << "V poli zaznamu nejsou zadne udaje\n";
        return 0;
    }
}
// vrátí velikost pole
unsigned int ArrayDbE::ArSize() const
{
    return size;
}
// umožní uživateli přistupovat k prvkům pomocí indexu (přiřazení povo-
leno)
double & ArrayDbE::operator[](int i) throw(BadIndex &)
{
    // kontrola indexu
    if (i < 0 || i >= size)
        throw BadIndex(i);
    return arr[i];
}
// umožní uživateli přistupovat k prvkům pomocí indexu (přiřazení zaká-
záno)
const double & ArrayDbE::operator[](int i)const throw(BadIndex &)
{
    // kontrola indexu
    if (i < 0 || i >= size)
```

```
        throw BadIndex(i);
    return arr[i];
}
// definuje přiřazení třídy
ArrayDbE & ArrayDbE::operator=(const ArrayDbE & a)
{
    if (this == &a)   // je-li objekt přiřazení sám sobě,
        return *this; // nic nedělej
    delete arr;
    size = a.size;
    arr = new double[size];
    for (int i = 0; i < size; i++)
        arr[i] = a.arr[i];
    return *this;
}
// úspornější rychlý výstup, 5 hodnot na řádek
ostream & operator<<(ostream & os, const ArrayDbE & a)
{
    int i;
    for (i = 0; i < a.size; i++)
    {
        os << a.arr[i] << " ";
        if (i % 5 == 4)
            os << "\n";
    }
    if (i % 5 != 0)
        os << "\n";
    return os;
}
```

Všimněte si, že místo řetězců jsou výjimkami objekty. Také si všimněte, že při vyvolání výjimky se pro vytvoření a inicializaci objektů výjimek použije konstruktor:

```
if (i < 0 || i >= size)
    throw BadIndex(i);  // vytvoří a inicializuje objekt třídy BadIndex
```

A co zachytávání takových výjimek? Výjimky jsou objekty, ne řetězce znaků, záchytný blok proto musí tuto skutečnost odrážet. Protože výjimka je také vnořeným typem, musí být použit operátor rozlišení.

```
try {
    ...
}
catch(ArrayDbE::BadIndex &) {
    ...
}
```

Tento proces demonstruje krátký program ve výpisu 14.14.

Výpis 14.14. exceptar.cpp

```
// exceptar.cpp - použití třídy ArrayDbE
// zkompilovat s arraydbe.cpp
#include <iostream>
using namespace std;
#include "arraydbe.h"
const int Players = 5;
int main()
{
    try {
        ArrayDbE Team(Players);
        cout << "Zadejte pro 5 "
             << " nejlepsich hracu libovolne procentualni ohodnoceni
jako"
             << "desetinnou cast cisla:\n";
        int player;
        for (player = 0; player < Players; player++)
        {
            cout << "Hrac " << (player + 1) << ": % = ";
            cin >> Team[player];
        }
        cout.precision(1);
        cout.setf(ios_base::showpoint);
        cout.setf(ios_base::fixed,ios_base::floatfield);
        cout << "Recapitulace procent:\n";
        for (player = 0; player <= Players; player++)
            cout << "Hrac #" << (player + 1) <<": "
                 << 100.0 * Team[player] << "%\n";
    }                   // konec pokusného bloku
    catch (ArrayDbE::BadIndex & bi)   // začátek handleru
    {
        cout << "ArrayDbE vyjimka: "
            << bi.badindex << " je neplatna hodnota indexu.\n";
    }                   // konec handleru
    cout << "Nashledanou!\n";
   return 0;
}
```

Všimněte si, že ve druhém cyklu jsou záměrně překročeny hranice pole, což spustí výjimku. Zde je ukázka běhu programu:

Zadejte pro 5 nejlepsich hracu libovolne procentualni ohodnoceni jako desetinnou cast cisla:

```
Hrac 1: % = 0.923
Hrac 2: % = 0.858
Hrac 3: % = 0.821
Hrac 4: % = 0.744
Hrac 5: % = 0.697
Recapitulace procent:
Hrac #1: 92.3%
Hrac #2: 85.8%
```

```
Hrac #3: 82.1%
Hrac #4: 74.4%
Hrac #5: 69.7%
ArrayDbe vyjimka: 5 je neplatna hodnota indexu.
Nashledanou!
```

Protože je cyklus v pokusném bloku, vyvolání výjimky ho ukončí, jakmile je řízení programu předáno záchytnému bloku následujícímu za blokem pokusným.

Pamatujte, že proměnné definované v bloku, včetně bloku pokusného, jsou lokálními proměnnými tohoto bloku. Například proměnná player ve výpisu 14.14 přestane být definována, jakmile řízení program přejde za pokusný blok.

Výjimky a dědičnost

Dědičnost a výjimky se ovlivňují v několika směrech. Za prvé, jestliže třída obsahuje veřejné vnořené třídy výjimek, pak odvozená třída tyto třídy výjimek zdědí. Za druhé, od existujících tříd výjimek můžete odvodit nové třídy výjimek. Na obě tyto možnosti se podíváme v následujícím příkladu.

Nejdříve od třídy `ArrayDbE` odvodíme třídu `LimitArE`. Třída `LimitArE` umožní, aby hodnoty indexů pole začínaly jinou hodnotou než 0. Toho lze dosáhnout uložením hodnoty představující počáteční index a předefinováním funkcí. Vnitřně bude indexování pole stále začínat hodnotou 0. Ale pokud například určíme jako počáteční index hodnotu 1900, pak metoda `operator[]()` přeloží externí index 1908 na interní index 1908-1900, čili 8. Podrobnosti obsahuje výpis 14.15.

Výjimka `BadIndex` deklarovaná ve třídě `ArrayDbE` ukládala hodnotu nevyhovujícího indexu. V případě měnitelných mezí by bylo vhodné, kdyby výjimka také ukládala správný rozsah indexů. Toho můžete dosáhnout odvozením nové třídy výjimky od třídy `BadIndex`.

```
class SonOfBad : public ArrayDbe::BadIndex
{
public:
        int l_lim; // dolni mez indexu
        int u_lim; // horni mez indexu
        SonOfBad(int i, int l, int u) : BadIndex(i),
                                l_lim(l), u_lim(u) {}
};
```

Deklaraci třídy SonOfBad můžete vnořit do deklarace třídy `LimitArE`. Výsledek je vidět ve výpisu 14.15.

Výpis 14.15. limarre.h

```
// limarre.h - třida LimitArE s využitim výjimek
  #ifndef _LIMARRE_H_
  #define _LIMARRE_H_

  #include "arraydbe.h"
```

```
class LimitArE : public ArrayDbE
{
public:
    class SonOfBad : public ArrayDbE::BadIndex
    {
    public:
        int l_lim;
        int u_lim;
        SonOfBad(int i, int l, int u) : BadIndex(i),
                l_lim(l), u_lim(u) {}
    };
private:
    unsigned int low_bnd;        // nová datová položka
protected:
    // kontrola mezí handleru
    virtual void ok(int i) const throw(SonOfBad &);
public:
// konstruktory
    LimitArE() : ArrayDbE(), low_bnd(0) {}
    LimitArE(unsigned int n, double val = 0.0)
                : ArrayDbE(n,val), low_bnd(0) {}
    LimitArE(unsigned int n, int lb, double val = 0.0)
                : ArrayDbE(n, val), low_bnd(lb) {}
    LimitArE(const double * pn, unsigned int n)
                : ArrayDbE(pn, n), low_bnd(0) {}
    LimitArE(const ArrayDbE & a) : ArrayDbE(a), low_bnd(0) {}
//nové metody
    void new_lb(int lb) {low_bnd = lb;}  // nastavení dolní meze
    int lbound() {return low_bnd;}        // hodnota dolní meze
    int ubound() {return ArSize() + low_bnd - 1;} // horní mez
// předefinované operátory
    double & operator[](int i);
    const double & operator[](int i) const;
};
#endif
```

V tomto návrhu je kontrola indexů přesunuta z přetížených metod operátoru [] do metody ok(), kterou přetížené metody operátoru [] volají. Je to tedy metoda `LimitArE::ok()`, která vyvolává výjimku `SonOfBad`. Metody třídy jsou obsaženy ve výpisu 14.16.

Výpis 14.16. limarre.cpp

```
// limarre.h – třída LimitArE s využitím výjimkamiek
#ifndef _LIMARRE_H_
#define _LIMARRE_H_

#include "arraydbe.h"

class LimitArE : public ArrayDbE
{
public:
```

```
        class SonOfBad : public ArrayDbE::BadIndex
        {
        public:
            int l_lim;
            int u_lim;
            SonOfBad(int i, int l, int u) : BadIndex(i),
                    l_lim(l), u_lim(u) {}
        };
    private:
        unsigned int low_bnd;        // nová datová položka
    protected:
        // kontrola mezí handleru
        virtual void ok(int i) const throw(SonOfBad &);
    public:
    // konstruktory
        LimitArE() : ArrayDbE(), low_bnd(0) {}
        LimitArE(unsigned int n, double val = 0.0)
                    : ArrayDbE(n,val), low_bnd(0) {}
        LimitArE(unsigned int n, int lb, double val = 0.0)
                    : ArrayDbE(n, val), low_bnd(lb) {}
        LimitArE(const double * pn, unsigned int n)
                    : ArrayDbE(pn, n), low_bnd(0) {}
        LimitArE(const ArrayDbE & a) : ArrayDbE(a), low_bnd(0) {}
    //nové metody
        void new_lb(int lb) {low_bnd = lb;}  // nastavení dolní meze
        int lbound() {return low_bnd;}      // hodnota dolní meze
        int ubound() {return ArSize() + low_bnd - 1;} // horní mez
    // předefinované operátory
        double & operator[](int i);
        const double & operator[](int i) const;
    };
    #endif
```

Předpokládejme program s objekty obou tříd `ArrayDbE` a `LimitArE`. V tom případě budete potřebovat pokusný blok zachytávající obě možné výjimky `BadIndex` a `SonOfBad`. Můžete to udělat tak, že za pokusný blok přidáte dva za sebou následující záchytné bloky:

```
try {
    LimitArE income(Years, FirstYear);
    ArrayDbE busywork(Years);
    ...
} // konec try bloku
catch (LimitArE::SonOfBad & bi)  // první handler
{
    ...
}
catch (LimitArE::BadIndex & bi)  // druhý handler
{
    ...
}
```

Jestliže existuje více záchytných bloků, program zkouší porovnat vyvolanou výjimku s prvním záchytným blokem, potom s druhým, atd. Jakmile odpovídající blok najde, provede ho. To ovšem za předpokladu, že tento záchytný blok program neukončí nebo nevyvolá další výjimku. Po provedení všech za sebou následujících záchytných bloků program skočí na první příkaz za posledním blokem.

Tato konkrétní sekvence záchytných bloků se vyznačuje zajímavou vlastností – blok s referencí `BadIndex` může zachytit výjimku `BadIndex` i výjimku `SonOfBad`. Reference základní třídy totiž může odkazovat na objekt třídy odvozené. Avšak záchytný blok s referencí `SonOfBad` objekt `BadIndex` zachytit nemůže. Reference objektu odvozené třídy nemůže odkazovat na třídu základní bez explicitního přetypování. Z tohoto důvodu se nabízí umístit záchytný blok `SonOfBad` před blok `BadIndex`. Takto blok SonOfBad výjimku SonOfBad zachytí a výjimku `BadIndex` předá následujícímu bloku. Tento způsob řešení předvádí program ve výpisu 14.17.

Výpis 14.17. excptinh.cpp

```
// excptinh.cpp - použití tříd ArrayDbE a LimitArE
// zkompilovat společně s arraydbe.cpp, limarre.h
#include <iostream>
using namespace std;
#include "arraydbe.h"
#include "limarre.h"
const int Years = 4;
const int FirstYear = 1998;
int main()
{
    int year;
    double total = 0;
    try {
        LimitArE income(Years, FirstYear);
        ArrayDbE busywork(Years);
        cout << "Zadejte prijem za posledni " << Years
              << " roky:\n";
        for (year = FirstYear; year < FirstYear + Years; year++)
        {
            cout << "Rok " << year << ": $";
            cin >> income[year];
            busywork[year - FirstYear] = 0.2 * income[year];
        }
        cout.precision(2);
        cout.setf(ios_base::showpoint);
        cout.setf(ios_base::fixed,ios_base::floatfield);
        cout << "Recapitulace udaju:\n";
        for (year = FirstYear; year <= FirstYear + Years; year++)
        {
           cout << year << ": $" << income[year] << "\n";
           total += income[year];
        }
        cout << "hodnoty pri zaneprazdneni: " << busywork;
```

```
    } // konec pokusného bloku
    catch (LimitArE::SonOfBad & bi)  // první handler
    {
    cout << "LimitArE vyjimka: "
            << bi.badindex << " je neplatna hodnota indexu.\n";
    cout << "Index musi byt v rozsahu od " << bi.l_lim
            << " do " << bi.u_lim << ".\n";
    }
    catch (LimitArE::BadIndex & bi)  // druhý handler
    {
        cout << "ArrayDbE vyjimka: "
                << bi.badindex << "e neplatna hodnota indexu.\n";
    }
    cout << "Celkovy prijem za " << (year - FirstYear)
            << " roky je $" << total << ".\n";
    cout << "Nashledanou!\n";
    return 0;
}
```

Zde je ukázka běhu programu:

```
Zadejte prijem za posledni 4 roky:
Rok 1998: $35000
Rok 1999: $34000
Rok 2000: $33000
Rok 2001: $38000
Recapitulace udaju:
1998: $35000.00
1999: $34000.00
2000: $33000.00
2001: $38000.00
LimitArE vyjimka: 2002 je neplatna hodnota indexu.
Index musi byt v rozsahu od 1998 do 2001.
Celkovy prijem za 4 roky je $140000.00.
```

Výjimka `SonOfBad` ukončila provádění pokusného bloku a předala provádění druhému záchytnému bloku. Jakmile program tento blok zpracoval, skočil na první příkaz následující za zachytávajícími bloky.

Následující tip shrnuje hlavní poučení z tohoto příkladu.

Tip

Jestliže máte hierarchii tříd výjimek vzniklou z dědičnosti, uspořádejte pořadí záchytných bloků takovým způsobem, aby poslední odvozená třída byla zachycena nejdříve a třída základní až nakonec.

Třída exception

Hlavním cílem výjimek v jazyku C++ je poskytnout jazykovou podporu pro tvorbu programů odolných proti chybám. To znamená, že je snazší ošetřit chyby zahrnutím výjimek do návrhu programu než přidávat dodatečně nějaké pevnější ošetření chyb. Pružnost a relativní vhodnost výjimek by měla programátory povzbudit, aby ošetření chyb zahrnuli do návrhu programu tam, kde je to vhodné. Stručně řečeno, výjimky jsou stejně jako třídy tím prostředkem, který může změnit váš přístup k programování.

Novější kompilátory jazyka C++ již výjimky do jazyka zahrnuly. Například hlavičkový soubor `exception` (dříve `exception.h` nebo `except.h`) definuje třídu `exception`, kterou C++ používá jako třídu základní pro podporu jazyka. V programu rovněž můžete vyvolat výjimku s objektem `exception` nebo použít třídu `exception` jako třídu základní. Jedna z virtuálních členských funkcí se nazývá `what()` a vrací řetězec, jehož vlastnost závisí na implementaci. Ale u odvozené třídy si můžete zvolit, jaký řetězec má vracet. Deklaraci `BadIndex` byste například mohli nahradit jinou, založenou na třídě `exception`. Protože obsah hlavičkového souboru `exception` je součástí oboru názvů `std`, můžete ho zpřístupnit pomocí direktivy `using`:

```
#include <exception>
using namespace std;
class ArrayDbE
{
private:
    unsigned int size; // počet prvků pole
protected:
    double * arr;     // adresa prvního prvku
public:
    class OutOfBounds : public exception
    {
    public:
        OutOfBounds() : exception() {};
        const char * what() {return "Prekroceny meze poli\n";}
    };
    ...
};
```

V tom případě byste mohli vyvolávat výjimku `OutOfBounds` pomocí metod přetíženého operátoru [], v programu mít třídu, která tento typ výjimky zachytí a pomocí metody `what()` problém identifikovat:

```
catch (ArrayDbE::OutOfBounds & ob)  // handler
{
    cout << "ArrayDbE vyjimka: "
         << ob.what() << endl;
}
```

Mimochodem, místo zpřístupnění všech deklarací tříd výjimek pomocí direktivy using byste mohli použít operátor rozlišení:

```
class OutOfBounds : public std::exception
```

Výjimka bad_alloc a operátor new

Pro ošetření problémů vzniklých při přidělování paměti operátorem new existují v jazyce C++ dvě možné implementace. První, a kdysi jediná, možnost spočívá ve vrácení nulového ukazatele, nelze-li požadavek na přidělení paměti uspokojit. Druhou možností je vyvolat pomocí operátoru new výjimku bad_alloc. Hlavičkový soubor new (dříve new.h) obsahuje deklaraci třídy bad_alloc, která je veřejně odvozena od třídy exception. Může se stát, že při implementaci budete moci použít pouze jednu z možností, nebo si budete moci díky nějakému přepínači kompilátoru nebo jiné metodě zvolit postup, kterému dáváte přednost.

Program ve výpisu 14.18 tuto otázku obchází a používá obě metody. Pokud je výjimka zachycena, zobrazí program pomocí zděděné metody what() zprávu, jejíž obsah závisí na implementaci, a předčasně skončí. V opačném případě zjistí, zda vrácenou hodnotou byl nulový ukazatel. (Účelem je předvést zde dva způsoby kontroly chyb při přidělování paměti a ne vyvolat dojem, že by se v normálním programu měly používat obě metody.)

Výpis 14.18. newexcp.cpp

```
// newexcp.cpp — výjimka bad_alloc
#include <iostream>
using namespace std;
#include <new>
#include <cstdlib>

struct Big
{
    double stuff[2000];
};
int main()
{
    Big * pb;
    try {
        cout << "Pokus pridelit velky blok pameti:\n";
        pb = new Big[10000];
        cout << "Novy pozadavek registrovan:\n";
    }
    catch (bad_alloc & ba)
    {
        cout << "Vyjimka zachycena!\n";
        cout << ba.what() << endl;
        exit(1);
    }
    if (pb != 0)
    {
        pb[0].stuff[0] = 4;
        cout << pb[0].stuff[0] << endl;
    }
    else
        cout << "pb je nulovy ukazatel\n";
```

```
    delete [] pb;
    return 0;
}
```

Když výjimky bloudí

Jakmile je výjimka vyvolána, má dvě příležitosti k tomu, aby způsobila problémy. Poprvé, je-li vyvolána ve funkci obsahující specifikaci výjimky, musí odpovídat jednomu z typů na seznamu specifikací. Pokud tomu tak není, je označena jako *neočekávaná výjimka* a implicitně způsobí ukončení programu. Jestliže překoná tuto první překážku (nebo se jí vyhne, protože funkce specifikaci výjimky postrádá), musí být zachycena. Pokud není, což se může stát v případě, jestliže neobsahuje pokusný blok ani odpovídající blok zachytávající, pak je označena jako *nezachycená výjimka* a to standardně způsobí ukončení programu. Reakci programu na výskyt neočekávaných a nezpracovaných výjimek však můžete změnit. Podíváme se jak a začneme nezachycenými výjimkami.

Nezachycená výjimka nezpůsobí okamžité ukončení programu. Namísto toho program nejdříve zavolá funkci `terminate()`. Standardně funkce `terminate()` volá funkci `abort()`. Chování funkce `terminate()` můžete upravit tak, že zaregistrujete funkci, která bude vyvolána místo funkce `abort()`. K tomu je potřeba zavolat funkci `set_terminate()`. Obě funkce `set_terminate()` a `terminate()` jsou deklarovány v hlavičkovém souboru `exception`:

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();
void terminate();
```

Funkce `BadIndex`má jako parametr název (adresu) funkce, která neobsahuje žádné parametry a má návratový typ `void`. Funkce vrací adresu předchozí registrované funkce. Zavoláte-li funkci `set_terminate()` vícekrát, zavolá funkce `terminate()` funkci nastavenou posledním voláním funkce `terminate()`.

Podívejme se na příklad. Předpokládejme, že chcete, aby program při nezachycené výjimce vytiskl příslušnou zprávu a potom zavolal funkci `exit()` s návratovým kódem `5`. Nejdříve vložte hlavičkový soubor `exception` a zpřístupněte deklarace v něm obsažené pomocí direktivy using:

```
#include <exception>
using namespace std;
```

Dále vytvořte funkci, která vykoná obě požadované akce a má náležitý prototyp:

```
void myQuit()
{
    cout << "Ukonceni z duvodu nezachycene vyjimky\n";
    exit(5);
}
```

Nakonec tuto funkci umístěte na začátek programu a pověřte ji provedením závěrečné činnosti:

```
set_terminate(myQuit);
```

Dále se podíváme na neočekávané výjimky. Použijete-li ve funkci specifikace výjimek, poskytnete uživateli funkce prostředek k určení výjimek, které se mají zachytávat. Předpokládejme následující prototyp:

```
double Argh(double, double) throw(exception &);
```

Tuto funkci byste mohli použít následujícím způsobem:

```
try {
    x = Argh(a, b);
}
catch(exception & ex)
{
    ...
}
```

Je dobré vědět, které výjimky zachytávat; vzpomeňte si, že implicitně nezachycená výjimka ukončí program.

To ale není všechno. V principu by specifikace výjimky měla obsahovat výjimky vyvolané funkcemi, které jsou volány dotyčnou funkcí. Jestliže například funkce `Argh()` volá funkci `Duh()`, která může vyvolat výjimku s objektem `retort`, pak by se výjimka `retort` měla objevit ve specifikaci výjimky funkce `Argh()` i ve specifikaci výjimky funkce `Duh()`. Jestliže si nedáte záležet a nenapíšete všechny funkce sami, není správné fungování zaručeno. Mohlo by se například stát, že použijete starší komerční knihovnu, jejíž funkce specifikace výjimek neobsahují. Z tohoto důvodu se nabízí podívat se blíže na to, co se stane, jestliže funkce vyvolá výjimku, která není specifikována.

Chování je velmi podobné tomu při nezachycené výjimce. Jestliže se vyskytne neočekávaná výjimka, program zavolá funkci `unexpected()`. (Funkci `unexpected()`jste neočekávali? Nikdo ji neočekává!) Tato funkce zase volá funkci `terminate()`, která standardně volá funkci `abort()`. Stejně jako pro úpravu chování funkce `terminate()` existuje funkce `set_terminate()`, existuje funkce `set_unexpected()`, která upravuje chování funkce `unexpected()`. Tyto dvě funkce jsou také deklarovány v hlavičkovém souboru výjimek:

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
void unexpected();
```

Avšak chování funkce, kterou dodáte funkci `set_unexpected()`, je více regulováno než chování funkce upravené funkcí `terminate()`. Konkrétně funkce `unexpected_handler` má tyto možnosti:

- Může ukončit program zavoláním funkcí `terminate()` (standardní chování), `abort()` nebo `exit()`.
- Může vyvolat výjimku.

Výsledek vyvolání výjimky (druhá možnost) závisí na výjimce vyvolané funkcí `exception_handler` a na její specifikaci:

- Jestliže nově vyvolaná výjimka odpovídá specifikaci výjimky, pak program od tohoto místa normálně pokračuje; to znamená, že bude hledat záchytný blok odpovídající této nově vyvolané výjimce.

- Pokud nově vyvolaná výjimka neodpovídá specifikaci výjimky a jestliže specifikace výjimky neobsahuje typ `std::bad_exception`, zavolá program funkci `terminate()`. Typ `bad_exception` je odvozen od typu `exception` a je deklarován v hlavičkovém souboru `exception`.
- Pokud nově vyvolaná výjimka neodpovídá specifikaci výjimky, ale specifikace výjimky typ `std::bad_exception` obsahuje, bude neodpovídající výjimka nahrazena výjimkou typu `std::bad_exception`.

Stručně řečeno, chcete-li zachytit všechny výjimky, očekávané nebo jiné, můžete postupovat následujícím způsobem:

Nejdříve se ujistěte, že jsou k dispozici deklarace z hlavičkového souboru exception:

```
#include <exception>
using namespace std;
```

Dále sestrojte pomocnou funkci, která převede neočekávané výjimky na výjimku typu bad_exception a která má náležitý prototyp:

```
void myUnexpected()
{
    throw std::bad_exception(); // nebo pouze throw;
}
```

Použijete-li pouze klíčové slovo `throw` bez určení výjimky, vyvolá se opět původní výjimka. Ta však bude nahrazena objektem `bad_exception`, pokud specifikace výjimek bude tento typ obsahovat.

Na začátku programu tuto funkci ustanovte provedením závěrečné činnosti při neočekávané výjimce:

```
set_unexpected(myUnexpeced);
```

Nakonec vložte typ `bad_exception` do specifikací výjimek a do záchytných bloků:

```
double Argh(double, double) throw(exception &, bad_exception &);
...
try {
  x = Argh(a, b);
}
catch(exception & ex)
{
    ...
}
catch(bad_exception & ex)
{
    ...
}
```

Opatření při používání výjimek

Z předchozí diskuse o používání výjimek byste mohli usoudit (a správně), že ošetření výjimek by mělo být zabudováno do programu a ne přidáno dodatečně. Používání výjimek

má také nevýhody, například roste velikost programu a snižuje se jeho rychlost. Specifikace výjimek nepracují dobře se šablonami, protože šablony funkcí mohou vyvolávat různé druhy výjimek v závislosti na konkrétně použité specializaci. Výjimky také ne vždy dobře spolupracují s dynamickým přidělováním paměti.

Podívejme se trochu blíže na dynamické přidělování paměti a výjimky. Nejdříve uvažujte následující funkci:

```
void test1(int n)
{
    String mesg("Chytil jsem se do nekonecne smycky");
    ...
    if (oh_no)
        throw exception();
    ...
    return;
}
```

Vzpomeňte si, že třída String používala dynamické přidělování paměti. Normálně by byl pro objekt mesg zavolán destruktor třídy, jakmile by funkce narazila na příkaz return a skončila. Díky uvolnění zásobníku umožňuje příkaz throw, přestože funkci předčasně ukončí, aby stále bylo možné destruktor zavolat. Takže zde je paměť spravována patřičně.

Nyní uvažujte tuto funkci:

```
void test2(int n)
{
    double * ar = new double[n];
    ...
    if (oh_no)
        throw exception();
    ...
    delete [] ar;
    return;
}
```

Zde je problém. Při uvolnění zásobníku je proměnná ar odebrána. Ale předčasné ukončení funkce znamená, že příkaz delete [] na konci funkce se přeskočí. Ukazatel je ztracen, ale blok paměti, na nějž ukazoval, zůstal nedotčen a je nepřístupný. Stručně řečeno, jedná se o únik paměti.

Úniku paměti se lze vyhnout. Můžete například zachytit výjimku ve stejné funkci, která ji vyvolává, vložit do záchytného bloku nějaké příkazy pro čištění a potom tuto výjimku znovu vyvolat:

```
void test3(int n)
{
    double * ar = new double[n];
    ...
    try {
        if (oh_no)
            throw exception();
```

```
    }
    catch(exception & ex)
    {
        delete [] ar;
        throw;
    }
    ...
    delete [] ar;
    return;
}
```

Tento způsob však zvyšuje možnost přehlédnutí a dalších chyb. Dalším řešením je použití šablony `auto_ptr`, probírané v kapitole 15.

Stručně řečeno, ošetření výjimek je pro některé projekty nesmírně důležité, ale vyžaduje programátorské úsilí, velikost programu a jeho rychlost. Rovněž podpora výjimek v kompilátorech a zkušenost uživatelů ještě nedosáhly zralé úrovně. Měli byste tedy tento prostředek používat s mírou.

RTTI

RTTI je zkratkou názvu runtime type information (informace o dynamických typech). Jedná se o jeden z posledních doplňků jazyka C++ a mnohé starší překladače ho nepodporují. Jiné implementace mohou mít přepínače pro zapnutí a vypnutí RTTI. Účelem RTTI je, aby v programu existoval standardní způsob, jak zjistit typ objektu za běhu. Mnoho knihoven tříd již nabízí podobné způsoby pro objekty svých tříd, ale vzhledem k absenci vestavěné podpory v jazyku C++ jsou mechanismy jednotlivých prodejců většinou vzájemně nekompatibilní.

K čemu je RTTI dobré

Předpokládejme hierarchii tříd se společnou základní třídou. Ukazatel na základní třídu můžete nastavit na objekt jakékoli třídy této hierarchie. Dále vyvoláte funkci, která po zpracování nějakých informací vybere jednu z těchto tříd, vytvoří objekt tohoto typu a vrátí jeho adresu, která se přiřadí ukazateli na základní třídu. Jak můžete zjistit druh objektu, na který ukazuje?

Než na tuto otázku odpovíme, zamysleme se, k čemu byste potřebovali typ znát. Možná potřebujete vyvolat správnou verzi nějaké metody. V takovém případě nemusíte typ objektu znát, pokud se jedná o funkci virtuální, kterou vlastní všechny třídy v hierarchii. Mohlo by se ale stát, že odvozený objekt obsahuje nezděděnou metodu. V takovém případě by tuto metodu mohly použít jen některé objekty. Nebo možná budete chtít za účelem ladění registrovat typy vytvářených objektů. Na tyto poslední dva případy dává odpověď RTTI.

Jak RTTI pracuje

Na podporu RTTI má jazyk C++ tři složky:

- Operátor `dynamic_cast` generuje ukazatel na typ odvozený od ukazatele na základní typ, pokud to je možné. Jinak vrátí hodnotu 0, tedy nulový ukazatel.

- Operátor `typeid` vrací hodnotu identifikující přesný typ objektu.
- Struktura `type_info` obsahuje informace o konkrétním typu.

RTTI můžete použít jenom v rámci hierarchie tříd obsahujících virtuální funkce. Důvodem je, že pouze u takovýchto hierarchií byste měli přiřazovat adresu objektu odvozené třídy ukazateli na objekt bázové třídy.

Upozornění

RTTI funguje pouze u tříd s virtuálními funkcemi.

Pojďme tyto tři složky prozkoumat.

Operátor dynamic_cast

Operátor `dynamic_cast` by měl být asi nejpoužívanější částí RTTI. Neodpovídá na otázku, na jaký typ objektu ukazatel ukazuje, ale zda můžete adresu objektu bezpečně přiřadit ukazateli na určitý typ. Podívejme se, co to znamená. Předpokládejme následující hierarchii tříd:

```
class Grand { // obsahuje virtuální metody};
class Superb : public Grand { … };
class Magnificent : public Superb { … };
```

Dále předpokládejme následující ukazatele:

```
Grand * pg = new Grand;
Grand * ps = new Superb;
Grand * pm = new Magnificent;
```

A nakonec uvažujte následující přetypování:

```
Magnificent * p1 = (Magnificent *) pm;     // #1
Magnificent * p2 = (Magnificent *) pg;     // #2
Superb * p3 = (Magnificent *) pm;          // #3
```

Která z výše uvedených přetypování jsou bezpečná? V závislosti na deklaraci tříd by mohla být bezpečná všechna, ale bezpečnost je zaručena pouze u těch, jejichž ukazatel je stejného typu jako objekt nebo je přímým nebo nepřímým základním typem objektu. Například přetypování č. 1 je bezpečné, protože nastavuje ukazatel typu `Magnificent` na objekt typu `Magnificent`. Přetypování č. 2 bezpečné není, protože přiřazuje adresu objektu základní třídy (`Grand`) ukazateli na třídu odvozenou (`Magnificent`). Program by v takovém případě očekával, že objekt základní třídy zdědil vlastnosti třídy odvozené, což obecně neplatí. Objekt třídy `Magnificent` například může mít datové položky, které objektu třídy `Grand` chybí. Přetypování č. 3 ovšem bezpečné je, protože přiřazuje adresu odvozeného objektu ukazateli na objekt základní třídy. Veřejné odvození tedy zaručuje, že objekt třídy `Magnificent` je také objektem třídy `Superb` (přímé odvození přímý předchůdce) a objektem třídy `Grand` (nepřímé odvození nepřímý předchůdce). Jeho adresu tedy můžete bezpečně přiřadit všem třem typům ukazatelů. Virtuální funkce zaručují, že použití kteréhokoli z těchto tří typů ukazatelů vyvolá metody třídy `Magnificent`.

Všimněte si, že otázka bezpečnosti typové konverze je obecnější a užitečnější než otázka typu objekt, na který ukazatel ukazuje. Obvykle tedy chcete typ znát z toho důvodu, abyste věděli, je-li vyvolání určité metody bezpečné. K vyvolání metody nepotřebujete nutně zcela přesný typ. Může jím být typ základní třídy, pro který je definována virtuální verze metody. Tuto myšlenku bude ilustrovat následující příklad.

Nejdříve se ale podívejme na syntaxi operátoru `dynamic_cast`. Proměnná `pg` je zde ukazatel na objekt:

```
Superb pm = dynamic_cast<Superb *>(pg);
```

Nabízí se otázka, zda lze ukazatel pg bezpečně přetypovat na ukazatel na typ `Superb`. Jestliže ano, vrátí operátor adresu tohoto objektu.

Pamatujte

Obecně výraz

```
dynamic_cast<Type *>(pt)
```

konvertuje ukazatel `pt` na ukazatel na typ `Type`, pokud je odkazovaný objekt (`*pt`) typu `Type` nebo je přímo či nepřímo odvozen od typu `Type`. Jinak se výraz vyhodnotí jako 0, tedy nulový ukazatel.

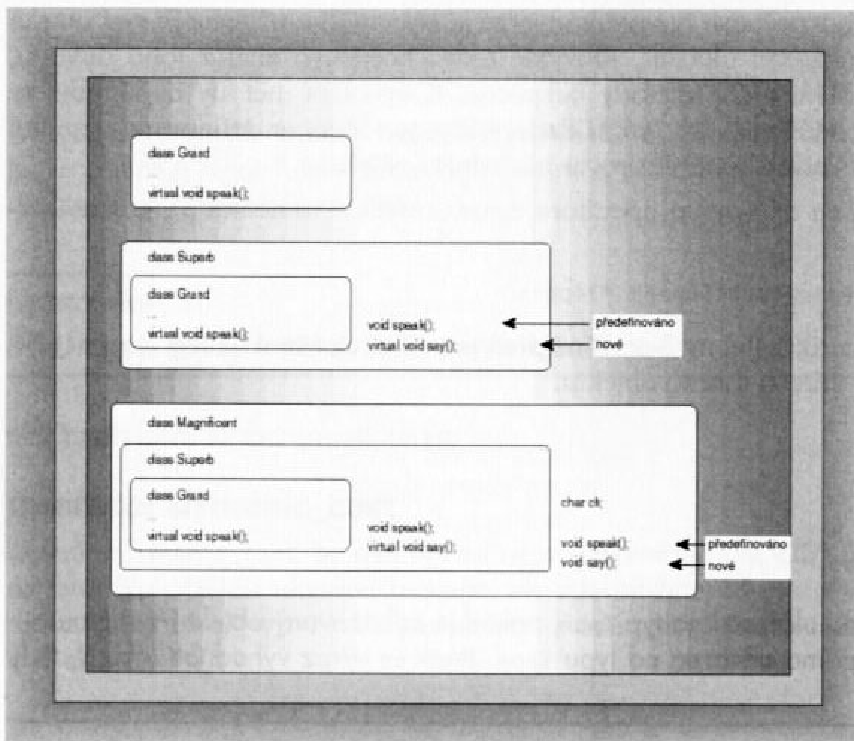
Proces ilustruje výpis 14.19. Nejdříve jsou definovány tři třídy pojmenované `Grand`, `Superb` a `Magnificent`. Třída `Grand` definuje virtuální funkci `Speak()`, která je v ostatních třídách předefinována. Třída `Superb` definuje virtuální funkci `Say()`, která je předefinována ve třídě `Magnificent` (viz obrázek 14.4). Program definuje funkci `GetOne()`, která náhodně vytvoří a inicializuje objekt jednoho z těchto tří typů a potom vrátí adresu jako ukazatel typu `Grand`. (Funkce `GetOne()` simuluje interaktivní práci uživatele.) V cyklu je tento ukazatel přiřazen proměnné `pg` typu `Grand *` a potom se proměnná `pg` použije k vyvolání funkce `Speak()`. Protože je tato funkce virtuální, vyvolá program správnou verzi odpovídající odkazovanému objektu:

```
for (int i = 0; i < 5; i++)
{
    pg = GetOne();
    pg->Speak();
    ...
}
```

Funkci `Say()` však úplně stejným způsobem vyvolat nemůžete; není pro třídu `Grand` definována. Můžete však použít operátor `dynamic_cast` a zjistit, zda je možné převést `pg` na ukazatel typu `Superb`. Bude to možné, pokud objekt bude typu `Superb` nebo `Magnificent`. V obou případech budete moci funkci `Say()` bezpečně vyvolat:

```
if (ps = dynamic_cast<Superb *>(pg))
    ps->Say();
```

Vzpomeňte si, že hodnotou přiřazovacího příkazu je hodnota jeho levé strany. Takže hodnotou podmínky `if` je hodnota ukazatele `ps`. Bude-li přetypování úspěšné, pak `ps` bude mít nenulovou hodnotu, bude tedy pravdivý. Jestliže se přetypování nepodaří, `ps` bude mít nulovou hodnotu, tedy bude nepravdivý. Úplný kód je ve výpisu 14.19.



Obrázek 14.4. Skupina tříd Grand

Výpis 14.19. rtti1.cpp

```
// rtti1.cpp — použiti operátoru dynamic_cast
#include <iostream>
using namespace std;
#include <cstdlib>
#include <ctime>

class Grand
{
private:
    int hold;
public:
    Grand(int h = 0) : hold(h) {}
    virtual void Speak() const { cout << "Jsem velkolepa trida!\n";}
    virtual int Value() const { return hold; }
};

class Superb : public Grand
{
public:
    Superb(int h = 0) : Grand(h) {}
    void Speak() const {cout << "Jsem super trida!!\n"; }
    virtual void Say() const
        { cout << "Obsahuji super hodnotu " << Value()
```

```
        << "!\n";}
};

class Magnificent : public Superb
{
private:
    char ch;
public:
    Magnificent(int h = 0, char c = 'A') : Superb(h), ch(c) {}
    void Speak() const {cout << "Jsem skvela trida!!!\n";}
    void Say() const {cout << "Obsahuji znak " << ch <<
               " a cele cislo " << Value() << "!\n"; }
};

Grand * GetOne();
int main()
{
    srand(time(0));
    Grand * pg;
    Superb * ps;
    for (int i = 0; i < 5; i++)
    {
        pg = GetOne();
        pg->Speak();
        if( ps = dynamic_cast<Superb *>(pg))
            ps->Say();
    }
    return 0;
}
Grand * GetOne()  // vytvoří náhodně jeden ze tří druhů objektů
{
    Grand * p;
    switch( rand() % 3)
    {
        case 0: p = new Grand(rand() % 100);
                break;
        case 1: p = new Superb(rand() % 100);
                break;
        case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26);
                break;
    }
    return p;
}
```

Kompatibilita:

I když váš kompilátor RTTI podporuje, může být tento prostředek standardně vypnut. Měli byste proto prostudovat dokumentaci nebo prozkoumat volby nabídky.

Tento program ilustruje důležitou myšlenku. Virtuální funkce byste měli používat při každé příležitosti a RTTI pouze v případě nutnosti. Zde je ukázka výstupu:

```
Jsem super trida!!
Obsahuji super hodnotu 68!
Jsem skvela trida!!!
Obsahuji znak R a cele cislo 68!
Jsem skvela trida!!!
Obsahuji znak D a cele cislo 12!
Jsem skvela trida!!!
Obsahuji znak V a cele cislo 59!
Jsem velkolepa trida!
```

Jak vidíte, byly metody `Say()` vyvolány pouze pro objekty tříd `Superb` a `Magnificent`.

Operátor `dynamic_cast` můžete použít také u referencí. Použití je trochu odlišné; reference nemá žádnou hodnotu odpovídající nulovému ukazateli, takže pro stanovení chyby žádnou speciální hodnotu reference použít nelze. Pokud však operátor `dynamic_cast` nemůže přetypování provést, vyvolá výjimku `bad_cast`, která je odvozena od třídy `exception` a je definována v hlavičkovém souboru `typeinfo`. Operátor lze tedy použít následujícím způsobem, přičemž `rg` je referencí na objekt třídy `Grand`:

```
#include <typeinfo> // pro bad_cast
...
try {
    Superb & rs = dynamic_cast<Superb &>(rg);
  ...
}
catch(bad_cast &){
    ...
};
```

Operátor typeid a třída type_info

Operátor `typeid` umožňuje určit, zda jsou dva objekty stejného typu. Podobně jako operátor `sizeof` má dva typy parametrů:

- Název třídy
- Výraz, který se vyhodnotí na objekt

Operátor `typeid` vrací odkaz na objekt třídy `type_info`, kde `type_info` je třída deklarovaná v hlavičkovém souboru `typeinfo` (dříve `typeinfo.h`). Třída `type_info` přetěžuje operátory `==` a `!=` tak, abyste s nimi mohli porovnávat typy. Například výraz

```
typeid(Magnificent) == typeid(*pg)
```

se vyhodnotí na hodnotu `true`, jestliže `pg` odkazuje na objekt třídy `Magnificent`, jinak se vyhodnotí na hodnotu `false`. Pokud bude ukazatel `pg` nulový, vyvolá program výjimku `bad_typeid`. Tento typ výjimky je odvozen od třídy `exception` a je deklarován v hlavičkovém souboru `typeinfo`.

Implementace třídy `type_info` se mezi různými prodejci liší, ale bude obsahovat metodu `name()` vracející implementačně závislý řetězec, kterým bývá název třídy. Například příkaz

```
cout << "Nyni se zpracovava typ " << typeid(*pg).name() << ".\n";
```

zobrazí řetězec definovaný pro třídu objektu, na kterou ukazatel `pg` ukazuje.

Výpis 14.20 upravuje výpis 14.19 tak, že používá operátor `typeid` a členskou funkci `name()`. Všimněte si, že jsou používány v situacích, které neošetřují operátor `dynamic_cast` a virtuální funkce. Pomocí operátoru typeid se vybírá činnost, kterou není ani metoda třídy a nelze ji tedy vyvolat pomocí ukazatele na třídu. Příkaz vyvolávající metodu `name()` ukazuje, jak lze tuto metodu použít při ladění. Všimněte si, že program obsahuje hlavičkový soubor `typeinfo`.

Výpis 14.20. rtti2.cpp

```
// rtti2.cpp - použití operátorů dynamic_cast, typeid, and type_info
#include <iostream>
using namespace std;
#include <cstdlib>
#include <ctime>
#include <typeinfo>

class Grand
{
private:
    int hold;
public:
    Grand(int h = 0) : hold(h) {}
    virtual void Speak() const { cout << "Jsem velkolepa trida!\n";}
    virtual int Value() const { return hold; }
};

class Superb : public Grand
{
public:
    Superb(int h = 0) : Grand(h) {}
    void Speak() const {cout << "Jsem super trida!!\n"; }
    virtual void Say() const
    { cout << "Obsahuji hodnotu " << Value() << "!\n";}
};

class Magnificent : public Superb
{
private:
    char ch;
public:
    Magnificent(int h = 0, char c = 'A') : Superb(h), ch(c) {}
    void Speak() const {cout << "Jsem skvela trida!!!\n";}
    void Say() const {cout << "Obsahuji znak " << ch <<
                " a cele cislo " << Value() << "!\n"; }
};

Grand * GetOne();
```

```
int main()
{
    srand(time(0));
    Grand * pg;
    Superb * ps;
    for (int i = 0; i < 5; i++)
    {
        pg = GetOne();
        cout << "Nyni se zpracovava typ " << typeid(*pg).name()
          << ".\n";
        pg->Speak();
       if( ps = dynamic_cast<Superb *>(pg))
           ps->Say();
       if (typeid(Magnificent) == typeid(*pg))
           cout << "Ano, opravdu jsi skvela.\n";
}
    return 0;
}
Grand * GetOne()
{
    Grand * p;

    switch( rand() % 3)
    {
        case 0: p = new Grand(rand() % 100);
                    break;
        case 1: p = new Superb(rand() % 100);
                    break;
        case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26);
                    break;
    }
    return p;
}
```

Zde je ukázka běhu:

```
Nyni se zpracovava typ Magnificent.
Jsem skvela trida!!!
Obsahuji znak P a cele cislo 52!
Ano, jsi opravdu skvela.
Nyni se zpracovava typ Superb.
Jsem super trida!!
Obsahuji super hodnotu 37!
Nyni se zpracovava typ Grand.
Jsem velkolepa trida class!
Nyni se zpracovava typ Superb.
Jsem super trida!!
Obsahuji super hodnotu 18!
Nyni se zpracovava typ Grand.
Jsem velkolepa trida!
```

Špatné použití RTTI

V komunitě uživatelů C++ má RTTI mnoho hlasitých kritiků. Považují RTTI za zbytečné, potencionální zdroj neefektivity programu a myslí si, že přispívá ke špatným programátorským návykům. Aniž bychom se pouštěli do debaty o RTTI, podívejme se na druh programování, kterému byste se měli vyhnout.

Uvažujte základ výpisu 14.19:

```
Grand * pg;
Superb * ps;
for (int i = 0; i < 5; i++)
{
  pg = GetOne();
  pg->Speak();
  if( ps = dynamic_cast<Superb *>(pg))
    ps->Say();
}
```

S použitím operátoru `typeid` a vynecháním operátoru `dynamic_cast` a virtuálních funkcí můžete tento kód přepsat takto:

```
Grand * pg;
Superb * ps;
Magnificent * pm;
for (int i = 0; i < 5; i++)
{
    pg = GetOne();
    if (typeid(Magnificent) == typeid(*pg))
    {
        pm = (Magnificent *) pg;
        pm->Speak();
        pm->Say();
    }
    else if (typeid(Superb) == typeid(*pg))
    {
        ps = (Superb *) pg;
        ps->Speak();
        ps->Say();
    }
    else
        pg->Speak();
}
```

Nejenže je tento kód škaredší a delší než kód původní, ale explicitní pojmenování každé třídy představuje vážnou chybu. Předpokládejme například, že budete muset od třídy `Magnificent` odvodit třídu `Insufferable`. Nová třída předefinuje metody `Speak()` a `Say()`. U verze testující každý typ explicitně pomocí operátoru `typeid` byste museli upravit kód smyčky `for` přidáním nové větve příkazu `else if`. V původní verzi však nejsou žádné změny potřeba. Příkaz

```
pg->Speak();
```

funguje pro všechny třídy odvozené od třídy Grand a příkaz

```
if( ps = dynamic_cast<Superb *>(pg))
    ps->Say();
```

funguje pro všechny třídy odvozené od třídy Superb.

Tip

Jestliže se přistihnete, že používáte operátor `typeid` v dlouhé sérii příkazů `if else`, zkontrolujte, zda jste neměli použít virtuální funkce a operátor `dynamic_cast`.

Operátory přetypování

Podle názoru Bjarne Stroustrupa je operátor přetypování jazyka C příliš nedbalý. Uvažujte například následující kód:

```
struct Data
{
    double data[200];
};
struct Junk
{
    int junk[100];
};
Data d = {2.5e33, 3.5e-19, 20.2e32};
char * pch = (char *) (&d);// přetypování č. 1 - konverze na řetězec
char ch = char (&d);      // přetypování č. 2 - konverze adresy na znak
Junk * pj = (Junk *) (&d); // přetypování č. 3 - ukazatel na ukazatel
                           // na typ Junk
```

Za prvé, které z těchto tří přetypování má nějaký smysl? Pokud se neuchýlíme k nepravděpodobným scénářům, pak nemá smysl ani jedno z nich. Za druhé, která z těchto tří přetypování jsou povolena? Všechna.

Stroustrupovou odpovědí na tuto nedbalost bylo přidání čtyř operátorů přetypování, které do procesu přetypování zavádějí větší řád:

```
dynamic_cast
const_cast
static_cast
reinterpret_cast
```

Místo obecného přetypování můžete vybrat operátor vhodný pro konkrétní účel. Tím dokládáte zamýšlený důvod přetypování a kompilátoru dáváte příležitost zkontrolovat, že jste udělali to, co jste chtěli.

Operátor `dynamic_cast` jste již viděli. Závěrem uvažujme třídy `Low` a `High`, přičemž `ph` je typu `High *` a `pl` je typu `Low*`. Potom příkaz

```
pl = dynamic_cast<Low *> ph;
```

přiřadí ukazatel `Low*` proměnné pl pouze tehdy, jestliže `Low` je základní třída (přímá či nepřímá) přístupná třídě `High`. Jinak příkaz přiřadí ukazateli `pl` nulovou hodnotu. Obecně má operátor tuto syntaxi:

```
dynamic_cast < název-typu > (výraz)
```

Účelem tohoto operátoru je umožnit v hierarchii tříd přetypování na předka (takové přetypování je díky vztahu *je* bezpečné) a znemožnit ostatní přetypování.

Operátor `const_cast` umožňuje přidávat a odebírat modifikátory `const` anebo `volatile`. Syntaxe je stejná jako u operátoru `dynamic_cast`:

```
const_cast < název-typu > (výraz)
```

Výsledek takového přetypování je chybný, pokud se změní některé jiné aspekty typu. To znamená, že *název_typu* a výraz musí být stejného typu a lišit se mohou pouze přítomností nebo absencí modifikátorů `const` a `volatile`. Opět předpokládáme třídy `Low` a `High`:

```
const High bar;
...
High * pb = const_cast<High *> (&bar);   // platné
Low * pl = const_cast<Low *> (& bar);    // neplatné
```

První příkaz vytvoří ukazatel `*pb`, s jehož pomocí lze změnit objekt `bar`; odstraní se modifikátor `const`. Druhé přetypování je neplatné, protože se také snaží změnit typ z `High *` na typ `Low *`.

Důvodem existence tohoto operátoru je skutečnost, že příležitostně potřebujete změnit hodnotu, která je po většinu doby konstantou. V takovém případě hodnotu deklarujete jako konstantu a potřebujete-li její hodnotu změnit, použijete operátor `const_cast`. To by šlo udělat i obecným přetypováním, ale při něm může dojít také ke změně typu:

```
const High bar;
...
High * pb = (const High *) (&bar);
// platné
Low * pl = (const Low *) (&bar);
// také platné
```

Protože současná změna typu i konstantnosti může být neúmyslnou programátorskou chybou, je použití operátoru `const_cast` bezpečnější.

Operátor `static_cast` má stejnou syntaxi jako ostatní operátory:

```
static_cast < název-typu > (výraz)
```

Příkaz je platný pouze tehdy, jestliže lze *název-typu* implicitně převést na stejný typ jako je *výraz* a obráceně. V jiných případech je přetypování chybné. Předpokládejme, že `High` je základní třídou třídy `Low` a že třída `Pond` s nimi není příbuzná. Konverze z `High` na Low a obráceně jsou platné, ale konverze z `Low` na Pond již povolena není:

```
const High bar;
const Low blow;
...
High * pb = static_cast<High *> (&blow);  // platné přetypování na předka
Low * pl = static_cast<Low *> (&bar);     // platné přetypování na potomka
Pond * pmer = static_cast<Pond *> (&blow); // neplatné
```

První konverze je platná, protože přetypování na předka lze provést explicitně. Druhou konverzi, z ukazatele na základní třídu na ukazatel na třídu odvozenou, nelze provést bez explicitního přetypování. Ale protože přetypování v opačném směru bez explicitního přetypování možné je, lze pro přetypování na potomka použít operátor `static_cast`.

Podobně protože výčtový typ může být převeden na typ celočíselný bez přetypování, může být celočíselný typ převeden na typ výčtový s použitím operátoru `static_cast`.

Operátor `reinterpret_cast` je určen pro riskantní přetypování. Nedovolí odstranit modifikátor `const`, ale jinak dovolí všechny příšernosti. Občas programátor musí dělat implementačně závislé příšernosti a operátor `reinterpre_cast` usnadňuje přehled o takovýchto činech. Má stejnou syntaxi jako ostatní tři operátory:

```
reinterpret_cast < název-typu > (výraz)
```

Zde je ukázka použití:

```
struct dat {short a; short b};
long value = 0xA224B118;
dat * pd = reinterpret_cast< dat *) (&value);
cout << pd->a;  // zobrazí první 2 bajty hodnoty
```

Takováto přetypování se používají při nízkoúrovňovém, implementačně závislém programování a nebývají přenositelná. Například výstup tohoto kódu bude jiný na počítačích kompatibilních s IBM a jiný v systému Macintosh, protože oba systémy ukládají bajty ve vícebajtových celočíselných typech v opačném pořadí.

Shrnutí

Díky přátelům je možné vytvořit pružnější rozhraní třídy. Jako přátele třída může mít jiné funkce, jiné třídy a členské funkce jiných tříd. V některých případech musíte použít předběžnou deklaraci a dát si pozor na pořadí deklarací tříd a metod tříd, aby do sebe přátelé správně zapadali.

Vnořené třídy jsou třídy deklarované v jiné třídě. Usnadňují návrh pomocných tříd, které implementují jiné třídy, ale nemusí být součástí veřejného rozhraní.

Mechanismus výjimek jazyka C++ představuje pružný způsob vypořádání se s ošemetnými programovacími případy, jako jsou nevhodné hodnoty, neúspěšné vstupně-výstupní operace a podobně. Vyvolání výjimky ukončí právě prováděnou funkci a předá řízení odpovídajícímu záchytnému bloku. Záchytné bloky následují ihned za blokem pokusným a aby výjimka mohla být zachycena, musí být volání funkce, které vede k přímému či nepřímému vyvolání výjimky, umístěno v pokusném bloku. Program poté provede kód v záchytném bloku. Tento kód se může pokusit chybu opravit nebo může program ukončit. Třída může být navržena s vnořenými třídami výjimek, které budou vyvolány při odhalení určitých problémů v dané třídě. Funkce může obsahovat specifikaci výjimek určující výjimky, které může funkce vyvolat. Nezachycené výjimky (výjimky bez záchytného bloku) vedou standardně k ukončení programu. Stejně končí program i v případě neočekávaných výjimek (výjimky neodpovídající specifikaci výjimek funkce).

Díky prostředkům RTTI může program rozpoznat typ objektu. Operátor `dynamic_cast` se používá pro přetypování ukazatele na objekt odvozené třídy na ukazatel na objekt zá-

kladní třídy; jeho hlavním účelem je zajistit, aby bylo volání virtuální funkce bezpečné. Operátor typeid vrací objekt typu type_info. Dvě hodnoty vrácené operátorem typeid lze porovnat a určit, zda je objekt určitého typu, a vrácený objekt typu type_info lze použít k získání informací o objektu.

Operátory dynamic_cast, static_cast, const_cast a reinterpret_cast nabízejí bezpečnější, lépe dokumentované způsoby přetypování než přetypování obecné.

Opakovací otázky

1. Co je špatně v následujících pokusech o vytvoření přátel?

a)
```
class snap {
    friend clasp;
    ...
};
class clasp { ... };
```

b)
```
class cuff {
public:
    void snip(muff &) { ... }
    ...
};
class muff {
    friend void cuff::snip(muff &);
    ...
};
```

c)
```
class muff {
    friend void cuff::snip(muff &);
    ...
};
class cuff {
public:
    void snip(muff &) { ... }
    ...
};
```

2. Viděli jste, jak lze vytvořit vzájemné přátele. Dokážete vytvořit omezenější formu přátelství, kdy jen některé metody třídy B budou přáteli třídy A a opačně? Vysvětlete.

3. Jaké problémy mohou nastat v následující deklaraci vnořené třídy?

```
class Ribs
{
private:
    class Sauce
    {
        int soy;
        int sugar;
    public:
        Sauce(int s1, int s2) : soy(s1), sugar(s2) { }
```

```
        };
        ...
    };
```

4. Jak se liší příkaz throw od příkazu return?
5. Předpokládejte hierarchii tříd výjimek odvozených od základní třídy výjimek. V jakém pořadí by měly být uspořádány záchytné bloky?
6. Uvažujte třídy `Grand`, `Superb` a `Magnificent` definované v této kapitole. Předpokládejte, že pb je ukazatel typu `Grand *` a je mu přiřazena adresa objektu jedné z těchto tří tříd a `ps` je ukazatel typu `Superb *`. Čím se liší chování následujících dvou příkladů?

```
if (ps = dynamic_cast<Superb *>(pg))
        ps->say(); // příklad č. 1
if (typeid(*pg) == typeid(Superb))
        (Superb *) pg)->say(); // příklad č. 2
```

7. Jak se liší operátor dynamic_cast od operátoru static_cast?

Programovací cvičení

1. Upravte třídy `Tv` a `Remote` podle následujících pokynů:
   a) Vytvořte z nich vzájemné přátele.
   b) Přidejte do třídy `Remote` stavovou položku, která bude popisovat, zda se ovladač nachází v normálním nebo interaktivním režimu.
   c) Přidejte třídě `Remote` metodu zobrazující režim ovladače.
   d) Doplňte do třídy `Tv` metodu pro přepínání režimu dálkového ovladače. Tato metoda by měla fungovat pouze tehdy, když je televize zapnutá.

   Napište krátký program testující tyto nové vlastnosti.
2. Upravte program ve výpisu 14.10 tak, aby funkce `hmean()` vyvolávala výjimku typu `hmeanexcp` a funkce `gmean()` výjimku typu `gmeanexcp`. Oba typy výjimek by měly být odvozeny od třídy `exception`. Při zachycení výjimky `hmeanexcp` by si měl program vyžádat další dvojici čísel a pokračovat, zatímco při zachycení výjimky `gmeanexcp` by měl program pokračovat prvním příkazem následujícím za cyklem.

KAPITOLA

15

Třída STRING a standardní knihovna šablon

Témata kapitoly:

- Standardní třída jazyka C++ `string`
- Šablona `auto_ptr`
- Standardní knihovna šablon
- Třídy kontejnerů
- Iterátory
- Funkční objekty (funktory)
- Algoritmy standardní knihovny šablon

Nyní již víte, že cílem jazyka C++ je vytvořit znovupoužitelný kód. Největší užitek vám přinese možnost použít kód napsaný jinými. Zde přicházejí ke slovu knihovny tříd. K dispozici je mnoho komerčních knihoven tříd jazyka C++ a existují také knihovny, které jsou součástí sady programů C++. Používáte například vstupně-výstupní třídy podporované hlavičkovým souborem `ostream`. V této kapitole se podíváme na jiný znovupoužitelný kód, který je pro vaše programátorské potěšení dostupný. Nejdříve prozkoumáme třídu `string`, která zjednodušuje programování řetězců, potom se podíváme na „inteligentní" ukazatel šablonové třídy `auto_ptr` usnadňující trochu správu paměti a nakonec se podíváme na standardní knihovnu šablon (Standard Template Library, STL), kolekci užitečných šablon pro správu různých druhů kontejnerových objektů. Standardní knihovna šablon uvádí příklady programovaní z poslední doby, tzv. generického programování.

Třída string

V kapitole 11 jsme zavedli skromnou třídu `String`, ilustrující některé aspekty návrhu třídy. Samotný jazyk C++ nabízí výkonnější verzi této třídy zvanou `string`. Podporuje ji hlavičkový soubor `string`. (Pamatujte si, že hlavičkové soubory `string.h` a `cstring` podporují funkce pro práci

s řetězci z knihovny jazyka C, ale třídu string nepodporují.) Klíčem k používání knihovny je znalost jejího veřejného rozhraní. Tato knihovna má rozsáhlou sadu metod zahrnující několik konstruktorů, přetížených operátorů pro přiřazování, spojování a porovnávání řetězců, přístup k jednotlivým prvkům a rovněž funkce pro vyhledávání znaků a podřetězců v řetězci a mnoho dalšího. Stručně řečeno, třída string je obsažná.

Vytvoření řetězce

Nejdříve se podívejme na konstruktory třídy string. Konec konců jednou z nejdůležitějších věcí, které o třídě musíte vědět, jsou možnosti, které vám při vytváření objektů nabízí. Program ve výpisu 15.1 používá všech šest konstruktorů třídy string (jsou označeny ctor, což je tradiční zkratka C++ pro konstruktory). V tabulce 15.1 jsou konstruktory popsány podle pořadí, v jakém jsou v programu použity. Reprezentace konstruktorů je zjednodušena, neboť zakrývají skutečnost, že třída string je vytvořena pomocí klíčového slova typedef ze specializované šablony basic_string<char>, a jsou vynechány nepovinné parametry související se správou paměti. (O tomto aspektu bude v kapitole pojednáno později a rovněž v příloze F.) Typ size_type je implementačně závislý celočíselný typ definovaný v hlavičkovém souboru string. Ve třídě je definována proměnná string::npos pro uložení maximální délky řetězce. Většinou bývá rovna maximální hodnotě typu unsigned int. V tabulce je také použita obecná zkratka NBTS označující řetězec ukončený bajtem s nulovou hodnotou, tedy tradiční řetězec jazyka C ukončený nulovým znakem.

Tabulka 15.1 Konstruktory třídy string.

Konstruktor	Popis
`string(const char * s)`	Inicializuje objekt string pomocí NBTS, na který ukazuje s.
`string(size_type n, char c)`	Vytvoří objekt string s n prvky, každý prvek bude inicializován na hodnotu znaku c.
`string(const string * str,`	Inicializuje objekt string pomocí objektu size_type pos = 0, size_type n = npos) str s počátkem na pozici pos do konce řetězce nebo o velikosti n znaků.
`string()`	Vytvoří implicitní objekt string s nulovou velikostí.
`string(const char *, size_type n)`	Inicializuje objekt string pomocí NBTS, na který ukazuje, s a počtem n znaků. Počet znaků může být větší než délka řetězce NBTS.
`template<class Iter>` `string(Iter begin, Iter end)`	Inicializuje objekt string hodnotami z oblasti (begin, end), kde begin a end se chovají jako ukazatele a určují místo v paměti; begin do oblasti patří, ale end ne.

Program také používá přetížené operátory: += pro přidání jednoho řetězce k druhému, = pro přiřazení řetězce jinému řetězci, << pro zobrazení řetězce a [] pro přístup k jednotlivým prvkům řetězce.

Výpis 15.1 str.cpp

```
// str1.cpp-úvod do třídy string
#include <iostream>
#include <string>
using namespace std;
// použití konstruktorů třídy string
int main()
{
    string one("Bolek zvitezil v loterii!");       //1.konstruktor (ctor #1)
    cout <<one <<endl;                              //přetížený operátor <<
    string two(20,'$');                             //2.konstruktor (ctor #2)
    cout <<two <<endl;
    string three(one);                              //3.konstruktor (ctor #3)
    cout <<three <<endl;
    one +="Moment ..." ;                            //přetížený operátor +=
    cout <<one <<endl;
    two ="Prominte.Melo byt ";
    three [0 ]='L ';
    string four;                                    //4.konstruktor (ctor #4)
    four =two +three;                               //přetížený operátor =
    cout <<four <<endl;
    char alls []="Kdo se smeje naposled,ten se smeje nejlip ";
    string five(alls,22);                           //5.konstruktor
    cout <<five <<"...\n ";
    string six(alls +22,alls +42);                  //6.konstruktor (ctor #6)
    cout <<six <<"!...";
    string seven(&six [0 ],&six [19 ]);             //opět 6.konstruktor
    cout <<seven <<"!\n ";

    return 0;
}
```

Kompatibilita:

Některé starší implementace třídy `string` nepodporují konstruktor č. 6.

Zde je výstup programu:

```
Bolek zvitezil v loterii!
$$$$$$$$$$$$$$$$$$$$
Bolek zvitezil v loterii!
Bolek zvitezil v loterii! Moment …
Prominte. Melo byt Lolek zvitezil v loterii!
Kdo se smeje naposled, …
ten se smeje nejlip! … ten se smeje nejlip!
```

Poznámky k programu

Začátek programu dokládá, že objekt třídy `string` můžete inicializovat pomocí běžného řetězce jazyka C a zobrazit pomocí operátoru <<:

```
string one("Bolek zvitezil v loterii!"); // 1. konstruktor
cout << one << endl;                     // přetížený operátor <<
```

Další konstruktor inicializuje druhý objekt třídy `string` pomocí řetězce s 20 znaky „$":

```
string two(20, '$');                     // 2. konstruktor
```

Kopírovací konstruktor inicializuje třetí objekt třídy `string` pomocí prvního:

```
string three(one);                       // 3. konstruktor
```

Přetížený operátor += přidá k prvnímu řetězci řetězec „Moment ...":

```
one += " Moment ...";                    // přetížený operátor +=
```

V tomto konkrétním příkladě je objektu třídy `string` přidán řetězec jazyka C. Operátor += je však přetížen vícekrát a můžete tedy také spojovat objekty `string` a jednotlivé znaky:

```
one += two;      // připojí objekt třídy string (není v programu)
one += '!';      // připojí hodnotu typu char (není v programu)
```

Podobným způsobem je přetížen operátor =, takže můžete přiřadit objekt třídy `string` jinému objektu této třídy, řetězec jazyka C objektu třídy `string` nebo jediný `znak objektu` třídy `string`:

```
two = "Prominte. Melo byt "; // přiřadí řetězec jazyka C
two = one;    // přiřadí objekt třídy string (není v programu)
two = '?';    // přiřadí hodnotu typu char (není v programu)
```

Díky přetížení operátoru `[]`, jako v příkladu s třídou `ArrayDb` v kapitole 13, je možné přistupovat k jednotlivým znakům objektu `string` pomocí zápisu používaného pro pole:

```
three[0] = 'L';
Implicitní konstruktor vytvoří prázdný řetězec, kterému lze hodnotu
  dodat později:
string four;                             // 4. konstruktor
four = two + three;                      // přetížené operátory +, =
```

V druhém řádku se pomocí přetíženého operátoru + vytvoří dočasný objekt třídy `string`, který se potom přiřadí objektu `four` pomocí přetíženého operátoru =. Jak asi očekáváte, operátor + spojí oba operandy do jediného objektu třídy string. Tento operátor je přetížen vícekrát, takže druhým operandem může být objekt třídy `string`, řetězec jazyka C i `znak`.

Pátý konstruktor použije jako parametry řetězec jazyka C a celé číslo, kde toto celé číslo udává počet zkopírovaných znaků:

```
string five(alls,22);                    // 5. konstruktor
```

Zde, jak je z výstupu vidět, se pro inicializaci objektu `five` použije pouze prvních 22 znaků („`Kdo se smeje naposled,`"). Podle poznámky z tabulky 15.1 se požadovaný počet znaků zkopíruje i tehdy, jestliže přesahuje délku řetězce jazyka C. Pokud tedy ve výše uve-

deném příkladě nahradíte číslo 22 číslem 50, zkopíruje se na konec objektu `five` 8 zbytečných znaků. (To znamená, že konstruktor bude interpretovat obsah paměti za řetězcem „Kdo se smeje naposled, ten se smeje nejlip" jako znakové kódy.)

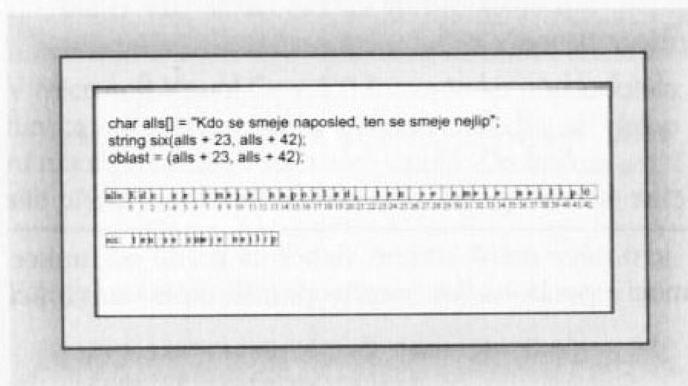
Šestý konstruktor používá šablonový parametr:

```
template<class Iter> string(Iter begin, Iter end);
```

Záměrem je, aby se `begin` a `end` chovaly jako ukazatele do dvou míst v paměti. (Obecně `begin` a `end` mohou být iterátory, generalizované ukazatele hojně používané ve standardní knihovně šablon.) Konstruktor potom hodnotami z oblasti mezi `begin` a `end` inicializuje vytvářený objekt třídy `string`. Zápis `[begin, end)` vypůjčený z matematiky znamená, že `begin` do oblasti patří, ale `end` ne. To znamená, že `end` ukazuje na místo do paměti, které následuje bezprostředně za poslední používanou hodnotou. Uvažujte následující příkaz:

```
string six(alls + 23, alls + 42);            // 6. konstruktor
```

Protože název pole je ukazatel, jsou oba výrazy `alls + 23` a `alls + 42` typu `char *`, takže v šabloně je parametr Iter nahrazen typem `char *`. První parametr ukazuje na pozici 23 v poli `array` a druhý do prostoru za koncem pole. To znamená, že objekt six je inicializován řetězcem „`ten se smeje nejlip`". Práci konstruktoru zachycuje obrázek 15.1.



Obrázek 15.1 Konstruktor třídy `string` používající rozsah

Předpokládejme, že chcete pomocí tohoto konstruktoru inicializovat objekt částí jiného objektu třídy `string`, například `five`. Následující příkaz fungovat nebude:

```
string seven(six + 0, six + 19);
```

Příčinou je, že s názvem objektu se na rozdíl od názvu pole nezachází jako s adresou objektu, a protože six není ukazatelem, je výraz `six + 0` bezvýznamný. Výraz `six[0]` však představuje hodnotu znaku a výraz `&six[0]` je adresou, kterou lze použít jako parametr konstruktoru:

```
string seven(&six[0], &six[19]);  // opět 6. konstruktor
```

Vstup třídy string

Další užitečnou věcí, kterou o třídě potřebujete vědět, jsou možnosti zadávání dat. Vzpomeňte si, že u řetězců jazyka C máte tři možnosti:

```
char info[100];
cin >> info;              // načte slovo
cin.getline(info, 100);   // načte řádek a zahodí znak \n
cin.get(info, 100);       // načte řádek a ponechá znak \n ve frontě
```

Jaké možnosti tedy máte při zadávání dat do objektu třídy string? Třída string přetěžuje operátor >> téměř stejně jako třída String v kapitole 11. Protože první operand není objektem třídy string, není funkce operátoru >> metodou třídy. Je to obecná funkce, jejímž prvním parametrem je objekt třídy istream a druhým objekt třídy string. Aby byl zachován soulad s operátorem >> z jazyka C, načte verze operátoru třídy string také jedno slovo a ukončí vstup při nalezení bílého znaku, znaku konce souboru nebo načte-li maximální možný počet znaků, který lze do objektu třídy string uložit. Funkce nejdříve vymaže obsah cílového řetězce a potom postupně načítá a přidává znaky. Pokud je délka vstupních znaků kratší než maximální možný počet znaků objektu třídy string, upraví funkce operator>>(istream &, string &) objekt třídy string automaticky na délku vstupního řetězce. Výsledek je, že můžete použít operátor >> u objektů třídy string stejně jako u řetězců jazyka C, ale nemusíte se obávat překročení délky pole:

```
char fname[10];
cin >> fname;      // bude-li počet vstupních znaků > 9, nastane problém
string lname;
cin >> lname;      // může načíst hodně dlouhé slovo
```

Použití rovnocenné funkce getline() je o něco méně zřejmé, neboť na rozdíl od funkce operator>>() ji nemůžete zapsat pomocí operátoru, ale musíte použít zápis označující členství:

```
cin.getline(fname, 10);
```

Pokud byste chtěli stejnou syntaxi použít u objektu třídy string, museli byste do třídy istream přidat novou členskou funkci, což by nebylo rozumné. V knihovně pro práci s řetězci je naproti tomu definována nečlenská funkce getline(), jejímž prvním parametrem je objekt třídy istream a druhým objekt třídy string. Pro načtení vstupního řádku do objektu třídy string se tedy funkce použije následujícím způsobem:

```
string fullName;
getline(cin, fullName);   // místo cin.getline(fname, 10)
```

Funkce getline() nejdříve vymaže obsah cílového řetězce a potom načítá postupně znaky ze vstupní fronty a přidává je k řetězci. To se děje až do chvíle, než dosáhne konce řádku, konce souboru nebo zaplní maximální kapacitu objektu. Pokud narazí na znak nového řádku, přečte ho, ale do řetězce ho neuloží. Všimněte si, že na rozdíl od verze třídy istream nemá verze třídy string parametr velikosti označující maximální počet znaků, který lze načíst. Program ve výpisu 15.2 předvádí použití obou možností vstupu:

Výpis 15.2 str2.cpp

```
// str2.cpp - vstup řetězce
#include <iostream>
#include <string>
using namespace std;
int main(){
    string word;
    cout << "Zadejte radek: ";
    cin >> word;
    while (cin.get() != '\n')
        continue;
    cout << "Chtel jsem jen " << word << ".\n";
    string line;
    cout << "Zadejte skutecny radek: ";
    getline(cin, line);
    cout << "Radek: " << line << endl;
    return 0;
}
```

Kompatibilita:

V Microsoft Visual C++ 5.0 (poznámka překladatele: týká se i verze 6.0) je v implementaci funkce `getline()` chyba, v jejímž důsledku se výstup z této funkce zobrazí teprve po zadání nějakých dalších vstupních údajů. Do funkce `getline()` od Borland C++ Builder 1.0 musíte přidat explicitně oddělovač: `getline(cin, line, '\n')`.

Zde je ukázka běhu: Cas jsou peníze.

```
Zadejte radek: Chtel jsem jen Cas.
Zadejte skutecny radek:
Kdo si pocka, ten se docka.
Radek: Kdo si pocka, ten se docka.
```

Práce s řetězci

Již jste se naučili různé způsoby vytváření objektů třídy `string`, zobrazovat jejich obsah, načíst a přidat data do objektu, přiřazovat a spojovat objekty. Co dalšího můžete dělat?

Můžete řetězce porovnávat. Všech šest relačních operátorů je pro objekty třídy `string` přetíženo, přičemž za menší se považuje ten objekt, který se v posloupnosti řazení počítače vyskytuje dříve. Je-li touto posloupností kód ASCII, mají číslice menší hodnotu než velká písmena a velká písmena mají menší hodnotu než písmena malá. Každý relační operátor je přetížen třemi způsoby, takže objekt třídy `string` můžete porovnat s jiným objektem téže třídy, s řetězcem jazyka C, a řetězec jazyka C můžete porovnat s objektem třídy `string`:

```
string snake1("kobra");
string snake2("krajta");
```

```
string snake3[20] "anakonda";
if (snake1 < snake2)      // operator<(const string &, const string &)
...
if (snake1 == snake3)     // operator==(const string &, const char *)
...
if (snake3 != snake2)     // operator!=(const char *, const string &)
```

Velikost řetězce můžete určit. Počet znaků v řetězci vracejí členské funkce `size()` a `length()`:

```
if (snake1.length() == snake2.size())
  cout << "Oba retezce jsou stejne dlouhe.\n"
```

Proč dělají stejnou věc dvě funkce? Členská funkce `length()` pochází z dřívějších verzí třídy `string`, zatímco funkce `size()` byla přidána kvůli kompatibilitě se standardní knihovnou šablon.

Vyhledávat určitý podřetězec nebo znak v řetězci můžete různými způsoby. V tabulce 15.2 je stručný popis čtyř variant metody `find()`. Vzpomeňte si, že proměnná `string::npos` označuje maximální možný počet znaků v řetězci a obvykle odpovídá největší hodnotě typů `unsigned int` nebo `unsigned long`.

Tabulka 15.2 Přetížená metoda `find()`.

Prototyp metody	Popis
`size_type find(const string & str, size_type pos = 0) const`	Najde první výskyt podřetězce *str* od pozice *pos* ve volaném řetězci. V případě nalezení vrací index prvního znaku podřetězce, jinak `string::npos`.
`size_type find(const char * s, size_type pos = 0) const`	Najde první výskyt podřetězce *s* od pozice *pos* ve volaném řetězci. V případě nalezení vrací index prvního znaku podřetězce, jinak `string::npos`.
`size_type find(const char * s, size_type pos = 0, size_type n)`	Najde první výskyt podřetězce *s* sestávajícího z prvních *n* znaků od pozice *pos* ve volaném řetězci. V případě nalezení vrací index prvního znaku podřetězce, jinak `string::npos`.
`size_type find(char ch, size_type pos = 0) const`	Najde první výskyt znaku *ch* od pozice *pos* ve volaném řetězci. V případě nalezení vrací index prvního znaku podřetězce, jinak `string::npos`.

Knihovna nabízí rovněž spřízněné metody `rfind()`, `find_first_of()`, `find_last_of()`, `find_first_not_of()` a `find_last_not_of()`, přičemž každá z nich obsahuje stejnou sadu přetížených funkčních signatur jako metoda `find()`. Funkce `rfind()` najde poslední výskyt podřetězce nebo znaku. Funkce `find_first_of()` najde ve volaném řetězci první výskyt libovolného ze znaků v parametru. Například příkaz

```
int where = snake1.find_first_of("harc");
```

by vrátil pozici znaku „r" v řetězci "kobra" (to znamená index č. 3), protože se jedná o první výskyt některého z písmen „hark" v řetězci „kobra". Metoda find_last_of() funguje stejně až na to, že vrací výskyt poslední. Příkaz

```
int where = snake1.find_last_of("harc");
```

by vrátil pozici znaku „a" v řetězci „kobra". Metoda find_first_not_of() najde první znak ve volaném řetězci, který se nenachází v parametru. Takže příkaz

```
int where = snake1.find_first_not_of("harc");
```

by vrátil pozici znaku „k" v řetězci „kobra", neboť se nenachází v parametru harc. Popis metody find_last_not_of() ponecháváme jako cvičení pro čtenáře.

Metod je mnohem víc, ale tyto stačí k sestavení ukázkového programu, který je graficky horší verzí hry se slovy zvané Hangman. Do pole objektů třídy string program uloží seznam slov, vybere náhodné slovo a nechá vás hádat písmena ve slově. Pokud šestkrát neuhodnete, prohráváte. Program kontroluje hádání pomocí funkce find() a pomocí operátoru += vytváří objekt třídy string, který zaznamenává špatné pokusy hádání. Pro správné pokusy vytvoří slovo, která má stejnou délku jako slovo hádané, ale obsahuje pomlčky. Ty jsou potom nahrazeny správně uhodnutými písmeny. Program je ve výpisu 15.3.

Výpis 15.3 str3.cpp // str3.cpp – ukázka některých metod třídy string

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <cctype>
using namespace std;
const int NUM = 26;
const string wordlist[NUM] = {"vcelin", "kobyla", "jecmen",
    "noviny", "vlajka", "ibisek", "fialka", "zdravi", "urazka",
    "medved", "stovik", "hribek", "moudry", "laciny", "stolek",
    "prehoz", "perina", "daleko", "siroko", "koleje", "cirkus",
    "platny", "candat", "kamzik", "touzit", "zirafa"};
int main()
{
    srand(time(0));
    char play;
    cout << "Chcete si zahrat hru se slovy? <a/n> ";
    cin >> play;
    play = tolower(play);
    while (play == 'a')
    {
      string target = wordlist[rand() % NUM];
      int length = target.length();
      string attempt(length, '-');
      string badchars;
      int guesses = 6;
      cout << "Dam vam hadat slovo. Ma " << length
           << " pismen a budete hadat\n"
           << "vzdy jedno pismeno. Jestlize " << guesses
```

```
                << "x neuhodnete, prohravate.\n";
            cout << "Zde je slovo: " << attempt << endl;
            while (guesses > 0 && attempt != target)
            {
                char letter;
                cout << "Hadejte pismeno: ";
                cin >> letter;
                if (badchars.find(letter) != string::npos
                    || attempt.find(letter) != string::npos)
                {
                    cout << "To jste jiz hadali. Zkuste znovu.\n";
                        continue;
                }
                int loc = target.find(letter);
                if (loc == string::npos)
                {
                    cout << "Chyba!\n";
                    --guesses;
                    badchars += letter; // přidá písmeno do řetězce
                }
                else
                {
                    cout << "Spravne!\n";
                    attempt[loc] = letter;
                    // kontroluje další výskyt písmene
                    loc = target.find(letter, loc + 1);
                    while (loc != string::npos)
                    {
                        attempt[loc]=letter;
                        loc = target.find(letter, loc + 1);
                    }
                }
                cout << "Slovo je: " << attempt << endl;
                if (attempt != target)
                {
                    if (badchars.length() > 0)
                        cout << "Chybne pokusy: " << badchars << endl;
                        cout << "Zbyva " << guesses << " pokusu.\n";
                }
            }
            if (guesses > 0)
                cout << "Gratuluji!\n";
            else
                cout << "Bohuzel, slovo je " << target << ".\n";
            cout << "Chcete pokracovat? <a/n> ";
            cin >> play;
            play = tolower(play);
        }
        cout << "Nashledanou!\n";
        return 0;
    }
```

Zde je ukázka běhu:

```
Chcete si zahrat hru se slovy? <a/n> a
Dam vam hadat slovo. Ma 6 pismen a budete hadat
vzdy jedno pismeno. Jestlize 6x neuhodnete, prohravate.
Zde je slovo: ------
Hadejte pismeno: e
Chyba!
Zde je slovo: ------
Chybne pokusy: e
Zbyva 5 pokusu.
Hadejte pismeno: a
Spravne!
Zde je slovo: --a--a
Chybne pokusy: e
Zbyva 5 pokusu.
Hadejte pismeno: t
Chyba!
Zde je slovo: --a--a
Chybne pokusy: et
Zbyva 4 pokusu.
Hadejte pismeno: k
Spravne!
Zde je slovo: --a-ka
Chybne pokusy: et
Zbyva 4 pokusu.
Hadejte pismeno: l
Spravne!
Zde je slovo: -la-ka
Chybne pokusy: et
Zbyva 4 pokusu.
Hadejte pismeno: v
Spravne!
Zde je slovo: vla-ka
Chybne pokusy: et
Zbyva 4 pokusu.
Hadejte pismeno: j
Spravne!
Zde je slovo: vlajka
Gratuluji!
Chcete pokracovat? <a/n> n
Nashledanou!
```

Poznámky k programu

Vzhledem k tomu, že relační operátory jsou přetíženy, můžete s nimi pracovat jako s numerickými proměnnými:

```
while (guesses > 0 && attempt != target)
```

Takový příkaz je srozumitelnější, než například funkce `strcmp()` používaná při práci s řetězci jazyka C.

Pomocí funkce find() program zjistí, zda byl znak již dříve vybrán; pokud ano, najde se buď v řetězci badchars (špatné pokusy) nebo v řetězci attempt (dobré pokusy):

```
if (badchars.find(letter) != string::pos
  || attemt.find(letter) != string::pos)
```

Proměnná npos je statickým členem třídy string. Vzpomeňte si, že její hodnotou je maximální možný počet znaků objektu třídy string. Protože indexování začíná od nuly, je tato hodnota o 1 větší než maximální možný index a lze ji použít k označení chyby při hledání znaku nebo řetězce.

Program využívá skutečnosti, že jedna z přetížených verzí operátoru += umožňuje přidávat k řetězci jednotlivé znaky:

```
badchars += letter;          // přidá znak k objektu třídy string
```

Srdce programu začíná kontrolou vybraného písmene v hádaném slově:

```
int loc = target.find(letter);
```

Pokud je hodnota proměnné loc platná, vloží se toto písmeno do odpovídajícího místa ve výstupním řetězci:

```
attempt[loc] = letter;
```

Toto písmeno se však může v hádaném slově vyskytnout vícekrát, proto program musí provádět další kontroly. Použije nepovinný druhý parametr funkce find(), kam můžete zadat počáteční pozici, od které se v řetězci bude hledat. Protože písmeno se našlo na pozici loc, začne další hledání na pozici loc + 1. Hledání pokračuje v cyklu while tak dlouho, dokud se nenalezne poslední výskyt tohoto znaku. Všimněte si, že funkce find() ohlásí chybu, jestliže se proměnná loc nachází za koncem řetězce:

```
// kontroluje další výskyt písmene
loc = target.find(letter, loc + 1);
while (loc != string::pos)
{
  attemt[loc] = letter;
  loc = target.find(letter, loc + 1);
}
```

Co dál

Knihovna pro práci s řetězci nabízí mnoho dalších prostředků. Existují funkce pro vymazání části řetězce nebo celého řetězce, nahrazení části nebo jednoho celého řetězce částí řetězce jiného nebo celým jiným řetězcem, vložení znaku do řetězce nebo vyjmutí znaků z řetězce, porovnání části nebo celého jednoho řetězce s částí nebo celým řetězcem jiným, a funkce pro vyjmutí podřetězce z řetězce. Existuje funkce pro zkopírování části jednoho řetězce do řetězce jiného a funkce pro výměnu obsahu dvou řetězců. Většina z těchto funkcí je přetížena, takže fungují jak pro řetězce jazyka C, tak i pro objekty třídy string. Funkce pro práci s řetězci jsou stručně popsány v příloze F.

V této části pracujeme s třídou `string` tak, jako by byla založena na typu `char`. Jak jsme se však zmínili dříve, je knihovna pro práci s řetězci ve skutečnosti založena na šablonové třídě:

```
template<class charT, class traits = char _traits<charT>,
class Allocator = allocator<charT> >
basic_string {...}
```

Třída obsahuje dva příkazy s klíčovým slovem `typedef`:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Díky tomu můžete používat řetězce založené na typu `wchar_t` i na typu `char`. Můžete dokonce vytvořit nějaký druh znakové třídy a použít u ní šablonu třídy `basic_string`, bude-li vaše třída splňovat určité požadavky. Třída `traits` popisuje určitá fakta o vybraném typu znaku, například způsob porovnávání hodnot. Pro typy `char` a `wchar_t` existují předdefinované specializace šablony `char_traits` a ty představují implicitní hodnoty třídy `traits`. Třída `Allocator` je třídou spravující přidělování paměti. Pro typy `char` a `wchar_t` existují předdefinované specializace šablony `allocator` a ty jsou implicitní. Používají obvyklým způsobem operátory `new` a `delete`, ale můžete si vyhradit úsek paměti a vytvořit vlastní přidělovací metody.

Třída auto_ptr

Třída `auto_ptr` je šablonová třída, která se stará o přidělování dynamické paměti. Podívejme se na případy, kdy ji můžeme potřebovat a jak při tom postupovat. Uvažujte následující funkci:

```
void remodel(string & str)
{
  string * ps = new string(str);
  ...
  str = ps;
  return;
}
```

Závadu pravděpodobně vidíte. Při každém volání funkce je přidělena paměť z haldy, která však již není vrácena a vzniká tak únik paměti. Řešení také znáte – stačí si vzpomenout a uvolnit přidělenou paměť vložením následujícího příkazu před příkaz `return`:

```
delete ps;
```

Avšak řešení opírající se o frázi „stačí si vzpomenout" je zřídkakdy nejlepší. Někdy si nevzpomenete nebo příkaz omylem odstraníte či z něj uděláte poznámku. A i když si vzpomenete, problémy přesto mohou nastat. Uvažujte následující variantu uvedené funkce:

```
void remodel(string & str)
{
  string * ps = new string(str);
  ...
```

```
    if (weird_thing())
        throw exception();
    str = *ps;
    delete ps;
    return;
}
```

Při vyvolání výjimky na provedení příkazu `delete` nedojde a opět nastane únik paměti.

Tuto chybu můžete opravit podle návodu z kapitoly 14, lepší by však bylo, kdyby existovalo čistší řešení. Zamysleme se, co je třeba udělat. Když taková funkce `remodel()` skončí, ať už normálně nebo vyvoláním výjimky, lokální proměnné se ze zásobníku odstraní a paměť obsazená ukazatelem `ps` se tedy uvolní. Bylo by dobré, kdyby se uvolnila i paměť, na kterou tento ukazatel ukazuje. Program by tedy po skončení platnosti ukazatele `ps` měl ještě něco provést. Tuto službu navíc nelze použít u základních typů, avšak u tříd ji použít můžete díky mechanismu destruktoru. Problém tedy představuje ukazatel `ps`, který je pouze obyčejným ukazatelem a ne objektem třídy. Pokud by objektem byl, mohli byste po skončení jeho platnosti uvolnit okazovanou paměť v destruktoru příkazem `delete`. A to je právě smyslem třídy `auto_ptr`.

Použití třídy auto_ptr

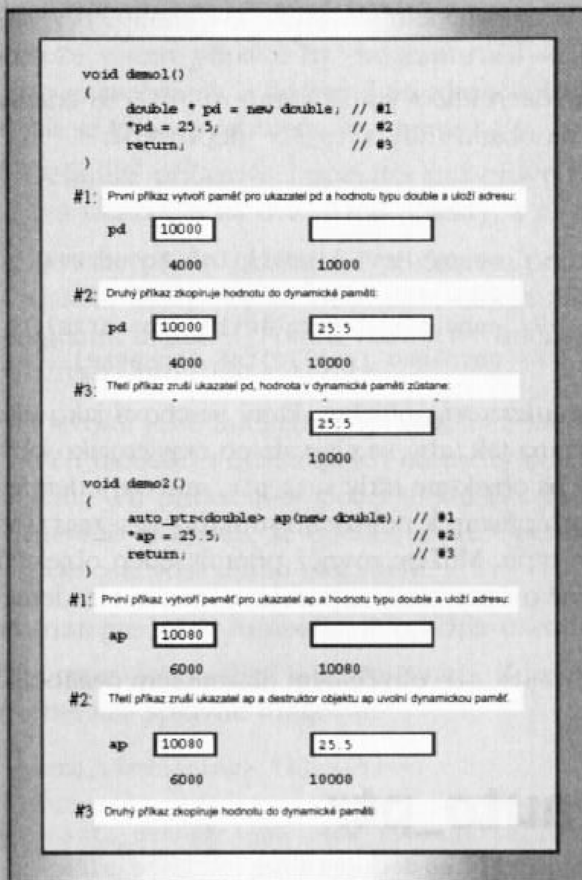
Šablona `auto_ptr` definuje objekt jako ukazatel, kterému bude přiřazena adresa získaná (přímo nebo nepřímo) pomocí operátoru `new`. Když platnost objektu třídy `auto_ptr` skončí, uvolní destruktor paměť příkazem `delete`. Jestliže tedy přiřadíte adresu získanou pomocí operátoru `new` objektu třídy `auto_ptr`, nebudete si muset pamatovat, že máte paměť později uvolnit; uvolní se automaticky, jakmile platnost tohoto objektu skončí. Rozdíl v chování mezi objektem třídy `auto_ptr` a obyčejným ukazatelem znázorňuje obrázek 15.2.

Abyste mohli vytvořit objekt třídy `auto_ptr`, vložte do programu hlavičkový soubor `memory` obsahující šablonu `auto_ptr`. Potom pomocí obvyklé syntaxe šablony vytvořte instanci požadovaného ukazatele. Šablona obsahuje následující příkaz:

```
template<class X> class auto_ptr {
public:
  explicit auto_ptr(X* p = 0) throw();
...}
```

(Vzpomeňte si, že zápis `throw()` znamená, že tento konstruktor výjimku nevolá.) Požádáte-li tedy o třídu `auto_ptr` typu `X`, získáte ukazatel na typ `X`:

```
auto_ptr<double> pd(new double);   // ukazatel auto_ptr na typ double
                                   // (místo double *)
auto_ptr<string> ps(new string);   // ukazatel auto_ptr na řetězec
                                   // (místo string *)
```



Obrázek 15.2

Zde výraz `new double` je ukazatel vrácený operátorem `new` na nově přidělený úsek paměti. Představuje parametr konstruktoru `auto_ptr<double>`, to znamená skutečný parametr odpovídající formálnímu parametru `p` v prototypu. Podobně je skutečným parametrem konstruktoru výraz `new string`.

Úpravu funkce `remodel()` tedy proveďte následujícími třemi kroky:

1. Vložte hlavičkový soubor `memory`.
2. Nahraďte ukazatel na `řetězec` objektem třídy `auto_ptr` na řetězec.
3. Odstraňte příkaz `delete`.

Zde je funkce s provedenými změnami:

```
#include <memory>
void remodel(string & str)
{
  auto_ptr<string> ps (new string(str));
  ...
  if (weird_thing())
          throw exception();
  str = *ps;
```

```
    //delete ps; není již potřeba
    return;
}
```

Všimněte si, že konstruktor auto_ptr je uvozen klíčovým slovem explicit, to znamená, že implicitně se přetypování z ukazatele na objekt třídy auto_ptr neprovede:

```
auto_ptr<double> pd;
double *p_reg = new double;
pd = p_reg;                          // nepovoleno (implicitní konverze)
pd = auto_ptr<double>(p_reg);        // povoleno (explicitní konverze)
auto_ptr<double> pauto = pd;         // nepovoleno (implicitní konverze)
auto_ptr<double> pauto(pd);          // povoleno (explicitní konverze)
```

Třída auto_ptr je příkladem ***inteligentního ukazatele***, objektu, který se chová jako ukazatel, ale má další vlastnosti. Třída je definována tak, aby se chovala co nejvíce jako obyčejný ukazatel. Pokud je například proměnná ps objektem třídy auto_ptr, můžete ji dereferencovat (*ps), inkrementovat (++ps), použít pro přístup k položkám struktury (ps->puffIndex) a přiřadit ji obyčejnému ukazateli stejného typu. Můžete rovněž přiřadit jeden objekt třídy auto_ptr jinému objektu stejného typu, čímž ovšem vznikne problém, kterému budeme čelit v další části.

Šablona umožňuje inicializovat objekt třídy auto_ptr obyčejným ukazatelem pomocí konstruktoru auto_ptr.

Poznámky ke třídě auto_ptr

Třída auto_ptr není všelékem. Uvažujte například následující kód:

```
auto_ptr<int> pi(new int [200]);  // nelze!
```

Nezapomeňte, že operátor delete musíte párovat s operátorem new a operátor delete[] s operátorem new[]. V šabloně auto_ptr se používá operátor delete, ne delete[], takže použít můžete pouze new a nikoli new[]. Ekvivalent šablony auto_ptr pro práci s dynamickými poli neexistuje. Mohli byste zkopírovat šablonu auto_ptr z hlavičkového souboru memory, přejmenovat ji na auto_arr_ptr a v kopii nahradit operátor delete operátorem delete[]. V takovém případě byste museli přidat podporu operátoru [].

```
Co třeba takto?string vacation(„Putoval jsem sam jako mrak.");
auto_ptr<string> pvac(&vacation); // nelze!
```

V tomto případě by operátor delete zasahoval do paměti mimo haldu, což nelze.

Upozornění

Objekt třídy auto_ptr používejte pouze při práci s pamětí přidělenou operátorem new, nikoli pro paměť přidělenou operátorem new[] nebo obyčejnou deklarací proměnné.

Nyní uvažujte přiřazení

```
auto_ptr<string> ps (new string("Panoval jsem sam jako mrak."));
auto_ptr<string> vocation;
vocation = ps;
```

K čemu by mělo toto přiřazení vést? Pokud by `ps` a `vocation` byly obyčejnými ukazateli, byly by výsledkem dva ukazatelé ukazující na stejný objekt třídy `string`. To není žádoucí, protože v tom případě by program rušil stejný objekt dvakrát, jednou při ukončení platnosti ukazatele `ps` a podruhé při ukončení platnosti ukazatele `vocation`. Existují způsoby, jak se tomuto problému vyhnout:

- Definujte přiřazovací operátor pro provedení hluboké kopie. Výsledkem budou dva ukazatele na dva různé objekty, z nichž jeden bude kopií druhého.
- Zaveďte pojem *vlastnictví*, při kterém pouze jeden inteligentní ukazatel může vlastnit určitý objekt. Destruktor objekt zruší pouze tehdy, bude-li ho vlastnit inteligentní ukazatel. Potom vlastnictví změňte pomocí přiřazení. Toto je strategie používaná pro třídu `auto_ptr`.
- Vytvořte ještě inteligentnější ukazatel, který bude zaznamenávat počet inteligentních ukazatelů odkazujících na určitý objekt. Tento jev se nazývá *počítání referencí*. Při přiřazení se počet ukazatelů o jeden zvýší a při skončení platnosti některého ukazatele se o jeden sníží. Operátor `delete` se vyvolá teprve při skončení platnosti posledního ukazatele.

Stejná strategie by se samozřejmě použila u kopírovacích konstruktorů.

Každý postup se používá v určité situaci. Zde je například situace, kdy objekty třídy `auto_ptr` nemusí správně fungovat:

```
auto_ptr<string> films[5] =
{
   auto_ptr<string> (new string("Skola zaklad zivota")),
   auto_ptr<string> (new string("Anton Spelec ostrostrelec")),
   auto_ptr<string> (new string("Svatba jako remen")),
   auto_ptr<string> (new string("Studaci a kantori")),
   auto_ptr<string> (new string("S tebou me bavi svet"))
};
auto_ptr<string> pwin(films[2]);
int i;
cout << "Do souteze o nejlepsi komedii stoleti byly nominovany tyto fil-
my\n";
for (i = 0; i < 5; i++)
   cout << *films[i] << endln;
cout << "Viteznym filmem je " << *pwin << "!\n";
```

Problémem je, že při změně vlastnictví z `films[2]` na `pwin` už `films[2]` nemusí na řetězec ukazovat. To znamená, že po předání vlastnictví již objekt třídy `auto_ptr` nemusí být použitelný. O jeho použitelnosti či nepoužitelnosti rozhoduje implementace.

Standardní knihovna šablon

Standardní knihovna šablon (Standard Template Library, STL) obsahuje kolekci šablon reprezentujících ***kontejnery***, ***iterátory***, ***funkční objekty*** a ***algoritmy***. Kontejner je jednotka, která může stejně jako pole obsahovat několik hodnot. Kontejnery standardní knihovny šablon jsou homogenní, všechny tedy obsahují hodnoty stejného typu. Algoritmy předsta-

vují recept pro splnění určitých úkolů, například setřídění pole nebo nalezení určité hodnoty v seznamu. Iterátory jsou objekty umožňující v kontejneru podobný pohyb, jaký umožňují ukazatele v poli; představují generalizované ukazatele. Funkční objekty jsou objekty, které se chovají jako funkce; mohou být objekty tříd nebo ukazateli na funkce (k tomu potřebují název funkce, protože název funkce se chová jako ukazatel). Díky standardní knihovně šablon můžete vytvářet různé kontejnery včetně polí, front a seznamů a provádět různé operace jako prohledávání, třídění a přímý přístup.

Standardní knihovnu šablon vyvinuli v laboratořích Hewlett-Laboratories Alex Stepanov a Meng Lee a na trh byla uvedena v roce 1994. Výbor ISO/ANSI C++ odhlasoval její začlenění jako součásti standardu jazyka C++. Standardní knihovna šablon není příkladem objektově orientovaného programování. Představuje naopak odlišný vzor programování, kterému se říká *generické programování*. Díky tomu je zajímavá jednak tím, co dělá, i tím, jak to dělá. Informací o knihovně STL je příliš mnoho, než abychom je mohli podat v jediné kapitole, takže se podíváme na některé reprezentativní příklady a prostudujeme podstatu řešení. Začneme několika specifickými příklady. Jakmile kontejnerům, iterátorům a algoritmům dostatečně porozumíte, podíváme se na zásadní filozofii návrhu a potom stručně na celou knihovnu STL. Různé metody a funkce knihovny STL jsou shrnuty v příloze G.

Šablonová třída `vector`

V počítačové terminologii odpovídá termín *vektor* spíše poli než matematickému vektoru probíranému v kapitole 10. Vektor obsahuje množinu podobných hodnot, ke kterým lze přistupovat přímo. To znamená, že například pomocí indexu získáte přístup k desátému prvku vektoru, aniž byste museli nejdříve projít předchozích devět prvků. Třída `vector` by tedy obsahovala podobné operace jako třída `ArrayDb` z kapitoly 13. To znamená, že byste mohli vytvořit objekt třídy `vector`, přiřadit jeden objekt této třídy jinému objektu stejné třídy a pomocí operátoru `[]` přistupovat k prvkům vektoru. Aby byla tato třída generická, uděláme ji třídou šablonovou. Právě to dělá knihovna STL – definuje v hlavičkovém souboru `vector` (dříve `vector.h`) šablonu `vector`. Objekt šablony `vector` vytvoříte pomocí obvyklého zápisu <*type*> označujícího použitý typ. Šablona `vector` také používá dynamicky přidělovanou paměť a pomocí inicializačního parametru můžete stanovit potřebný počet prvků:

```
#include vector
using namespace std:
vector<int> ratings(5);           // vektor s 5 prvky typu int
int n;
cin >> n;
vector<double> scores(n);         // vektor s n prvky typu double
```

Jakmile objekt třídy `vector` vytvoříte, získáte díky přetíženému operátoru [] možnost přistupovat k jednotlivým prvkům obvyklým zápisem používaným pro pole:

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
   cout << scores[i] << endln;
```

Opět alokátory

Stejně jako třída `string` mají různé kontejnerové šablony knihovny STL nepovinný parametr určující alokační objekt, který se použije pro správu paměti. Šablona `vector` například začíná takto:

```
template <class> T, class Allocator = allocator<T> >
        class vector {...
```

Pokud hodnotu parametru v této šabloně vynecháte, použije kontejnerová šablona implicitně třídu `allocator<T>`. V této třídě se operátory `new` a `delete` používají standardním způsobem.

Program ve výpisu 15.4 tuto třídu používá v nenáročné aplikaci. Vytváří dva objekty třídy `vector`, jeden se specializací typu `int` a jeden se specializací typu `string`; každý objekt má pět prvků.

Výpis 15.4 vect1.cpp

```
// vect1.cpp – úvod do šablony vector
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const int NUM = 5;
int main(){
    vector<int> ratings(NUM);
    vector<string> titles(NUM);
    cout << "Postupujte presne podle navodu. Zadate\n"
         << NUM << " nazvu knih a ohodnotite je znamkou 0-10.\n";
    int i;
    for (i = 0; i < NUM; i++)
    {
        cout << "Zadejte nazev c. " << i + 1 << ": ";
        getline(cin,titles[i]);
        cout << "Zadejte hodnoceni znamkou 0-10: ";
        cin >> ratings[i];
        cin.get();
    }
    cout << "Dekuji. Zadali jste nasledujici nazvy:\n"
          << "Hodnoceni\tKniha\n";
    for (i = 0; i < NUM; i++)
    {
        cout << ratings[i] << "\t" << titles[i] << endl;
    }
    return 0;
}
```

Kompatibilita:

Starší implementace používají hlavičkový soubor `vector.h` namísto souboru `vector`. Ačkoli na pořadí vložených souborů by záležet nemělo, v `g++ 2.7.1` musel být hlavičkový soubor `string` uveden před hlavičkovými soubory knihovny STL. Funkce `getline()` v Microsoft Visual C++ 5.0 (poznámka překladatele: týká se i verze 6.0) obsahuje chybu při synchronizaci vstupu a výstupu, a funkce `getline()` v Borland C++ Builder požaduje jako parametr explicitní oddělovač.

Zde je ukázku běhu programu:

```
Postupujte presne podle navodu. Zadate
5 nazvu knih a ohodnotite je znamkou 0-10.
Zadejte nazev c. 1: Kocka, ktera znala C++
Zadejte hodnoceni znamkou 0-10: 6
Zadejte nazev c. 2: Pes, který steka, nekouse
Zadejte hodnoceni znamkou 0-10: 4
Zadejte nazev c. 3: Valecnici z Wonku
Zadejte hodnoceni znamkou 0-10: 3
Zadejte nazev c. 4: Nevhodna metafora
Zadejte hodnoceni znamkou 0-10: 5
Zadejte nazev c. 5: Panicky orientovane programovani
Zadejte hodnoceni znamkou 0-10: 8
Dekuji. Zadali jste nasledujici knihy:
HodnoceÍ Kniha
6        Kocka, ktera znala C++
4        Pes, který steka, nekouse
3        Valecnici z Wonku
4        Nevhodna metafora
7        Panicky orientovane programovani
```

Tento program pouze používá šablonu `vector` jako vhodný způsob pro vytvoření dynamicky alokovaného pole. Podívejme se na příklad, který používá více metod této třídy.

Co se dá s vektory dělat

K čemu dalšímu můžete šablonu `vector` použít? Všechny kontejnery knihovny STL obsahují určité základní metody, mezi které patří metoda `size()`, vracející počet prvků v kontejneru, metoda `swap()` měnící obsah dvou kontejnerů, metoda `begin()` vracející iterátor odkazující na první prvek v kontejneru a metoda `end()` vracející iterátor, který ukazuje *na místo bezprostředně za posledním prvkem kontejneru.*

Co je iterátor? Je to generalizovaný ukazatel. Může být ukazatelem nebo objektem, pro nějž jsou definovány operace používané pro ukazatele, například dereferencování (`operator*()`) a inkrementování (`operator++()`). Jak uvidíte později, může knihovna STL díky generalizování ukazatelů na iterátory nabídnout jednotné rozhraní pro kontejnerové třídy včetně těch, u kterých by jednoduché ukazatele nefungovaly. V každé kontejnerové třídě je definován vhodný iterátor. Typ tohoto iterátoru se nazývá `iterator` a jeho plat-

nost je v rozsahu třídy. Pokud byste například chtěli deklarovat iterátor pro šablonu `vector` se specializací typu `double`, napsali byste následující příkaz:

```
vector<double>::iterator pd;        // iterátor pd
```

Předpokládejme, že `scores` je objektem šablony `vector<double>`:

```
vector<double> scores;
```

Potom budete moci iterátor `pd` použít k následujícím operacím:

```
pd = scores.begin(); // pd ukazuje na první prvek
*pd = 22.3;          // dereferencuje pd a hodnotu přiřadí prvnímu prvku
++pd;                // pd ukazuje na následující prvek
```

Jak vidíte, chová se iterátor jako ukazatel.

Co je místo za kontejnerem? Jedná se o iterátor odkazující na prvek následující bezprostředně za posledním prvkem v kontejneru. Smysl je podobný jako u nulového znaku, který představuje poslední prvek za skutečným posledním znakem řetězce jazyka C, ale nulový znak je hodnotou prvku a místo za kontejnerem je ukazatel (nebo iterátor) na prvek. Toto místo identifikuje členská funkce `end()`. Jestliže iterátor nastavíte na první prvek v kontejneru a budete ho inkrementovat, dostanete se nakonec na místo za kontejnerem a projdete celý jeho obsah. Pokud tedy budou iterátory `scores` a `pd` definovány výše uvedeným způsobem, můžete obsah kontejneru zobrazit tímto kódem:

```
for (pd = scores.begin(); pd != scores.end(); pd++)
  cout << *pd << endl;
```

Právě probrané metody mají všechny kontejnery. Šablonová třída `vector` obsahuje také několik metod, které mají pouze některé kontejnery knihovny STL. Jednou z užitečných metod je metoda `push_back()`, která přidává prvek na konec vektoru. Přitom dohlíží na správu paměti, takže velikost vektoru se zvětšuje podle přidávaných prvků. Můžete tedy napsat následující kód:

```
vector<double> scores;    // vytvoří prázdný vektor
double temp;
while (cin >> temp && temp >= 0)
  scores.push_back(temp);
cout << "Zadali jste " scores.size() << "objektu scores.\n";
```

Při každém průchodu cyklem je do vektoru `scores` přidán jeden prvek. Počet prvků nemusíte určovat ani při psaní programu ani při jeho spuštění. Dokud program bude mít dostatek paměti, bude se velikost vektoru scores zvětšovat podle potřeby.

Existuje metoda `erase()`, která stanovený rozsah vektoru ruší. Má jako dva parametry iterátory definující rozsah, který má být zrušen. Porozumět způsobu, jakým knihovna STL definuje rozsah pomocí iterátorů, je důležité. První iterátor ukazuje na začátek bloku, zatímco druhý ukazuje o jednu pozici dál, než je konec. Například příkaz

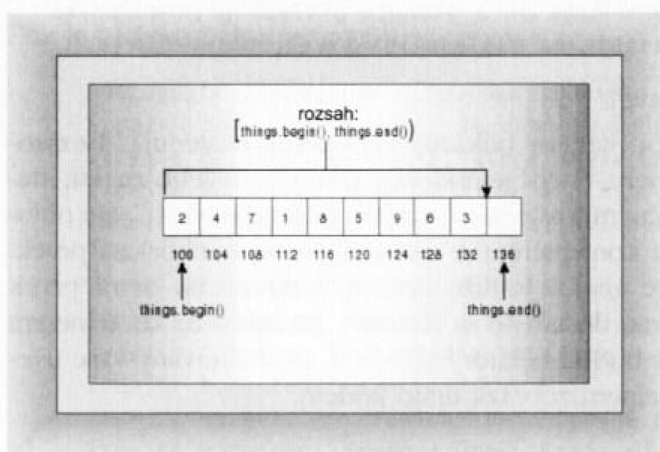
```
scores.erase(scores.begin(), scores.begin() + 2);
```

vymaže první dva prvky, to znamená ty, na které ukazují funkce `begin()` a `begin() + 2`. (Protože třída vector umožňuje přímý přístup, jsou pro iterátory definovány takové operace jako `begin() + 2`.) Pokud jsou `it1` a `it2` dva iterátory, používá se v literatuře STL

zápis `[p1, p2)`, který označuje rozsah se začátkem od p1 a koncem před `p2`. Rozsah `[begin(), end())` tedy pokrývá celý obsah kolekce (viz obrázek 15.3). Rozsah `[p1,p1)` označuje prázdný blok. Zápis `[)` není součástí jazyka C++ a v kódu se tedy nevyskytuje; objevuje se pouze v dokumentaci.

Pamatujte

Rozsah `[it1, it2)` je stanoven dvěma iterátory, přičemž `it1` do oblasti spadá, ale `it2` ne.



Obrázek 15.3 Koncept rozsahu v knihovně STL

Metodu `erase()` doplňuje metoda `insert()`. Jako parametry má tři iterátory. První udává pozici, před kterou budou vloženy nové prvky. Druhý a třetí parametr definují velikost vkládaného rozsahu. Tento rozsah bývá běžně součástí jiného objektu. Například kód

```
vector<int> old;
vector<int> new;
...
old.insert(old.begin(), new.begin() + 1, new.end());
```

vloží všechny prvky kromě prvního z vektoru `new` před první prvek vektoru `old`. Mimochodem se jedná o případ, kdy se hodí prvek následující bezprostředně za koncem vektoru, neboť přidání prvků na konec usnadňuje:

```
old.insert(old.end(), new.begin() + 1, new.end());
```

V tomto případě budou nové prvky vloženy před `old.end()`, tedy za poslední prvek vektoru.

Program ve výpisu 15.5 předvádí použití metod `size()`, `begin()`, `end()`, `push_back()` a `insert()`. Z důvodu jednoduší práce s daty jsou komponenty `rating` a `title` z výpisu 15.4 zahrnuty do jediné struktury `Review` a o vstup a výstup objektů této struktury se starají funkce `FillReview()` a `ShowReview()`.

Výpis 15.5 vect2.cpp

```
// vect2.cpp - metody a iterátory
#include <iostream>
#include <string>
#include <vector>
using namespace std;
struct Review {
    string title;
    int rating;
};
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
int main()
{
    vector<Review> books;
    Review temp;
    while (FillReview(temp))
        books.push_back(temp);
    cout << "Dekuji. Zadali jste nasledujici knihy:\n"
         << "Hodnoceni\tKniha\n";
    int num = books.size();
    for (int i = 0; i < num; i++)
        ShowReview(books[i]);
    cout << "Opakovani zadanych knih:\n"
         << "Hodnoceni\tKniha\n";
  vector<Review>::iterator pr;
    for (pr = books.begin(); pr != books.end(); pr++)
        ShowReview(*pr);
    vector <Review> oldlist(books);     // použije kopírovací konstruktor
    if (num > 3)
    {
        // odstraní 2 položky
        books.erase(books.begin() + 1, books.begin() + 3);
        cout << "Seznam knih po vymazani:\n";
        for (pr = books.begin(); pr != books.end(); pr++)
            ShowReview(*pr);
        // vloží 1 položku
        books.insert(books.begin(), oldlist.begin() + 1, oldlist.begin()+2);
        cout << " Seznam knih po vlozeni:\n";
        for (pr = books.begin(); pr != books.end(); pr++)
            ShowReview(*pr);
    }
    books.swap(oldlist);
    cout << "Puvodni seznam knih:\n";
    for (pr = books.begin(); pr != books.end(); pr++)
        ShowReview(*pr);
    return 0;
}
bool FillReview(Review & rr)
{
```

```
    cout << "Zadejte nazev knihy (konec pro ukonceni): ";
    getline(cin,rr.title);
    if (rr.title == "konec")
        return false;
    cout << "Zadejte hodnoceni knihy: ";
    cin >> rr.rating;
    if (!cin)
        return false;
    cin.get();
    return true;
}
void ShowReview(const Review & rr)
{
    cout << rr.rating << "\t" << rr.title << endl;
}
```

Kompatibilita:

Starší implementace používají hlavičkový soubor `vector.h` namísto souboru `vector`. Ačkoli na pořadí vložených souborů by záležet nemělo, v `g++ 2.7.1` musel být hlavičkový soubor `string` uveden před hlavičkovými soubory knihovny STL. Funkce `getline()` v Microsoft Visual C++ 5.0 (poznámka překladatele: týká se i verze 6.0) obsahuje chybu, v jejímž důsledku se výstup následujícího řádku zobrazí až po dalším vstupu, a operátory `<` a `==` musí být definovány pro typ uložený v objektu třídy `vector`. To znamená, že pro typ `Review` byste museli přidat definice funkcí `operator<()` a `operator==()`. Funkce `getline()` v Borland C++ Builder 1.0 požaduje jako parametr explicitní oddělovač.

Zde je ukázku běhu programu:

```
Zadejte nazev knihy (konec pro ukonceni): Kocka, ktera znala vektory
Zadejte hodnoceni: 5
Zadejte nazev knihy (konec pro ukonceni): Pes, který steka, nekouse
Zadejte hodnoceni: 7
Zadejte nazev knihy (konec pro ukonceni): Valecnici z Wonku
Zadejte hodnoceni: 4
Zadejte nazev knihy (konec pro ukonceni): Kvantovy styl
Zadejte hodnoceni: 8
Zadejte nazev knihy (konec pro ukonceni): konec
Dekuji. Zadali jste nasledujici knihy:
Hodnoceni Kniha
5         Kocka, ktera znala vektory
7         Pes, který steka, nekouse
4         Valecnici z Wonku
8         Kvantovy styl
Opakovani zadanych knih:
Hodnoceni Kniha
5         Kocka, ktera znala vektory
7         Pes, který steka, nekouse
4         Valecnici z Wonku
8         Kvantovy styl
```

```
Seznam knih po vymazani:
5         Kocka, ktera znala vektory
8         Kvantovy styl
Seznam knih po vlozeni:
7         Pes, který steka, nekouse
5         Kocka, ktera znala vektory
8         Kvantovy styl
Puvodni seznam knih:
5         Kocka, ktera znala vektory
7         Pes, který steka, nekouse
4         Valecnici z Wonku
8         Kvantovy styl
```

Další možnosti práce s vektory

Existuje mnoho operací, které programátoři s poli běžně provádějí, například prohledávání pole, třídění, libovolné uspořádání prvků pole a tak dále. Obsahuje šablonová třída `vector` metody pro tyto běžné operace? Ne. Cíle knihovny STL jsou dalekosáhlejší. Pro tyto operace definuje *nečlenské funkce*. Místo definování samostatné členské funkce `find()` pro každou kontejnerovou třídu definuje jedinou nečlenskou funkci `find()`, kterou lze použít ve všech kontejnerových třídách. Tato filozofie návrhu ušetří mnoho stejné práce. Předpokládejme například, že byste měli 8 kontejnerových tříd a potřebovali podporu pro 10 operací. Pokud by každá třída měla svou vlastní členskou funkci, museli byste definovat 8 * 10, čili 80 členských funkcí. Při způsobu řešení používaném knihovnou STL potřebujete definovat pouze 10 nečlenských funkcí. A jestliže nadefinujete novou kontejnerovou třídu a budete postupovat podle příslušných pokynů, bude tato nová třída také moci používat 10 již existujících nečlenských funkcí pro vyhledávání, třídění a tak dále.

Prostudujeme tři reprezentativní funkce knihovny STL: `for_each()`, `random_shuffle()` a `sort()`. Funkci `for_each()` lze použít s každou kontejnerovou třídou. Má tři parametry: prvními dvěma jsou iterátory definující rozsah kontejneru a posledním je ukazatel na funkci. (Obecněji řečeno je posledním parametrem funkční objekt; o funkčních objektech se dozvíte za chvíli.) Funkce `for_each()` tento ukazatel na funkci používá pro každý prvek v rozsahu. Hodnotu prvků kontejneru tato funkce měnit nesmí. Funkci `for_each()` můžete použít místo cyklu `for`. Například následující kód:

```
vector<Review>::iterator pr;
for (pr = books.begin(); pr != books.end(); pr++)
    ShowReview(*pr);
```

můžete nahradit tímto:

```
for_each(books.begin(), books.end(), ShowReview);
```

Takto si nebudete muset špinit ruce (a kód) s používáním iterátorů jako explicitních proměnných.

Funkce `random_shuffle()` má jako parametry dva iterátory určující rozsah a libovolně změní pořadí prvků v tomto rozsahu. Například příkaz

```
random_shuffle(books.begin(), books.end());
```

změní náhodně pořadí všech prvků ve vektoru books. Na rozdíl od funkce for_each(), pracující s každou kontejnerovou třídou, požaduje tato funkce kontejnerovou třídu s přímým přístupem, což třída vector splňuje.

Třída sort() také požaduje kontejnerovou třídu podporující přímý přístup. Má dvě verze. První má jako parametry dva iterátory definující rozsah a třídí prvky v tomto rozsahu pomocí operátoru <, který je definován pro typ prvku uloženého v kontejneru. Například kód

```
vector<int> coolstuff;
...
sort(coolstuff.begin(), coolstuf.end());
```

setřídí obsah vektoru coolstuff vzestupně, přičemž hodnoty bude porovnávat pomocí vestavěného operátoru <.

Jestliže jsou prvky kontejneru uživatelem definované objekty, musíte pro tento typ definovat funkci operator<(), abyste mohli funkci sort() používat. Například vektor obsahující objekty struktury Review byste mohli třídit, pokud byste vytvořili členskou funkci struktury Review nebo nečlenskou funkci pro operator<(). Protože Review je struktura, jsou její položky veřejné a následující nečlenská funkce by vyhovovala:

```
bool operator<(const Review & r1, const Review & r2)
{
    if (r1.title < r2.title)
            return true;
    else if (r1.title == r2.title && r1.rating < r2.rating)
            return true;
    else
            return false;
}
```

Pomocí takové funkce byste potom mohli objekty struktury Review (například books) třídit:

```
sort(books.begin(), books.end());
```

Tato verze funkce operator<() třídí knihy podle názvů. Pokud budou mít dva objekty stejný název knihy, budou uspořádány podle hodnocení. Předpokládejme však, že chcete třídit sestupně nebo podle hodnocení místo názvů. V takovém případě můžete použít druhý formát funkce sort(). Ten má tři parametry. Prvními dvěma jsou opět iterátory označující rozsah. Posledním parametrem je ukazatel na funkci (obecněji řečeno funkční objekt), který se použije pro porovnání místo funkce operator<(). Vrácená hodnota by se měla převést na typ bool, přičemž hodnota false bude znamenat, že oba parametry se nacházejí ve špatném pořadí. Zde je příklad takové funkce:

```
bool WorseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
            return true;
    else
            return false;
}
```

Při existenci takové funkce můžete pomocí následujícího příkazu setřídit objekty `Review` vektoru `books` vzestupně podle jejich hodnocení:

```
sort(books.begin(), books.end(), WorseThan);
```

Všimněte si, že funkce `WorseThan()` netřídí objekty struktury `Review` tak dokonale jako funkce `operator<()`. Jestliže dva objekty obsahují knihu se stejným názvem, provede funkce `operator<()` třídění podle hodnocení. Pokud však budou mít dva objekty položku se stejným hodnocením, bude je funkce `WorseThan()` považovat za rovnocenné. První druh se nazývá *úplné uspořádání* a druhému se říká *striktní slabé uspořádání*. Při úplném uspořádání platí, že jestliže $a < b$ a $b < a$, musí být a i b totožné. Při striktním slabém uspořádání tomu tak není. Prvky mohou být totožné, nebo mohou mít stejný pouze jeden aspekt, například položku `rating` v příkladu s funkcí `WorseThan()`. Takže při použití striktně slabého uspořádání je lepší říkat, že objekty jsou rovnocenné, než že jsou totožné. Použití těchto funkcí knihovny STL objasňuje program ve výpisu 15.6.

Výpis 15.6 vect3.cpp

```
// vect3.cpp - použiti funkci ze standardni knihovny šablon
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
struct Review {
    string title;
    int rating;
};
bool operator<(const Review & r1, const Review & r2);
bool worseThan(const Review & r1, const Review & r2);
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
int main()
{
    vector<Review> books;
    Review temp;
    while (FillReview(temp))
        books.push_back(temp);
    cout << "Dekuji. Zadali jste nasledujici "
         << books.size() << " hodnoceni:\n"
          << "Hodnoceni\tKniha\n";
    for_each(books.begin(), books.end(), ShowReview);

    sort(books.begin(), books.end());
    cout << "Trideni dle nazvu:\nHodnoceni\tKniha\n";
    for_each(books.begin(), books.end(), ShowReview);

    sort(books.begin(), books.end(), worseThan);
    cout << "Trideni dle hodnoceni:\nHodnoceni\tKniha\n";
    for_each(books.begin(), books.end(), ShowReview);
```

```
    random_shuffle(books.begin(), books.end());
    cout << "Po promichani:\nHodnoceni\tKniha\n";
    for_each(books.begin(), books.end(), ShowReview);
    return 0;
}
bool operator<(const Review & r1, const Review & r2)
{
    if (r1.title < r2.title)
        return true;
    else if (r1.title == r2.title && r1.rating < r2.rating)
        return true;
    else
        return false;
}
bool worseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}
bool FillReview(Review & rr)
{
    cout << "Zadejte nazev knihy (konec pro ukonceni): ";
    getline(cin,rr.title);
    if (rr.title == "konec")
        return false;
    cout << "Zadejte hodnoceni knihy: ";
    cin >> rr.rating;
    if (!cin)
        return false;
    cin.get();
    return true;
}
void ShowReview(const Review & rr)
{
    cout << rr.rating << "\t" << rr.title << endl;
}
```

Kompatibilita:

Starší implementace používají hlavičkový soubor `vector.h` namísto souboru `vector`. Ačkoli na pořadí vložených souborů by záležet nemělo, v `g++ 2.7.1` musel být hlavičkový soubor `string` uveden před hlavičkovými soubory knihovny STL. Funkce `getline()` v Microsoft Visual C++ 5.0 (poznámka překladatele: týká se i verze 6.0) obsahuje chybu, v jejímž důsledku se výstup následujícího řádku zobrazí až po dalším vstupu. V Microsoft Visual C++ 5.0 musíte kromě funkce `operator<()` definovat také funkci `operator==()`. Funkce `getline()` v Borland C++ Builder 1.0 požaduje jako parametr explicitní oddělovač.

Zde je ukázku běhu programu:

```
Zadejte nazev knihy (konec pro ukonceni): Kocka, ktera vas nauci zhubnout
Zadejte hodnoceni: 8
Zadejte nazev knihy (konec pro ukonceni): Psi z Dharmy
Zadejte hodnoceni: 7
Zadejte nazev knihy (konec pro ukonceni): Straspytlove z Wonku
Zadejte hodnoceni: 4
Zadejte nazev knihy (konec pro ukonceni): Sbohem a konec
Zadejte hodnoceni: 8
Zadejte nazev knihy (konec pro ukonceni): konec
Dekuji. Zadali jste nasledujici knihy:
Hodnoceni Kniha
8         Kocka, ktera vas nauci zhubnout
6         Psi z Dharmy
3         Straspytlove z Wonku
7         Sbohem a konec
Trideni dle nazvu
Hodnoceni Kniha
7         Sbohem a konec
8         Kocka, ktera vas nauci zhubnout
6         Psi z Dharmy
3         Straspytlove z Wonku
Trideni dle hodnoceni
Hodnoceni Kniha
3         Straspytlove z Wonku
6         Psi z Dharmy
7         Sbohem a konec
8         Kocka, ktera vas nauci zhubnout
Po promichani:
Hodnoceni Kniha
7         Sbohem a konec
3         Straspytlove z Wonku
6         Psi z Dharmy
8         Kocka, ktera vas nauci zhubnout
```

Generické programování

Nyní již máte se standardní knihovnou šablon nějaké zkušenosti a podíváme se tedy na podstatu její filozofie. Knihovna STL je příkladem *generického programování*. Zatímco objektově orientované programování se soustředí na datový aspekt, generické programování se soustředí na algoritmus. Mezi hlavní věci, které mají oba přístupy společné, patří abstrakce a vytvoření znovupoužitelného kódu. Jejich filozofie se však naprosto liší.

Cílem generického programování je napsat kód nezávislý na datových typech. Nástrojem jazyka C++ pro vytváření generických programů jsou šablony. Umožňují vám definovat funkci nebo třídu podle generického typu. Knihovna STL jde dále a nabízí generickou reprezentaci algoritmů. Šablony to také umožňují, ale ne bez pečlivého a svědomitého návrhu. Abychom si ukázali, jak tato směsice šablon a návrhu funguje, podívejme se, k čemu jsou potřeba iterátory.

K čemu jsou potřeba iterátory

Klíčem k porozumění knihovně STL je zřejmě pochopení iterátorů. Stejně jako šablony činí algoritmy nezávislými na typu uložených dat, činí iterátory algoritmy nezávislými na typu použitého kontejneru. Představují tedy podstatnou komponentu generického přístupu knihovny STL.

Abychom si potřebu iterátorů ozřejmili, podíváme se, jak byste mohli implementovat vyhledávací funkci pro dva různé typy reprezentací dat a potom si ukážeme, jak byste tento postup mohli generalizovat. Nejdříve uvažujte funkci, která v obyčejném poli typu double hledá určitou hodnotu. Mohli byste ji napsat takto:

```
double * find_ar(double * ar, int n, const double & val)
{
   for (int i = 0; i < n; i++;)
          if (ar[i] == val)
                 return &ar[i];
   return 0;
}
```

Jestliže funkce tuto hodnotu v poli najde, vrátí adresu prvku pole, kde se tato hodnota našla; v opačném případě vrátí nulový ukazatel. Pro pohyb v poli se používá zápis s indexem. Pomocí šablony byste mohli generalizovat pole libovolného typu s operátorem ==. Tento algoritmus je však stále vázán na konkrétní datovou strukturu, kterou je pole.

Podívejme se na prohledávání jiné datové struktury – spojového seznamu. (V kapitole 11 se spojový seznam použil pro implementaci třídy Queue.) Seznam sestává ze spojených struktur Node:

```
struct Node
{
   double item;
   Node * p_next;
};
```

Předpokládejme, že máte ukazatel na první uzel seznamu. Ukazatel p_next v každém uzlu ukazuje na uzel následující. Ukazatel v posledním uzlu je nastaven na nulu. Funkci find_ll() byste mohli napsat takto:

```
Node* find_ll(Node * head, const double & val)
{
   Node * start;
   for (start = head; start != 0; start = start->next)
          if (start->item == val)
                 return start;
   return 0;
}
```

Opět byste mohli pomocí šablony generalizovat tuto funkci na seznamy libovolného datového typu podporující operátor ==. Tento algoritmus je však stále vázán na konkrétní datovou strukturu, kterou je spojový seznam.

Jestliže uvážíte implementační detaily, používají obě funkce různé algoritmy: jedna prochází seznam položek pomocí indexů pole, zatímco druhá nastavuje hodnotu `start` pomocí výrazu `start->next`. Celkem vzato jsou však oba algoritmy stejné: porovnávají hodnotu postupně s jednotlivými hodnotami v kontejneru, dokud nenajdou hodnotu odpovídající.

Cílem generického programování by měla být jediná funkce pracující s poli, spojovými seznamy nebo jakýmkoli jiným kontejnerovým typem. To znamená, že by měla být nejen nezávislá na datovém typu uloženém v kontejneru, ale měla by být nezávislá i na datové struktuře samotného kontejneru. Šablony nabízejí generickou reprezentaci pro datový typ uložený v kontejneru. Potřeba je také generická reprezentace procesu procházení hodnotami v kontejneru. Tuto generalizovanou reprezentaci představuje iterátor.

Jaké vlastnosti by měl mít iterátor implementující vyhledávací funkci? Zde je jejich seznam:

- Abyste získali přístup k hodnotě, měli byste mít možnost dereferencovat iterátor, který na ni odkazuje. Pokud tedy `p` bude iterátor, definujte výraz `*p`.
- Měli byste mít možnost iterátory přiřazovat. Jestliže `p` a `q` budou iterátory, definujte výraz `p = q`.
- Měli byste mít možnost iterátory porovnávat. Jestliže `p` a `q` budou iterátory, definujte výrazy `p == q` a `p != q`.
- Měli byste mít možnost projít pomocí iterátoru všechny prvky kontejneru. Pro iterátor `p` toho dosáhnete definováním výrazů `++p` a `p++`.

Iterátor by toho mohl dělat více, ale nic dalšího nepotřebujete, alespoň ne pro účely vyhledávací funkce. Ve skutečnosti knihovna STL definuje několik úrovní iterátorů s většími schopnostmi, a později se k tomuto tématu vrátíme. Všimněte si, že požadavky kladené na iterátor splňuje obyčejný ukazatel. Funkci `find_arr()` byste tedy mohli přepsat takto:

```
typedef double * iterator;
iterator find_ar(iterator ar, int n, const double & val)
{
   for (int i = 0; i < n; i++, ar++)
            if (*ar == val)
                   return ar;
   return 0;
}
```

Pro funkci `find_ll()` můžete definovat třídu iterator, ve které budou definovány operátory * a ++:

```
struct Node
{
   double item;
   Node * p_next;
};
class iterator
{
   Node * pt;
public:
   iterator() : pt(0) {}
```

```
    iterator (Node * pn) : pt(pn) {}
    double operator*() {return pt->item;}
    iterator& operator++()          // pro ++it
    {
            pt = pt->next;
            return *this;
    }
    iterator operator++(int)                // pro it++
    {
            iterator tmp = *this;
            pt = pt->next;
            return tmp;
    }
// … operator==(), operator!=(), atd.
};
```

(Pro rozlišení mezi prefixovou a postfixovou verzí operátoru ++ přijal jazyk C++ konvenci, při které funkce `operator++()` označuje prefixovou verzi a funkce `operator++(int)` verzi postfixovou; parametr se nikdy nepoužívá a proto nemusí mít název.)

Hlavní myšlenku zde nepředstavuje detailní definice třídy `iterator`, ale skutečnost, že pomocí této třídy lze druhou funkci zapsat následovně:

```
iterator find_ll(iterator head, const double & val)
{
  iterator start;
  for (start = head; start != 0; ++start)
        if (*start == val)
              return start;
  return 0;
}
```

Tato funkce je téměř stejná jako funkce `find_ar()`. Obě funkce se liší pouze ve způsobu, kterým určují, zda již dosáhly konce prohledávaných hodnot. Funkce `find_ar()` používá prvek počítání, zatímco funkce `find_ll()` používá nulovou hodnotu uloženou v posledním uzlu. Pokud tento rozdíl odstraníte, vytvoříte dvě totožné funkce. Mohli byste například vyžadovat, aby pole i spojový seznam měly za posledním oficiálním prvkem ještě jeden prvek pomocný. To znamená, aby měly prvek za koncem a prohledávání by skončilo, jakmile by iterátor dosáhl pozice tohoto prvku. V tom případě by obě funkce `find_ar()` a `find_ll()` poznaly konec dat stejným způsobem a jejich algoritmy by byly totožné. Všimněte si, že požadavek na prvek následující bezprostředně za koncem se přesouvá z iterátorů na kontejnerovou třídu.

Knihovna STL postupuje právě podle nastíněného postupu. Nejdříve je v každé kontejnerové třídě (`vector`, `list`, `deque`, atd.) definován typ iterátoru vhodný pro tuto třídu. Pro jednu třídu by tímto iterátorem mohl být ukazatel, pro jinou objekt. Ať už bude implementace jakákoli, iterátor poskytne potřebné operace, například * a ++. (Některé třídy mohou potřebovat více operací než třídy jiné.) Dále bude mít každá kontejnerová třída indikátor představující prvek bezprostředně následující za koncem kontejneru, což bude hodnota přiřazená iterátoru inkrementovanému po poslední hodnotě kontejneru. Použijte metody `begin()` a `end()` vracející iterátory na první prvek v kontejneru a na pozici za

koncem kontejneru. Pomocí operace ++ projděte iterátorem od prvního prvku za konec kontejneru a přitom navštivte každý prvek kontejneru.

Pro použití kontejnerové třídy nemusíte znát způsob implementace iterátorů nebo pozice za koncem. Stačí vědět, že třída iterátory obsahuje, že funkce `begin()` vrací iterátor na první prvek a funkce `end()` vrací iterátor na místo za koncem. Předpokládejme například, že chcete vytisknout hodnoty z objektu šablony vector<double>. V tom případě napište tento kód:

```
vector<double>::iterator pr;
for (pr = scores.begin(); pr! = scores.end(); pr++)
  cout << *pr << endl;
```

Zde řádek

```
vector<double>::iterator pr;
```

identifikuje `pr` jako typ iterátor, definovaný pro třídu `vector<double>`. Pokud byste pro ukládání hodnot scores použili šablonovou třídu `list<double>`, mohli byste napsat tento kód:

```
list<double>::iterator pr;
for (pr = scores.begin(); pr! = scores.end(); pr++)
  cout << *pr << endl;
```

Jedinou změnou je typ deklarovaný pro `pr`. Díky definování vhodných iterátorů pro každou třídu a díky jednotnému způsobu návrhu tříd umožňuje knihovna STL napsat stejný kód pro kontejnery s naprosto odlišnými vnitřními reprezentacemi.

Ve skutečnosti je z hlediska stylu lépe vyhnout se přímému použití iterátorů; pokud je to možné, používejte místo nich nějakou funkci knihovny STL, například `for_each()`, která detaily obstará za vás.

Shrňme si způsob řešení používaný knihovnou STL. Nejdříve vytvoříte algoritmus pro práci s kontejnerem. Napište ho co možná nejobecněji, aby byl nezávislý na typu dat a typu kontejneru. Aby obecný algoritmus fungoval ve specifických případech, definujte iterátory odpovídající potřebám algoritmu, a požadavky vložte do návrhu kontejneru. To znamená, že základní vlastnosti iterátorů a vlastnosti kontejneru budou vycházet z požadavků vložených do nich algoritmem.

Druhy iterátorů

Různé algoritmy mají na iterátory různé požadavky. Pro vyhledávací algoritmus musí být například definován operátor ++, aby se iterátor mohl v kontejneru pohybovat. Je potřeba přístup pro čtení dat, nikoli však pro jejich zápis. (Iterátor data pouze prohlíží, ale nemění je). Třídící algoritmus však vyžaduje přímý přístup, aby bylo možné vzájemně prohodit dva nesousední prvky. Jestliže `iter` je iterátor, pak přímý přístup získáte definováním operátoru +, čímž budete moci používat výrazy typu `iter + 10`. Třídicí algoritmus musí mít možnost data číst i zapisovat.

V knihovně STL je definováno pět druhů iterátorů a algoritmy jsou popsány na základě potřeby těchto druhů. Pět druhů iterátorů tvoří *vstupní iterátor*, *výstupní iterátor*, *dopřed-*

ný iterátor, obousměrný iterátor a *iterátor přímého přístupu.* Například prototyp funkce `find()` vypadá takto:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Tímto způsobem vám prototyp oznamuje, že algoritmus vyžaduje vstupní iterátor. Podobně prototyp

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

oznamuje, že třídicí algoritmus vyžaduje iterátor přímého přístupu.

Všech pět druhů iterátorů lze dereferencovat (to znamená, že je pro ně definován operátor *) a porovnávat (pomocí operátorů == a !=, které je možné přetížit). Jestliže jsou dva iterátory vyhodnoceny jako stejné, pak byste dereferencováním jednoho měli získat stejnou hodnotu jako při dereferencování druhého. Jestliže tedy platí

```
iter1 == iter2;
```

platí i

```
*iter1 == *iter2;
```

Uvedené vlastnosti samozřejmě platí pro vestavěné operátory a ukazatele, takže tyto požadavky slouží jako vodítka, co musíte udělat, když tyto operátory přetěžujete pro třídu iterátoru. Nyní se podíváme na další vlastnosti iterátoru.

Vstupní iterátor

Termín „vstupní" vyjadřuje hledisko programu. To znamená, že za vstup se považují informace přicházející do programu z kontejneru, stejně jako se za vstup považují informace, které do programu přicházejí z klávesnice. Vstupní iterátor je tedy ten, který program může použít pro načtení hodnot z kontejneru. Dereferencování vstupního iterátoru musí programu umožnit hodnotu přečíst, ale nemusí mu povolit hodnotu změnit. Algoritmy vyžadující vstupní iterátor jsou tedy takové, které hodnoty v kontejneru nemění.

Vstupní iterátor musí umožnit přístup ke všem hodnotám v kontejneru. Činí tak podporou operátoru ++, jak v prefixové tak v postfixové formě. Jestliže vstupní iterátor nastavíte na první prvek kontejneru a budete ho inkrementovat až do místa za koncem kontejneru, bude během své cesty postupně ukazovat na každou položku kontejneru. Mimochodem není zaručeno, že při dalším průchodu kontejnerem se vstupní iterátor bude po hodnotách pohybovat ve stejném pořadí. Také není zaručeno, že po inkrementování iterátoru bude ještě možné dereferencovat předchozí hodnotu. Každý algoritmus založený na vstupním iterátoru by tedy měl být určen pro jeden průchod a neměl by se spoléhat na hodnoty získané z průchodu předchozího ani na dřívější hodnoty získané ze stejného průchodu.

Všimněte si, že vstupní iterátor je jednocestný; můžete ho inkrementovat, ale nemůžete ho vrátit.

Výstupní iterátor

Zde termín „výstupní" označuje, že iterátor se používá pro přenos informací z programu do kontejneru. Výstupní iterátor se podobá iterátoru vstupnímu s tím rozdílem, že dereferencování zaručí, že program může hodnotu kontejneru změnit, ale nemůže ji přečíst. Pokud vám možnost zapisovat data bez jejich čtení připadá divná, uvědomte si, že tuto vlastnost má také výstup poslaný na zobrazovací zařízení; objekt `cout` může proud znaků poslaných na zobrazovací zařízení upravit, ale číst z obrazovky nemůže. Knihovna STL je natolik obecná, že její kontejnery mohou představovat výstupní zařízení, takže při používání kontejnerů se můžete dostat do stejné situace. Pokud také nějaký algoritmus obsah kontejneru upraví (například generováním nových hodnot, které budou uloženy), aniž by ho musel číst, není důvod používat iterátor pro čtení.

Stručně řečeno, vstupní iterátor můžete použít pro jednoprůchodové algoritmy určené pro čtení, zatímco výstupní iterátor pro jednoprůchodové algoritmy určené pro zápis.

Dopředný iterátor

Dopředný iterátor používá stejně jako vstupní a výstupní iterátory pro pohyb v kontejneru pouze operátory ++. Může se tedy v kontejneru pohybovat pouze dopředu, a to vždy o jeden prvek. Na rozdíl od vstupního a výstupního kontejneru však prochází sekvencí hodnot vždy ve stejném pořadí. Také po jeho inkrementování máte stále možnost dereferencovat iterátor předchozí a získat stejnou hodnotu. Díky těmto vlastnostem je možné psát algoritmy pro více průchodů.

Dopředný iterátor umožňuje data číst i upravovat, nebo pouze číst:

```
int * pirw;              // iterátor pro čtení a zápis
const int * pir;  // iterátor pouze pro čtení
```

Obousměrný iterátor

Předpokládejme algoritmus, který musí umět projít kontejner v obou směrech. Inverzní funkce by například mohla prohodit první a poslední prvek, inkrementovat ukazatel na první prvek, dekrementovat ukazatel na druhý prvek a tento proces zopakovat. Obousměrný iterátor má všechny vlastnosti iterátoru dopředného a navíc podporuje dva operátory dekrementování (prefixový a postfixový).

Iterátor přímého přístupu

Některé algoritmy, například třídění nebo binární vyhledávání, vyžadují možnost přistupovat přímo na libovolný prvek v kontejneru. Takový přístup se nazývá přímý a vyžaduje iterátor přímého přístupu. Tento iterátor má všechny vlastnosti iterátoru obousměrného a navíc obsahuje operace (jako sčítání ukazatelů), které přímý přístup podporují, a relační operátory pro uspořádání prvků. Seznam vlastností, které má iterátor přímého přístupu a které neobsahuje iterátor obousměrný, je uveden v tabulce 15.3. V této tabulce představuje `X` libovolný typ iterátoru, `T` je typ, na který ukazuje, `a` a `b` jsou hodnoty iterátoru, `n` je celé číslo a `r` je libovolná iterátorová proměnná nebo reference.

Tabulka 15.3 Operace, které obsahuje iterátor přímého přístupu.

Výraz	Komentář
a + n	Ukazuje na n-tý prvek za prvkem, na který ukazuje a
n + a	Stejné jako a + n
a - n	Ukazuje na n-tý prvek před prvkem, na který ukazuje a
r += n	Ekvivalent výrazu r = r + n
r -= n	Ekvivalent výrazu r = r - n
a[n]	Ekvivalent výrazu *(a + n)
b - a	Výsledná hodnota n odpovídá výrazu b = a + n
a < b	Platí, pokud b - a > 0
a > b	Platí, pokud b < a
a >= b	Platí, pokud !(a < b)
a <= b	Platí, pokud !(a > b)

Výrazy typu a + n platí pouze v případě, že a i a + n leží v rozsahu kontejneru (včetně místa za koncem).

Hierarchie iterátorů

Pravděpodobně jste si všimli, že druhy iterátorů tvoří hierarchii. Dopředný iterátor má všechny schopnosti vstupního a výstupního iterátoru a navíc svoje vlastní. Obousměrný iterátor má všechny schopnosti iterátoru dopředného a navíc svoje vlastní. A iterátor přímého přístupu má všechny schopnosti iterátoru obousměrného a navíc svoje vlastní. Hlavní schopnosti iterátorů jsou shrnuty v tabulce 15.4. Zde i představuje iterátor a n celé číslo.

Tabulka 15.4 Schopnosti interátorů

Schopnost	Vstupní	Výstupní	Dopředný	Obousměrný	Přímého přístupu
Dereferencování při čtení	ano	ne	ano	ano	ano
Dereferencování při zápisu	ne	ano	ano	ano	ano
Pevné a opakovatelné pořadí	ne	ne	ano	ano	ano
++i	ano	ano	ano	ano	ano
i++					
—i	ne	ne	ne	ano	ano
i—					
i[n]	ne	ne	ne	ne	ano
i + n	ne	ne	ne	ne	ano
i – n	ne	ne	ne	ne	ano
i += n	ne	ne	ne	ne	ano
i -= n	ne	ne	ne	ne	ano

Algoritmus napsaný podle určitého druhu operátoru může používat tento druh iterátoru nebo kterýkoli jiný druh, který má požadované schopnosti. Takže například kontejner s iterátorem přímého přístupu může používat algoritmus napsaný pro iterátor vstupní.

Proč existují všechny tyto druhy iterátorů? Smyslem je napsat algoritmus pomocí iterátoru s co nejmenšími požadavky, který bude možné použít pro co nejvíce kontejnerů. Takže funkci find() s obyčejným vstupním iterátorem bude možné použít pro každý kontej-

ner s hodnotami určenými pro čtení. Avšak funkci `sort()`, která vyžaduje iterátor přímého přístupu, bude možné použít pouze u kontejnerů, které tento druh iterátorů podporují.

Všimněte si, že různé druhy iterátoru nejsou definovanými typy; spíše charakterizují koncept. Jak jsme se zmínili dříve, je v každé kontejnerové třídě definován příkazem `typedef` název `iterator`. Třída `vector<int>` má tedy iterátory typu `vector<int>::iterator`. Ale dokumentace k této třídě by vás informovala, že iterátory třídy `vector` jsou iterátory přímého přístupu. Mohli byste tedy použít algoritmy založené na libovolném typu iterátoru, protože iterátor přímého přístupu má všechny schopnosti iterátorů. Podobně třída `list<int>` má iterátory typu `list<int>::iterator`. Knihovna STL implementuje obousměrný spojový seznam a používá tedy obousměrný iterátor. Nemůže tedy používat algoritmy založené na iterátorech přímého přístupu, ale může používat algoritmy založené na méně náročných iterátorech.

Koncepty, vylepšení a modely

Knihovna STL obsahuje několik prostředků, například druhy iterátoru, které v jazyce C++ nelze vyjádřit. To znamená, že ačkoli můžete například navrhnout třídu s vlastnostmi dopředného iterátoru, nemůžete dosáhnout toho, aby kompilátor použil nějaký algoritmus pouze při použití této třídy. Dopředný iterátor je totiž sada požadavků, není to typ. Tyto požadavky může uspokojit navržená iterátorová třída, ale také obyčejný ukazatel. Literatura o knihovně STL popisuje sadu požadavků slovem *koncept*. Existuje tedy koncept vstupního iterátoru, koncept dopředného iterátoru, a tak dále. Mimochodem, jestliže iterátory potřebujete například při navrhování kontejnerové třídy, vloží knihovna STL šablony standardních druhů iterátorů.

Koncepty mohou mít vztah dědičnosti. Například obousměrný iterátor dědí schopnosti iterátoru dopředného. Avšak mechanismus dědičnosti z jazyka C++ na iterátory použít nemůžete. Mohli byste například dopředný iterátor implementovat jako třídu a iterátor obousměrný jako obyčejný ukazatel. Podle jazyka C++ by tedy tento obousměrný iterátor byl vestavěným typem a nemohl by být odvozený od třídy. Z koncepčního hlediska však dědí. V některé literatuře o knihovně STL se pro tuto koncepční dědičnost používá termín *vylepšení* (refinement). Obousměrný iterátor je tedy vylepšením konceptu iterátoru dopředného.

Konkrétní implementace konceptu se nazývá *model*. Obyčejný ukazatel na typ `int` je tedy modelem konceptu iterátoru přímého přístupu a je také modelem iterátoru dopředného, neboť uspokojuje požadavky tohoto konceptu.

Ukazatel jako iterátor

Iterátory jsou generalizované ukazatele a ukazatel uspokojuje všechny požadavky iterátoru. Iterátory tvoří rozhraní algoritmů knihovny STL a ukazatelé jsou iterátory, takže algoritmy knihovny STL je mohou používat při práci s kontejnery, které součástí knihovny nejsou. Algoritmy knihovny STL můžete například použít při práci s poli. Předpokládejme, že `Receipts` je pole s hodnotami typu `double` a vy chcete tyto hodnoty setřídit vzestupně:

```
const int SIZE = 100;
double Receipts[SIZE];
```

Vzpomeňte si, že funkce `sort()` z knihovny STL má jako parametry iterátor ukazující na první prvek v kontejneru a iterátor ukazující na prvek následující bezprostředně za koncem kontejneru. `&Receipts[0]` (nebo jen `Receipts`) je adresa prvního prvku a `&Receipts[SIZE]` (nebo jen `Receipts + SIZE`) je adresa prvku za posledním prvkem pole. Voláním funkce

```
sort(Receipts, Receipts + SIZE);
```

tedy pole setřídíte. Mimochodem, C++ zaručuje, že výraz `Receipts + n` je definován, pokud se jeho výsledek nachází v daném poli nebo o jeden prvek za prvkem posledním.

Skutečnost, že ukazatelé jsou iterátory a že algoritmy jsou na iterátorech založené, umožňuje použít algoritmy z knihovny STL na obyčejná pole. Podobně můžete tyto algoritmy použít na vlastní datové formy za předpokladu, že dodáte vhodné iterátory (což mohou být ukazatele nebo objekty) a indikátory představující místo za koncem.

Funkce `copy()` a šablony `ostream_iterator` a `istream_iterator`

Knihovna STL obsahuje některé předdefinované iterátory. Abychom si objasnili proč, vytvoříme si určité prostředí. Existuje algoritmus (`copy()`) pro kopírování dat z jednoho kontejneru do kontejneru druhého. Tento algoritmus je vyjádřen iterátory, takže dokáže kopírovat z jednoho druhu kontejneru do druhého nebo dokonce i z pole či do pole, protože ukazatele do tohoto pole můžete použít jako iterátory. Následující kód například zkopíruje pole do vektoru:

```
int casts[10] = {6, 7, 2, 9, 4, 11, 8, 7, 10, 5};
vector<int> dice[10];
copy(casts, casts + 10, dice.begin());   // zkopíruje pole do vektoru
```

Prvními dvěma parametry funkce `copy()` jsou iterátory označující velikost kopírovaného rozsahu a poslední parametr určuje místo, kam se zkopíruje první položka. Prvními dvěma parametry musí být vstupní iterátory (nebo lepší) a posledním parametrem musí být iterátor výstupní (nebo lepší). Funkce `copy()` přepíše existující data v cílovém kontejneru a tento kontejner musí být dostatečně veliký, aby se do něho kopírované prvky vešly. Funkci `copy()` tedy nemůžete použít pro vložení dat do prázdného vektoru, pokud se ovšem neuchýlíte k triku, který odhalíme později.

Nyní předpokládejme, že chcete informace zkopírovat na zobrazovací zařízení. Pokud budete mít iterátor představující výstupní proud, mohli byste použít funkci `copy()`. Knihovna STL takový iterátor poskytuje pomocí šablony `ostream_iterator`. Podle terminologie knihovny STL je tato šablona modelem konceptu výstupního iterátoru. Je také příkladem adaptéru, třídy nebo funkce, který konvertuje některá jiná rozhraní na rozhraní používané knihovnou STL. Iterátor tohoto typu můžete vytvořit vložením hlavičkového souboru iterator (dříve `iterator.h`) a deklarací:

```
#include <iterator>
...
ostream_iterator<int, char> out_iter(cout, " ");
```

Iterátor `out_iter` se nyní stane rozhraním, které vám umožní zobrazit informace pomocí objektu `cout`. První parametr šablony (v tomto případě `int`) označuje typ dat, který bude do výstupního proudu poslán. Druhý parametr šablony (v tomto případě `char`) označuje typ znaků, který výstupní proud použije. (Další možnou hodnotou byl typ `wchar_t`.) Prv-

ní parametr konstruktoru (v tomto případě cout) identifikuje použitý výstupní proud. Mohl by jím být také proud používaný pro výstupní soubor, což bude probráno v kapitole 16. Poslední parametr, znakový řetězec, je oddělovač, který se zobrazí za každou položkou poslanou do výstupního proudu.

Upozornění

Starší implementace používají pro šablonu ostream_iterator pouze první parametr:

```
ostream_iterator<int> out_iter(cout, " ");// starší implementace
```

Iterátor byste mohli použít takto:

```
*out_iter++ = 15; // funguje jako cout << 15 << " ";
```

Pro normální ukazatel by tento příkaz znamenal přiřazení hodnoty 15 do místa, na které ukazatel ukazuje, a následnou inkrementaci ukazatele. Pro šablonu ostream_iterator však tento příkaz znamená odeslání hodnoty 15 následované řetězcem tvořeným mezerou do výstupního proudu, který spravuje objekt cout. Připravte se na následující výstupní operaci. Iterátor můžete s funkcí copy() použít následujícím způsobem:

```
copy(dice.begin(), dice.end(), out_iter); // zkopíruje vektor do výstupního
                                          // proudu
```

Tento příkaz znamená, že se do výstupního proudu zkopíruje celý obsah kontejneru dice, že se tedy zobrazí celý obsah kontejneru.

Vytvoření iterátoru s názvem můžete přeskočit a místo toho vytvořit iterátor anonymní. To znamená, že adaptér můžete použít takto:

```
copy(dice.begin(), dice.end(),ostream_iterator<int, char>(cout, " ") );
```

Podobným způsobem definuje hlavičkový soubor iterator šablonu istream_iterator pro přizpůsobení vstupu istream rozhraní iterátoru. Tato šablona je modelem vstupního iterátoru. Pro definování vstupního rozsahu funkce copy() můžete použít dva objekty šablony istream_iterator:

```
copy(istream_iterator<int, char>(cin), istream_iterator<int, char>,
  dice.begin());
```

Stejně jako šablona ostream_iterator používá i šablona istream_iterator dva parametry. První parametr označuje typ čtených dat a druhý typ znaků, který vstupní proud používá. Použití parametru cin znamená, že se použije vstupní proud spravovaný objektem cin. Vynechání tohoto parametru v konstruktoru označuje neúspěšný vstup, takže výše uvedený kód znamená, že se bude číst ze vstupního proudu až do konce souboru, nebo dokud nedojde k neshodě typů či k nějaké jiné vstupní chybě.

Ostatní užitečné iterátory

Hlavičkový soubor iterator nabízí kromě šablon ostream_iterator a istream_iterator některé další předdefinované typy iterátorů určené pro speciální účely. Jsou jimi iterátory reverse_iterator, back_insert_iterator, front_insert_iterator a insert_iterator.

Nejdříve se podíváme, co dělá iterátor reverse_iterator. Při inkrementování ho vlastně dekrementujeme. Ale proč nedekrementovat obyčejný ukazatel? Hlavním důvodem je zjednodušit používání již existujících funkcí. Předpokládejme, že chcete zobrazit obsah kontejneru dice. Jak jste právě viděli, zkopírovat obsah do výstupního proudu můžete pomocí funkce copy() a šablony ostream_iterator:

```
ostream_iterator<int, char> out_iter(cout, " ");
copy(dice.begin(), dice.end(), out_iter); // zobrazí obsah kontejneru
```

Nyní předpokládejme, že chcete obsah vytisknout v opačném pořadí. (Možná provádíte studie, které se zabývají změnou toku času.) Existuje několik nefunkčních způsobů, ale těmi se zabývat nebudeme a raději se podíváme na ten, který funguje. Třída vector má členskou funkci rbegin(), která vrací inverzní iterátor na místo následující bezprostředně za koncem, a členskou funkci rend() vracející inverzní iterátor na první prvek. Protože při inkrementování inverzního iterátoru dochází k jeho dekrementaci, můžete použít tento příkaz:

```
copy(dice.rbegin(), dice.rend(), out_iter);   // zobrazí obsah kontejneru
                                              // v opačném pořadí
```

Tímto příkazem zobrazíte obsah kontejneru v opačném pořadí a nemusíte ani použít inverzní iterátor.

Pamatujte

Funkce rbegin() i funkce end() vracejí stejnou hodnotu (za koncem), ale různý typ iterátoru (reverse_iterator, iterator). Podobně vracejí stejnou hodnotu (iterátor na první prvek) ale jiný typ iterátoru funkce rend() a begin().

Inverzní ukazatele musí zaplatit speciální daň. Předpokládejme, že rp je inverzní ukazatel inicializovaný funkcí dice.rbegin(). Co by mělo vyjadřovat *rp? Protože funkce rbegin() vrací ukazatel na místo bezprostředně za koncem kontejneru, neměli byste tuto adresu dereferencovat. Podobně jestliže funkce rend() vlastně ukazuje na první prvek, ukončila by funkce copy() kopírování prvků o jedno místo dříve, protože koncové místo do oblasti nepatří. Inverzní ukazatele oba problémy řeší tak, že se nejdříve dekrementují a teprve pak dereferencují. To znamená, že výraz *rp dereferencuje hodnotu iterátoru, která bezprostředně předchází aktuální hodnotě *rp. Jestliže rp ukazuje na šestou pozici, je *rp hodnotou z pozice páté a tak dále. Program ve výpisu 15.7 demonstruje používání funkce copy(), iterátoru istream a inverzního iterátoru.

Výpis 15.7 copy.cpp

```
// copy.cpp - funkce copy() a iterátory
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
```

```
int main(){
        int casts[10] = {6, 7, 2, 9, 4, 11, 8, 7, 10, 5};
        vector<int> dice(10);
        // kopírování z pole do vektoru
        copy(casts, casts + 10, dice.begin());
        cout << "Hodte kostky!\n";
        // vytvoří iterátor pro výstupní proud
        ostream_iterator<int, char> out_iter(cout, " ");
        // kopírování z vektoru do výstupního proudu
      copy(dice.begin(), dice.end(), out_iter);
      cout << endl;
        cout <<"Implicitni pouziti inverzniho iteratoru.\n";
        copy(dice.rbegin(), dice.rend(), out_iter);
      cout << endl;
        cout <<"Explicitni pouziti inverzniho iteratoru.\n";
        vector<int>::reverse_iterator ri;
        for (ri = dice.rbegin(); ri != dice.rend(); ++ri)
            cout << *ri << ' ';
        cout << endl;

      return 0;
}
```

Kompatibilita:

Starší implementace mohou používat hlavičkové soubory `iterator.h` a `vector.h` a také iterátor `ostream_iterator<int>` místo `ostream_iterator<int, char>`.

Zde je výstup: Hodte kostky!

```
6 7 2 9 4  11 8 7 10 5
Implicitni pouziti inverzniho iteratoru.
5 10 7 8 11 4 9 2 7 6
Explicitni pouziti inverzniho iteratoru.
5 10 7 8 11 4 9 2 7 6
```

Jestliže si můžete vybrat mezi explicitním deklarováním iterátorů a použitím funkcí z knihovny STL, například předat nějaké funkci hodnotu vrácenou funkcí `rbegin()`, zvolte druhou možnost. Budete mít méně práce i méně příležitostí zakusit lidskou omylnost.

Ostatní tři iterátory (`back_insert_iterator`, `front_insert_iterator` a `insert_iterator`) také přispívají ke generalizování algoritmů knihovny STL. Mnoho funkcí knihovny STL odesílá své výsledky stejně jako funkce `copy()` na místo označené výstupním iterátorem. Vzpomeňte si, že příkaz

```
copy(casts, casts + 10, dice.begin());
```

zkopíruje hodnoty do paměti s počátkem na pozici `dice.begin()`. Tyto hodnoty přepíší předchozí obsah kontejneru `dice` a funkce předpokládá, že v kontejneru `dice` je dostatek místa, aby se sem hodnoty vešly. Funkce `copy()` tedy velikost cílového kontejneru podle množství zaslaných informací automaticky neupravuje. Program ve výpisu 15.7 tuto situ-

aci ošetřil tak, že deklaroval kontejner dice s 10 prvky, ale co když dopředu nevíte, jak velký by kontejner dice měl být nebo chcete spíše prvky do kontejneru přidávat než je přepisovat?

Tři operátory vkládání tyto problémy řeší tak, že z procesu kopírování udělají proces vkládání. Vkládání přidává nové prvky bez přepisování existujících dat a pomocí automatického přidělování paměti zajišťuje, že se nové informace do kontejneru vejdou. Iterátor back_insert_iterator vkládá položky na konec kontejneru, zatímco iterátor front_insert_iterator je vkládá na začátek. Konečně iterátor insert_iterator položky vkládá před určené místo, které představuje parametr konstruktoru tohoto iterátoru. Všechny tři jsou modely konceptu výstupního kontejneru.

Existují jistá omezení. Iterátor back_insert_iterator můžete použít pouze s takovými typy kontejnerů, které umožňují rychlé vkládání na konec. (Rychle znamená konstantní časový algoritmus; tímto konceptem se podrobněji zabývá část věnovaná kontejnerům.) Třída vector pro tyto účely vyhovuje. Iterátor front_insert_iterator lze použít pouze u takových typů kontejnerů, které umožňují konstantní časové vkládání na začátek. Zde třída vector nevyhovuje. Iterátor insert_iterator tato omezení nemá. Můžete ho tedy použít pro vkládání na začátek vektoru. Iterátor front_insert_iterator však tuto činnost pro podporované typy kontejnerů provádí rychleji.

Tip

Pomocí iterátoru vkládání můžete změnit algoritmus kopírující data na algoritmus, který bude data vkládat.

Tyto iterátory používají typ kontejneru jako parametr šablony a skutečný identifikátor kontejneru jako parametr konstruktoru. Chcete-li tedy vytvořit iterátor back_insert_iterator pro kontejner dice šablony vector<int>, napište tento příkaz:

```
back_insert_iterator<vector<int> > back_iter(dice);
```

Deklarace iterátoru front_insert_iterator má stejný tvar. Deklarace iterátoru insert_iterator má v konstruktoru navíc parametr identifikující místo vkládání:

```
insert_iterator<vector<int> > back_iter(dice, dice.begin());
```

Použití dvou z těchto iterátorů předvádí program ve výpisu 15.8.

Výpis 15.8 inserts.cpp

```
// inserts.cpp - funkce copy() a iterátory vkládani
#include <iostream>
#include <string>
#include <iterator>
#include <vector>
using namespace std;

int main(){
    string s1[4] = {"pekna", "ryba", "osud", "mody"};
    string s2[2] = {"pilni", "netopyri"};
```

```
    string s3[2] = {"hloupi", "zpevaci"};
    vector<string> words(4);
    copy(s1, s1 + 4, words.begin());
    ostream_iterator<string, char> out(cout, " ");
    copy (words.begin(), words.end(), out);
    cout << endl;
// vytvoří anonymní objekt back_insert_iterator
    copy(s2, s2 + 2, back_insert_iterator<vector<string> >(words));
    copy (words.begin(), words.end(), out);
    cout << endl;

// vytvoří anonymní objekt insert_iterator
    copy(s3, s3 + 2, insert_iterator<vector<string> >(words, words.be-
gin()));
    copy (words.begin(), words.end(), out);
    cout << endl;
  return 0;
}
```

Kompatibilita:

Starší verze mohou používat hlavičkové soubory `list.h` a `iterator.h` a také iterátor `ostream_iterator<int>` místo iterátoru `ostream_iterator<int, char>`.

Zde je výstup:

```
pekna ryba osud mody
pekna ryba osud mody pilni netopyri
hloupi zpevaci pekna ryba osud mody pilni netopyri
```

Pokud vás již všechny rozmanitosti iterátorů přemáhají, mějte na paměti, že díky jejich používání se s nimi obeznámíte. Také si uvědomte, že tyto předdefinované operátory rozšiřují všeobecnost algoritmů knihovny STL. Funkce `copy()` tedy dokáže nejen kopírovat informace z jednoho kontejneru do druhého, ale umí také kopírovat informace z kontejneru do výstupního proudu a ze vstupního proudu do kontejneru. A můžete ji rovněž použít pro vkládání prvků do dalšího kontejneru. Vystačíte tedy s jednou funkcí, která zastane práci mnoha jiných. A protože je funkce `copy()` jen jednou z několika funkcí knihovny STL, které používají výstupní iterátor, znásobují také tyto předdefinované iterátory schopnosti oněch funkcí.

Druhy kontejnerů

Knihovna STL obsahuje koncepty kontejnerů i kontejnerové typy. Koncepty jsou obecné kategorie s názvy jako kontejner, sekvenční kontejner, asociativní kontejner a tak dále. Kontejnerové typy jsou šablony, s jejichž pomocí můžete vytvářet určité kontejnerové objekty. Jedenáct kontejnerových typů tvoří `deque`, `list`, `queue`, `priority_queue`, `stack`, `vector`, `map`, `multimap`, `set`, `multiset` a `bitset`. (`Bitset`, kontejner pro práci s daty na bitové úrovni, v této kapitole probírat nebudeme.) Protože koncepty kategorizují typy, začneme jimi.

Koncept kontejneru

Pro základní koncept kontejneru odpovídající typ neexistuje, koncept však popisuje prvky společné všem kontejnerovým třídám. Jedná se o takovou koncepční abstraktní základní třídu – koncepční, protože kontejnerová třída ve skutečnosti mechanismus dědičnosti nepoužívá. Jinak řečeno, koncept kontejneru stanoví sadu požadavků, které musí splňovat všechny kontejnerové třídy knihovny STL.

Kontejner je objekt, který ukládá objekty jiné. Ty jsou všechny jednoduchého typu. Uloženými objekty mohou být objekty ve smyslu objektově orientovaného programování nebo jimi mohou být hodnoty vestavěných typů. Data uložená v kontejneru jsou jeho *vlastnictvím*. To znamená, že se zánikem kontejneru zaniknou i data v něm uložená. (Pokud jsou však těmito daty ukazatele, nemusí data, na která ukazují, zaniknout.)

Do kontejneru nemůžete uložit jakýkoli druh objektu. Tento typ musí být možné zkopírovat a přiřadit. Základní typy tyto požadavky splňují a stejně tak typy tříd, pokud kopírovací konstruktor nebo přiřazovací operátor nejsou ve třídě definovány jako soukromé nebo chráněné.

Základní kontejner nezaručuje, že prvky budou uloženy v určitém pořadí nebo že se jejich pořadí nezmění, ale vylepšené koncepty mohou takovou záruku dodat. Všechny kontejnery mají určité vlastnosti a poskytují určité operace. Některé společné vlastnosti jsou shrnuty v tabulce 15.5. Zde `X` představuje typ kontejneru (například `vector`), `T` typ uloženého objektu, `a` a `b` jsou hodnoty typu `X` a `u` představuje identifikátor typu `X`.

Tabulka 15.5 Některé vlastnosti základního kontejneru.

Výraz	Návratový typ	Komentář	Složitost
X::iterátor	Typ iterátoru ukazující na `T`	Každá kategorie iterátoru kromě výstupního iterátoru	Kompilační čas
X::typ_hodnoty	`T`	Typ pro `T`	Kompilační čas
X u;		Vytvoří kontejner s nulovou velikostí a s názvem `u`	Konstantní čas
X();		Vytvoří anonymní kontejner s nulovou velikostí	Konstantní čas
X u(a);		Kopírovací konstruktor	Lineární čas
X u = a;		Stejný účinek jako `X u(a);`	
(&a)->~X();	void	Použije destruktor na každý prvek kontejneru	Lineární čas
a.begin()	iterátor	Vrátí iterátor na první prvek kontejneru	Konstantní čas
a.end()	iterátor	Vrátí iterátor, který je hodnotou prvku následujícího bezprostředně za koncem kontejneru	Konstantní čas
a.size()	unsigned	Vrátí počet prvků, odpovídá a.end() a a.begin()	Konstantní čas
a.swap(b)	void	Vymění obsah kontejnerů `a` a `b`	Konstantní čas
a == b	konvertibilní na typ `bool`	Platí, pokud mají `a` a `b` stejnou velikost a každý prvek v `a` je roven odpovídajícímu prvku v `b`	Lineární čas
a != b	konvertibilní na typ `bool`	totéž jako `!(a == b)`	Lineární čas

Sloupec Složitost popisuje čas potřebný k provedení operace. V této tabulce jsou tři možnosti seřazené od nejrychlejší k nejpomalejší:

- Kompilační čas
- Konstantní čas
- Lineární čas

Pokud je složitost označena jako kompilační čas, provede se činnost během kompilace a nepoužije se žádný prováděcí čas. Složitost označená jako konstantní čas znamená, že se operace uskuteční za běhu programu, ale není závislá na počtu prvků v objektu. Složitost označená jako lineární čas znamená, že čas je úměrný počtu prvků. Jestliže tedy jsou `a` a `b` kontejnery, má porovnání `a == b` lineární složitost, protože operace porovnání se musí použít na každý prvek kontejneru. Jedná se vlastně o nejhorší případ. Pokud mají dva kontejnery rozdílnou velikost, nemusí se žádná individuální porovnání provádět.

Složitost v čase konstantním a v čase lineárním

Představte si dlouhou, úzkou krabici, zaplněnou velikými balíky, uspořádanými v řadě a předpokládejte, že je krabice otevřena jen na jednom konci. Mým úkolem je vyložit balík na otevřeném konci. Jedná se o úkol s konstantním časem. Nezáleží na tom, je-li za vyloženým balíkem balíků 10 nebo 1 000.

Nyní předpokládejte, že mým úkolem je vytáhnout balík na uzavřeném konci. Toto je úkol v lineárním času. Jestliže celkový počet balíků bude 10, musím jich vyložit 10, abych se dostal na konec. Bude-li jich 100, musím jich vyložit 100. Za předpokladu, že jsem neúnavný dělník vytahující jeden balík za určitou dobu, potrvá vykládání desetkrát déle.

Nyní předpokládejte, že je mým úkolem vytáhnout náhodně určený balík. Může se stát, že tímto balíkem bude hned první balík po ruce. Průměrně však počet balíků, který musím přebrat, je stále úměrný počtu balíků v kontejneru, takže složitost úkolu odpovídá lineárnímu času.

Pokud byste tuto krabici nahradili podobnou krabicí otevřenou po stranách, změnila by se složitost úkolu na čas konstantní, neboť v tom případě bych mohl jít k požadovanému balíku přímo a vytáhnout ho, aniž bych přebíral ostatní balíky.

Myšlenka časové složitosti popisuje účinek velikosti kontejneru na čas provedení, ale ostatní faktory ignoruje. Pokud nějaký superhrdina dokáže vyložit balíky z krabice otevřené na jednom konci stokrát rychleji než já, bude mít provedený úkol stále složitost v lineárním čase. V případě tohoto superhrdiny by lineární čas provedení při zavřené krabici (s otevřeným koncem) byl rychlejší než můj konstantní čas provedení při krabici otevřené, pokud by balíků v krabici nebylo příliš mnoho.

Požadavky týkající se složitosti jsou pro knihovnu STL charakteristické. Zatímco detaily implementace je možné skrýt, specifikace provedení by měly být veřejné, abyste věděli, jak je určitá operace počítačově náročná.

Sekvence

Koncept základního kontejneru lze vylepšit přidáním požadavků. Důležitým vylepšením je sekvence, neboť šest kontejnerových typů z knihovny STL (deque, list, queue, priority_queue, stack a vector) je sekvencemi. (Vzpomeňte si, že fronta umožňuje přidávat prvky na konec a odebírat je ze začátku. Obousměrná fronta, představovaná kontejnerem deque, umožňuje prvky přidávat a odebírat na obou koncích.) Koncept sekvence přidává požadavek, že iterátorem musí být minimálně iterátor dopředný. Tento iterátor zaručuje, že prvky budou uspořádány v určitém pořadí, které se během dvou iteračních cyklů nezmění.

Tato sekvence také vyžaduje, aby prvky byly uspořádány v přísně lineárním pořadí. To znamená, že existuje jeden první prvek a jeden prvek poslední a každý prvek kromě prvního a posledního má jeden prvek, který ho bezprostředně předchází, a jeden prvek, který ho bezprostředně následuje. Příkladem sekvence je pole a spojový seznam, zatímco struktura s větvemi (kde každý uzel ukazuje na dva uzly dceřiné) sekvencí není.

Protože prvky v sekvenci mají stanovené pořadí, jsou možné takové operace jako vkládání hodnot do určitého místa nebo vymazání určité oblasti. Tyto a jiné operace vyžadované po sekvenci jsou uvedeny v tabulce 15.6. V tabulce je použita stejná notace jako v tabulce 15.5 a přidány jsou symboly t vyjadřující hodnotu typu T (tedy hodnotu uloženou v kontejneru), n je celé číslo a p, q, i a j představují iterátory.

Tabulka 15.6 Požadavky kladené na sekvenci.

Výraz	Návratový typ	Komentář
X a(n, t);		Deklaruje sekvenci a s n kopiemi hodnot t
X(n, t)		Vytvoří anonymní sekvenci s n kopiemi hodnot t
X a(i, j)		Deklaruje sekvenci a inicializovanou obsahem z rozsahu [i, j)
X(i, j)		Vytvoří anonymní sekvenci inicializovanou obsahem z rozsahu [i, j)
a.insert(p, t)	iterátor	Vloží kopii t před p
a.insert(p, t, n)	void	Vloží n kopií t před p
a.insert(p, i, j)	void	Vloží před p kopie prvků z rozsahu [i, j)
a.erase(p)	iterátor	Vymaže prvek, na který ukazuje p
a.erase(p, q)	iterátor	Vymaže prvky z rozsahu [p, q)
a.clear()	void	Stejný účinek jako erase(begin(), end())

Protože šablonové třídy deque, list, queue, priority_queue, stack a vector jsou všechny modelem konceptu sekvence, podporují všechny operátory z tabulky 15.6. Kromě toho existují operace, které jsou dostupné některým z těchto pěti modelů. Kde je to povoleno, mají tyto operace složitost odpovídající konstantnímu času. Seznam těchto doplňkových operací je v tabulce 15.7.

Tabulka 15.7 Nepovinné požadavky kladené na sekvenci.

Výraz	Návratový typ	Význam	Kontejner
a.front()	T&	*a.begin()	vector, list, deque
a.back(0	T&	*—a.end()	vector, list, deque
a.push_front(t)	void	a.insert(a.begin(), t)	vector, deque
a.push_back(t)	void	a.insert(a.end(), t)	vector, list, deque
a.pop_front(t)	void	a.erase(a.begin())	vector, deque
a.pop_back(t)	void	a.erase(—a.end())	vector, list, deque
a[n]	T&	*(a.begin() + n)	vector, deque
a.at(n)	T&	*(a.begin() + n)	vector, deque

Tabulka 15.7 si zaslouží určitý komentář. Za prvé si všimněte, že oba výrazy `a[n]` a `a.at(n)` vracejí referenci na n-tý prvek (číslováno od 0) kontejneru. Rozdíl spočívá v tom, že výraz `a.at(n)` provádí kontrolu a v případě, že se prvek `n` nenachází v platném rozsahu, vyvolá výjimku `out_of_range`. Dále se asi ptáte, proč například výraz `push_front()` je definován pro kontejnery `list` a `deque` a není definován pro kontejner `vector`. Předpokládejme, že chcete vložit novou hodnotu na začátek vektoru se 100 prvky. Abyste získali prostor, musíte přesunout prvek č. 99 na pozici č. 100, prvek č. 98 na pozici č. 99 a tak dále. To-to je operace s lineární složitostí, protože přesun 100 prvků bude trvat 100krát déle než přesun prvku jediného. Ale operace uvedené v tabulce 15.7 mají být implementovány pouze v případě, že mohou být provedeny se složitostí konstantní. Návrh seznamů a obousměrných front však umožňuje přidat prvek na začátek bez přesunutí ostatních prvků na nové pozice, takže operaci `push_front()` se složitostí v konstantním čase implementovat mohou. Operace `push_front()` a `push_back()` ilustruje obrázek 15.4.

Nyní se podrobněji podíváme na šest sekvenčních kontejnerových typů.

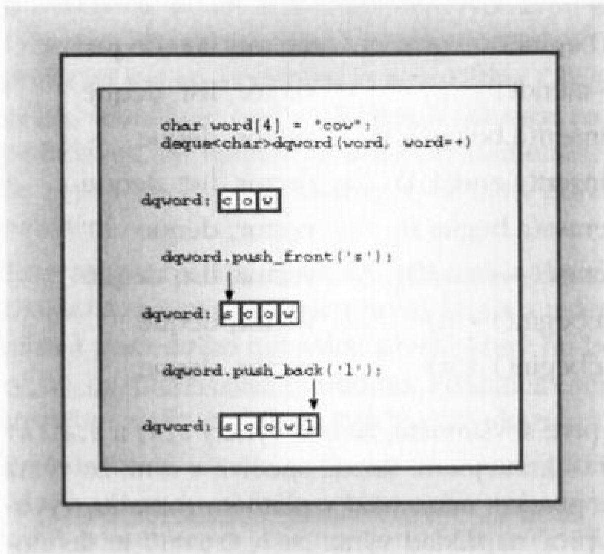
Vektor (`vector`)

Několik příkladů používajících šablonu `vector` (je definována v hlavičkovém souboru `vector.h`) jste již viděli. Stručně řečeno je `vector` třídní reprezentací pole. Třída poskytuje automatickou správu paměti, díky níž lze velikost objektu třídy `vector` dynamicky měnit, zvětšovat a zmenšovat podle toho, jak jsou prvky přidávány nebo odebírány. Přístup k prvkům je přímý. Prvky lze na konec přidávat nebo je z konce odebírat v konstantním čase, ale vkládání a odebírání ze začátku nebo z prostředku je operace v čase lineárním.

Vektor není jen sekvencí, je také modelem konceptu *reverzibilního kontejneru*. Díky tomu má navíc dvě metody třídy: metodu `rbegin()` vracející iterátor na první prvek reverzní sekvence, a metodu `rend()`, která pro reverzní sekvenci vrací iterátor na prvek za koncem kontejneru. Je-li tedy `dice` kontejnerem `vector<int>` a `Show(int)` funkcí zobrazující celé číslo, zobrazí následující kód obsah kontejneru `dice` nejdříve v normálním pořadí a potom v pořadí obráceném:

```
for_each(dice.begin(), dice.end(), Show); // normální pořadí
cout << endl;
```

```
for_each(dice.rbegin(), dice.rend(), Show);          // obrácené pořadí
cout << endl;
```



Obrázek 15.4 Operace push_front() a push_back()

Operátor vrácený oběma metodami je typu `reverse_iterator` a jeho platnost je v rozsahu třídy. Vzpomeňte si, že inkrementováním tohoto iterátoru ho posunujete kontejnerem v opačném pořadí.

Šablonová třída `vector` je nejjednodušším ze sekvenčních typů a je považována za typ, který by se měl používat implicitně, pokud požadavky programu neuspokojí lépe určité vlastnosti jiného typu.

Obousměrná fronta (deque)

Šablonová třída `deque` (deklarovaná v hlavičkovém souboru `deque`) představuje obousměrnou frontu. Podle způsobu implementace v knihovně STL připomíná hodně vektor podporující přímý přístup. Hlavní rozdíl spočívá v tom, že operace vkládání prvků na začátek objektu třídy `deque` a jejich odebírání ze začátku jsou operace časově konstantní, zatímco stejné operace u objektu třídy `vector` jsou časově lineární. Jestliže tedy většina operací bude probíhat na začátku a na konci sekvence, uvažujte o použití datové struktury `deque`.

Vzhledem k tomu, že vkládání a odebírání na obou stranách fronty má být časově konstantní, je návrh třídy `deque` složitější než návrh třídy `vector`. Ačkoli tedy obě třídy nabízejí přímý přístup k prvkům a vkládání a odebírání prvků z prostředku je časově lineární, měla by třída `vector` tyto operace provádět rychleji.

Seznam (list)

Šablonová třída `list` (deklarovaná v hlavičkovém souboru `list`) představuje obousměrný spojový seznam. Každý prvek kromě prvního a posledního je spojen s prvkem předcházejícím a s prvkem následujícím, což znamená, že seznam lze procházet oběma směry. Hlavní rozdíl mezi třídami `list` a `vector` je ten, že třída `list` vkládá prvky na kterékoli místo v seznamu a ze seznamu je odebírá v konstantním čase. (Jak si jistě vzpomínáte, šablo-

na vector provádí vkládání a odebírání v konstantním čase pouze na konci, jinak vkládání a odebírání probíhá v čase lineárním.) Třída vector tedy klade důraz na rychlý přístup (možný díky přímému přístupu), zatímco třída list klade důraz na rychlé vkládání a odebírání prvků.

Třída list je stejně jako třída vector reverzibilním kontejnerem. Na rozdíl od třídy vector však třída list nepodporuje zápis používaný pro pole ani přímý přístup. Iterátor třídy list na rozdíl od iterátoru třídy vector ukazuje i po vložení prvků do kontejneru či jejich odebrání stále na stejný prvek. Toto konstatování si musíme objasnit. Předpokládejme iterátor ukazující na pátý prvek kontejneru třídy vector. Potom na začátek kontejneru vložíte prvek. Všechny ostatní prvky se musí přesunout, aby vytvořily místo, takže po vložení obsahuje pátý prvek hodnotu, která byla v prvku čtvrtém. Iterátor tedy ukazuje na stejné místo, ale na jiná data. Při vložení nového prvku do objektu třídy list však k přesunu existujících prvků nedojde; změní se pouze informace o vazbách. Iterátor ukazující na určitý prvek na něj bude ukazovat stále, ale může být spojen s jinými prvky než předtím.

Šablonová třída list obsahuje kromě funkcí ze sekvencí a reverzibilních kontejnerů několik členských funkcí, orientovaných na seznam. Mnoho z nich je uvedeno v tabulce 15.8. (Úplný seznam metod a funkcí knihovny STL najdete v příloze G). O šablonový parametr Alloc se normálně starat nemusíte, protože jeho hodnota je implicitní.

Tabulka 15.8 Některé členské funkce třídy list.

Funkce	Popis
void merge(list<T, Alloc>& x)	Sloučí seznam x se seznamem volajícím. Oba seznamy musí být setříděny. Výsledný setříděný seznam bude v seznamu volajícím, seznam x bude prázdný.
void remove(const T & val)	Odstraní ze seznamu všechny instance val. Složitost této funkce je časově lineární.
void sort()	Setřídí seznam pomocí operátoru <; pro N prvků je složitost přirozený logaritmus N.
void splice(iterator pos, list<T, Alloc> x)	Vloží obsah seznamu x před pozici pos a seznam x ponechá prázdný. Složitost této funkce je časově konstantní.
void unique()	Zredukuje každou souvislou skupinu stejných prvků na prvek jediný. Složitost této funkce je časově lineární.

Tyto metody společně a také metodu insert() ilustruje program ve výpisu 15.9.

Výpis 15.9 list.cpp

```
// list.cpp - použití kontejneru list
#include <iostream>
#include <list>
#include <iterator>
```

```
using namespace std;
int main()
{
    list<int> one(5, 2); // seznam 5 kvót hodnota 2
    int stuff[5] = {1,2,4,8, 6};
    list<int> two;
    two.insert(two.begin(),stuff, stuff + 5 );
    int more[6] = {6, 4, 2, 4, 6, 5};
    list<int> three(two);
    three.insert(three.end(), more, more + 6);
    cout << "Seznam one: ";
    ostream_iterator<int,char> out(cout, " ");
    copy(one.begin(), one.end(), out);
    cout << endl << "Seznam two: ";
    copy(two.begin(), two.end(), out);
    cout << endl << "Seznam three: ";
    copy(three.begin(), three.end(), out);
    three.remove(2);
    cout << endl << "Seznam three bez dvojek: ";
    copy(three.begin(), three.end(), out);
    three.splice(three.begin(), one);
    cout << endl << "Seznam three po provedeni funkce splice(): ";
    copy(three.begin(), three.end(), out);
    cout << endl << "Seznam one: ";
    copy(one.begin(), one.end(), out);
    three.unique();
    cout << endl << "Seznam three po provedeni funkce unique(): ";
    copy(three.begin(), three.end(), out);
    three.sort();
    three.unique();
    cout << endl << "Seznam three po setrideni a po provedeni funkce
       unique(): ";
    copy(three.begin(), three.end(), out);
    two.sort();
    three.merge(two);
    cout << endl << "Setrideny seznam two slouceny se seznamem three: ";
    copy(three.begin(), three.end(), out);
    cout << endl;
    return 0;
}
```

Kompatibilita:

Starší verze mohou používat hlavičkové soubory `list.h` **a** `iterator.h` **a také třídu** `ostream_iterator<int>` **místo třídy** `ostream_iterator<int, char>`.

Zde je výstup:

```
Seznam one: 2 2 2 2 2
Seznam two: 1 2 4 8 6
Seznam three: 1 2 4 8 6 6 4 2 4 6 5
```

```
Seznam three bez dvojek: 1 4 8 6 6 4 4 6 5
Seznam three po provedeni funkce splice(): 2 2 2 2 2 1 4 8 6 6 4 4 6 5
Seznam one:Seznam three po provedeni funkce unique(): 2 1 4 8 6 4 6 5
Seznam three po setrideni a po provedeni funkce unique(): 1 2 4 5 6 8
Setrideny seznam two slouceny se seznamem three: 1 1 2 2 4 4 5 6 6 8 8
```

Poznámky k programu

V programu je použita technika probíraná dříve při použití všeobecné funkce `copy()` z knihovny STL a objektu šablony `ostream_iterator` pro zobrazení obsahu kontejneru.

Hlavní rozdíl mezi metodami `insert()` a `splice()` spočívá v tom, že metoda `insert()` původní rozsah do rozsahu cílového zkopíruje, zatímco metoda `slice()` původní rozsah do cílového přesune. Po přesunutí obsahu seznamu `one` do seznamu `three` zůstane seznam `one` prázdný. (Metoda `slice()` má i jiné prototypy pro přesunutí samotných prvků a rozsahu prvků). Platnost iterátorů po provedení metody `slice()` zůstává. Jestliže tedy iterátor nastavíte na nějaký prvek v seznamu `one`, bude na stejný prvek ukazovat i potom, kdy se tento prvek zásluhou metody `slice()` přemístí do seznamu `three`.

Všimněte si, že metoda `unique()` zredukuje stejné hodnoty sousedních prvků na hodnotu jedinou. Jakmile program provede metodu `three.unique()`, bude seznam `three` obsahovat dvě čtyřky a dvě šestky, které spolu nesousedí. Avšak použijete-li před metodou `unique()` metodu `sort()`, bude každá hodnota omezena na jediný výskyt.

Existuje i nečlenská funkce `sort()` (výpis 15.6), ta však vyžaduje iterátory přímého přístupu. Protože dohoda o rychlém vkládání se přímého přístupu vzdala, nemůžete nečlenskou funkci `sort()` u seznamu použít. Třída tedy obsahuje verzi členskou, která funguje v rámci třídních omezení.

Sada nástrojů třídy list

Metody třídy `list` tvoří užitečnou sadu nástrojů. Předpokládejme například, že máte uspořádat dva poštovní seznamy. Můžete uspořádat každý seznam zvlášť, sloučit je a potom pomocí metody `unique()` odstranit položky, které se vykytují vícekrát.

Metody `sort()`, `merge()` a `unique()` existují i ve verzích s přídavným parametrem, který určuje alternativní funkci pro porovnávání prvků. Podobně má metoda `remove()` verzi s přídavným parametrem specifikujícím funkci, která určuje, zda byl prvek odstraněn. Tyto parametry jsou příklady predikátních funkcí, což je téma, ke kterému se vrátíme později.

Fronta (`queue`)

Šablonová třída `queue` (deklarovaná v hlavičkovém souboru `queue` (dříve `queue.h`)) je adaptér. Vzpomeňte si, že šablona `ostream_iterator` je adaptér, díky němuž může výstupní proud použít rozhraní iterátoru. Podobně šablona `queue` umožňuje základní třídě (implicitně `deque`) zobrazit typické rozhraní fronty.

Šablona `queue` má více omezení než šablona `deque`. Nejenže neumožňuje přímý přístup k prvkům fronty, ale neumožňuje ve frontě ani iteraci. Omezuje vás na základní operace, které jsou pro frontu definovány. Můžete přidat prvek na konec fronty, odebrat prvek ze začátku fronty, zobrazit hodnotu prvního a posledního prvku, zjistit počet prvků a ověřit, zda je fronta prázdná. Seznam těchto operací je uveden v tabulce 15.9.

Tabulka 15.9 Operace definované pro třídu queue

Metoda	Popis
bool empty() const	Vrátí `true`, je-li fronta prázdná, jinak vrátí `false`.
size_type size() const	Vrátí počet prvků ve frontě.
T& front()	Vrátí referenci na prvek na začátku fronty.
T& back()	Vrátí referenci na prvek na konci fronty.
void push(const T& x)	Vloží na konec fronty prvek `x`.
void pop()	Odebere prvek ze začátku fronty.

Všimněte si, že metoda `pop()` data nenačítá, ale odstraňuje je. Jestliže chcete nějakou hodnotu z fronty použít, vyberte ji nejdříve pomocí metody `front()` a potom ji metodou `pop()` odstraňte.

Fronta s prioritou (`priority_queue`)

Šablonová třída `priority_queue` (deklarovaná také v hlavičkovém souboru `queue`) je dalším adaptérem. Podporuje stejné operace jako třída `queue`. Hlavní rozdíl spočívá v tom, že prvek s největší hodnotou se přesune na začátek fronty. (Život není vždycky spravedlivý a fronty také ne.) Vnitřní rozdíl je ten, že základní třídou je třída `vector`. Porovnání rozhodující o tom, který prvek se dostane do čela fronty, můžete změnit dodáním volitelného parametru konstruktoru:

```
priority_queue<int> pq1;                     // implicitní verze
priority_queue<int> pq2(greater<int>);  // určí pořadí pomocí funkce
                                              // greater<>()
```

Funkce `greater<>()` je předdefinovaný funkční objekt, který bude v této kapitole probrán později.

Zásobník (`stack`)

Stejně jako třída `queue` je i třída `stack` (deklarovaná v hlavičkovém souboru `stack` – dříve `stack.h`) adaptérem. Vytváří pro základní třídu (implicitně `vector`) typické rozhraní zásobníku.

Šablona `stack` má více omezení než šablona `vector`. Nejenže neumožňuje přímý přístup k prvkům zásobníku, ale neumožňuje v zásobníku ani iteraci. Omezuje vás na základní operace, které jsou pro zásobník definovány. Můžete vložit hodnotu na vrchol zásobníku, odebrat prvek z vrcholu zásobníku, zobrazit hodnotu vrchního prvku, zjistit počet prvků a ověřit, zda je zásobník prázdný. Seznam těchto operací je uveden v tabulce 15.10.

Tabulka 15.10 Operace definované pro třídu stack.

Metoda	Popis
bool empty() const	Vrátí `true`, je-li zásobník prázdný, jinak vrátí `false`.
size_type size() const	Vrátí počet prvků v zásobníku.
T& top()	Vrátí referenci na prvek na vrcholu zásobníku.
void push(const T& x)	Vloží na vrchol zásobníku prvek `x`.
void pop()	Odebere prvek z vrcholu zásobníku.

Stejně jako při práci s třídou `queue`, chcete-li použít hodnotu ze zásobníku, musíte ji nejdříve načíst pomocí metody `top()` a potom odstranit metodou `pop()`.

Asociativní kontejnery

Asociativní kontejner představuje další vylepšení konceptu kontejneru. Sdružuje hodnotu s klíčem a pomocí klíče tuto hodnotu vyhledá. Hodnotami by například mohly být struktury představující informace o zaměstnanci jako jsou jméno, adresa, číslo kanceláře, telefonní číslo domů a do práce, plán zdravotních vyšetření, a tak dále, a klíčem by mohlo být jednoznačné číslo pracovníka. Budete-li chtít informace o pracovníkovi přečíst, vyhledá program strukturu zaměstnance pomocí klíče. Vzpomeňte si, že pro kontejner `X` výraz `X::value_type` obecně označuje typ hodnoty uložené v kontejneru. Pro asociativní kontejner výraz `X::key_type` označuje typ použitý pro klíč.

Síla asociativního kontejneru spočívá v možnosti rychlého přístupu k prvkům. Stejně jako sekvence umožňuje asociativní kontejner vkládat nové prvky; nemůžete však konkretizovat místo, kam budou prvky vloženy. Asociativní kontejner totiž obvykle data ukládá podle určitého algoritmu, aby pak mohl informace rychle načíst.

Knihovna STL nabízí čtyři asociativní kontejnery: `set`, `multiset`, `map` a `multimap`. První dva typy jsou definovány v hlavičkovém souboru `set` (dříve odděleně v souborech `set.h` a `multiset.h`) a druhé dva typy v hlavičkovém souboru `map` (dříve odděleně v souborech `map.h` a `multimap.h`).

Nejjednodušší z této skupiny je `set`; jeho návratový typ je shodný s typem klíče a klíče jsou jedinečné, což znamená, že v kontejneru existuje pouze jedna instance klíče. Hodnotou kontejneru `set` je ve skutečnosti klíč. Typ `multiset` je stejný jako typ `set`, ale může mít více klíčů se stejnou hodnotou. Jestliže například klíč a hodnota jsou typu `int`, mohl by objekt kontejneru `multiset` obsahovat hodnoty 1, 2, 2, 2, 3, 5, 7, 7.

U typu `map` se typ hodnoty od typu klíče liší a klíče jsou jedinečné – každý klíč má jinou hodnotu. Typ `multimap` je typu `map` podobný až na to, že jeden klíč může být sdružen s více hodnotami.

Informací o uvedených typech je příliš mnoho na to, abychom je obsáhli v této kapitole (seznam metod je však uveden v příloze G), podíváme se tedy pouze na jednoduché příklady používající asociativní kontejnery `set` a `multimap`.

Příklad s kontejnerem `set`

Asociativní kontejner `set` z knihovny STL modeluje několik konceptů. Je to asociativní množina, je reverzibilní, setříděný a jeho klíče jsou jedinečné – nemůže tedy obsahovat více klíčů určité hodnoty. Stejně jako třídy `vector` a `list` používá kontejner typu `set` šablonový parametr obsahující uložený typ:

```
set<string> A;    // kontejner set s objekty typu řetězec
```

Pomocí druhého nepovinného parametru lze označit porovnávací funkci nebo objekt, který se použije pro řazení klíčů. Implicitně se používá šablona `less<>` (bude probrána později). Ve starších implementacích nemusí implicitní hodnota existovat a je tedy nutné explicitní šablonový parametr dodat:

```
set<string, less<string> > A;    // starší implementace
```

Uvažujte následující kód:

```
const int N = 6;
string s1[N] = {"daleko", "obyvatelstvo", "pro", "laso", "moci", "pro"};
set<string> A(s1, s1 + N);                // inicializuje set A pomocí
                                           // rozsahu z pole
ostream_iterator<string, char> out(cout, " ");
copy(A.begin(), A.end(), out);
```

Stejně jako ostatní kontejnery má asociativní kontejner `set` konstruktor (viz tabulka 15.6), jehož parametry jsou iterátory označující rozsah. Představují jednoduchý způsob inicializace kontejneru obsahem pole. Nezapomeňte, že poslední prvek rozsahu je prvek následující bezprostředně za koncem a výraz `s1 + N` ukazuje na pozici za koncem pole `s1`. Výstup tohoto fragmentu kódu ilustruje, že klíče jsou jedinečné (řetězec „`pro`" se v poli objevuje dvakrát, ale v kontejneru jednou) a kontejner je setříděn:

```
daleko laso obyvatelstvo moci pro
```

Matematika definuje pro množinu některé standardní operace. Sjednocením dvou množin vznikne množina kombinující obsahy obou množin. Jestliže je určitá hodnota oběma množinám společná, objeví se ve sjednocené množině díky jedinečnosti klíče pouze jednou. Průnik množin je množina obsahující prvky společné oběma množinám. Rozdíl dvou množin je první množina bez prvků společných oběma množinám.

Knihovna STL obsahuje algoritmy, které tyto operace podporují. Spíše než o metody se jedná o obecné funkce a nejsou tedy objekty typu `set` omezeny. Všechny objekty typu set však předpoklady pro použití těchto algoritmů splňují, především pak to, že kontejner musí být setříděný. Funkce `set_union()` má jako parametry pět iterátorů. První dva definují rozsah v prvním objektu typu set, druhé dva rozsah v druhém objektu stejného typu a posledním parametrem je výstupní iterátor identifikující místo, kam se zkopíruje výsledek. Chcete-li například zobrazit sjednocení množin `A` a `B`, napište tento příkaz:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
          ostream_iterator<string, char> out(cout, " "));
```

Předpokládejme, že výsledek nechcete zobrazit, ale chcete ho vložit do množiny `C`. V tom případě by posledním parametrem byl iterátor ukazující do množiny `C`. Evidentně se nabízí napsat `C.begin()`, ale takto to ze dvou důvodů fungovat nebude. Prvním důvodem

je to, že asociativní kontejnery set pokládají klíče za konstantní hodnoty, takže iterátor vrácený funkcí `C.begin()` je konstantní a nelze ho použít jako výstupní. Druhým důvodem, proč nelze přímo použít funkci `C.begin()` je, že funkce `set_union()` stejně jako funkce `copy()` přepisuje v kontejneru již existující data a požaduje, aby v kontejneru bylo dost místa pro uložení nových informací. Kontejner `C` tento požadavek nesplňuje, protože je prázdný. Oba problémy však řeší dříve probíraná šablona `insert_iterator`. Již dříve jste viděli, že z kopírování dělá vkládání. A také modeluje koncept výstupního iterátoru, takže ji můžete použít pro zápis do kontejneru. Pro zkopírovaní informací do kontejneru `C` tedy můžete vytvořit anonymní `insert_iterator`. Vzpomeňte si, že konstruktor má jako parametry kontejner a iterátor:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
          insert_iterator<set<string> >(C, C.begin()));
```

Funkce `set_intersection()` a `set_difference()` najdou průnik obou množin a jejich rozdíl. Mají stejné rozhraní jako funkce `set_union()`.

Dvě užitečné metody jsou `lower_bound()` a `upper_bound()`. Metoda `lower_bound()` má jako parametr klíč a vrací iterátor na první člen množiny, který není menší než klíč. Podobně metoda `upper_bound()` má jako parametr klíč a vrací iterátor na první člen množiny, který je větší než klíč. Kdybyste například měli množinu řetězců, mohli byste pomocí těchto metod identifikovat oblast zahrnující všechny řetězce v množině od „`b`" až do „`f`".

Protože o umístění přidávaných prvků rozhoduje třídění, obsahuje třída metody pro vkládání, které vkládané prvky pouze specifikují, ale neurčují jejich umístění. Jestliže například `A` a `B` jsou množiny řetězců, můžete napsat následující kód:

```
string s("tenis");
A.insert(s);                        // vloží hodnotu
B.insert(A.begin(), A.end());       // vloží rozsah
```

Použití těchto dvou množin ilustruje program ve výpisu 15.10.

Výpis 15.10 set.cpp

```
// set.cpp - několik operací s kontejnerem set
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
  const int N = 6;
  string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
  string s2[N] = {"metal", "any", "food", "elegant", "deliver", "for"};
  set<string> A(s1, s1 + N);
  set<string> B(s2, s2 + N);
  ostream_iterator<string, char> out(cout, " ");
```

```
    cout << "Mnozina A: ";
    copy(A.begin(), A.end(), out);
    cout << endl;
    cout << "Mnozina B: ";
    copy(B.begin(), B.end(), out);
    cout << endl;

    cout << "Sjednoceni mnozin A a B:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;
    cout << "Prunik mnozin A a B:\n";
    set_intersection(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;
    cout << "Rozdil mnozin A a B:\n";
    set_difference(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;
    set<string> C;
    cout << "Mnozina C:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
    insert_iterator<set<string> >(C, C.begin()));
    copy(C.begin(), C.end(), out);
    cout << endl;

    string s3("grungy");
    C.insert(s3);
    cout << "Mnozina C po vlozeni:\n";
    copy(C.begin(), C.end(),out);
    cout << endl;
    cout << "Zobrazeni rozsahu:\n";
    copy(C.lower_bound("ghost"),C.upper_bound("spook"), out);
    cout << endl;

    return 0;
}
```

Kompatibilita:

Starší implementace mohou používat hlavičkové soubory `set.h`, `iterator.h` a `algo.h`. Jako druhý šablonový parametr pro typ `set` mohou také vyžadovat šablonu `less<string>` a místo šablony `ostream_iterator<string, char>` mohou používat šablonu `ostream_iterator<string>`.

Zde je výstup:

```
Množina A: buffoon can for heavy thinkers
Množina B: any deliver elegant food for metal
Sjednocení množin A a B: any buffoon can deliver elegant food for heavy
metal thinkers
Průnik množin A a B:
for
```

```
Rozdíl množin A a B:
buffoon can heavy thinkers
Množina C:
any buffoon can deliver elegant food for heavy metal thinkers
Množina C po vložení:
any buffoon can deliver elegant food for grungy heavy metal thinkers
Zobrazeni rozsahu:
grungy heavy metal
```

Příklad s kontejnerem multimap

Stejně jako kontejner set je i kontejner multimap setříděný, asociativní kontejner. Typ klíče se v něm však liší od typu hodnoty a objekt typu multimap může s určitým klíčem spojovat více hodnot.

V základní deklaraci typu multimap jsou jako parametry šablony specifikovány typ klíče a typ hodnoty. Například následující deklarace vytvoří objekt multimap s klíčem typu int a s uloženou hodnotou typu string:

```
multimap<int, string> codes;
```

Pomocí nepovinného třetího parametru lze označit porovnávací funkci nebo objekt, který se použije pro řazení klíčů. Implicitně se používá šablona less<> (bude probrána později), která má jako parametr typ klíče. Ve starších implementacích bývá nutné dodat tento šablonový parametr explicitně.

Aby byly informace spolu, kombinuje skutečný typ hodnoty typ klíče a typ dat do jediného páru. Knihovna STL k tomu používá šablonovou třídu pair<class T, class U>, která do jediného objektu ukládá dva druhy hodnot. Jestliže keytype je typ klíče a datatype typ uložených dat, potom je typ hodnoty pair<const keytype, datatype>. Například typ hodnoty výše deklarovaného objektu codes je pair<const int, string>.

Předpokládejme například, že byste chtěli uložit názvy měst pomocí klíče, kterým je směrovací číslo. Jednou z možností je vytvořit pár a ten pak vložit:

```
pair<const int, string> item(213, "Los Angeles");
codes.insert(item);
```

Jinou možností je vytvořit anonymní objekt pair a vložit ho do jediného příkazu:

```
codes.insert(pair<const int, string> (213, "Los Angeles"));
```

Vzhledem k tomu, že položky jsou tříděny podle klíče, nemusíte místo vložení identifikovat.

Ke dvěma složkám objektu pair můžete přistupovat pomocí členů first a second:

```
pair<const int, string> item(213, "Los Angeles");
cout << item.first << ' ' << item.second << endl;
```

Jak se získávají informace o objektu typu multimap? Členská funkce count() má jako parametr klíč a vrací počet položek s tímto klíčem. Členské funkce lower_bound() a upper_bound() s klíčem jako parametrem fungují stejně jako u typu set. Členská funkce equal_range() má jako parametr klíč a vrací iterátory představující rozsah odpovídající tomuto klíči. Aby bylo možné vrátit dvě hodnoty, balí je tato metoda do jediného objektu pair, přičemž v tomto pří-

padě jsou oba šablonové parametry typu iterátor. Například následující kód by vytiskl seznam měst z objektu `codes` se směrovacím číslem 718:

```
pair<multimap>KeyType, string>::iterator,
  multimap<KeyType,string>::iterator> range
                      = codes.equal_range(718);
cout << "Mesta se smerovacim cislem 718:";
for (it = range.first; it != range.second; ++it)
  cout << (*it).second << endl;
```

Většinu z těchto technik demonstruje program ve výpisu 15.11. Pro zjednodušení některého kódu se také používá příkaz `typedef`.

Výpis 15.11 multmap.cpp

```
// multmap.cpp - použiti kontejneru multimap
#include <iostream>
#include <string>
#include <map>
#include <algorithm>
using namespace std;
typedef int KeyType;
typedef pair<const KeyType, string> Pair;
typedef multimap<KeyType, string> MapCode;
int main()
{
    MapCode codes;

    codes.insert(Pair(415, "San Francisco"));
    codes.insert(Pair(510, "Oakland"));
    codes.insert(Pair(718, "Brooklyn"));
    codes.insert(Pair(718, "Staten Island"));
    codes.insert(Pair(415, "San Rafael"));
    codes.insert(Pair(510, "Berkeley"));
    cout << "Pocet mest se smerovacim cislem 415:" << codes.count(415) <<
      endl;
    cout << "Pocet mest se smerovacim cislem 718: " << codes.count(718) <<
      endl;
    cout << "Pocet mest se smerovacim cislem 510: " << codes.count(510) <<
      endl;
    cout << "Smerovaci cislo   Mesto\n";
    MapCode::iterator it;
    for (it = codes.begin(); it != codes.end(); ++it)
        cout << "     " << (*it).first << "     "
            << (*it).second    << endl;

    pair<MapCode::iterator, MapCode::iterator> range =
                                      codes.equal_range(718);
    cout << "Mesta se smerovacim cislem 718:\n";
    for (it = range.first; it != range.second; ++it)
        cout <<  (*it).second    << endl;
    return 0;
}
```

Kompatibilita:

Starší implementace mohou používat hlavičkové soubory `multimap.h` a `algo.h`. Jako třetí šablonový parametr pro typ `multimap` mohou vyžadovat šablonu `less<Pair>` a místo šablony `ostream_iterator<string, char>` mohou používat šablonu `ostream_iterator<string>`. Borland C++ Builder 1.0 vyžaduje definici objektu `Pair` příkazem `typedef` bez klíčového slova `const`.

Zde je výstup:

```
Pocet mest se smerovacim cislem 415: 2
Pocet mest se smerovacim cislem 718: 2
Pocet mest se smerovacim cislem 510: 2
Smerovaci cislo   Mesto
415               San Francisco
415               San Rafael
510               Oakland
510               Berkeley
718               Brooklyn
718               Staten Island
Mesta se smerovacim cislem 718:
Brooklyn
Staten Island
```

Funkční objekty (aka funktory)

Mnoho algoritmů knihovny STL používá *funkční objekty*, které jsou známé také jako *funktory*. Funktor je jakýkoli objekt, který lze použít po vzoru funkce s kulatými závorkami. Patří sem normální názvy funkcí, ukazatele na funkce a objekty tříd, pro něž je operátor `()` přetížen, to znamená třídy, pro něž je definována podivně vyhlížející funkce `operator()()`. Například třídu byste mohli definovat takto:

```
class Linear
{
private:
   double slope;
   double y0;
public:
   Linear(double _sl = 1, double _y = 0)
         : slope(_sl), y0(_y) {}
   double operator()(double x) {return y0 + slope * x;}
};
```

Potom byste objekty třídy `Linear` mohli použít jako funkce:

```
Linear f1;
Linear f2(2.5, 10.0);
```

```
double y1 = f1(12.5);                    // f1.operator()(12.5)
double y2 = f2(0.4);
```

Vzpomínáte si na funkci `for_each()`? Na každý člen rozsahu použila určitou funkci:

```
for_each(books.begin(), books.end(), ShowReview);
```

Obecně lze říct, že třetím parametrem by mohl být funktor, nejen běžná funkce. To vyvolává otázku – jak tento třetí parametr deklarovat? Jako ukazatel na funkci ho deklarovat nemůžete, neboť ukazatel na funkci specifikuje typ parametru. Vzhledem k tomu, že kontejner může obsahovat téměř libovolný typ, nevíte předem, jaký konkrétní typ parametru by se měl použít. Knihovna STL tento problém řeší pomocí šablon. Prototyp funkce `for_each()` vypadá takto:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

Funkce `ShowReview()` měla tento prototyp:

```
void ShowReview(const Review &);
```

Identifikátor `ShowReview` má tedy typ `void (*)(const Review &)` a tento typ je přiřazen šablonovému parametru `Function`.

Koncepty funktorů

Stejně jako definuje knihovna STL koncepty pro kontejnery a iterátory, definuje také koncepty funktorů:

- *Generátor* je funktor, který lze volat bez použití parametrů.
- *Unární funkce* je funktor, který lze volat s jedním parametrem.
- *Binární funkce* je funktor, který lze volat se dvěma parametry.

Například funktorem dodaným funkci `for_each()` by měla být unární funkce, protože se v určitém okamžiku používá na jeden prvek kontejneru.

Tyto koncepty jsou samozřejmě vylepšeny:

- Unární funkce vracející hodnotu typu `bool` se nazývá *predikát*.
- Binární funkce vracející hodnotu typu `bool` se nazývá *binární predikát*.

Několik funkcí z knihovny STL požaduje jako parametry predikáty nebo binární predikáty. Například program ve výpisu 15.6 používá verzi funkce `sort()`, která má jako třetí parametr binární predikát:

```
bool WorseThan(const Review & r1, const Review & r2);
...
sort(books.begin(), books.end(), WorseThan);
```

Šablona list obsahuje členskou funkci `remove_if()`, která má jako parametr predikát. Tento predikát použije na každý člen ve vyznačeném rozsahu a prvky, u nichž predikát vrátí hodnotu true, odstraní. Následující kód by například odstranil ze seznamu `three` všechny prvky větší než 100:

```
bool tooBig(int n) { return n > 100; }
list<int> scores;
...
scores.remove_if(tooBig);
```

Tento poslední příklad mimochodem ukazuje situaci, ve které je funktor třídy užitečný. Předpokládejme, že byste ze druhého seznamu chtěli odstranit každou hodnotu větší než 200. Bylo by příjemné, kdybyste mohli odstraněnou hodnotu předat funkci `tooBig()` jako druhý parametr a mohli byste tedy funkci použít s různými hodnotami, ale predikát nemůže mít více jak jeden parametr. Pokud však navrhnete třídu `TooBig`, budete moci další informace předávat pomocí položek tříd namísto pomocí parametrů funkce:

```
template<class T>
class TooBig
{
private:
  T cutoff;
public:
  TooBig(const T & t) : cutoff(t) {}
  bool operator()(const T & v) { return v > cutoff; }
};
```

Zde je jedna hodnota (`v`) předána jako parametr funkce, zatímco druhý parametr (`cutoff`) je nastaven konstruktorem třídy. Při této definici mohou být hodnoty použité pro inicializaci objektů třídy `TooBig` různé:

```
TooBig<int> f100(100);
list<int> froobies;
list<int> scores;
...
froobies.remove_if(f100);           // použije pojmenovaný funkční objekt
scores.remove_if(TooBig<int>(200)); // vytvoří funkční objekt
```

Kompatibilita:

Metoda `remove_if()` je šablonová metoda šablonové třídy. Šablonové metody představují poslední rozšíření vlastností šablon jazyka C++ (hlavně proto, že je používá knihovna STL) a v čase psaní této knihy je většina kompilátorů ještě neimplementovala. Existuje však i nečlenská funkce `remove_if()`, která používá jako parametry rozsah (dva iterátory) a predikát.

Předpokládejme, že byste již šablonovou funkci s dvěma parametry měli:

```
template <class T>
bool tooBig(const T & val, const T & lim)
{
  return val > lim;
}
```

Třídu můžete převést na funkční objekt s jedním parametrem:

```
template<class T>
class TooBig2
{
private:
   T cutoff;
public:
   TooBig2(const T & t) : cutoff(t) {}
   bool operator()(const T & v) { return tooBig<T>(v > cutoff); }
};
```

Můžete tedy psát následující příkazy:

```
TooBig2<int> tB100(100);
int x;
cin >> x;
if (tB100(x))                    // stejné jako (tooBig(x, 100))
```

Volání `tB100(x)` je stejné jako volání `tooBig(x, 100)`, ale z funkce se dvěma parametry se stane funkční objekt s jedním parametrem, přičemž druhý parametr se použije k vytvoření tohoto funkčního objektu. Stručně řečeno, třídní funktor `TooBig2` je funkční adaptér, upravující funkci pro odlišné rozhraní.

Předdefinované funktory

V knihovně STL je definováno několik elementárních funkčních objektů. Provádějí takovou činnost jako je sčítání dvou hodnot, porovnávání dvou hodnot a tak podobně. Mají sloužit jako podpora těch funkcí z knihovny STL, které mají jako parametry funkce. Uvažujte například funkci `transform()`. Má dvě verze. První verze má čtyři parametry. Prvními dvěma jsou iterátory určující rozsah v kontejneru. (Tento způsob již určitě znáte.) Třetím parametrem je iterátor určující místo, kam se zkopíruje výsledek. Posledním parametrem je funktor, který se použije na každý prvek v dané oblasti a vede k vytvoření všech nových prvků. Uvažujte například následující kód:

```
const int LIM = 5;
double arr1[LIM] = {36, 39, 42, 45, 48};
vector<double> gr8(arr1, arr1 + LIM);
ostream_iterator<double, char> out(cout, " ");
transform(gr8.begin(), gr8.end(), out, sqrt);
```

Tento kód by vypočítal mocninu každého prvku a výslednou hodnotu by odeslal do výstupního proudu. Cílový iterátor se může nacházet v původním rozsahu. Pokud byste například ve výše uvedeném příkladu nahradili funkci `out` funkcí `gr8.begin()`, nahradily by nově zkopírované hodnoty hodnoty původní. Je jasné, že použitý funktor musí pracovat s jediným parametrem.

Druhá verze používá funkci se dvěma parametry a tato funkce se použije na jeden prvek z obou rozsahů. Má další, v pořadí třetí parametr, který identifikuje začátek druhého rozsahu. Pokud by například `m8` byl druhý objekt typu `vector<double>` a funkce `mean(double, double)` by vracela průměr dvou hodnot, převedl by následující kód průměrnou hodnotu z každého páru hodnot v oblasti od `gr8` do `m8`:

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, mean);
```

Nyní předpokládejme, že chcete sečíst dvě pole. Jako parametr nemůžete použít operátor +, protože pro typ `double` je + vestavěný operátor a ne funkce. Mohli byste definovat a použít funkci pro sčítání dvou čísel:

```
double add(double x, double y) { return x + y; }
...
transform(gr8.begin(), gr8.end(), m8.begin(), out, add);
```

V tom případě byste ovšem pro každý typ museli definovat samostatnou funkci. Bylo by lepší definovat šablonu, pokud již ovšem neexistuje v knihovně STL. V hlavičkovém souboru functional (dříve `function.h`) je definováno několik funkčních objektů šablonových tříd včetně `plus<>()`.

Použití třídy `plus<>` pro obyčejné sčítání je možné, i když trochu těžkopádné:

```
#include <functional>
...
plus<double> add; // vytvoří objekt plus<double>
double y = add(2.2, 3.4); // použije funkci plus<double>::operator()()
```

Snadnější ovšem je, tvoří-li parametr funkční objekt:

```
transform(gr8.begin(), gr.end(), m8.begin(), out, plus<double>() );
```

V tomto kódu se pojmenovaný objekt nevytváří, ale pomocí konstruktoru plus<double> se vytvoří funkční objekt, který provede sčítání. (Závorky označují volání implicitního konstruktoru; funkci `transform()` je předán vytvořený funkční objekt.)

Pro všechny aritmetické, relační a logické operátory nabízí knihovna STL ekvivalenty ve formě funkčních objektů. Názvy těchto funktorů jsou uvedeny v tabulce 15.11. Lze je použít s vestavěnými typy jazyka C++ nebo s libovolným uživatelem definovaným typem, který přetěžuje odpovídající operátor.

Upozornění

Starší implementace používají název `times` namísto názvu `multiplies`.

Tabulka 15.11 Operátory a jejich ekvivalenty ve formě funkčních objektů.

Operátor	Funkční objekt
+	plus
-	minus
*	multiplies
/	divides
%	modulus
-	negate
==	equal_to
!=	not_equal_to

>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not

Přizpůsobivé funktory a funkční adaptéry

Všechny předdefinované funktory z tabulky 15.11 jsou *přizpůsobivé*. Knihovna STL má vlastně pět příbuzných konceptů. Jsou jimi *přizpůsobivý generátor, přizpůsobivá unární funkce, přizpůsobivá binární funkce, přizpůsobivý predikát* a *přizpůsobivý binární predikát*.

Objekt funktoru je přizpůsobivý, jestliže obsahuje položky vytvořené pomocí příkazu `typedef`, které identifikují typy parametrů a návratový typ. Těmito položkami jsou `result_type`, `first_argument_type` a `second_argument_type`. Například návratový typ objektu `plus<int>` je `plus<int>::result_type`.

Význam přizpůsobivosti funkčního objektu spočívá v tom, že tento objekt pak mohou používat funkční adaptéry, které existenci položek vytvořených příkazem `typedef` předpokládají. Například funkce s parametrem, kterým je přizpůsobivý funktor, může pomocí položky `result_type` deklarovat proměnnou odpovídající návratovému typu tohoto funktoru.

Knihovna STL nabízí několik tříd s funkčními adaptéry, které tyto vlastnosti používají. Předpokládejme například, že chcete každý prvek vektoru `gr8` vynásobit číslem `2.5`. To vyžaduje použití verze funkce `transform()` s unární funkcí jako parametrem, jako v dříve uvedeném příkladě:

```
transform(gr8.begin(), gr8.end(), out, sqrt);
```

Násobení dokáže provést funktor `multiplies()`, ale funkce je binární. Potřebujete tedy funkční adaptér, který převede funktor se dvěma parametry na funktor s parametrem jedním. Jeden způsob jste viděli dříve v příkladu s třídou `TooBig2`, knihovna STL však tento proces zautomatizovala pomocí tříd `binder1st` a `binder2nd`, které přizpůsobivé binární funkce převádějí na přizpůsobivé funkce unární.

Podívejme se na třídu `binder1st`. Předpokládejme, že máte objekt přizpůsobivé binární funkce `f2()`. Vytvoříte objekt `binder1st`, který váže určitou hodnotu – nazvěme ji `val` – a ta se použije jako první parametr funkce `f2()`:

```
binder1st(f2, val) f1;
```

Potom volání `f1(x)` s jediným parametrem vrátí stejnou hodnotu jako volání `f2(val, x)`, kde val je první parametr a `f1()` parametr druhý. To znamená, že `f1(x)` je ekvivalentem `f2(val, x)` až na to, že místo binární funkce je použita funkce unární. Funkce `f2()`

je přizpůsobena. To je opět možné pouze proto, že funkce `f2()` je funkcí přizpůsobivou. Může to vypadat trochu těžkopádně, avšak knihovna STL používání třídy binder1st díky funkci bind1st zjednodušuje. Dodáte jí název funkce a hodnotu pro vytvoření objektu binder1st a funkce vám vrátí objekt tohoto typu. Převeďme například binární funkci `multiplies()` na funkci unární, která násobí parametr číslem `2.5`. Stačí napsat toto:

```
bind1st(multiplies<double>(), 2.5)
```

Řešení vynásobení každého prvku ve vektoru `gr8` číslem `2.5` a jeho zobrazení vypadá tedy takto:

```
transform(gr8.begin(), gr8.end(), out,
        bind1st(multiplies<double>(), 2.5));
```

Třída `binder2nd` je podobná až na to, že konstantu přiřazuje druhému parametru. Má pomocnou funkci `bind2nd`, která funguje analogicky funkci `bind1st`.

Tip

Pokud nějaká funkce z knihovny STL potřebuje unární funkci a vy máte přizpůsobivou funkci binární, která dělá totéž, můžete tuto binární funkci přizpůsobit unárnímu rozhraní pomocí funkcí `bind1st()` a `bind2nd()`.

Některé poslední příklady obsahuje krátký program ve výpisu 15.12.

Výpis 15.12 funadap.cpp

```
// funadap.cpp - použití funkčních adaptérů
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>
using namespace std;
const int LIM = 5;int main(){
    double arr1[LIM] = {36, 39, 42, 45, 48};
    double arr2[LIM] = {25, 27, 29, 31, 33};
    vector<double> gr8(arr1, arr1 + LIM);
    vector<double> m8(arr2, arr2 + LIM);
    ostream_iterator<double, char> out(cout, " ");
    copy(gr8.begin(), gr8.end(), out);
    cout << endl;
    copy(m8.begin(), m8.end(), out);
    cout << endl;
    transform(gr8.begin(), gr8.end(), m8.begin(), out, plus<double>());
    cout << endl;
     transform(gr8.begin(), gr8.end(), out, bind1st(multiplies<double>(),
      2.5));
    cout << endl;
     return 0;
}
```

Kompatibilita:

Starší implementace mohou používat hlavičkové soubory `vector.h`, `iterator.h`, `algo.h` a `function.h`. Také místo názvu `multiplies` mohou používat název `times`.

Zde je výsledek:

```
36 39 42 45 48
25 27 29 31 33
61 66 71 76 81
90 97.5 105 112.5 120
```

Algoritmy

Knihovna STL obsahuje mnoho nečlenských funkcí pro práci s kontejnery. Několik z nich jste již viděli: `sort()`, `copy()`, `for_each()`, `random_shuffle()`, `set_union()`, `set_intersection()`, `set_difference()` a `transform()`. Pravděpodobně jste si všimli, že se vyznačují stejným celkovým návrhem, rozsah zpracovávaných dat identifikují pomocí iterátorů a stanovují místo jejich určení. Některé mají jako parametr funkční objekt, který se použije jako součást zpracování dat.

Návrhy algoritmů funkcí obsahují dvě hlavní složky. Za prvé vytvářejí obecné typy pomocí šablon, a za druhé, vytvářejí obecnou reprezentaci pro přístup k datům v kontejneru. Funkce `copy()` tedy může pracovat s kontejnerem obsahujícím hodnoty typu `double` v poli, s kontejnerem obsahujícím hodnoty typu `string` ve spojovém seznamu nebo s kontejnerem obsahujícím uživatelem definované objekty ve stromové struktuře, jak například činí kontejner typu `set`. Vzhledem k tomu, že ukazatele představují speciální případ iterátorů, lze funkce z knihovny STL, například `copy()`, použít u obyčejných polí.

Třída `string` sice součástí knihovny STL není, ale je s ohledem na ni navržena. Obsahuje například členské funkce `start()` a `end()` a může tedy používat rozhraní knihovny STL.

Díky jednotnému návrhu rozhraní existují mezi kontejnery různých druhů smysluplné vztahy. Můžete například pomocí funkce `copy()` zkopírovat hodnoty z obyčejného pole do objektu třídy `vector`, z objektu třídy `vector` do objektu třídy `list`, a z objektu třídy `list` do objektu třídy `set`. Pomocí operátoru `==` můžete různé druhy kontejnerů porovnávat, například kontejner `deque` s kontejnerem `vector`. Takové porovnání je možné proto, že u kontejnerů operátor `==` porovnává obsah pomocí iterátorů, takže objekty typu `deque` a `vector` jsou stejné v případě, že mají stejný obsah ve stejném pořadí.

Skupiny algoritmů

Knihovna STL dělí knihovnu algoritmů na čtyři základní skupiny:

- Nemodifikující sekvenční operace
- Změnové sekvenční operace
- Třídicí a relační operace
- Generalizované numerické operace

První tři skupiny jsou popsány v hlavičkovém souboru algorithm (dříve algo.h), zatímco čtvrtá skupina se speciálně orientuje na numerická data a má svůj vlastní hlavičkový soubor numeric. (Dříve byly také v hlavičkovém souboru algo.h.)

Nemodifikující sekvenční operace pracují s každým prvkem v oblasti. Tyto operace ponechávají kontejner beze změny. Do této kategorie patří například funkce find() a for_each().

Změnové sekvenční operace také pracují s každým prvkem v oblasti. Jak však název napovídá, mohou obsah kontejneru změnit. Tato změna se může týkat hodnot nebo pořadí, v němž jsou uloženy. Do této kategorie spadají například funkce transform(), random_shuffle() a copy().

Třídicí a relační operace obsahují několik třídicích funkcí (včetně funkce sort()) a různé jiné funkce včetně operací s množinami.

Numerické operace zahrnují funkce pro výpočet obsahu oblasti, počítají vnitřní výsledek ze dvou kontejnerů, částečné sumy a odlišnosti sousedních prvků. Tyto operace jsou typické pro pole, takže jejich nejpravděpodobnějším uživatelem je kontejner vector.

Úplný souhrn těchto funkcí je uveden v příloze G.

Všeobecné vlastnosti

Jak se neustále přesvědčujete, pracují funkce knihovny STL s iterátory a rozsahy, které iterátory stanovují. Předpoklady iterátorů jsou naznačeny ve funkčním prototypu. Například funkce copy() má tento prototyp:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Protože identifikátory InputIterator a OutputIterator jsou šablonovými parametry, mohlo by se místo nich použít T a U. V dokumentaci ke knihovně STL se však jako šablonové parametry používají názvy označující koncept, který parametr modeluje. Tato deklarace nám tedy říká, že parametry určujícími rozsah musí být vstupní iterátory nebo lepší, a že iterátorem označujícím umístění výsledku musí být iterátor výstupní nebo lepší.

Jeden ze způsobů klasifikace algoritmů vychází z umístění výsledku vyprodukovaného algoritmem. Některé algoritmy pracují na místě, jiné vytvářejí kopie. Když například skončí funkce sort(), zabírá výsledek stejné místo jako původní data. Funkce sort() je tedy *algoritmus pracující na místě*. Funkce copy() však posílá výsledek své práce na jiné místo a je tedy *algoritmem kopírovacím*. Funkce transform() může provádět obojí. Stejně jako funkce copy() označuje pomocí výstupního iterátoru místo pro uložení výsledku, na rozdíl od ní však umožňuje, aby výstupní iterátor ukazoval do oblasti vstupní, takže zkopírované transformované hodnoty mohou překrýt hodnoty původní.

Některé algoritmy existují ve dvou verzích – ve verzi pracující na místě a ve verzi kopírovací. Podle konvence knihovny STL se k názvu kopírovací verze přidává přípona _copy. Druhá verze je doplněna o parametr s výstupním iterátorem, který specifikuje místo, na které se výsledek zkopíruje. Například funkce replace() má tento prototyp:

```
template<class ForwardIterator, class T>
void replace(ForwardIterator, first, ForwardIterator last,
        const T& old_value, const T& new_value);
```

Funkce nahrazuje každou instanci old_value instancí new_value. To se děje na místě. Protože tento algoritmus prvky kontejneru čte i do nich zapisuje, musí být typem iterátoru iterátor dopředný nebo lepší. Kopírovací verze má následující prototyp:

```
template<class ForwardIterator, class OutputIterator, class T>
OutputIterator replace_copy(ForwardIterator, first, ForwardIterator last,
                    OutputIterator result,
                    const T& old_value, const T& new_value);
```

V tomto případě jsou výsledná data zkopírovaná do místa určeného výstupním iterátorem result, takže pro stanovení oblasti stačí čtecí vstupní iterátor.

Všimněte si, že návratovým typem funkce replace_copy() je OutputIterator. Podle konvence vrací kopírovací algoritmy iterátor na pozici následující bezprostředně za poslední zkopírovanou hodnotou.

Další společnou variantu tvoří některé funkce, které mají verze provádějící určitou činnost podmíněně, v závislosti na výsledku aplikace funkce na prvek kontejneru. Tyto verze přidávají ke svému názvu příponu _if. Funkce replace_if() například nahradí starou hodnotu hodnotou novou, pokud aplikace funkce na starou hodnotu vrátí hodnotu true. Zde je její prototyp:

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate před, const T& new_value);
```

Vzpomeňte si, že predikát je název unární funkce vracející hodnotu typu bool. Existuje také verze nazvaná replace_copy_if(). Co dělá a jaký je její prototyp si pravděpodobně dokážete představit.

Stejně jako v případě iterátoru InputIterator je i Predicate názvem šablonového parametru a mohl by se stejně dobře nazývat T nebo U. Knihovna STL však dává přednost názvu Predicate, aby uživateli připomněla, že skutečný parametr by měl být modelem konceptu Predicate. Podobně používá knihovna STL termíny jako Generator a BinaryPredicate pro označení parametrů, které by měly modelovat jiné koncepty funkčních objektů.

Použití standardní knihovny šablon

Části ve standardní knihovně šablon jsou navrženy tak, aby spolupracovaly. Komponenty jsou tvořeny nástroji, ale mohou jimi být i stavební bloky pro vytváření jiných nástrojů. Ukážeme si to na příkladu. Předpokládejme, že chcete napsat program umožňující uživateli zadávat slova. Na konci budete chtít statistiku zadaných slov, seznam slov v abecedním pořadí (nebudete rozlišovat velká a malá písmena) a statistiku udávající, kolikrát bylo každé slovo zadáno. Pro jednoduchost předpokládejme, že vstup neobsahuje čísla ani interpunkční znaménka.

Zadání a uložení seznamu slov je dosti jednoduché. Podle příkladu z výpisu 15.5 můžete vytvořit objekt třídy `vector<string>` pomocí funkce `push_back()` do vektoru přidávat slova:

```
vector<string> words;
string input;
while (cin >> input && input != "konec")
  words.push_back(input);
```

A co abecedně uspořádaný seznam slov? Můžete použít funkci `sort()` a za ní funkci `unique()`, ale při tomto způsobu řešení dojde k přepsání původních dat, protože funkce `sort()` představuje algoritmus pracující na místě. Existuje snazší způsob, který tento problém obchází. Vytvořte objekt třídy `set<string>` a zkopírujte (pomocí vkládacího iterátoru) slova z objektu třídy vector do objektu třídy set. Typ set třídí obsah automaticky, volá funkci `sort()`, a dovolí pouze jednu kopii klíče, takže volá funkci `unique()`. Počkejte! Specifikace má ignorovat velká a malá písmena. Možným řešením je použít ke zkopírování dat z objektu vector do objektu set funkci `transform()` místo funkce `copy()`. Transformační funkci použijte takovou, která všechna písmena v řetězci převede na malá.

```
set<string> wordset;
transform(words.begin(), words.end(),
  insert_iterator<set<string> > (wordset, wordset.begin()), ToLower);
```

Napsat funkci `ToLower()` je snadné. Stačí ve funkci `transform()` použít funkci `tolower()` na každý prvek řetězce, přičemž řetězec bude představovat jak zdroj, tak i cíl. Vzpomeňte si, že objekty třídy `string` mohou také používat funkce knihovny STL. Předání a vrácení řetězce jako reference znamená, že algoritmus pracuje s původním řetězcem a nemusí vytvářet kopie.

```
string & ToLower(string & st)
{
  transform(st.begin(), st.end(), st.begin(), tolower);
  return st;
}
```

Počet výskytů každého slova zadaného na vstupu můžete získat pomocí funkce count(). Parametry funkce jsou rozsah a hodnota, a funkce vrací počet výskytů hodnoty v tomto rozsahu. Rozsah můžete dodat pomocí objektu třídy `vector`, zatímco objekt třídy `set` poskytne seznam slov, která se budou počítat. To znamená, že pro každé slovo uvedené v objektu `set` se bude počítat počet jeho výskytů v objektu `vector`. Aby byl výsledný počet spojen se správným slovem, uložte slovo a počet výskytů jako objekt typu `pair<const string, int>` do objektu typu `map`. Slovo bude představovat klíč (pouze jedna kopie) a počet výskytů hodnotu. Tento kód lze napsat jediným cyklem:

```
map<string, int> wordmap;
set<string>::iterator si;
for (si = wordset.begin(); si != wordset.end(); si++)
  wordmap.insert(pair<string, int>(*si, count(words.begin(),
  words.end(), *si)));
```

Upozornění

Starší implementace deklarují funkci `count()` jako typ `void`. Místo návratové hodnoty musíte dodat čtvrtý parametr předaný jako referenci, a počet položek se uloží do tohoto parametru:

```
int ct = 0;
count(words.begin(), words.end(), *si), ct)); // počet se přidá do ct
```

Třída map má zajímavou vlastnost – můžete použít zápis jako pro pole, přičemž klíče slouží jako indexy pro přístup k uloženým hodnotám. Například zápis `wordmap["the"]` by představoval hodnotu spojenou s klíčem „`the`“, což je v tomto případě počet výskytů řetězce „`the`“. Vzhledem k tomu, že kontejner `wordset` obsahuje všechny klíče používané kontejnerem `wordmap`, můžete následující kód použít jako alternativní a atraktivnější způsob ukládání výsledků:

```
for (si = wordset.begin(); si != wordset.end(); si++)
   wordmap[*si] = count(words.begin(0, words.end(), *si);
```

Protože iterátor si ukazuje na kontejner `wordset`, označuje výraz `*si` řetězec a může sloužit jako klíč pro kontejner `wordmap`. V tomto kódu jsou klíče i hodnoty uloženy do kontejneru `wordmap`.

Podobně můžete pomocí zápisu pro pole hlásit výsledky:

```
for (si = wordset.begin(); si != wordset.end(); si++)
   cout << si << ": " << wordmap[*si] << endl;
```

Při platnosti klíče je odpovídající hodnotou 0.

Program ve výpisu 15.13 tyto myšlenky shrnuje a obsahuje kód pro zobrazení obsahu tří kontejnerů (kontejneru typu vector se vstupem, kontejneru typu set se seznamem slov a kontejneru typu map s počtem slov).

Výpis 15.13 usealgo.cpp

```
//usealgo.cpp
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <iterator>
#include <algorithm>
#include <cctype>
using namespace std;
string & ToLower(string & st);
int main()
{
    vector<string> words;
    cout << "Zadejte slova (\"konec\" pro ukonceni):\n";
    string input;
```

```
    while (cin >> input && input != "konec")
        words.push_back(input);
    cout << "Zadali jste nasledujici slova:\n";
    ostream_iterator<string,char> out(cout, " ");
    copy(words.begin(), words.end(), out);
    cout << endl;
    // vložení slov do kontejneru typu set, konverze na malá písmena
    set<string> wordset;
    transform(words.begin(), words.end(),
        insert_iterator<set<string> > (wordset, wordset.begin()),
        ToLower);
    cout << "\nSeznam slov podle abecedy:\n";
    copy(wordset.begin(), wordset.end(), out);
    cout << endl;
    // vložení slov a počet výskytů do kontejneru typu map
    map<string, int> wordmap;
    set<string>::iterator si;
    for (si = wordset.begin(); si != wordset.end(); si++)
        wordmap[*si] = count(words.begin(), words.end(), *si);
    // zobrazeni obsahu kontejneru typu map
    cout << "\nPocet vyskytu:\n";
    for (si = wordset.begin(); si != wordset.end(); si++)
        cout << *si << ": " << wordmap[*si] << endl;
    return 0;
}
string & ToLower(string & st)
{
    transform(st.begin(), st.end(), st.begin(), tolower);
    return st;
}
```

Kompatibilita:

Starší implementace mohou používat hlavičkové soubory `vector.h`, `set.h`, `map.h`, `iterator.h`, `algo.h` a `ctype.h`. V šablonách `set` a `map` také mohou vyžadovat další parametr `less<string>`. Starší verze mohou místo šablony `ostream_iterator<string, char>` používat šablonu `ostream_iterator<string>` a funkci `count()` používají jako typ `void`, což jsme zmínili již dříve.

Zde je ukázka běhu:

```
Zadejte slova ("konec" pro ukonceni):
The dog saw the cat and thought the cat fat
The cat thought the cat perfect
konec
Zadali jste nasledujici slova:
The dog saw the cat and thought the cat fat The cat thought the cat perfect

Seznam slov podle abecedy:
and cat dog fat perfect saw that the thought
```

```
Pocet vyskytu:
and: 1
cat: 4
dog: 1
fat: 1
perfect: 1
saw: 1
that: 1
the: 4
thought: 2
```

Ponaučení vyplývající z používání knihovny STL zní, že byste měli vždy uvažovat, jestli je nutné, abyste určitý kód psali sami. Obecný a pružný návrh knihovny STL by vám měl hodně práce ušetřit. Návrháři knihovny jsou lidé, kteří při psaní algoritmů myslí především na efektivitu. Algoritmy jsou tedy dobře volené a jsou spřažené.

Ostatní knihovny

V jazyce C++ existuje několik dalších knihoven tříd, které jsou více specializované než příklady uvedené v této kapitole. V hlavičkovém souboru `complex` existuje šablonová třída `complex` určená pro komplexní čísla, se specializací na typy `float`, `long` a `long double`. Disponuje standardními operacemi s komplexními čísly a rovněž standardními funkcemi, které lze s těmito čísly použít.

Hlavičkový soubor `valarray` obsahuje šablonovou třídu `valarray`. Třída je navržena pro reprezentaci numerických polí a podporuje různé operace, které s těmito poli souvisejí, jako například přidání obsahu jednoho pole k poli jinému, použití matematických funkcí na jednotlivé prvky pole a použití lineární algebry.

Shrnutí

Jazyk C++ obsahuje sadu výkonných knihoven nabízejících řešení pro mnoho programových problémů a také nástroje, které mnoho problémů zjednodušují. Třída `string` obsahuje prostředky vhodné pro správu řetězců jako objektů. Poskytuje automatickou správu paměti a velké množství metod a funkcí pro práci s řetězci. S jejich pomocí můžete například řetězce spojovat, vkládat jeden řetězec do druhého, pracovat s řetězcem od konce, vyhledávat v něm určité znaky nebo podřetězce a provádět vstupní a výstupní operace.

Šablona `auto_ptr` usnadňuje správu paměti, přidělené operátorem `new`. Jestliže pro uložení adresy vrácené operátorem `new` použijete místo obyčejného ukazatele objekt třídy `auto_ptr`, nebudete si muset pamatovat, že musíte také použít operátor `delete`. Po skončení platnosti objektu třídy `auto_ptr` zavolá destruktor operátor `delete` automaticky.

Standardní knihovnu šablon tvoří kolekce šablon kontejnerových tříd, iterátorových tříd, funkčních objektů a algoritmických funkcí, které se vyznačují jednotným návrhem podle zásad generického programování. Algoritmy zevšeobecňují šablony podle typu uložené-

ho objektu, zatímco rozhraní iterátorů zevšeobecňují podle typu kontejneru. Iterátory jsou generalizované ukazatele.

Pro označení sady požadavků používá knihovna STL termín „koncept". Koncept dopředného iterátoru například obsahuje požadavky na dereferencování objektu iterátoru pro čtení a psaní a na inkrementování iterátoru. Implementacím samotným se říká „modelování" konceptu. Koncept dopředného iterátoru by bylo možné modelovat obyčejným ukazatelem nebo objektem navrženým pro procházení spojového seznamu. Konceptům založeným na jiných konceptech se říká „vylepšení". Obousměrný iterátor je například vylepšením konceptu iterátoru dopředného.

Kontejnerové třídy, například `vector` a `set`, jsou modely konceptů kontejnerů jako jsou kontejner, sekvence a asociativní kontejner. Knihovna STL definuje několik šablon kontejnerových tříd: `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap` a `bitset`. Také definuje šablony tříd adaptérů `queue`, `priority_queue` a `stack`; tyto třídy přizpůsobují základní kontejnerovou třídu tak, aby získala rozhraní charakteristické pro adaptér. Proto například třída `stack`, ačkoli je založená implicitně na třídě `vector`, umožňuje vkládat pouze na vrchol zásobníku a také pouze z vrcholu odebírat.

Některé algoritmy jsou vyjádřeny jako metody kontejnerové třídy, ale většinu z nich tvoří obecné, nečlenské funkce. Umožňují to iterátory představující rozhraní mezi kontejnery a algoritmy. Jednou z výhod tohoto přístupu je, že stačí mít pouze jednu funkci `for_each()` nebo jednu funkci `copy()` namísto samostatných verzí pro jednotlivé kontejnery. Druhou výhodou je, že algoritmy z knihovny STL lze použít u kontejnerů, které součástí knihovny nejsou, jako například obyčejná pole, objekty třídy `string` a jiné třídy navržené v souladu s iterátory knihovny STL a ve stylu kontejnerů.

Kontejnery i algoritmy jsou charakterizovány typem iterátoru, který poskytují nebo potřebují. Měli byste kontrolovat, aby se kontejner vyznačoval takovým konceptem iterátoru, který podporuje potřeby daného algoritmu. Například algoritmus funkce `for_each()` používá vstupní iterátor, jehož minimální požadavky splňují všechny typy kontejnerových tříd knihovny STL. Avšak funkce `sort()` požaduje iterátory přímého přístupu a ty všechny kontejnerové třídy nepodporují. V případě, že kontejnerová třída požadavky určitého algoritmu nesplňuje, může jako alternativu nabídnout specializovanou metodu. Třída `list` například obsahuje metodu `sort()` založenou na obousměrných iterátorech a může ji tedy použít místo funkce obecné.

Knihovna STL také nabízí funkční objekty neboli funktory, což jsou třídy s přetíženým operátorem `()`; je pro ně tedy definována metoda `operator()()`. Objekty těchto tříd lze vyvolat pomocí zápisu funkce, mohou však obsahovat další informace. Například přizpůsobivé funkční objekty obsahují příkazy `typedef`, identifikující typy parametrů a typ návratové hodnoty funkčního objektu. Tyto informace mohou využít jiné komponenty, například funkční adaptéry.

Knihovna STL reprezentuje obecné typy kontejnerů a nabízí celou řadu obecných operací implementovaných efektními algoritmy. Toto všechno je děláno genericky a představuje vynikající zdroj znovupoužitelného kódu. Programovací problémy můžete řešit přímo pomocí nástrojů knihovny STL nebo tyto nástroje můžete použít jako stavební bloky pro vypracování potřebných řešení.

Šablonové třídy `complex` a `valarray` podporují numerické operace pro komplexní čísla a pole.

Opakovací otázky

1. Uvažujte následující deklaraci třídy:

```
class RQ1
{
private:
            char * st;      // ukazuje na řetězec ve stylu jazyka C
public:
            RQ1() { st = new char[1]; strcpy(st, ""); }
RQ1(const char * s);
{ st = new char[strlen(s) + 1; strcpy(st; s); }
RQ1(const RQ1 & rq);
{ st = new char[strlen(rq.st) + 1; strcpy(st; rq.st); }
~RQ1() { delete [] st };
RQ & operator=(const RQ & rq);
// další příkazy
};
```

2. Vytvořte z této deklarace deklaraci používající objekt třídy `string`. Které metody se obejdou bez explicitních definic?
3. Vyjmenujte alespoň dvě výhody z hlediska snadnosti používání, které mají objekty třídy `string` ve srovnání s řetězci jazyka C.
4. Napište funkci, která bude mít jako parametr referenci na objekt třídy `string` a převede všechna písmena v objektu na velká.
5. Které následující příkazy představují nesprávné (koncepčně nebo syntakticky) použití třídy `auto_ptr`?

```
auto_ptr<int> pia(new int[20]);
auto_ptr<str> (new string);
int rigue = 7;
auto_ptr<int>pr(&rigue);
auto_ptr dbl (new double);
```

6. Kdybyste mohli vytvořit mechanický ekvivalent zásobníku na golfové hole, proč by se (koncepčně) jednalo o špatný pytel?
7. Proč se kontejner typu `set` nehodí pro uložení záznamu golfových zásahů?
8. Jestliže je ukazatel iterátorem, proč návrháři knihovny STL nepoužívají místo iterátorů jednoduše ukazatele?
9. Proč návrháři knihovny STL jednoduše nedefinovali iterátor základní třídy, neodvodili pomocí dědičnosti jiné typy iterátorů a nevyjádřili algoritmy podle iterátorů těchto tříd?
10. Uveďte alespoň tři příklady praktických výhod, které má objekt třídy `vector` oproti obyčejnému poli.
11. Kdyby byl program ve výpisu 15.6 implementován pomocí třídy `list` namísto `vector`, které části programu by byly nepoužitelné? Bylo by možné nepoužitelné části snadno opravit? Pokud ano, tak jak?

Programovací cvičení

1. Palindrom je řetězec, který se čte stejně zezadu i zepředu. Krátkými palindromy jsou například řetězce „tot" a „otto". Napište program, který si od uživatele vyžádá řetězec a předá referenci na řetězec na funkci, vracející hodnotu typu `bool`. Jestliže bude řetězec palindromem, vrátí funkce hodnotu `true`, jinak vrátí `false`. Zatím se nemusíte zabývat komplikacemi vyplývajícími z velkých písmen, mezer nebo interpunkce. V této verzi řetězce typu „Otto" a „Madam, I´m Adam" za palindrom nepovažujte. Abyste si úkol zjednodušili, klidně se můžete podívat do přílohy F na seznam metod třídy `string`.
2. Vytvořte stejný problém jako v programovacím cvičení č. 1, avšak tentokrát se komplikacemi způsobenými velkými písmeny, mezerami a interpunkcí zabývejte. Považujte tedy řetězec „Madam, I´m Adam" za palindrom. Testovací funkce by například mohla řetězec zredukovat na „madamimadam" a potom ověřit, zda je čtení zezadu stejné. Nezapomeňte na užitečnou knihovnu `cctype`. Můžete tam najít nějakou vhodnou, i když ne nezbytnou, funkci knihovny STL.
3. Máte za úkol napsat funkci podle rozhraní ve starém stylu. Prototyp je následující:

   ```
   int reduce(long ar[], int n);
   ```

   Parametry tvoří název pole a počet prvků v poli. Funkce pole setřídí, odstraní duplicitní hodnoty a vrátí hodnotu odpovídající počtu prvků ve zredukovaném poli. Při psaní této funkce využijte funkcí knihovny STL. (Pokud se rozhodnete použít obecnou funkci `unique()`, nezapomeňte, že vrací konec výsledného rozsahu.) Funkci vyzkoušejte v krátkém programu.
4. Řešte stejný problém jako v programovacím cvičení č. 3, ale z funkce udělejte šablonu:

   ```
   template<class T>
   int reduce(T ar[], int n);
   ```

   Vyzkoušejte funkci v krátkém programu s instancemi typu `long` a `string`.
5. Přepracujte program z výpisu 11.13 tak, že místo třídy `Queue` z kapitoly 11 použijete šablonovou třídu `queue` z knihovny STL.
6. Běžnou hrou je loterijní lístek. Obsahuje číslované puntíky, z nichž je náhodně vybrán určitý počet. Napište funkci `Lotto()` obsahující dva parametry. Prvním je počet puntíků na loterijním lístku a druhým počet náhodně vybraných puntíků. Funkce vrátí objekt třídy `vector<int>`, v němž budou náhodně vybraná čísla setříděna. Funkci byste mohli použít například následujícím způsobem:

   ```
   vector<int> winners;
   winners = Lotto(51, 6);
   ```

   Zde je objektu `winners` přiřazen objekt třídy `vector` s šesti náhodně vybranými čísly z intervalu od 1 do 51. Všimněte si, že prosté použití funkce `rand()` nestačí, neboť se mohou objevit duplicitní hodnoty.

Návrh: vytvořte ve funkci objekt třídy `vector` s různými hodnotami, použijte funkci `random_shuffle()` a potom tyto hodnoty čtěte ze začátku takto upraveného vektoru. Napište také krátký program, který funkci ověří.

7. Petr a Pavel chtějí pozvat přátele na večírek. Požádají vás o napsání programu, který bude provádět následující činnost:

   Petr zadá seznam jmen svých přátel. Jména budou uložena do kontejneru a po setřídění zobrazena.

   Pavel zadá seznam jmen svých přátel. Jména budou uložena do kontejneru a po setřídění zobrazena.

   Vytvořte třetí kontejner, ve kterém budou oba seznamy sloučeny a po vyloučení duplicit se zobrazí jeho obsah.

Vstup, výstup a soubory

KAPITOLA 16

Témata kapitoly:

- Vstup a výstup v jazyce C++
- Skupina tříd iostream
- Přesměrování
- Metody třídy ostream
- Formátování výstupu
- Metody třídy istream
- Stavy proudů
- Vstup ze souboru a výstup do souboru
- Použití třídy `ifstream` pro vstup ze souborů
- Použití třídy `ofstream` pro výstup do souborů
- Použití třídy `fstream` pro vstup ze souborů a výstup do souborů
- Zpracování příkazového řádku
- Binární soubory
- Přímý přístup k souborům
- Formátování incore

Vstupy a výstupy (zkráceně I/O) v jazyce C++ představují problém. Na jedné straně prakticky každý program používá vstup a výstup a naučit se je používat představuje jeden z prvních úkolů při studiu počítačového jazyka. Na straně druhé jazyk C++ implementuje vstup a výstup pomocí pokročilých vlastností, kterými jsou třídy, odvozené třídy, přetížené funkce, virtuální funkce, šablony a vícenásobná dědičnost. Máte-li tedy vstupu a výstupu v C++ skutečně rozumět, musíte toho o jazyce hodně znát. Když jste začínali, seznámili jste se v úvodních kapitolách se základními způsoby používání objektu `cin` třídy `istream` pro vstup a objektu `cout` třídy `ostream` pro výstup. Nyní se na vstupní a výstupní třídy jazyka C++ podíváme podrobněji, ukážeme si, jak jsou navrženy, a naučíme se výstup formátovat. (Jestliže jste několik kapitol přeskočili jenom proto, abyste se dozvěděli o pokročilém formátování, můžete části týkající se tohoto tématu prolistovat a povšimnout si technik. S vysvětlením se zdržovat nemusíte.)

Vstupní a výstupní vlastnosti C++ vycházejí ze stejných definic tříd, na kterých jsou založeny objekty `cin` a `cout`. V této kapitole tedy probereme konzolový vstup a výstup (klávesnice a obrazovka), což bude odrazový můstek pro zkoumání vstupu ze souboru a výstupu do souboru.

Výbor pro standardy ANSI/ISO C++ usiloval o větší kompatibilitu I/O jazyka C++ s I/O jazyka C a v důsledku toho došlo v tradičních postupech C++ k některým změnám.

Přehled vstupu a výstupu

Většina počítačových jazyků má vstup a výstup zabudován v sobě. Podíváte-li se například na seznam klíčových slov jazyků BASIC nebo Pascal, zjistíte, že součástí slovníku jazyka jsou příkazy `PRINT`, `writeln` a podobné. Jazyk C ani jazyk C++ však vstup a výstup zabudovaný nemají. Jestliže se podíváte na klíčová slova těchto jazyků, najdete `for` a `if`, ale nic, co by souviselo se vstupem a výstupem. Jazyk C původně starost o vstup a výstup přenechával implementátorům kompilátoru. Jedním z důvodů bylo ponechat jim volnost pro navržení takových vstupně-výstupních funkcí, které budou nejlépe vyhovovat cílovému počítači. V praxi většina implementátorů založila vstup a výstup na sadě knihovních funkcí, vyvinutých původně pro prostředí systému UNIX. Norma ANSI C tento vstupně-výstupní balík formálně uznala, nazvala ho standardní vstupně-výstupní balík a učinila ho povinnou součástí standardní knihovny jazyka C. Jazyk C++ tento balík také rozeznává, takže pokud znáte funkce jazyka C deklarované v hlavičkovém souboru `stdio.h`, můžete je používat i v programech napsaných v C++. (Novější implementace podporují tyto funkce pomocí hlavičkového souboru `cstdio`).

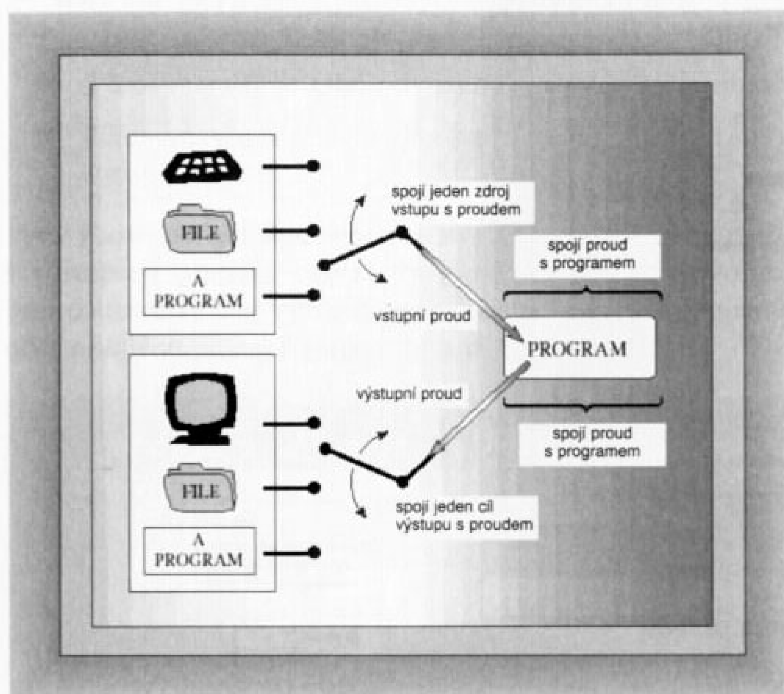
C++ se však při vstupu a výstupu spoléhá spíše na vlastní řešení než na řešení vytvořené v C a toto řešení představuje sada tříd definovaných v hlavičkových souborech `iostream` (dříve `iostream.h`) a `fstream` (`fstream.h`). Tato knihovna tříd není součástí formální definice jazyka (`cin` a `cout` nejsou klíčová slova); koneckonců počítačový jazyk definuje pravidla, podle kterých něco vytváříte, například třídy, a ne co byste podle těchto pravidel vytvářet měli. Ale stejně jako mají implementace jazyka C standardní knihovnu funkcí, má jazyk C++ standardní knihovnu tříd. Nejdříve tato standardní knihovna tříd představovala neformální standard a obsahovala pouze třídy definované v hlavičkových souborech `iostream` a `fstream`. Výbor pro ANSI/ISO C++ se rozhodl tuto knihovnu formálně uznat jako standardní knihovnu tříd a přidat k ní několik dalších standardních tříd, například ty, které jsme probírali v kapitole 15. V této kapitole se budeme zabývat standardním vstupem a výstupem. Nejdříve však prostudujeme koncepční vstupně-výstupní rámec v C++.

Proudy a vyrovnávací paměti

Program v C++ pohlíží na vstup a výstup jako na proud bajtů. Při vstupu načte bajty ze vstupního proudu a při výstupu je vloží do proudu výstupního. Pro textově orientované programy představuje každý bajt jeden znak. Obecněji lze říct, že bajty mohou tvořit binární reprezentaci znakových nebo numerických dat. Do vstupního proudu mohou bajty přicházet z klávesnice, ale také z paměťového zařízení jako je pevný disk nebo z jiného programu. Podobně mohou bajty z výstupního proudu téci na obrazovku, na tiskárnu, do paměťového zařízení nebo do jiného programu. Proud se chová jako prostředník mezi programem a zdrojem nebo cílem proudu. Zásluhou tohoto přístupu zachází program v C++ se vstupem z klávesnice stejným způsobem jako se vstupem ze souboru; stačí mu proud bajtů prozkoumat, nemusí vědět, odkud bajty přicházejí. Podobně může pomocí proudů zpracovat výstup nezávisle na místě určení bajtů. Správa vstupu se tedy skládá ze dvou fází:

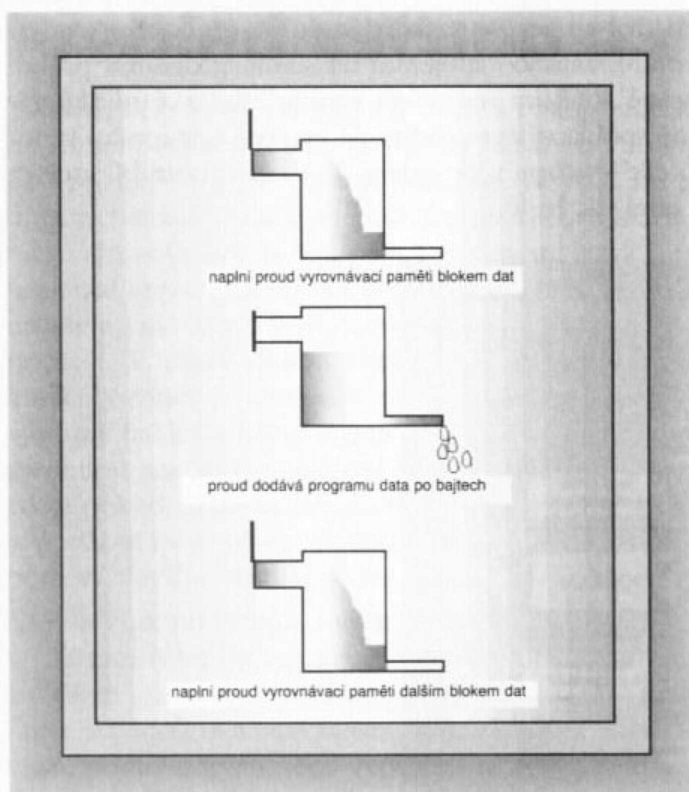
- Spojení proudu se vstupem do programu
- Spojení proudu se souborem

Jinými slovy potřebuje vstupní proud dvě spojení – na každém konci jedno. Spojení na straně souboru představuje zdroj proudu, zatímco spojením na straně programu proudí bajty do programu. (Spojením na straně souboru může být soubor, ale také nějaké zařízení, například klávesnice.) Podobně správa výstupu zahrnuje spojení výstupního proudu s programem a spojení nějakého cíle výstupu s proudem. Je to jako potrubí, kterým místo vody protékají informace (viz obrázek 16.1).



Obrázek 16.1 Vstup a výstup v C++

Obvykle lze vstup a výstup zpracovat účinněji pomocí *vyrovnávací paměti* (buffer). Jedná se o blok paměti, která je dočasná, a zprostředkovává přenos informací ze zařízení do programu nebo naopak. Disková zařízení běžně přenášejí informace v blocích o velikosti 512 bajtů nebo větších, zatímco programy často zpracovávají informace po bajtech. Vyrovnávací paměť pomáhá vyrovnávat rozdíly mezi oběma rychlostmi přenosu. Předpokládejme například, že nějaký program má spočítat počet znaků dolar v souboru na pevném disku. Mohl by ze souboru přečíst jeden znak, zpracovat ho, přečíst další znak ze souboru a tak dále. Čtení souboru z disku po znacích potřebuje hodně hardwarové aktivity a je pomalé. Při použití vyrovnávací paměti se načte z disku velký blok, uloží se do této vyrovnávací paměti a čtou se jednotlivé znaky. Protože čtení jednotlivých bajtů dat z paměti je mnohem rychlejší než čtení z pevného disku, je tento způsob mnohem rychlejší a také méně náročný na hardware. Samozřejmě, že jakmile program čtení z vyrovnávací paměti skončí, musí z disku načíst další blok dat. Princip je podobný vodní nádrži, která musí po velké bouřce zachytit hektolitry přívalové vody a potom ji dovést do vašeho domova civilizovanou rychlostí (viz obrázek 16.2). Podobně při výstupu program nejdříve zaplní vyrovnávací paměť, potom přenese celý blok dat na pevný disk a vyčistí vyrovnávací paměť, aby byla přichystána pro další dávku. Tomuto procesu se říká *vyprázdnění vyrovnávací paměti*. Analogický proces s potrubím si pravděpodobně dokážete představit sami.



Obrázek 16.2 Proud s vyrovnávací pamětí

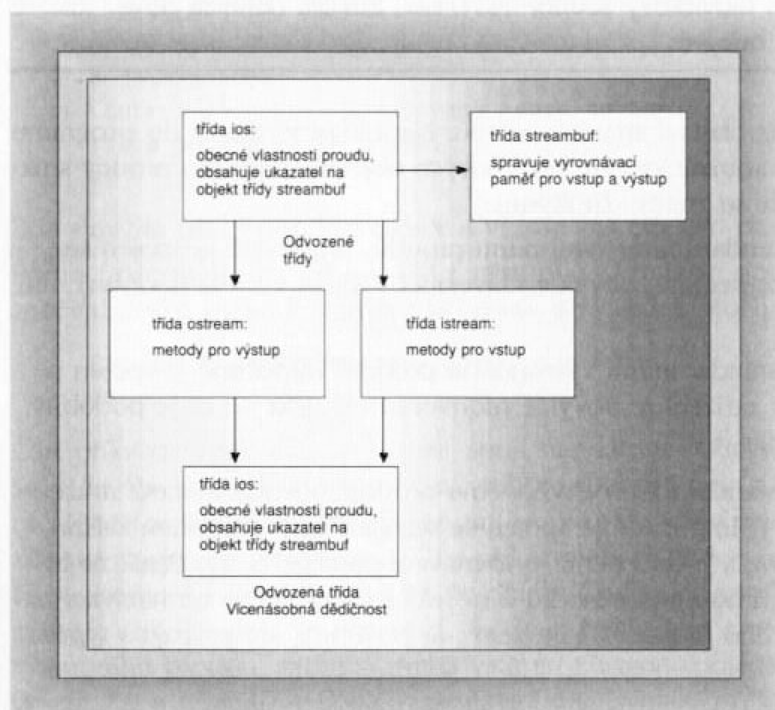
Vstup z klávesnice dodává znaky postupně, takže v tomto případě program vyrovnávací paměť pro vyrovnání rozdílu v rychlostech přenosu dat nepotřebuje. Vyrovnávací paměť však uživateli umožňuje vstup vrátit a opravit ještě předtím, než ho předá programu. Program v C++ za normálních okolností vyprázdní vyrovnávací paměť po stisku klávesy Enter. Proto také v příkladech z této knihy zpracování začíná až stisknutím této klávesy. Při výstupu na obrazovku program v C++ vyrovnávací paměť normálně vyprázdní při předání znaku označujícího nový řádek. V závislosti na implementaci může program vstup vyprázdnit i při jiných příležitostech, například při čekajícím vstupu. Jakmile tedy program narazí na příkaz vstupu, vyprázdní jakýkoli výstup ve výstupní vyrovnávací paměti. Implementace C++ odpovídající normě ANSI C by se měly takto chovat.

Proudy, vyrovnávací paměti a soubor iostream

Správa proudů a vyrovnávacích pamětí může být trochu komplikovaná, ale vložením hlavičkového souboru `iostream` (dříve `iostream.h`) dostanete do programu několik tříd navržených pro implementaci a správu proudů a vyrovnávacích pamětí. Nejnovější verze I/O v C++ vlastně definují šablonové třídy podporující data `char` i `wchar_t`. Pomocí příkazu `typedef` jazyk C++ napodobuje šablonovými specializacemi typu `char` tradiční nešablonové vstupně-výstupní implementace. Uvádíme některé z těchto tříd (viz obrázek 16.3):

- Třída `streambuf` poskytuje vyrovnávací paměť společně s metodami pro její zaplnění, přístup k jejímu obsahu, vyprázdnění a správu.
- Třída `ios_base` reprezentuje obecné vlastnosti proudu, například zda je otevřený a zda se jedná o proud binární nebo textový.
- Třída `ios` je založena na třídě `ios_base` a obsahuje ukazatel na objekt třídy `streambuf`.
- Třída `ostream` je odvozena od třídy `ios` a obsahuje metody pro výstup.
- Třída `istream` je také odvozena od třídy `ios` a obsahuje metody pro vstup.
- Třída `iostream` je založena na třídách `istream` a `ostream` a dědí tedy metody pro vstup i výstup.

Tyto vlastnosti můžete využívat, vytvoříte-li objekty patřičné třídy. Například při práci s výstupem použijte objekt třídy `ostream cout`. Vytvořením takového objektu otevřete proud, automaticky vytvoříte vyrovnávací paměť a spojíte ji s proudem. Získáte také možnost používat členské funkce této třídy.



Obrázek 16.3 Některé třídy pro vstup a výstup

Předefinování vstupu a výstupu

Ve standardu ISO/ANSI C++ byl vstup a výstup poněkud přepracován. Za prvé, z hlavičkového souboru `ostream.h` se stal `ostream` a třídy ze souboru `ostream` byly umístěny do prostoru jmen `std`. Za druhé, třídy pro vstup a výstup byly přepsány. Jako mezinárodní jazyk musí C++ umět pracovat s mezinárodními sadami znaků, které vyžadují 16 bitový typ zna-

ku. Do jazyka byl tedy k tradičnímu 8 bitovému (úzkému) typu znaku přidán 16 bitový typ `wchar_t` (wide, široký). Oba typy mají pro vstup a výstup své vlastní prostředky. Místo dvou samostatných sad tříd vyvinul výbor pro standardizaci šablonovou sadu vstupně-výstupních tříd, která obsahuje šablony `basic_istream<charT, traits<charT> >` a `basic_ostream<charT, traits<charT> >`. Šablona `traits<charT>` zase definuje určité rysy znaku, například způsob porovnání a hodnotu `EOF` (konec souboru). Standard poskytuje pro vstupně-výstupní třídy specializace `char` a `wchar_t`. Například třídy `istream` a `ostream` jsou definovány pomocí příkazu `typedef` jako specializace typu `char`. Podobně jsou třídy `wistream` a `wostream` specializacemi typu `wchar_t`. Například pro výstup proudů širokých znaků existuje objekt `wcout`. Definice jsou obsaženy v hlavičkovém souboru `ostream`.

Určité typově nezávislé informace, obsažené v základní třídě `ios`, byly přesunuty do nové třídy `ios_base`. Tato třída obsahuje různé formátovací konstanty jako `ios::fixed`, která se nyní nazývá `ios_base::fixed`. Třída `ios_base` obsahuje také některé parametry, které v dřívější třídě `ios` nebyly.

V některých případech odpovídá změna názvu souboru změně v definici třídy. Například v Microsoft Visual C++ 5.0 můžete vložit hlavičkový soubor `iostream.h` a získáte původní definice tříd, zatímco vložíte-li hlavičkový soubor `iostream`, získáte definice nové. Taková synchronizace však pravidlem nebývá.

Knihovna tříd `iostream` za vás obstará mnoho detailů. Například vložíte-li do programu soubor `iostream`, vytvoří se automaticky osm proudových objektů (čtyři pro proudy s úzkými znaky a čtyři pro proudy se znaky širokými):

- Objekt `cin` odpovídá standardnímu vstupnímu proudu. Implicitně je spojen se standardním vstupním zařízením, obvykle klávesnicí. Objekt `wcin` je podobný, ale pracuje s typem `wchar_t`.
- Objekt `cout` odpovídá standardnímu výstupnímu proudu. Implicitně je spojen se standardním výstupním zařízením, obvykle monitorem. Objekt `wcout` je podobný, ale pracuje s typem `wchar_t`.
- Objekt `cerr` odpovídá standardnímu chybovému proudu, s jehož pomocí můžete zobrazit chybové zprávy. Implicitně je spojen se standardním výstupním zařízením, obvykle monitorem, a proud nemá vyrovnávací paměť. To znamená, že informace jsou poslány přímo na obrazovku a nečekají na zaplnění vyrovnávací paměti nebo na znak nového řádku. Objekt `wcerr` je podobný, ale pracuje s typem `wchar_t`.
- Objekt `clog` také odpovídá standardnímu chybovému proudu. Implicitně je spojen se standardním výstupním zařízením, obvykle monitorem, a proud má vyrovnávací paměť. Objekt `wclog` je podobný, ale pracuje s typem `wchar_t`.

Co znamená, když se řekne, že objekt reprezentuje proud? Když například soubor `iostream` deklaruje v programu objekt `cout`, bude mít tento objekt datové položky s informacemi souvisejícími s výstupem, například šířku polí pro zobrazení dat, počet desetinných míst, číselný základ pro zobrazení celých čísel a adresu objektu `streambuf`, popisující vyrovnávací paměť pro zpracování výstupu. Příkaz

```
cout << "Bjarne free";
```

vloží znaky z řetězce „Bjarne free" pomocí ukazatele na objekt streambuf do vyrovnávací paměti spravované objektem cout. Třída ostream definuje funkci operator<<() použitou v tomto příkazu a také podporuje datové položky objektu cout s celou řadou metod jiných tříd, z nichž některé budou probrány později v této kapitole. C++ se dále stará, aby byl výstup z vyrovnávací paměti směřován na standardní výstup operačního systému (obvykle monitor). Stručně řečeno, jedna strana proudu je spojena s programem, druhá se standardním výstupem a objekt cout, s pomocí typového objektu streambuf, řídí tok bajtů v proudu.

Přesměrování

Standardní vstupní a výstupní proudy jsou normálně spojeny s klávesnicí a obrazovkou. Ale mnoho operačních systémů včetně UNIXu a MS-DOSu podporuje přesměrování, vlastnost, která umožňuje spojení standardního vstupu a standardního výstupu změnit. Předpokládejme například, že máte v MS-DOSu spustitelný program nazvaný counter.exe, který zjistí počet znaků na vstupu a zobrazí výsledek. Ukázkový běh by mohl vypadat takto:

```
C>counter
Dobrý den
a nashledanou!
Control-Z           simulovaný konec souboru
Vstup obsahoval 24 znaků.
C>
```

Zde sloužila jako vstup klávesnice a výstup byl odeslán na obrazovku.

Pomocí přesměrování vstupu (<) a výstupu (>) můžete docílit toho, aby stejný program spočítal počet znaků v souboru oklahoma a výsledek uložil do souboru cow_cnt:

```
C>counter <oklahoma >cow_cnt
C>
```

Část příkazového řádku <oklahoma spojí standardní vstup se souborem oklahoma a způsobí, že objekt cin bude vstup číst z tohoto souboru a ne z klávesnice. Jinými slovy, operační systém změní spojení na vstupní straně vstupního proudu, zatímco strana výstupní zůstane spojena s programem. Část příkazového řádku >cow_cnt spojí standardní výstup se souborem cow_cnt a způsobí, že objekt cout bude posílat výstup do tohoto souboru místo na obrazovku. To znamená, že operační systém změní spojení na výstupní straně výstupního proudu, zatímco strana vstupní zůstane spojena s programem. Systémy DOS (verze 2.0 a novější) a UNIX rozeznávají tuto syntaxi přesměrování automaticky. (UNIX a MS-DOS 3.0 a novější povolují libovolný počet mezer mezi znaky přesměrování a názvy souborů.)

Standardní výstupní proud, reprezentovaný objektem cout, je normální kanál pro programový výstup. Standardní chybové proudy (reprezentované objekty cerr a clog) jsou určeny pro chybové zprávy programu. Implicitně jsou všechny tři posílány na monitor. Přesměrování standardního výstupu objekty cerr a clog neovlivní; jestliže budete chtít pomocí některého z těchto objektů chybovou zprávu vytisknout, program ji pošle na obrazovku bez ohledu na to, že normální výstup objektu cout je přesměrován jinam. Uvažujte například tuto část kódu:

```
if (success)
    cout << "Tady jsou dobroty!\n";
else
{
    cerr << "Stalo se neco hrozneho.\n"
    exit(1);
}
```

Jestliže nepoužijete přesměrování, budou obě zprávy poslány na obrazovku. Pokud však výstup programu přesměrujete do souboru, bude první zpráva poslána do souboru, zatímco druhá na obrazovku. Některé systémy však umožňují přesměrovat i standardní chybový proud. Například v UNIXu lze standardní chybový proud přesměrovat pomocí operátoru 2>.

Třídy istream a ostream přesměrování podporovat nemusí. V implementaci Borland C++ 3.1 je například pro přesměrování odvozena od třídy istream třída istream_withassign a objekt cin je objektem této třídy. Podobně objekt cout patří třídě ostream_withassign, která je odvozena od třídy ostream a umožňuje přesměrování výstupu. Jinak tyto standardní objekty používají stejné metody jako jejich základní třídy. Pro jednoduchost budeme objekt cin nazývat objektem třídy istream a objekt cout objektem třídy ostream.

Výstup pomocí objektu cout

Jak jsme již řekli, považuje jazyk C++ výstup za proud bajtů. (Jestliže použijete třídu wostream, mohou být tyto bajty široké 16 bitů, ale stále jsou to bajty.) Mnoho dat v programu je však uspořádáno do větších jednotek, než je jeden bajt. Typ int může být například reprezentován dvou nebo čtyřbajtovou binární hodnotou. A hodnota typu double může být reprezentována 8 bajty binárních dat. Když však proud bajtů posíláte na obrazovku, chcete, aby každý bajt reprezentoval hodnotu znaku. To znamená, že chcete-li na obrazovce vidět číslo –2.34, musíte na ni poslat pět znaků -, 2, ., 3, 4 a ne vnitřní 8 bajtovou reprezentaci této hodnoty typem float. Jedním z nejdůležitějších úkolů při práci s třídou ostream je tedy převedení numerických typů, například int nebo float, do proudu znaků, které představují hodnoty v textové formě. To znamená, že třída ostream přeloží vnitřní reprezentaci dat jako binárních bitových vzorců do výstupního proudu znakových bajtů. (Jednoho dne můžeme mít bionické implantáty umožňující interpretovat binární data přímo. Tento vývoj ponecháme jako cvičení pro čtenáře.) Pro tyto překladatelské úkoly má třída ostream několik metod. Nyní se na ně podíváme, shrneme metody používané v knize a popíšeme další, díky nimž je možné řídit vzhled výstupu dokonaleji.

Přetížený operátor <<

Velmi často jsme v této knize používali s objektem cout operátorní (insertion operator):

```
int clients = 22;
cout << clients;
```

Implicitně označuje tento operátor v C++, stejně jako v C, operátor bitového posunu doleva (viz příloha E). Výraz `x<<3` znamená, že se vezme binární reprezentace `x` a všechny bity se posunou o tři jednotky doleva. Tato operace nemá samozřejmě s výstupem mnoho společného. Třída ostream však pomocí přetížení operátoru `<<` předefinovává a používá jako výstup. Takto maskovaný operátor `<<` se nazývá operátor vložení a ne operátor bitového posunu doleva. (Operátor posunu si tuto novou roli zasloužil díky svému vizuálnímu aspektu, který připomíná tok informací doleva.) Přetížený operátor vložení rozeznává všechny základní typy C++:

- `unsigned char`
- `signed char`
- `char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `float`
- `double`
- `long double`

Třída `ostream` obsahuje definici funkce `operator<<()` pro každý z výše uvedených typů. (Funkce zahrnující operátor do názvu se používají pro přetěžování operátorů, jak jsme probírali v kapitole 10.) Jestliže tedy použijete příkaz ve tvaru

```
cout << value;
```

a jestliže value je jedním z předchozích typů, může ho program v C++ porovnat s funkcí operátoru, která má odpovídající signaturu. Například výraz cout `<< 88` porovnává následující prototyp metody:

```
ostream & operator<<(int);
```

Vzpomeňte si, že tento prototyp označuje, že funkce `operator<<()` má jeden parametr typu int. Tato část porovnává číslo `88` v předchozím příkazu. Prototyp také označuje, že funkce vrací referenci na objekt třídy `ostream`. Tato vlastnost umožňuje výstup řetězit podobně jako v následujícím starém rokovém hitu:

```
cout << "I'm feeling sedimental over " << boundary << "\n";
```

Pokud jste programátory v C, kteří trpí nesčetnými typovými specifikátory `%` a problémy vzniklými vinou nestejných typů specifikátoru a hodnoty, je pro vás používání objektu cout téměř hříšně snadné.

Výstup a ukazatele

Třída `ostream` definuje funkce operátoru vkládání také pro následující typy ukazatelů:

- `const signed char *`
- `const unsigned char *`
- `const char *`
- `void *`

Nezapomeňte, že C++ vyjadřuje řetězec pomocí ukazatele na místo, kde je tento řetězec uložen. Ukazatel může být ve formátu názvu pole typu `char`, nebo explicitně ukazatel na znak nebo na řetězec. Všechny následující příkazy s `cout` tedy zobrazí řetězec:

```
char name[20] = "Dudly Diddlemore";
char * pn = "Violet D'Amore";
cout << "Nazdar!";
cout << name;
cout << pn;
```

Podle nulového ukončovacího znaku v řetězci metody poznají, kdy mají zobrazování znaků ukončit.

C++ porovnává ukazatel na libovolný jiný typ s typem `void *` a tiskne numerickou reprezentaci adresy. Chcete-li adresu řetězce, musíte ho přetypovat na jiný typ:

```
int eggs = 12;
char * amount = "tucet";
cout << &eggs;             // vytiskne adresu proměnné eggs
cout << amount;            // vytiskne řetězec "tucet"
cout << (void *) amount;   // vytiskne adresu řetězce "tucet"
```

Poznámka

Ne všechny současné implementace C++ mají prototyp s parametrem `void *`. V takovém případě musíte ukazatel přetypovat na `unsigned` nebo `unsigned long`, chcete-li vytisknout hodnotu adresy.

Řetězení výstupu

Všechna ztělesnění operátoru vložení jsou definována tak, aby vracela typ `ostream &`. Prototypy mají tedy tento tvar:

```
ostream & operator<<(type);
```

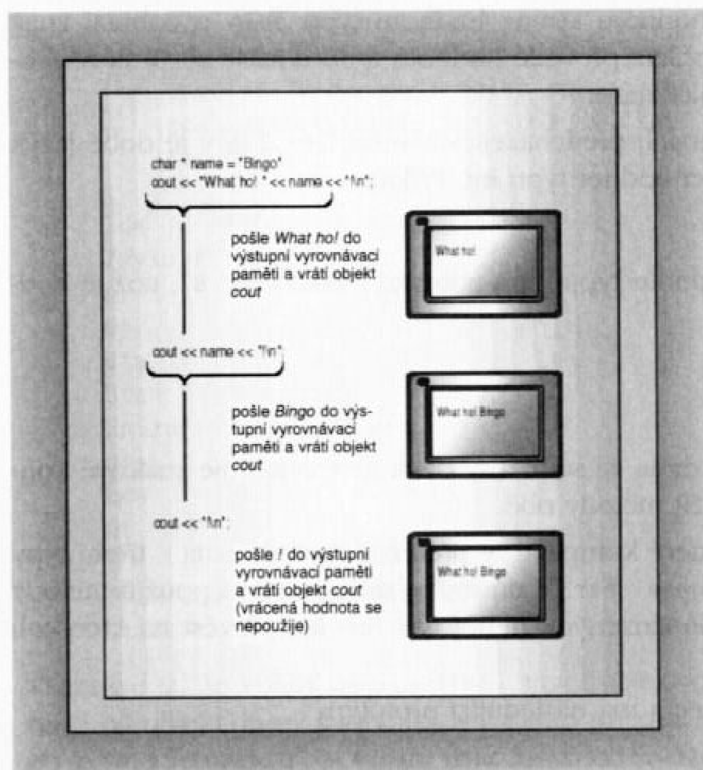
(Zde je *type* zobrazený typ.) Návratový typ `ostream &` znamená, že tento operátor vrací referenci na objekt třídy `ostream`. Na který objekt? Definice funkcí říkají, že se jedná o referenci na objekt, kterým se operátor vyvolá. Jinými slovy, návratovou hodnotou funkce operátoru je stejný objekt, který operátor vyvolal. Například výraz cout << „potluck" vrací objekt cout. Tato vlastnost vám umožňuje výstup pomocí operátoru vkládání spojovat. Uvažujte například následující příkaz:

```
cout << "Mame " << count << " nevylihnutych kurat.\n"
```

Výraz `cout << „Mame "` zobrazí řetězec a vrátí objekt `cout`, přičemž příkaz zredukuje na následujícím způsobem:

```
cout << count << " nevylihnutych kurat.\n"
```

Potom výraz `cout << count` zobrazí hodnotu proměnné `count` a vrátí objekt `cout`, který zpracuje poslední parametr příkazu (viz obrázek 16.4). Tato technika návrhu představuje příjemnou vlastnost a z tohoto důvodu jsme ji v předchozích kapitolách v příkladech s přetíženým operátorem `<<` bezostyšně napodobovali.



Obrázek 16.4 Řetězení výstupu

Ostatní metody třídy ostream

Kromě různých funkcí `operator<<()` obsahuje třída `ostream` metodu `put()`, zobrazující znaky a metodu `write()`, zobrazující řetězce.

Metoda `put()` byla zřejmě dost špatně implementována. Tradičně mívala následující prototyp:

```
ostream & put(char);
```

Současný standard je ekvivalentní s tím rozdílem, že šablona má umožnit typ `wchar_t`. Metodu vyvoláte obvyklým zápisem:

```
cout.put('W');     // zobrazí znak W
```

Zde je cout volající objekt a `put()` členskou funkcí třídy. Stejně jako funkce operátoru `<<` vrací i tato funkce referenci na volající objekt, takže výstup můžete spojit:

```
cout.put('I').put('t');    // dvojím voláním funkce put() zobrazí It
```

Funkce zavolá `cout.put('I')` a vrátí objekt `cout`, který se potom chová jako volající objekt při volání cout.put('t').

Při správném prototypu můžete funkci `put()` použít s parametry numerického typu, například `int`, a prototyp funkce parametr automaticky převede na správnou hodnotu typu `char`. Mohli byste například napsat následující příkazy:

```
cout.put(65);    // zobrazí znak A
cout.put(66.3);  // zobrazí znak B
```

První příkaz převede celočíselnou hodnotu `65` na hodnotu typu `char` a zobrazí znak s ASCII-kódem `65`. Podobně druhý příkaz převede hodnotu typu double `66.3` na znakovou hodnotu `66` a zobrazí odpovídající znak.

Takové chování bylo užitečné ve verzích předcházejících verzi C++ 2.0; v té době jazyk vyjadřoval znakové konstanty pomocí hodnot typu int. Příkaz

```
cout << 'W';
```

tedy interpretoval znak `'W'` jako hodnotu typu int a zobrazil ji jako číslo `87`, což je hodnota znaku v tabulce ASCII.

Příkaz

```
cout.put('W');
```

však fungoval správně. Vzhledem k tomu, že současný jazyk C++ vyjadřuje znakové konstanty typem `char`, můžete nyní použít metody obě.

Problémem implementace je, že některé kompilátory přetěžují metodu `put()` třemi typy parametrů: `char`, unsigned `char` a `signed char`. V důsledku toho se stává použití metody `put()` s parametrem typu `int` nejednoznačným, neboť typ `int` lze převést na kterýkoli z uvedených tří typů.

Metoda `writes()` vypisuje celý řetězec a má následující prototyp:

```
basic_ostream<charT, traits>& write(const char_type* s, streamsize n);
```

První parametr dodává adresu zobrazovaného řetězce a druhý počet zobrazovaných znaků. Vyvoláte-li metodu `writes()` pomocí objektu cout, vyvoláte specializaci typu `char` a návratovým typem tedy bude `ostream &`. Způsob práce metody `writes()` ukazuje program ve výpisu 16.1.

Výpis 16.1 write.cpp

```
// write.cpp - použití metody cout.write()
#include <iostream>
using namespace std;
#include <cstring>  // nebo string.h
int main()
{
    const char * state1 = "Ohio";
    const char * state2 = "Utah";
    const char * state3 = "Euphoria";
```

```
    int len = strlen(state2);
    cout << "Zvysuji index cyklu:\n";
    int i;
    for (i = 1; i <= len; i++)
    {
        cout.write(state2,i);
        cout << "\n";
    }
// spojení výstupu
    cout << "Snizuji index cyklu:\n";
    for (i = len; i > 0; i--)
        cout.write(state2,i) << "\n";
// překročení délky řetězce
    cout << "Prekracuji delku retezce:\n";
    cout.write(state2, len + 5) << "\n";
    return 0;
}
```

```
Zde je výstup:
Zvysuji index cyklu:
U
Ut
Uta
Utah
Snizuji index cyklu:
Utah
Uta
Ut
U
Prekracuji delku retezce:
Utah    0
```

Všimněte si, že volání `cout.write()` vrací objekt `cout`. Metoda `write()` totiž vrací referenci na objekt, který ji vyvolal, a v tomto případě ji vyvolal objekt `cout`. Tím je umožněno výstup řetězit, neboť volání `cout.write()` je nahrazeno svou návratovou hodnotou – objektem `cout`:

```
cout.write(state2, i) << "\n";
```

Všimněte si také, že metoda `write()` neukončí tisk znaků automaticky, když narazí na nulový znak. Tiskne prostě tolik znaků, kolik jí řeknete, i když tento počet přesahuje hranice určitého řetězce! V takovém případě program sloučí řetězec „`Utah`" s ostatními dvěma řetězci, takže přilehlá paměť bude obsahovat data. V pořadí ukládání dat do paměti a ve způsobu zarovnávání paměti se kompilátory liší. Například řetězec „Utah" obsadí pět bajtů, ale náš konkrétní kompilátor řetězce zarovnává na násobky čtyř bajtů, takže „Utah" je doplněn na osm bajtů.

Metodu `write()` lze použít také u numerických dat. Nepřevede číslo na správné znaky, ale přenese bitovou reprezentaci tak, jak je uložena v paměti. Například čtyřbajtovou hodnotu 560031841 typu `long` by přenesla jako čtyři samostatné bajty. Výstupní zařízení jako monitor by se potom snažilo interpretovat každý bajt jako kód ASCII. Číslo 560031841 by se tedy na obrazovce objevilo jako nějaká čtyřznaková kombinace, pravděpodobně nesmyslná. Metoda `write()` však představuje kompaktní, přesný způsob uložení dat do souboru. K této možnosti se vrátíme v kapitole později.

Vyprázdnění výstupní vyrovnávací paměti

Uvažujte, co se děje, když program pomocí objektu `cout` pošle bajty na standardní výstup. Vzhledem k tomu, že třída `ostream` používá pro výstup pomocí objektu `cout` vyrovnávací paměť, není výstup poslán na místo určení přímo. Místo toho se ve vyrovnávací paměti hromadí do té doby, dokud se vyrovnávací paměť nezaplní. Potom ji program vyprázdní, odešle obsah a paměť vyčistí, aby byla připravena pro nová data. Velikost vyrovnávací paměti je obvykle 512 bajtů nebo celočíselný násobek této hodnoty. Pokud je standardní výstup spojen se souborem na pevném disku, ušetří používání vyrovnávací paměti spoustu času. Nechcete přece, aby program při posílání 512 bajtů musel 512krát přistupovat na pevný disk. Mnohem efektivnější je načíst 512 bajtů do vyrovnávací paměti a potom je zapsat na pevný disk jedinou operací.

Při výstupu na obrazovku však zaplnění vyrovnávací paměti není tak důležité. Určitě by nebylo příjemné, kdybyste zprávu „Stiskněte libovolnou klávesu" museli přeformulovat tak, aby obsahovala předepsaných 512 bajtů a zaplnila vyrovnávací paměť. Naštěstí v případě výstupu na obrazovku program na zaplnění vyrovnávací paměti čekat nemusí. Pošlete-li do vyrovnávací paměti například znak nového řádku, stačí to normálně k jejímu vyprázdnění. Většina implementací vyrovnávací paměť vyprázdní také při nevyřízeném vstupu, jak jsme se již dříve zmínili. Předpokládejme například následující kód:

```
cout << "Zadejte cislo: ";
float num;
cin >> num;
```

Skutečnost, že program očekává vstup způsobí, že zobrazí zprávu objektu `cout` (to znamená, že vyprázdní zprávu „`Zadejte cislo: `") ihned, přestože ve výstupním řetězci chybí znak nového řádku. Bez této vlastnosti by program čekal na vstup a zprávu objektu cout by uživateli neposlal.

Jestliže vaše implementace výstup, nevyprázdní tehdy, když to potřebujete, můžete vyprázdnění vynutit pomocí jednoho ze dvou manipulátorů. Manipulátor `flush` vyprázdní vyrovnávací paměť a manipulátor `endl` vyprázdní vyrovnávací paměť a vloží znak nového řádku. Tyto manipulátory se používají stejně jako názvy proměnných:

```
cout << "Ahoj Pepo! " << flush;
cout << "Pockejte chvili, prosim." << endl;
```

Manipulátory jsou vlastně funkce. Například vyrovnávací paměť objektu `cout` můžete vyprázdnit přímo voláním funkce `flush()`:

```
flush(cout);
```

Třída `ostream` však přetěžuje operátor vkládání `<<` takovým způsobem, že se výraz

```
cout << flush
```

nahradí voláním funkce `flush(cout)`. Pro úspěšné vyprázdnění vyrovnávací paměti tedy můžete použít pohodlnější zápis s operátorem vkládání.

Formátování pomocí objektu cout

Operátory vkládání třídy `ostream` převádí hodnoty do textového tvaru. Implicitně se hodnoty formátují následujícím způsobem:

- Hodnota typu `char`, pokud představuje tisknutelný znak, se zobrazí jako znak v poli o šířce jednoho znaku.
- Numerické celočíselné typy se zobrazí jako celá desítková čísla v poli, jehož šířka odpovídá danému číslu a případnému znaku mínus.
- Řetězce jsou zobrazeny v poli, jehož šířka se rovná délce řetězce.

Implicitní chování typu `float` se změnilo. V následujícím seznamu jsou podrobně uvedeny rozdíly mezi staršími a novějšími implementacemi:

- (Původní styl) Typy `float` jsou zobrazeny na šest desetinných míst bez koncových nul. (Všimněte si, že počet číslic nesouvisí s přesností, s jakou je číslo uloženo.) Číslo se v závislosti na hodnotě zobrazí ve tvaru s pevnou desetinnou tečkou nebo v semilogaritmickém tvaru (viz kapitola 3). Šířka pole opět odpovídá šířce čísla s případným znakem mínus.
- (Nový styl) Typy `float` jsou zobrazeny celkem na šest číslic bez koncových nul. (Všimněte si, že počet číslic nesouvisí s přesností, s jakou je číslo uloženo.) Číslo se v závislosti na hodnotě zobrazí ve tvaru s pevnou desetinnou tečkou nebo v semilogaritmickém tvaru (viz kapitola 3). Zápis v semilogaritmickém tvaru se použije, jestliže je exponent větší nebo roven 6, nebo jestliže je menší než 5. Šířka pole opět odpovídá šířce čísla s případným znakem mínus. Implicitní chování odpovídá použití funkce `fprintf()` se specifikátorem `%g`. Tato funkce je součástí standardní knihovny jazyka C.

Vzhledem k tomu, že každá hodnota je zobrazena v šířce odpovídající své velikosti, musíte mezery mezi hodnotami vytvořit explicitně; v opačném případě sousední hodnoty splynou.

Mezi formátováním v dřívějších verzích a současným standardem je několik malých rozdílů; jejich souhrn naleznete v této kapitole později v tabulce 16.3.

Program ve výpisu 16.2 ilustruje implicitní vlastnosti výstupu. Abyste viděli u každého případu šířku pole, zobrazuje za každou hodnotou dvojtečku. Pomocí výrazu 1.0/9.0 generuje nekončící část, takže vidíte počet tisknutelných míst.

Kompatibilita:

Ne všechny kompilátory generují formátovaný výstup v souladu se současným standardem. Současný standard také umožňuje regionální varianty. Například evropská implementace používá pro oddělení desetinné části čárku místo tečky. To znamená, že je možné psát `2,54` místo `2.54`. Lokální knihovna (hlavičkový soubor `locale`) poskytuje mechanismus, s jehož pomocí se ve vstupním nebo výstupním proudu nastaví určitý styl. Jednotlivé kompilátory tedy mohou nabízet víc než jedno místní nastavení. V této kapitole budeme používat místní nastavení typické pro Spojené státy.

Výpis 16.2 defaults.cpp

```
// defaults.cpp - standardní formáty objektu cout
#include <iostream>
using namespace std;
int main()
{
    cout << "12345678901234567890\n";
    char ch = 'K';
    int t = 273;
    cout << ch << ":\n";
    cout << t << ":\n";
    cout << -t <<":\n";
    double f1 = 1.200;
    cout << f1 << ":\n";
    cout << (f1 + 1.0 / 9.0) << ":\n";
    double f2 = 1.67E2;
    cout << f2 <<":\n";
    f2 += 1.0 / 9.0;
    cout << f2 << ":\n";
    cout << (f2 * 1.0e4) << ":\n";

    double f3 = 2.3e-4;
    cout << f3 << ":\n";
    cout << f3 / 10 << ":\n";
    return 0;
}
```

Zde je výstup:

```
12345678901234567890:
K:
273:
-273:
1.2:
1.31111:
167:
167.111:
1.67111e+006:
0.00023:
2.3e-005:
```

Každá hodnota vyplní pole. Všimněte si, že koncové nuly u čísla 1.200 zobrazeny nejsou, ale že hodnoty s pohyblivou řádovou čárkou bez koncových nul jsou zobrazeny na šest míst vpravo od desetinného oddělovače. V exponentu zobrazuje tato konkrétní implementace tři číslice.

Změna číselného základu používaného při zobrazení

Třída `ostream` dědí od třídy `ios` a ta zase od třídy `ios_base`. Třída `ios_base` ukládá informace popisující stav formátu. Například určité bity v jedné položce třídy určují použitý číselný základ, zatímco druhá položka určuje šířku pole. Číselný základ použitý při zobra-

zení můžete ovládat pomocí manipulátorů. Pomocí členských funkcí třídy `ios_base` můžete ovládat šířku pole a počet zobrazených desetinných míst. Protože třída `ios_base` je nepřímou základní třídou třídy `ostream`, můžete u objektů třídy `ostream` (například `cout`) použít její metody.

Poznámka

Členské proměnné a metody obsažené v třídě `ios_base` se dříve nacházely v třídě `ios`. Nyní je třída `ios_base` základní třídou třídy `ios`. V novém systému je `ios` šablonová třída se specializacemi typu `char` a `wchar_t`, zatímco `ios_base` má vlastnosti nešablonové.

Podívejme se, jak se číselný základ pro zobrazení celých čísel nastaví. Chcete-li určit, mají-li být čísla zobrazena v desítkové, šestnáctkové nebo osmičkové soustavě, můžete použít manipulátory `dec`, `hex` a `oct`. Například volání funkce

```
hex(cout);
```

nastaví pro objekt `cout` šestnáctkový stav formátu číselné soustavy. Po tomto příkazu bude program tisknout hodnoty celých čísel v šestnáctkovém tvaru do té doby, než stav formátu nastavíte jinak. Všimněte si, že manipulátory nejsou členskými funkcemi a nemusejí tedy být vyvolány nějakým objektem.

Ačkoli jsou manipulátory skutečnými funkcemi, obvykle se používají následujícím způsobem:

```
cout << hex;
```

Třída ostream přetěžuje operátor << takovým způsobem, aby byl uvedený příkaz rovnocenný volání funkce `hex(cout)`. Použití těchto manipulátorů ilustruje program ve výpisu 16.3. Zobrazuje celočíselnou hodnotu a její druhou mocninu ve třech různých číselných soustavách. Všimněte si, že manipulátor můžete použít samostatně nebo jako součást řady vložení.

Výpis 16.3 manip.cpp

```
// manip.cpp - formátování pomocí manipulátorů
#include <iostream>
using namespace std;
int main(){
    cout << "Zadejte cele cislo: ";
    int n;
    cin >> n;

    cout << "n     n*n\n";
    cout << n << "     " << n * n << " (desitkove)\n";
// nastavení šestnáctkového režimu
    cout << hex;
    cout << n << "     ";
    cout << n * n << " (sestnactkove)\n";
```

```
// nastavení osmičkového režimu
    cout << oct << n << "          " << n * n << " (osmickove)\n";
// alternativní způsob volání manipulátoru
    dec(cout);
    cout << n << "          " << n * n << " (desitkove)\n";
    return 0;
}
```

Zde je ukázka výstupu:

```
Zadejte cele cislo: 13
n n*n
13 169 (desitkove)        d a9 (sestnactkove)
15 251 (osmickove)
13 169 (desitkove)
```

Úprava šířky polí

Pravděpodobně jste si všimli, že sloupce v předchozím příkladě nejsou zarovnané; je to tím, že čísla mají různou šířku pole. Pomocí členské funkce `width()` můžete různě široká čísla vložit do polí o stejné šířce. Metoda má tyto prototypy:

```
int width();
int width(int i);
```

První tvar vrátí aktuální nastavení šířky pole. Druhý nastaví šířku pole na i mezer a vrátí předchozí hodnotu šířky pole. Tu můžete uložit pro případ, že byste později chtěli šířku opět nastavit na tuto hodnotu.

Metoda `width()` ovlivní pouze následující zobrazenou položku, potom se šířka pole vrátí na implicitní hodnotu. Uvažujte například následující příkazy:

```
cout << '#';
cout.width(12);
cout << 12 << "#" << 24 << "#\n";
```

Protože `width()` je funkcí členskou, musíte pro její vyvolání použít objekt (v tomto případě `cout`). Výstupní příkaz zobrazí následující:

```
#          12#24#
```

Číslo `12` je vloženo do pole o šířce 12 znaků na pravou stranu. Tomuto se říká zarovnání vpravo. Potom se pole vrátí k implicitní hodnotě a oba znaky # a číslo `24` jsou vytištěny v polích odpovídající šířky.

Pamatujte

Metoda `width()` ovlivní pouze následující zobrazenou položku, potom se šířka pole vrátí na implicitní hodnotu.

Jazyk C++ data nikdy neořezává. Pokud se tedy budete snažit vytisknout sedmimístnou hodnotu do pole o velikosti 2 znaky, C++ pole rozšíří podle potřeby dat. (Některé jazyky pole pouze vyplní hvězdičkami, pokud se data do něj nevejdou. Filozofie jazyků

C a C++ je taková, že zobrazit data je důležitější než zachovat úhlednost sloupců; jazyk C++ upřednostňuje látku před formou.) Práci členské funkce `width()` představuje program ve výpisu 16.4.

Výpis 16.4 width.cpp

```
// width.cpp - použití metody width()
#include <iostream>
using namespace std;

int main(){
    int w = cout.width(30);
    cout << "implicitni sirka pole = " << w << ":\n";

    cout.width(5);
    cout << "N" <<':';
    cout.width(8);
    cout << "N * N" << ":\n";

    for (long i = 1; i <= 100; i *= 10)
    {
        cout.width(5);
        cout << i <<':';
        cout.width(8);
        cout << i * i << ":\n";
    }
    return 0;
}
```

Zde je výstup:

```
implicitni sirka pole = 0:
    N:   N * N:
    1:       1:
   10:     100:
  100:    1000:
```

Výstup zobrazí hodnoty zarovnané v polích vpravo a je doplněn mezerami. To znamená, že objekt `cout` dosáhne plné šířky pole přidáním mezer. Při zarovnání vpravo jsou mezery vloženy vlevo od hodnoty. Znaku sloužícímu pro doplnění se říká *výplňový znak*. Zarovnání vpravo je implicitní.

Všimněte si, že program použije pole o šířce 30 znaků na řetězec zobrazený prvním příkazem `cout`, ale již ne na hodnotu `w`. Metoda `width()` totiž ovlivní pouze jednu následující zobrazenou položku. Také si všimněte, že hodnota `w` je 0. Volání `cout.width(30)` vrátí předchozí šířku pole a ne tu, která byla právě nastavena. Protože C++ pole vždy upraví podle velikosti dat, pojme tato velikost všechna data. Nakonec program použije metodu `width()` pro zarovnání záhlaví sloupců a dat. Šířku prvního sloupce nastaví na pět znaků a šířku druhého na osm.

Výplňové znaky

Implicitně doplní objekt cout nevyužité části pole mezerami. Chcete-li mezery nahradit jiným znakem, můžete použít členskou funkci fill(). Například volání

```
cout.fill('*');
```

změní výplňový znak na hvězdičku. To se může hodit například při tisku šeků, aby příjemce nemohl nějakou číslici doplnit. Použití této členské funkce předvádí program ve výpisu 16.5.

Výpis 16.5 fill.cpp

```
// fill.cpp - změna výplňového znaku polí
#include <iostream>
using namespace std;
int main(){
    cout.fill('*');
    char * staff[2] = { "Waldo Whipsnade",  "Wilmarie Wooper"};
    long bonus[2] = {900, 1350};
    for (int i = 0; i < 2; i++)
    {
        cout << staff[i] << ": $";
        cout.width(7);
        cout << bonus[i] << "\n";
    }
    return 0;
}
```

Zde je výstup:

```
Waldo Whipsnade: $****900
Wilmarie Wooper: $***1350
```

Všimněte si, že na rozdíl od šířky polí výplňový znak zůstává v platnosti až do doby, než ho změníte.

Nastavení přesnosti zobrazování hodnoty s pohyblivou řádovou čárkou

Přesnost zobrazení hodnoty s pohyblivou řádovou čárkou závisí na výstupním režimu. V implicitním režimu znamená celkový počet zobrazených číslic. V pevných a vědeckých režimech, kterými se budeme brzy zabývat, přesnost znamená počet číslic zobrazených vpravo od desetinného oddělovače. Jak jste viděli, implicitní přesnost v C++ je 6 číslic. (Pamatujte však, že koncové nuly se nezobrazují.) Členská funkce precision() umožňuje zvolit jinou hodnotu. Například příkazem

```
cout.precision(2);
```

nastavíte přesnost v objektu cout na 2. Na rozdíl od šířky výstupního pole zůstává nová přesnost v platnosti až do opětovného nastavení, stejně jako výplňový znak. Tuto vlastnost demonstruje program ve výpisu 16.6.

Výpis 16.6 precise.cpp

```
// precise.cpp – nastavení přesnosti
#include <iostream>
using namespace std;
int main()
{
    float price1 = 20.40;
    float price2 = 1.9 + 8.0 / 9.0;
    cout << "Prvni vec stoji Kc" << price1 << "!\n";
    cout << "Druha vec stoji Kc" << price2 << "!\n";
    cout.precision(2);
    cout << "Prvni vec stoji Kc" << price1 << "!\n";
    cout << "Druha vec stoji Kc" << price2 << "!\n";
    return 0;
}
```

Kompatibilita:

Starší verze C++ implementují přesnost pro implicitní režim jako počet číslic vpravo od desetinného oddělovače a ne jako celkový počet číslic.

Zde je výstup:

```
Prvni vec stoji Kc20.4!
Druha vec stoji Kc2.78889!
Prvni vec stoji Kc20!
Druha vec stoji Kc2.8!
```

Všimněte si, že ve třetím řádku se nevytiskla koncová desetinná tečka a že čtvrtý řádek obsahuje celkem dvě číslice.

Tisk koncových nul a desetinných oddělovačů

Některé formáty výstupu, například ceny nebo čísla ve sloupcích, vypadají lépe s koncovými nulami. Například výstup z programu ve výpisu 16.6 by vypadal lépe, kdyby místo Kc20.4 bylo zobrazeno Kc20.40. Skupina tříd `iostream` neobsahuje funkci, jejímž jediným účelem by bylo starat se o zobrazování. Avšak třída `ios_base` obsahuje funkci `setf()` (pro *set flag*), která ovládá několik formátovacích způsobů. Definuje také několik konstant, které lze použít jako její parametry. Například volání funkce

```
cout.setf(ios_base::showpoint);
```

přinutí objekt cout zobrazovat desetinné oddělovače. V dřívějších verzích by toto volání také způsobilo zobrazení koncových nul. To znamená, že objekt `cout` by číslo 2.00 nezobrazil jako 2, ale jako 2.000000 (původní formátování) nebo jako 2. (současné formátování), pokud by byla nastavena implicitní přesnost na 6 číslic. Výše uvedený příkaz byl přidán do programu ve výpisu 16.7.

Upozornění

Jestliže váš kompilátor používá místo hlavičkového souboru `iostream` soubor `iostream.h`, budete pravděpodobně jako parametr ve funkci `setf()` muset použít místo třídy `ios_base` třídu `ios`.

Jestliže se divíte zápisu `ios_base::showpoint`, vězte, že `showpoint` je statická konstanta s platností v rozsahu třídy definovaná v deklaraci třídy `ios_base`. Platnost v rozsahu třídy znamená, že název konstanty musíte použít s operátorem rozlišení (`::`), jestliže ho používáte mimo definici členské funkce. Výraz `ios_base::showpoint` tedy pojmenovává konstantu definovanou v třídě `ios_base`.

Výpis 16.7 showpt.cpp

```
// showpt.cpp - nastavení přesnosti, zobrazení desetinného oddělovače
#include <iostream>
using namespace std;
int main(){
    float price1 = 20.40;
    float price2 = 1.9 + 8.0 / 9.0;
    cout.setf(ios_base::showpoint);
    cout << "Prvni vec stoji Kc" << price1 << "!\n";
    cout << "Druha vec stoji Kc" << price2 << "!\n";

    cout.precision(2);
    cout << "Prvni vec stoji Kc" << price1 << "!\n";
    cout << "Druha vec stoji Kc" << price2 << "!\n";
    return 0;
}
```

Zde je výstup při použití aktuálního formátování. Všimněte si, že koncové nuly zobrazeny nejsou, ale koncový desetinný oddělovač ve třetím řádku zobrazen je.

```
Prvni vec stoji Kc20.4!
Druha vec stoji Kc2.78889!
Prvni vec stoji Kc20.!
Druha vec stoji Kc2.8!
```

A jak tedy můžete zobrazit koncové nuly? Abychom mohli odpovědět na tuto otázku, musíme probrat funkci setf() podrobněji.

Další informace o metodě `setf()`

Metoda `setf()` ovládá několik formátovacích způsobů, a proto se na ni podíváme podrobněji. Třída `ios_base` obsahuje chráněnou datovou položku, ve které jednotlivé bity (v tomto kontextu se nazývají *příznaky*, flags) řídí různé formátovací aspekty jako je číselný základ nebo zda se mají zobrazit koncové nuly. Zapnutí příznaku se nazývá *nastavení příznaku* (setting the flag) a znamená nastavení bitu na hodnotu 1. (Jestliže jste museli při konfiguraci počítačového hardwaru nastavovat přepínače DIP, vězte, že bitové příznaky jsou programovacím ekvivalentem.) Manipulátory `hex`, `dec` a `oct` například upravují tři příznakové bity,

které ovládají číselný základ. Funkce `setf()` nabízí další prostředky pro úpravu příznakových bitů.

Funkce `setf()` má dva prototypy. První je tento:

```
fmtflags setf(fmtflags);
```

Zde je `fmtflags` název definovaný příkazem `typedef` pro typ *bitmask* (viz poznámka), který obsahuje formátovací příznaky. Název je definován ve třídě `ios_base`. Tato verze metody `setf()` se používá pro nastavení formátovacích informací řízených jediným bitem. Parametrem je hodnota `fmtflags` označující bit, který se má nastavit. Návratovou hodnotou je číslo typu `fmtflags`, označující dřívější nastavení všech příznaků. Tuto hodnotu můžete uložit pro případ, že byste chtěli později obnovit původní nastavení. Jakou hodnotu předáváte metodě `setf()`? Jestliže chcete nastavit bit č. 11 na 1, předáte číslo, které tento bit nastaví. Návratová hodnota by bitu č.11 přiřadila předchozí hodnotu. Evidování bitů vypadá nudně (a je nudné). Tuto práci však dělat nemusíte; třída `ios_base` definuje konstanty, které bitové hodnoty reprezentují. Některé z těchto definic najdete v tabulce 16.1.

Poznámka

Typ bitmask se používá pro uložení jednotlivých hodnot bitů. Může jím být typ `celočíselný`, `výčtový` nebo kontejner `bitset` z knihovny STL. Smyslem je, aby každý bit byl jednotlivě dostupný a měl svůj vlastní význam. Balík `iostream` používá typy bitmask pro uložení informací o stavu.

Tabulka 16.1 Formátovací konstanty.

Konstanta	Význam
ios_base::boolapha	Boolovské vstupní a výstupní hodnoty zobrazí jako `true` a `false`
ios_base::showbase	Pro výstup v šestnáctkové soustavě použije předponu `0x`
ios_base::showpoint	Zobrazí koncový desetinný oddělovač
ios_base::uppercase	Použije pro výstup v šestnáctkové soustavě a semilogaritmickém tvaru velká písmena
ios_base::showpos	Před kladnými čísly použije znak +

Vzhledem k tomu, že tyto formátovací konstanty jsou definovány ve třídě `ios_base`, musíte je používat s rozlišovacím operátorem – tedy `ios_base::uppercase` a ne pouze `uppercase`. Změny zůstanou v platnosti až do chvíle, než je přepíšete. Činnost některých konstant ilustruje program ve výpisu 16.8.

Výpis 16.8 setf.cpp

```
// setf.cpp – řízení formátování pomocí funkce setf()
#include <iostream>
using namespace std;
int main()
```

```
{
    int temperature = 26;
    cout << "Dnesni teplota vody: ";
    cout.setf(ios_base::showpos); // zobrazí znaménko plus
    cout << temperature << "\n";
    cout << "Pro nase programovaci pratele to je \n";
    cout << hex << temperature << "\n"; // použije šestnáctkovou soustavu
    cout.setf(ios_base::uppercase);// použije v šestnáctkové soustavě
                                   // velká písmena
    cout.setf(ios_base::showbase); // použije v šestnáctkové soustavě
                                   // předponu 0X
    cout << "nebo\n";
    cout << temperature << "\n";
    cout << "Skutecne " << true << "!  vlastne - Skutecne ";
    cout.setf(ios_base::boolalpha);
    cout << true << "!\n";

    return 0;
}
```

Kompatibilita:

Starší implementace mohou místo třídy `ios_base` používat třídu `ios` a nemusejí obsahovat volbu `boolapha`.

Zde je výstup:

```
Dnesni teplota vody: +26
Pro nase programovaci pratele to je
1a
or
0X1A
Skutecne 1!  vlastne - Skutecne true!
```

Všimněte si, že znaménko plus se používá pouze při zobrazení v desítkové soustavě. Hodnoty v šestnáctkové a osmičkové soustavě znaménko nemají a není tedy při zobrazování potřeba. (Některé implementace však přesto mohou znaménko plus zobrazovat.)

Druhý prototyp metody `setf()` má dva parametry a vrací předchozí nastavení:

```
fmtflags setf(fmtflags, fmtflags);
```

Tento přetížený tvar funkce se používá pro formátování ovládané více než jedním bitem. Prvním parametrem je stejně jako v předchozí verzi hodnota typu `fmtflags`, obsahující požadované nastavení. Druhým parametrem je hodnota, která příslušné bity nejdříve vyčistí. Předpokládejme například, že bit č. 3 nastavený na hodnotu 1 znamená desítkovou číselnou soustavu, nastavený bit č. 4 soustavu osmičkovou a nastavený bit č. 5 sestavu šestnáctkovou. Předpokládejme, že výstup je v soustavě desítkové a vy chcete nastavit šestnáctkovou. Nejenže budete muset nastavit bit č. 5 na hodnotu 1, ale také budete muset bit č. 3 nastavit na hodnotu 0 – tomuto se říká ***čištění bitu***. Inteligentní manipulátor hex provádí oba úkoly automaticky. Práce s metodou `setf()` je trochu nároč-

nější, protože musíte pomocí druhého parametru označit, které bity chcete vyčistit a pomocí prvního parametru označit bit, který chcete nastavit. Není to tak komplikované, jak to vypadá, neboť třída `ios_base` definuje pro tento účel konstanty (viz tabulka 16.2). Při této konkrétní změně číselné soustavy byste použili jako druhý parametr konstantu ios_base::basefield a jako parametr první konstantu `ios_base::hex`. Volání funkce

```
cout.setf(ios_base::hex, ios_base::basefield);
```

má tedy stejný účinek jako použití manipulátoru hex.

Tabulka 16.2 Parametry metody setf(long, long).

Druhý parametr	První parametr	Význam
ios_base::basefield	ios_base::dec	Číselný základ 10
	ios_base::oct	Číselný základ 8
	ios_base::hex	Číselný základ 16
ios_base::floatfield	ios_base::fixed	Zápis s desetinným oddělovačem
	ios_base::scientific	Vědecký zápis
ios_base::adjustfield	ios_base::left	Zarovnání vlevo
	ios_base::right	Zarovnání vpravo
	ios_base::internal	Znaménko nebo předpona základu zarovnány vlevo, hodnota vpravo

Třída `ios_base` definuje tři sady formátovacích příznaků, které lze tímto způsobem použít. Každá se skládá z jedné konstanty sloužící jako druhý parametr, a dvou až tří konstant, které se používají jako parametr první. Druhý parametr vyčistí skupinu příbuzných bitů a potom první parametr nastaví jeden z nich na hodnotu 1. V tabulce 16.2 jsou názvy konstant používaných v metodě `setf()` jako druhé parametry, konstanty pro první parametr, které jsou s nimi spojeny, a jejich význam. Chcete-li například zarovnání vlevo, použijte jako druhý parametr `ios_base::adjustfield` a `ios_base::left` jako parametr první. Zarovnání vlevo znamená, že hodnota bude zarovnána k levému okraji pole, zatímco při zarovnání vpravo bude zarovnána k okraji pravému. Vnitřní zarovnání znamená umístění znamének nebo předpon číselných základů na levou stanu pole a zbytek čísla na stranu pravou. (Jazyk C++ bohužel nemá režim pro samozarovnávání.)

Zápis s pevným oddělovačem znamená použít pro hodnoty typu `float` styl 123.4 bez ohledu na velikost čísla, zatímco při zápisu vědeckém se bez ohledu na velikost čísla používá styl 1.23e04.

Ve standardní verzi se vyznačuje zápis s pevným oddělovačem i zápis vědecký následujícími dvěma vlastnosti:

- Přesnost znamená spíše počet číslic vpravo od desetinného oddělovače než jejich celkový počet.
- Jsou zobrazeny koncové nuly.

Ve starších verzích je pro zobrazení koncových nul nutné nastavit `ios:showpoint` a přesnost vždy znamenala (i v implicitním režimu) počet číslic vpravo od desetinného oddělovače.

Funkce `setf()` je členskou funkcí třídy `ios_base`. Vzhledem k tomu, že se jedná o základní třídu třídy `ostream`, můžete funkci vyvolat pomocí objektu `cout`. Chcete-li například zarovnání vlevo, použijte toto volání:

```
ios_base::fmtflags old = cout.setf(ios::left, ios::adjustfield);
```

Předchozí nastavení obnovíte následujícím způsobem:

```
cout.setf(old, ios::adjustfield);
```

Další příklady použití funkce `setf()` se dvěma parametry ilustruje program ve výpisu 16.9.

Kompatibilita:

Tento program používá matematickou funkci, ale některé systémy C++ knihovnu `math` automaticky neprohledávají. Například v systémech UNIX musíte vložit následující řádek:

```
$ CC setf2.C -lm
```

Volba `-lm` informuje linker, že má prohledávat knihovnu `math`.

Výpis 16.9 setf2.cpp

```
// setf2.cpp – řízení formátování pomocí funkce setf() se dvěma parametry
#include <iostream>
using namespace std;
#include <cmath>
int main()
{
    // použití zarovnání vlevo, zobrazení znaménka plus, zobrazení koncových
    // nul, přesnost na 3 místa
    cout.setf(ios_base::left, ios_base::adjustfield);
    cout.setf(ios_base::showpos);
    cout.setf(ios_base::showpoint);
    cout.precision(3);
    // použití e-notace a uložení původního nastavení formátování
    ios_base::fmtflags old = cout.setf(ios_base::scientific,
        ios_base::floatfield);
    cout << "Zarovnani vlevo:\n";
    long n;
    for (n = 1; n <= 41; n+= 10)
    {
            cout.width(4);
            cout << n << "|";
            cout.width(12);
            cout << sqrt(n) << "|\n";
    }
    // změna na vnitřní zarovnání
    cout.setf(ios_base::internal, ios_base::adjustfield);
    // obnovení implicitního stylu zobrazení typu float
    cout.setf(old, ios_base::floatfield);
    cout << "Vnitrni zarovnani:\n";
```

```
    for (n = 1; n <= 41; n+= 10)
    {
        cout.width(4);
        cout << n << "|";
        cout.width(12);
        cout << sqrt(n) << "|\n";
    }
    // použití zarovnání vpravo, zápis s pevným oddělovačem
    // cout.setf(ios_base::right, ios_base::adjustfield);
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Zarovnani vpravo:\n";
    for (n = 1; n <= 41; n+= 10)
    {
        cout.width(4);
        cout << n << "|";
        cout.width(12);
        cout << sqrt(n) << "|\n";
    }

    return 0;
}
```

Zde je výstup:

```
Zarovnani vlevo:
+1  |+1.000e+000 |
+11 |+3.317e+000 |
+21 |+4.583e+000 |
+31 |+5.586e+000 |
+41 |+6.403e+000 |
Vnitrni zarovnani:
+  1|+        1.00|
+ 11|+        3.32|
+ 21|+        4.58|
+ 31|+        5.57|
+ 41|+        6.40|
Zarovnani vpravo:
  +1|      +1.000|
 +11|      +3.317|
 +21|      +4.583|
 +31|      +5.568|
 +41|      +6.403|
```

Všimněte si, že nastavení přesnosti na 3 místa způsobí, že typ s pohyblivou řádovou čárkou se implicitně zobrazí na tři číslice celkem (v tomto programu použito při vnitřním zarovnání), zatímco v režimu s pevným oddělovačem a při vědeckém zápisu se zobrazí tři číslice vpravo od desetinného oddělovače.

Účinky volání metody `setf()` lze zrušit metodou `unsetf()`, která má následující prototyp:

```
void unsetf(fmtflags mask);
```

Zde mask označuje vzorek bitů. Všechny bity nastavené v mask na hodnotu 1 budou nastaveny na hodnotu 0. Metoda `setf()` tedy nastavuje bity na hodnotu 1 a metoda `unsetf()` je nastavuje zpět na hodnotu 0.

Standardní manipulátory

Použití metody `setf()` nepředstavuje pro uživatele ten nejvhodnější způsob formátování. Jazyk C++ nabízí několik manipulátorů, které tuto metodu vyvolají za vás a automaticky dodají správné parametry. Již jste se setkali s manipulátory `dec`, `hex` a `oct`. Zmíněné manipulátory, z nichž většina ve starších implementacích neexistuje, pracují stejně jako manipulátor `hex`. Například příkaz

```
cout << left << fixed;
```

zapne volbu pro zarovnání vlevo a zobrazení s pevným oddělovačem. Tyto manipulátory a některé další jsou uvedeny v tabulce 16.3.

Tip

Pokud váš systém tyto manipulátory podporuje, využívejte je. V opačném případě máte stále možnost použít metodu `setf()`.

Tabulka 16.3 Některé standardní manipulátory.

Manipulátor	Volání
boolapha	setf(ios_base::boolapha)
noboolapha	unsetf(ios_base::noboolapha)
showbase	setf(ios_base::showbase)
noshowbase	unsetf(ios_base::noshowbase)
showpoint	setf(ios_base::showpoint)
noshowpoint	unsetf(ios_base::noshowpoint)
showpos	setf(ios_base::showpos)
noshowpos	unsetf(ios_base::noshowpos)
uppercase	setf(ios_base::uppercase)
nouppercase	unsetf(ios_base::nouppercase)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base:: basefield)
oct	setf(ios_base::oct, ios_base:: basefield)
fixed	setf(ios_base::fixed, ios_base:: floatfield)
scientific	setf(ios_base::scientific, ios_base:: floatfield)

Hlavičkový soubor iomanip

Nastavit některé formátovací hodnoty, například šířku pole, pomocí nástrojů třídy ostream může být nepohodlné. Aby vám jazyk C++ ulehčil život, obsahuje další manipulátory v hlavičkovém souboru iomanip. Tyto manipulátory poskytují stejné služby, jaké jsme již probírali, ale jejich zápis je pohodlnější. Mezi tři nejpoužívanější patří setprecision() nastavující přesnost, setfill() nastavující výplňový znak a setw() nastavující šířku pole. Na rozdíl od dříve probíraných manipulátorů mají tyto uvedené parametry. Manipulátor setprecision() má jako parametr celé číslo určující přesnost, manipulátor setfill() má parametr typu char označující výplňový znak a manipulátor setw() specifikuje celočíselným parametrem šířku pole. Protože se jedná o manipulátory, mohou být v příkazu s objektem cout řetězeny. Díky tomu se manipulátor setw() hodí zvláště při zobrazování několika sloupců hodnot. Program ve výpisu 16.10 ilustruje několik změn šířky pole a výplňového znaku ve výstupním řádku a používá rovněž některé novější manipulátory.

Kompatibilita:

Tento program používá matematickou funkci, ale některé systémy C++ knihovnu math automaticky neprohledávají. Například v systémech UNIX musíte vložit následující řádek:

```
$ CC setf2.C -lm
```

Volba -lm informuje linker, že má prohledávat knihovnu math. Starší kompilátory také nemusejí rozeznat nové standardní manipulátory jako například showpoint. V takovém případě můžete použít ekvivalenty s metodou setf().

Výpis 16.10 iomanip.cpp

```
// iomanip.cpp - použití manipulátorů z hlavičkového souboru iomanip
// některé systémy vyžadují explicitní vazbu na matematickou knihovnu
#include <iostream>
using namespace std;
#include <iomanip>
#include <cmath>

int main()
{
    // použití nových standardních manipulátorů
    cout << showpoint << fixed << right;

    // použití manipulátorů z iomanip
    cout << setw(6) << "Cislo" << setw(14) << "2. odmocnina"
         << setw(15) << "4. odmocnina\n";

    double root;
    for (int n = 10; n <=100; n += 10)
    {
        root = sqrt(n);
        cout << setw(6) << setfill('.') << n << setfill(' ')
```

```
                    << setw(12) << setprecision(3) << root
                    << setw(14) << setprecision(4) << sqrt(root)
                    << "\n";
    }
    return 0;
}
```

Zde je výstup:

```
Cislo     2. odmocnina     4. odmocnina
....10       3.162            1.7783
....20       4.472            2.1147
....30       5.477            2.3403
....40       6.325            2.5149
....50       7.071            2.6591
....60       7.746            2.7832
....70       8.367            2.8925
....80       8.944            2.9907
....90       9.487            3.0801
...100      10.000            3.1623
```

Nyní můžete vytvářet úhledně zarovnané sloupce. Všimněte si, že tento program vytvoří stejné formátování se starší i s novější implementací. Pomocí manipulátoru showpoint zobrazí koncové nuly ve starších implementacích a pomocí manipulátoru `fixed` v implementacích současných. Při použití manipulátoru `fixed` se v obou systémech použije zobrazení s pevným oddělovačem a v současných systémech bude přesnost označovat počet číslic vpravo od desetinného místa. Ve starších systémech má přesnost tento význam vždy bez ohledu na režim zobrazení typu s pohyblivou řádovou čárkou.

V tabulce 16.4 jsou shrnuty některé rozdíly mezi starším formátováním v C++ a současným stavem. Ponaučení vyplývající z této tabulky zní, že byste neměli být překvapeni, pokud spustíte vzorový program, na který jste někde narazili a výsledný formát nebude odpovídat formátu z daného příkladu.

Tabulka 16.4 Změny ve formátování

Vlastnost	Starší C++	Současné C++
precision(n)	Zobrazí `n` číslic vpravo od desetinného oddělovače	V implicitním režimu zobrazí `n` číslic celkem, zatímco v režimu s pevným oddělovačem nebo ve vědeckém zobrazí `n` číslic vpravo od desetinného oddělovače
ios::showpoint	Zobrazí koncový desetinný oddělovač a koncové nuly	Zobrazí koncový desetinný oddělovač
ios::fixed, ios::scientific		Zobrazí koncové nuly (viz také komentáře u precision())

Vstup pomocí objektu `cin`

Nyní je na čase podívat se na vstup a zadávání dat do programu. Objekt `cin` reprezentuje standardní vstup jako proud bajtů. Normálně generujete proud znaků pomocí klávesnice. Jestliže zadáte sekvenci znaků 1998, vytáhne objekt `cin` tyto znaky ze vstupního proudu. Vstupem může být také část řetězce, hodnoty typu `int`, `float` nebo nějaký jiný typ. Součástí vytažení je tedy také typová konverze. Objekt `cin` se řídí typem proměnné, která má hodnotu obdržet, a musí proto používat metody převádějící tuto sekvenci znaků na zamýšlený typ hodnoty.

Objekt `cin` můžete běžně použít následujícím způsobem:

```
cin >> value_holder;
```

Zde `value_holder` identifikuje místo v paměti, kam se vstup uloží. Může jím být název proměnné, reference, dereferencovaný ukazatel nebo člen struktury nebo třídy. Způsob, jakým objekt `cin` vstup interpretuje, závisí na datovém typu `value_holder`. Třída `istream` definovaná v hlavičkovém souboru iostream přetěžuje operátor vytažení `>>` pro následující základní typy:

- `signed char &`
- `unsigned char &`
- `char &`
- `short &`
- `unsigned short &`
- `int &`
- `unsigned int &`
- `long &`
- `unsigned long &`
- `float &`
- `double &`
- `long double &`

Říká se jim funkce pro formátovaný vstup, protože převádějí vstupní data do formátu určeného výsledným typem.

Typická funkce operátoru má následující prototyp:

```
istream & operator>>(int &);
```

Parametrem i návratovou hodnotou je reference. Reference jako parametr (viz kapitola 9) znamená, že příkaz

```
cin >> staff_size;
```

způsobí, že funkce `operator>>()` bude pracovat se skutečnou proměnnou `staff_size` a ne s kopií jako při běžném parametru. Je-li parametr typu reference, může objekt `cin` modifikovat hodnotu předávané proměnné přímo. Výše uvedený příkaz například přímo

upraví hodnotu proměnné staff_size. Jaký význam má reference jako návratová hodnota si ukážeme za chvíli. Nejdříve prozkoumejme aspekt typové konverze operátoru vytažení. Pro každý typ parametru ve výše uvedeném seznamu typů převádí operátor vytažení znakový vstup na hodnotu určeného typu. Předpokládejme například, že proměnná staff_size je typu int. Kompilátor v takovém případě porovná příkaz

```
cin >> staff_size;
```

s následujícím prototypem:

```
istream & operator>>(int &);
```

Funkce odpovídající tomuto prototypu potom načte proud znaků poslaných programu, například znaky 2, 3, 1, 8 a 4. Používá-li systém typ int o velikosti 2 bajtů, převede funkce tyto znaky na dvoubajtovou binární reprezentaci čísla 23184. Pokud by proměnná staff_size byla typu double, použil by objekt cin funkci operator>>(double) a stejný vstup by převedl na osmibajtovou reprezentaci čísla s pohyblivou řádovou čárkou 23184.0.

Chcete-li určit, že celočíselný vstup se má interpretovat v šestnáctkovém, osmičkovém nebo desítkovém formátu, můžete s objektem cin použít manipulátory hex, oct a dec. Například příkaz

```
cin >> hex;
```

způsobí, že se vstupní číslo 12 nebo 0x12 načte šestnáctkově jako 12 nebo desítkově 18, a ff nebo FF se desítkově načte jako 255.

Třída istream také přetěžuje operátor vytažení >> pro ukazatele na znakové typy:

- Signed char *
- char *
- unsigned char *

Při tomto typu parametru načte operátor vytažení ze vstupu následující slovo, umístí je na označenou adresu a přidá nulový znak ukončující řetězec. Předpokládejme například tento kód:

```
cout << "Zadejte jmeno:\n";
char name[20];
cin >> name;
```

Pokud na žádost odpovíte zadáním jména Hilary, vloží operátor vytažení do pole name znaky Hilary\0. (Jako obvykle \0 reprezentuje ukončovací nulový znak.) Identifikátor name, označující pole typu char, se chová jako adresa prvního prvku tohoto pole a stává se z něho typ char * (ukazatel na typ char).

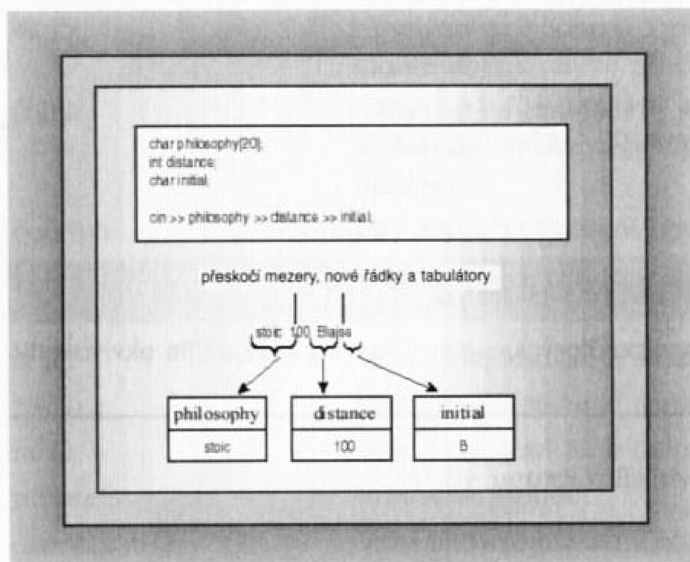
Vzhledem k tomu, že každý operátor vytažení vrací referenci na volající objekt, můžete vstupy spojovat stejným způsobem jako výstupy:

```
char name[20];
float fee;
int group;
cin >> name >> fee >> group;
```

Zde se například objekt cin vrácený výrazem cin >> name stane objektem pracujícím s proměnnou `fee`.

Jak pohlíží výraz `cin` >> na vstupní proud

Různé verze operátoru vytažení se na vstupní proud dívají stejným způsobem. Přeskakují bílé znaky (mezery, nové řádky a tabulátory), dokud nenarazí na tisknutelný znak. Platí to i o jednoznakových režimech (to jsou ty, v nichž je parametr typu `char`, `unsigned char` nebo `signed char`), na rozdíl od funkcí pro vstup znaku jazyka C (viz obrázek 16.5). V jednoznakových režimech operátorostatních režimech operátor načítá do jednotky označeného typu. To znamená, že čte vše od prvního tisknutelného znaku až do prvního znaku, který neodpovídá typu určení.



Obrázek 16.5 Výraz cin >> přeskočí bílé znaky

Uvažujte například následující kód:

```
int elevation;
cin >> elevation;
```

Předpokládejme, že zadáte následující znaky:

```
-123Z
```

Operátor přečte znaky -, `1`, `2` a `3`, protože toto jsou všechno platné součásti celého čísla. Znak Z však platný není, takže posledním přijatým znakem ze vstupu je 3. Znak `Z` zůstane ve vstupním proudu a další příkaz cin začne číst od něho. Mezitím operátor převede sekvenci znaků –123 na hodnotu celého čísla a toto číslo přiřadí proměnné `elevation`.

Může se stát, že vstup nebude odpovídat očekávání programu. Předpokládejme například, že jste místo –123Z zadali `Zcar`. V takovém případě ponechá operátor vytažení hodnotu proměnné `elevation` beze změny a vrátí nulovou hodnotu. (Řečeno více technicky, příkaz `if` nebo `while` vyhodnotí objekt třídy `istream` jako `false`, jestliže došlo k nastavení chybového stavu – hlouběji se tímto problémem budeme zabývat v této kapitole

později.) Návratová hodnota `false` umožní programu ve výpisu 16.11 zkontrolovat, zda vstup odpovídá požadavkům.

Výpis 16.11 check_it.cpp

```
#include <iostream>
using namespace std;
int main(){
    cout.precision(2);
    cout << showpoint << fixed;
    cout << "Zadavejte cisla: ";
    double sum = 0.0;
    double input;
    while (cin >> input)    {
        sum += input;
    }
    cout << "Posledni zadana hodnota = " << input << "\n";
    cout << "Soucet = " << sum << "\n";
    return 0;
}
```

Komapatibilita

Pokud váš kompilátor nepodporuje manipulátory `showpoint` a `fixed`, použijte ekvivalenty, které nabízí metoda `setf()`.

Zde je výstup, který vznikl z nevyhovujícího vstupu `-123Z`:

```
Zadavejte cisla: 200.0
1.0E1 -50 -123Z 60
Posledni zadana hodnota = -123.00
Soucet = 37.00
```

Vzhledem k tomu, že vstup používá vyrovnávací paměť, byl druhý řádek vstupních hodnot z klávesnice poslán programu teprve po stisknutí klávesy `Enter`. Cyklus však ukončí zpracování u znaku `Z`, protože znak neodpovídá formátu s pohyblivou řádovou čárkou. Neodpovídající vstup způsobí, že se výraz `cin >> input` vyhodnotí jako `false` a cyklus `while` skončí.

Stavy proudů

Podívejme se podrobněji, co se stane při nevyhovujícím vstupu. Objekty `cin` nebo `cout` obsahují členská data (zděděná od třídy `ios_base`) popisující *stav proudu*. Stav proudu (definovaný jako typ `iostate`, který je zase dříve popsaným typem bitmask) obsahuje tři prvky třídy `ios_base`: `eofbit`, `badbit` a `failbit`. Každý prvek je tvořen jediným bitem s hodnotou 1 (nastaven) nebo 0 (vyčištěn). Když operace `cin` dosáhne konce souboru, nastaví se `eofbit`. Jestliže se operaci `cin` nepodaří načíst očekávané znaky jako ve výše uvedeném příkladě, nastaví se `failbit`. Chyby při vstupu a výstupu, například pokus o čtení nepří-

stupného souboru nebo z diskety chráněné proti zápisu, mohou také nastavit `failbit` na hodnotu 1. Prvek `badbit` se nastaví, jestliže nějaká nediagnostikovaná chyba poškodila proud. (Implementace se nemusejí shodovat v tom, které události nastavují `failbit` a které `badbit`.) Pokud jsou všechny tři stavy bitů nastaveny na hodnotu 0, je vše v pořádku. Programy mohou kontrolovat stav proudu a pomocí těchto informací rozhodují o další činnosti. V tabulce 16.5 je seznam těchto bitů uveden společně s několika metodami třídy `ios_base`, které stav proudu hlásí nebo mění. (Starší kompilátory nepodporují žádnou z metod `exceptions()`).

Tabulka 16.5 Stavy proudů.

Člen	Popis
eofbit	Nastaven na hodnotu 1 při dosažení konce souboru.
badbit	Nastaven na hodnotu 1 při poškozeném proudu; například mohlo dojít k chybě při čtení ze souboru.
failbit	Nastaven na hodnotu 1, jestliže vstupní operace nenačetla očekávané znaky nebo výstupní operace očekávané znaky nezapsala.
goodbit	Jiný způsob vyjádření hodnoty 0.
good()	Vrátí `true`, jestliže lze proud použít (všechny bity jsou vyčištěny).
eof()	Vrátí `true`, jestliže je nastaven `eofbit`.
bad()	Vrátí `true`, jestliže je nastaven `badbit`.
fail()	Vrátí `true`, jestliže je nastaven `badbit`.
rdstate()	Vrátí stav proudu.
exceptions()	Vrátí bitovou masku identifikující příznak, který vyvolal výjimku.
exceptions(iostate ex)	Nastaví stavy, které pomocí funkce `clear()` vyvolají výjimku; například jestliže `ex` označuje `eofbit`, vyvolá funkce `clear()` výjimku při nastaveném `eofbit`.
clear(iostate s)	Nastaví stav proudu na stav `s`; implicitní hodnotou `s` je 0 (goodbit); vyvolá výjimku `basic_ios::failure`, jestliže platí `rdstate() & exceptions() != 0`.
setstate(iostate s)	Volá funkci `clear(rdstate() | s)`. Toto volání nastaví stavové bity proudu podle odpovídajících bitů v `s`; stavové bity jiných proudů zůstanou beze změny.

Nastavení stavů

Dvě z metod v tabulce 16.5, `clear()` a `setstate()`, jsou podobné. Obě nastavují stav, ale odlišným způsobem. Metoda `clear()` nastavuje stav podle svého parametru. Volání

```
clear();
```

tedy použije implicitní parametr s hodnotou 0 a vyčistí všechny tři stavové bity (eofbit, badbit a failbit). Podobně volání

```
clear(eofbit);
```

vytvoří stav rovnající se eofbit, to znamená, že eofbit bude nastaven a ostatní dva bity vyčištěny.

Metoda setstate() však ovlivňuje pouze ty bity, které jsou nastaveny v parametru. Volání

```
setstate(eofbit);
```

nastaví eofbit, ale na ostatní bity vliv nemá. Jestliže byl failbit již nastaven, zůstane nastaven.

Proč máte nastavovat stav proudu? Nejběžnější důvod, proč programátor použije metodu clear() bez parametrů je ten, že chce vstup znovu otevřít, jakmile narazí na neshodu typů na vstupu nebo na konec souboru; zda tato činnost má či nemá smysl závisí na tom, čeho má program dosáhnout. Za chvíli uvidíte několik příkladů. Hlavním účelem metody setstate() je poskytnout funkcím pro vstup a výstup prostředky pro změnu stavu. Například jestliže num je proměnná typu int, volání

```
cin >> num;        // přečte celé číslo
```

může vést k volání funkce operator>>(int &), která pomocí metody setstate() nastaví failbit nebo eofbit.

Vstup, výstup a výjimky

Předpokládejme například, že nějaká vstupní funkce nastaví eofbit. Způsobí tato činnost vyvolání výjimky? Standardně ne. Chcete-li však zpracování výjimek ovládat, můžete použít metodu exceptions().

Nejprve nějaké pozadí. Metoda exceptions() vrací bitové pole s třemi bity odpovídajícími bitům eofbit, failbit a badbit. Chcete-li stav proudu změnit, musíte použít buď metodu clear() nebo metodu setstate(), která požívá clear(). Jakmile se stav proudu změní, porovná metoda clear() současný stav proudu s hodnotou vrácenou metodou exceptions(). Jestliže je nastaven bit v návratové hodnotě a odpovídající bit v aktuálním stavu, vyvolá metoda clear() výjimku basic_ios::failure. K tomu by například došlo, pokud by obě hodnoty měly nastaven badbit. Z uvedeného vyplývá, že jestliže metoda exceptions() vrátí goodbit, k vyvolání výjimky nedojde.

Implicitně je metoda exceptions() nastavena na goodbit, takže k vyvolání výjimky nedojde. Přetížená funkce exceptions(iostate) vám však umožňuje chování ovládat:

```
cin.exceptions(badbit);   // nastavení badbitu vyvolá výjimku
```

Bitový operátor OR (|) uvedený v příloze E umožňuje specifikovat více bitů. Například příkaz

```
cin.exceptions(badbit | eofbit);
```

způsobí vyvolání výjimky, jestliže je následně nastaven badbit nebo eofbit.

Účinky proudových stavů

Test příkazy if nebo while, například

```
while (cin >> input)
```

vrátí hodnotu true pouze v případě, že stav proudu vyhovuje (všechny bity jsou vyčištěny). Jestliže test neuspěje, můžete pomocí členských funkcí uvedených v tabulce 16.5 zjišťovat možné příčiny. Střední část výpisu 16.11 byste například mohli upravit takto:

```
while (cin >> input)
{
  sum += input;
}
if (cin.eof())
  cout << "Cyklus skoncil v dusledku znaku EOF\n");
```

Nastavení bitu označujícího stav proudu má velmi důležitý následek: proud se uzavře pro další vstup nebo výstup až do vyčištění bitu. Následující kód například fungovat nebude:

```
while (cin >> input)
{
  sum += input;
}
cout << "Posledni zadana hodnota = " << input << "\n";
cout << "Soucet = " << sum << "\n";
cout << "Nyni zadejte nove cislo: ";
cin >> input;      // nefunguje
```

Jestliže chcete, aby program četl ze vstupu i po nastavení stavového bitu proudu, musíte tento bit nastavit jako dobrý. Toho dosáhnete zavoláním metody `clear()`:

```
while (cin >> input)
{
  sum += input;
}
cout << "Posledni zadana hodnota = " << input << "\n";
cout << "Soucet = " << sum << "\n";
cout << "Nyni zadejte nove cislo: ";
cin.clear();          // vyčistí stavové bity proudu
while (!isspace(cin.get()))
  continue;           // přeskočí nevyhovujici vstup
cin >> input;         // nyni fungovat bude
```

Všimněte si, že vyčistit stavové bity proudu nestačí. Neodpovídající typ na vstupu, který ukončil cyklus, je stále ve vstupní frontě a program se přes něj musí dostat. Jednou z možností je načítat znaky do té doby, než se narazí na bílý znak. Funkce `isspace()` (viz kapitola 6) je funkce `cctype` vracející hodnotu true, jestliže je jejím parametrem bílý znak. Jinou možností je zbavit se nejen následujícího slova, ale i zbytku řádku:

```
while (cin.get() != '\n')
  continue;           // zbaví se zbytku řádku
```

Tento příklad předpokládá, že cyklus skončil v důsledku neodpovídajícího vstupu. Předpokládejme však, že cyklus skončil v důsledku konce souboru nebo hardwarové chyby. V tom případě by nový kód přeskakující špatný vstup neměl smysl. Nápravu zařídíte metodou `fail()`, která ověří správnost předpokladů. Vzhledem k tomu, že z historických dů-

vodů vrací metoda hodnotu `true`, pokud je nastavený jeden z bitů `failbit` nebo `badbit`, musíte v kódu tuto druhou možnost vyloučit.

```
while (cin >> input)
{
   sum += input;
}
cout << "Posledni zadana hodnota = " << input << "\n";
cout << "Soucet = " << sum << "\n";
cout << "Nyni zadejte nove cislo: ";
if (cin.fail() && !cin.bad() )    // chyba na vstupu
{
   cin.clear();                   // vyčistí stavové bity proudu
while (!isspace(cin.get()))
         continue;                // přeskočí nevyhovující vstup
}
else      // ukončí činnost
{
   cout << "Nemohu pokracovat!\n";
   exit(1);
}
cout << "Nyni zadejte nove cislo: ";
cin >> input;        // nyní fungovat bude
```

Další metody třídy `istream`

V kapitolách 3, 4 a 5 jsou rozebrány metody `get()` a `getline()`. Jak si zřejmě vzpomínáte, disponují tyto metody pro vstup následujícími možnostmi:

- Metody `get(char &)` a `get(void)` čtou ze vstupu jednotlivé znaky a bílé znaky nepřeskakují.
- Funkce `get(char *, int, char)` a `getline(char *, int, char)` načítají implicitně spíše celé řádky než jednotlivá slova.

Těmto metodám se říká *funkce pro neformátovaný vstup*, protože jednoduše čtou znaky ze vstupu tak jak jsou a bílé znaky nepřeskakují a neprovádějí konverzi dat.

Pojďme se na tyto dvě skupiny členských funkcí třídy `istream` podívat.

Jednoznakový vstup

Jestliže jsou metody `get()` použity s parametrem typu `char` nebo bez parametru, přečtou ze vstupu následující znak, i když je jím mezera, tabulátor nebo znak nového řádku. Verze `get(char & ch)` přiřadí znak ze vstupu do svého parametru, zatímco verze `get(void)` použije znak ze vstupu převedený na celé číslo (obvykle typu `int`) jako svou návratovou hodnotu.

Nejdříve vyzkoušíme verzi `get(char & ch)`. Předpokládejme program s následujícím cyklem:

```
int ct = 0;
char ch;
cin.get(ch);
while (ch != '\n')
{
   cout << ch;
   ct++;
   cin.getch(ch);
}
cout << ct << '\n';
```

Dále předpokládejme následující optimistický vstup:

```
V C++ programuji dobre.<Enter>
```

Stisknutím klávesy Enter odešlete tento vstupní řádek do programu. Úsek programu nejdříve přečte znak V, zobrazí ho pomocí objektu `cout` a hodnotu proměnné `ct` zvýší na 1. Potom přečte mezeru, zobrazí ji a hodnotu proměnné ct zvýší na 2. Tak pokračuje, dokud nezpracuje klávesu Enter jako znak nového řádku a neukončí cyklus. Důležité je, že kód používající metodu `get(ch)` čte, zobrazuje a počítá mezery stejným způsobem jako tisknutelné znaky.

Nyní předpokládejme, že byste v tomto programu použili operátor >>:

```
int ct = 0;
char ch;
cin >> ch;
while (ch != '\n')          // CHYBA
{
   cout << ch;
   ct++;
   cin >> ch;
}
cout << ct << '\n';
```

Program by nejdříve přeskočil mezery a tedy by je nezapočítal, přičemž odpovídající výstup by zhustil následujícím způsobem:

```
VC++programujidobre.
```

Horší však je, že takový cyklus nikdy neskončí. Vzhledem k tomu, že operátor vytažení znak nového řádku přeskakuje, program by ho nikdy nepřiřadil proměnné `ch` a ověřovací cyklus `while` by nikdy neskončil.

Členská funkce `getchar(char &)` vrací referenci na objekt třídy `istream`, který ji vyvolal. To znamená, že další operátory vytažení následující za výrazem `get(char &)` můžete řetězit:

```
char c1, c2, c3;
cin.get(c1).get(c2) >> c3;
```

Nejdříve výraz `cin.get(c1)` přiřadí první znak ze vstupu proměnné c1 a vrátí volající objekt, kterým je `cin`. Kód se zredukuje na `cin.get(c2) >> c3` a přiřadí druhý vstupní znak proměnné `c2`. Volání funkce vrátí objekt cin a zredukuje kód na `cin >> c3`. Tím se zase přiřadí následující nebílý znak proměnné `c3`. Všimněte si, že proměnným `c1` a `c2` by bílý znak přiřazen být mohl, ale proměnné `c3` ne.

Jestliže metoda cin.getchar(char &) narazí na konec souboru, ať již skutečného nebo simulovaného, z klávesnice (Ctrl+Z v DOSu a Ctrl+D na začátku řádku v UNIXu), nepřiřadí hodnotu do svého parametru. Je to správné, neboť jestliže program dosáhne konce souboru, nemá se hodnota přiřadit. Metoda navíc volá metodu setstate(failbit), která vyhodnotí stav objektu cin jako false:

```
char ch;
while (cin.get(ch))
{
    // zpracování vstupu
}
```

Dokud jsou vstupní hodnoty platné, je návratovou hodnotou funkce cin.get(ch) objekt cin, což se vyhodnotí jako true a cyklus pokračuje. Při dosažení konce souboru se návratová hodnota vyhodnotí jako false a cyklus skončí.

Členská funkce get(void) čte bílé znaky také, ale návratovou hodnotu používá pro předání vstupu do programu. Možné je tedy následující použití:

```
int ct = 0;
char ch;
ch = cin.get(ch);
while (ch != '\n')
{
    cout << ch;
    ct++;
    ch = cin.get(ch);
}
cout << ct << '\n';
```

Některé starší implementace C++ tuto členskou funkci neobsahují.

Členská funkce cin.get() vrací hodnotu typu int (nebo nějakého většího celočíselného typu – závisí na znakové sadě a místním nastavení). Následující kód proto použít nelze:

```
char c1, c2, c3;
cin.get().get() >> c3;     // chybný příkaz
```

Výraz cin.get() vrátí hodnotu typu int. Protože touto návratovou hodnotou není objekt třídy, nemůžete na ni použít operátor výběru členu – obdrželi byste zprávu o syntaktické chybě. Na konci sekvence vytažení však funkci get() použít můžete:

```
char c1;
cin.get(c1).get(); // platný příkaz
```

Skutečnost, že funkce get(void) vrací hodnotu typu int znamená, že za ní nemůžete použít operátor vytažení. Ale protože výraz cin.get(c1) vrátí objekt cin, použije se tento objekt při dalším volání funkce get(). Tento konkrétní kód by načetl první znak ze vstupu, přiřadil ho proměnné c1, potom by načetl druhý znak a ten by vyřadil.

Při dosažení konce souboru, skutečného či simulovaného, vrátí volání cin.get(void) hodnotu EOF, což je symbolická konstanta, obsažená v hlavičkovém souboru iostream. Díky takovému návrhu lze pro načítání vstupu použít následující konstrukci:

```
int ch;
while ((ch = cin.get()) != EOF)
{
    // zpracování vstupu
}
```

Zde byste místo typu `char` měli používat typ `int`, protože hodnota `EOF` nemusí být vyjádřena jako typ `char`.

Tyto funkce jsou trochu podrobněji popsány v kapitole 5, zatímco v tabulce 16.6 je souhrn vlastností funkcí pro vstup jednoho znaku.

Tabulka 16.6 Funkce cin.get(ch) a funkce cin.get().

Vlastnost	cin.get(ch)	ch = cin.get()
Metoda předávající znak e vstupu	Přiřaďte parametru `ch`	Návratovou hodnotu funkce přiřaďte proměnné `ch`
Návratová hodnota funkce pro znak ze vstupu	Reference na objekt třídy `istream`	Kód znaku jako hodnota typu `int`
Návratová hodnota funkce při dosažení konce souboru	Převedena na `false`	`EOF`

Který tvar použít pro vstup znaku

Máte-li na výběr operátor `>>` a metody `get(char &)` a `get(void)`, nabízí se otázka, co z toho použít. Nejdříve se rozhodněte, zda chcete na vstupu přeskakovat bílé znaky nebo ne. Pro přeskakování bílých znaků se nejlépe hodí operátor vytažení `>>`. Dobře ho využijete například při volbě nabídky:

```
cout    << a. otravny klient    b. klient se smenkou\n"
    << c. klidny klient         d. podvodny klient\n"
    << q.\n";
cout    << "Zadejte a, b, c, d, or q: ";
char ch;
cin << ch;
while (ch != q)
{
    switch(ch)
    {
        ...
    }
    cout << "Enter a, b, c, d or q: ";
    cin << ch;
}
```

Chcete-li například odpovědět `b`, zadáte `b` a stisknete klávesu Enter, čímž vygenerujete dvouznakovou odpověď složenou z `b\n`. Kdybyste použili některý z tvarů metody `get()`, museli byste do každého cyklu přidat kód pro zpracování znaku `\n`, zatímco operátor vytažení ho pohodlně přeskočí. (Pokud jste programovali v jazyce C, setkali jste se pravdě-

podobně se situací, kde program považoval znak nového řádku za chybnou odpověď. Tento problém není těžké opravit, ale je to nepříjemné.)

Jestliže chcete, aby program zkoumal každý znak, použijte jednu z metod `get()`. Například program počítající slova by mohl pomocí bílých znaků určovat konec slova. Z obou metod `get()` má metoda `get(char &)` klasičtější rozhraní. Hlavní výhodou metody `get(void)` je, že velmi připomíná standardní funkci jazyka C `getchar()` a umožní vám převést program z C do C++, pokud místo hlavičkového souboru `stdio.h` vložíte soubor `iostream` a globálně nahradíte funkci `getchar()` funkcí `cin.get()` a funkci jazyka C `putchar(ch)` funkcí `cout.put(ch)`.

Vstup řetězce: metody `getline()`, `get()` a `ignore()`

Nyní si zopakujeme členské funkce pro vstup řetězce, které jsme uvedli v kapitole 4. Členská funkce `getline()` a třetí verze funkce `get()` načítají řetězce a obě mají stejnou signaturu (zde je zjednodušená z obecnější deklarace šablony):

```
istream & get(char *, int, char = '\n');
istream & getline(char *, int, char = '\n');
```

Vzpomeňte si, že prvním parametrem je adresa, na kterou se řetězec umístí. Druhým je číslo o jedničku větší než je maximální počet znaků, které lze načíst. (Znak navíc rezervuje místo pro ukončovací nulový znak, který se při ukládání řetězce použije.) Jestliže vynecháte třetí parametr, budou funkce načítat do maximálního počtu znaků nebo dokud nenarazí na znak nového řádku.

Například kód

```
char line[50];
cin.get(line, 50);
```

načte znaky ze vstupu do pole znaků `line`. Funkce `cin.getline()` skončí načítání vstupu do pole po načtení 49 znaků, nebo implicitně jestliže narazí na znak nového řádku. Hlavní rozdíl mezi funkcemi `get()` a `getline()` je ten, že funkce `get()` ponechává znak nového řádku ve vstupním proudu, kde zůstane jako první znak pro další vstupní operaci, zatímco funkce `getline()` znak nového řádku ze vstupního proudu vytáhne a odloží ho.

V kapitole 4 byly příklady, demonstrující používání implicitních forem obou těchto členských funkcí. Nyní se podíváme na poslední parametr, který implicitní chování funkce upravuje. Třetím parametrem s implicitní hodnotou `'\n'` je ukončovací znak. Při načtení ukončovacího znaku čtení ze vstupu skončí, i když nebyl přečten maximální počet znaků. Implicitně tedy obě metody skončí čtení ze vstupu, pokud narazí na konec řádku, aniž by načetly určený počet znaků. Stejně jako v implicitním případě ponechá metoda `get()` ukončovací znak ve vstupním proudu, zatímco metoda `getline()` ne.

Program ve výpisu 16.12 předvádí práci obou metod. Také uvádí členskou funkci `ignore()`. Ta má dva parametry: číslo určující maximální počet znaků, které se mají načíst, a znak, který se chová jako ukončovací znak pro čtení ze vstupu. Například volání funkce

```
cin.ignore(80, '\n');
```

přečte a odloží buď 80 znaků, nebo tolik, kolik jich předchází prvnímu znaku nového řádku. Implicitními hodnotami obou parametrů v prototypu jsou `1` a `EOF` a návratovou hodnotou funkce je reference na třídu `istream`:

```
istream & ignore(int = 1, int = EOF);
```

Funkce vrací volající objekt, proto můžete volání funkcí řetězit jako v následujícím případě:

```
cin.ignore(80, '\n').ignore(80, '\n');
```

První metoda `ignore()` v kódu načte a odloží jeden řádek a druhé volání načte a odloží řádek druhý. Dohromady přečtou obě funkce dva řádky.

Nyní prozkoumáme program ve výpisu 16.12.

Výpis 16.12 get_fun.cpp

```
// get_fun.cpp - použití funkcí get() a getline()
#include <iostream>
using namespace std;
const int Limit = 80;

int main(){
    char input[Limit];
    cout << "Zadejte retezec pro funkci getline():\n";
    cin.getline(input, Limit, '#');
    cout << "Zadany vstup:\n";
    cout << input << "\nKonec faze 1\n";

    char ch;
    cin.get(ch);
    cout << "Dalsim znakem na vstupu je " << ch << "\n";

    if (ch != '\n')
        cin.ignore(Limit, '\n');     // zbytek řádku odloží
    cout << "Zadejte retezec pro funkci get():\n";
    cin.get(input, Limit, '#');
    cout << "Zadany vstup:\n";
    cout << input << "\nKonec faze 2\n";
    cin.get(ch);
    cout << " Dalsim znakem na vstupu je " << ch << "\n";
    return 0;
}
```

Kompatibilita:

Verze funkce `getline()` z hlavičkového souboru `iostream` Microsoft Visual C++ 5.0 (poznámka překladatele: týká se i verze 6.0) obsahuje chybu, jejíž vinou se řádek výstupu zobrazí až po zadání dalších vstupních údajů. Verze z hlavičkového souboru `iostream.h` však funguje správně.

Zde je ukázka běhu programu:

```
Zadejte retezec pro funkci getline():
Podejte mi prosim meloun #3 vpravo!
Zadany vstup:
Podejte mi prosim meloun
Konec faze 1
Dalsim znakem na vstupu je 3
Zadejte retezec pro funkci get():
Stale chci meloun #3 vpravo!
Zadany vstup:
Stale chci meloun
Konec faze 2
Dalsim znakem na vstupu je #
```

Všimněte si, že funkce `getline()` ukončovací znak # ze vstupního proudu odkládá, zatímco funkce `get()` ne.

Neočekávaný vstup řetězce

Některé formy vstupu pro funkce `get(char *, int)` a `getline()` mají vliv na stav proudu. Stejně jako ostatní funkce pro vstup nastavují při načtení znaku konce souboru `eofbit`, zatímco vše, co proud poškozuje, jako například chyba zařízení, nastavuje `badbit`. Dalšími dvěma speciálními případy jsou chybějící vstup a vstup, který odpovídá maximálnímu počtu znaků specifikovanému ve volání funkce nebo tento počet přesahuje. Na tyto případy se nyní podíváme.

Jestliže se metodě nepodaří načíst žádné znaky, vloží do řetězce nulový znak a pomocí funkce `setstate()` nastaví `failbit`. (Starší implementace C++ v takovém případě `failbit` nenastavují.) V jakých případech se metodě nepodaří znaky načíst? Jednou z možností je, že metoda na vstupu ihned narazí na znak konce souboru. Další možnost týkající se metody `get(char *, int)` nastane, jestliže zadáte prázdný řádek:

```
char temp[80];
while (cin.get(temp, 80))          // skončí z důvodu prázdného řádku
```

Je zajímavé, že metoda `getline()` při načtení prázdného řádku `failbit` nenastaví. Důvodem je, že znak nového řádku načte, i když ho neuloží. Pokud budete chtít, aby cyklus s metodou `getline()` skončil po načtení prázdného řádku, můžete napsat následující kód:

```
char temp[80];
while (cin.getline(temp, 80) && temp[0] != '\0') // skončí z důvodu
                                                 // prázdného řádku
```

Nyní předpokládejme, že počet znaků ve vstupní frontě odpovídá maximálnímu počtu specifikovanému v metodě pro vstup nebo tento maximální počet přesahuje. Nejdříve uvažujte metodu `getline()` a následující kód:

```
char temp[30];
while (cin.get(temp, 30))
```

Metoda `getline()` bude číst sekvenci znaků ze vstupní fronty a vkládat je do sekvence prvků pole `temp`, dokud nenarazí na konec souboru, znak nového řádku nebo dokud ne-

uloží 29 znaků. Jestliže narazí na EOF, nastaví eofbit. Pokud bude dalším znakem znak nového řádku, načte ho a odloží. A pokud přečte 29 znaků a následujícím znakem nebude znak nového řádku, nastaví failbit. Vstupní řádek o délce 30 nebo více znaků tedy způsobí ukončení vstupu.

Nyní uvažujte metodu get(char *, int). Ta nejdříve testuje počet znaků, potom konec souboru a na třetím místě znak nového řádku. Načte-li maximální počet znaků, příznak failbit nenastaví. Přesto však poznáte, že metoda ukončila čtení z důvodu velkého množství znaků na vstupu. Následující znak na vstupu můžete ověřit pomocí metody peek() (viz další část). Pokud se jedná o znak nového řádku, musí metoda get() načíst celý řádek. Jestliže se o nový řádek nejedná, ukončila metoda get() čtení určitě před koncem. U metody getline() tuto techniku použít nelze, protože ta znak nového řádku načte a odloží, a zkoumáním následujícího znaku tedy nic nezjistíte. Jestliže však použijete metodu get() a celý řádek se nenačte, můžete něco podniknout. V další části najdete příklad takového postupu. Následující tabulka 16.7 shrnuje některé rozdíly mezi staršími metodami pro vstup a současným standardem.

Tabulka 16.7 Změny v chování metod pro vstup.

Metoda	Starší C++	Současné C++
getline()	Nenastaví `failbit`, pokud nenačte žádné znaky nebo pokud přečte maximální počet znaků.	Nastaví `failbit`, jestliže nenačte žádné znaky (znak nového řádku však počítá jako přečtený znak) a také pokud přečte maximální počet znaků a řádek ještě obsahuje další znaky.
get(char *, int)	Nenastaví `failbit`, pokud nenačte žádné znaky.	Nastaví `failbit`, pokud nenačte žádné znaky.

Další metody třídy `istream`

Mezi další metody třídy istream patří `read()`, `peek()`, `gcount()` a `putback()`. Funkce `read()` přečte zadaný počet bajtů a uloží je na specifikované místo. Například příkaz

```
char gross[144];
cin.read(gross, 144);
```

přečte ze standardního vstupu 144 znaků a vloží je do pole gross. Na rozdíl od metod `getline()` a `get()` nepřidává metoda `read()` ke vstupu nulový znak, takže ho nepřevádí do tvaru řetězce. Není primárně určena pro vstup z klávesnice a nejčastěji se používá ve spojení s funkcí `write()` třídy `ostream` pro souborový vstup a výstup. Návratovým typem je `istream &` a lze ji tedy řetězit následujícím způsobem:

```
char gross[144];
char score[20];
cin.read(gross, 144).read(score, 20);
```

Funkce `peek()` vrací následující znak ze vstupu, který však ze vstupního proudu nenačte. Dovolí vám podívat se na něj. Předpokládejme, že jste chtěli načítat vstup až do prvního

znaku nového řádku nebo do znaku tečka. Pomocí funkce peek() se můžete na následující znak ve vstupním proudu podívat a posoudit, zda budete pokračovat nebo ne:

```
char great_input[80];
char ch;
int i = 0;
while ((ch = cin.peek()) != '.' && ch != '\n')
  cin.get(great_input[i++]);
great_input [i] = '\n';
```

Zavoláním funkce cin.peek() se podíváte na následující vstupní znak a jeho hodnotu přiřadíte proměnné ch. Potom podmínkou v cyklu while otestujete, zda hodnotou proměnné ch není znak tečka nebo nový řádek. Pokud ne, načte se znak do pole a aktualizuje se index pole. Když cyklus skončí, zůstane znak tečka nebo nový řádek ve vstupním proudu a bude prvním znakem, který se načte při následující vstupní operaci. Nakonec kód přidá do pole nulový znak, čímž z něho udělá řetězec.

Metoda gcount() vrací počet znaků načtených poslední metodou pro neformátovaný vstup, čili jednou z metod get(), getline(), ignore() a read(), ale ne operátorem vytažení (>>), který vstup formátuje a přizpůsobuje na určitý datový typ. Předpokládejme například, že jste pro načtení řádku do pole myarray použili metodu cin.get(myarray, 80) a chcete znát počet načtených znaků. Spočítat znaky v poli byste mohli pomocí funkce strlen(), ale použijete-li metodu gcount(), dozvíte se počet znaků načtených ze vstupního proudu rychleji.

Funkce putback() vloží znak zpět do vstupního řetězce. Tento znak pak bude první, který se načte následujícím příkazem pro vstup. Metoda putback() má jeden parametr, kterým je vkládaný znak, a vrací typ istream &, takže volání této metody lze řetězit s ostatními metodami třídy istream. Zatímco pomocí metod peek() a get() lze znak načíst, pomocí metody putback() ho vrátíte do vstupního proudu. Můžete však vrátit i jiný znak než ten, který jste právě načetli.

Program ve výpisu 16.13 používá dva způsoby čtení a vstup potvrzuje až do znaku #. První způsob znak # přečte a potom ho pomocí metody putback() vrátí do vstupního proudu. Druhý způsob používá metodu peek() a sleduje, který znak ze vstupu bude následovat.

Výpis 16.13 peeker.cpp

```
// peeker.cpp - některé metody z istream
#include <iostream>
using namespace std;
#include <cstdlib>                // nebo stdlib.h

int main()
{

//  čte a zobrazuje vstup až do znaku #
    char ch;
    while(cin.get(ch))            // skončí při EOF
    {
```

```
            if (ch != '#')
                cout << ch;
            else
            {
                cin.putback(ch);    // vloží znak zpět
                break;
            }
        }
        if (!cin.eof())
        {
            cin.get(ch);
            cout << "n\Dalsim vstupnim znakem je " << ch << ".\n";
        }
        else
        {
            cout << "Konec souboru.\n";
            exit(0);
        }
        while(cin.peek() != '#')    // předem kontroluje vstupní znaky
        {
            cin.get(ch);
            cout << ch;
        }
        if (!cin.eof())
        {
            cin.get(ch);
      cout << "\nDalsim vstupnim znakem je " << ch << ".\n";
        }
        else
            cout << "Konec souboru.\n";
        return 0;
    }
```

Zde je ukázka běhu:

```
Pouzil jsem tuzku #3 misto tuzky #2.
Pouzil jsem tuzku
Dalsim vstupnim znakem je #.
3 misto tuzky
Dalsim vstupnim znakem je #.
```

Poznámky k programu

Podívejme se na část uvedeného kódu podrobněji. První způsob čte vstup pomocí cyklu `while`. Výraz `cin.get(ch)` vrátí za podmínky dosažení konce souboru hodnotu `0`, to znamená, že simulací konce souboru z klávesnice cyklus ukončíte. Pokud se dříve načte znak `#`, vrátí ho program zpět do vstupního proudu a ukončí cyklus pomocí příkazu `break`.

```
while(cin.get(ch))          // skončí při EOF
{
        if (ch != '#')
            cout << ch;
```

```
        else
        {
            cin.putback(ch);    // vloží znak zpět
            break;
        }
    }
```

Druhý způsob je zdánlivě jednodušší:

```
while(cin.peek() != '#')     // předem kontroluje vstupní znaky
{
    cin.get(ch);
    cout << ch;
}
```

Program se podívá na následující znak. Jestliže jím není znak #, přečte ho, zobrazí a podívá se na následující znak. Tak pokračuje až do ukončovacího znaku.

Nyní se podíváme, jak jsme slíbili, na příklad (výpis 16.14), ve kterém se pomocí metody `peek()` určuje, zda byl či nebyl načten celý řádek. Pokud se do vstupního pole vejde pouze část řádku, program zbytek odloží.

Výpis 16.14 truncate.cpp

```
// truncate.cpp - v případě nutnosti zkrátí vstupní řádek pomocí
// metody get()
#include <iostream>
using namespace std;
const int SLEN = 11;
inline void eatline() { while (cin.get() != '\n') continue; }
int main()
{
    char name[SLEN];
    char title[SLEN];
    cout << "Zadejte jmeno: ";
    cin.get(name,SLEN);
    if (cin.peek() != '\n')
        cout << "Misto je bohuzel pouze pro "
                << name << endl;
    eatline();
    cout << "Dobry den " << name << ", zadejte funkci: \n";
    cin.get(title,SLEN);
    if (cin.peek() != '\n')
        cout << "Byli jsme nuceni funkci zkratit.\n";
    eatline();
    cout << "Jmeno: " << name
         << "\nFunkce: " << title << endl;
    return 0;
}
```

Zde je ukázka běhu:

```
Zadejte jmeno: Jiri Karasek
Misto je bohuzel pouze pro Jiri Karas
Dobry den Jiri Karas, zadejte funkci:
Vedouci obchodu
Byli jsme nuceni funkci zkratit.
Jmeno: Jiri Karas
Funkce: Vedouci ob
```

Všimněte si, že následující kód lze použít bez ohledu na to, zda první příkaz pro vstup načte nebo nenačte celý řádek:

```
while (cin.get() != '\n') continue;
```

I když metoda `get()` celý řádek načte, znak nového řádku ponechává na místě a uvedený kód ho tedy odloží. Jestliže metoda načte pouze část řádku, kód zbytek přečte a odloží. Kdybyste se zbytku řádku nezbavili, začal by následující příkaz pro vstup číst zbývající vstup na prvním vstupním řádku. V uvedeném příkladě by program načetl do pole `title` řetězec `ek`.

Vstup ze souboru a výstup do souboru

Většina počítačových programů pracuje se soubory. Textové procesory pracují s dokumenty. Databázové programy vytvářejí a prohledávají soubory informací. Kompilátory čtou soubory se zdrojovými kódy a generují soubory spustitelné. Samotný soubor je množství bajtů uložených na nějakém zařízení, například na magnetické pásce, optickém disku, disketě nebo pevném disku. Soubory spravuje operační systém, který zaznamenává jejich místo v paměti, velikost, údaje o vytvoření a tak dále. Pokud neprogramujete na systémové úrovni, nemusíte se normálně o tyto věci starat. Potřebujete nějak spojit program se souborem, přečíst obsah souboru a vytvářet soubory a zapisovat do nich. Určitou podporu souborům poskytuje přesměrování (probírané dříve v této kapitole), má však omezenější možnosti než explicitní přístup k souborům z programu. Přesměrování je také záležitostí operačního systému, nikoli jazyka C++, a není tedy dostupné v každém systému. Nyní se podíváme, jak se explicitně přistupuje k souborům z programu v C++.

Balík tříd pro vstup a výstup řeší vstup ze souboru a výstup do souboru podobně jako vstup a výstup standardní. Chcete-li do souboru zapisovat, vytvoříte objekt proudu a použijete metody třídy `ostream`, jako jsou operátor vložení << nebo metoda `write()`. Jestliže chcete soubor číst, vytvoříte objekt proudu a použijete metody třídy `istream`, jako jsou operátor vytažení >> nebo metoda `get()`. Správa souborů je však náročnější než standardní vstup a výstup. Soubor můžete otevřít v režimu pouze pro čtení, pouze pro zápis, nebo pro čtení i pro zápis. Pokud chcete do souboru zapisovat, můžete vytvořit soubor nový, přepsat původní nebo data k původnímu souboru přidat. Také můžete chtít souborem pouze procházet. Pro zvládnutí těchto úkolů definuje C++ v hlavičkovém souboru `fstream` (dříve `fstream.h`) několik nových tříd, včetně třídy `ifstream` pro vstup ze souboru a třídy `ofstream` pro výstup do souboru. Rovněž definuje třídu `fstream` pro současný vstup i výstup. Ty-

to třídy jsou odvozeny od tříd v hlavičkovém souboru `iostream`, takže jejich objekty mohou používat metody, které již znáte.

Jednoduchý vstup ze souboru a výstup do souboru

Předpokládejme, že potřebujte program, který bude zapisovat do souboru. Musíte provést následující kroky:

- Vytvořit objekt třídy `ofstream`, který bude spravovat výstupní proud.
- Přiřadit tento objekt určitému souboru.
- Použít objekt stejným způsobem jako objekt `cout`; jediný rozdíl spočívá v tom, že výstup půjde do souboru místo na obrazovku.

Pro splnění tohoto úkolu vložte nejdříve do programu hlavičkový soubor `fstream`. Tím se automaticky vloží pro většinu implementací (ale ne pro všechny) soubor `iostream` a nebudete ho tedy muset vkládat explicitně. Potom deklarujte objekt třídy `ofstream`:

```
ofstream fout;     // vytvoří objekt třídy ofstream s názvem fout
```

Názvem objektu může být libovolné platné jméno C++, například `fout`, `outFile`, `cgate` nebo `didi`.

Dále musíte tento objekt přiřadit určitému souboru. Můžete tak učinit pomocí metody `open()`. Předpokládejme například, že chcete pro výstup otevřít soubor `cookies`. Provedli byste to následovně:

```
fout.open("cookies");     // přiřadí objekt fout souboru cookies
```

Tyto dva kroky (vytvoření objektu a přiřazení souboru) můžete spojit do jediného příkazu pomocí jiného konstruktoru:

```
ofstream fout("cookies"); // vytvoří objekt fout a přiřadí ho souboru
                          //cookies
```

Když se dostane až sem, použijete `fout` (nebo libovolný jiný název) stejným způsobem jako objekt `cout`. Jestliže budete například chtít vložit do souboru slova `Dull Data`, může napsat následující příkaz:

```
fout << "Dull Data";
```

Vzhledem k tomu, že základní třídou třídy ofstream je třída `ostream`, můžete používat všechny metody třídy `ostream` včetně různých definic operátoru vložení, formátovacích metod a manipulátorů. Třída `ostream` používá pro výstup vyrovnávací paměť, takže program při vytvoření objektu jako `fout` přidělí vyrovnávací paměti určité místo. Jestliže vytvoříte objekty dva, vytvoří program dvě vyrovnávací paměti – pro každý objekt jednu. Objekt třídy `ostream` jako `fout` hromadí výstup z programu bajt po bajtu. Jakmile je vyrovnávací paměť zaplněna, přenese se obsah hromadně do cílového souboru. Vzhledem k tomu, že diskové jednotky jsou navrženy tak, aby se data přenášela po velkých blocích a ne bajt po bajtu, použití vyrovnávací paměti značně zvyšuje rychlost přenosu dat z programu do souboru.

Jestliže otevřete soubor pro výstup tímto způsobem, ale žádný soubor s tímto názvem neexistuje, vytvoří se soubor nový. Pokud otevřete pro výstup soubor s exitujícím názvem, dojde při otevření k jeho zkrácení, takže výstup začne do prázdného souboru. Později se dozvíte, jak otevřít existující soubor a zachovat jeho obsah.

Upozornění

Otevřete-li soubor pro výstup v implicitním režimu, dojde automaticky k jeho zkrácení na nulovou velikost a odstranění předchozího obsahu.

Pro čtení ze souboru musíte splnit podobné požadavky jako při zapisování do něho.

1. Vytvořit objekt třídy `ifstream`, který bude spravovat vstupní proud.
2. Přiřadit objekt určitému souboru.
3. Použít objekt stejným způsobem jako objekt `cin`.

Kroky, kterými toho dosáhnete, jsou podobné krokům, které musíte provést, abyste do souboru mohli zapisovat. Nejdříve samozřejmě musíte vložit hlavičkový soubor `fstream`. Potom deklarujete objekt třídy `ifstream` a přiřadíte ho názvu souboru. Učinit tak můžete dvěma příkazy nebo i jedním:

```
// dva příkazy
ifstream fin;                // vytvoří objekt třídy ifstream s názvem fin
fin.open("jellyjar.dat");    // otevře soubor jellyjar.dat pro čtení
// jeden příkaz
ifstream ifs("jamjar.dat"); // vytvoří objekt ifs a přiřadí ho souboru
                            //jamjar.dat
```

Nyní můžete objekty `fin` a `ifs` používat stejně jako objekt `cin`. Můžete například napsat následující příkazy:

```
char ch;
fin >> ch;                  // přečte znak ze souboru jellyjar.dat
char buf[80];
fin >> buf;                 // přečte ze souboru slovo
fin.getline(buf, 80);       // přečte ze souboru řádek
```

Vstup používá stejně jako výstup vyrovnávací paměť, takže při vytvoření objektu třídy `ifstream` jako `fin` se vytvoří vyrovnávací paměť pro vstup, kterou bude objekt `fin` spravovat. Stejně jako u výstupu se data díky vyrovnávací paměti přesouvají mnohem rychleji než bajt po bajtu.

Spojení se souborem skončí automaticky, jakmile skončí platnost objektů vstupních a výstupních proudů, například při ukončení programu. Ukončit spojení se souborem můžete také explicitně pomocí metody close():

```
fout.close();               // ukončí spojení výstupního proudu se souborem
fin.close();                // ukončí spojení vstupního proudu se souborem
```

Při ukončení takového spojení se proud nezruší; pouze se odpojí od souboru. Aparát pro správu proudu však zůstane na místě. Objekt `fin` například bude stále existovat spole-

ně s vyrovnávací pamětí pro vstup, kterou spravuje. Jak uvidíte později, můžete proud znovu spojit se stejným nebo jiným souborem.

Podívejme se na krátký příklad. Program ve výpisu 16.15 si vyžádá název souboru. Vytvoří soubor tohoto názvu, zapíše do něho nějaké informace a zavře ho. Při zavření souboru se vyrovnávací paměť vyprázdní, což je zárukou aktualizace souboru. Program potom stejný soubor otevře pro čtení a zobrazí jeho obsah. Všimněte si, že program používá objekty `fin` a `fout` stejným způsobem jako objekty `cin` a `cout`.

Výpis 16.15 file.cpp

```
// file.cpp — uložení do souboru
#include <iostream> // pro mnoho systémů není potřeba
using namespace std;
#include <fstream>
int main()
{
    char filename[20];
    cout << "Zadejte nazev noveho souboru: ";
    cin >> filename;
// vytvoření objektu výstupního proudu pro nový soubor s názvem fout
    ofstream fout(filename);
    fout << "Pouze pro vase oci!\n";          // zapisuje do souboru
    cout << "Zadejte svoje tajne cislo: ";   // zapisuje na obrazovku
    float secret;
    cin >> secret;
    fout << "Vase tajne cislo je " << secret << "\n";
    fout.close();               // uzavře soubor
// vytvoření objektu vstupního proudu pro nový soubor s názvem fin
    ifstream fin(filename);
    cout << "Zde je obsah souboru " << filename << ":\n";
    char ch;
    while (fin.get(ch))      // čte znak ze souboru
        cout << ch;           // a zapisuje ho na obrazovku
    cout << "Konec\n";
    fin.close();
    return 0;
}
```

Zde je ukázka běhu:

```
Zadejte nazev noveho souboru: pythag
Zadejte svoje tajne cislo: 3.14159
Zde je obsah souboru pythag:
Pouze pro vase oci!
Vase tajne cislo je 3.14159
Konec
```

Jestliže se podíváte do adresáře s vaším programem, měli byste najít soubor s názvem `pythag` a pomocí kteréhokoli textového editoru byste měli zobrazit stejný obsah, jako vypsal výstup programu.

Otevření více souborů

Můžete požadovat, aby program otevřel více než jeden soubor. Strategie otevření více souborů závisí na způsobu, jakým budou použity. Jestliže potřebuje současně otevřít dva soubory, musíte pro každý z nich vytvořit samostatný proud. Například program slučující dva setříděné soubory do třetího by potřeboval dva objekty třídy `ifstream` pro dva vstupní soubory a jeden objekt třídy `ofstream` pro soubor výstupní. Počet souborů, které lze současně otevřít, závisí na operačním systému, ale obvykle je jich kolem 20.

Můžete však naplánovat sekvenční zpracování skupiny souborů, například spočítat, kolikrát se nějaký název objeví v množině deseti souborů. V takovém případě stačí otevřít jediný proud a spojit ho postupně s každým souborem. Tím šetříte počítačové zdroje účinněji, než když pro každý soubor otevřete samostatný proud. Chcete-li použít tento postup, deklarujte objekt bez jeho inicializace a potom pomocí metody `open()` spojte proud se souborem. Následující kód ukazuje příklad, jak byste mohli řešit postupně čtení ze dvou souborů:

```
ifstream fin;          // vytvoří proud pomocí implicitního konstruktoru
fin.open("fat.dat");   // spojí proud se souborem fat.dat
...                    // nějaká činnost
fin.close();           // ukončí spojení se souborem fat.dat
fin.open("rat.dat");   // spojí proud se souborem rat.dat
...
fin.close();
```

Brzy si ukážeme příklad, ale nejdříve prozkoumejme techniku dodání seznamu souborů do programu takovým způsobem, při kterém je program bude moci zpracovat pomocí cyklu.

Zpracování příkazového řádku

Programy zpracovávající soubory používají pro jejich identifikaci často parametry z příkazového řádku. Parametry příkazového řádku jsou ty, které se na příkazovém řádku objeví, když napíšete příkaz. Kdybyste například chtěli spočítat počet slov v nějakých souborech v systému UNIX, napsali byste tento příkaz:

```
wc report1 report2 report3
```

Wc zde představuje název programu a `report1`, `report2` a `report3` jsou názvy souborů předané programu jako parametry z příkazového řádku.

V C++ existuje mechanismus, umožňující programu přístup k těmto parametrům. Ve funkci main() použijte následující alternativní hlavičku:

```
int main(int argc, char *argv[])
```

Parametr `argc` vyjadřuje počet parametrů na příkazovém řádku. Do tohoto počtu je zahrnut také samotný název příkazu. Proměnná `argv` je ukazatelem na ukazatel na typ `char`. Zní to trochu abstraktně, ale s `argv` můžete zacházet, jako by to bylo pole ukazatelů na parametry příkazového řádku, přičemž prvek `argv[0]` je ukazatelem na první znak řetězce obsahujícího název příkazu, `argv[1]` ukazatelem na první znak řetězce obsahujícího

první parametr příkazového řádku a tak dále. To znamená, že `argv[0]` je prvním řetězcem na příkazovém řádku a tak dále. Předpokládejme například následující příkazový řádek:

```
wc report1 report2 report3
```

V tomto případě bude mít `argc` hodnotu 4, `argv[0]` bude obsahovat `wc`, `argv[1]` `report1` a tak dále. Následující cyklus vytiskne každý parametr příkazového řádku na samostatný řádek:

```
for (int i = 1; i < argc; i++)
   cout << argv[i] << "\n";
```

Při počáteční hodnotě `i = 1` se vytisknou pouze parametry příkazového řádku; pokud by počáteční hodnota `i = 0`, vytiskl by se také název příkazu.

Parametry příkazového řádku se pochopitelně využívají v takových operačních systémech jako DOS nebo UNIX, ovšem můžete je použít i v některých jiných prostředích:

- Mnoho prostředí DOSu a Windows IDE (integrated development evironment, integrované vývojové prostředí) obsahuje volbu pro parametry příkazového řádku. Běžně můžete procházet řadou nabídek, které vás dovedou k poli, do kterého zadáte parametry příkazového řádku. Přesný sled kroků se u jednotlivých prodejců a upgradů liší, takže se musíte řídit dokumentací.
- IDE DOSu a mnoho Windows IDE dokáže vytvořit spustitelné soubory, které běží pod DOSem nebo v okně DOSu v obvyklém DOSovském režimu s příkazovým řádkem.
- Pod Symantec C++ pro Macintosh a pod Metrowerks CodeWarrior pro Macintosh budete moci parametry příkazového řádku simulovat, pokud do programu vložíte následující kód:

```
...
#include <console.h>      // pro emulaci parametrů příkazového řádku
int main(int argc, char *argv[])
{
            argc = ccommand(&argv); // ano ccommand, ne command
            ...
```

Když program spustíte, umístí funkce `ccommand()` na obrazovku dialogové okno obsahující pole, do kterého zadáte parametry příkazového řádku. Také bude moci simulovat přesměrování.

Program ve výpisu 16.16 kombinuje techniku příkazového řádku s technikami proudu a počítá znaky obsažené v seznamu souborů na příkazovém řádku.

Výpis 16.16 count.cpp

```
// count.cpp — počítání znaků v seznamu souborů
#include <iostream>
using namespace std;
#include <fstream>
#include <cstdlib>           // nebo stdlib.h
// #include <console.h>      // pro Macintosh
```

```
int main(int argc, char * argv[])
{
    // argc = ccommand(&argv);      // pro Macintosh
    if (argc == 1)              // pokud nejsou parametry, skonči
    {
    cerr << "Pouzity soubor: " << argv[0] << " \n";
    exit(1);
    }
    ifstream fin;               // otevře proud
    long count;
    long total = 0;
    char ch;
    for (int file = 1; file < argc; file++)
    {
        fin.open(argv[file]);   // spojí proud se souborem argv[file]
        count = 0;
        while (fin.get(ch))
      count++;
        cout << count << " znaku v souboru " << argv[file] << "\n";
        total += count;
        fin.clear();            // potřeba pro některé implementace
        fin.close();            // odpojí soubor
    }
    cout << total << " znaku ve vsech souborech\n";
    return 0;
}
```

Kompatibilita:

V některých implementacích je nutné použít metodu `fin.clear()`, jiné se bez ní obejdou. Závisí to na tom, zda spojení nového souboru s objektem třídy `fstream` automaticky nastavuje stav proudu na dobrý nebo ne. Použití metody `fin.clear()` rozhodně neuškodí, i když není potřeba.

V systému DOS byste mohli program z výpisu 16.16 zkompilovat do spustitelného souboru `count.exe`. Ukázkový běh by mohl vypadat takto:

```
C>count
Pouzity soubor: c:\count.exe
C>count paris rome
3580 znaku v souboru paris
4886 znaku v souboru rome
8466 znaku ve všech souborech
C>
```

Všimněte si, že program používá pro chybovou zprávu proud `cerr`. Za zmínku stojí, že zpráva používá prvek `argv[0]` místo názvu `count.exe`:

```
cerr << "Pouzity soubor: " << argv[0] << " \n";
```

Když v tomto případě změníte název spustitelného souboru, použije program automaticky nový název.

Předpokládejme, že programu pro počítání znaků předáte nesprávný název souboru. Příkaz pro vstup `fin.get(ch)` neuspěje, cyklus `while` okamžitě skončí a program ohlásí 0 znaků. Program však můžete upravit a testovat úspěšnost spojení proudu se souborem. Je to jedna z věcí, kterou se budeme zabývat v následující části.

Kontrola stavu proudu a metoda is_open()

Třídy souborových proudů v C++ dědí položku stavu proudu od třídy `ios_base`. Tato již dříve probíraná položka ukládá informace odrážející stav proudu: vše v pořádku, bylo dosaženo konce souboru, neúspěšná vstupně-výstupní operace a tak dále. Pokud je vše v pořádku, má stav proudu hodnotu nula (žádné zprávy jsou dobré zprávy). Různé jiné stavy jsou zaznamenány nastavením určitých bitů na hodnotu 1. Třídy souborových proudů dědí také metody třídy `ios_base`, které stav proudu hlásí, a které byly shrnuty v tabulce 16.5. Pomocí nich můžete stav proudu sledovat. Například pomocí metody `good()` zjistíte, že všechny bity stavu proudu jsou vyčištěny. Novější implementace C++ však nabízejí lepší způsob kontroly, zda byl nějaký soubor otevřen – metodu `is_open()`. Program ve výpisu 16.16 můžete upravit tak, aby hlásil nesprávné názvy souborů a skočil na další soubor, pokud přidáte následujícím způsobem volání `fin.is_open()` do cyklu `for`:

```
for (int file = 1; file < argc; file++)
{
        fin.open(argv[file]);  // spojí proud se souborem argv[file]
  // přidaný kód
  if (!fin.is_open())
  {
        cerr << "Nelze otevrit soubor " argv[file] << "\n";
        continue;
  }
  // konec přidání kódu
        count = 0;
        while (fin.get(ch))
       count++;
        cout << count << " znaku v souboru " << argv[file] << "\n";
        total += count;
        fin.clear();              // potřeba pro některé implementace
        fin.close();              // odpojí soubor
}
```

Volání `fin.is_open()` vrátí při neúspěšném otevření `fin.open()` hodnotu `false`. V takovém případě vás program na problém upozorní a příkaz `continue` způsobí, že program zbytek cyklu `for` přeskočí a začne další cyklus.

Upozornění

V minulosti se testy na úspěšnost otevření souboru prováděly následovně:

```
if (!fin.good()) … // neúspěšné otevření
if (!fin) …        // neúspěšné otevření
```

Jestliže se objekt `fin` použije v testovací podmínce, převede se na `false`, pokud `fin.good()` vrátí false, a v opačném případě na `true`, takže obě formy jsou rovnocenné. Může však nastat situace, kterou tento test neodhalí, a tou je pokus otevřít soubor v nevhodném režimu (viz část Režimy souboru). Metoda `is_open()` tuto formu chyby zachytí a stejně tak i chyby odhalené metodou `good()`. Starší implementace ji však neobsahují.

Režimy souboru

Režim souboru popisuje, jakým způsobem se soubor použije: pro čtení, pro zápis, pro přidání dat na konec souboru a podobně. Při spojování proudu se souborem, buď inicializováním objektu souborového proudu názvem souboru nebo pomocí metody `open()`, můžete uvést další parametr specifikující režim souboru:

```
ifstream fin("banjo", mode1);      // konstruktor s parametrem režimu
ofstream fout();
fout.open("harp", mode2);          // metoda open() s parametrem režimu
```

Třída `ios_base` definuje typ `openmode` reprezentující režim; je typu `bitmask`, stejně jako typy `fmtflags` a `iostate`. (Dříve byl typu `int`.) Pro specifikaci režimu nabízí třída `ios_base` několik definovaných konstant. Jejich seznam a význam je uveden v tabulce 16.8. Vstup ze souboru a výstup do souboru v C++ prošel několika změnami, aby byl kompatibilní s ANSI C.

Tabulka 16.8 Režimy souboru a konstanty.

Konstanta	Význam
ios_base::in	Otevře soubor pro čtení.
ios_base::out	Otevře soubor pro zápis.
ios_base::ate	Při otevření hledá konec souboru.
ios_base::app	Přidá obsah na konec souboru.
ios_base::trunc	Zkrátí existující soubor na nulovou délku.
ios_base::binary	Binární soubor.

Jestliže konstruktory tříd `ifstream` a `ofstream` a metoda `open()` mají dva parametry, proč nám stačilo používat v předchozích příkladech pouze jeden? Asi jste uhodli, že prototypy těchto členských funkcí obsahují pro druhý parametr (parametr pro režim souboru) implicitní hodnotu. Například metoda třídy `ifstream open()` a konstruktor používají jako implicitní hodnotu pro parametr režimu konstantu `iso_base::in` (otevření pro čtení), zatímco metoda třídy `ofstream open()` a konstruktor používají jako implicitní hodnotu konstanty `ios_base::out | ios_base::trunc` (otevření pro zápis a zkrácení souboru na nulo-

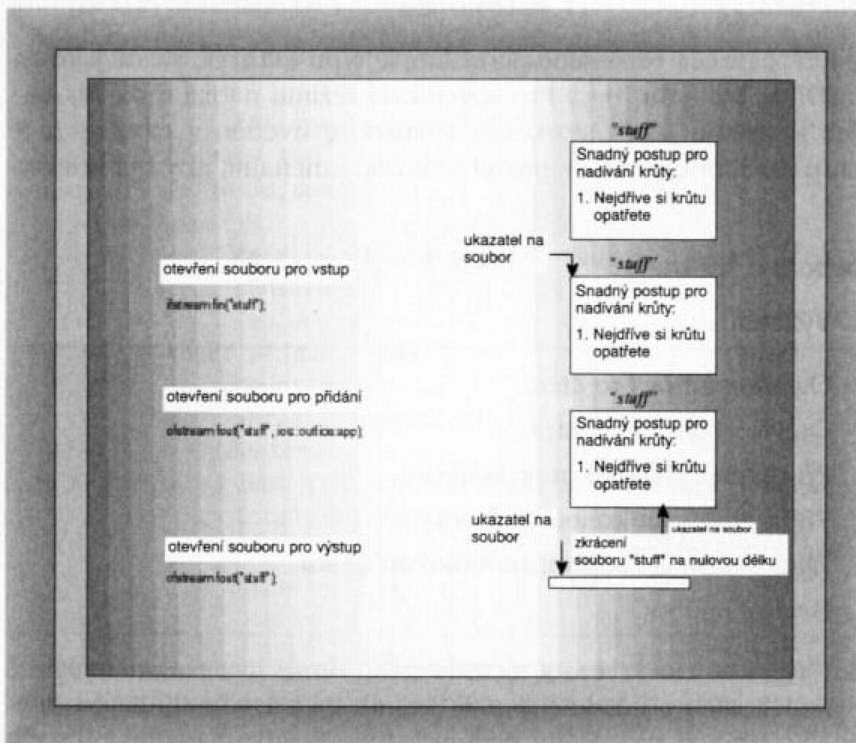
vou délku). Použitý bitový operátor (|) kombinuje dvě bitové hodnoty do jedné, kterou lze použít pro nastavení obou bitů. Třída `fstream` hodnotu implicitního režimu neobsahuje, takže při vytvoření objektu této třídy ji musíte dodat explicitně.

Všimněte si, že příznak `ios_base::trunc` znamená, že při otevření existujícího souboru pro výstup bude tento soubor zkrácen na nulovou délku; to znamená, že předchozí obsah bude odstraněn. Takové chování sice chvályhodně minimalizuje nebezpečí, že se vám nebude dostávat prostoru na disku, ale zřejmě si dokážete představit situace, kdy nebudete chtít obsah souboru při jeho otevření smazat. Jazyk C++ samozřejmě nabízí i jiné možnosti. Jestliže chcete například obsah souboru zachovat a přidat na konec souboru nová data, můžete použít režim `ios_base::app`:

```
ofstream fout("bagels", ios_base::out | ios_base::app);
```

Operátor | opět režimy kombinuje. Výraz `ios_base::out | ios_base::app` znamená, že se vyvolají režimy pro výstup a přidání (viz obrázek 16.6).

Ve starších implementacích můžete očekávat nějaké odlišnosti. Některé například umožňují ve výše uvedeném příkladě `ios_base::out` vynechat, jiné ne. Jestliže nepoužíváte implicitní režim, je nejbezpečnější dodat všechny prvky režimu explicitně. Některé kompilátory nepodporují všechny možnosti uvedené v tabulce 16.7 a některé nabízejí možnosti, které v této tabulce uvedeny nejsou. Důsledkem těchto odlišností je, že možná budete muset v následujících příkladech provést nějaké úpravy, aby ve vašem systému fungovaly. Dobrou zprávou je, že vývoj standardu C++ nabízí větší uniformitu.



Obrázek 16.6 Některé režimy otevření souboru

Standard C++ definuje části pro vstup ze souboru a výstup do souboru podle standardu ANSI C. Příkaz

```
ifstream fin(filename, c++mode);
```

je implementován, jako kdyby byla použita funkce jazyka C `fopen()`:

```
fopen(filename, cmode);
```

Zde *cmode++* představuje hodnotu typu openmode, například `ios_base::in`, a *cmode* je odpovídající řetězec jazyka C pro označení režimu, například „`r`". V tabulce 16.9 jsou uvedeny odpovídající režimy v jazycích C++ a C. Všimněte si, že samotná konstanta `ios_base::out` způsobí zkrácení na nulovou délku, ale v kombinaci s konstantou `ios_base::in` zkrácení nezpůsobí. Neuvedené kombinace, například `ios_base::in [vn] ios_base::trunc`, nedovolí soubor otevřít.

Tabulka 16.9 Režimy pro otevření souboru v jazycích C++ a C.

Režim v C++	Režim v C	Význam
ios_base::in	„r"	Otevře soubor pro čtení.
ios_base::out	„w"	(Stejné jako ios_base::out | ios_base::trunc).
ios_base::out | ios_base::trunc	„w"	Otevře soubor pro zápis, existující soubor zkrátí na nulovou délku.
ios_base::out | ios_base::app	„a"	Otevře soubor pro zápis a pouze pro přidání.
ios_base::in | ios_base::out	„r+"	Otevře soubor pro čtení a zápis, přičemž zapisovat bude možné kdekoli v souboru.
ios_base::in | ios_base::out | ios_base::trunc	„w+"	Otevře soubor pro čtení a zápis, přičemž soubor nejdříve zkrátí na nulovou délku.
c++mode | ios_base::binary	„cmodeb"	Otevře soubor v režimu c++mode nebo odpovídajícím režimu cmode a binárním; například z ios_base::in | ios_base::binary se stane „rb".
c++mode | ios_base::ate	„cmode"	Otevře soubor v příslušném režimu a přejde na konec souboru. Jazyk C místo režimu kódu používá samostatné volání funkce. Například ios_base::in | ios_base::ate se přeloží na daný režim a volání funkce C fseek(file, 0, SEEK_END).

Všimněte si, že pomocí výrazů `ios_base::ate` a `ios_base::app` se dostanete (přesněji řečeno ukazatel na soubor) na konec právě otevřeného souboru. Rozdíl mezi oběma spočívá v tom, že `ios_base::app` umožní data na konec souboru pouze přidávat, zatímco režim `ios_base::ate` pouze nastaví ukazatel na konec souboru.

Samozřejmě existuje mnoho možných kombinací režimů. Na několik typických se podíváme.

Přidání dat do souboru

Začneme programem, který přidává data na konec souboru. Bude udržovat soubor obsahující seznam hostů. Pokud soubor existuje, zobrazí nejdříve jeho aktuální obsah. Pro ověření existence souboru lze po otevření použít metodu `is_open()`. Potom program otevře soubor pro výstup pomocí režimu `ios_base::app` a požádá o vstup z klávesnice, který bude do souboru přidávat. Nakonec aktualizovaný obsah souboru zobrazí. Dosažení těchto cílů ilustruje program ve výpisu 16.17. Všimněte si, jak pomocí metody `is_open()` testuje úspěšnost otevření souboru.

Kompatibilita:

Vstup ze souboru a výstup do souboru byly dříve asi nejméně standardizovaným aspektem jazyka C++ a mnoho starších kompilátorů zcela neodpovídá současnému standardu. Některé například používají režimy jako `nocreate`, které součástí aktuálního standardu nejsou a také jen některé vyžadují volání funkce `fin.clear()` před opětovným otevřením stejného souboru pro čtení.

Výpis 16.17 append.cpp

```
// append.cpp — přidání informací do souboru
#include <iostream>
using namespace std;
#include <fstream>
#include <cstdlib>      // (nebo stdlib.h) kvůli funkci exit()
const char * file = "guests1.dat";
const int Len = 40;
int main()
{
    char ch;
// zobrazí aktuální obsah
    ifstream fin;
    fin.open(file);
    if (fin.is_open())
    {
        cout << "Tady je aktualni obsah souboru "
             << file << ":\n";
        while (fin.get(ch))
            cout << ch;
    }
    fin.close();
// přidání nových jmen
    ofstream fout(file, ios::out | ios::app);
    if (!fout.is_open())
    {
        cerr << "Soubor " << file << " nelze otevrit pro vystup.\n";
        exit(1);
    }
    cout << "Zadavejte jmena hostu (prazdny radek pro ukonceni):\n";
```

```
    char name[Len];
    cin.get(name, Len);
    while (name[0] != '\0')
    {
        while (cin.get() != '\n')
            continue;    // odstraní \n a dlouhé řádky
        fout << name << "\n";
        cin.get(name, Len);
    }
    fout.close();
// zobrazení aktualizovaného souboru
    fin.clear();    // pro některé kompilátory není nutné
    fin.open(file);
    if (fin.is_open())
    {
        cout << "Tady je aktualizovany obsah souboru "
             << file << ":\n";
        while (fin.get(ch))
            cout << ch;
    }
    fin.close();
    return 0;
}
```

Zde je první ukázka běhu. V této chvíli nebyl ještě soubor `guests.dat` vytvořen, takže program původní obsah nezobrazuje.

```
Zadavejte jmena hostu (prazdny radek pro ukonceni):
Sylvester Ballone
Phil Kates
Bill Ghan

Tady je aktualizovany obsah souboru guests.dat:

Sylvester Ballone
Phil Kates
Bill Ghan
```

Při dalším spuštění programu však již soubor `guests.dat` existuje, takže program původní obsah zobrazí. Také si všimněte, že nová data původní nenahrazují, ale jsou přidána k původnímu obsahu.

```
Tady je aktualni obsah souboru guests.dat:
Sylvester Ballone
Phil Kates
Bill Ghan
Zadavejte jmena hostu (prazdny radek pro ukonceni):
Greta Greppo
LaDonna Mobile
Fannie Mae

Tady je aktualizovany obsah souboru guests.dat:
```

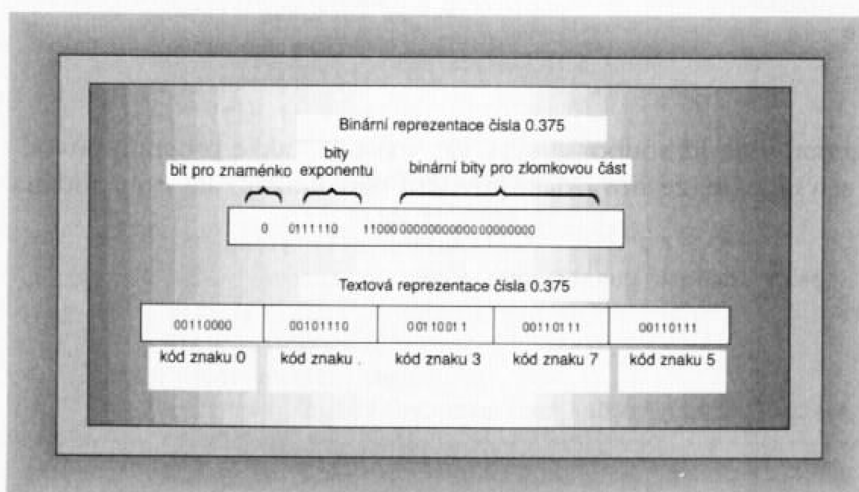
```
Sylvester Ballone
Phil Kates
Bill Ghan
Greta Greppo
LaDonna Mobile
Fannie Mae
```

Obsah souboru `guests.dat` byste měli být schopni přečíst libovolným textovým editorem včetně editoru, který používáte pro psaní kódu.

Binární soubory

Data můžete do souboru ukládat v textovém nebo v binárním formátu. Textový formát znamená, že vše ukládáte jako text, dokonce i čísla. Například uložení čísla –2.324216e+07 v textovém tvaru znamená uložení 13 znaků, které se pro jeho napsání použijí. K tomu je potřeba konverze vnitřní reprezentace čísla v pohyblivé řádové čárce do znakové formy a to právě dělá operátor vkládání `<<`. Binární formát však znamená uložení hodnoty vnitřní reprezentace. Místo uložení znaků se tedy uloží (obvykle) 64 bitová reprezentace hodnoty typu `double`. Pro znak znamená binární reprezentace totéž co reprezentace textová – binární reprezentaci znakového kódu ASCII (nebo ekvivalentu). U čísel se však binární reprezentace od textové značně liší (viz obrázek 16.7).

Každý formát má své výhody. Textový se snadno čte. Textový soubor můžete číst a editovat pomocí obyčejného editoru nebo textového procesoru. Textový soubor snadno přenesete z jednoho počítačového systému na jiný. Pro čísla je mnohem přesnější formát binární, protože ukládá přesnou vnitřní reprezentaci hodnoty. Nedochází k chybám při konverzi nebo při zaokrouhlování. Uložení dat v binárním formátu může být mnohem rychlejší, protože nedochází ke konverzím a data lze ukládat po velkých blocích. Binární formát také obvykle v závislosti na povaze dat zabírá méně místa. Přenést data do jiného systému však může být problém, jestliže tento nový systém používá pro hodnoty odlišnou vnitřní reprezentaci. V takovém případě budete muset napsat program, který převede data z jednoho formátu do druhého.



Obrázek 16.7 Binární a textová reprezentace čísla s pohyblivou řádovou čárkou

Podívejme se na konkrétnější příklad. Uvažujte definici a deklaraci následující struktury:

```
struct planet
{
   char name[20];  // název planety
   double population;     // počet obyvatel
   double g;              // tíhové zrychlení
};
planet pl;
```

Budete-li chtít uložit strukturu `pl` v textovém tvaru, napíšete následující kód:

```
ofstream fout("planets.dat", ios_base::app);
fout << pl.name << " " << pl.population << " " << pl.g << "\n";
```

Všimněte si, že každý člen struktury musíte dodat explicitně pomocí operátoru výběru člena a sousední data musíte z důvodu čitelnosti oddělit. Pokud by struktura obsahovala například 30 položek, byla by taková práce únavná.

Jestliže budete chtít stejné informace uložit v binárním formátu, napíšete následující kód:

```
ofstream fout("planets.dat", ios_base::app | ios_base::binary);
fout.write( (char *) &pl, sizeof pl);
```

Tento kód uloží celou strukturu jako jedinou jednotku pomocí vnitřní reprezentace dat počítače. Soubor nebudete moci číst jako text, ale informace budou uloženy kompaktněji a přesněji. A takový kód se určitě snadněji píše. Tento přístup vyžadoval dvě změny:

- použití binárního režimu pro soubor,
- použití členské funkce `write()`.

Podívejme se na tyto změny podrobněji.

Některé systémy, jako například DOS, podporují dva formáty souborů: textový a binární. Jestliže chcete data uložit v binárním tvaru, použijete binární formát souboru. V C++ tak učiníte, když režim souboru nastavíte pomocí konstanty `ios_base::binary`. Chcete-li vědět proč, přečtěte si pojednání Binární soubory a soubory textové v následující poznámce.

Binární soubory a soubory textové

Při použití binárního režimu u souboru program přenáší data z paměti do souboru a naopak a přitom nedochází k žádnému skrytému převodu. V implicitním textovém režimu tomu tak být nemusí. Uvažujte například textové soubory v systému DOS. Nový řádek v nich reprezentuje kombinace dvou znaků: návrat vozíku (carriage return) a nový řádek (linefeed). V textových souborech systému je nový řádek reprezentován pomocí návratu vozíku. UNIXovské soubory reprezentují nový řádek pomocí nového řádku (linefeed). Z důvodu přenositelnosti program napsaný v C++ v systému DOS automaticky převádí znak nového řádku na carriage return a linefeed, pokud zapisuje do souboru otevřeného v textovém režimu; program v systému Macintosh při zápisu do souboru převádí znak nového řádku na carriage return. Při čtení textového souboru tyto programy zase převedou znak nového řádku do formy používané v C++. Textový formát by u binárních dat mohl způsobit problémy, neboť bajt uprostřed hodnoty typu `double` by mohl mít stejný bitový vzorek jako kód ASCII pro znak nového řádku. Odlišnosti jsou

také v rozpoznávání konce souboru. Při ukládání dat v binárním formátu byste tedy měli používat binární režim souboru. (Systémy UNIX mají pouze jeden režim souborů, takže na nich se binární režim rovná režimu textovému.)

Chcete-li data místo v textovém režimu uložit v režimu binárním, můžete použít členskou funkci `write()`. Vzpomeňte si, že tato metoda kopíruje stanovený počet bajtů z paměti do souboru. Dříve jsme ji používali pro zkopírování textu, ale dokáže zkopírovat jakýkoli datový typ bajt po bajtu bez konverze. Předáte-li jí například adresu proměnné typu `long` a budete chtít zkopírovat 4 bajty, zkopíruje do souboru přesně 4 bajty hodnoty typu `long` a neprovede konverzi na text. Adresu však budete muset přetypovat na ukazatel na typ `char`. Stejný postup můžete použít pro zkopírování celé struktury `planet`. Chcete-li získat počet bajtů, použijte operátor `sizeof`:

```
fout.write( (char *) &pl, sizeof pl);
```

Tento příkaz použije adresu struktury `pl` a zkopíruje do souboru, spojeného s objektem `fout`, 36 bajtů (hodnota výrazu `sizeof pl`) od začátku této adresy.

Jestliže budete chtít informace ze souboru získat, použijete metodu `read()` s objektem třídy `ifstream`:

```
ifstream fin("planets.dat", ios_base::binary);
fin.read((char *) &pl, sizeof pl);
```

Metoda `read()` zkopíruje počet `sizeof pl` bajtů ze souboru do struktury `pl`. Stejný postup lze použít u tříd, které nepoužívají virtuální funkce. V takovém případě se uloží pouze datové položky, zatímco metody ne. Jestliže třída virtuální funkce obsahuje, zkopíruje se také skrytý ukazatel na tabulku ukazatelů na virtuální funkce. Vzhledem k tomu, že při dalším spuštění programu může být tabulka virtuálních funkcí uložena na jiném místě, způsobí zkopírování informací o původních ukzatelích ze souboru do objektů zmatek.

Tip

Členské funkce `read()` a `write()` se vzájemně doplňují. Funkci `read()` používejte pro získání dat, která byla do souboru zapsána pomocí funkce `write()`.

Program ve výpisu 16.18 používá tyto metody k vytvoření binárního souboru a čtení z něho. Podobá se programu z výpisu 16.17, ale místo operátoru vkládání a metody `get()` používá metody `read()` a `write()`. K formátování výstupu na obrazovku také využívá manipulátory.

Kompatibilita:

Ačkoli je pojem binární soubor součástí ANSI C, některé implementace C a C++ režim binárních souborů nepodporují. Důvodem takového opomenutí je skutečnost, že některé systémy mají jen jeden typ souboru a binární operace jako `read()` a `write()` můžete použít u standardního formátu souboru. Jestliže tedy vaše implementace konstantu `ios_base::binary` jako platnou odmítne, stačí ji z programu vynechat. Pokud vaše implementace nepod-

poruje manipulátor `fixed`, můžete použít `cout.setf(ios_base::fixed, ios_base::floatfield)`. Symantec C++ 8 vyžaduje nahradit dva výskyty výrazu

```
while (fin.read((char *) &pl, sizeof pl))
```

následujícím výrazem:

```
while (fin.read((char *) &pl, sizeof pl) && !fin.eof())
```

Výpis 16.18 binary.cpp

```
#include <iostream> // většina systémů nepožaduje
using namespace std;
#include <fstream>
#include <iomanip>
#include <cstdlib>  // (nebo stdlib.h) kvůli funkci exit()
inline void eatline() { while (cin.get() != '\n') continue; }
struct planet
{
    char name[20];       // název planety
    double population;   // počet obyvatel
    double g;            // tíhové zrychlení
};
const char * file = "planets.dat";
int main()
{
    planet pl;
    cout << fixed << right;
// zobrazení počátečního obsahu
    ifstream fin;
    fin.open(file, ios::in |ios::binary);   // binární soubor
    // Poznámka: některé systémy neakceptují režim ios::binary
    if (fin.is_open())
    {
    cout << "Soucasny obsah souboru "
        << file << ":\n";
    while (fin.read((char *) &pl, sizeof pl))
        {
        cout << setw(20) << pl.name << ": "
               << setprecision(0) << setw(12) << pl.population
               << setprecision(2) << setw(6) << pl.g << "\n";
        }
    }
    fin.close();
// přidání nových údajů
    ofstream fout(file, ios::out | ios::app | ios::binary);
    //Poznámka: některé systémy neakceptují režim ios::binary
    if (!fout.is_open())
    {
        cerr << "Soubor " << file << " nelze otevrit pro vystup:\n";
        exit(1);
```

```
    }
    cout << "Zadejte nazev planety (prazdny radek pro ukonceni):\n";
    cin.get(pl.name, 20);
    while (pl.name[0] != '\0')
    {
        eatline();
        cout << "Zadejte pocet obyvatel planety: ";
        cin >> pl.population;
        cout << "Zadejte tihove zrychleni planety: ";
        cin >> pl.g;
        eatline();
        fout.write((char *) &pl, sizeof pl);
        cout << "Zadejte nazev planety (prazdny radek pro ukonceni):\n";
        cin.get(pl.name, 20);
    }
    fout.close();
// zobrazeni aktualizovaného souboru
    fin.clear();    // některé implementace nevyžadují
    fin.open(file, ios::in | ios::binary);
    if (fin.is_open())
    {
    cout << "Novy obsah souboru "
        << file << ":\n";
    while (fin.read((char *) &pl, sizeof pl))
        {
        cout << setw(20) << pl.name << ": "
              << setprecision(0) << setw(12) << pl.population
              << setprecision(2) << setw(6) << pl.g << "\n";
        }
    }
    fin.close();
    return 0;
}
```

Zde je ukázka počátečního běhu:

```
Zadejte nazev planety (prazdny radek pro ukonceni):
Zeme
Zadejte počet obyvatel planety: 5932000000
Zadejte tihove zrychleni planety: 9.81
Zadejte nazev planety (prazdny radek pro ukonceni):
Novy obsah souboru planets.dat:
                Zeme:   5932000000   9.81
```

Zde je ukázka následujícího běhu:

```
Soucasny obsah souboru
     Zeme:   5932000000   9.81
Zadejte nazev planety (prazdny radek pro ukonceni):
Billova planeta
Zadejte počet obyvatel planety: 23020020
Zadejte tihove zrychleni planety: 8.82
Zadejte nazev planety (prazdny radek pro ukonceni):
```

```
Novy obsah souboru planets.dat:
                 Zeme:      5932000000    9.81
Billova planeta:      23020020       8.82
```

Hlavní rysy programu jste již viděli, ale prozkoumejme znovu původní myšlenku. Po přečtení hodnoty proměnné `g` stuktury `planet` používá program tento kód (ve formě vložené funkce `eatline()`):

```
while (cin.get() != '\n') continue;
```

Tento příkaz čte a odkládá znaky ze vstupu, dokud nepřečte znak nového řádku. Uvažujte v cyklu následující příkaz pro vstup:

```
cin.get(pl.name, 20);
```

Pokud by byl ve vstupu ponechán znak nového řádku, načetl by tento příkaz prázdný řádek a cyklus by skončil.

Přímý přístup

Náš poslední příklad se bude týkat přímého přístupu. Přímý přístup znamená, že se můžete přímo přesunout na libovolné místo v souboru, aniž byste soubor museli procházet sekvenčně. Tento přístup se často používá v databázových souborech. Program uchovává samostatný indexový soubor obsahující údaje o umístění dat v hlavním souboru. V takovém případě lze na dané místo skočit přímo, přečíst data a třeba je i upravit. Nejsnáze je takový přístup použitelný, pokud soubor obsahuje kolekci záznamů stejné délky. Každý záznam reprezentuje příbuznou kolekci dat. Například v předcházejícím příkladě by každý záznam souboru reprezentoval všechny údaje o určité planetě. Záznam souboru odpovídá v programu nejčastěji struktuře nebo třídě.

Náš příklad založíme na programu pro binární soubor z výpisu 16.18, přičemž využijeme skutečnosti, že struktura `planet` nabízí vzorek pro záznam souboru. Aby programování získalo tvůrčí napětí, bude soubor v příkladu otevřen v režimu pro čtení a zápis, takže záznam bude možné číst i upravovat. Dosáhnete toho vytvořením objektu třídy `fstream`. Třída `fstream` je odvozena od třídy `iostream`, která je zase založena na třídách `istream` a `ostream`, takže dědí metody obou tříd. Také dědí dvě vyrovnávací paměti, jednu pro vstup a jednu pro výstup, a správu obou vyrovnávacích pamětí synchronizuje. To znamená, že když program ze souboru čte nebo do něj zapisuje, vstupní ukazatel ve vstupní vyrovnávací paměti i výstupní ukazatel ve výstupní vyrovnávací paměti se pohybují v tandemu.

Program v příkladu bude provádět následující činnost:

1. Zobrazí aktuální obsah souboru `planets.dat`.
2. Zeptá se, který záznam se má upravit.
3. Záznam upraví.
4. Zobrazí obsah aktualizovaného souboru.

Ambicióznější program by použil nabídku a pomocí cyklu by vás nechal donekonečna vybírat ze seznamu činností, ale naše verze provede každou činnost pouze jednou. Tento zjednodušený přístup vám umožní prozkoumat několik aspektů souborů otevřených pro čtení i zápis, aniž byste museli zabřednout do záležitostí, týkajících se návrhu programu.

Upozornění

Tento program předpokládá, že soubor `planets.dat` již existuje a byl vytvořen programem `binary.cpp`.

První otázka, kterou je třeba zodpovědět, se týká režimu souboru. Pro čtení ze souboru potřebujete režim `ios_base::in`. Pro binární vstup či výstup musíte mít režim `ios_base::binary`. (Na některých nestandardních systémech ho budete moci nebo i muset vynechat.) Abyste do souboru mohli zapisovat, potřebujete režim `ios_base::out` nebo `ios_base::app`. Druhý režim však programu umožňuje data na konec souboru pouze přidat. Zbytek souboru bude otevřen pouze pro čtení; původní data tedy můžete číst, ale ne upravovat – musíte tedy použít režim `ios_base::out`. Jak je patrno z tabulky 16.9, současné použití režimů pro vstup a výstup poskytuje režim pro čtení a zápis, takže musíte pouze přidat binární prvek. Jak jsme se dříve zmínili, režimy zkombinujete pomocí operátoru `|`. K dosažení úspěchu tedy potřebujete následující příkaz:

```
finout.open(file, ios_base::in | ios_base::out | ios:base_binary);
```

Dále potřebujete způsob, jak se v souboru pohybovat. Třída `fstream` dědí k tomu účelu dvě metody: metoda `seekg()` nastavuje na stanovené místo v souboru vstupní ukazatel a metoda `seekp()` ukazatel výstupní. (Vzhledem k tomu, že třída `fstream` používá pro bezprostřední uložení dat vyrovnávací paměť, ukazují tyto ukazatele ve skutečnosti do ní a ne do souboru samotného.) Metodu `seekg()` můžete rovněž použít u objektu třídy ifstream a metodu `seekg()` u objektu třídy `ofstream`. Zde jsou prototypy metody `seekg()`:

```
basic_istream<charT, traits>& seekg(off_type, ios_base::seekdir);
basic_istream<charT, traits>& seekg(pos_type);
```

Jak vidíte, jedná se o šablony. V této kapitole se bude specializace šablon používat pro typ `char`. Pro tento typ jsou oba prototypy rovnocenné následujícím:

```
istream & seekg(streamoff, ios_base::seekdir);
istream & seekg(streampos);
```

První prototyp reprezentuje pozici v souboru měřenou v bajtech jako offset od místa udávaného druhým parametrem. Druhý prototyp reprezentuje pozici v souboru měřenou v bajtech od začátku souboru.

Eskalace typů

Když byl jazyk C++ mladý, byl život pro metody `seekg()` jednodušší. Typy `streamoff` a `streampos` byly definovány pomocí příkazu `typedef` pro některé standardní typy, například `long`. Avšak snaha vytvořit přenositelný standard se musela potýkat se skutečností, že celočíselný parametr nemusí v některých systémech obsahovat dostatek informací, takže bylo povoleno, aby `streamoff` a `streampos` byly strukturální nebo třídní typy a umožňovaly některé základní operace, například použití celočíselné hodnoty jako hodnoty inicializační. Dále byla původní třída `istream` nahrazena šablonou `basic_istream` a `streampos` a `streamoff` byly nahrazeny šablonovými typy `pos_type` a `off_type`. Avšak `streampos` a `streamoff` nadále existují jako specializace typu `char` typů `pos_type` a `off_type`. Podobně můžete použít typy `wstreampos` a `wstreamoff`, jestliže metodu `seekg()` použijete u objektu třídy `wistream`.

Podívejme se na parametry prvního prototypu metody `seekg()`. Hodnoty typu `streamoff` jsou použity pro změření offsetu v bajtech od určitého místa v souboru. Parametr `streamoff` reprezentuje pozici v souboru v bajtech měřenou jako offset od jednoho ze tří míst. (Typ může být definován jako celé číslo nebo třída.) Parametr `seek_dir` je další celočíselný typ definovaný společně se třemi možnými hodnotami ve třídě `ios_base`. Konstanta `ios_base::beg` označuje offset měřený od začátku souboru, konstanta `ios_base::cur` offset měřený od aktuální pozice a konstanta `ios_base::end` offset měřený od konce souboru.

Zde je několik příkladů jejich volání. Předpokládáme, že `fin` je objekt třídy `ifstream`:

```
fin.seekg(30, ios_base::beg);    // 30 bajtů od začátku souboru
fin.seekg(-1, ios_base::cur);    // zpět o jeden bajt
fin.seekg(0, ios_base::end);     // běž na konec souboru
```

Nyní se podívejme na druhý prototyp. Hodnoty typu `streampos` označují pozici v souboru. Tímto typem může být třída, ale v takovém případě obsahuje jeden konstruktor s parametrem `streamoff` a jeden konstruktor s celočíselným parametrem a cestu pro převedení obou typů na hodnoty `streampos`. Hodnota `streampos` představuje absolutní místo v souboru, měřeno od jeho počátku. Na místo udávané hodnotou `streampos` se můžete dívat jako na vzdálenost v bajtech, měřenou od začátku souboru, přičemž první bajt má hodnotu 0. Příkaz

```
fin.seekg(112);
```

tedy nastaví ukazatel na bajt 112, což je 113. bajt v souboru. Chcete-li zjistit aktuální pozici ukazatele v souboru, můžete ve vstupních proudech použít metodu `tellg()` a v proudech výstupních metodu `tellp()`. Obě vrací hodnotu streampos reprezentující aktuální pozici v bajtech měřenou od začátku souboru. Když vytvoříte objekt třídy fstream, pohybují se vstupní a výstupní ukazatele v tandemu, takže metody `tellg()` a `tellp()` vrátí stejnou hodnotu. Použijete-li však ve stejném souboru pro správu vstupního proudu objekt třídy istream a pro správu výstupního proudu objekt třídy ostream, budou se vstupní a výstupní ukazatel pohybovat nezávisle na sobě a metody `tellp()` a `tellg()` budou vracet hodnoty rozdílné.

Pomocí metody `seekg()` můžete ukazatel nastavit na začátek souboru. Uvádíme část kódu, který otevírá soubor, nastaví ukazatel na začátek souboru a zobrazí jeho obsah:

```
fstream finout;   // proudy pro čtení a zápis
finout.open(file, ios_base::in | ios_base::out, ios_base::binary);
// Poznámka: v některých systémech UNIX je nutné konstantu ios_base::
// binary vynechat
int ct = 0;
if (finout.is_open())
{
    finout.seekg(0);// jdi na začátek
    cout << "Soucasny obsah souboru: " << file << "\n";
    while (finout.read(char *) &pl, sizeof pl))
    {
            cout << ct++ << ": " << setw(20) << pl.name << ": "
            << setprecision(0) << setw(12) << pl.population
            << setprecision(2) << setw(6) << pl.g << "\n";
    }
```

```
    if (finout.eof())
        finout.clear(); // vyčistí příznak konce souboru eof
    else
    {
        cerr << "Chyba pri cteni souboru " << file << ".\n";
        exit(1);
    }
  }
else
  {
    cerr << "Soubor " << file << " se nepodarilo otevrit.\n";
    exit(2);
  }
```

Tento kód se podobá začátku programu z výpisu 16.18, ale jsou v něm některé změny a něco bylo přidáno. Za prvé, jak bylo popsáno, používá program objekt třídy `fstream` s režimem pro čtení a zápis a pomocí metody `seekg()` nastavuje ukazatel na začátek souboru. (V tomto příkladě to nebylo nezbytně nutné, ale ukazuje to použití metody `seekg()`). Dále program provádí menší změny, týkající se číslování zobrazovaných záznamů. Potom je přidána následující důležitá část:

```
    if (finout.eof())
        finout.clear(); // vyčistí příznak konce souboru eof
    else
    {
        cerr << "Chyba pri cteni souboru " << file << ".\n";
        exit(1);
    }
```

Problém je, že jakmile program přečte a zobrazí celý soubor, nastaví prvek `eofbit`. To ho ubezpečí, že práce se souborem skončila a další čtení ze souboru nebo zápis do něho jsou nemožné. Pomocí metody `clear()` vynulujete stav proudu a bit `eof` vypnete. Program nyní může k souboru opět přistupovat. Část s `else` ošetřuje možnost, že program skončí čtení ze souboru z nějakého jiného důvodu, než je dosažení konce souboru, například chybou hardwaru.

Dalším krokem je identifikace záznamu, který se má změnit, a provedení změny. Za tímto účelem program požádá uživatele o číslo záznamu a vynásobením tohoto čísla počtem bajtů v záznamu získá číslo počátečního bajtu požadovaného záznamu. Jestliže rec představuje číslo záznamu, je požadované číslo bajtu `rec * sizeof pl`:

```
cout << "Zadejte cislo zaznamu, ktery chcete zmenit: ";
long rec;
cin >> rec;
eatline(); // zbaví se znaku nového řádku
if (rec < 0 || rec >= ct)
{
  cer << "Neplatne cislo zaznamu\n";
  exit(3);
}
```

```
streampos place = rec * sizeof pl; // konverze na typ streampos
finout.seekg(place);        // přímý přístup
```

Proměná ct reprezentuje počet záznamů; jestliže zadáte vyšší číslo záznamu než je jejich počet, program skončí chybou.

Program potom zobrazí aktuální záznam:

```
finout.read(char *) &pl, sizeof pl);
cout << "Vybrany zaznam:\n";
cout << rec << ": " setw(20) << pl.name << ": "
<< setprecision(0) << setw(12) << pl.population
<< setprecision(2) << setw(6) << pl.g << "\n";
if (finout.eof())
  finout.clear(); // vyčistí příznak konce souboru eof
```

Zobrazený záznam vám umožní změnit:

```
cout << "Zadejte nazev planety: ";
cin.get(pl.name, 20);
eatline();
cout << "Zadejte pocet obyvatel planety: ";
cin >> pl.population;
cout << "Zadejte tihove zrychleni planety: ";
cin >> pl.g
finout.seekp(place);        // návrat
finout.write((char *) &pl, sizeof pl) << flush;
if (finout.fail())
{
  cerr << "Chyba pri zapisu\n";
  exit(5);
}
```

Vyprázdněním vyrovnávací paměti program zaručí aktualizaci souboru.

Nakonec program aktualizovaný soubor zobrazí – pomocí metody `seekg()` nastaví ukazatel na začátek souboru. Výpis 16.19 obsahuje úplný program. Nezapomeňte, že program předpokládá existenci souboru `planets.dat`, vytvořeného pomocí programu `binary.cpp`.

Kompatibilita:

Starší implementace se pravděpodobně budou chovat nestandardně. Některé systémy nerozeznávají příznak `binary`. Symantec C++ přidá místo nahrazení označeného záznamu záznam nový a také budete muset nahradit (dvakrát) výraz

```
while (fin.read(char *) &pl, sizeof pl))
```

výrazem

```
while (fin.read((char *) &pl, sizeof pl) && !fin.eof())
```

Výpis 16.19 random.cpp

```
// random.cpp - přímý přístup k binárnímu souboru
#include <iostream>     // většina systémů nepožaduje
using namespace std;
#include <fstream>
#include <iomanip>
#include <cstdlib>      // nebo stdlib.h kvůli funkci exit()
struct planet
{
    char name[20];       // název planety
    double population;  // počet obyvatel
    double g;            // tíhové zrychlení
};
const char * file = "planets.dat";// předpokládá existenci souboru
                                  // (například pomocí programu binary.cpp)
inline void eatline() { while (cin.get() != '\n') continue; }
int main()
{
    planet pl;
    cout << fixed;
// zobrazení počátečního obsahu
    fstream finout;     // proudy pro čtení a zápis
    finout.open(file,ios::in | ios::out |ios::binary);
    // Poznámka: V některých systémech UNIX je nutné vynechat | ios::bina-
ry
    int ct = 0;
    if (finout.is_open())
    {
        finout.seekg(0);    // jdi na začátek
        cout << "Soucasny obsah souboru "
             << file << ":\n";
        while (finout.read((char *) &pl, sizeof pl))
        {
            cout << ct++ << ": " << setw(20) << pl.name << ": "
            << setprecision(0) << setw(12) << pl.population
            << setprecision(2) << setw(6) << pl.g << "\n";
        }
        if (finout.eof())
            finout.clear(); // vyčisti příznak konce souboru eof
        else
        {
            cerr << "Chyba pri cteni souboru " << file << ".\n";
            exit(1);
        }
    }
    else
    {
        cerr << "Soubor " << file << " se nepodarilo otevrit.\n";
        exit(2);
    }
```

```
// změna záznamu
    cout << "Zadejte cislo zaznamu, ktery chcete zmenit: ";
    long rec;
    cin >> rec;
    eatline();              // zbaví se znaku nového řádku
    if (rec < 0 || rec >= ct)
    {
        cerr << " Neplatne cislo zaznamu\n";
        exit(3);
    }
    streampos place = rec * sizeof pl;  // konverze na typ streampos
    finout.seekg(place);      // přímý přístup
    if (finout.fail())
    {
        cerr << "Chyba pri hledani zaznamu\n";
        exit(4);
    }
    finout.read((char *) &pl, sizeof pl);
    cout << "Pozadovany zaznam:\n";
    cout << rec << ": " << setw(20) << pl.name << ": "
    << setprecision(0) << setw(12) << pl.population
    << setprecision(2) << setw(6) << pl.g << "\n";
    if (finout.eof())
        finout.clear();      // vyčistí příznak konce souboru eof
    cout << "Zadejte nazev planety: ";
    cin.get(pl.name, 20);
    eatline();
    cout << "Zadejte pocet obyvatel planety: ";
    cin >> pl.population;
    cout << "Zadejte tihove zrychleni planety: ";
    cin >> pl.g;
    finout.seekp(place);     // návrat
    finout.write((char *) &pl, sizeof pl) << flush;
    if (finout.fail())
    {
        cerr << "Chyba pri zapisu\n";
        exit(5);
    }
// zobrazení aktualizovaného souboru
    ct = 0;
    finout.seekg(0);              // jdi na začátek souboru
    cout << "Novy obsah souboru " << file << ":\n";
    while (finout.read((char *) &pl, sizeof pl))
    {
        cout << ct++ << ": " << setw(20) << pl.name << ": "
                << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << "\n";
    }
    finout.close();
    return 0;
}
```

Zde je ukázka běhu při použití souboru `planets.dat` s několika přidanými záznamy:

```
Soucasny obsah souboru planets.dat:
0:        Zeme:             5932000000    9.81
1:        Billova planeta:  23020020      8.82
2:        Trantor:          58000000000   15.03
3:        Trellan:          4256000       9.62
4:        Freestone:        3845120000    8.68
5:        Taanagoot:        350000002     10.23
6:        Marin:            232000        9.79
Zadejte cislo zaznamu, ktery chcete zmenit: 2
Pozadovany zaznam:
2:        Trantor:          58000000000   15.03
Zadejte nazev planety: Trantor
Zadejte počet obyvatel planety: 59500000000
Zadejte tihove zrychleni planety: 10.53
Novy obsah souboru planets.dat:
0:        Zeme:             5932000000    9.81
1:        Billova planeta:  23020020      8.82
2:        Trantor:          59500000000   10.53
3:        Trellan:          4256000       9.62
4:        Freestone:        3845120000    8.68
5:        Taanagoot:        350000002     10.23
6:        Marin:            232000        9.79
```

Techniky použité v programu byste mohli rozšířit tak, abyste mohli přidávat nové záznamy a rušit existující. Pokud byste program rozšiřovali, bylo by dobré přeorganizovat ho pomocí tříd a funkcí. Strukturu `planet` byste například mohli změnit na definici třídy a potom operátor vkládání `<<` přetížit tak, aby výraz `cout << pl` zobrazil datové položky třídy formátované stejným způsobem jako v příkladě.

Formátování `incore`

Skupina metod třídy `iostream` podporuje vstupní a výstupní operace mezi programem a terminálem. Skupina metod třídy `fstream` používá stejné rozhraní pro vstupní a výstupní operace mezi programem a souborem. Knihovna jazyka C++ také obsahuje skupinu metod souboru `sstream`, které používají stejné rozhraní pro vstupní a výstupní operace mezi programem a objektem třídy `string`. Můžete tedy použít stejné metody třídy `ostream`, které jste používali u objektu `cout`, a zapsat formátované informace do objektu třídy `string`, a metody třídy `istream` jako `getline()` pro načtení informací z objektu třídy `string`. Proces čtení formátovaných informací z objektu třídy `string` nebo zápis formátovaných informací do objektu třídy `string` se nazývá formátování *incore*. Ve stručnosti se na tyto prostředky podíváme. (Skupina metod třídy `sstream` podporujících třídu `string` nahrazuje skupinu metod z hlavičkového souboru `strstream.h`, podporujících pole typu `char`.)

Hlavičkový soubor `sstream` definuje třídu `ostringstream`, odvozenou od třídy `ostream`. (Existuje také třída `wostringstream`, založená na třídě `wostream` pro sadu širokých znaků.) Vytvoříte-li objekt třídy `ostringstream`, můžete do něho zapisovat informace a objekt

je uloží. U objektu třídy `ostringstream` můžete použít stejné metody jako u objektu `cout`. Můžete tedy napsat následující kód:

```
ostringstream outstr;
double price = 55.00;
char * ps = " za kopii konceptu standardu C++!";
outstr.precision(2);
outstr << fixed;
outstr << "Zaplatte pouze Kc " << price << ps << end;
```

Formátovaný text jde do vyrovnávací paměti a objekt zvětší její velikost podle potřeby pomocí dynamického přidělení paměti. Třída `ostringstream` obsahuje členskou funkci `str()`, která do vyrovnávací paměti vrací inicializovaný objekt třídy `string`:

```
string mesg = outstr.str();   // vrací řetězec s formátovanými informacemi
```

Použití metody `str()` objekt „zmrazí" a další zápis do něho nebude možný.

Krátký příklad představuje program ve výpisu 16.20.

Výpis 16.20 strout.cpp

```
// strout.cpp — formátování incore do výstupního proudu
#include <iostream>
using namespace std;
#include <sstream>
#include <string>
int main()
{
    ostringstream outstr;   // spravuje řetězcový proud

    string hdisk;
    cout << "Jak se jmenuje vas pevny disk? ";
    getline(cin, hdisk);
    int cap;
    cout << "Jaka je jeho kapacita v MB? ";
    cin >> cap;
    // zápis formátovaných informací do řetězcového proudu
    outstr << "Pevny disk " << hdisk << " ma kapacitu "
           << cap << " megabytu.\n";
    string result = outstr.str();   // uloží výsledek
    cout << result;                 // zobrazí obsah
    return 0;
}
```

Zde je ukázka běhu:

```
Jak se jmenuje vas pevny disk? Rocky
Jaka je jeho kapacita v MB? 2425
Pevny disk Rocky ma kapacitu 2425 megabytu.
```

Třída `istringstream` umožňuje použít skupinu metod třídy `istream` pro čtení dat z objektu třídy `istringstream`, kterou lze inicializovat objektem třídy `string`. Předpokládejme,

že facts je objekt třídy `string`. Chcete-li vytvořit objekt třídy istringstream spojený s tímto objektem, napište následující kód:

```
istringstream instr(facts);          // inicializuje proud pomocí facts
```

Potom načtěte data z objektu instr pomocí metod třídy `istream`. Obsahuje-li například objekt instr množství celých čísel ve znakovém formátu, můžete je číst následovně:

```
int n;
int sum = 0;
while (instr << n)
   sum += num;
```

Program ve výpisu 16.21 čte po slovech obsah řetězce pomocí přetíženého operátoru >>.

Výpis 16.21 strin.cpp

```
// strin.cpp - formátované čtení ze znakového pole
#include <iostream>
using namespace std;
#include <sstream>
#include <string>
int main()
{
    string lit = "Byl tmavy a bourlivy den a "
                 " cely mesic jasne zaril. ";
    istringstream instr(lit);   // použití vyrovnávací paměti pro vstup
    string word;;
    while (instr >> word)       // čtení po slovech
        cout << word << endl;
    return 0;
}
```

Zde je výstup programu:

```
Byl
tmavy
a
bourlivy
den
a
cely
mesic
jasne
zaril.
```

Stručně řečeno, díky třídám `istringstream` a `ostringstream` získáte schopnost používat metody tříd `istream` a `ostream` při práci se znakovými údaji uloženými v řetězcích.

Co dále

Jestliže jste se v knize dostali až sem, měli byste pravidlům jazyka C++ dobře rozumět. Je to však jen začátek učení se tohoto jazyka. Druhou fází je naučit se jazyk efektně používat a k tomu vede delší cesta. Nejlepší situací je pracovní nebo studijní prostředí, při kterém se dostanete do kontaktu s dobrým kódem v C++ a programátory. Když teď C++ znáte, můžete číst také knihy, které se soustředí na pokročilá témata a na objektově orientované programování. Seznam některých zdrojů najdete v příloze H.

Jedním ze slibů OOP je usnadnit vývoj a zvýšit spolehlivost velkých projektů. Jednou z podstatných aktivit přístupu OOP je vynalézat třídy, které reprezentují situaci (říká se jí *public domain*), kterou modelujete. Vzhledem k tomu, že skutečné problémy jsou často složité, může nalezení vhodné množiny tříd představovat výzvu. Vytvořit složitý systém zcela od začátku se obvykle nepodaří; místo toho je lepší použít řešení založené na opakovaném postupném vývoji. Badatelé v této oblasti dosud vyvinuli několik technik a strategií. Obzvláště je důležité při analýze a ve fázích návrhu co možná nejvíce projekt opakovaně procházet a vyvíjet, a ne psát nebo přepisovat samotný kód.

Mezi dvě obvyklé techniky patří *use-case analysis* a karty *CRC*. Při use-case analysis vytvoří vývojový tým seznam obvyklých způsobů neboli scénářů, tak jak očekávají, že se bude konečný systém používat, přičemž identifikují prvky, akce a zodpovědnosti, ze kterých vyplynou možné třídy a jejich vlastnosti. Karty CRC (zkratka pro Class/Responsibilities/Collaborators) představují jednoduchý způsob pro analýzu takových scénářů. Vývojový tým vytvoří pro každou třídu kartu s indexem. Karta obsahuje název třídy, její odpovědnost, jako jsou reprezentovaná data a prováděné akce, a spolupracující třídy – další třídy, se kterými musí tato třída spolupracovat. Potom může tým scénář procházet za pomoci rozhraní poskytnutého kartami CRC. Toto může vést k navržení nových tříd, přesunu zodpovědnosti a tak dále.

Větší váhu mají systematické metody pro práci na celém projektu. Mezi nejnovější patří Unified Modeling Language (jednotný jazyk pro modelování) neboli *UML*. Nejedná se o programovací jazyk, ale spíše o jazyk reprezentující analýzu a návrh programovacího projektu. Vyvinuli ho Grady Booch, Jim Rumbaugh a Ivar Jacobson, kteří byli hlavními vývojáři tří dřívějších modelovacích jazyků: Booch Method, OMT (Object Modeling Technique) a OOSE (Object-Oriented Software Engineering). UML je jejich vývojový následník.

Kromě dalšího celkového porozumění jazyku C++ také můžete studovat specifické knihovny tříd. Microsoft, Borland a Symantec například nabízejí rozsáhlé knihovny tříd, usnadňující programování v prostředí Windows; Symantec a Metrowerk nabízejí podobné prostředky pro programování v prostředí Macintosh.

Shrnutí

Proud je tok bajtů do programu nebo z programu. Vyrovnávací paměť je dočasná oblast v paměti obsahující data, která se chová jako prostředník mezi programem a souborem nebo jinými vstupně-výstupními zařízeními. Informace mezi vyrovnávací pamětí a souborem lze přenášet pomocí velkých bloků dat ve velikosti, která nejvíce vyhovuje takovým

zařízením jako jsou diskové jednotky. Informace mezi vyrovnávací pamětí a programem lze přenášet bajt po bajtu, což je často pro zpracování prováděné v programu výhodnější. Při zpracování vstupu spojí C++ proud ve vyrovnávací paměti s programem a se zdrojem vstupu. Podobně zpracovává výstup – proud ve vyrovnávací paměti spojí s programem a s cílem výstupu. Soubory `iostream` a `fstream` tvoří vstupně-výstupní třídy knihovny, která definuje bohatou sadu tříd pro správu proudů. Programy v C++ obsahující soubor `iostream` otevírají automaticky osm proudů a spravují je pomocí osmi objektů. Objekt `cin` spravuje standardní vstupní proud, který je implicitně spojen se standardním vstupním zařízením, obvykle klávesnicí. Objekt `cout` spravuje standardní výstupní proud, který je implicitně spojen se standardním výstupním zařízením, obvykle monitorem. Objekty `cerr` a `clog` spravují proudy s vyrovnávací pamětí i bez ní a jsou spojeny se standardním chybovým zařízením, obvykle monitorem. Tyto čtyři objekty mají své protějšky pracující se sadou širokých znaků; nazývají se `wcin`, `wcout`, `wcerr` a `wclog`.

Knihovna vstupně-výstupních tříd nabízí celou řadu užitečných metod. Třída `istream` definuje verze operátoru vytažení (>>), které rozeznávají všechny základní typy v C++ a převádějí na tyto typy znakový vstup. Metody `get()` a `getline()` poskytují další podporu pro vstup jednotlivých znaků a pro vstup řetězce. Podobným způsobem definuje třída `ostream` verze operátoru vložení (<<), který rozeznává všechny základní typy v C++ a převádí je na odpovídající znakový výstup. Další podporu pro výstup jednoho znaku poskytuje metoda `put()`. Třídy `wistream` a `wostream` poskytují podobnou podporu širokým znakům.

Způsob, jakým bude program výstup formátovat, můžete ovládat pomocí metod třídy `ios_base` a pomocí manipulátorů (funkce, které lze při vkládání řetězit), definovaných v souborech `iostream` a `iomanip`. Tyto metody a manipulátory vám umožní stanovit základ číselné soustavy, počet zobrazených desetinných míst, systém použitý pro zobrazení hodnot s pohyblivou řádovou čárkou a další prvky.

Soubor `fstream` obsahuje definice tříd, které obohacují metody třídy `iostream` o vstupně -výstupní metody pracující se souborem. Od třídy `istream` je odvozena třída `ifstream`. Jestliže spojíte objekt třídy `ifstream` se souborem, můžete pro čtení ze souboru používat metody třídy `istream`. Podobně spojíte-li se souborem objekt třídy `ofstream`, můžete metody této třídy používat pro zápis do souboru. A pokud spojíte se souborem objekt třídy `fstream`, budete moci při práci se souborem používat metody pro vstup i výstup.

Když spojujete soubor s proudem, můžete dodat název souboru již při inicializaci objektu proudu, nebo nejdříve objekt proudu vytvořit a potom ho spojit se souborem pomocí metody `open()`. K ukončení spojení mezi proudem a souborem slouží metoda `close()`. Konstruktory třídy a metoda `open()` mají volitelný druhý parametr pro označení režimu souboru. Režim souboru například určuje, zda bude soubor otevřen pro čtení, pro zápis, čtení i zápis, zda se při otevření pro zápis zkrátí na nulovou délku, zda pokus otevřít neexistující soubor způsobí chybu a zda se použije textový nebo binární režim.

Textový soubor ukládá všechny informace v textovém tvaru. Numerické hodnoty jsou například převedeny na znakové reprezentace. Tento režim podporují obvyklé operátory vložení a vytažení a také metody `get()` a `getline()`. Binární soubor ukládá všechny informace pomocí stejné binární reprezentace, jakou vnitřně používá počítač. Binární soubory ukládají data, především hodnoty s pohyblivou řádovou čárkou, přesněji a kompaktněji než soubory textové, ale nejsou tak přenositelné. Binární vstup a výstup podporují metody `read()` a `write()`.

Pro přímý přístup k souborům v C++ slouží funkce `seekg()` a `seekp()`. Tyto metody umožňují nastavit ukazatel v souboru relativně vzhledem k začátku souboru, konci souboru nebo aktuální pozici. Aktuální pozici v souboru hlásí metody `tellg()` a `tellp()`.

Hlavičkový soubor `sstream` definuje třídy `istringstream` a `ostringstream`, díky nimž můžete pomocí metod tříd `istream` a `ostream` číst informace ze řetězce a formátovat informace do řetězce zapisované.

Opakovací otázky

1. Jakou roli hraje při vstupně-výstupních operacích v C++ soubor `iostream`?
2. Proč bude muset program při zadání čísla 121 provést konverzi?
3. Jaký je rozdíl mezi standardním výstupním zařízením a standardním chybovým zařízením?
4. Proč dokáže objekt `cout` zobrazit různé typy C++, aniž byste mu museli pro každý typ zadávat explicitní instrukce?
5. Jaká vlastnost v definicích metod pro výstup umožňuje řetězení výstupu?
6. Napište program, který si vyžádá celé číslo, a potom ho zobrazí v desítkovém, osmičkovém a šestnáctkovém tvaru. Každý tvar zobrazte na samostatném řádku do pole o šířce 15 znaků a použijte prefixy C++, označující základ číselné soustavy.
7. Napište program, který si vyžádá uvedené informace a zformátuje je uvedeným způsobem:

```
Zadejte jmeno: Josef Novak
Zadejte hodinovou mzdu: 120
Zadejte pocet odpracovanych hodin: 8.5
Prvni format:
                          Josef Novak: Kc 120.00:  8.5
Druhy format:
Josef Novak                       : Kc120.00        :8.5
```

8. Uvažujte následující program:

```
//rq16-8.cpp
#include <iostream>
using namespace std;
int main()
{
        char ch;
        int ct1 = 0;
        cin >> ch;
        while (ch != 'q')
        {
                ct1++;
                cin >> ch;
        }
        int ct2 = 0;
        cin.get(ch);
```

```
        while (ch != 'q')
        {
                ct2++;
                cin.get(ch);
        }
        cout << "ct1 = " << ct1 << "; ct2 = " << ct2 << "\n";
        return 0;
}
```

Co se vytiskne při následujícím vstupu?

```
Vidim jedno q <Enter>
Vidim jedno q <Enter>
```

`<Enter>` zde označuje stisknutí klávesy Enter.

9. Oba následující příkazy čtou a odkládají znaky až do načtení, včetně znaku nového řádku. Jak se chování prvního příkazu liší od chování druhého?

```
while (cin.get() != '\n')
    continue;
cin.ignore(80, '\n');
```

Programovací cvičení

1. Napište program, který bude počítat znaky ze vstupu až do prvního výskytu znaku $ a tento znak ponechá ve vstupním proudu.
2. Napište program, který zkopíruje vstup z klávesnice (až do simulovaného konce souboru) do souboru, jehož název zadáte na příkazovém řádku.
3. Napište program, který zkopíruje jeden soubor do druhého. Názvy souborů si program vyžádá z příkazového řádku a pokud se mu nepodaří soubor otevřít, podá zprávu.
4. Napište program, který otevře dva textové soubory pro vstup a jeden pro výstup. Program zřetězí odpovídající řádky vstupních souborů, přičemž jako oddělovač použije mezeru, a výsledek zapíše do výstupního souboru. Pokud bude jeden ze souborů kratší, zkopírují se do výstupního souboru i zbývající řádky delšího souboru. Předpokládejte například, že první vstupní soubor má tento obsah:

```
eggs kites donuts
balloons hammers
stones
```

Druhý vstupní soubor bude mít následující obsah:

```
zero lassitude
finance drama
```

Obsah výsledného souboru pak bude následující:

```
eggs kites donuts zero lassitude
balloons hammers finance drama
stones
```

5. Petr a Pavel chtějí pozvat přátele na večírek podobně jako v kapitole 15, cvičení 5, ale nyní chtějí program, který bude používat soubory. Požádají vás o program, který bude provádět následující činnost:

   Přečte seznam Petrových přátel z textového souboru petr.dat. Každé jméno v seznamu bude na jiném řádku. Jména se uloží do kontejneru a zobrazí se setříděná.

   Přečte seznam Pavlových přátel z textového souboru pavel.dat. Každé jméno v seznamu bude na jiném řádku. Jména se uloží do kontejneru a zobrazí se setříděná.

   Oba seznamy sloučí, vyloučí duplicity a výsledek uloží do souboru petpav.dat, každé jméno na jiný řádek.

6. Uvažujte definice tříd z programovacího cvičení 13.5. Pokud jste cvičení dosud neprovedli, učiňte tak nyní. Potom postupujte následovně:

   Napište program, který používá standardní vstupně-výstupní zařízení a vstupní a výstupní soubory společně s daty typu `employee`, `manager`, `fink` a `highfink` tak, jak byla definována v programovacím cvičení 13.5. Program by měl po vzoru programu z výpisu 16.17 umožnit přidávat do souboru nová data. Při prvním spuštění by si měl vyžádat data od uživatele, všechny položky zobrazit a uložit do souboru. Při následném použití by měl program data nejdříve přečíst a zobrazit, potom uživateli umožnit nějaká data přidat a nakonec je všechna zobrazit. Jediný rozdíl by měl spočívat v tom, že data budou spravována pomocí pole ukazatelů na typ `employee`. Ukazatel může ukazovat na objekt `employee` nebo na objekty kteréhokoli ze tří odvozených typů. Pro lepší kontrolu programu použijte malé pole:

```
const int MAX = 10;        // ne víc než 10 objektů
...
employee * pc[MAX];
```

Při vstupu z klávesnice by měl program použít nabídku a nabídnout uživateli výběr z typů objektu, který chce vytvořit. Nabídka použije pro vytvoření nového objektu požadovaného typu přepínač `switch` a adresu objektu přiřadí ukazateli v poli `pc`. Objekt potom použije virtuální funkci `setall()` a požádá uživatele o zadání náležitých údajů:

```
pc[i]->setall();  // vyvolá funkci odpovídající typu objektu
```

Pro uložení dat do souboru napište virtuální funkci `writeall()`:

```
for (i = 0; i < index; i++)
        pc[i]->writeall(fout);  // fout je objekt třídy ostream spojený
                                // s výstupním souborem
```

Poznámka

V tomto cvičení použijte pro vstup a výstup textové soubory, nikoli binární. (Virtuální objekty obsahují bohužel ukazatele na tabulky ukazatelů na virtuální funkce a funkce `write()` kopíruje tyto informace do souboru. Objekt naplněný ze souboru pomocí funkce `read()` získá pro ukazatele na funkce podivné hodnoty, které v chování virtuálních funkcí vytvoří dokonalý zmatek.) Každé pole dat oddělte od následujícího pomocí znaku nového řádku; při vstupu budou

pole snáze identifikovatelná. Pokud byste přece jen chtěli použít binární soubory, nezapisujte objekty jako celky. Místo toho napište metody třídy, které použijí pro každou jednotlivou položku třídy funkce `write()` a `read()`. Tímto způsobem program uloží do souboru pouze určená data.

Obtížnou část představuje načtení dat ze souboru. Problémem je, jak má program poznat, že příští čtenou položkou bude objekt typu `employee`, `manager`, `fink` nebo `highfink`? Jedním z možných řešení je označit data celočíselnou položkou, určující typ následujícího objektu. Při vstupu ze souboru potom program číslo přečte a pomocí příkazu `switch` vytvoří pro získaná data objekt odpovídající typu:

```
enum classkind{Employee, Manager, Fink, Highfink}; // v hlavičce třídy
...
int classtype;
while ((fin >> classtype).get(ch)) {      // znak nového řádku odděluje
int od //dat
  switch(classtype) {
    case Employee : pc[i] = new employee;
                  : break;
```

Potom můžete pomocí ukazatele vyvolat virtuální funkci `getall()` a informace přečíst:

```
pc[i++]->getall();
```

DODATEK A

Základy číselných soustav

Naše metoda pro zapisování čísel je založena na mocnině 10. Uvažujte například číslo 2468. Číslice 2 reprezentuje 2 tisíce, číslice 4 reprezentuje 4 stovky, číslice 6 reprezentuje 6 desítek a číslice 8 reprezentuje 8 jednotek:

2468 = 2x1000 + 4x100 + 6x10 + 8x1

Jeden tisíc je 10x10x10, což lze zapsat jako 10^3 neboli 3. mocninu 10. Pomocí této notace můžeme předcházející vztah zapsat takto:

$2468 = 2x10^3 + 4x10^2 + 6x10^1 + 8x10^0$

Vzhledem k tomu, že naše číselná notace je založena na mocnině 10, označujeme ji jako základ 10 neboli desítkovou. Stejně dobře lze jako číselný základ vybrat jiné číslo. Jazyk C++ umožňuje pro psaní celých čísel zápis se základem 8 (osmičkový) a se základem 16 (šestnáctkový). (Poznámka: 10^0 je 1, stejně jako libovolné nenulové číslo s mocninou nula.)

Čísla v osmičkové soustavě

Čísla v osmičkové soustavě jsou založena na mocnině 8, takže při zápisu v osmičkové soustavě se používají číslice 0 – 7. Jazyk C++ používá pro označení osmičkového zápisu prefix 0. Číslo 0177 je tedy osmičková hodnota. Ekvivalentní hodnotu v desítkové soustavě najdete pomocí mocnin 8:

```
0177 (osmičkově)  = 1x8^2 + 7x8^1 + 7x8^0
                  = 1x64 + 7x8 + 7x1
                  = 127 (desítkově)
```

Operační systém UNIX používá osmičkovou reprezentaci hodnot často a z tohoto důvodu ji jazyky C++ a C poskytují.

Čísla v šestnáctkové soustavě

Čísla v šestnáctkové soustavě jsou založena na mocnině 16. To znamená, že číslo 10 v šestnáctkové soustavě reprezentuje hodnotu 16 + 0 neboli 16. Chcete-li šestnáctkově vyjádřit hodnoty mezi 9 a 16, potřebujete více číslic. Standardní šestnáctkový zápis používá k tomuto účelu písmena a – f. Jak je vidět z tabulky A.1, jazyk C++ akceptuje verze těchto znaků s malými i velkými písmeny.

Tabulka A. 1 Číslice v šestnáctkové soustavě.

Šestnáctková číslice	Hodnota v desítkové soustavě	Šestnáctková číslice	Hodnota v desítkové soustavě
a nebo A	10	d nebo D	13
b nebo B	11	e nebo E	14
c nebo C	12	f nebo F	15

Pro označení šestnáctkového zápisu používá C++ notace 0x nebo 0X. Zápis 0x2B3 tedy označuje hodnotu v šestnáctkové soustavě. Chcete-li najít její ekvivalent v soustavě desítkové, můžete vyhodnotit mocniny 16:

```
0x2B3 (šestnáctkově)        = 2x16^2 + 11x16^1 + 3x16^0
                            = 2x256 + 11x16 + 3x1
                            = 691 (desítkově)
```

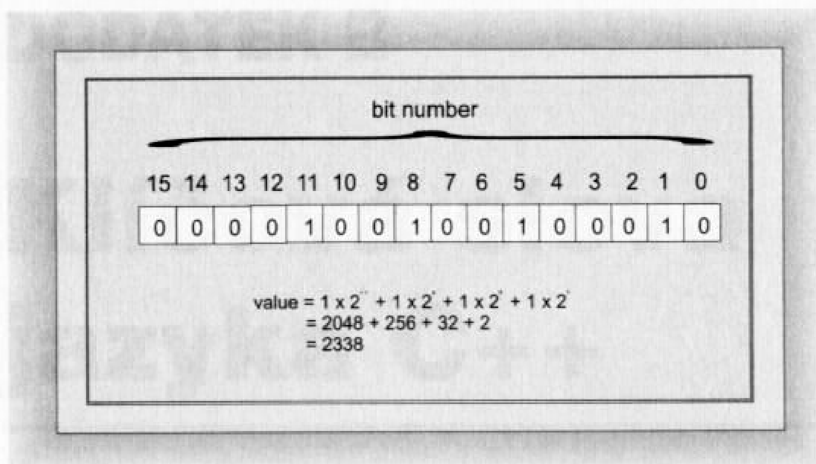
V hardwarové dokumentaci se pomocí zápisu v šestnáctkové soustavě často vyjadřují hodnoty jako místo v paměti nebo čísla portů.

Čísla ve dvojkové soustavě

Ať už použijete pro zápis čísla notaci desítkovou, osmičkovou nebo šestnáctkovou, počítač číslo uloží jako binární hodnotu, čili se základem 2. Zápis ve dvojkové soustavě používá pouze dvě číslice, 0 a 1. Například 10011011 je binární číslo. Všimněte si však, že binární notaci pro zápis čísla jazyk C++ neposkytuje. Čísla ve dvojkové soustavě jsou založena na mocnině 2:

```
10011011  = 1x2^7 + 0x2^6 + 0x2^5 + 1x2^4 + 1x2^3
          + 0x2^2 + 1x2^1 + 1x2^0
          = 128 + 0 + 0 + 16 + 8 + 0 +2 + 1
          =   155
```

Binární zápis pěkně odpovídá paměti počítače, ve které každá individuální jednotka, zvaná bit může být buď vypnuta nebo zapnuta. Stačí jen identifikovat vypnutí hodnotou 0 a zapnutí hodnotou 1. Paměť je běžně uspořádána do jednotek zvaných bajty, přičemž každý bajt je tvořen 8 bity. Bity v bajtu jsou číslovány podle mocniny 2. Bit nejvíce vpravo má tedy číslo 0, další bit má číslo 1 a tak dále. Obrázek A.1 přestavuje celé číslo o velikosti 2 bajtů.



Obrázek A.1 Celočíselná hodnota o velikosti dvou bajtů

Dvojková a šestnáctková soustava

Zápis v šestnáctkové soustavě se často používá pro komfortnější zobrazení binárních dat, například adres v paměti nebo čísel obsahujících bity s příznaky. Důvodem je, že každá šestnáctková číslice odpovídá čtyřbitové jednotce. Tabulka A.2 zobrazuje odpovídající hodnoty.

Tabulka A.2 Číslice v šestnáctkové soustavě a jejich binární ekvivalenty.

Šestnáctková číslice	Binární ekvivalent	Šestnáctková číslice	Binární ekvivalent
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Chcete-li převést šestnáctkovou hodnotu na binární, stačí nahradit každou šestnáctkovou číslici odpovídajícím binárním ekvivalentem. Napřílad šestnáctkové číslo 0xA4 odpovídá binárnímu 1010 0100. Podobně můžete snadno převést binární hodnoty do šestnáctkového zápisu převedením každé čtyřbitové jednotky na odpovídající šestnáctkovou číslici. Například z binární hodnoty 1001 0101 se stane 0x95.

DODATEK B

Klíčová slova jazyka C++

Klíčová slova jsou identifikátory, které tvoří slovní zásobu programovacího jazyka. Pro jiné účely, jako například název proměnné, je použít nelze. Klíčová slova jazyka jsou zobrazena v následujícím seznamu; ne všechna jsou v současné době implementována. Klíčová slova zobrazená tučně jsou také klíčovými slovy v ANSI C.

asm	**auto**	bool	**break**	**case**
catch	**char**	class	const	const_cast
continue	**default**	dejete	**do**	**double**
dynamic_cast	**else**	**enum**	explicit	extern
false	**float**	**for**	friend	goto
if	inline	**int**	**long**	mutable
namespace	new	operator	private	protected
public	**register**	reinterpret_cast	**return**	**short**
signed	**sizeof**	**static**	static_cast	**struct**
switch	template	this	throw	true
try	**typedef**	typeid	typename	**union**
unsigned	using	virtual	**void**	**volatile**
wchar_t	**while**			

DODATEK C

Znaková sada ASCII

Počítače ukládají znaky pomocí číselného kódu. Kód ASCII (American Standard Code for Information Interchange) je nejběžnějším kódem používaným ve Spojených státech. Jazyk C++ umožňuje vyjádřit jednodušší znaky přímo vložením znaku do apostrofu, například 'A' pro znak A. Jeden znak můžete také vyjádřit pomocí osmičkového nebo šestnáctkového kódu, před který umístíte zpětné lomítko; například '\012' a '\0xa' reprezentují znak linefeed (LF). Takové escape sekvence mohou být také součástí řetězce, například „Hello, \012my dear".

Znak ^ používaný v následující tabulce jako prefix označuje použití klávesy Ctrl.

Desítkově	Osmičkově	Šestnáctkově	Dvojkově	Znakově	Název v ASCII
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I,tabulátor	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4

Desítkově	Osmičkově	Šestnáctkově	Dvojkově	Znakově	Název v ASCII
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^[, esc	ESC
28	034	0x1c	00011100	^\	FS
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	mezera	SP
33	041	0x21	00100001	!	
34	042	0x22	00100010	“	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	
38	046	0x26	00100110	&	
39	047	0x27	00100111	‘	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	‘	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110	.	
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	

Desítkově	Osmičkově	Šestnáctkově	Dvojkově	Znakově	Název v ASCII
58	072	0x3a	00111010	:	
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0177	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	
92	0134	0x5c	01011100	\	
93	0135	0x5d	01011101	]	

Desítkově	Osmičkově	Šestnáctkově	Dvojkově	Znakově	Název v ASCII
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	_	
96	0140	0x50	01100000	‘	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	del. rubout	

DODATEK D

Priorita operátorů

Priorita operátorů určuje pořadí, ve kterém jsou operátory aplikovány na hodnoty. Operátory v C++ se dělí podle priority do 18 skupin uvedených v tabulce D.1, přičemž operátory ve skupině 1 mají prioritu nejvyšší. Jestliže jsou dva operátory použity se stejným operandem (něco, s čím operátor pracuje), použije se nejdříve operátor s vyšší prioritou. Mají-li oba operátory prioritu stejnou, určí jazyk C++ pomocí asociativních pravidel operátor, který je svázán těsněji. Všechny operátory ve stejné skupině mají stejnou prioritu a stejnou asociativitu, která je buď zleva doprava (v tabulce označena L-P) nebo zprava doleva (v tabulce označena P-L). Asociativita zleva doprava znamená, že se nejdříve použije levý operátor, zatímco při asociativitě zprava doleva se nejdříve použije operátor pravý.

Některé symboly, například * a &, se používají s více operátory. V takových případech je jedna forma *unární* (jeden operand) a druhá *binární* (dva operandy). Kompilátor určí význam podle kontextu. V tabulce jsou skupiny operátorů označeny jako unární nebo binární v těch případech, kdy se stejný symbol používá oběma způsoby.

Uvádíme několik příkladů priority a asociativity:

```
3 + 5 * 6
```

Operátor * má vyšší prioritu než operátor +, takže se nejdříve použije s číslem 5 a z výrazu se stane 3 + 30, čili 33.

```
120 / 6 * 5
```

Operátory / a * mají stejnou prioritu, ale sdružují se zleva doprava. To znamená, že operátor vlevo od sdíleného operandu (6) se použije první, takže z výrazu se stane 20 * 5, čili 100.

```
char / str = "Whoa";
char ch = *str++;
```

Oba unární operátory * a ++ mají stejnou prioritu, ale sdružují se zprava doleva. To znamená, že operátor inkrementování pracuje se `str` a ne se `*str`. Operace tedy inkrementuje ukazatel, posune ho na následující znak a nemění znak, na který ukazuje. Protože však je operátor ++ v postfixovém tvaru, inkrementuje se ukazatel až potom, co je původní hodnota `*str` přiřazena proměnné `ch`. Tento výraz tedy přiřadí znak `W` proměnné `ch` a potom nastaví ukazatel `str` na znak `h`.

Všimněte si, že v tabulce je ve sloupci Priorita uvedeno binární nebo unární, aby byly od sebe odlišeny dva operátory používající stejný symbol, například unární operátor adresy a binární operátor bitového součinu.

Priorita	Operátor	Asociace	Význam
1	::		Operátor rozlišení
	(výraz)		Seskupení
2	()	L - P	Volání funkce
	()		Vytvoření hodnoty, tj. typ (výraz)
	[]		Indexování pole
	->		Nepřímý výběr členu
	.		Přímý výběr členu
	const_cast		Specializované přetypování
	dynamic_cast		Specializované přetypování
	reinterpret_cast		Specializované přetypování
	static_cast		Specializované přetypování
	typeid		Typová identifikace
	++		Operátor postfixové inkrementace
	—		Operátor postfixové dekrementace
3 (všechny unární)	!	P - L	Logická negace
	~		Bitová negace
	+		Unární plus (znak kladného čísla)
	-		Unární mínus (znak záporného čísla)
	++		Operátor prefixové inkrementace
	—		Operátor prefixové dekrementace
	&		Adresa
	*		Dereferencování (nepřímá hodnota)
	()		Přetypování, tj. (typ) výraz
	sizeof		Velikost v bajtech
	new		Dynamické přidělení paměti
	new []		Dynamicky vytvořené pole
	delete		Uvolnění dynamicky přidělené paměti
	delete []		Zrušení dynamicky vytvořeného pole
4	.*	L - P	Dereference členu
	->*		Dereference nepřímého členu
5 (všechny binární)	*	L – P	Násobení
	/		Dělení
	^		Modulo (zbytek po dělení)
6 (všechny binární)	+	L - P	Sčítání

Priorita	Operátor	Asociace	Význam
	-		Odčítání
7	<<	L - P	Posun doleva
	>>		Posun doprava
8	<	L - P	Menší než
	<=		Menší nebo rovno
	>		Větší než
	>=		Větší nebo rovno
9	==	L - P	Rovno
	!=		Nerovno
10 (binární)	&	L - P	Bitový součin (AND)
11	^	L - P	Exkluzivní bitový součet (XOR)
12	|	L - P	Bitový součet (OR)
13	&&	L - P	Logický součin (AND)
14	||	L - P	Logický součet (OR)
15	=	P - L	Jednoduché přiřazení
	*=		Násobení a přiřazení
	/=		Dělení a přiřazení
	%=		Modulo a přiřazení
	+=		Sčítání a přiřazení
	-=		Odčítání a přiřazení
	&=		Bitový součin a přiřazení
	^=		Exkluzivní bitový součet a přiřazení
	|=		Bitový součet a přiřazení
	<<=		Posun doleva a přiřazení
	>>=		Posun doprava a přiřazení
16	:?	P - L	Podmíněný výraz
17	throw	L - P	Vyvolání výjimky
18	,	L - P	Slučování dvou výrazů do jednoho

DODATEK E

Ostatní operátory

Aby nedošlo k bezmeznému nárůstu této knihy, neobsahuje hlavní text dvě skupiny operátorů. První skupina se skládá z bitových operátorů, která umožňuje manipulovat s jednotlivými bity v hodnotě; tyto operátory byly zděděny od jazyka C. Druhou skupinu tvoří dvoučlenné operátory dereferencování; ty byly do C++ přidány. Tyto operátory jsou stručně shrnuty v příloze.

Bitové operátory

Bitové operátory pracují s bity celočíselných hodnot. Například operátor posunu doleva posouvá bity doleva, zatímco operátor bitové negace nastaví každý bit s hodnotou jedna na hodnotu nula a každý bit s hodnotou nula na hodnotu jedna. C++ má celkem šest takových operátorů: `<<`, `>>`, `~`, `|` a `^`.

Operátory posunu

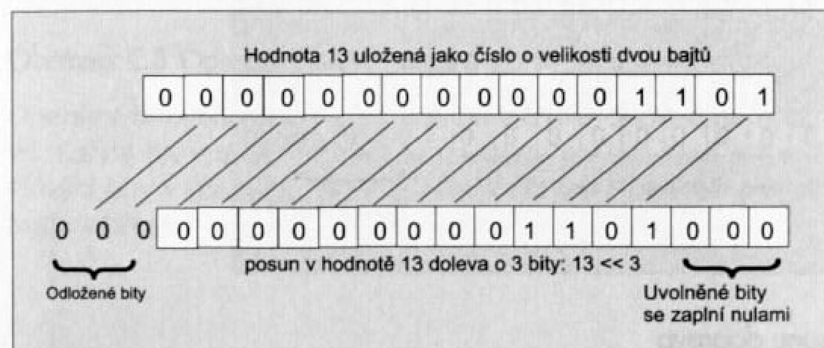
Operátor posunu doleva má následující syntaxi:

```
hodnota << posun
```

Hodnota představuje celočíselnou hodnotu, která se má posunout, a *posun* označuje počet bitů, o které se posune. Například výraz

```
13 << 3
```

znamená posun všech bitů v hodnotě 13 o tři místa doleva. Uvolněná místa se zaplní nulami a bity posunuté za konec budou odloženy (viz obrázek E.1).



Obrázek E.1 Operátor posunu doleva

Vzhledem k tomu, že každá bitová pozice reprezentuje hodnotu, která je dvojnásobkem bitu vpravo (viz příloha A), rovná se posun bitu o jednu pozici násobení hodnoty dvěma. Podobně posun bitu o dvě pozice se rovná násobení 2^2 a posun o n pozic se rovná násobení 2^n.

Operátor posunu vlevo má schopnost, která se často vyskytuje v jazycích assembleru. Avšak assemblerovský operátor posunu doleva mění přímo obsah registru, zatímco operátor C++ vytvoří novou hodnotu bez změny hodnot existujících. Uvažujte například následující kód:

```
int x = 20;
int y = x << 3;
```

Tento kód hodnotu proměnné x nezmění. Výraz `x << 3` vytvoří pomocí proměnné x hodnotu novou stejně tak jako výraz `x + 3`, který hodnotu x také nezmění.

Jestliže chcete změnit hodnotu proměnné pomocí operátoru posunu doleva, musíte použít také přiřazení. Můžete použít běžný operátor přiřazení nebo operátor <<=, který kombinuje posun s přiřazením.

```
x = x << 4;        // běžné přiřazení
y <<= 2;  // posun a přiřazení
```

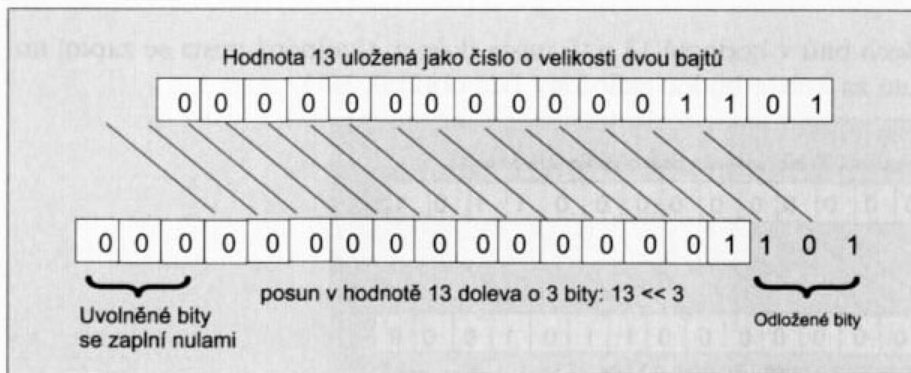
Operátor posunu doprava (>>) posouvá bity podle očekávání doprava. Jeho syntaxe je následující:

```
hodnota >> posun
```

Hodnota představuje celočíselnou hodnotu, která se má posunout, a *posun* označuje počet bitů, o které se posune. Například výraz

```
17 >> 2
```

znamená posun všech bitů v hodnotě 17 o dvě místa doprava. U celých čísel bez znaménka se uvolněná místa zaplní nulami a bity posunuté za konec budou odloženy. U celých čísel se znaménkem se uvolněná místa mohou zaplnit nulami nebo hodnotou původního bitu nejvíce vlevo. Volba závisí na implementaci (obrázek E.2 ukazuje příklad s vyplněnými nulami).



Obrázek E.2 Operátor posunu doprava

Posun o jedno místo doprava je ekvivalentní dělení celého čísla hodnotou 2. Obecně platí, že posun o n míst doprava je ekvivalentní dělení celého čísla hodnotou 2^n.

C++ také definuje operátor posunu doprava a přiřazení pro případ, že byste chtěli hodnotu proměnné nahradit hodnotou vzniklou posunem:

```
int q = 43;
q >>= 2;  // nahradí 43 výrazem 43 >> 2, čili 10
```

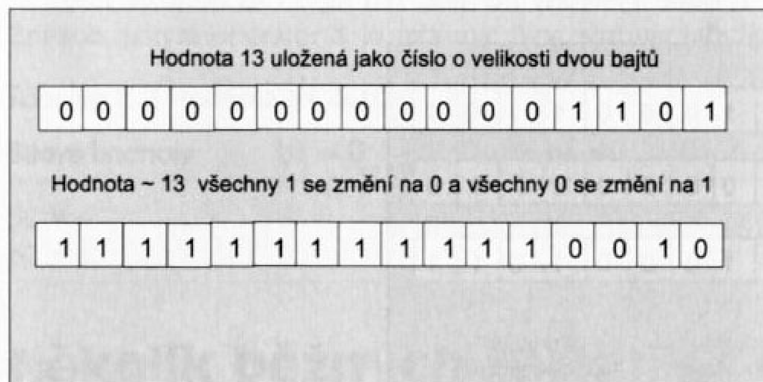
V některých systémech mohou operátory posunu doleva a doprava nahrazovat násobení celých čísel a jejich dělení dvěma. Protože se však kompilátory v optimalizaci kódu zlepšují, jsou takové rozdíly zanedbatelné.

Bitové logické operátory

Bitové logické operátory jsou analogické obyčejným logickým operátorům, ale v hodnotě se používají spíše na jednotlivé bity než na celek. Uvažujte například obyčejný operátor negace (`!`) a bitový operátor negace (~). Operátor `!` změní hodnotu `true` (nebo nenulovou) na `false` a `false` na `true`. Operátor ~ změní každý jednotlivý bit na jeho protějšek (jedničku na nulu a nulu na jedničku). Uvažujte například hodnotu 3 typu `unsigned char`:

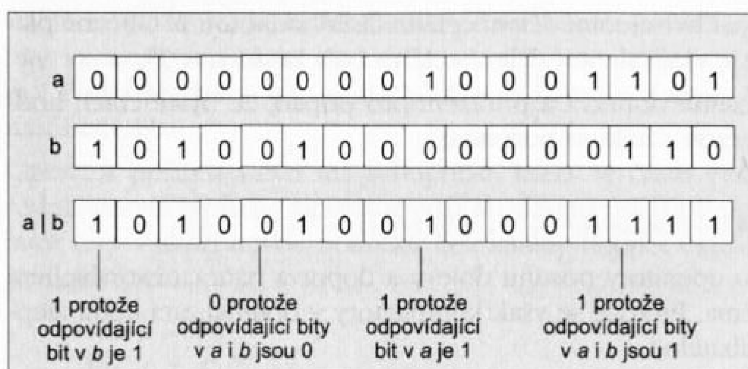
```
unsigned char x = 3;
```

Výraz !x má hodnotu 0. Jestliže chcete zjistit hodnotu výrazu ~x, napište ji v binárním tvaru: 00000011. Potom změňte všechny jedničky na nuly a nuly na jedničky. Získáte hodnotu 11111100, čili v desítkové soustavě hodnotu 252 (viz obrázek E.3 s 16bitovým příkladem).



Obrázek E.3 Operátor bitové negace

Operátor bitového součtu (|) kombinuje dvě celočíselné hodnoty do jedné hodnoty nové. Každý bit v nové hodnotě je nastaven na 1, pokud jeden či druhý nebo i oba odpovídající bity v původní hodnotě jsou nastaveny 1. Mají-li oba odpovídající bity hodnotu 0, bude výsledný bit nastaven na 0 (viz obrázek E.4).



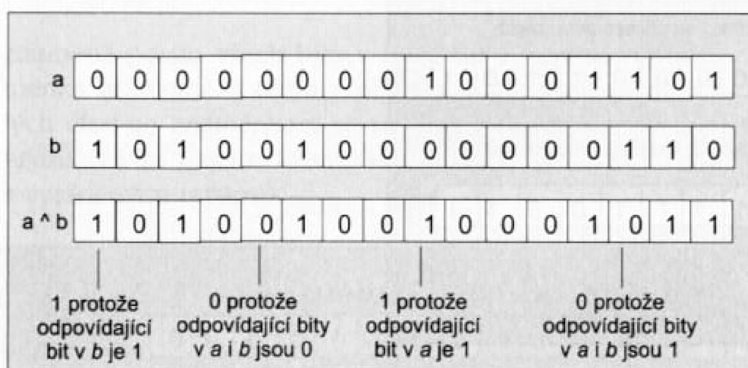
Obrázek E.4 Operátor bitového součtu (OR)

Způsob, jakým operátor | kombinuje bity, shrnuje tabulka E.1.

Tabulka E.1 Hodnota b1 | b2.

Bitové hodnoty	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	1

Operátor exkluzivního bitového součtu (^) kombinuje dvě celočíselné hodnoty a vytváří jednu hodnotu novou. Každý bit v nové hodnotě je nastaven na 1, pokud jeden či druhý, nikoli však oba odpovídající bity v původní hodnotě jsou nastaveny 1. Mají-li oba odpovídající bity hodnotu 0 nebo 1, bude výsledný bit nastaven na 0 (viz obrázek E.5).



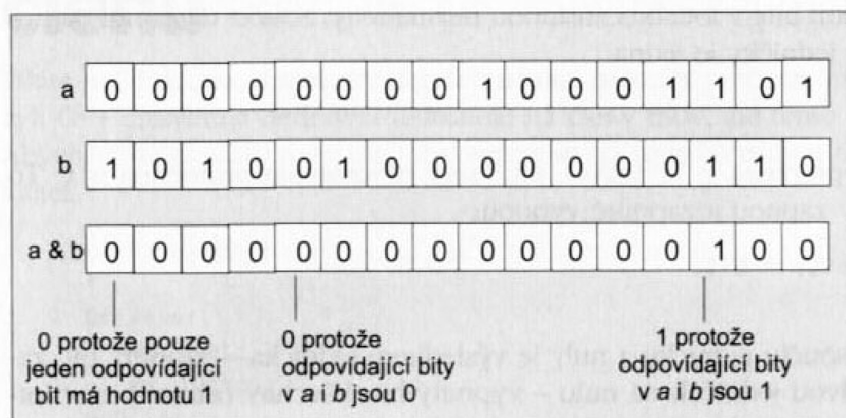
Obrázek E.5 Operátor exkluzivního bitového součtu (XOR)

Způsob, jakým operátor ^ kombinuje bity, shrnuje tabulka E.2.

Tabulka E.2 Hodnota b1 ^ b2.

Bitové hodnoty	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	0

Operátor bitového součinu (&) kombinuje dvě celočíselné hodnoty a vytváří jednu hodnotu novou. Každý bit v nové hodnotě je nastaven na 1 pouze tehdy, pokud oba odpovídající bity v původní hodnotě jsou nastaveny 1. Má-li některý z odpovídajících bitů nebo oba hodnotu 0, bude výsledný bit nastaven na 0 (viz obrázek E.6).



Obrázek E.6 Operátor bitového součinu (AND)

Způsob, jakým operátor & kombinuje bity, shrnuje tabulka E.3.

Tabulka E.3 Hodnota b1 & b2.

Bitové hodnoty	b1 = 0	b1 = 1
b2 = 0	0	0
b2 = 1	0	1

Několik běžných technik s bitovými operátory

Pro ovládání hardwaru je často potřeba určité bity zapnout nebo vypnout nebo zkontrolovat jejich stav. Bitové operátory představují prostředky pro takovou činnost. Uvedené metody si v rychlosti projdeme.

V následujících příkladech představuje `lottabits` obecnou hodnotu a `bit` reprezentuje hodnotu odpovídající určitému bitu. Bity jsou číslovány zprava doleva počínaje nulou, takže hodnota odpovídající bitu na pozici `n` je `2n`. Například celé číslo, které má nastaven pouze bit č. 3, má hodnotu 2^3 čili 8. Obecně platí, že každý jednotlivý bit odpovídá moc-

nině 2, tak jak bylo popsáno u binárních čísel v příloze A. Termín bit tedy bude vyjadřovat mocninu 2; to odpovídá určitému bitu nastavenému na hodnotu 1 při všech ostatních bitech nastavených na hodnotu 0.

Zapnutí bitu

Obě následující operace zapínají v `lottabits` bit, který představuje proměnná `bit`:

```
lottabits = lottabits | bit;
lottabits |= bit;
```

Obě operace nastavují odpovídající bit na hodnotu 1 bez ohledu na jeho dřívější hodnotu. Při bitovém součtu s první hodnotou 1 je výsledná hodnota 1, ať už je druhá hodnota 0 nebo 1. Všechny ostatní bity v lottabits zůstanou nezměněny. Součet dvou nul dá nulu, zatímco součet nuly a jedničky je jedna.

Přepnutí bitu

Obě následující operace přepínají v `lottabits` bit, který představuje proměnná `bit`. To znamená, že vypnuté bity zapnou a zapnuté vypnou:

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

Při exkluzivním bitovém součtu jedničky a nuly je výsledkem jednička – zapnutý bit, zatímco exkluzivní součet dvou jedniček dá nulu – vypnutý bit. Všechny ostatní bity v lottabits zůstanou nezměněny. Exkluzivní součet dvou nul dá nulu, zatímco součet nuly a jedničky je jedna.

Vypnutí bitu

Následující operace vypne v `lottabits` bit, který představuje proměnná `bit`:

```
lottabits = lotabits & ~bit;
```

Tento příkaz vypne bit bez ohledu na jeho dřívější stav. Nejdříve operátor ~bit vytvoří celé číslo, ve kterém budou na hodnotu 1 nastaveny všechny bity kromě bitu, který již hodnotu 1 má; tento bit získá hodnotu 0. Součin nuly s jakýmkoli bitem vede k výsledku 0 – vypnutý bit. Všechny ostatní bity v `lottabits` zůstanou nezměněny. Součin hodnoty 1 s kterýmkoli bitem dá hodnotu, kterou již uvedený bit měl.

Zde je stručnější zápis stejné operace:

```
lotabits &= ~bit;
```

Test hodnoty bitu

Předpokládejme, že chcete určit, zda bit odpovídající proměnné `bit` je v `lottabits` nastaven na hodnotu 1.

Následující test fungovat nemusí:

```
if (lottabits == bits)    // není dobré
```

I když bude odpovídající bit v `lottabits` nastaven na hodnotu 1, mohou být na tuto hodnotu nastaveny i jiné bity. Výše uvedený výraz bude pravdivý pouze v tom případě, že

pouze odpovídající bit bude mít hodnotu 1. Nápravu provedete přidáním bitového AND mezi `lottabits` a `bit`. Takový součin dá hodnotu 0 ve všech ostatních bitových pozicích, protože 0 násobená jakoukoli hodnotou je zase 0. Pouze `bit` odpovídající hodnotě 1 zůstane nezměněn, protože součin dvou jedniček dá stejnou hodnotu. Správně tedy test vypadá takto:

```
if (lottabits & bit == bits)        // test bitu
```

Operátory dereferencování členů

Dříve než budeme operátory dereferencující položky probírat, musíme nastínit situaci. Jazyk C++ umožňuje definovat ukazatele na členy třídy, ale tento proces není jednoduchý. Abychom si ukázali, o co jde, podívejme se na ukázku třídy a na problémy přitom vznikající:

```
class example
{
private:
   int feet;
   int inches;
public:
   example();
   example(int ft);
   ~example();
   void show_in(); // zobrazí položku inches
   example operator+(example &ex);
};
```

Nyní předpokládejme, že chcete definovat ukazatel na položku této třídy `inches`. Následující pokus fungovat nebude:

```
int * pi = &inches;        // v C++ nelze
```

Příkaz je neplatný, protože proměnná `inches` není typu `int`. Protože je deklarovaná ve třídě, má třídní oblast platnosti. Typ proměnné `inches` proto musí také specifikovat třídu, do které položka patří. Aby byla deklarace platná, musíte identifikovat třídu ukazatele a položky pomocí operátoru rozlišení:

```
int example::* pi = &example::inches;      // platný příkaz
```

V této deklaraci výraz `int example::*` označuje typ „ukazatel na položku typu `int` ze třídy `example`". Výraz `&example::inches` znamená „adresa položky `inches` ze třídy `example`".

Tuto formu deklarace můžete použít v členských nebo spřátelených funkcích. Ukazatel `pi` se chová jako položka třídy v tom smyslu, že musí být vyvolán objektem třídy. Tady přicházejí ke slovu operátory dereferencování. Předpokládejme například, že proměnná `ex` je objekt třídy `example` deklarovaný v členské funkci. Abyste získali přístup k položce inches tohoto objektu, můžete použít standardní zápis `ex.inches`. Můžete však také použít operátor `.*` s ukazatelem `pi`:

```
cout << ex.inches;// zobrazí položku inches
cout << ex.*pi;   // totéž
```

Zatímco operátor . položku zpřístupní pomocí jejího názvu, operátor .* položku zpřístupní pomocí ukazatele na ni.

Předpokládejme podobně, že `px` je ukazatel na objekt třídy `example`. V takovém případě můžete přístup k položce `inches` získat buď pomocí jejího názvu, nebo pomocí operátoru dereferencování ->* přes ukazatel na ni:

```
px = &ex;              // px je ukazatel na objekt třídy example
cout << px->inches;    // zobrazí položku inches
cout << px->*pi;       // totéž
```

Všimněte si, že zatímco proměnná px je ukazatel na celý objekt, proměnná `pi` je ukazatel na položku třídy.

Abychom si ukázali, jak tyto nové operátory fungují v praxi, použijeme je trochu oklikou při implementaci funkce `operator+()`. Funkce bude sčítat dva objekty. Jeden z nich bude jejím parametrem. Protože se jedná o objekt a ne ukazatel, můžete získat přístup k položce `inches` pomocí operátoru .*. Druhým objektem je objekt vyvolávající a ten je, vzpomeňte si, reprezentován ukazatelem `this`. Můžete s ním tedy použít operátor -> tak, jak je ukázáno v následující části kódu:

```
example example::operator+(example &ex)
{
  example sum;

  int example::*pi = &example::inches;
  // ukazuje na položku inches ze třídy examples
  sum.inches = ex.*pi + this->*pi;
  sum.feet = 12 * sum.inches;
  return sum;
}
```

Výraz `ex.*pi` zde reprezentuje položku inches z objektu `ex` a výraz `this->*pi` položku inches z toho objektu, na který `this` ukazuje. Všimněte si, že `*pi` je použito jako název položky.

Ve výpisu E.1 je program se zbývajícími definicemi metod a funkcí `main()`, která třídu používá.

Výpis E.1 memb_pt.cpp

```
// memb_pt.cpp - dereferencující ukazatele na položky třídy
#include <iostream>
using namespace std;
class example
{
private:
  int feet;
  int inches;
public:
```

```
    example();
    example(int ft);
    ~ example();
    void show_in();
    example operator+(example &ex);
};
example::example()
{
    feet = 0;
    inches = 0;
}
example::example(int ft)
{
    feet = ft;
    inches = 12 * feet;
}
example::~example()
{
}
void example::show_in()
{
    cout << inches << " inches\n";
}
example example::operator+(example &ex)
{
    example sum;

    int example::*pi = &example::inches;
    // ukazuje na položku inches ze třídy examples
    sum.inches = ex.*pi + this->*pi;
    sum.feet = 12 * sum.inches;
    return sum;
}
int main()
{
    example car(15);
    example van(20);
    example garage;
    garage = car + van;
    car.show_in();
    van.show_in();
    garage.show_in();
    return 0;
}
Zde je ukázka běhu:
180 inches
240 inches
420 inches
```

DODATEK F

Šablonová třída STRING

Třída string je založena na definici šablony:

```
template<class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
   class basic_string {...};
```

Zde charT reprezentuje typ uložený v řetězci. Parametr traits reprezentuje třídu, která definuje nezbytné vlastnosti, kterými musí typ disponovat, aby byl reprezentován jako řetězec. Měl by mít například metodu `length()` vracející délku řetězce reprezentovanou jako pole typu `charT`. Konec takového pole označuje hodnota `charT(0)`, generalizovaný nulový znak. (Výraz `charT(0)` označuje nulu přetypovanou na typ `charT`. Stačilo by použít 0 stejně jako u typu `char` nebo, obecněji, objekt vytvořený konstruktorem `charT`.) Třída také obsahuje metody porovnávající hodnoty a podobně. Parametr `Allocator` reprezentuje třídu, která se stará o přidělování paměti pro řetězec. Implicitní šablona `allocator<charT>` používá standardním způsobem operátory `new` a `delete`.

Existují dvě předdefinované specializace:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Tyto specializace zase používají následující specializace:

```
char_traits<char>
allocator<char>
char_traits<wchar_t>
allocator< wchar_t>
```

Třídu `string` můžete použít i pro některé jiné typy než pouze char nebo `wchar`, pokud definujete třídu `traits` a použijete šablonu `basic_string`.

Třináct typů a jedna konstanta

Šablona basic_string definuje několik typů, které se později použijí při definování metod:

```
typedef traits                                   traits_type;
typedef typename traits::char_type               value_type;
typedef Allocator                                allocator_type;
typedef typename Allocator::size_type            size_type;
typedef typename Allocator::difference_type      difference_type;
typedef typename Allocator::reference            reference;
typedef typename Allocator::const_reference      const_reference;
typedef typename Allocator::pointer              pointer;
typedef typename Allocator::const_pointer        const_pointer;
```

Všimněte si, že šablona traits je parametrem, který bude odpovídat nějakému konkrétnímu typu, například char_traits<char>; traits_type bude definován pomocí příkazu typedef na tento konkrétní typ. Zápis

```
typedef typename traits::char_typevalue_type;
```

znamená, že char_type je název definovaný ve třídě reprezentované traits. Použité klíčové slovo typename kompilátoru oznamuje, že výraz traits::char_type představuje typ. Například pro řetězcovou specializaci bude value_char typu char.

Typ size_type se používá stejně jako size_of, ale vrací velikost řetězce podle uloženého typu. U řetězcové specializace by to byl char a v takovém případě by size_type odpovídal size_of. Jedná se o typ bez znaménka.

Typ difference_type se používá pro měření vzdálenosti mezi dvěma prvky řetězce a také v jednotkách odpovídajících velikosti jednoho prvku. Běžně se bude jednat o verzi základního typu size_type se znaménkem.

U specializace char bude ukazatel typu char * a reference typu char &. Jestliže však vytvoříte specializaci typu podle vlastního návrhu, budou tyto typy (ukazatel a reference) moci odkazovat na třídy, které mají stejné vlastnosti jako většina ukazatelů a referencí třídy základní.

Aby bylo možné používat s řetězci algoritmy knihovny STL, definuje šablona několik typů iterátorů:

```
typedef (models random access iterator)          iterator;
typedef (models random access iterator)          const_iterator;
typedef std::reverse_iterator<iterator>          reverse_iterator;
typedef std::reverse_iterator<const_iterator>    const_reverse_iterator;
```

Šablona definuje také jednu statickou konstantu:

```
static const size_type npos = -1;
```

Vzhledem k tomu, že typ size_type je bez znaménka, rovná se přiřazení hodnoty –1 do proměnné npos vlastně přiřazení největší hodnoty bez znaménka. Tato hodnota odpovídá 1 větší než největší možný index pole.

Datové informace, konstruktory a další

Konstruktory lze popsat podle jejich účinků. Vzhledem k tomu, že soukromé části třídy mohou být implementačně nezávislé, měly by být tyto účinky popsány podle informací dostupných jako součást veřejného rozhraní. V tabulce F.1 je seznam několika metod a pomocí jejich návratových hodnot lze popsat účinky konstruktorů i jiných metod. Všimněte si, že většina terminologie pochází z knihovny STL.

Tabulka F. Několik metod pro práci s řetězci.

Metoda	Návratová hodnota
begin()	Iterátor na první znak v řetězci (existuje také konstantní verze vracející konstantní iterátor).
end()	Iterátor s hodnotou prvku za koncem (existuje také konstantní verze).
rbegin()	Inverzní iterátor s hodnotou prvku za koncem (existuje také konstantní verze).
rend()	Inverzní iterátor na první znak (existuje také konstantní verze).
size()	Počet prvků v řetězci, odpovídá vzdálenosti od `begin()` do `end()`.
length()	Stejná jako u `size()`.
capacity()	Alokovaný počet prvků v řetězci.
max_size()	Maximální možná velikost řetězce.
data()	Ukazatel typu `const charT*` na první prvek pole, jehož počet prvků je stejný jako počet odpovídajících prvků v řetězci, ovládaném ukazatelem `*this`. Platnost ukazatele skončí, jakmile se objekt třídy `string` změní.
c_str()	Ukazatel typu `const charT*` na první prvek pole, jehož počet prvků je stejný jako počet odpovídajících prvků v řetězci ovládaném ukazatelem `*this`, a jehož následující prvek pro typ `charT` je znak `charT(0)` (znak konce řetězce). Platnost ukazatele skončí, jakmile se objekt třídy `string` změní.
get_allocator	Kopie objektu alokátoru použitá pro přidělení paměti pro objekt třídy `string`.

Dávejte pozor na rozdíly mezi metodami `begin()`, `rend()`, `data()` a `c_str()`. Všechny se vztahují na první znak v řetězci, ale jiným způsobem. Metody `begin()` a `rend()` vracejí iterátory, které jsou generalizovanými ukazateli, které byly popsány v kapitole 15 při probírání knihovny STL. Metoda `begin()` vrací model dopředného iterátoru, zatímco metoda `rend()` kopii iterátoru reverzního. Obě ukazují na skutečný řetězec spravovaný objektem třídy `string`. (Vzhledem k tomu, že třída `string` používá dynamické přidělování paměti, nemusí být skutečný obsah řetězce v objektu, a proto vztah mezi objektem a řetězcem popisujeme termínem *spravovat.*) Metody vracející iterátory můžete použít u algoritmů knihovny STL, které jsou na iterátorech založeny. Pro konverzi prvků řetězce na velká písmena můžete například použít funkci knihovny STL `transform()`:

```
string word;
cin >> word;
transform(word.begin(), word.end(), toupper);
```

Naproti tomu metody data() a c_str() vracejí obyčejné ukazatele. Vrácené ukazatele navíc ukazují na první prvek pole obsahujícího znaky řetězce. Toto pole může, ale nemusí být kopií původního řetězce spravovaného objektem třídy string. (Vnitřní reprezentace použitá objektem třídy string může být pole, ale také nemusí.) Vzhledem k tomu, že vracené ukazatele mohou ukazovat na původní data, jsou konstantní a nelze je tedy použít ke změně dat. Také není zaručena platnost ukazatelů, jestliže byl řetězec modifikován; to odráží skutečnost, že mohou ukazovat na původní data. Rozdíl mezi metodami data() a c_str() spočívá v tom, že pole, na které ukazuje c_str(), je ukončeno nulovým znakem (nebo ekvivalentem), zatímco metoda data() pouze garantuje přítomnost znaků skutečného řetězce. Metodu c_str() lze tedy použít například jako parametr funkce očekávající řetězec ve stylu jazyka C:

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

Podobně lze metody data() a size() použít u funkce očekávající ukazatel na prvek pole a hodnotu vyjadřující počet prvků, které se mají zpracovat:

```
string vampire("Do not stake me, oh my darling!");
int vlad = byte_check(vampire.data(), vampire.size());
```

Implementace si může vybrat, zda bude řetězec objektu třídy string reprezentovat jako dynamicky alokovaný řetězec ve stylu jazyka C a dopředný iterátor implementovat jako ukazatel char *. V takovém případě by implementace mohla vybírat mezi metodami begin(), data() a c_str(), které všechny vracejí stejný ukazatel. Stejně tak legitimně (i když ne tak snadno) by mohla vracet reference na tři různé datové objekty.

V tabulce F.2 je seznam šesti konstruktorů a jednoho destruktoru šablonové třídy basic_string. Všimněte si, že všech šest konstruktorů má parametr v následujícím tvaru:

```
const Allocator& a = Allocator()
```

Vzpomeňte si, že termín Allocator je název šablonového parametru třídy alokátoru pro správu paměti. Termín Allocator() označuje implicitní konstruktor této třídy. Konstruktory jsou tedy standardně implicitními verzemi objektu alokátoru, ale umožňují vám zvolit i některé jiné verze objektu alokátoru. Prozkoumejme tyto konstruktory jednotlivě.

Tabulka F.2 Konstruktory třídy string.

Prototypy konstruktorů a destruktorů

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(const basic_string& str, size_type pos = 0,
   size_type n = npos, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n,
   const Allocator& a = Allocator());
                  basic_string(size_type n, charT c, const Allocator&
a = Allocator());
         template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
```

```
const Allocator& a = Allocator());
        ~basic_string();
```

Implicitní konstruktor

Implicitní konstruktor vypadá takto:

```
explicit basic_string(const Allocator& a = Allocator());
```

Použitím implicitního konstruktoru akceptujete implicitní parametr a vytváříte prázdné řetězce:

```
string bean;
wstring theory;
```

Po zavolání konstruktoru budou platit následující vztahy:

- Metoda `data()` vrátí nenulový ukazatel, ke kterému lze přidat 0.
- Metoda `size()` vrátí hodnotu 0.
- Návratová hodnota metody `capacity()` není specifikována.

Předpokládejme, že návratovou hodnotu metody `data()` přiřadíte ukazateli `str`. V tom případě bude podmínečný výraz `str + 0` platit, což znamená, že platí také výraz `str[0]`.

Konstruktor používající pole

Následující konstruktor vám umožňuje inicializovat objekt třídy `string` pomocí řetězce ve stylu jazyka C; obecněji řečeno, můžete inicializovat specializaci `charT` pomocí pole hodnot typu `charT`:

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

Pro určení počtu znaků, které má zkopírovat, použije konstruktor metodu `traits::length()` na pole odkazované ukazatelem `s`. (Ukazatel s nesmí být nulový). Například příkaz

```
string toast("Podivej se na tu dobrotu, klucino.");
```

inicializuje objekt toast pomocí uvedeného řetězce znaků. Metoda traits::length() pro typ char určí pomocí nulového znaku počet znaků, které má zkopírovat.

Po zavolání konstruktoru budou platit následující vztahy:

- Metoda `data()` vrátí ukazatel na první prvek kopie pole `s`.
- Metoda `size()` vrátí hodnotu rovnající se hodnotě vrácené metodou `traits::length()`.
- Metoda `capacity()` vrátí hodnotu alespoň takové velikosti, jako metoda `size()`.

Konstruktor používající část pole

Následující konstruktor vám umožňuje inicializovat objekt třídy string pomocí řetězce ve stylu jazyka C; obecněji řečeno, můžete inicializovat specializaci charT pomocí části pole hodnot typu charT:

```
basic_string(const charT* s, size_type n, const Allocator& a=Allocator());
```

Tento konstruktor zkopíruje do vytvořeného objektu celkem n znaků z pole, na které ukazuje s. Všimněte si, že kopírování neskončí, má-li s méně znaků než n. Jestliže je n delší než s, bude konstruktor interpretovat obsah paměti následujícího řetězce, jako kdyby obsažená data byla typu charT.

Konstruktor požaduje, aby ukazatel s nebyl nulový a aby platil výraz n < npos. (Vzpomeňte si, že npos je statická konstanta třídy rovnající se maximálnímu možnému počtu prvků v řetězci.) Jestliže je n rovno npos, vyvolá konstruktor výjimku out_of_range. (Vzhledem k tomu, že n je typu size_type a npos je maximální hodnota typu size_type, nemůže být n větší než npos.) V ostatních případech budou po zavolání konstruktoru platit následující vztahy:

- Metoda data() vrátí ukazatel na první prvek kopie pole s.
- Metoda size() vrátí hodnotu n.
- Metoda capacity() vrátí hodnotu alespoň takové velikosti, jako metoda size().

Kopírovací konstruktor

Kopírovací konstruktor má několik parametrů s implicitními hodnotami:

```
basic_string(const basic_string& str, size_type pos = 0, size_type
n = npos, const Allocator& a = Allocator());
```

Zavoláte-li ho pouze s parametrem basic_string, bude konstruktor inicializovat nový objekt pomocí parametru objektu string:

```
string mel("Jsem v poradku!");
string ida(mel);
```

Zde by objekt ida dostal kopii řetězce spravovanou objektem mel.

Volitelný druhý parametr pos specifikuje pozici ve zdrojovém řetězci, od které se začne kopírovat:

```
string att("Zavolej domu.");
string et(att, 2);
```

Číslování pozic začíná číslem 0, takže na pozici 2 je znak v. Objekt et je tedy inicializován řetězcem „volej".

Volitelný třetí parametr n specifikuje maximální počet znaků, které se zkopírují. Kód

```
string att("Zavolej domu.");
string pt(att, 2, 5);
```

tedy inicializuje objekt pt řetězcem „volej". Tento konstruktor se však zastaví na konci zdrojového řetězce; například výraz

```
string pt(att, 2, 200);
```

ukončí kopírování po zkopírování znaku tečka. Konstruktor tedy zkopíruje počet znaků, který je roven nebo menší než je hodnota n a str.size() - pos.

Tento konstruktor požaduje, aby platil výraz pos <= str.size(), to znamená, aby se počáteční pozice nacházela ve zdrojovém řetězci; pokud tomu tak není, vyvolá výjimku out_of_range. V ostatních případech, pokud počet prvků copy_len představuje hodnotu menší než je hodnota n a str.size() - pos, budou po zavolání konstruktoru platit následující vztahy:

- Metoda data() vrátí ukazatel na prvky copy_len zkopírované z řetězce str od pozice pos.
- Metoda size() vrátí počet prvků copy_len.
- Metoda capacity() vrátí hodnotu alespoň takové velikosti, jako metoda size().

Konstruktor používající n kopií znaku

Následující konstruktor vytvoří objekt třídy string obsahující sekvenci n znaků s hodnotou c:

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

Konstruktor požaduje, aby platil výraz n < pos. Jestliže se bude n rovnat npos, vyvolá výjimku out_of_range. V ostatních případech budou po zavolání konstruktoru platit následující vztahy:

- Metoda data() vrátí ukazatel na první prvek řetězce s n prvky, přičemž každý prvek bude nastaven na hodnotu c.
- Metoda size() vrátí hodnotu n.
- Metoda capacity() vrátí hodnotu alespoň takové velikosti jako metoda size().

Konstruktor pracující s rozsahem

Poslední konstruktor používá iterátorem definovaný rozsah ve stylu knihovny STL:

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
                                const Allocator& a = Allocator());
```

Iterátor begin ukazuje ve zdroji na prvek, kde kopírování začne, zatímco iterátor end na místo za posledním zkopírovaným prvkem.

Tento tvar můžete používat u polí, řetězců nebo kontejnerů knihovny STL:

```
char cole[40] = "Old King Cole was a merry old soul."
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
```

```
    input.push_back(ch);
string str_input(input.begin(), input.end());
```

Při prvním použití je iterátor `InputIterator` vyhodnocen na typ `const char *`. Při druhém použití je vyhodnocen na typ `vector<char>::iterator`.

Po zavolání konstruktoru budou platit následující vztahy:

- Metoda `data()` vrátí ukazatel na první prvek řetězce vytvořeného kopírováním prvků z rozsahu `[begin, end)`.
- Metoda `size()` vrátí hodnotu označující vzdálenost mezi `begin` a `end`. (Vzdálenost je měřena v jednotkách, rovnajících se velikosti datového typu, získaného při dereferencování ukazatele.)
- Metoda `capacity()` vrátí hodnotu alespoň takové velikosti, jako metoda `size()`.

Různé metody pro práci s pamětí

Existuje několik metod pracujících s pamětí, které například čistí obsah paměti, mění velikost řetězce nebo upravují jeho kapacitu. Některé z těchto metod jsou uvedeny v tabulce F.3.

Tabulka F.3 Několik metod pro práci s pamětí.

Metoda	Účinek
void resize(size_type n)	Vyvolá výjimku `out_of_range`, jestliže platí výraz `n > npos`. V opačném případě změní velikost řetězce na hodnotu `n`, zkrátí řetězec, jestliže platí výraz `n < size()` a vyplní řetězec znaky `charT(0)`, jestliže platí výraz `n > size()`.
void resize(size_type n, charT c)	Vyvolá výjimku `out_of_range`, jestliže platí výraz `n > npos`. V opačném případě změní velikost řetězce na hodnotu `n`, zkrátí řetězec, jestliže platí výraz `n < size()` a vyplní řetězec znaky s hodnotou `c`, jestliže platí výraz `n > size()`.
void reserve(size_type res_arg = 0)	Nastaví hodnotu vrácenou metodou `capacity()` na hodnotu větší nebo rovnou parametru `res_arg`. Protože přitom dochází k realokaci řetězce, ruší se platnost předchozích referencí, iterátorů a ukazatelů do řetězce.
void clear()	Odstraní ze řetězce všechny znaky.
bool empty() const	Vrátí hodnotu `true`, jestliže platí výraz `size() == 0`.

Přístup k řetězci

Pro přístup k jednotlivým znakům existují čtyři metody; dvě používají operátor `[]` a dvě metodu `at()`:

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type pos);
const_reference at(size_type pos) const;
```

První metoda operator[]() umožňuje přístup k jednotlivým prvkům řetězce pomocí zápisu používaného pro pole; lze ji použít pro získání hodnoty nebo při její změně. Druhou metodu operator[]() lze použít u konstantních objektů, a to pouze pro získání hodnoty:

```
string word("tack");
cout << word[0];              // zobrazí t
word[3] = 't';                // přepíše znak k znakem t
const ward("garlic");
cout << ward[2];              // zobrazí r
```

Metody at() nabízejí podobný přístup, ale index je dodán jako parametr funkce:

```
string word("tack");
cout << word.at(0);           // zobrazí t
```

Rozdíl (kromě syntaxe) spočívá v tom, že metody at() provádějí kontrolu mezí a vyvolávají výjimku out_of_range, jestliže platí výraz pos >= size(). Všimněte si, že pos je typu size_type, tedy bez znaménka, a proto nemůže mít zápornou hodnotu. Metody operator[]() kontrolu mezí neprovádějí, takže chování při platnosti výrazu je pos >= size() je nedefinované. Konstantní verze pouze vrátí ekvivalent nulového znaku, jestliže platí výraz pos == size().

Můžete si tedy vybrat mezi bezpečností (použitím metody at() a testem na výjimky) a rychlostí provedení (při použití notace pro pole).

Existuje také funkce, která vrací nový řetězec, jenž je podřetězcem původního:

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

Tato funkce vrací řetězec, který je kopií řetězce s počátkem na pozici pos a jehož délka je buď n znaků nebo počet znaků zbývajících do konce původního řetězce. Například následující kód inicializuje řetězec pet řetězcem „donkey“:

```
string message("Maybe the donkey wil learn to sing.");
string pet(message.substr(10, 6));
```

Základní přiřazení

Pro přiřazení existují tři přetížené metody:

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
```

První metoda přiřazuje jeden objekt třídy string druhému, druhá přiřazuje objektu třídy string řetězec ve stylu jazyka C a třetí přiřazuje objektu třídy string jediný znak. Všechny čtyři následující operace jsou tedy možné:

```
string name("Georege Wash");
string pres, veep, source;
pres = name;
veep = "Road Runner";
source = 'X';
```

Prohledávání řetězce

Třída `string` nabízí pro prohledávání šest funkcí a každá má čtyři prototypy. Stručně si je popíšeme.

Skupina metod find()

Prototypy metod `find()` jsou tyto:

```
size_type find(const basic_string& str, size_type pos = 0) const;
size_type find(const charT* s, size_type pos = 0) const;
size_type find(const charT* s, size_type pos, size_type n) const;
size_type find(charT c, size_type pos = 0) const;
```

První člen vrací pozici prvního výskytu podřetězce `str` ve vyvolávajícím objektu; hledání začne na pozici `pos`. Pokud se podřetězec nenajde, vrátí metoda hodnotu `npos`.

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter);          // nastaví loc1 na 1
size_type loc2 = longer.find(shorter, loc1 + 1); // nastaví loc2 na 16
```

Vzhledem k tomu, že druhé hledání začne na pozici 2 (tím je v řetězci `That` znak `a`), bude první výskyt podřetězce `hat` ten z konce řetězce. Chcete-li testovat úspěšnost operace, použijte hodnotu `string::npos`.

```
if (loc1 == string::npos)
  cout << "Podretezec nenalezen\n";
```

Druhá metoda dělá totéž, pouze místo objektu třídy `string` používá jako podřetězec pole znaků:

```
size_type loc3 = longer.find("is");             // nastaví loc3 na 5
```

Třetí metoda dělá totéž co druhá, ale použije pouze prvních n znaků řetězce `s`. Účinek je stejný jako při použití konstruktoru `basic_string(const charT* s, size_type n)` a výsledného objektu jako parametru objektu prvního tvaru metody `find()`. Například následující příkaz hledá podřetězec „`fun`":

```
size_type loc4 = longer.find("funds", 3);       // nastaví loc4 na 10
```

Čtvrtá metoda dělá totéž co první, pouze místo objektu třídy string používá jako podřetězec jediný znak:

```
size_type loc5 = longer.find('a');     // nastaví loc5 na 5
```

Skupina metod rfind()

Metody rfind() mají tyto prototypy:

```
size_type rfind(const basic_string& str, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(charT c, size_type pos = npos) const;
```

Tyto metody pracují analogicky metodám find(), ale hledají poslední výskyt řetězce nebo znaku od pozice pos nebo před ní. Pokud se podřetězec nenajde, vrátí metoda hodnotu npos.

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter);          // nastaví loc1 na 16
size_type loc2 = longer.rfind(shorter, loc1 - 1); // nastaví loc2 na 1
```

Skupina metod find_first_of()

Metody find_first_of() mají tyto prototypy:

```
size_type find_first_of(const basic_string& str, size_type pos = 0) const;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const;
```

Tyto metody fungují jako odpovídající metody find(), ale místo hledání celého odpovídajícího podřetězce hledají první výskyt jednoho odpovídajícího znaku v podřetězci.

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 = longer.find_first_of(shorter);  // nastaví loc1 na 10
size_type loc2 = longer.find_first_of("fat");    // nastaví loc2 na 2
```

Prvním výskytem kteréhokoli z pěti znaků obsažených v řetězci fluke, který se najde v objektu longer, bude znak f ze slova funny. Prvním výskytem kteréhokoli ze tří znaků obsažených v řetězci fat, který se najde v objektu longer, bude znak a ze slova That.

Skupina metod find_last_of()

Metody find_last_of() mají tyto prototypy:

```
size_type find_last_of(const basic_string& str, size_type pos = npos) const;
size_type find_last_of(const charT* s, size_type pos, size_type n) const;
size_type find_last_of(const charT* s, size_type pos = npos) const;
size_type find_last_of(charT c, size_type pos = npos) const;
```

Tyto metody fungují jako odpovídající metody rfind(), ale místo hledání celého odpovídajícího podřetězce hledají poslední výskyt jednoho odpovídajícího znaku v podřetězci.

```
string longer("That is a funny hat.");
```

```
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter);     // nastaví loc1 na 18
size_type loc2 = longer. find_last_of("any");      // nastaví loc2 na 17
```

Posledním výskytem kteréhokoli ze tří znaků obsažených v řetězci hat, který se najde v objektu longer, bude znak t ze slova hat. Prvním výskytem kteréhokoli ze tří znaků obsažených v řetězci any, který se najde v objektu longer, bude znak a ze slova hat.

Skupina metod find_first_not_of()

Metody find_first_not_of() mají tyto prototypy:

```
size_type find_first_not_of(const basic_string& str,
                            size_type pos = 0) const;
size_type find_first_not_of(const charT* s, size_type pos,
                            size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const;
```

Tyto metody fungují stejně jako odpovídající metody find_first_of(), ale hledají první výskyt kteréhokoli znaku, který se v podřetězci nenachází.

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter);    // nastaví loc1 na 2
size_type loc2 = longer.find_first_not_of("Thatch");   // nastaví loc2 na 4
```

Znak a ve slově That je v objektu longer první, který se nevyskytuje v řetězci This. První mezera v řetězci longer je prvním znakem, který není přítomen v řetězci Thatch.

Skupina metod find_last_not_of()

Metody find_last_not_of() mají tyto prototypy:

```
size_type find_last_not_of(const basic_string& str,
                           size_type pos = npos) const;
size_type find_last_not_of(const charT* s, size_type pos,
                           size_type n) const;
size_type find_last_not_of(const charT* s, size_type pos = npos) const;
size_type find_last_not_of(charT c, size_type pos = npos) const;
```

Tyto metody fungují stejně jako odpovídající metody find_last_of(), ale hledají poslední výskyt kteréhokoli znaku, který se v podřetězci nenachází.

```
string longer("That is a funny hat.");
string shorter("That");
size_type loc1 = longer.find_last_not_of(shorter); // nastaví loc1 na 15
size_type loc2 = longer.find_last_not_of(shorter, 10); //nastaví loc2 na 10
```

Poslední mezera v řetězci longer je posledním znakem, který není přítomen v podřetězci shorter. Znak f v řetězci longer je posledním znakem, který není přítomen v podřetězci shorter do pozice 10.

Metody a funkce pro porovnání řetězců

Třída `string` nabízí metody a funkce pro porovnání dvou řetězců. Nejprve se podíváme na prototypy metod:

```
int compare(const basic_string& str) const;
int compare(size_type pos1, size_type n1,
            const basic_string& str) const;
int compare(size_type pos1, size_type n1,
            const basic_string& str, size_type pos2,
            size_type n2) const;
int compare(const charT* s);
int compare(size_type pos1, size_type n1,
            const charT* s, size_type pos2 = npos) const;
```

Tyto metody používají metodu `traits::compare()` definovanou pro konkrétní typ znaku, používaný v řetězci. První metoda vrací hodnotu menší než nula, jestliže první řetězec předchází druhý podle pořadí stanoveném metodou `traits::compare()`. Jsou-li oba řetězce stejné, vrací hodnotu nula, a pokud první řetězec následuje za druhým, vrací hodnotu větší než nula. Jestliže jsou oba řetězce identické až do konce kratšího z obou řetězců, kratší řetězec delší předchází.

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3); // a13 je < 0
int a12 = s1.compare(s2); // a12 je > 0
```

Druhá metoda je jako první, ale pro porovnání používá pouze `n1` znaků od pozice `pos1`.

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2);    // a2 je 0
```

Třetí metoda je jako první, ale pro porovnání používá pouze `n1` znaků od pozice `pos1` v prvním řetězci a `n2` znaků od pozice `pos2` v řetězci druhém. Například následující kód porovnává podřetězec `out` z řetězce `stout` se stejným podřetězcem z řetězce `about`:

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3);    // a3 je 0
```

Čtvrtá metoda je jako první, ale pro porovnání používá jako druhý řetězec pole znaků místo objektu třídy `string`.

Pátá metoda je jako třetí, ale pro porovnání používá jako druhý řetězec pole znaků místo objektu třídy `string`.

Nečlenskými porovnávacími funkcemi jsou přetížené relační operátory:

```
operator==()
operator<()
```

```
operator<=()
operator>()
operator>=()
operator!=()
```

Každý operátor je přetížen tak, že může porovnávat dva objekty třídy `string`, objekt třídy string s řetězcovým polem a řetězcové pole s objektem třídy `string`. Definovány jsou podle metody `compare()`, takže umožňují pohodlnější způsob zápisu porovnání.

Modifikátory třídy string

Třída `string` nabízí několik metod pro úpravu řetězců. Většina má množství přetížených verzí, takže je lze použít s objekty třídy `string`, řetězcovými poli, jednotlivými znaky a u rozsahů stanovených iterátory.

Připojení a přidání

Jeden řetězec můžete ke druhému připojit pomocí přetíženého operátoru += nebo pomocí metody `append()`. V obou případech dojde k vyvolání výjimky `length_error`, pokud by měl být výsledný řetězec delší než je maximální délka řetězce. Operátory += umožňují k jinému řetězci připojit objekt třídy `string`, řetězcové pole nebo jednotlivý znak:

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

Metody `append()` také umožňují k jinému řetězci připojit objekt třídy `string`, řetězcové pole nebo jednotlivý znak. Kromě toho umožňují připojit část objektu třídy `string` zadáním počáteční pozice a počtu znaků, které se mají přidat, nebo zadáním rozsahu. Část řetězce můžete připojit, pokud zadáte, kolik znaků se má použít. Verze pro připojení znaku umožňuje zadat, kolik instancí daného znaku se má zkopírovat.

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(size_type n, charT c);     // připojí n kopií znaku c
```

Zde je několik příkladů:

```
string test("The");
test.append("ory");        // test bude obsahovat "Theory"
test.append(3, '!');       // test bude obsahovat "Theory!!!"
```

Funkce `operator+()` je přetížená pro spojování řetězců. Přetížené funkce řetězec nemění, ale vytvářejí řetězec nový, který obsahuje jeden řetězec připojený ke druhému. Funkce pro přidávání nejsou členskými funkcemi a umožňují přidat jeden objekt třídy `string` ke dru-

hému, řetězcové pole k objektu třídy `string`, objekt třídy `string` k řetězcovému poli, znak k objektu třídy `string` a objekt třídy `string` ke znaku. Zde je několik příkladů:

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce"; // st3 obsahuje "reduce"
string st4 = 't' + st1;   // st4 obsahuje "train"
string st5 = st1 + st2;   // st5 obsahuje "redrain"
```

Další přiřazení

Kromě základního operátoru přiřazení obsahuje třída `string` metody `assign()`, které umožňují přiřadit objektu třídy `string` celý řetězec, jeho část nebo sekvenci identických znaků.

```
basic_string& assign(const basic_string& str);
basic_string& assign(const basic_string& str, size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c); // přiřadí n kopií znaku c
template<class InputIterator>
   basic_string& assign(InputIterator first, InputIterator last);
```

Zde je několik příkladů:

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5);          // test obsahuje "et tu"
test.assign(6, '#');               // test obsahuje "######"
```

Metody vkládání

Metody `insert()` umožňují vložit do objektu třídy string objekt třídy string, řetězcové pole, znak nebo několik znaků. Metody jsou podobné metodám `append()`, ale mají další parametr označující, kam se nová data vloží. Tímto parametrem může být pozice nebo iterátor. Data se vloží před bod vložení. Některé metody vracejí referenci na výsledný řetězec. Jestliže se pozice `pos1` nachází za koncem cílového řetězce nebo jestliže se pozice `pos2` nachází za koncem řetězce, do kterého se má vkládat, vyvolá metoda výjimku `out_of_range`. Pokud bude výsledný řetězec delší než maximální velikost, metoda vyvolá výjimku `length_error`.

```
basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(iterator p, charT c = charT());
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
   void insert(iterator p, InputIterator first, InputIterator last);
```

Následující kód například vloží řetězec „former“ před znak b ve slově banker:

```
string st3("The banker.");
st3.insert(4, "former");
```

Potom následující kód vloží řetězec „waltzed“ (bez znaku !, který by byl devátým znakem) před tečku na konec řetězce „The former banker.“:

```
st3.insert(st3.size() - 1, "waltzed!", 8);
```

Metody odstraňující znaky

Metody erase() odstraňují znaky z řetězce. Zde jsou jejich prototypy:

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

První tvar odstraní n znaků od pozice pos nebo odstraní znaky do konce řetězce. Druhý odstraní jediný znak z pozice, na kterou ukazuje iterátor, a vrátí iterátor na prvek následující, nebo vrátí end() v případě, že již další prvky nejsou. Třetí tvar odstraní znaky z daného rozsahu (first, last; to znamená, včetně first, ale bez last). Metoda vrátí iterátor na prvek následující za prvkem zrušeným.

Metody pro nahrazování

Různé metody replace() identifikují část řetězce, která bude nahrazena a také identifikují, čím bude nahrazena. Část, která má být nahrazena, lze identifikovat pomocí počáteční pozice a počtem znaků, nebo rozsahem stanoveným iterátory. Nahrazující částí může být objekt třídy string, řetězcové pole nebo určitý znak několikrát duplikovaný. Nahrazující objekty třídy string a pole lze dále upravovat označením určité části, pomocí pozice a počtu, jenom pomocí počtu nebo rozsahem stanoveným iterátory.

```
basic_string& replace(size_type pos1, size_type n,
         const basic_string& str);
basic_string& replace(size_type pos1, size_type n1,
         const basic_string& str, size_type pos2, size_type n2);
basic_string& replace(size_type pos1, size_type n1, const charT* s,
  size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(iterator i1, iterator i2, const basic_string& str);
basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);
basic_string& replace(iterator i1, iterator i2, const charT* s);
basic_string& replace(iterator i1, iterator i2, size_type n, charT c);
template<class InputIterator>
  basic_string& replace(iterator i1, iterator i2,
               InputIterator j1, InputIterator j2);
```

Zde je příklad:

```
string test("Na rohu hlavni ulice zahnete doprava.");
test.replace(29, 7, "doleva");  // nahradí řetězec doprava řetězcem doleva
```

Další upravující metody: copy() a swap()

Metoda copy() kopíruje objekt třídy string nebo jeho část do určeného řetězcového pole:

```
size_type copy(charT*, size_type n, size_type pos = 0) const;
```

Zde ukazatel s ukazuje na cílové pole, parametr n označuje počet znaků, které se budou kopírovat a pos označuje pozici v objektu třídy string, od které kopírování začne. Kopíruje se buď n znaků nebo do posledního znaku v objektu. Funkce vrací počet zkopírovaných znaků. Metoda nepřipojuje nulový znak a je tedy na programátorovi, aby zajistil, že pole bude dostatečně velké.

Upozornění

Metoda copy() nepřipojuje nulový znak ani nekontroluje, zda je pole určení dostatečně velké.

Metoda swap() mění obsah dvou objektů třídy string pomocí konstantního časového algoritmu.

```
void swap(basic_string<charT>, traits, Allocator>&);
```

Výstup a vstup

Pro zobrazení objektů přetěžuje třída string operátor <<. Vrací referenci na objekt třídy istream, takže výstup lze řetězit:

```
string claim("Trida string ma mnoho vlastnosti.");
cout << claim << endln;
```

Třída string přetěžuje také operátor >>, takže můžete vstup načíst do řetězce:

```
string who;
cin >> who;
```

Vstup skončí načtením znaku konce souboru, načtením maximálního povoleného počtu znaků do řetězce nebo při dosažení bílého znaku. (Definice bílého znaku závisí na znakové sadě a na typu, který reprezentuje charT.)

Existují dvě funkce getline(). První má tento prototyp:

```
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>& getline(basic_istream<charT,traits>& is,
        basic_string<charT, traits, Allocator>& str, charT delim);
```

Metoda čte znaky ze vstupního proudu `is` do řetězce `str` až do znaku oddělovače `delim`, dosažení maximální velikosti řetězce nebo do načtení znaku konce souboru. Znak `delim` se přečte (je odstraněn ze vstupního proudu), ale neuloží. Druhá verze postrádá třetí parametr a místo oddělovače používá znak nového řádku (nebo jeho generalizaci):

```
string str1 str2;
getline(cin, str1);        // čte do konce řádku
getline(cin, str2, '.');   // čte do znaku tečka
```

DODATEK G

Metody a funkce knihovny STL

Cílem knihovny STL je nabídnout efektivní implementace běžných algoritmů. Tyto algoritmy jsou vyjádřeny všeobecnými funkcemi, které lze použít u libovolného kontejneru splňujícího požadavky konkrétního algoritmu, a v metodách, které lze použít s instancemi určitých kontejnerových tříd. Tato příloha předpokládá, že máte určité znalosti o knihovně STL, například získané přečtením kapitoly 15. Kapitola například předpokládá, že víte, co jsou iterátory a konstruktory.

Položky společné všem kontejnerům

Všechny kontejnery definují typy uvedené v tabulce G.1. Zde `X` označuje typ kontejneru (například `vector<int>`) a `T` je typ v kontejneru uložený (například `int`).

Tabulka G.1 Typy definované pro všechny kontejnery.

Typ	Hodnota
X::value_type	T, typ prvku
X::reference	Chová se jako T&
X::const_reference	Chová se jako const T&
X::iterator	Typ iterátor ukazující na T*, chová se jako T*
X::const_iterator	Typ iterátor ukazující na const T*, chová se jako const T*
X::difference_type	Celočíselný typ se znaménkem, použitý pro vyjádření vzdálenosti od jednoho iterátoru ke druhému, například vzdálenosti mezi dvěma ukazateli.
X::size_type	Celočíselný typ `size_type` bez znaménka může reprezentovat velikost datových objektů, počet prvků a indexy.

V definici třídy jsou tyto členy definovány příkazem `typedef`. Pomocí nich můžete deklarovat vhodné proměnné. Abyste viděli, jak lze pomocí typů položek deklarovat proměnné, je v následujícím příkladu první výskyt řetězce „`bonus`“ ve vektoru objektů třídy `string` nahrazen řetězcem „`bogus`“:

```
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
   input.push_back(temp);
vector<string>::iterator want =
   find(input.begin(), input.end(), string("bonus"));
if (want != input.end())
{
   vector<string>::reference r = *want;
   r = "bogus";
}
```

Tento kód udělá z proměnné r referenci na prvek v objektu `input`, na který ukazuje `want`.

Tyto typy lze také použít v obecnějším kódu, ve kterém typ kontejneru a prvku jsou generické. Předpokládejme například, že potřebujete funkci `min()`, která má jako parametr referenci na kontejner a vrací hodnotu nejmenší položky v kontejneru. To předpokládá, že pro tento typ hodnoty je definován operátor < a že nechcete použít algoritmus `min_element()` z knihovny STL, který používá iterátorové rozhraní. Protože parametrem by mohl být `vector<int>` nebo `list<string>` nebo `deque<double>`, použijte pro reprezentaci kontejneru šablonu s šablonovým parametrem, například `Bag`. Typem parametru funkce tedy bude const `Bag & b`. A co návratový typ? Měla by jím být hodnota typu kontejneru, to znamená `Bag::value_type`. V tomto okamžiku je však `Bag` pouze šablonový parametr a kompilátor nemá možnost zjistit, že položka `value_type` je skutečný typ. Pomocí klíčového slova `typename` však můžete dát najevo, že položka třídy je vytvořena příkazem `typedef`:

```
vector<string>::value_type st;    // vector<string> je definovaná třída
typename Bag::value_type m;       // Bag je zatím nedefinovaný typ
```

Pokud jde o první definici, kompilátor má přístup k definici šablony `vector`, která stanoví, že položka `value_type` je vytvořena pomocí příkazu `typedef`. U druhé definice klíčové slovo zajistí `typename`, že kombinace `Bag::value_type` je vytvořena pomocí příkazu typedef. Tyto úvahy vedou k následující definici:

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
   typename Bag::const iterator it;
   typename Bag::value_type m = *b.begin();
   for (it = b.begin(); it != b.end(); ++it)
           if (*it < m)
                   m = *it;
   return m;
}
```

Všechny kontejnery také obsahují členské funkce nebo operace uvedené v tabulce G.2. V ní `X` opět představuje typ kontejneru (například `vector<int>`) a `T` je typ v kontejneru uložený (například `int`). Také `a` a `b` jsou hodnoty typu `X`.

Tabulka G.2 Metody definované pro všechny kontejnery.

Metoda/Operace	Popis
begin()	Vrací iterátor na první prvek
end()	Vrací iterátor na prvek za posledním prvkem kontejneru
rbegin()	Vrací reverzní iterátor na prvek za posledním prvkem kontejneru
rend()	Vrací reverzní iterátor na první prvek
size()	Vrací počet prvků
maxsize()	Vrací největší možnou velikost kontejneru
empty()	Vrací hodnotu `true`, jestliže je kontejner prázdný

swap()	Mění obsah dvou kontejnerů
==	Vrací hodnotu true, jestliže mají dva kontejnery stejnou velikost a stejné prvky ve stejném pořadí
!=	a != b je totéž jako !(a == b)
<	Vrací hodnotu true, jestliže a lexikálně předchází b
>	a > b je totéž jako b < a
<=	a <= b je totéž jako !(a > b)
>== operator>=>	a >= b je totéž jako !(a < b)

Operátor > v kontejneru předpokládá, že je definován pro daný typ hodnoty. Lexikální porovnání je generalizací alfabetického třídění. Porovnává dva kontejnery prvek po prvku, dokud v jednom kontejneru nenarazí na prvek, který se nerovná odpovídajícímu prvku v kontejneru druhém. V takovém případě se předpokládá, že pořadí prvků v kontejnerech je stejné jako neodpovídající si dvojice prvků. Jestliže mají dva kontejnery například prvních deset prvků totožných, ale jedenáctý prvek v prvním kontejneru je menší než jedenáctý prvek v kontejneru druhém, předchází první kontejner druhý. Pokud jsou dva kontejnery stejné až do okamžiku, kdy jednomu z nich prvky dojdou, předchází kratší kontejner delší.

Další položky kontejnerů

vector, list a deque

Kontejnery vector (vektor), list (seznam) a deque (obousměrná fronta) jsou všechno sekvence a všechny obsahují metody uvedené v tabulce G.3. V ní je X opět typ kontejneru (například vector<int>) a T je typ v kontejneru uložený (například int), a je hodnota typu X, t hodnota typu X::value_type, i a j jsou vstupní iterátory, q2 a p jsou iterátory, q a q1 jsou dereferencovatelné iterátory (můžete na ně použít operátor *) a n je celé číslo typu X::size_type.

Tabulka G.3 Metody definované pro kontejnery vektor, seznam a obousměrná fronta.

Metoda	Popis
a.insert(p, t)	Vloží kopii t před p; vrátí iterátor ukazující na vloženou kopii t. Implicitní hodnotou t je T(), to je hodnota použitá pro typ T při absenci explicitní inicializace.
a.insert(p, n, t)	Vloží n kopií t před p; nevrací žádnou hodnotu.
a.insert(p, i, j)	Vloží kopie prvků z rozsahu [i, j) před p; nevrací žádnou hodnotu.
a.resize(n, t)	Jestliže platí výraz n > a.size(), vloží n - a.size() kopií t před a.end(); t má implicitní hodnotu T(), to je hodnota použitá pro typ T při absenci explicitní inicializace. Jestliže platí výraz n < a.size(), budou prvky za n-tým prvkem vymazány.
a.assign(i, j)	Nahradí aktuální obsah a kopiemi prvků v rozsahu [i, j).

a.assign(n, t)	Nahradí aktuální obsah a n kopiemi t. Implicitní hodnotou t je T(), to je hodnota použitá pro typ T při absenci explicitní inicializace.
a.erase(q)	Vymaže prvek, na který ukazuje q; vrátí iterátor na prvek, který následoval za q.
a.erase(q1, q2)	Vymaže prvky v rozsahu [q1, q2); vrátí iterátor ukazující na prvek, na který původně ukazoval q2.
a.clear()	Totéž jako erase(a.begin(), a.end()).
a.front()	Vrátí *a.begin() (první prvek).
a.back()	Vrátí *--a.end() (poslední prvek).
a.push_back(t)	Vloží t před a.end().
a.pop_back()	Vymaže poslední prvek.

V tabulce G.4 je seznam metod, které jsou společné dvěma ze tří sekvenčních tříd.

Tabulka G.4 Metody definované pro některé sekvence.

Metoda	Popis	Kontejner
a.push_front(t)	Vloží kopii t před první prvek.	seznam, obousměrná fronta
a.pop_front()	Vymaže první prvek.	seznam, obousměrná fronta
a[n]	Vrátí *(a.begin() + n).	vektor, obousměrná fronta
a.at(n)	Vrátí *(a.begin() + n), vyvolá výjimku, jestliže platí výraz n > a.size().	vektor, obousměrná fronta

Šablona seznam má další metody, uvedené v tabulce G.5. Zde a a b představují kontejnery seznam a T je typ v seznamu uložený (například int), t je hodnota typu T, i a j jsou vstupní iterátory, q2 a p jsou iterátory, q a q1 jsou dereferencovatelné iterátory a n je celé číslo typu X::size_type. V tabulce je použita standardní notace knihovny STL [i, j), která znamená, že i do rozsahu patří, ale j již ne.

Tabulka G.5 Další metody seznamů.

Metoda	Popis
a.splice(p, b)	Přesune obsah seznamu b do seznamu a, vloží ho před p.
a.splice(p, b, i)	Přesune prvek ze seznamu b, na který ukazuje i, do seznamu a před pozici p.
a.splice(p, b, i, j)	Přesune prvky v rozsahu [i, j) ze seznamu b do seznamu a před pozici p.
a.remove(const T& t)	Vymaže ze seznamu a všechny prvky s hodnotou t.
a.remove_if(Predicate pred)	Pokud je i iterátor do seznamu a, vymaže všechny hodnoty, pro které platí výraz pred(*i). (Predicate je booleovská funkce neboli funkční objekt, který byl probírán v kapitole 15.)

a.unique()	Vymaže všechny prvky kromě prvního z každé skupiny za sebou jdoucích stejných prvků.
a.unique (BinaryPredicate bin_pred)	Vymaže všechny prvky kromě prvního z každé skupiny za sebou jdoucích stejných prvků, pro které platí výraz bin_ pred(*i, *(i - 1)). (BinaryPredicate je booleovská funkce neboli funkční objekt, který byl probírán v kapitole 15.)
a.merge(b)	Sloučí obsah seznamu b se seznamem a, přičemž použije operátor <, definovaný pro daný typ hodnoty. Jestliže prvek v seznamu a je ekvivalentní prvku v seznamu b, bude prvek ze seznamu a umístěn jako první. Po sloučení bude seznam b prázdný.
a.merge(b, Compare comp)	Sloučí obsah seznamu b se seznamem a pomocí funkce comp nebo funkčního objektu. Jestliže prvek v seznamu a je ekvivalentní prvku v seznamu b, bude prvek ze seznamu a umístěn jako první. Po sloučení bude seznam b prázdný.
a.sort()	Setřídí seznam a pomocí operátoru <.
a.sort(Compare comp)	Setřídí seznam a pomocí funkce comp nebo funkčního objektu.
a.reverse()	Umístí prvky v seznamu a v obráceném pořadí.

Další položky kontejnerů

set a map

Asociativní kontejnery, jejichž modely jsou set (množina) a map (mapa), mají šablonové parametry Key a Compare označující typ klíče použitého pro uspořádání obsahu, a funkční objekt, nazývaný *porovnávací objekt* (comparison object), který se používá pro porovnání hodnot klíčů. U kontejnerů set a multiset jsou uložené klíče uloženými hodnotami, takže typ klíče je stejný jako typ hodnoty. U kontejnerů map a multimap jsou uložené hodnoty jednoho typu (šablonový parametr T) sdruženy s typem klíče (šablonový parametr Key) a typem hodnoty je pair<const Key, T>. Asociativní kontejnery zavádějí pro popis těchto vlastností další položky, které jsou uvedeny v tabulce G.6.

Tabulka G.6 Typy definované pro asociativní kontejnery.

Typ	Hodnota
X::key_type	Key, typ klíče.
X::key_compare	Compare, jehož implicitní hodnotou je less<key_type>.
X::value_compare	Binární predikátní typ, který je pro kontejnery set a multiset stejný jako key_compare a který řadí hodnoty typu pair<const Key, T> v kontejnerech map a multimap.

X::mapped_type	T, typ sdružených dat (pouze u kontejnerů map a multimap).

Asociativní kontejnery obsahují metody uvedené v tabulce G.7. Obecně platí, že porovnávací objekt nemusí požadovat, aby hodnoty se stejným klíčem byly identické; termín *ekvivalentní klíče* (equivalent keys) znamená, že dvě hodnoty, které mohou, ale nemusí být sobě rovné, mají stejný klíč. V tabulce X označuje třídu kontejneru, a a je objekt typu X. Jestliže X používá unikátní klíč (kontejnery set a map), je a_eq objekt typu X. Stejně jako dříve jsou i a j vstupní iterátory odkazující na prvky typu value_type, [i, j) je platný rozsah, p a q2 jsou iterátory na a, q a q1 jsou dereferencovatelné iterátory na a, [q1, q2) je platný rozsah, t je hodnota typu X::value_type (může jí být dvojice) a k je hodnotou typu X::key_type.

Tabulka G.7 Metody definované pro kontejnery set, multiset, map a multimap.

Metoda	Popis
a.key_comp()	Vrátí porovnávací objekt použitý při vytváření a.
a.value_comp()	Vrátí objekt typu value_compare.
a.unique_insert(t)	Vloží hodnotu t do kontejneru a pouze tehdy, jestliže a neobsahuje hodnotu s ekvivalentním klíčem. Metoda vrací hodnotu typu pair<iterator, bool>. Komponenta bool má hodnotu true, jestliže vložení proběhlo, jinak má hodnotu false. Iterátor ukazuje na prvek, jehož klíč je ekvivalentní klíči hodnoty t.
a.eq_insert(t)	Vloží hodnotu t a vrátí iterátor na toto místo.
a. insert(p, t)	Vloží hodnotu t a pomocí iterátoru p určí, kde metoda insert() začne hledání. Jestliže je a kontejner s unikátními klíči, vložení se provede pouze tehdy, jestliže a neobsahuje prvek s ekvivalentním klíčem; jinak se vložení neuskuteční. Bez ohledu na úspěch operace vložení vrátí metoda iterátor na místo s ekvivalentním klíčem.
a.insert(i, j)	Vloží do a prvky z rozsahu [i, j).
a.erase(k)	Vymaže v a všechny prvky, jejichž klíče jsou ekvivalentní k a vrátí počet vymazaných prvků.
a.erase(q)	Vymaže prvek, na který ukazuje iterátor q.
a.erase(q1, q2)	Vymaže prvky v rozsahu [q1, q2).
a.clear()	Totéž jako erase(a.begin(), a.end()).
a.find(k)	Vrátí iterátor na prvek, jehož klíč je ekvivalentní k; jestliže se takový prvek nenajde, vrátí a.end().
a.count(k)	Vrátí počet prvků s klíčem ekvivalentním k.
a.lower_bound(k)	Vrátí iterátor na první prvek s klíčem, který není menší než k.
a.upper_bound(k)	Vrátí iterátor na první prvek s klíčem větším než k.
a.equal_range(k)	Vrátí dvojici, jejíž první položkou je a.lower_bound(k) a druhou a.upper_bound(k).
a.operator[](k)	Vrátí referenci na hodnotu sdruženou s klíčem k (platí pouze pro kontejnery map).

Funkce knihovny STL

Knihovna algoritmů knihovny STL, podporovaná hlavičkovými soubory `algorithm` a `numeric`, obsahuje velké množství nečlenských funkcí založených na iterátorových šablonách. Jak jsme uvedli v kapitole 15, jsou názvy šablonových parametrů voleny tak, aby označovaly koncept, který mají konkrétní parametry modelovat. Například `ForwardIterator` se používá pro označení parametru, který by měl minimálně modelovat požadavky dopředného iterátoru, zatímco `Predicate` se používá pro označení parametru, kterým by měl být funkční objekt s jedním parametrem a návratovou hodnotou typu `bool`. Standard rozděluje algoritmy do čtyřech skupin: nemodifikující sekvenční operace, změnové sekvenční operace třídicí a relační operace, a numerické operace. Termín *sekvenční operace* označuje, že funkce má jako parametry dvojici iterátorů definujících rozsah nebo sekvenci, nad kterou se bude pracovat. Termín *změnové* znamená, že funkce může měnit obsah kontejneru.

Nemodifikující sekvenční operace

Nemodifikující sekvenční operace jsou shrnuty v tabulce G.8. Parametry nejsou uvedeny a přetížené funkce jsou v seznamu pouze jednou. Podrobnější popis včetně prototypů následuje za tabulkou. Prohlédnutím tabulky tedy získáte představu, co která funkce dělá, a pokud vás některá zaujme, můžete si vyhledat podrobnosti.

Tabulka G.8 Nemodifikující sekvenční operace.

Funkce	Popis
for_each()	Použije na každý prvek v rozsahu nemodifikující funkční objekt.
find()	Najde první výskyt hodnoty v rozsahu.
find_if()	Najde první hodnotu splňující v rozsahu kritérium testu predikátu.
find_end()	Najde poslední výskyt subsekvence, jejíž hodnoty odpovídají hodnotám druhé sekvence. Srovnávat lze na základě rovnosti nebo použitím binárního predikátu.
find_first_of()	Najde první výskyt libovolného prvku z druhé sekvence, který odpovídá hodnotě v první sekvenci. Srovnávat lze na základě rovnosti nebo použitím binárního predikátu.
adjacent_find()	Najde první prvek, který odpovídá prvku bezprostředně následujícímu. Srovnávat lze na základě rovnosti nebo použitím binárního predikátu.
count()	Vrátí číslo udávající, kolikrát se daná hodnota v rozsahu vyskytuje.
count_if()	Vrátí číslo udávající, kolikrát se daná hodnota v rozsahu vyskytuje, přičemž srovnání se provede pomocí binárního predikátu.
mismatch()	Najde první prvek v jednom rozsahu, který neodpovídá odpovídajícímu prvku v rozsahu druhém a vrátí iterátory na oba. Srovnávat lze na základě rovnosti nebo použitím binárního predikátu.
equal()	Vrátí hodnotu `true`, jestliže se každý prvek v jednom rozsahu rovná odpovídajícímu prvku v rozsahu druhém. Srovnávat lze na základě rovnosti nebo použitím binárního predikátu.

search()	Najde první výskyt subsekvence, jejíž hodnoty odpovídají hodnotám druhé sekvence. Srovnávat lze na základě rovnosti nebo použitím binárního predikátu.
search_n()	Najde první subsekvenci s `n` prvky, které mají všechny danou hodnotu. Srovnávat lze na základě rovnosti nebo použitím binárního predikátu.

Nyní se podívejme na prototypy. Dvojice iterátorů označují rozsahy, přičemž zvolený název šablonového parametru označuje typ iterátoru. Jako obvykle rozsah ve tvaru `[first, last)` začíná na pozici `first`, ale pozice `last` do něho již nepatří. Některé funkce mají rozsahy dva, avšak druh kontejneru nemusí být stejný. Pomocí funkce `compare()` můžete například porovnat kontejnery `list` a `vector`. Funkce předané jako parametry jsou funkčními objekty, což mohou být ukazatele (příkladem jsou názvy funkcí) nebo objekty, pro něž je definována operace (). Jak bylo uvedeno v kapitole 15, predikát je booleovská funkce s jedním parametrem, zatímco binární predikát je booleovská funkce se dvěma parametry. (Funkce nemusí být typu `bool`, pokud bude vracet hodnotu `0` jako hodnotu `false` a nenulovou hodnotu jako hodnotu `true`.)

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

Funkce `for_each()` používá na každý prvek v rozsahu `[first, last)` funkční objekt `f` a také ho vrací.

```
template<class InputIterator, class T>
Function find(InputIterator first, InputIterator last, const T& value);
```

Funkce `find()` vrací iterátor na první prvek v rozsahu `[first, last)`, který má hodnotu `value`.

```
template<class InputIterator, class Predicate>
Function find_if(InputIterator first, InputIterator last,
                                     Predicate pred);
```

Funkce `find_if()` vrací iterátor `it` na první prvek v rozsahu `[first, last)`, pro který volání funkčního objektu `pred(*i)` vrátí hodnotu `true`.

```
template<class ForwardIterator1, class ForwardIterator2>
Function find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredi-
cate>
Function find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);
```

Funkce `find_end()` vrací iterátor `it` na poslední prvek v rozsahu `[first1, last1)`, který označuje počátek subsekvence odpovídající rozsahu `[first2, last2)`. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují `it1` a `it2` jsou si rovny, jestliže platí výraz pred(*it1, *it2).

```
template<class ForwardIterator1, class ForwardIterator2>
```

```
Function find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredi-
cate>
Function find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);
```

Funkce `find_first_of()` vrací iterátor it na první prvek v rozsahu `[first1, last1)`, který odpovídá libovolnému prvku z rozsahu `[first2, last2)`. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují `it1` a `it2` jsou si rovny, jestliže platí výraz `pred(*it1, *it2)`.

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
BinaryPredicate pred);
```

Funkce `adjacent_find()` vrací iterátor it na první prvek v rozsahu `[first1, last1)`, který odpovídá prvku následujícímu. Jestliže se takový prvek nenajde, vrátí funkce last. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují `it1` a `it2` jsou si rovny, jestliže platí výraz `pred(*it1, *it2)`.

```
template<class InputIterator, class T>
iterator_traits<InputIterator>::difference_type count (
        InputIterator first, InputIterator last, const T& value);
```

Funkce `count()` vrací počet prvků v rozsahu `[first, last)`, které odpovídají hodnotě value. Pro porovnání hodnot se používá operátor ==. Návratovým typem je číslo typu `integer`, které je dostatečně velké, aby mohlo obsahovat maximální počet položek, které může kontejner pojmout.

```
template<class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type count_if (
        InputIterator first, InputIterator last, Predicate pred);
```

Funkce `count_if()` vrací počet prvků v rozsahu `[first, last)`, pro které funkční objekt pred vrátí hodnotu `true`, když je prvek předán jako parametr.

```
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch (InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);
```

Tato verze funkce `mismatch()` najde první prvek v rozsahu `[first1, last1)`, který se nerovná odpovídajícímu prvku v rozsahu začínajícím na `first2`, a vrátí dvojici obsahující iterátory na oba neshodné prvky. Pokud k žádné neshodě nedojde, bude návratová

hodnota pair<last1, first2 + (last1 - first1)>. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují it1 a it2 si nejsou rovny, jestliže neplatí výraz pred(*it1, *it2).

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputItertor1 last1,
        InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputItertor1 last1,
        InputIterator2 first2,
        BinaryPredicate pred);
```

Funkce equal() vrací hodnotu true, jestliže každý prvek v rozsahu [first1, last1) je roven odpovídajícímu prvku v sekvenci začínající na first2, a hodnotu false v případě opačném. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují it1 a it2 jsou si rovny, jestliže platí výraz pred(*it1, *it2).

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardItertor1 last1,
    ForwardIterator2 first2, ForwardItertor2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredi-
cate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardItertor1 last1,
    ForwardIterator2 first2, ForwardItertor2 last2, BinaryPredicate pred);
```

Funkce search() najde první výskyt subsekvence v rozsahu [first1, last1), která se rovná odpovídající sekvenci nalezené v rozsahu [first2, last2). Jestliže se taková sekvence nenajde, vrátí last1. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují it1 a it2 jsou si rovny, jestliže platí výraz pred(*it1, *it2)).

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardItertor last,
        Size count, const T& value);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredi-
cate>
ForwardIterator search_n(ForwardIterator first, ForwardItertor last,
        Size count, const T& value, BinaryPredicate pred);
```

Funkce search_n() najde první výskyt subsekvence v rozsahu [first, last), který odpovídá sekvenci sestávající z count za sebou jdoucích výskytů s hodnotou value. Jestliže se taková sekvence nenajde, vrátí last. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují it1 a it2 jsou si rovny, jestliže platí výraz pred(*it1, *it2).

Změnové sekvenční operace

V tabulce G.9 jsou shrnuty změnové sekvenční operace. Parametry nejsou uvedeny a přetížené funkce jsou v seznamu pouze jednou. Podrobnější popis včetně prototypů následuje za tabulkou. Prohlédnutím tabulky tedy získáte představu, co která funkce dělá, a pokud vás některá zaujme, můžete si vyhledat podrobnosti.

Tabulka G.9 Změnové sekvenční operace.

Funkce	Popis
copy()	Kopíruje prvky z rozsahu na místo určené iterátorem.
copy_backward()	Kopíruje prvky z rozsahu na místo určené iterátorem. Kopírování začne na konci rozsahu a pokračuje na začátek.
swap()	Vymění dvě hodnoty uložené v místech určených referencemi.
swap_ranges()	Vymění odpovídající hodnoty ve dvou rozsazích.
iter_swap()	Vymění dvě hodnoty uložené v místech určených iterátory.
transform()	Použije funkční objekt na každý prvek v rozsahu (nebo každou dvojici prvků ve dvou rozsazích) a návratovou hodnotu zkopíruje na odpovídající místo v jiném rozsahu.
replace()	Nahradí každý výskyt hodnoty v rozsahu jinou hodnotou.
replace_if()	Nahradí každý výskyt hodnoty v rozsahu jinou hodnotou, jestliže funkční objekt predikátu použitý na původní hodnotu vrátí hodnotu `true`.
replace_copy()	Zkopíruje jeden rozsah do druhého, přičemž každý výskyt specifikované hodnoty nahradí jinou hodnotou.
replace_copy_if()	Zkopíruje jeden rozsah do druhého, přičemž každou hodnotu, pro kterou funkční objekt predikátu vrátí hodnotu `true`, nahradí stanovenou hodnotou.
fill()	Nastaví každou hodnotu v rozsahu na stanovenou hodnotu.
fill_n()	Nastaví `n` za sebou jdoucích prvků na určitou hodnotu.
generate()	Nastaví každou hodnotu v rozsahu na návratovou hodnotu generátoru, což je funkční objekt bez parametrů.
generate_n()	Nastaví prvních `n` prvků v rozsahu na návratovou hodnotu generátoru, což je funkční objekt bez parametrů.
remove()	Odstraní z rozsahu všechny výskyty určené hodnoty a pro výsledný rozsah vrátí iterátor ukazující na prvek za posledním prvkem kontejneru.
remove_if()	Odstraní z rozsahu všechny výskyty hodnoty, pro kterou objekt predikátu z rozsahu vrátí hodnotu `true`, a vrátí iterátor ukazující na prvek za poslední prvkem kontejneru.
remove_copy()	Kopíruje prvky z jednoho rozsahu do druhého, přičemž vynechává prvky odpovídající určité hodnotě.

remove_copy_if()	Kopíruje prvky z jednoho rozsahu do druhého, přičemž vynechává prvky, pro které funkční objekt predikátu vrátí hodnotu `true`.
unique()	Zredukuje každou sekvenci dvou nebo více ekvivalentních prvků v rozsahu na jediný prvek.
unique_copy()	Kopíruje prvky z jednoho rozsahu do druhého, přičemž zredukuje každou sekvenci dvou nebo více ekvivalentních prvků v rozsahu na jediný prvek.
reverse()	Obrátí pořadí prvků v rozsahu.
reverse_copy()	Kopíruje obsah jednoho rozsahu do druhého rozsahu v obráceném pořadí.
rotate()	Zachází s rozsahem jako s kruhovým uspořádáním a otočí prvky doleva.
rotate_copy()	Kopíruje jeden rozsah do druhého v rotačním uspořádání.
random_shuffle()	Libovolně změní pořadí prvků v rozsahu.
partition()	Všechny prvky odpovídající funkčnímu objektu predikátu umístí před prvky, které mu neopovídají.
stable_partition()	Všechny prvky odpovídající funkčnímu objektu predikátu umístí před prvky, které mu neodpovídají. Relativní pořadí prvků v každé skupině zůstane zachováno.

Nyní se podívejme na prototypy. Jak jste viděli dříve, dvojice iterátorů označují rozsahy, přičemž zvolený název šablonového parametru označuje typ iterátoru. Jako obvykle rozsah ve tvaru `[first, last)` začíná na pozici `first`, ale pozice `last` do něho již nepatří. Funkce předané jako parametry jsou funkčními objekty, což mohou být ukazatele (příkladem jsou názvy funkcí) nebo objekty, pro něž je definována operace `()`. Jak bylo uvedeno v kapitole 15, predikát je booleovská funkce s jedním parametrem, zatímco binární predikát je booleovská funkce se dvěma parametry. (Funkce nemusí být typu `bool`, pokud bude vracet hodnotu `0` jako hodnotu `false` a nenulovou hodnotu jako hodnotu `true`.) V kapitole 15 jste se také dozvěděli, že unární funkční objekt má jediný parametr, zatímco binární má parametry dva.

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator, first, InputIterator last,
                    OutputIterator result);
```

Funkce `copy()` kopíruje prvky z rozsahu `[first, last)` do rozsahu `[result, result + (last - first))`. Vrací `result + (last - first)`, to znamená iterátor ukazující na místo za poslední zkopírovaný prvek. Funkce požaduje, aby se pozice result nenacházela v rozsahu [first, last), cíl tedy nemůže překrývat zdroj.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1, first,
BidirectionalIterator1, last, BidirectionalIterator2 result);
```

Funkce `copy_backward()` kopíruje prvky z rozsahu `[first, last)` do rozsahu `[result, result - (last - first)`, result). Kopírování začíná zkopírováním prvku z pozice `last - 1` na pozici `result - 1` a pokračuje zpětně k pozici `first`. Funkce vrací `result - (last`

- first), to znamená iterátor ukazující na místo za poslední zkopírovaný prvek. Funkce požaduje, aby se pozice result nenacházela v rozsahu [first, last). Protože však kopírování probíhá odzadu, je možné, aby se cíl a zdroj překrývaly.

```
template<class T> void swap(T& a, T& b);
```

Funkce swap() mění hodnoty uložené na dvou místech určených referencemi.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first, ForwardIterator1 last,
                             ForwardIterator2 first2);
```

Funkce swap_ranges() mění hodnoty v rozsahu [first1, last1) za odpovídající hodnoty v rozsahu začínajícím na pozici first2. Oba rozsahy se nesmí překrývat.

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Funkce iter_swap() mění hodnoty uložené na dvou místech určených ukazateli.

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);
```

Tato verze funkce transform() používá unární funkční objekt op na každý prvek v rozsahu [first1, last1) a návratovou hodnotu přiřazuje odpovídajícímu prvku v rozsahu s počátkem na pozici result. To znamená, že prvek na pozici result je nastaven na op(*first1, *first2) a tak dále. Vrací result + (last - first), hodnotu prvku za posledním prvkem v cílovém rozsahu.

```
template<class ForwardIterator, class T)
void replace(ForwardIterator first, ForwardIterator last, const T& old_value,
        const T& new_value);
```

Funkce replace() nahrazuje každý výskyt hodnoty old_value v rozsahu [first, last) hodnotou new_value.

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
        Predicate pred, const T& new_value);
```

Funkce replace_if() nahrazuje každou původní hodnotu v rozsahu [first, last), pro kterou je pravdivý výraz pred(old) hodnotou new_value.

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
        OutputIterator result, const T& old_value,
        const T& new_value);
```

Funkce replace_copy() kopíruje prvky z rozsahu [first, last) do rozsahu s počátkem na pozici result, ale každý výskyt hodnoty old_value nahrazuje hodnotou new_value. Vrací result + (last - first), hodnotu prvku za posledním prvkem v cílovém rozsahu.

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
        OutputIterator result, Predicate pred, const T& new_value);
```

Funkce replace_copy_if() kopíruje prvky z rozsahu [first, last) do rozsahu s počátkem na pozici result, ale každou původní hodnotu, pro kterou je pravdivý výraz pred(old) nahrazuje hodnotou new_value. Vrací result + (last - first), hodnotu prvku za posledním prvkem v cílovém rozsahu.

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

Funkce fill() nastaví každý prvek v rozsahu [first, last) na hodnotu value.

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

Funkce fill() nastaví prvních n prvků od pozice first na hodnotu value.

```
template class<ForwardIterator class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

Funkce generate() nastaví každý prvek v rozsahu [first, last) na hodnotu gen(), kde gen je funkční objekt generátoru, který nemá žádné parametry. Gen může být například ukazatel na funkci rand().

```
template<class OutputIterator, class Size, class T>
void generate_n(OutputIterator first, Size n, Generator gen);
```

Funkce generate_n() nastaví prvních n prvků v rozsahu s počátkem na pozici first na hodnotu gen(), kde gen je funkční objekt generátoru, který nemá žádné parametry. Gen může být například ukazatel na funkci rand().

```
template class<ForwardIterator class T>
void remove(ForwardIterator first, ForwardIterator last, const T& value);
```

Funkce remove() odstraňuje všechny výskyty hodnot z rozsahu [first, last) a vrací iterátor ukazující na prvek za posledním prvkem výsledného rozsahu. Funkce je stabilní, to znamená, že pořadí neodstraněných prvků se nezmění.

Poznámka

Vzhledem k tomu, že různé funkce remove() a unique() nejsou funkcemi členskými a protože nejsou omezeny na kontejnery knihovny STL, nemohou velikost kontejneru měnit. Místo toho vracejí iterátor ukazující na prvek bezprostředně za posledním prvkem nové oblasti. Obvykle jsou odstraněné prvky prostě přesunuty na konec kontejneru. U kontejnerů knihovny STL však můžete pomocí vráceného iterátoru a jedné z metod erase() nastavit funkci end().

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
        Predicate pred);
```

Funkce remove_if() odstraňuje všechny výskyty hodnot z rozsahu [first, last), pro které je pravdivý výraz pred(val) a vrací iterátor ukazující na prvek za posledním prvkem

výsledného rozsahu. Funkce je stabilní, to znamená, že pořadí neodstraněných prvků se nezmění.

```
template<class Iterator, class OutputIterator, class T>
OutputIterator remove_copy (Iterator first, Iterator last,
          OutputIterator result, const T& new_value);
```

Funkce `remove_copy()` kopíruje hodnoty z rozsahu `[first, last)` do rozsahu s počátkem na pozici `result`, přičemž hodnoty value přeskakuje. Vrací iterátor ukazující na prvek za posledním prvkem výsledného rozsahu. Funkce je stabilní, to znamená, že pořadí neodstraněných prvků se nezmění.

```
template<class Iterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(Iterator first, Iterator last,
          OutputIterator result, Predicate pred);
```

Funkce `remove_copy_if()` kopíruje hodnoty z rozsahu `[first, last)` do rozsahu s počátkem na pozici `result`, ale přeskakuje hodnoty `val`, pro které je pravdivý výraz `pred(val)`. Vrací iterátor ukazující na prvek za posledním prvkem výsledného rozsahu. Funkce je stabilní, to znamená, že pořadí neodstraněných prvků se nezmění.

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
BinaryPredicate pred);
```

Funkce `unique()` zredukuje každou sekvenci dvou nebo více ekvivalentních prvků v rozsahu `[first, last)` na jediný prvek a vrátí iterátor ukazující na prvek za posledním prvkem nového rozsahu. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu pred. To znamená, že prvky, na které ukazují `it1` a `it2` jsou si rovny, jestliže platí výraz `pred(*it1, *it2)`.

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

Verze `unique_copy()` kopíruje prvky z rozsahu `[first, last)` do rozsahu s počátkem na pozici `result`, přičemž každou sekvenci dvou nebo více totožných prvků zredukuje na prvek jediný. Vrací iterátor ukazující na prvek následující bezprostředně za posledním prvkem nového rozsahu. První verze používá pro porovnání prvků operátor ==. Druhá verze porovnává prvky pomocí funkčního objektu binárního predikátu `pred`. To znamená, že prvky, na které ukazují `it1` a `it2` jsou si rovny, jestliže platí výraz `pred(*it1, *it2)`.

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Funkce `reverse()` obrátí pořadí prvků z rozsahu `[first, last)` vyvoláním funkce `swap(first, last - 1)` a tak dále.

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
OutputIterator result);
```

Funkce `reverse_copy()` kopíruje prvky z rozsahu `[first, last)` do rozsahu s počátkem na pozici `result` v obráceném pořadí. Oba rozsahy se nesmí překrývat.

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
ForwardIterator last);
```

Funkce `rotate()` provede levou rotaci na prvky z rozsahu `[first, last`. Prvek z pozice middle se posune na pozici `first`, prvek `middle + 1` na pozici `first + 1` a tak dále. Prvky před `middle` jsou přesunuty na konec kontejneru, takže prvek na pozici `first` bude za prvkem, který byl původně na pozici `last - 1`.

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
ForwardIterator last, OutputIterator result);
```

Funkce `rotate_copy()` kopíruje prvky z rozsahu `[first, last)` do rozsahu s počátkem na pozici `result`, přičemž použije rotovanou sekvenci popsanou u funkce `rotate()`.

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

Tato verze funkce `random_shuffle()` mění pořadí prvků v rozsahu `[first, last)`. Distribuce je rovnoměrná, to znamená, že změna pořadí kteréhokoli z prvků je stejně pravděpodobná.

```
template<class RandomAccessIterator, class RandomNumbergenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last
RandomNumbergenerator& random);
```

Tato verze funkce `random_shuffle()` mění pořadí prvků v rozsahu `[first, last)`. Distribuci určuje funkční objekt `random`. Při n prvcích by měl výraz `random(n)` vrátit hodnotu v rozsahu `[0, n)`.

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

Funkce `partition()` umístí každý prvek s hodnotou `val`, pro který platí výraz `pred(val)`, před všechny prvky, které tento test nesplňují. Vrátí iterátor na pozici za poslední pozicí, která obsahuje hodnotu, pro niž funkční objekt predikátu vrátí hodnotu `true`.

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       Predicate pred);
```

Funkce `stable_partition()` umístí každý prvek s hodnotou `val`, pro který platí výraz `pred(val)`, před všechny prvky, které tento test nesplňují. Relativní pořadí prvků v obou rozsazích zůstane zachováno. Funkce vrátí iterátor na pozici za poslední pozicí, která obsahuje hodnotu, pro niž funkční objekt predikátu vrátí hodnotu `true`.

Třídicí a relační operace

Třídicí a relační operace jsou shrnuty v tabulce G.10. Parametry nejsou uvedeny a přetížené funkce jsou v seznamu pouze jednou. Každá funkce má verzi používající pro řazení prvků operátor < a verzi, která prvky řadí pomocí porovnávacího funkčního objektu. Podrobnější popis včetně prototypů následuje za tabulkou. Prohlédnutím tabulky tedy získáte představu, co která funkce dělá, a pokud vás některá zaujme, můžete si vyhledat podrobnosti.

Tabulka G.10 Třídicí a relační operace.

Funkce	Popis
sort()	Třídí rozsah.
stable_sort()	Třídí rozsah a zachovává relativní pořadí ekvivalentních prvků.
partial_sort()	Třídí rozsah částečně, prvních `n` prvků setřídí úplně.
partial_sort_copy()	Zkopíruje částečně setříděný rozsah do jiného rozsahu.
nth_element()	Pro určitý iterátor do rozsahu najde prvek, který by se na této pozici nacházel při setříděném seznamu a prvek sem umístí.
lower_bound()	Pro určitou hodnotu najde první pozici v setříděném rozsahu, před kterou lze tuto hodnotu vložit, a přitom zachová pořadí.
upper_bound()	Pro určitou hodnotu najde poslední pozici v setříděném rozsahu, před kterou lze tuto hodnotu vložit, a přitom zachová pořadí.
equal_range()	Pro určitou hodnotu najde největší podrozsah v setříděném rozsahu, aby tato hodnota mohla být vložena před libovolný prvek v podrozsahu a přitom nenarušit pořadí prvků.
binary_search()	Vrátí hodnotu `true`, jestliže setříděný rozsah obsahuje hodnotu ekvivalentní dané hodnotě. V opačném případě vrátí hodnotu `false`.
merge()	Sloučí dva setříděné rozsahy do rozsahu třetího.
inplace_merge()	Sloučí na místě dva spojité setříděné rozsahy.
includes()	Vrátí hodnotu `true`, jestliže každý prvek v jedné množině se také nachází v množině druhé.
set_union()	Vytvoří sjednocení dvou množin, což je množina obsahující všechny prvky přítomné v obou množinách.

set_intersection()	Vytvoří průnik dvou množin, což je množina obsahující pouze ty prvky, které se nacházejí v obou množinách.
set_difference()	Vytvoří rozdíl dvou množin, což je množina obsahující pouze ty prvky, které se nacházejí v množině první, ale ne v množině druhé.
set_symmetric_difference()	Vytvoří množinu obsahující prvky, které se nacházejí v množině první nebo v množině druhé, ale ne v obou.
make_heap()	Převede rozsah na haldu.
push_heap()	Přidá prvek na haldu.
pop_heap()	Odebere z haldy největší prvek.
sort_heap()	Setřídí haldu.
min()	Vrátí menší z obou hodnot.
max()	Vrátí větší z obou hodnot.
min_element()	Najde první výskyt nejmenší hodnoty v rozsahu.
max_element()	Najde první výskyt největší hodnoty v rozsahu.
lexicographic_compare()	Porovná lexikálně dvě sekvence a vrátí hodnotu `true`, jestliže první sekvence je lexikálně menší než sekvence druhá. V opačném případě vrátí `false`.
next_permutation()	Vygeneruje v sekvenci další změnu pořadí.
previous_permutation()	Vygeneruje v sekvenci předchozí změnu pořadí.

Funkce v této části určují pořadí dvou prvků pomocí operátoru < definovaného pro tyto prvky nebo pomocí porovnávacího objektu stanoveného šablonovým typem `Compare`. Jestliže je `comp` objekt typu `Compare`, je výraz `comp(a, b)` generalizací výrazu `a < b` a vrátí hodnotu `true`, pokud `a` podle třídicího schématu předchází `b`. Jestliže oba výrazy `a < b` a `b < a` vrátí hodnotu `false`, jsou si `a` a `b` rovny. Porovnávací objekt musí obsahovat alespoň striktně slabé uspořádání. To znamená následující:

```
Výraz comp(a, a) musí vrátit hodnotu false, protože hodnota nemůže být
menší než je sama. (To je striktní část.)
Jestliže výrazy comp(a, b)  i comp(b, c) vrátí hodnotu true, potom vrátí
hodnotu true i výraz comp(a, c) (porovnání je tedy tranzitivní vztah).
Jestliže a = b a b = c, potom se a = c (ekvivalence je tedy tranzitivní
vztah).
```

Jestliže uvažujete o používání operátoru < u celých čísel, potom ekvivalence znamená rovnost. V obecnějších případech to však platit nemusí. Mohli byste například definovat strukturu s několika položkami, popisujícími poštovní adresu a definovat porovnávací objekt `comp`, který strutury setřídí podle poštovního směrovacího čísla. V takovém případě dvě adresy se stejným poštovním směrovacím číslem budou ekvivalentní, ale ne rovné.

Nyní přejděme k prototypům. Tuto část rozdělíme do několika dílčích částí. Jak jste viděli dříve, dvojice iterátorů označují rozsahy, přičemž zvolený název šablonového parametru označuje typ iterátoru. Jako obvykle rozsah ve tvaru `[first, last)` začíná na pozici `first`, ale pozice `last` do něho již nepatří. Funkce předané jako parametry jsou funkčními objekty, což mohou být ukazatele (příkladem jsou názvy funkcí) nebo objekty, pro

něž je definována operace (). Jak jste se dozvěděli v kapitole 15, predikát je booleovská funkce s jedním parametrem, zatímco binární predikát je booleovská funkce se dvěma parametry. (Funkce nemusí být typu `bool`, pokud bude vracet hodnotu `0` jako hodnotu `false` a nenulovou hodnotu jako hodnotu `true`.) V kapitole 15 jste se také dozvěděli, že unární funkční objekt má jediný parametr, zatímco binární má parametry dva.

Třídění

Nejdříve prozkoumejme třídicí algoritmy.

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

Funkce `sort()` třídí vzestupně rozsah `[first, last)`, přičemž pro porovnávání používá operátor < pro daný typ hodnoty. První verze určuje pořadí pomocí operátoru <, zatímco druhá používá porovnávací objekt `comp`.

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

Funkce `stable_sort()` třídí rozsah `[first, last)`, přičemž zachovává relativní pořadí ekvivalentních prvků. První verze určuje pořadí pomocí operátoru <, zatímco druhá používá porovnávací objekt `comp`.

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last, Compare comp);
```

Funkce `partial_sort()` třídí částečně rozsah `[first, last)`. První prvky `middle - first` setříděného rozsahu jsou umístěny do rozsahu `[first, middle)`, zatímco zbývající zůstanou nesetříděny. První verze určuje pořadí pomocí operátoru <, zatímco druhá používá porovnávací objekt `comp`.

```
template<class InputIterator class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
                                       InputIterator last,
                                       RandomAccessIterator result_first,
                                       RandomAccessIterator result_last);
template<class InputIterator class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(InputIterator first,
                                       InputIterator last,
                                       RandomAccessIterator result_first,
                                       RandomAccessIterator result_last,
                                       Compare comp);
```

Funkce partial_sort_copy() zkopíruje prvních n prvků setříděného rozsahu [first, last) do rozsahu [result_first, result_first + n). Hodnota n je menší než last - first a result_last - result_first. Funkce vrátí result_first + n. První verze určuje pořadí pomocí operátoru <, zatímco druhá používá porovnávací objekt comp.

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);
```

Funkce nth_element() najde v rozsahu [first, last) prvek, který by se při setříděném rozsahu nacházel na n-té pozici a tento prvek sem umístí. První verze určuje pořadí pomocí operátoru <, zatímco druhá používá porovnávací objekt comp.

Binární hledání

Algoritmy ze skupiny pro binární hledání předpokládají setříděný rozsah. Požadují pouze dopředný iterátor, ale efektivněji fungují s iterátory přímého přístupu.

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);
template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

Funkce lower_bound() najde v setříděném rozsahu [first, last) první pozici, před kterou lze vložit hodnotu value, aniž by přitom došlo k narušení pořadí. Vrací iterátor ukazující na tuto pozici. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt comp.

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);
template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

Funkce upper_bound() najde v setříděném rozsahu [first, last) poslední pozici, před kterou lze vložit hodnotu value, aniž by přitom došlo k narušení pořadí. Vrací iterátor ukazující na tuto pozici. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt comp.

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first,
                 ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first,
                 ForwardIterator last, const T& value, Compare comp);
```

Funce `equal_range()` najde v setříděném rozsahu `[first, last)` takový největší podrozsah `[it1, it2)`, aby bylo možné vložit hodnotu `value` před libovolný iterátor v tomto rozsahu a přitom nedošlo k narušení pořadí. Funkce vrací dvojici tvořenou iterátory it1 a it2. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);
template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

Funkce `binary_search()` vrátí hodnotu true, jestliže se v setříděném rozsahu `[first, last)` najde ekvivalent hodnoty `value`. V opačném případě vrací hodnotu false. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

Poznámka

Vzpomeňte si, že jestliže se pořadí určuje pomocí operátoru `<`, jsou hodnoty `a` a `b` ekvivalentní, jestliže oba výrazy `a < b` i `b < a` vrátí hodnotu `false`. U obyčejných čísel ekvivalence znamená rovnost, ale toto neplatí pro struktury, setříděné pouze podle jedné položky. Může existovat více míst, kam lze novou hodnotu vložit a přitom zachovat pořadí dat. Podobně jestliže se pro učení pořadí použije porovnávací objekt `comp`, znamená ekvivalence, že oba výrazy `comp(a, b)` i `comp(b, a)` vrátí hodnotu `false`. (Je to generalizace tvrzení, že `a` a `b` jsou ekvivalentní, jestliže `a` není menší než `b` a `b` není menší než `a`.)

Slučování

Funkce pro slučování předpokládají setříděné rozsahy.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>.
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

Funkce `merge()` slučuje prvky ze setříděných rozsahů `[first1, last1)` a `[first2, last2)` a výsledek uloží do rozsahu s počátkem na pozici `result`. Cílový rozsah nesmí překrývat žádný ze sloučených rozsahů. Jestliže se v obou rozsazích najdou ekvivalentní prvky, předchází prvky z prvního rozsahu prvky z rozsahu druhého. Návratovou hodnotou je iterátor, ukazující na prvek za posledním prvkem výsledného rozsahu. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator mid-
dle,
                   BidirectionalIterator last);
template<class BidirectionalIterator, class Compare >
void inplace_merge(BidirectionalIterator first, BidirectionalIterator mid-
dle,
                   BidirectionalIterator last, Compare comp);
```

Funkce `inplace_merge()` slučuje dva sousední setříděné rozsahy `[first, middle)` a `[middle, last)` do jediné setříděné sekvence v rozsahu `[first, last)`. Prvky z prvního rozsahu budou předcházet ekvivalentní prvky z rozsahu druhého. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

Množinové operace

Množinové operace pracují se všemi setříděnými sekvencemi včetně `set` a `multiset`. Pro kontejnery obsahující více instancí jedné hodnoty jako je multiset, jsou definice generalizované. Sjednocení dvou kontejnerů `multiset` obsahuje větší počet výskytů každého prvku, zatímco průnik obsahuje menší počet výskytů každého prvku. Předpokládejme například, že kontejner `multiset` A obsahuje řetězec „apple", zatímco kontejner `multiset` B obsahuje čtyři výskyty stejného řetězce. V takovém případě bude sjednocení A a B obsahovat sedm instancí řetězce „apple", zatímco průnik bude obsahovat instance čtyři.

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Funkce `includes()` vrátí hodnotu true, jestliže se každý prvek z rozsahu `[first2, last2)` nachází také v rozsahu `[first1, last1)`. V opačném případě vrátí hodnotu `false`. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
              class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
```

Funkce `set_union()` vytvoří množinu, která je sjednocením rozsahů `[first1, last1)` a `[first2, last2)`, a výsledek zkopíruje na místo, kam ukazuje result. Výsledný rozsah nesmí překrývat žádný z původních rozsahů. Funkce vrací iterátor ukazující na prvek následující bezprostředně za posledním prvkem vytvořeného rozsahu. Sjednocení je množina obsahující všechny prvky nalezené v obou množinách. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1
last1,
          InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1
last1,
          InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
```

Funkce `set_intersection()` vytvoří množinu, která je průnikem rozsahů `[first1, last1)` a `[first2, last2)`, a výsledek zkopíruje na místo, kam ukazuje `result`. Výsledný rozsah nesmí překrývat žádný z původních rozsahů. Funkce vrací iterátor ukazující na prvek následující bezprostředně za posledním prvkem vytvořeného rozsahu. Průnik je množina obsahující všechny prvky, které jsou oběma množinám společné. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
```

Funkce `set_difference()` vytvoří množinu, která je rozdílem rozsahů `[first1, last1)` a `[first2, last2)`, a výsledek zkopíruje na místo, kam ukazuje `result`. Výsledný rozsah nesmí překrývat žádný z původních rozsahů. Funkce vrací iterátor ukazující na prvek následující bezprostředně za posledním prvkem vytvořeného rozsahu. Rozdíl je množina obsahující prvky, které se nacházejí v prvním rozsahu, ale ne v rozsahu druhém. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator set_symmetric_difference(InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
```

Funkce `set_symmetric_difference()` vytvoří množinu, která je symetrickým rozdílem rozsahů `[first1, last1)` a `[first2, last2)`, a výsledek zkopíruje na místo, kam ukazuje result. Výsledný rozsah nesmí překrývat žádný z původních rozsahů. Funkce vrací iterátor ukazující na prvek následující bezprostředně za posledním prvkem vytvořeného rozsahu. Symetrický rozdíl je množina obsahující prvky, které se nacházejí v prvním rozsahu, ale ne v rozsahu druhém, a prvky nacházející se v rozsahu druhém, ale ne v prvním. Je to

totéž jako rozdíl sjednocení a průniku. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt comp.

Operace pracující s haldou

Halda je forma společných dat, která má tu vlastnost, že první prvek v haldě je největší. Kdykoli je první prvek odstraněn nebo je nějaký prvek přidán, musí se halda znovu uspořádat, aby si tuto vlastnost zachovala. Halda je navržena tak, aby tyto dvě operace probíhaly efektivně.

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

Funkce make_heap() vytvoří haldu z rozsahu [first, last). První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt comp.

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

Funkce push_heap() předpokládá platnou haldu v rozsahu [first, last - 1) a přidá do ní hodnotu na pozici last - 1 (tedy na konec předpokládané platné haldy), čímž vytvoří platnou haldu [first, last). První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt comp.

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

Funkce pop_heap() předpokládá platnou haldu v rozsahu [first, last). Hodnotu na pozici last – 1 vymění s hodnotou na pozici first a vytvoří platnou haldu v rozsahu [first, last - 1). První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt comp.

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

Funkce pop_heap() předpokládá platnou haldu v rozsahu [first, last) a setřídí ji. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt comp.

Funkce Minimum a Maximum

Funkce minimum a maximum vracejí minimální a maximální hodnotu z dvojice hodnot a sekvence hodnot.

```
template<class T> const T& min(const T& a, const T& b);
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

Funkce `min()` vrací menší ze dvou hodnot. Jestliže jsou obě hodnoty ekvivalentní, vrátí první hodnotu. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class T> const T& max(const T& a, const T& b);
template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

Funkce `max()` vrací větší ze dvou hodnot. Jestliže jsou obě hodnoty ekvivalentní, vrátí první hodnotu. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

Funkce `min_element()` vrací takový první iterátor it v rozsahu [first, last), aby žádný prvek v rozsahu nebyl menší než hodnota `*it`. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

Funkce `max_element()` vrací takový první iterátor it v rozsahu [first, last), aby žádný prvek v rozsahu nebyl větší než hodnota `*it`. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first, InputIterator2 last2
Compare comp);
```

Funkce `lexicographical_compare()` vrátí hodnotu true, jestliže sekvence prvků v rozsahu `[first1, last1)` je lexikálně menší než sekvence prvků v rozsahu `[first2, last2)`. V opačném případě vátí hodnotu `false`. Lexikální porovnávání porovnává první prvek z jedné sekvence s prvním prvkem ze sekvence druhé, porovnává tedy hodnotu `*first1` s hodnotou `*first2`. Jestliže je hodnota `*first1` menší než `*first2`, vrátí hodnotu true.

Pokud je hodnota `*first2` menší než `*first1`, vrátí hodnotu false. Jsou-li obě hodnoty ekvivalentní, porovná následující prvky z obou sekvencí. Tento proces pokračuje tak dlouho, dokud se nenajdou dva odpovídající prvky, které nejsou ekvivalentní, nebo dokud není dosaženo konce sekvence. Jestliže jsou obě sekvence ekvivalentní až do chvíle, kdy je dosaženo konce jedné z nich, je kratší sekvence menší. Pokud jsou obě sekvence ekvivalentní a mají stejnou délku, není žádná z nich menší a funkce tedy vrátí hodnotu false. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

Změny pořadí

Změna pořadí sekvence znamená změnu pořadí jejich prvků. Například sekvence obsahující tři prvky může mít šest možných pořadí prvků, protože první prvek můžete zvolit ze tří možností. Jakmile se rozhodnete, že určitý prvek bude na první pozici, zůstanou vám dva prvky pro obsazení pozice druhé a jeden pro pozici třetí. Příklad se změnou pořadí číslic 1, 3 a 5 vypadá následovně:

```
123 132 213 231 312 321
```

Obecně platí, že sekvence n prvků má `n * (n-1)*..*` čili n! možných pořadí. Funkce pro změnu pořadí předpokládají, že množinu všech možných pořadí lze uspořádat v lexikálním pořadí jako ve výše uvedeném příkladu. To obecně znamená, že existuje specifické pořadí, které všechny předchází a následuje. Avšak první pořadí (v příkladu 123) nemá žádného předchůdce a poslední pořadí (321) nemá žádného následníka.

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last,
Compare comp);
```

Funkce `next_permutation()` změní pořadí v sekvenci určené rozsahem `[first, last)` na pořadí, které lexikálně následuje. Jestliže následující pořadí existuje, vrátí funkce hodnotu `true`. Pokud neexistuje (to znamená, že rozsah obsahuje lexikálně poslední pořadí), vrátí funkce hodnotu false a změní rozsah na lexikálně první pořadí. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last,
Compare comp);
```

Funkce `prev_permutation()` změní pořadí v sekvenci určené rozsahem `[first, last)` na pořadí, které lexikálně předchází. Jestliže předcházející pořadí existuje, vrátí funkce hodnotu true. Pokud neexistuje (to znamená, že rozsah obsahuje lexikálně první pořadí), vrátí funkce hodnotu `false` a změní rozsah na lexikálně poslední pořadí. První verze používá pro určení pořadí operátor <, zatímco druhá porovnávací objekt `comp`.

Numerické operace

V tabulce G.11 jsou shrnuty numerické operace, které jsou popsány v hlavičkovém souboru `numeric`. Parametry nejsou uvedeny a přetížené funkce jsou v seznamu pouze jednou. Každá funkce má verzi používající pro řazení prvků operátor < a verzi, která prvky řadí pomocí porovnávacího funkčního objektu. Podrobnější popis včetně prototypů následuje za tabulkou. Prohlédnutím tabulky tedy získáte představu o tom, co která funkce dělá, a pokud vás některá zaujme, můžete si vyhledat podrobnosti.

Tabulka G.11 Numerické operace.

Funkce	Popis
accumulate()	Vypočítá celkový součet hodnot v rozsahu.
inner_product()	Vypočítá vnitřní produkt dvou rozsahů.
partial_sum()	Kopíruje částečné součty vypočítané z jednoho rozsahu do rozsahu druhého.
adjacent_difference	Kopíruje sousední rozdíly vypočítané z prvků v jednom rozsahu do rozsahu druhého.

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);
template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
BinaryOperation binary_op);
```

Funkce `accumulate()` inicializuje hodnotu `acc` na hodnotu `init`. Potom provede operaci `acc = acc + *i` (první verze) nebo `acc = binary_op(acc, *i)` (druhá verze) pro každý iterátor i v rozsahu `[first, last)`. Výslednou hodnotu `acc` vrátí.

```
template<class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, T init);
template<class InputIterator1, class InputIterator2, class T,
class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, T init,
          BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

Funkce `inner_product()` inicializuje hodnotu `acc` na hodnotu `init`. Potom provede operaci `acc = *i * *j` (první verze) nebo `acc = binary_op(*i, *j)` (druhá verze) pro každý iterátor `i` v rozsahu `[first1, last1)` a každý odpovídající iterátor `j` v rozsahu `[first2, first2 + (last1 - first1))`. To znamená, že vypočítá hodnotu z prvních prvků každé sekvence, potom hodnotu z druhých prvků každé sekvence a tak dále, dokud nedosáhne konce první sekvence. (Druhá sekvence tedy musí být alespoň tak dlouhá jako sekvence první.) Výslednou hodnotu acc vrátí.

```
template<class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
OutputIterator result);
```

```
template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
OutputIterator result, BinaryOperation binary_op);
```

Funkce `partial_sum()` přiřadí hodnotu `*first1` do hodnoty `*result`, hodnotu `*first + *(first + 1)` do hodnoty `*(result + 1)` (první verze), nebo `binary_op(first, *first + *(first + 1))` do `*(result + 1)` (druhá verze) a tak dále. To znamená, že *n-tý* prvek sekvence s počátkem na pozici `result` obsahuje součet (nebo ekvivalent `binary_op`) prvních *n* prvků sekvence s počátkem na pozici `first`. Funkce vrací iterátor ukazující na prvek za posledním prvkem sekvence `result`. Algoritmus umožňuje, aby `result` byl `first`, to znamená že `result` může být v případě potřeby zkopírován do původní sekvence.

```
template<class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
OutputIterator result, BinaryOperation binary_op);
```

Funkce `adjacent_difference()` přiřadí hodnotu `*first` na pozici `result(*result = *first)`. Na následující pozice v cílovém rozsahu jsou přiřazeny rozdíly (nebo ekvivalent `binary_op`) sousedních hodnot ze zdrojového rozsahu. To znamená, že na následující pozici v cílovém rozsahu `(result + 1)` je přiřazena hodnota `*(first + 1) - *first` (první verze), nebo `binary_op(*(first + 1), *first)` (druhá verze) a tak dále. Funkce vrací iterátor ukazující na prvek za posledním prvkem sekvence `result`. Algoritmus umožňuje, aby result byl `first`, to znamená že result může být v případě potřeby zkopírován do původní sekvence.

DODATEK H

Vybraná literatura

- Booch, Grady. *Object-Oriented Analysis and Design.* Druhé vydání. Redwood City, CA: Benjamin/Cummings, 1994.

  Tato kniha představuje koncepty OOP, rozebírá metody a obsahuje ukázkové aplikace. Příklady jsou v C++.

- Booch, Grady, Jim Rumbaugh a Ivar Jacobson. *Unified Modeling Language User Guide.* Reading, MA: Addison-Wesley, 1998.

  Tato kniha od tvůrců jazyka Unified Modeling Language představuje jádro UML a mnoho příkladů jeho použití.

- Ellis, Margaret A., a Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Reading, MA: Addison-Wesley, 1990.

  Tato kniha, obvykle nazývaná ARM, slouží jako základní dokumentace výboru pro standard ANSI/ISO C++. Nejedná se o knihu pro učení jazyka, ale jsou zde odpovědi na většinu technických otázek týkajících se fungování jazyka. Kniha nepokrývá vše, co výbor pro ANSI/ISO přidal. Její další vydání však bude brzy k dispozici.

- Jacobson, Ivar. *Object-Oriented Software Engineering*: A Use Case Driven Approach. Reading, MA: Addison-Wesley, 1994.

  Tato kniha popisuje úspěšná vodítka a metody (Object-Oriented Software Engineering, OOSE) pro vývoj softarových systémů ve velkém.

- Meyers, Scott. Effective C++: 50 Ways to Improve Your Programs and Designs. Druhé vydání. Reading, MA: Addison-Wesley, 1998.

  Kniha je určena programátorům, kteří již C++ znají. Nabízí 50 pravidel a vodítek. Některá jsou technická, například vysvětlení, kdy definovat kopírovací konstruktor a operátory přiřazení. Jiná jsou obecnější, například vysvětlení týkající se vztahů je a má.

- Meyers, Scott. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Reading, MA: Addison-Wesley, 1996.

  Tato kniha pokračuje v tradici *Effective C++*, objasňuje některé málo srozumitelné aspekty jazyka a ukazuje, jak dosáhnout různých cílů, například návrhu inteligentních ukazatelů. Odráží i zkušenosti, které programátoři v C++ získali v několika posledních letech.

- Murray, Robert B. C++ *Strategies and Tactics.* Reading, MA: Addison-Wesley, 1993.

  Kniha má pomoci novým a mírně pokročilým programátorům v C++ naučit se používat jazyk efektivně. Rozebírá třídy, dědičnost, šablony, výjimky a několik dalších témat, nabízí praktické rady a popisuje obvyklé techniky.

- Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen, William Lorenson. *Object-Oriented Modeling and Design*. Englewood Cliffs, Prentice Hall, 1991.

  Kniha představuje a zkoumá objektově modelovou techniku (OMT), metodu rozkládající problémy do vhodných objektů.

- Rumbaugh, James, Ivar Jacobson a Grady Booch. *Unified Modeling Reference Manual*. Reading, MA: Addison-Wesley, 1998.

  Tato kniha od tvůrců jazyka Unified Modeling Language představuje úplný popis jazyka UML ve formátu příručky.

- Stroustrup, Bjarne. *The C++ Programming Language*. Třetí vydání. Reading, MA: Addison-Wesley, 1997.

  Stroustrup C++ vytvořil, takže se jedná o autoritativní text. Snadněji ho však strávíte, pokud již nějaké znalosti o C++ máte. Nejedná se jen o popis jazyka, ale je také uvedeno mnoho příkladů týkajících se použití a rozebírána je i metodologie OOP. Další vydání této knihy rostly společně s jazykem a toto vydání obsahuje rozbor prvků standardní knihovny, jako je knihovna STL a řetězce.

- Stroustrup, Bjarne. The Design and Evolution of C++. Reading, MA: Addison-Wesley, 1994.

  Pokud se zajímáte, jak se C++ vyvíjí a proč právě tímto způsobem, přečtěte si tuto knihu.

- Standard ISO/ANSI.

  Konečný standard bude k dispozici na následujících adresách. Cena v čase psaní není známa ale Commitee Draft 2 (CD2) se prodával za 50 dolarů.

  American National Standards Institute
11 W. 42nd St.
New York, NY 10036
Global Engineering Documents, Inc.
Inverness Way East
15 Englewood, CO 80122-5704

  Kopie dřívějšího CD2 lze stále stáhnout ve formátech ASCII, PostScript, HTML a Adobe Acorbat PDF z následujících dvou stránek WWW:

  http://www.setech.com/x3.html

  http://www.maths.warwick.ac.uk/c++/pub/

  Následující stránka FAQ (Frequently Asked Questions) obsahuje aktuálnější informace:

  http://reality.sgi.com/austern/std-c++/faq.html

DODATEK I

Konverze na ANSI/ISO standard C++

Můžete mít programy (nebo programovací zvyky) vytvořené v jazyce C nebo ve starších verzích C++, které byste chtěli převést do standardu C++. Tato příloha nabízí několik vodítek. Některá se týkají konverze z C do C++, jiná ze starších verzí C++ do standardu.

Direktivy preprocesoru

Preprocesor C/C++ obsahuje pole direktiv. Pro praxi C++ obecně platí, že tyto direktivy máte používat pro správu kompilačního procesu a vyhýbat se jim tam, kde by měly nahrazovat kód. Například direktiva `#include` je důležitá komponenta pro správu souborů programu. Jiné direktivy, například `#ifndef` a `#endif` umožňují určit, zda budou určité bloky kódu zkompilovány. Direktiva `#pragma` umožňuje ovládat kompilační možnosti. Všechno jsou to užitečné a někdy nutné nástroje. Měli byste se však mít na pozoru, pokud jde o direktivu `#define`.

Definujte konstanty pomocí modifikátoru `const` místo direktivy `#define`

Symbolické konstanty činí kód čitelnějším a lépe udržovatelným. Název konstanty naznačuje její význam a když budete potřebovat hodnotu změnit, bude stačit učinit tak pouze jednou v definici a potom kód znovu zkompilovat. Jazyk C používal k tomuto účelu preprocesor:

```
#define MAX_LENGTH 100
```

Preprocesor potom ještě před kompilací nahradil ve zdrojovém kódu všechny výskyty `MAX_LENGTH` hodnotou 100.

C++ řeší deklaraci proměnné použitím modifikátoru const:

```
const int MAX_LENGTH = 100;
```

MAX_LENGTH bude jako proměnná typu int určená pouze pro čtení.

Řešení s použitím modifikátoru const má několik výhod. Za prvé, deklarace explicitně pojmenovává typ. U direktivy #define musíte použít pro číslo různé přípony, abyste označili jiný typ než char, int nebo double; například pomocí 100L označíte typ long nebo zápisem 3.14F typ float. Důležitější však je, že řešení s modifikátorem const lze stejně dobře použít u odvozených typů:

```
const int base_vals[5] = {1000, 2000, 3500, 6000, 10000};
const string ans[3] = {"ano", "ne", "snad"};
```

Konečně idetifikátory const se řídí stejnými pravidly o rozsahu platnosti jako proměnné. Můžete tedy vytvořit konstanty s globální platností a pojmenovat je namespace, a také konstanty s platností v bloku. Jestliže například definujete konstantu v určité funkci, nemusíte se strachovat, že dojde ke konfliktu s globální konstantou použitou jinde v programu. Uvažujte například následující kód:

```
#define n 5
const int dz = 12;
...
void fizzle()
{
  int n;
  int dz;
}
```

Preprocesor nahradí definici

```
int n;
definici
int 5;
```

a dojde ke kompilační chybě. Proměnná dz však bude lokální proměnnou ve funkci fizzle(). Funkce fizzle() může v případě nutnosti také použít rozlišovací operátor a přistupovat ke konstantě jako ::dz.

Jazyk C si vypůjčily klíčové slovo const z C++, ale verze v C++ je užitečnější. Na externí konstantní hodnoty má například spíše vnitřní než vnější vazbu, kterou používají proměnné a konstanty jazyka C. To znamená, že každý soubor v programu používající konstantu, potřebuje, aby tato konstanta byla definována v určitém souboru. Vypadá to jako práce navíc, ale ve skutečnosti to usnadňuje život. U vnitřní vazby můžete definice konstant umístit do hlavičkového souboru, který používají různé soubory v projektu. Vnější vazba by způsobila chybu kompilátoru, ale vnitřní ne. Vzhledem k tomu, že konstanta také musí být definována v souboru, který ji používá (v hlavičkovém souboru), můžete konstantní hodnoty používat jako parametry velikosti polí:

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
for (int i = 0; i < MAX_LENGTH; i++)
  loads[i] = 50;
```

V jazyce C takový kód fungovat nebude, protože definující deklarace MAX_LENGTH by mohla být při kompilování tohoto konkrétního souboru nedostupná v odděleném souboru. Slušelo by se dodat, že v jazyce C byste mohli vytvořit konstanty s vnitřní vazbou pomocí modifikátoru static. V C++ je však toto implicitní a proto si musíte pamatovat o jednu věc méně.

Direktiva #define je však přesto užitečnou součástí standardního idiomu, který určuje, kdy se bude kompilovat hlavičkový soubor:

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// zde bude kód
#endif
```

U typických symbolických konstant si však zvykněte používat modifikátor const místo direktivy #define. Jinou dobrou alternativou je použít výčet enum, zvláště když máte množinu příbuzných celočíselných konstant:

```
enum {LEVEL1 = 1, LEVEL2 = 2, LEVEL3 = 4, LEVEL4 = 8};
```

Definujte krátké funkce pomocí klíčového slova inline místo direktivy #define

Tradičním způsobem jazyka C pro vytvoření ekvivalentu blízkého vložené funkci byla definice makra pomocí direktivy #define:

```
#define Cube(X) X*X*X
```

To vede k tomu, že preprocesor v textu nahradí X parametrem funkce Cube():

```
y = Cube(x);          // nahradí výrazem y = x*x*x;
y = Cube(x + z++);    // nahradí y výrazem x + z++*x + z++*x + z++
```

Vzhledem k tomu, že preprocesor používá nahrazení v textu místo skutečného předání parametrů, může používání takových maker vést k neočekávaným a nesprávným výsledkům. Snížit pravděbodobnost takové chyby můžete pomocí množství závorek, které zajistí správné pořadí operací:

```
#define Cube(X) ((X)*(X)(*X))
```

Ani tento způsob však nedokáže ošetřit takové způsoby, jako je použití hodnot z++.

Řešení jazyka C++, který identifikuje vložené funkce pomocí klíčového slova inline, je mnohem spolehlivější, protože se jedná o skutečné předání parametrů. Navíc mohou být vložené funkce běžnými funkcemi nebo metodami třídy.

Jednou z pozitivních stránek makra vytvořeného pomocí direktivy #define je to, že není žádného typu a může tedy být použito s libovolným typem, pro nějž má operace smysl. V C++ můžete dosáhnout typově nezávislých funkcí vytvořením vložených šablon a přitom si ponecháte možnost předávat parametry.

Stručně řečeno, používejte vložené funkce a šablony místo maker jazyka C vytvořených pomocí direktivy #define.

Používání funkčních prototypů

Vlastně nemáte na výběr. Zatímco v jazyce C je používání prototypů volitelné, v C++ je povinné. Všimněte si, že funkce, která je definována před prvním použitím jako funkce vložená, slouží jako svůj vlastní prototyp.

Kde se to hodí, používejte ve funkčních prototypech a hlavičkách modifikátor `const`. Obzvláště ho používejte u parametrů typu ukazatel nebo reference reprezentující data, která se nebudou měnit. Tímto způsobem nejen umožníte kompilátoru zachytit chyby související se změnou dat, ale funkce bude také obecnější. To znamená, že funkce používající konstantní ukazatel nebo konstantní referenci může zpracovávat jak konstantní, tak i nekonstantní data, zatímco funkce, která modifikátor `const` nepoužívá, může zpracovávat pouze data nekonstantní.

Přetypování

Jednou z věcí, která Stroustrupa na jazyku C otravuje, je nedisciplinovanost operátoru přetypování. Přetypování je samozřejmě často nezbytné, ale standardní přetypování je příliš volné. Uvažujte například následující kód:

```
struct Doof
{
   double feeb;
   double steeb;
   char sgif[10];
};
Doof leam;
short * ps = (short *) & leam;    // původní syntaxe
int * pi = int * (&leam);         // nová syntaxe
```

Nic vám v jazyce nezabrání přetypovat ukazatel jednoho typu na ukazatel na typ zcela nepříbuzný.

Do určité míry je situace podobná jako u příkazu goto. Problém s příkazem `goto` byl, že jeho pružnost vede k zamotanému kódu. Řešením bylo nabídnout omezenější, strukturovanou verzi příkazu `goto`, která by zvládla většinu obvyklých úkolů, pro které byl příkaz `goto` potřeba. To vedlo ke vzniku takových prvků jazyka, jako jsou příkazy cyklu `for` a `while` a příkazy `if else`. Standardní C++ nabízí podobné řešení problému nedisciplinovaného přetypování, konkrétně omezené přetypování, které zvládne nejběžnější situace, při kterých je přetypování nutné. Následující operátory přetypování byly probírány v kapitole 14:

```
dynamic_cast
static_cast
const_cast
reinterpret_cast
```

Jestliže tedy přetypováváte ukazatele, použijte pokud možno jeden z těchto operátorů. Tímto způsobem dosáhnete zamýšleného přetypování i jeho kontroly.

Seznamte se s vlastnostmi C++

Jestliže jste používali funkce `malloc()` a `free()`, začněte místo nich používat operátory `new` a `delete`. Pokud jste pro ošetření chyb používali funkce `setjmp()` a `longjmp()`, začněte místo nich používat `try`, `throw` a `catch`. Pro hodnoty reprezentující `true` a `false` se snažte používat typ `bool`.

Používejte nové uspořádání hlavičkových souborů

Standard specifikuje nové názvy pro hlavičkové soubory, které byly popsány v kapitole 2. Jestliže jste používali hlavičkové soubory ve starém stylu, měli byste přejít na používání názvů v novém stylu. Nejedná se o změnu kosmetickou, protože nové verze mohou mít nové vlastnosti. Například hlavičkový soubor `ostream` podporuje pro vstup a výstup široké znaky. Poskytuje také nové manipulátory jako `boolapha` a `fixed` (byly popsány v kapitole 16). Ty nabízí pro nastavení mnoha formátovacích možností jednodušší rozhraní než funkce `setf()` nebo `iomanip`. Pokud přesto používáte funkci `setf()`, používejte při specifikaci konstant `ios_base` místo `ios`, tedy `ios_base::fixed` místo `ios::fixed`. Nové hlavičkové soubory také začleňují prostory jmen.

Používejte prostory jmen

Prostory jmen pomáhají organizovat identifikátory používané v programu a vyhnout se tak konfliktu názvů. Vzhledem k tomu, že standardní knihovna implementovaná pomocí nového uspořádání hlavičkových souborů vkládá názvy do prostoru jmen `std`, musíte při používání těchto hlavičkových souborů použít prostory jmen.

Příklady v této knize používají pro přístup ke všem názvům z prostoru jmen `std` z důvodu jednoduchosti direktivu `using`:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;        // direktiva using
```

Avšak hromadný export všech názvů z prostoru jmen, ať už jsou potřeba nebo ne, je v protikladu s cíli prostoru jmen.

Místo toho dopručujeme používat buď deklaraci pomocí `using`, nebo operátor rozlišení(::), a zpřístupnit pouze ty názvy, které program bude potřebovat. Například kód

```
#include <iostream>
using std::cin;                 // deklarace using
using std::cout;
using std::endl;
```

zpřístupňuje objekty `cin`, `cout` a `endl` pro zbytek souboru. Použitím operátoru rozlišení je však zpřístupníte pouze ve výrazu, který operátor používá:

```
cout << std::fixed << x << endl; // použití operátoru rozlišení
```

To by mohlo být únavné, ale běžné deklarace using můžete shromáždit do hlavičkového souboru:

```
// mynames - hlavičkový soubor
#include <iostream>
using std::cin;                 // deklarace using
using std::cout;
using std::endl;
```

Půjdeme-li o krok dále, můžete deklarace `using` shromáždit do hlavičkového souboru:

```
// mynames - hlavičkový soubor
#include <iostream>
namespace io
{
using std::cin;
using std::cout;
using std::endl;
}
namespace formats
{
  using std::fixed;
  using std::scientific;
  using std::boolalpha;
}
```

Potom můžete tento soubor vložit do programu a potřené prostory jmen používat :

```
#include "mynames"
using namespace io;
```

Používejte šablonu autoptr

Každé použití operátoru `new` musí být doplněno použitím operátoru `delete`. To může přivodit problémy, jestliže funkce, v níž jste operátor `new` použili, skončí předčasně v důsledku vyvolání výjimky. Jak jsme probírali v kapitole 15, použijete-li objekt `autoptr` zaznámenávající vytvoření objektů pomocí operátoru `new`, bude operátor `delete` aktivován automaticky.

Používejte třídu `string`

Tradiční řetězec ve stylu jazyka C trpí tím, že není skutečným typem. Řetězec můžete uložit do pole znaků a pole znaků můžete inicializovat pomocí řetězce. Přiřadit řetězec poli znaků pomocí operátoru přiřazení však nemůžete; místo toho musíte používat funkce `strcpy()` nebo `strncpy()`. Řetězce jazyka C nemůžete porovnávat pomocí relačních operátorů; místo toho musíte použít funkci `strcmp()`. (A jestliže se zapomenete a použije například operátor >, překladač syntaktickou chybu neohlásí. Program však bude místo obsahů řetězců porovnávat jejich adresy.)

Třída `string` (viz kapitola 15 a příloha F) vám naproti tomu umožňuje vyjádřit řetězce pomocí objektů. Definovány jsou všechny operátory přiřazení, relační operátory a operátor sčítání (pro spojování řetězců). Navíc třída `string` disponuje automatickou správou paměti, takže se normálně nemusíte obávat, že někdo zadá řetězec přesahující pole nebo že bude tento řetězec ještě před uložením zkrácen.

Třída `string` má mnoho metod pro usnadnění práce. Můžete například připojit jeden objekt třídy `string` k jinému, ale k objektu třídy `string` můžete také připojit řetězec ve stylu jazyka C nebo i hodnotu typu `char`. U funkcí, které požadují jako parametr řetězec ve stylu jazyka C, můžete použít metodu `c_str()`, která vrátí příslušný ukazatel na typ `char`.

Třída `string` nenabízí pouze propracovanou množinu metod pro úkoly související s řetězci, jako například hledání podřetězců, ale je také navržena tak, aby byla kompatibilní s knihovnou STL a mohli jste tedy při práci s jejími objekty využívat algoritmů z knihovny STL.

Používejte knihovnu STL

Standardní knihovna šablon (Standard Template Library, STL – viz kapitola 15 a příloha G) nabízí hotová řešení pro mnoho programovacích potřeb. Používejte ji tedy. Například místo deklarování pole typu `double` nebo objektů třídy `string` můžete vytvořit objekt `vector<double>` nebo `vector<string>`. Má to podobné výhody jako používání objektů třídy `string` namísto řetězců ve stylu jazyka C. Operátor přiřazení je definován, takže jeden objekt `vector` můžete přiřadit jinému. Objekt `vector` můžete předávat odkazem a funkce, která jej obdrží, může pomocí metody `size()` určit počet prvků, které objekt `vector` obsahuje. Vestavěná správa paměti umožňuje automatickou změnu velikosti objektu, jestliže do něho přidáte prvky pomocí metody `pushback()`. K vašim službám je samozřejmě několik užitečných metod tříd a všeobecných algoritmů.

Jestliže potřebujete seznam (list), obousměrnou frontu (double-ended queue, deque), zásobník, obyčejnou frontu, množinu nebo mapu, nabízí knihovna STL užitečné šablony kontejnerů. Algoritmus knihovny je navržen tak, abyste snadno mohli zkopírovat obsah vektoru do seznamu nebo porovnat obsahy množiny a vektoru. Tento návrh dělá z knihovny STL sadu nástrojů nabízející součástky, které můžete podle potřeby montovat dohromady.

Jedním z hlavních cílů návrhu rozsáhlých algoritmů knihovny je efektivnost, takže s poměrně malým programátorským úsilím můžete dosáhnout bleskových výsledků. A koncept iterátorů použitý při implementaci algoritmů znamená, že nejsou omezeny na použití s kontejnery knihovny STL. Konkrétně je lze také použít na tradiční pole.

DODATEK J

Odpovědi na opakovací otázky

Kapitola 2

1. Nazývají se funkce.
2. Direktiva způsobí, že obsah souboru `iostream` ještě před dokončením kompilace tuto direktivu nahradí.
3. Zpřístupní definice vytvořené v prostoru jmen `std` programu.
4. `cout << "Hello, world\n";`
5. `int cheeses;`
6. `cheeses = 32;`
7. `cin >> cheeses;`
8. `cout << "Mame " << cheeses << " druhu syra\n";`
9. Informuje nás, že funkce `froop()` očekává, že bude volána s jedním parametrem typu `double` a že jejím návratovým typem bude hodnota typu `int`.
10. Klíčové slovo `return` se nepoužívá ve funkci, která má návratový typ `void`.

Kapitola 3

1. Více celočíselných typů umožňuje vybrat si ten, který nejvíce vyhovuje určité potřebě. Například použitím typu `short` ušetříte místo, zatímco typ `long` zaručí dostatečnou kapacitu. Můžete také zjistit, že použitím určitého typu urychlíte určitý výpočet.
2. ```
   short rbis = 80;          // nebo short int rbis = 80;
   unsigned int q = 42110;   // nebo unsigned q = 42110;
   unsigned long ants = 3000000000;
   ```

Poznámka

Nespoléhejte se na to, že pro uložení čísla 3000000000 je typ `int` dostatečně velký.

---

3. C++ neposkytuje žádné záruky, že nedojde k překročení rozsahu čísla.
   ```

4. Konstanta `33L` je typu `long`, zatímco konstanta `33` je typu `int`.
5. Ačkoli mají tyto dva příkazy na některých systémech stejný účinek, nejsou ve skutečnosti ekvivalentní. Především první příkaz přiřazuje písmeno `A` proměnné `grade` pouze v systému používajícím kód ASCII, zatímco druhý příkaz funguje i pro jiné kódy. Za druhé, `65` je konstanta typu `int`, zatímco `'A'` je konstanta typu `char`.
6. Zde jsou čtyři způsoby:

```
char c = 88;
cout << c << "\n";           // typ char se vytiskne jako znak
cout.put(char(88));          // funkce put() vytiskne typ char jako znak
cout << char(88) << "\n"; // nový styl přetypování hodnoty na typ char
cout << (char)88 << "\n"; // původní styl přetypování hodnoty na typ char
```

7. Odpověď závisí na velikosti obou typů. Jestliže je typ `long` velký 4 bajty, nic se neztratí. Největší hodnota uložená do typu `long` se pohybuje kolem 2 miliard, což je deset číslic. Vzhledem k tomu, že typ `double` disponuje alespoň 15 platnými číslicemi, nebude zaokrouhlování potřeba.

   a. 8 * 9 + 2 je 72 + 2 je 74
   b. 6 * 3 / 4 je 18 / 4 je 4
   c. 3 / 4 * 6 je 0 * 6 je 0
   d. 6.0 * 3 / 4 je 18.0 / 4 je 4.5
   e. 15 % 4 je 3
8. Oba následující příkazy budou fungovat:

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

Kapitola 4

1. a. `char actors[30];`
   b. `short betsie[100];`
   c. `float chuck[13];`
   d. `long double dipsea[64]`
2. `int oddly[5] = {1, 3, 5, 7, 9};`
3. `int even = oddly[0] + oddly[4];`
4. `cout << ideas[1] << "\n"; // nebo << endl;`
5. `char lunch[13] = "cheeseburger"; // počet znaků + 1`

   nebo

   `char lunch[] = "cheeseburger";    // počet prvků vypočítá kompilátor`
6.
```
struct fish {
    char kind[20];
    int weight;
    float length;
};
```
7.
```
fish petes =
{
```

```
    "pstruh",
    13,
    12.25
};
```

8. `enum Response {Ne, Ano, Snad};`
9. ```
   double * pd = &ted;
   cout << *pd << "\n";
   ```
10. ```
    float * pf = treacle;               // nebo = &treacle[0];
    cout << pf[0] << " " << pf[9] << "\n";    // nebo použijte *pf  a *(pf + 9)
    ```
11. ```
    unsigned int size;
    cout << "Zadejte kladne cislo: ";
    cin >> size;
    int * dyn = new int[size];
    ```
12. Ano, kód je v pořádku. Výraz `"Home of the jolly bytes"` je řetězcová konstanta a vyhodnotí se tedy jako adresa začátku řetězce. Objekt `cout` interpretuje adresu znaku jako pozvání k tisku řetězce, ale přetypování `(int *)` adresu změní na typ ukazatel na `int`, který se potom vytiskne jako adresa. Stručně řečeno, příkaz vytiskne adresu řetězce.
13. ```
    struct fish {
        char kind[20];
        int weight;
        float length;
    };
    fish * pole = new fish;
    cout << "Zadejte druh ryby: ";
    cin >> pole->kind;
    ```
14. Výraz `cin >> address` způsobí, že program bude přeskakovat bílé znaky, dokud nenajde znak nebílý. Potom čte znaky, dokud opět nenarazí na bílý znak. Přeskočí tedy znak nového řádku za číselným vstupem a tak uvedený problém obejde. Přečte však jediné slovo a ne celý řádek.

Kapitola 5

1. Podmínka na začátku cyklu vyhodnotí výraz před vstupem do těla cyklu. Jestliže je neplatná, příkazy v těle cyklu se neprovedou ani jednou. Podmínka na konci cyklu výraz vyhodnotí po provedení příkazů v těle cyklu. Cyklus tedy alespoň jednou proběhne, i když podmínka je od začátku neplatná. Cykly `for` a `while` vyhodnocují podmínku před vstupem do těla cyklu, zatímco cyklus `do while` až po provedení příkazů v těle cyklu.
2. Vytisknou se následující číslice:

   ```
   01234
   ```

   Všimněte si, že výraz `cout << "\n";` není součástí těla cyklu (není v závorkách).
3. Vytisknou se následující číslice:

   ```
   0369
   12
   ```

4. Vytisknou se následující číslice:

```
6
8
```

5. Vytiskne se výraz

```
k = 8
```

6. Nejjednodušší je použít operátor *=:

```
for (int num = 1; num <= 64; num *= 2)
    cout << num << " ";
```

7. Příkazy uzavřete do složených závorek a vytvoříte složený příkaz neboli blok.
8. Ano, první příkaz je v pořádku. Výraz 1,024 se skládá ze dvou výrazů – 1 a 024 – spojených operátorem čárky. Hodnotou je hodnota výrazu vpravo, tedy 024, což v osmičkové soustavě vyjadřuje číslo 20. Deklarace tedy přiřadí do proměnné x hodnotu 20. Druhý příkaz je také správný, avšak na základě priority operátorů bude vyhodnocen následovně:

```
(y = 1), 024;
```

   To znamená, že výraz vlevo nastaví proměnnou y na hodnotu 1, a hodnota celého výrazu, která se nepoužije, je 024 čili 20.
9. Forma `cin >> ch` přeskočí nalezené mezery, znaky nového řádku a tabulátory. Ostatní dva tvary tyto znaky přečtou.

Kapitola 6

1. Obě verze přinesou stejný výsledek, ale verze s `if else` je efektivnější. Uvažujte, co stane, jestliže proměnná `ch` bude mezera. První verze mezery inkrementuje a otestuje, zda znak představuje nový řádek. To je ztráta času, protože program se již ujistil, že znak je mezera a nemůže to tedy být nový řádek. Druhá verze ve stejné situaci test na znak nového řádku přeskočí.
2. Oba výrazy `++ch` i `ch + 1` mají stejnou numerickou hodnotu. Ale `++ch` je typu `char` a vytiskne se jako znak, zatímco výraz `ch + 1`, protože přičítá `char` do typu `int`, je typu `int` a vytiskne se jako číslo.
3. Vzhledem k tomu, že program používá výraz `ch = '$'` místo výrazu `ch == '$'`, bude kombinovaný vstup a výstup vypadat takto:

```
Hi!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

   Než se každý znak podruhé vytiskne, převede se na znak `$`. Hodnotou výrazu `ch = $` je kód znaku `$`, proto je výraz nenulový a má hodnotu `true` a proměnná `ct2` se pokaždé inkrementuje.

4. a. `weight >= 115 && weight < 125`

   b. `ch == 'q' || ch == 'Q'`

   c. `x % 2 == 0 && x != 26`

   d. `donation >= 1000 && donation <= 2000 || guest == 1`

   e. `(ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')`

5. Nemusí tomu tak být. Jestliže je `x` například `10`, pak `!x` je `0` a `!!x` je `1`. Pokud je však `x` proměnná typu `bool`, pak `!!x` je `x`.

6.
```
(x < 0) ? -x : x
switch(ch)
{
  case 'A': a_grade++;
            break;
  case 'B': b_grade++;
            break;
  case 'C': c_grade++;
            break;
  case 'D': d_grade++;
            break;
  default: f_grade++;
            break;
}
```

7. Jestliže použijete číselná označení a uživatel nezadá číslo, ale například znak `q`, program bude čekat, protože číselný vstup nedokáže znak zpracovat. Pokud však použijete označení znaková a uživatel zadá číslo (například `5`), znakový vstup zpracuje `5` jako znak. V takovém případě může část default příkazu switch požádat o zadání dalšího znaku.

8. Zde je jedna z verzí:

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
  if (ch == '\n')
  line++;
}
```

Kapitola 7

1. Těmito třemi kroky jsou definování funkce, dodání prototypu a volání funkce.

2. a. `void igor(void);`

   b. `float tofu(int n);                // nebo float tofu(int);`

   c. `double mpg(double miles, double gallons);`

   d. `long summation(long harray[], int size);`

   e. `double doctor(const char * str);`

   f. `void ofcourse(boss dude);`

   g. `char * plot(map *pmap);`

3.
```
void set_array(int arr[], int size, int value)
{
   for (int i = 0; i < size; i++)
          arr[i] = value;
}
```

4.
```
double biggest (const double foot[], int size)
{
double max;
if (size < 1)
{
          cout << "Velikost pole nemuze byt " << size << ".\n";
          cout << "Navratova hodnota 0\n";
          return 0;
}
else      // není nutné, protože příkaz return program ukončí
{
          max = foot[0];
          for (int i = 1; i < size; i++)
                 if (foot[i] > max)
                        max = foot[i];
                 return max;
}
}
```

5. Kvalifikátor `const` se u ukazatelů používá z toho důvodu, aby data, na která původně ukazují, nemohla být změněna. Jestliže program předává nějaký základní typ, jako například `int` nebo `double`, předává ho hodnotou, takže funkce pracuje s kopií a původní data jsou chráněna.

6.
```
int fill_array(double ar[], int limit)
{
          double temp;
          for (int i = 0; i < limit; i++)
          {
                 cout << "Zadejte hodnotu #" << ": ";
                 if (!(cin >> temp))               // nenumerický vstup
                 {
                        cin.clear();               // nuluje vstup
                        while (cin.get() != '\n')
                               continue;           // zbaví se předchozího
vstupu
                        break;
                 }
                 ar[i] = temp;
          }
}
```

7. Řetězec lze uložit do pole typu `char`, lze ho vyjádřit řetězcovou konstantou uzavřenou v uvozovkách a také ho lze vyjádřit ukazatelem, ukazujícím na první znak řetězce.

8.
```
int replace(char * str, char c1, char c2)
{
        int count = 0;
        while (*str)            // dokud nenarazí na konec řetězce
        {
                if (*str1 == c1)
                {
                        *str2 = c2;
                        count++;
                }
                str++;          // přejde na další znak
        }
        return count;
}
```

9. Vzhledem k tomu, že jazyk C++ interpretuje řetězec „`pizza`“ jako adresu prvního prvku, získáte pomocí operátoru * hodnotu tohoto prvního prvku, což je znak `p`. Protože jazyk C++ interpretuje řetězec „taco“ jako adresu prvního prvku, bude výraz „`taco`“`[2]` interpretovat jako hodnotu prvku o dvě pozice za kurzorem, tedy jako znak c. Jinými slovy, řetězcová konstanta se chová stejně jako název pole.

10. Budete-li chtít strukturu předat hodnotou, předáte její název `glitz`. Jestliže budete chtít předat její adresu, použijete k tomu adresový operátor `&glitz`. Při předávání hodnotou jsou původní data automaticky chráněna, ale stojí to čas a paměť. Předání adresy šetří čas i paměť, ale nechrání původní data, pokud nepoužijete v parametru funkce modifikátor `const`. Při předávání hodnotou také můžete použít obyčejný zápis se členem struktury, zatímco při předávání adresy nesmíte zapomenout použít operátor nepřímého členství.

10. `int judge (int (*pf) (const char *));`

Kapitola 8

1. Krátké, nerekurzivní funkce, pro které stačí jeden řádek kódu.
2. a. `void song(char * name, int times = 1);`

   b. Žádné. Implicitní informace o hodnotě obsahují pouze prototypy.

   c. Ano, pokud uchováte implicitní hodnotu parametru `times`.

   ```
   void song(char * name = "O, My Papa", int times = 1);
   ```

3. Chcete-li vytisknout znak uvozovky, použijte buď řetězec `"\""` nebo znak `'"'`. Obě metody ukazují následující funkce:

```
#include <iostream.h>
void iquote(int n)
{
  cout << "\"" << n << "\"";
}
void iquote(double x)
{
        cout << '"' << x << '"';
}
```

```
void iquote(const char * str)
{
  cout << "\"" << str << "\"";
}
```

4. a. Tato funkce nesmí položky struktury měnit, použijte tedy kvalifikátor `const`.

```
void show_box(const box & container)
{
        cout << "Vyrobce: " << container.maker << "\n";
        cout << "Vyska = " << container.height << "\n";
        cout << "Sirka = " << container.width << "\n";
        cout << "Delka = " << container.length << "\n";
        cout << "Objem = " << container.volume << "\n";
}
```

   b.
```
void set_volume(box & crate)
{
        crate.volume = crate.height * crate.width * crate length;
}
```

5. a. Dosáhnete toho, jestliže druhému parametru dodáte implicitní hodnotu.

```
double mass(double d, double v = 1.0);
```

   Můžete také použít přetížení:

```
double mass(double d, double v);
double mass(double d);
```

   b. Pro funkci `repeat` nemůžete použít implicitní hodnotu, protože implicitní hodnoty musí následovat zleva doprava. Můžete použít přetížení:

```
void repeat(int times, char * str);
void repeat(const char * str);
```

   c. Můžete použít přetížení funkcí:

```
int average(int a, int b);
double average(double x, double y);
```

   d. Nemůžete to provést, protože obě verze by měly stejnou signaturu.

   e. Alespoň jedna verze musí být definována jako statická funkce v jednom ze souborů:

```
static int average(int a, int b);   // definice v prvním souboru
static double average(int a, int b); // definice ve druhém souboru
```

6.
```
template<class T>
T max(T t1, T t2) // nebo T max(const T & t1, const T & t2)
{
  return t1 > t2? t1 : t2;
}
```

7.
```
template<> box max(box b1, box b2)
{
```

```
    return b1.volume > b2.volume? b1 : b2;
}
```

8. a. Proměnná `homer` je automatickou proměnnou automaticky.
   b. Proměnná `secret` musí být v jednom souboru definována jako externí proměnná a ve druhém deklarována pomocí klíčového slova extern.
   c. Proměnná `topsecret` musí být definována jako statická externí proměnná a jako takovou ji bude v externí definici předcházet klíčové slovo static.
   d. Proměnná `beencalled` musí být definována jako lokální statická proměnná a jako takovou bude její deklaraci ve funkci předcházet klíčové slovo static.
9. Deklarace `using` zpřístupňuje z prostoru jmen jediný název a její oblast platnosti odpovídá té oblasti deklarace, ve které se nachází. Direktiva `using` zpřístupňuje všechny názvy z prostoru jmen. Když ji použijete, je to jako byste deklarovali názvy v nejmenší oblasti deklarace, která obsahuje deklaraci `using` i samotný prostor jmen.

Kapitola 9

1. Třída je typ definovaný uživatelem. Deklarace třídy specifikuje způsob uložení dat a metody (členské funkce), které se budou používat pro přístup k datům a manipulaci s nimi.
2. Třída reprezentuje operace, které lze s objektem této třídy provádět pomocí veřejného rozhraní představovaného metodami; toto je abstrakce. Pro datové položky může třída použít soukromou viditelnost (implicitní), což znamená, že k datům lze přistupovat pouze pomocí členských funkcí; toto je skrývání dat. Podrobnosti implementace, jako je reprezentace dat a kód metod, jsou skryty; toto je zapouzdření.
3. Třída definuje typ včetně možností jeho použití. Objekt je proměnná nebo jiný datový objekt (vytvořený například pomocí operátoru `new`) vytvořený a používaný v souladu s definicí třídy. Vztah mezi nimi je stejný jako mezi standardním typem a proměnnou tohoto typu.
4. Jestliže vytvoříte několik objektů dané třídy, bude mít každý objekt místo v paměti pro uložení své množiny dat. Ale všechny tyto objekty budou používat jednu množinu členských funkcí. (Metody bývají obvykle veřejné, zatímco datové položky soukromé. To je ale otázka strategie a s požadavky třídy nesouvisí.)

Poznámka

Program čte názvy pomocí výrazu `cin.get(char *, int)` místo `cin >>`, protože výraz `cin.get()` přečte celý řádek a ne pouze jedno slovo (viz kapitola 4).

5.
```
#include <iostream>
using namespace std;
// definice třídy
class BanAccount
{
private:
```

```
    char name[40];
    char acctnum[25];
    double balance;
public:
    BankAccount(char * client, char * num, double bal = 0.0);
    void set(void);
    void show(void) const;
    void deposit(double cash);
    void withdraw(double cash);
};
```

6. Konstruktor třídy se volá při vytváření objektu této třídy nebo ho lze zavolat explicitně. Destruktor třídy se volá, když platnost objektu skončí.
7. Pamatujte, že abyste mohli používat funkci strncpy(), musíte vložit hlavičkový soubor `cstring` nebo `string.h`.

```
BankAccount::BankAccount(char * client, char * num, double bal)
{
    strncpy(name, client, 39);
    name[39] = '\0';
    strncpy(acctnum, num, 24);
    acctnum[24] = '\0';
    balance = bal;
}
```

Pamatujte, že implicitní parametry jsou v prototypu funkce, ne v její definici.

8. Implicitní konstruktor je konstruktor bez parametrů nebo takový, jehož všechny parametry mají implicitní hodnotu. Díky takovému konstruktoru můžete deklarovat objekty bez jejich inicializování, i když jste již inicializační konstruktor definovali. Můžete pomocí něho také deklarovat pole.
9.

```
// stock3.h
#ifndef _STOCK3_H_
#define _STOCK3_H_
class Stock
{
private:
    char company[30];
  int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();// bezparametrický konstruktor
    Stock(const char * co, int n, double pr);
    ~Stock() {}      // destruktor, který nic nedělá
    void buy(int num, double price);
    void sell(int num, double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
    int numshares() const { return shares; }
    double shareval() const { return share_val; }
```

```
    double totalval() const { return total_val; }
    const char * co_name() const { return company; }
};
```

10. Ukazatel `this` je dostupný metodám třídy. Ukazuje na objekt, pomocí kterého se metoda vyvolá. To znamená, že `this` představuje adresu objektu, zatímco `*this` reprezentuje samotný objekt.

Kapitola 10

1. Zde je prototyp souboru s definicí třídy a definice funkce pro metody souboru:

```
// prototyp
Stonewt operator*(double mult);
// definice - práci provede konstruktor
Stonewt Stonewt::operator*(double mult)
{
  return Stonewt(mult * pounds);
}
```

2. Členská funkce je součástí definice třídy a je vyvolána určitým objektem. Členská funkce může přistupovat k položkám volajícího objektu implicitně a nemusí používat operátor výběru členu. Spřátelená funkce součástí třídy není, takže je jako funkce volána přímo. Nemá implicitní přístup k položkám třídy a musí tedy na objekt předaný jako parametr použít operátor výběru členu.
3. Musí být spřátelenou funkcí, pokud chce přistupovat k soukromým položkám. Pro přístup k položkám veřejným spřátelenou funkcí být nemusí.
4. Zde je prototyp souboru s definicí třídy a definice funkce pro metody souboru:

```
// prototyp
friend Stonewt operator*(double mult, const Stonewt & s);
// definice - práci provede konstruktor
Stonewt Stonewt::operator*(double mult, const Stonewt & s)
{
  return Stonewt(mult * s.pounds);
}
```

5. Přetíženo nemůže být následujících pět operátorů:

```
sizeof . .* :: ? :
```

6. Tyto operátory musí být definovány pomocí členské funkce.
7. Zde je možný prototyp a definice:

```
// prototyp a definice vložené funkce
operator double() { return mag; }
```

Pamatujte však, že rozumnější je použít metodu `magval()` než definovat tuto konverzní funkci.

Kapitola 11

1. a. Syntaxe je v pořádku, ale tento konstruktor ponechává ukazatel `str` neinicializovaný. Měl by ho nastavit buď na hodnotu `NULL`, nebo inicializovat pomocí operátoru `new[]`.

   b. Tento konstruktor nový řetězec nevytvoří, pouze zkopíruje adresu původního řetězce. Měl by použít operátor `new[]` a funkci `strcpy`.

   c. Konstruktor kopíruje řetězec na místo, které nemá přiděleno. Měl by použít výraz `new char[len + 1]`, který přidělí dostatečné množství paměti.

2. Za prvé, když platnost objektu tohoto typu skončí, data, na která ukazuje členský ukazatel objektu, zůstanou v paměti, kde budou zabírat místo a přístup k nim nebude možný, protože ukazatel je ztracen. Nápravu provedete dodáním destruktoru třídy, který uvolní paměť přidělenou v konstruktoru operátorem `new`. Za druhé, jakmile destruktor tuto paměť uvolní a v případě, že program inicializoval jeden objekt pomocí druhého, bude se destruktor snažit uvolnit paměť i podruhé. Příčinou je skutečnost, že při implicitní inicializaci jednoho objektu druhým se kopírují hodnoty ukazatelů, ale již ne data, na která ukazují. Tím se vytvoří dva ukazatele na stejná data. Řešením je definovat kopírovací konstruktor třídy, který zkopíruje také data. Za třetí, přiřazením jednoho objektu k druhému může vzniknout stejná situace jako se dvěma ukazateli na stejná data. Řešením je přetížit operátor přiřazení tak, aby zkopíroval data, ale ne ukazatele.

3. Jazyk C++ automaticky poskytuje následující členské funkce:
   - Implicitní konstruktor, pokud sami žádný nedefinujete.
   - Kopírovací konstruktor, pokud sami žádný nedefinujete.
   - Operátor přiřazení, pokud sami žádný nedefinujete.
   - Implicitní destruktor, pokud sami žádný nedefinujete.
   - Adresový operátor, pokud sami žádný nedefinujete.

   Implicitní konstruktor nedělá nic, ale umožňuje deklarovat pole a neinicializované objekty. Implicitní kopírovací konstruktor a implicitní operátor přiřazení přiřazují podle položek. Implicitní destruktor nedělá nic. Implicitní adresový operátor vrací adresu vyvolávajícího objektu (to znamená hodnotu ukazatele `this`).

4. Položka personality by měla být deklarována buď jako pole znaků, nebo jako ukazatel na typ `char`. Další možností je vytvořit objekt třídy `String`. Uvádíme dvě možná řešení. Změny jsou vyznačeny tučně.

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private:  // nepovinné
  char personality[40];   // velikost pole
  int talents;
public:           // nutné
// metody
```

```
nifty();
nifty(const char * s);
friend ostream & operator <<(ostream & os, const nifty & n);
};       // nezapomeňte na uzavírací středník
nifty()
{
  personality[0] = '\0';
  talents = 0;
}
nifty::nifty(const char * s)
{
  strcpy(personality, s);
  talents = 0;
}
ostream & operator<<(ostream & os, const nifty & n)
{
  os << n.personality << '\n';
  os << n.talents << '\n';
  return os;
}
```

Můžete napsat kód i takto:

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private:  // nepovinné
  char * personality;     // vytvoří ukazatel
  int talents;
public:             // nutné
// metody
nifty();
nifty(const char * s);
nifty(const nifty & n);
~nifty() { delete personality; }
nifty & operator=(const nifty & n) const;
friend ostream & operator <<(ostream & os, const nifty & n);
};       // nezapomeňte na uzavírací středník
nifty()
{
  personality = NULL;
  talents = 0;
}
nifty::nifty(const char * s)
{
  personality = new char [strlen(s) + 1];
  strcpy(personality, s);
  talents = 0;
}
ostream & operator<<(ostream & os, const nifty & n)
{
```

```
    os << n.personality << '\n';
    os << n.talents << '\n';
    return os;
}
```

5. a.
```
Golfer nancy;  // implicitní destruktor
Golfer lulu(Little Lulu);      // Golfer(const char * name, int g)
Golfer roy(Roy Hobbs, 12); // Golfer(const char * name, int g)
Golfer * par = new Golfer;  // implicitní konstruktor
Golfer next = lulu; // Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker";  // Golfer(const char * name, int g)
*par = nancy; // implicitní operátor přiřazení
nancy = "Nancy Putter"; // Golfer(const char * name, int g),
                        //potom implicitní operátor přiřazení
```

Poznámka

Některé kompilátory budou volat implicitní operátor přiřazení navíc pro pátý a šestý příkaz.

b. Třída by měla definovat operátor přiřazení, který zkopíruje místo adres data.

Kapitola 12

1. Veřejné položky základní třídy budou veřejnými i v odvozené třídě. Chráněné položky základní třídy budou chráněnými i v odvozené třídě. Soukromé položky se dědí, ale nelze k nim přistupovat přímo. Odpověď na opakovací otázku č. 2 představuje výjimku z těchto obecných pravidel.
2. Metody konstruktoru se nedědí, nedědí se ani destruktor, operátor přiřazení a spřátelené třídy.
3. Hodnotu prvku byste sice mohli použít, ale nemohli byste do ní nic přiřadit:

```
ArrayDb tosanjose[4];
double n = tosanjose[2];          // ok
tosanjose[1] = 68.8;              // nelze
```

   Provedení metody by se trochu zpomalilo, protože příkaz `return` zahrnuje kopírování objektu.
4. Konstruktory jsou volány v pořadí podle odvození, přičemž nejvzdálenější předek se volá nejdříve. Destruktory se volají v opačném pořadí.
5. Ano, každá třída musí mít vlastní konstruktory. Pokud do odvozené třídy nejsou přidány žádné nové položky, může mít konstruktor prázdné tělo, ale musí existovat.
6. Volána bude pouze metoda odvozené třídy. Nahrazuje definici základní třídy. Metoda základní třídy se volá pouze tehdy, jestliže tato metoda není v odvozené třídě předefinována. Ve skutečnosti byste však jako virtuální měli deklarovat všechny funkce, které budou předefinovány.
7. Odvozená třída by měla definovat operátor přiřazení, jestliže konstruktory třídy inicializují členské ukazatele pomocí operátorů `new` nebo `new[]`. Obecněji lze říct, že

odvozená třída by měla definovat operátor přiřazení, jestliže se implicitní operátor přiřazení pro položky odvozené třídy nehodí.

8. Ano, adresu objektu odvozené třídy můžete přiřadit ukazateli na třídu základní. Adresu objektu základní třídy můžete přiřadit ukazateli na třídu odvozenou (přetypování na potomka) pouze explicitně a použití takového ukazatele nemusí být bezpečné.

9. Ano, objekt odvozené třídy můžete přiřadit objektu třídy základní. Žádné datové položky, kterou jsou v odvozeném typu nové, se však nezkopírují. Program používá operátor přiřazení základní třídy. Přiřazení v opačném směru (objekt základní třídy přiřazený objektu třídy odvozené) je možné pouze tehdy, jestliže odvozená třída definuje konverzní operátor, což je konstruktor, který má jako jediný parametr referenci na základní typ.

10. Může ho použít, protože jazyk C++ umožňuje, aby reference na základní typ odkazovala libovolný typ odvozený ze základní třídy.

11. Při předání objektu hodnotou se vyvolá kopírovací konstruktor. Vzhledem k tomu, že formálním parametrem je objekt základní třídy, vyvolá se konstruktor této základní třídy. Kopírovací konstruktor má jako parametr referenci na základní třídu a tato reference může odkazovat na objekt odvozené třídy předaný jako parametr. Výsledkem je vytvoření nového objektu základní třídy, jehož položky odpovídají části základní třídy odvozeného objektu.

12. Předání objektu odkazem umožňuje funkci využít virtuálních funkcí. Předáváte-li objekt referencí místo hodnotou, stojí to méně paměti a času, zvláště v případě velkých objektů. Hlavní výhodou při předávání hodnotou je ochrana původních dat. Stejného výsledku však můžete dosáhnout, jestliže předáte referenci jako konstantní typ.

13. Jestliže bude `head()` normální funkcí, potom výraz `p->head()` vyvolá metodu `Corporation::head()`. Pokud bude `head()` funkcí virtuální, potom výraz `p->head()` vyvolá metodu `PublicCorporation::head()`.

14. Za prvé, situace neodpovídá modelu *je*, takže veřejná dědičnost se nehodí. Za druhé, definice metody `area()` ve třídě `House` skryje verzi této metody ze třídy `Kitchen`, protože obě metody mají rozdílné signatury.

Kapitola 13

1.

```
class Bear     class PolarBear       Veřejná - lední medvěd je druhem medvěda.
class Kitchen  class Home            Soukromá - dům má kuchyni.
class Person   class Programmer      Veřejná - programátor je druh osoby.
class Person   class HorseAndJockey  Soukromá - kůň a jezdecký tým obsahují oso-
bu. class Person.                    class Driver
Veřejná - řidič je osoba. Automobile
class                                je soukromá, protože řidič má
Automobile                           automobil.
```

2.

```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & f) : glip(g), fb(f) { }
// poznámka: výše uvedená metoda používá implicitní kopírovací konstruktor
// Frabjous
```

```
void Gloam::tell()
{
   fb.tell();
   cout << glip << '\n';
}
```

3.
```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & f) : glip(g), fb(f) { }
// poznámka: výše uvedená metoda používá implicitní kopírovací konstruktor
// Frabjous
void Gloam::tell()
{
   Frabjous.tell();
   cout << glip << '\n';
}
```

4.
```
class Stack<Worker *>
{
private:
   enum {MAX = 10};             // specifická konstanta třídy
   Worker * items[MAX];   // obsahuje zásobník položek
   int top;                     // index horního prvku zásobníku
public:
   Stack();
   Boolean isemty();
Boolean isfull();
Boolean push(const Worker * & item); // přidá položku do zásobníku
Boolean pop(Worker * & item); // odebere horní položku
};
```

5.
```
ArrayTP<String> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTp< StackTP<Worker *> > arr_stk_wpr;
```

6. Jestliže budou ve třídě dva řádky dědičnosti sdílet stejného předchůdce, bude třída mít dvě kopie položek předchůdce. Jestliže ze třídy předchůdce uděláte virtuální základní třídu pro bezprostředně následujícího potomka, bude tento problém vyřešen.

Kapitola 14

1. a. Deklarace spřátelené funkce musí vypadat následovně:

```
friend class clasp;
```

b. Musí být provedena dopředná deklarace, aby kompilátor dokázal funkci `void snip(muff &)` interpretovat:

```
class muff;    // dopředná deklarace
class cuff {
public:
        void snip(muff &) { … }
        …
        };
```

c. Za prvé, deklarace třídy `cuff` musí předcházet deklaraci třídy `muff`, aby kompilátor rozuměl výrazu `cuff::snip()`. Za druhé, kompilátor potřebuje dopřednou deklaraci třídy `muff`, aby rozuměl funkci `snip(muff &)`.

```
class muff;  // dopředná deklarace
class cuff {
public:
    void snip(muff &) { … }
    …
};
```

2. Ne. Aby třída `A` mohla mít za přítele členskou funkci třídy `B`, musí deklarace třídy `B` předcházet deklaraci třídy `A`. Dopředná deklarace nepostačuje, neboť ta by třídu `A` informovala, že `B` je třídou, ale neodhalila by názvy členů třídy. Podobně aby třída `B` mohla mít za přítele členskou funkci třídy `A`, musela by úplná deklarace třídy `A` předcházet deklaraci třídy `B`. Tyto dva požadavky se vzájemně vylučují.
3. Jediný přístup ke třídě je pomocí jejího veřejného rozhraní, což znamená, že jediné, co můžete s objektem třídy `Sauce` udělat, je vytvořit ho zavoláním konstruktoru. Ostatní položky (`soy` a `sugar`) jsou soukromé implicitně.
4. Předpokládejte, že funkce `f1()` volá funkci `f2()`. Příkaz `return` ve funkci `f2()` způsobí, že provádění programu přejde na další příkaz následující ve funkci `f1()` za voláním funkce `f2()`. Příkaz `throw` způsobí, že program bude zpětně procházet sekvenci funkčních volání, až najde pokusný blok (try block), který přímo nebo nepřímo obsahuje volání funkce `f2()`. Ten se může nacházet ve funkci `f1()` nebo v jiné funkci, která `f1()` volala. Potom se budou provádět příkazy do následujícího záchytného bloku (catch block), ne do prvního příkazu za voláním funkce.
5. Záchytné bloky musíte uspořádat v pořadí od nejpozději odvozené třídy ke třídě, která byla odvozena nejdříve.
6. V prvním příkladě bude podmínka `if` pravdivá, jestliže ukazatel `pg` ukazuje na objekt třídy `Superb` nebo na objekt jakékoli třídy od této třídy odvozené. Konkrétně je podmínka pravdivá, jestliže ukazuje na objekt třídy `Magnificent`. V druhém příkladě bude podmínka `if` pravdivá pouze pro objekt třídy `Superb`, ale ne pro objekty tříd odvozených.
7. Operátor `dynamic_cast` umožňuje přetypování na předka pouze v rámci hierarchie tříd, zatímco operátor `static_cast` umožňuje přetypování na předka i na potomka a také konverze z výčtových typů na celočíselné a naopak.

Kapitola 15

1.
```
#include <string>
using namespace std;
class RQ1
{
private:
  string st;     // objekt třídy string
public:
  RQ1() : st(" ") {}
  RQ1(const char * s) : st(s) {}
```

```
    ~RQ1() {}
    // další metody
    };
```

Explicitní kopírovací konstruktor, destruktor ani operátor přiřazení již nejsou potřeba, protože objekt třídy `string` má vlastní správu paměti.

2. Můžete přiřadit jeden objekt třídy `string` jinému. Objekt třídy `string` má svou vlastní správu paměti, takže normálně se nemusíte obávat, že kapacita řetězce přeteče.

3.
```
#include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
{
    for (int i = 0; i < str.size(); i++)
        str[i] = toupper(str[i]);
}
```

4.
```
auto_ptr<int> pia = new int[20];  // špatně, použijte new, ne new[]
auto_ptr<int>(new string);// špatně, chybí název ukazatele
int rigue = 7;
auto_ptr<int>(&rigue);    // špatně, není přidělena paměť pomocí new
auto_ptr dbl (new double);// špatně, vynecháno <double>
```

5. U zásobníku aspekt LIFO znamená, že asi budete muset vytáhnout hodně holí, než se dostanete k té, kterou potřebujete.

6. Množina (set) uloží pouze jednu kopii každé hodnoty, takže například pět zásahů čísla 5 se uloží pouze jednou.

7. Pomocí iterátorů se můžete pomocí objektů s ukazatelovým rozhraním pohybovat daty, která jsou uspořádána jinak, než v poli, například daty v obousměrném seznamu.

8. Řešení knihovny STL umožňuje, aby její funkce mohly být používány s obyčejnými ukazateli na obyčejná pole stejně dobře jako s iterátory na její kontejnerové třídy. Tím se zvyšuje obecnost.

9. Můžete přiřadit jeden objekt třídy `vector` druhému. Objekt třídy `vector` spravuje svou vlastní paměť, takže do něho můžete vkládat prvky, přičemž dochází automaticky k jeho zvětšování. Pomocí metody `at()` můžete automaticky kontrolovat meze.

10. Obě funkce `sort()` i funkce `random_shuffle()` požadují iterátor přímého přístupu, zatímco objekt třídy list má pouze obousměrný iterátor. Ke třídění můžete místo všeobecných funkcí použít členské funkce `sort()` ze šablonové třídy `list`, ale členská funkce ekvivalentní funkci `random_shuffle()` neexistuje. Mohli byste však zkopírovat seznam do vektoru, potom změnit pořadí prvku ve vektoru a výsledek zkopírovat zpět do seznamu. (V čase psaní této knihy většina kompilátorů ještě neimplementovala členskou funkci `sort()`, která má jako parametr objekt `Compare`. Rovněž implementace třídy list v Microsoft Visual C++ 5.0 obsahuje několik chyb, které brání provedení navržených konverzí.)

Kapitola 16

1. Soubor `iostream` definuje třídy, konstanty a manipulátory pro správu vstupu a výstupu. Tyto objekty spravují proudy a vyrovnávací paměti pro vstup a výstup. Soubor také vytváří standardní objekty (`cin`, `cout`, `cerr` a `clog` a jejich ekvivalenty pro široké znaky) pro správu standardních vstupních a výstupních proudů spojených s každým programem.
2. Vstup z klávesnice generuje sérii znaků. Napíšete-li `121`, vygenerují se tři znaky a každý z nich bude reprezentován jednobajtovým binárním kódem. Pokud se má hodnota uložit jako typ `int`, musí tyto tři znaky být převedeny na jednoduchou binární reprezentaci hodnoty `121`.
3. Implicitně standardní výstupní proud a standardní chybový proud posílají výstup na standardní výstupní zařízení, kterým je běžně monitor. Jestliže však necháte výstup přesměrovat do souboru, spojí se standardní výstup se souborem místo s obrazovkou. Standardní chybový proud však bude nadále spojen s obrazovkou.
4. Třída `ostream` definuje verzi funkce `operator<<()` pro každý základní typ jazyka C++. Kompilátor interpretuje výraz `cout << spot` následujícím způsobem:

   ```
   cout.operator<<(spot)
   ```

   Potom může toto funkční volání porovnávat s prototypem, který má stejný typ parametru.
5. Řetězit můžete metody pro výstup, které vracejí typ `ostream &`. To způsobí, že vyvolání metody s objektem vrátí tento objekt. Vrácený objekt potom může vyvolat další metodu v řadě.
6.
   ```
   // rq16-6.cpp
   #include <iostream>
   #include <iomanip>
   using namespace std;
   int main()
   {
     cout << "Zadejte cele cislo: ";
     int n;
     cin >> n;
     cout << setw(15) << "desitkova soustava" << setw(15)
          << "sestnactkova soustava" << setw(15) << "osmickova soustava"
          << "\n";
     cout.setf(ios::showbase);       // nebo cout << showbase;
     cout << setw(15) << n << hex << setw(15) << n
          << oct << setw(15) << n << "\n";
     return 0;
   }
   ```
7.
   ```
   // rq15-7.cpp
   #include <iostream>
   #include <iomanip>
   using namespace std;
   int main()
   {
   ```

```
    char name[20];
    float hourly;
    float hours;
    cout << "Zadejte jmeno: ";
    cin.get(name, 20).get();
    cout << "Zadejte hodinovou mzdu: ";
    cin >> hourly;
    cout << "Zadejte počet odpracovanych hodin: ";
    cin >> hours;
    cout.setf(ios::showpoint);
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::right, ios::adjustfield);
// nebo cout << showpoint << fixed << right;
    cout << "Prvni format:\n";
    cout << setw(30) << name << ": Kc" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    cout << "Druhy format:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": Kc" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    return 0;
}
```

8. Zde je výstup:

```
ct1 = 5; ct2 = 9
```

První část programu ignoruje mezery a nové řádky, zatímco druhá ne. Všimněte si, že druhá část programu začíná čtení znakem nového řádku za prvním znakem `q` a započítá ho do celkového počtu.

9. Forma `ignore()` nebude fungovat, jestliže vstupní řádek bude mít více než 80 znaků. V takovém případě prvních 80 znaků jen přeskočí.

Rejstřík

4-bajtové ukazatele 116

A

abstrakce 355
– základní třídy 550
adresy struktur 263
akce preprocesoru 23
aktivace dynamické vazby 532
algoritmy 764
alokace paměti 121
alternativa s modifikátorem const 571
ANSI 8
ANSI C 204
ANSI/ISO standard C++ 929
aplikace MFC Windows 13
aritmetické operátory v C++ 75
aritmetický
– operátor 158
– problém 77
aritmetika ukazatelů 128, 131
ASCII-kód 163, 170
asociativita 77
asociativní kontejnery 751
AT&T 11
automatické
– proměnné 141, 322
– typové konverze 441
autoritářská funkce 21

B

Basic 164
běžná spojení 182
binární
– hledání 918
– soubory 836
bitová pole 110
bitové
– logické operátory 873
– operátory 871, 875
– pole 110
bity 51
bloky 159
booleovské literály 148
Borland C++ 3.1 (DOS) 12
budoucnost 347

C

C++ 1, 6, 17
celočíselné
– hodnoty 82
– konstanty 59
– typy 51
– typy short, int a long 51
cin pro vstup 177
cin.get(char) 178
cout 32
cyklus
– do while 175
– for 92, 145. 169

cyklus for v C++ 145
cyklus while 4, 169. 172, 176
cykly 17, 219
cykly 219

Č

čárkový operátor 161
částečné specializace 612
čísla 121
čítač VCR 57
členské funkce 362

D

další
– příkazy C++ 33
– přiřazení 895
datová struktura 5, 58
datový typ 391
dědičnost 515
– tříd 511
dědictví 558
definice
– konstruktorů 370
– šablony třídy 588
definované funkce 41
deklarace odvozené třídy 517
deklarační příkaz 30, 120
deklarování ukazatele 268
dělení 78
delete 124
dereferencování
– členů 877
– ukazatelů 131
desetinné oddělovače 795
destruktory 373, 3793 538
– třídy 369
direktiva
– #define 56
– preprocesoru 929
dobrý cyklus 166
dočasné proměnné 288
dopředný iterátor 733
druhá generace 309
druhy
– iterátorů 731
– kontejnerů 741
dvojkova soustava 858
dvourozměrná pole 186
dynamická
– alokace 339
– paměť 452
– pole 125
dynamické
– přidělení paměti 543
– přidělování paměti 451
enumerátor 114
explicitní
– instance 611
– specializace 308, 611
externí
– deklarace 107
– proměnné 328, 333
faktoriál 153
formátování
– incore 848
– pomocí objektu cout 789
– příkazů 197
– příkazů if else 197
– zdrojového kódu C++ 28
FORTRAN 4
fronta (queue) 749
fronta s prioritou (priority_queue) 750
funkce 36, 228
– cin.get(ch1) 185
– copy() 736

– cout 73
– cout.put() 63
– generované kompilátorem 553
– is_int() 206
– jako rozhraní 20
– main() 19, 154
– Minimum 923
– pow() 40
– pro porovnání řetězců 893
– putchar() 183
– s návratovou hodnotou 37
– stonetolb() 45
– strcmp() 168
– strcpy() 137, 140
– strlen() 133
funkční
– adaptéry 762
– objekty (aka funktory) 757
– parametry 283
– prototyp 230
generické programování 6, 727
get() 99
getline() 99

H

halda 922
hierarchie iterátorů 734
historie 2
hlavičkový soubor iomanip 803
hlavičky funkcí 43
hlavní bod 141
hodnot yenumerátorů 114
hodnoty pro výčty 114

Ch

chráněná dědičnost 586
chybový bit 102
chytré návrhy 33

I

IBM 49
IBM PC 173
IDE 14
if else 198
implementace 488
implementace
– členských funkcí 360
– odvozené třídy 520
implementační soubor 375
implicitní
– členské funkce 464
– instance 611
– konstruktor 371, 465, 553, 885
index roku 188
inicializace 53
– komponent 580
– objektů 523
– obsažených objektů 576
– pole 91, 93
– ukazatelů 118
inkrementování 157
integrovaná vývojová prostředí 10
iterátor přímého přístupu 733
iterátory 728

J

jazyk C 3
jazyková vazba 339
jednoduché proměnné 49
jednoduchý
– datový objekt 122
– vstup ze souboru 824
jednoznakový vstup 812
jiné přátelské vztahy 646
jména proměnných 50

K

klávesa Enter 64
klávesnice 180
klíčové slovo using 586
klientský soubor 377
knihovna
– cctype 208
– STL 905, 935
komentáře v C++ 22
kompilátor 165
kompilátory
– pro Macintosh 14
– Windows 13
kompozice 585
koncept kontejneru 742
koncepty funktorů 758
konec souboru 180, 182
konstanta CLOCKS_PER_SEC 173
konstantní členské funkce 379
konstanty 60
– char 64
– v pohyblivé řádové čárce 74
konstruktor
– používající část pole 885
– používající pole 885
konstruktory 538
– používajícími operátor new 485
kontejner
– multimap 755
– set 752, 903
kontejner set 903
konverze
– při předávání parametrů 83
– při přiřazení 81
– typů 80
– ve výrazech 82
konverzní
– funkce 439, 485
– problémy 81
kopie znaku 887
kopírovací konstruktor 465, 553, 886
kvalifikátor const 68
kvalifikátory paměťové třídy 336

L

lahůdka OOP 89
lexikální jednotky 28
logické
– operátory 193
– výrazy 199
logický
– operátor A 201
– operátor NEBO 199

M

Macintosh 5
manuál C++ 2
mechanismus výjimek 658
menu
– File New 14
– Project 15
metoda
– is_open() 830
– setf() 796
metody
– getline(), get() a ignore() 816
– třídy 490
– třídy istream 819
– třídy ostream 785
– vkládání 895
množinové operace 920
modifikace pole 247
modifikátor
– const 557
– static 330

možnost výběru 445
MS-DOS 180

N

náhodné chůze 430
nahrazování 896
naplnění pole 245
násobení 428
násobné parametry 235
nastavení
– rozmezí 203
– stavů 809
návěští 215
navrácení
– reference 294
– struktur 256
nečlenské funkce 417
nejlepší shody 315
nekonečný cyklus 170
nemodifikující sekvenční operace 905
neočekávaný vstup řetězce 818
nepojmenované prostory jmen 347
neprovedení předefinování 539
netopýr 84
netypové parametry 599
nevirtuální základní třídy 626
nevýrazy 151
nové třídy 578
nový
– hlavičkový soubor 156
– rys C++ 25
numerické operace 925
numerický vstup 103

O

objekt
– cin 182
– cout 63
objektově orientované
programování 4, 354
obousměrná fronta (deque) 746
obousměrný iterátor 733
obsažený objekt 577
oddělená kompilace 317
odvozená třída 545
odvozené typy 89
omezení přetížení 406
OOP 1
operátor
– ? 210
– << 74
– delete 124
– delete [] 126
– dynamic_cast 686
– modulo 79
– new 124, 479
– přiřazení 470, 554, 558
– sčítání 404
– sizeof 93
– typeid 690
operátory 871
– inkrementování 156
– inkrementování (++) 156
– new a delete 461
– posunu 871
– přetypování 694
oprava přiřazení 471
osmičková soustava 857
otevření více souborů 827
otočení řetězce 162

P

parametrparametr 287
parametry 317
– typu odkaz 294
Pasc… 164

platnosti třídy 389
pohled na šablonu třídy 594
pohyblivá řádová čárka 69
pokusné bloky 664
pole 89, 240
– objektů 386
– polí 187
– qualify 203
– stacks[1] 130
– struktur 110
polymorfismus 298
porovnání řetězců 166
použití
– cin 34
– dynamického pole 126
– get() 103
– konstruktoru 370
– new a delete 139
– polí 242
– řetězců v poli 96
– RTTI 693
– šablony třídy 591
– třídy 365
– třídy auto_ptr 712
– upravené třídy Student 584
– zásobníku ukazatelů 595
používání
– metod základních tříd 581
– ukazatelů na objekty 481
– výjimek 683
poznámky ke třídě auto_ptr 714
práce
– s daty 49
– s pamětí 888
– s řetězci 705
– s vektory 723
– se znaky 208
přátelé 443, 409, 539, 637
pravidla pro konstruktory 617
předávání
– hodnotou 234
– reference 556
předdefinované funktory 760
přenositelnost 7
přepnutí bitu 876
přesměrování 781
přetěžování 427
– funkcí 303
– operátorů 400
přetížené
– operátory 408
– šablony 306
přetížení 300
– funkcí 179
– operátoru 26, 412, 485
– přetíženého operátoru 428
přetížený operátor << 782
přetypování 84, 932
– tříd 433
přidání dat do souboru 834
příkaz
– = výraz + středník 152
– continue 217
– if 193, 194
– if else 195
– switch 211
příkazový řádek 827
příkazy 149
– C++ 29
– if else 216
– větvení 172
přímý přístup 841
priorita operátorů 867
připojení a přidání 894

přiřazovací
– operátory 158
– příkaz 32
přirozená velikost 58
přístup
– k řetězci 888
– k vnořeným třídám 650
privátní metody 560
přizpůsobivé funktory 762
přizpůsobování pravidel 152
problémy 102
procedure 41
procházení řetězců 155
programování v C 3
programové moduly v C++ 227
programový tok 175
prohledávání řetězce 890
proměnná int 81
proměnné 30
– reference 280
prostory jmen 340, 933
prostředí Windows 180
prototypy 232
proveditelný kód 9
první generace 309
referenční
– deklarace 31
– proměnné 280
register 326
rekurze 265
relační
– operace 915
– operátory 163
– výrazy 145, 163
režimy souboru 831
rozhraní 487
rozsah platnosti 322
RTTI 685
rysy C++ 7

Ř

řádové čárky 71
řetězce 94
– ve stylu jazyka C 252
řetězcový literál 94
řetězení výstupu 784
řízení přístupu 652
– chráněný režim 528

S

sada ASCII 863
sdílení přátelé 646
sekvence 744
sekvenční operace 909
sestavení 11
sestavování pod Unixem 11
seznam (list) 746
simulace 499
– fronty 486
simulovaný EOF 181
skupiny
– algoritmů 764
– tříd 602
složený příkaz 159
slučování 919
smíšené přiřazení 541
soubor
– cstring 96, 167
– iostream 23, 179, 778
soukromá dědičnost 579
specializace 313
– šablon 611
spojování řetězců 95
spřátelené
– členské funkce 642

– funkce 410
– třídy 638
srovnání řetězců 166
standardní
– ASCII 10
– knihovna šablon 699, 715, 766
– manipulátory 802
– parametry 295
standardy 7
statická paměť 141
statické položky tříd 452
stavové položky 425
stavy proudů 808
struktura v C++ 104
struktury 104, 256
styl
– C 163
– zdrojového kódu C++ 29
subroutine 41
swap() 897
syntaktické triky 160
syntaxe prototypu 232
systém V/V 85

Š

šablona autoptr 934
šablonová
– třída STRING 881
– třída vector 716
šablony 647
– funkcí 303
– istream_iterator 736
– ostream_iterator 736
– tříd 588
šestnáctková soustava 858, 859
široký datový typ 67

T

test hodnoty bitu 876
testovací-výraz 147
testování cyklu 146
třetí generace 310
třída
– ArrayDb 569
– auto_ptr 711
– Customer 496
– exception 679
– istream 178
– paměti static 327
– Queue 487
– Stock 373
– string 462, 699, 894, 935
– Student 575
– Student (nová verze) 580
– type_info 690
– Vector 418
– Worker 614
třídění 917
třídy 35
Turbo C++ 12
tvar funkce 42
typ
– bool 68
– char 61
– double 79
– přetypování 171
– unsigned short 83
typy bez znaménka 56

U

účinky proudových stavů 810
ukazatel
– jako iterátor 735
– this 381

ukazatele 132, 249, 529, 784
– na funkce 267
uniony 111
univerzálnost
šablon 608
univerzálnost výjimek 661
UNIX 3, 180
úprava šířky polí 792
upřednostňované formátování 198
using-direktivy 343
uvolnění zásobníku 665
užitečné iterátory 737

V

varianty funkcí 40
vazba 321
vektor (vector) 745
vektory 718
velikost kroku 155
velké pole char 135
veřejná dědičnos 557
vícenásobná dědičnost 613, 628
virtuální
– členské funkce 531
– funkce 536
– základní třídy 627
vlastnosti
– hodnot výrazů 150
– prostoru jmen 342
– struktury 108
– ukazatelů 131
vložené
– funkce 277
– metody 363
vnořené třídy 649
vnořování do šablony 652
volaná funkce 37
volání
– cin.get() 179
– funkce 148
– get() 101
volba QuickWin 13
volná paměť 115, 141
vrácení reference 556
vstup 897
vstup
– pomocí objektu cin 805
– řetězce 97
– textu 176
– třídy string 704
– znaku 815
vstupní
– iterátor 732
– proud 807
výbor ANSI/ISO 24
výčtové typy 112
výjimka bad_alloc 680
výjimky 637
vylepšená třída String 472
vylepšení
– a modely 735
– operátoru [] 571
výplňové znaky 794
vypnutí bitu 876
vyprázdnění výstupní vyrovnávací paměti 788
výraz int 43
výrobce kompilátoru 54
vyrovnávací paměť 776
výstup
– a soubory 775
– C++ pomocí cout 25
– do souboru 823
– pomocí objektu cout 782
výstupní iterátor 733
výstupy 776

vytváření programu 9
vytvoření řetězce 700
vyvolání funkce 269

W

wchar_t 67
while 171
Windows 36
Windows 95 181
Windows NT 52

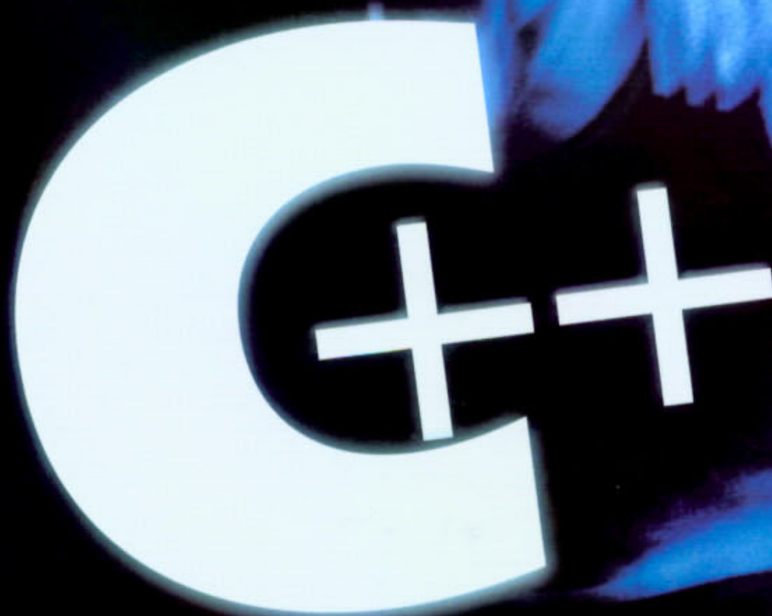
Z

zahájení C++ 17
základní
– přiřazení 889
– třída 512
základy ukazatelů na funkce 268
zapnutí bitu 876
zásobník (stack) 325, 750
– ukazatelů 594
zbloudilý středník 172
zdrojový kód 10
získání adresy funkce 268
změna číselného základu používaného při zobrazení 790
změnovou část cyklu 146
změny pořadí 924
znak
– # 177
– nového řádku 27
– nového řádku (\n) 27
znaková sada ASCII 61, 167
zobrazení pole 246
zobrazování hodnoty 794
zpracování času 401
zvláštnosti referencí 287

EDICE PROGRAMOVÁNÍ

PRO KAŽDÉHO UŽIVATELE

mistrovství v



Necháte-li v internetovém knihkupectví vyhledat řetězec „C++", vypadne vám několik stovek anglicky psaných titulů k tomuto programovacímu jazyku. Aktuální česká nabídka však činí stěží tolik knih, kolik je bitů do bajtu, a těm, kdo hledají důkladnější učebnici, která by byla oporou při prvních krůčcích v procesu zvládání jazyka i později při programování skutečných aplikací, zatím k růstu v úspěšného programátora příliš nenapomohla.

Ke zvládnutí nejrozšířenějšího jazyka profesionálních aplikací jsme pro vás proto ze záplavy světových publikací – která sama o sobě ještě neznamená kvalitu – vybrali k překladu třetí vydání knihy *The Waite Group's C++ Primer Plus* od Stephena Praty, jež získala v posledních letech mezi programátory celého světa nadšený ohlas. Pro svou uspořádanost výkladu, až průzračnou srozumitelnost a množství názorných příkladů podle mínění mnohých předčila i legendární Stroustrupovu knihu Programming in C. Čtenáři na ní vždy oceňují, že vysvětlování i experimentování s nosnými prvky objektově orientovaného programování, jakými jsou třídy, polymorfismus, dědičnost, šablony či výjimky, je po celou četbu poutavé a zvládnutelné i těmi, kdo se nepovažují za technické typy.

Tato kniha, kterou jsme v překladu příznačně nazvali *Mistrovství v C++*, nevyžaduje předchozí znalosti jazyka C ani C++ a dovede čtenáře na úroveň profesionálního programátora. Naučí vás v jazyce C++ myslet i programovat! Začíná u samých základů objektového programování a dospěje až k pokročilým tématům, jako jsou dědičnost tříd, výjimky nebo využití knihoven – včetně nové STL, tak potřebné pro řádnou konkurenceschopnost programátora na pracovním trhu.

Ačkoli diskutuje otázku kompatibility různých kompilátorů (Microsoft Visual C++, Borland C++ Builder, GNU C++ pod Linuxem aj.), není výklad omezen na žádné konkrétní vývojové prostředí. Popisuje poslední implementaci jazyka ANSI/ISO C++ a vše, co se zde naučíte, upotřebíte na kterékoli platformě.

Z desítek ohlasů ke knize na *www.amazon.com*:

- ***Dr. Leonilda A. Farrow z Highlands, USA:*** *„Díky této knize jsem se naučil C++ a považuji ji po všech stránkách za výtečnou. Můj názor je založen na mnohaletých zkušenostech s jinými učebnicemi. Složitou povahu jazyka C++ vysvětluje s neobvyklou srozumitelností a cvičení na konci každé kapitoly opravdu stojí za to udělat."*
- ***TOP 500 reviewer:*** *„Programy, které (po přečtení knihy) v C vytvářím, nemají nic společného s tím, co jsem programoval předtím."*
- ***Čtenář z Austrálie:*** *„Programoval jsem několik let ve VB a potřeboval jsem se naučit VC++. Vybral jsem si tuto knihu a přečetl ji od A do Z během několika týdnů. Neměl jsem v ní problémy s porozuměním čemukoli, ani s takovými věcmi jako jsou ukazatele, které ve Visual Basicu nejsou."*

Vydalo vydavatelství a nakladatelství Computer Press®
Hornocholupická 22,
143 00 Praha 4,
http://www.cpress.cz

Distribuce:

Computer Press Brno,
náměstí 28. dubna 48,
635 00 Brno-Bystrc,
tel. (05) 46 12 21 11,
fax: (05) 46 12 21 12,
e-mail: distribuce@cpress.cz

Computer Press Bratislava,
Hattalova 12,
831 03 Bratislava, SR,
tel.: +421 (7) 44 45 20 48,
44 25 17 20,
fax: +421 (7) 44 45 20 46,
e-mail: distribucia@cpress.sk

Publikaci lze objednat také na adrese **http://www.vltava.cz**

ISBN 80-7226-339-0
Prodejní kód: K0356

Doporučená cena 890 Kč
1282 Sk

computer press
VŠECHNY CESTY K INFORMACÍM

SAMS