mistrovství v

BESTSELLER

Stephen Prata

C++

4. aktualizované vydání

Od základů
po pokročilé postupy

Nezávislé na konkrétní
implementaci jazyka

Novinky verze C++11

Kontrolní otázky
a cvičení

Kompletní průvodce vývojáře

computer press

SAMS

Stephen Prata

Mistrovství v C++

4. aktualizované vydání

Computer Press
Brno
2013

Mistrovství v C++
4. aktualizované vydání

Stephen Prata

Překlad: Boris Sokol, David Vozák, Libor Beroun, Petr Dokoupil, Lubomír Ptáček, Jiří Huf
Odborná korektura: Lubomír Ptáček
Obálka: Ivana Mitáčková
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Translation © Jiří Huf, 2013

Authorized translation from the English language edition, entitled C++ PRIMER PLUS, 6th Edition, 0321776402 by PRATA, STEPHEN, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright © 2012 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. CZECH language edition published by ALBATROS MEDIA A.S., Copyright © 2013.

Autorizovaný překlad z anglického jazyka publikace nazvané C++ PRIMER PLUS, 6th Edition, 0321776402, kterou vytvořil PRATA, STEPHEN, vydalo nakladatelství Pearson Education Inc., publikující jako Addison-Wesley Professional, Copyright © 2012 Pearson Education, Inc.

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-3828-1

Vydalo nakladatelství Computer Press v Brně roku 2013 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 18 157.
© Albatros Media a. s. Všechna práva vyhrazena. Žádná část této publikace nesmí být kopírována a rozmnožována za účelem rozšiřování v jakékoli formě či jakýmkoli způsobem bez písemného souhlasu vydavatele.

Stručný obsah

Úvod 29
1. Začínáme 39
2. Vzhůru do světa C++ 55
3. Práce s daty 89
4. Složené typy 131
5. Cykly a relační výrazy 195
6. Příkazy větvení a logické operátory 245
7. Funkce – programové moduly jazyka C++ 291
8. Rozšířené možnosti funkcí 347
9. Paměťové modely a jmenné prostory 399
10. Objekty a třídy 449
11. Práce s třídami 501
12. Třídy a dynamické přidělování paměti 555
13. Dědičnost tříd 621
14. Opakované používání kódu v C++ 685
15. Přátelé, výjimky a další 765
16. Třída string a standardní knihovna šablon 829
17. Vstupy, výstupy a soubory 917
18. Nový standard C++ 999

A. Základy číselných soustav 1053
B. Rezervovaná slova v C++ 1057
C. Znaková sada ASCII 1061
D. Priorita operátorů 1067
E. Ostatní operátory 1071
F. Šablonová třída string 1081
G. Metody a funkce knihovny STL 1097
H. Vybraná literatura a internetové prameny 1127
I. Konverze na ANSI/ISO Standard C++ 1131
J. Odpovědi na otázky 1139

Obsah

Úvod 29

Předmluva ke čtvrtému vydání 29
Základní přístup 30
Vzory kódu 30
Uspořádání knihy 30

Kapitola 1: Začínáme 30
Kapitola 2: Vzhůru do světa C++ 31
Kapitola 3: Práce s daty 31
Kapitola 4: Složené typy 31
Kapitola 5: Cykly a relační výrazy 31
Kapitola 6: Příkazy větvení a logické operátory 31
Kapitola 7: Funkce – programové moduly jazyka C++ 32
Kapitola 8: Rozšířené možnosti funkcí 32
Kapitola 9: Paměťové modely a jmenné prostory 32
Kapitola 10: Objekty a třídy 32
Kapitola 11: Práce s třídami 32
Kapitola 12: Třídy a dynamické přidělování paměti 33
Kapitola 13: Dědičnost tříd 33
Kapitola 14: Opakované používání kódu v C++ 33
Kapitola 15: Přátelé, výjimky a další 33
Kapitola 16: Třída string a standardní knihovna šablon 34
Kapitola 17: Vstupy, výstupy a soubory 34
Kapitola 18: Nový standard C++ 34
Příloha A: Základy číselných soustav 34
Příloha B: Rezervovaná slova v C++ 34
Příloha C: Znaková sada ASCII 34
Příloha D: Priorita operátorů 34
Příloha E: Ostatní operátory 34
Příloha F: Šablonová třída string 35
Příloha G: Metody a funkce knihovny STL 35
Příloha H: Vybraná literatura a internetové prameny 35
Příloha I: Konverze na ANSI/ISO Standard C++ 35
Příloha J: Odpovědi na otázky 35

Poznámky pro lektory 35
Konvence použité v této knize 36
Systémy, v nichž se vytvářely vzorové příklady 37
Zpětná vazba od čtenářů 37
Zdrojové kódy ke knize 37
Errata 37

Kapitola 1
Začínáme 39
Učíme se C++: Na co se můžete těšit 39
Počátky C++: Trochu historie 40
Jazyk C 40
Filozofie programování v jazyce C 41
Rozšíření C++: Objektově orientované programování 42
C++ a obecné programování 43
Zrození C++ 43
Přenositelnost a standardy 45
Postup vytváření programu 47
Vytváření souboru se zdrojovým kódem 48
Překlad a sestavení 49
Překlad a sestavení pod Unixem 49
Překlad a sestavení pod Linuxem 50
Překladače pro Windows ovládané z příkazového řádku 51
Překladače pro systémy Windows 51
Jazyk C++ v operačních systémech Macintosh 53
Shrnutí 54

Kapitola 2
Vzhůru do světa C++ 55
Úvod do C++ 55
Funkce main() 57
Hlavička funkce jako rozhraní 58
Proč se funkce main() nemůže jmenovat jinak 59
Komentáře v jazyce C++ 60
Preprocesor jazyka C++ a soubor iostream 61
Jména hlavičkových souborů 61
Jmenné prostory 62
Výstup jazyka C++ pomocí cout 63
Manipulační symbol endl 65
Znak nového řádku 66
Formátování zdrojového kódu jazyka C++ 66
Symboly a bílé znaky 67
Styl zdrojového kódu jazyka C++ 67
Příkazy jazyka C++ 68
Deklarační příkazy a proměnné 68
Přiřazovací příkazy 70
Nový trik objektu cout 70
Další příkazy jazyka C++ 71
Použití objektu cin 72
Zřetězení s cout 72

cin a cout: První seznámení s třídami 73
Funkce **75**
Použití funkce, která má návratovou hodnotu 75
Varianty funkcí 78
Funkce definované uživatelem 79
Tvar funkce 80
Hlavičky funkcí 80
Jak se používá funkce definovaná uživatelem, která má návratovou hodnotu 82
Umístění direktivy using v multifunkčních programech 84
Shrnutí **86**
Otázky k opakování 86
Programátorská cvičení 87

Kapitola 3
Práce s daty **89**
Jednoduché proměnné **89**
Jména proměnných 90
Celočíselné typy 91
Celočíselné typy short, int a long 92
Poznámky k programu 95
Operátor sizeof a hlavičkový soubor climits 95
Inicializace 95
Neznaménkové typy 97
Jaký zvolit celočíselný typ? 98
Celočíselné konstanty 99
Jak C++ určuje typ konstanty 102
Typ char: Znaky a malá celá čísla 102
Poznámky k programu 104
Členská funkce: cout.put() 105
Konstanty typu char 105
Univerzální jména znaků 108
Typ signed char a unsigned char 109
Když potřebujete větší velikost: wchar_t 109
Nový typ bool 109
Kvalifikátor const **110**
Čísla s pohyblivou desetinnou čárkou **111**
Zápis čísel s pohyblivou desetinnou čárkou 112
Typy s pohyblivou desetinnou čárkou 113
Poznámky k programu 115
Konstanty s pohyblivou desetinnou čárkou 116
Výhody a nevýhody typů s pohyblivou desetinnou čárkou 116
Aritmetické operátory jazyka C++ **117**
Jaké pořadí: Priorita a asociativita operátorů 118
Odlišnosti dělení 119

Operátor modulo (zbytek po celočíselném dělení) 121
Typové konverze 122
Konverze během přiřazování 122
Konverze ve výrazech 124
Konverze při předávání argumentů 125
Přetypování 125
Shrnutí 127
Otázky k opakování 127
Programátorská cvičení 128

Kapitola 4
Složené typy 131
Úvod do polí 131
Poznámky k programu 134
Pravidla inicializace polí 135
Řetězce 136
Spojování řetězců 137
Použití řetězců v polích 138
Poznámky k programu 139
Rizika spojená se vstupem řetězce 139
Vstup řetězce z celého řádku 141
Vstup řádku pomocí getline() 141
Vstup řádku pomocí get() 143
Prázdné řádky a další problémy 144
Smíchání řetězcového a číselného vstupu 145
Úvod do třídy string 146
Přiřazování, zřetězování a připojování 147
Další operace s třídou string 148
Vstupní a výstupní operace třídy string 150
Úvod do struktur 152
Použití struktury v programu 153
Poznámky k programu 154
Může používat struktura člena třídy string? 156
Další vlastnosti struktur 156
Pole struktur 158
Bitová pole ve strukturách 159
Uniony 159
Enumerace (výčtové typy) 161
Nastavení hodnot enumerátorů 162
Rozsahy hodnot výčtového typu 163
Ukazatele a volná paměť 163
Deklarace a inicializace ukazatelů 166
Nebezpečné ukazatele 169

Ukazatele a čísla 169
Alokace paměti pomocí operátoru new 170
Poznámky k programu 171
Uvolnění paměti operátorem delete 172
Vytváření dynamických polí pomocí operátoru new 173
Vytvoření dynamického pole operátorem new 173
Použití dynamického pole 174
Ukazatele, pole a aritmetika ukazatelů 176
Poznámky k programu 177
Shrnutí vlastností ukazatelů 179
Deklarace ukazatelů 179
Přiřazování hodnot ukazatelům 179
Dereferencování ukazatelů 179
Rozlišování mezi ukazatelem a hodnotou, na kterou ukazatel ukazuje 179
Jména polí 180
Aritmetika ukazatelů 180
Dynamická a statická vazba polí 180
Zápis polí a ukazatelů 180
Ukazatele a řetězce 181
Poznámky k programu 182
Vytváření dynamických struktur pomocí new 185
Příklad použití operátoru new a delete 187
Poznámky k programu 188
Automatické, statické a volné úložiště 189
Automatické úložiště 189
Statické úložiště 189
Dynamické úložiště 189
Shrnutí 190
Otázky k opakování 191
Programátorská cvičení 192

Kapitola 5
Cykly a relační výrazy 195
Úvod do cyklu for 195
Části cyklu for 196
Výrazy a příkazy 199
Nevýrazy a příkazy 201
Přizpůsobování pravidel 201
Zpět k cyklu for 202
Poznámky k programu 203
Změna velikosti kroku 204
Procházení řetězců pomocí cyklu for 205
Operátor inkrementace (++) a dekrementace (–) 205
Vedlejší účinky a sekvenční ukazatele 206
Prefix a postfix 207

Operátory ++ / – a ukazatele 208
Sdružování přiřazovacích operátorů 209
Složené příkazy neboli bloky 209
Operátor čárka (další syntaktické triky) 211
Poznámky k programu 212
Zvláštnosti použití operátoru čárka 213
Relační výrazy 214
Chyba, kterou možná uděláte 215
Porovnávání řetězců 217
Poznámky k programu 218
Porovnávání řetězců třídy string 219
Poznámky k programu 220
Cyklus while 220
Poznámky k programu 222
Porovnání cyklů for a while 223
Okamžik – jak se dělá cyklus s časovým zpožděním 224
Cyklus do while 226
Cykly a vstup textu 229
Použití prostého objektu cin pro vstup 229
Poznámky k programu 230
Řešení je v cin.get(char) 230
Která verze cin.get()? 231
Podmínka konce souboru 232
EOF ukončuje vstup 234
Běžné postupy při vstupu znaků 234
Ještě další verze cin.get() 235
Vnořené cykly a dvojrozměrná pole 238
Inicializace dvojrozměrného pole 239
Shrnutí 241
Otázky k opakování 242
Programátorská cvičení 243

Kapitola 6
Příkazy větvení a logické operátory 245
Příkaz if 245
Příkaz if else 247
Formátování příkazů if else 249
Konstrukce if else if else 250
Logické výrazy 251
Logický operátor NEBO (OR): || 252
Logický operátor A (AND): && 253
Poznámky k programu 255
Nastavení rozsahů pomocí operátoru && 255

Poznámky k programu 257
Logický operátor NE (NOT): ! 257
Poznámky k programu 258
Další informace o logických operátorech 259
Alternativní zástupci 260
Knihovna znakových funkcí cctype **260**
Operátor ?: 262
Příkaz switch **264**
Použití výčtových typů (enumerátorů) jako návěští 267
Příkazy switch a if else 268
Příkazy break a continue **269**
Poznámky k programu 270
Načítání čísel v cyklech **271**
Poznámky k programu 274
Jednoduchý vstup a výstup do/ze souboru **274**
Textové V/V a textové soubory 275
Zápis do textového souboru 276
Poznámky k programu 279
Čtení z textového souboru 279
Poznámky k programu 283
Shrnutí **284**
Otázky k opakování **285**
Programátorská cvičení **287**

Kapitola 7
Funkce – programové moduly jazyka C++ **291**
Přehled funkcí **291**
Definování funkce 292
Vytváření funkčních prototypů a volání funkcí 294
Proč prototypy? 295
Syntaxe prototypu 296
Co pro nás prototypy dělají 297
Argumenty funkcí a předávání hodnotou **298**
Vícenásobné argumenty 299
Poznámky k programu 301
Další funkce se dvěma argumenty 301
Poznámky k programu 303
Funkce a pole **304**
Jak ukazatele umožňují funkcím zpracovávat pole 305
Důsledky použití polí jako argumentů 306
Poznámky k programu 308
Další příklady funkcí pracujících s poli 308
Naplnění pole 309

Výpis pole a jeho ochrana pomocí klíčového slova const 310
Změny v poli 311
Sestavení všech částí 311
Poznámky k programu 313
Funkce používající rozsahy polí 313
Poznámky k programu 314
Ukazatele a const 315
Funkce a dvojrozměrná pole 318
Funkce a řetězce ve stylu jazyka C 320
Řetězce ve stylu jazyka C jako argumenty funkcí 320
Poznámky k programu 321
Funkce vracející řetězce ve stylu jazyka C 322
Poznámky k programu 323
Funkce a struktury 323
Předávání a vracení struktur 324
Další příklad použití funkce se strukturami 325
Poznámky k programu 329
Předávání adres struktur 330
Funkce a objekty třídy string 332
Rekurze 333
Rekurze s jediným rekurzivním voláním 334
Rekurze s několika rekurzivními voláními 335
Poznámky k programu 336
Ukazatele na funkce 337
Základy ukazatelů na funkce 337
Získání adresy funkce 337
Deklarace ukazatele na funkci 338
Volání funkce pomocí ukazatele 339
Příklad na ukazatel na funkci 339
Shrnutí 341
Otázky k opakování 342
Programátorská cvičení 342

Kapitola 8
Rozšířené možnosti funkcí 347
Vložené (inline) funkce v C++ 347
Odkazové (referenční) proměnné 350
Vytvoření odkazové (referenční) proměnné 350
Odkazy jako funkční parametry 353
Poznámky k programu 355
Vlastnosti a zvláštnosti odkazů 356
Dočasné proměnné, odkazové argumenty a konstanty 358
Použití odkazů na struktury 360
Poznámky k programu 361

Opatrně s vracenými odkazy 362
Proč se při vracení odkazu používá const? 363
Odkazy na objekty třídy 363
Poznámky k programu 365
Další poznatky o objektech: Objekty, dědičnost a odkazy 367
Poznámky k programu 369
Kdy používat odkazové argumenty 370
Implicitní argumenty 371
Poznámky k programu 372
Přetěžování funkcí 373
Příklad přetížení 375
Kdy používat přetěžování funkcí 377
Šablony funkcí 378
Přetížené šablony 381
Explicitní specializace 383
Třetí generace specializace (standard ISO/ANSI C++) 384
Příklad na explicitní specializaci 385
Dřívější přístupy ke specializaci 386
Instance a specializace 387
Kterou verzi funkce vybere překladač? 389
Přesné a nejlepší shody 390
Příklad pravidel částečného uspořádání 392
Funkce s více parametry 394
Shrnutí 394
Otázky k opakování 395
Programátorská cvičení 396

Kapitola 9
Paměťové modely a jmenné prostory 399
Oddělený překlad 399
Doba trvání úložiště, rozsah platnosti a vazba 405
Rozsah platnosti a vazba 405
Automatická doba trvání úložiště 406
Inicializace automatických proměnných 409
Automatické proměnné a zásobník (stack) 409
Registrové proměnné 410
Proměnné se statickou dobou trvání 411
Statická doba trvání, vnější vazba 413
Poznámky k programu 414
Statická doba trvání, vnější vazba 416
Statická doba trvání, bez vazby 419
Specifikátory a kvalifikátory 421
Více o const 422

Funkce a vazba 423
Vazba jazyka 424
Schémata úložišť a dynamická alokace 425
Operátor new s umístěním 425
Poznámky k programu 428
Jmenné prostory 428
Tradiční jmenné prostory v jazyce C++ 429
Nové vlastnosti jmenných prostorů 429
Deklarace using a direktivy using 432
Direktiva using versus deklarace using 433
Další vlastnosti jmenných prostorů 435
Nepojmenované jmenné prostory 437
Příklad jmenných prostorů 438
Jmenné prostory a budoucnost 441
Shrnutí 442
Otázky k opakování 443
Programátorská cvičení 445

Kapitola 10
Objekty a třídy 449
Procedurální a objektově orientované programování 449
Abstrakce a třídy 451
Co je typ 451
Třídy v C++ 451
Řízení přístupu k členům: Veřejné nebo privátní? 456
Implementace členských funkcí třídy 456
Poznámky ke členským funkcím 459
Vložené metody 459
Který objekt používá metodu? 460
Jak se používají třídy 461
Přehled dosavadních poznatků 465
Konstruktory a destruktory třídy 466
Deklarace a definice konstruktorů 466
Použití konstruktoru 468
Implicitní konstruktor 468
Destruktory 470
Vylepšení třídy Stock 470
Hlavičkový soubor 471
Implementační soubor 471
Klientský soubor 473
Poznámky k programu 475
Členské funkce const 476
Opakování konstruktorů a destruktorů 477

Poznáváme objekty: ukazatel this 478
Pole objektů 484
Změny rozhraní a implementace 487
Třídní rozsah 488
Konstanty třídního rozsahu 489
Abstraktní datový typ 490
Shrnutí 495
Otázky k opakování 496
Programátorská cvičení 497

Kapitola 11
Práce s třídami 501
Přetěžování operátorů 502
Čas pracuje pro nás: Příklad vývoje přetížení operátoru 503
Přidání operátoru sčítání 506
Omezení přetížení 509
Další přetížené operátory 511
Představíme vám přátele 513
Jak vytváříme přátele 515
Obecný druh přítele: přetížení operátoru << 516
První verze přetížení operátoru << 517
Druhá verze přetížení operátoru << 518
Přetížené operátory: členské a nečlenské funkce 522
Další přetěžování – třída Vector 523
Použití stavové položky 530
Přetěžování aritmetických operátorů pro třídu Vector 532
Násobení 533
Další vylepšení: přetížení přetíženého operátoru 533
Komentář k implementaci 534
Použití třídy Vector na problém náhodné chůze 534
Poznámky k programu 537
Automatické konverze a přetypování tříd 538
Poznámky k programu 542
Konverzní funkce 543
Použití automatické typové konverze 546
Konverze a přátelé 548
Co můžeme doplnit při implementaci 549
Shrnutí 551
Otázky k opakování 552
Programátorská cvičení 553

Kapitola 12
Třídy a dynamické přidělování paměti 555

Dynamická paměť a třídy 556
Příklad na zopakování a statické položky tříd 556
Poznámky k programu 562
Implicitní členské funkce 564
Standardní konstruktory 564
Kopírovací konstruktory 565
Kdy se používá kopírovací konstruktor 565
Co dělá kopírovací konstruktor 566
Kdy funguje kopírovací konstruktor špatně 566
Jak odstranit chybu s explicitním kopírovacím konstruktorem 568
Operátor přiřazení 568
Kdy se použije operátor přiřazení 569
Co dělá operátor přiřazení 570
Kdy přiřazení nefunguje 570
Oprava přiřazení 570
Nová, vylepšená třída String 571
Revidovaný implicitní konstruktor 572
Porovnávací členy 573
Zpracování znaků pomocí hranatých závorek 574
Členské funkce statické třídy 575
Další přetěžování operátoru přiřazení 575
Kdy se v konstruktoru používá operátor new 581
Postřehy k vracení objektů 584
Vracení odkazu na objekt typu const 584
Vracení odkazu na nekonstantní objekt 584
Vracení objektu 585
Vracení objektu typu const 585
Používání ukazatelů na objekty 586
Znovu operátory new a delete 589
Shrnutí o ukazatelích a objektech 590
Ještě jednou new s umístěním 592
Přehled programovacích postupů 596
Přetížení operátoru << 596
Konverzní funkce 596
Třídy s konstruktory používajícími operátor new 596
Simulace fronty 597
Třída Queue 598
Rozhraní třídy Queue 599
Implementace třídy Queue 599
Metody třídy 601
Další metody třídy? 605
Třída Customer 607
Simulace 610

Shrnutí 614
Otázky k opakování 616
Programátorská cvičení 617

Kapitola 13
Dědičnost tříd 621

Začneme jednoduchou základní třídou 622
Odvození třídy 624
Konstruktory: Úvahy o přístupu 625
Použití odvozené třídy 628
Zvláštní vztah mezi odvozenou a základní třídou 630
Dědičnost: Vztah je 632
Polymorfní veřejná dědičnost 633
Jak vytvoříme třídy Brass a BrassPlus 634
Implementace třídy 637
Použití tříd Brass a BrassPlus 642
Jak se chová virtuální metoda 643
Virtuální destruktory jsou nutné 645
Statická a dynamická vazba 646
Kompatibilita typu ukazatele a odkazu 646
Virtuální členské funkce a dynamická vazba 647
Proč dva druhy vazeb a proč je statická vazba implicitní 648
Jak pracují virtuální funkce 648
Co bychom měli vědět o virtuálních funkcích 650
Konstruktory 650
Destruktory 650
Přátelé 651
Neprovedení předefinování 651
Předefinování skrývá metody 651
Řízení přístupu: protected 653
Abstraktní základní třídy 655
Aplikace koncepce AZT 657
Filozofie AZT 661
Dědičnost a dynamické přidělování paměti 662
Případ 1: Odvozená třída nepoužívá new 662
Případ 2: Odvozená třída používá new 663
Příklad na dědičnost s dynamickým přidělováním paměti a s přáteli 665
Opakování návrhu tříd 669
Členské funkce generované překladačem 670
Implicitní konstruktor 670
Kopírovací konstruktor 670
Operátor přiřazení 671
Další metody třídy 671

Poznámky ke konstruktorům 671
Poznámky k destruktorům 671
Poznámky ke konverzím 671
Předávání objektu hodnotou a předávání reference 672
Vrácení objektu a vrácení reference 673
Použití const 673
Poznámky k veřejné dědičnosti 674
Vztah je 674
Co není součástí dědictví 674
Operátor přiřazení 675
Privátní a chráněné členy 676
Virtuální metody 677
Destruktory 677
Přátelé 677
Poznámky k používání metod základní třídy 678
Shrnutí funkcí třídy 678
Shrnutí 679
Otázky k opakování 680
Cvičení 681

Kapitola 14
Opakované používání kódu v C++ 685
Třídy, jejichž členy jsou objekty (objektové členy) 685
Třída valarray: Zběžné seznámení 686
Návrh třídy Student 687
Příklad třídy Student 688
Inicializace komponovaných objektů 690
Použití rozhraní pro komponovaný objekt 691
Použití nové třídy Student 694
Soukromá dědičnost 695
Příklad třídy Student (nová verze) 696
Inicializace komponent základní třídy 696
Přístup k metodám základní třídy 697
Přístup k objektům základní třídy 698
Přístup k přátelům základní třídy 699
Použití upravené třídy Student 701
Kompozice nebo soukromá dědičnost? 703
Chráněná dědičnost 703
Předefinování přístupu pomocí using 704
Vícenásobná dědičnost 705
Kolik tříd Worker? 710
Virtuální základní třídy 711
Nová pravidla pro konstruktory 712
Která metoda? 714
Smíšené virtuální a nevirtuální základní třídy 722
Virtuální základní třídy a dominance 723

Přehled vícenásobné dědičnosti 724
Šablony tříd 724
Definice šablony třídy 725
Použití šablony třídy 728
Bližší pohled na šablonu třídy 730
Nesprávné použití zásobníku ukazatelů 731
Správné použití zásobníku ukazatelů 732
Poznámky k programu 736
Příklad šablony pole a netypové parametry 736
Univerzálnost šablon 738
Rekurzívní použití šablon 738
Šablony s několika typy parametrů 740
Implicitní typové parametry šablon 741
Specializace šablon 742
Implicitní instance 742
Explicitní instance 742
Explicitní specializace 742
Částečné specializace 743
Členské šablony 744
Šablony jako parametry 747
Šablonové třídy a přátelé 748
Nešablonové přátelské funkce šablonových tříd 749
Vázané šablonové přátelské funkce šablonových tříd 751
Nevázané šablonové přátelské funkce šablonových tříd 754
Shrnutí 755
Otázky k opakování 757
Programátorská cvičení 758

Kapitola 15
Přátelé, výjimky a další 765
Přátelé 765
Přátelské třídy 765
Přátelské členské funkce 770
Jiné přátelské vztahy 774
Sdílení přátelé 774
Vnořené třídy 775
Vnořené třídy a přístup k nim 777
Rozsah platnosti 777
Řízení přístupu 778
Vnořování do šablony 778
Výjimky 782
Volání abort() 782
Vracení chybového kódu 783
Poznámky k programu 784

Mechanismus výjimek 785
Poznámky k programu 786
Objekty jako výjimky 788
Uvolnění zásobníku 792
Poznámky k programu 796
Další možnosti 797
Třída exception 799
Třídy výjimek stdexcept 800
Výjimka bad_alloc a příkaz new 802
Výjimky, třídy a dědičnost 803
Když výjimky bloudí 808
Opatření při používání výjimek 811
RTTI **813**
K čemu je dobré 813
Jak pracuje 813
Operátor dynamic_cast 813
Operátor typeid a třída type_info 818
Špatné použití RTTI 820
Operátory přetypování 821
Shrnutí **824**
Otázky k opakování **825**
Programátorská cvičení **826**

Kapitola 16
Třída string a standardní knihovna šablon **829**
Třída string **829**
Vytvoření řetězce 830
Poznámky k programu 832
Vstup třídy string 833
Práce s řetězci 836
Poznámky k programu 840
Co může třída string ještě nabídnout 841
Třída auto_ptr **843**
Použití třídy auto_ptr 844
Poznámky ke třídě auto_ptr 846
Standardní knihovna šablon **848**
Šablonová třída vector 848
Co se dá dělat s vektory 851
Další možnosti práce s vektory 855
Generické programování **859**
K čemu jsou potřeba iterátory 860
Druhy iterátorů 863
Vstupní iterátor 864

Výstupní iterátor 865
Dopředný iterátor 865
Obousměrný iterátor 865
Iterátor přímého přístupu 865
Hierarchie iterátorů 866
Koncepty, vylepšení a modely 867
Ukazatel jako iterátor 867
copy(), ostream_iterator a istream_iterator 868
Ostatní užitečné iterátory 869
Druhy kontejnerů 873
Koncepty kontejnerů 874
Posloupnosti 876
vector 877
deque 878
list 879
Poznámky k programu 881
Sada nástrojů třídy list 881
queue 881
priority_queue 882
stack 882
Asociativní kontejnery 883
Příklad s kontejnerem set 883
Příklad s kontejnerem multimap 887
Funkční objekty (jinak také funktory) 889
Koncepty funktorů 890
Předdefinované funktory 892
Přizpůsobivé funktory a funkční adaptéry 894
Algoritmy 896
Skupiny algoritmů 897
Obecné vlastnosti algoritmů 898
STL a třída string 899
Funkce a kontejnerové metody 900
Použití standardní knihovny šablon 902
Ostatní knihovny 905
vector a valarray 906
Shrnutí 911
Otázky k opakování 913
Programátorská cvičení 914

Kapitola 17
Vstupy, výstupy a soubory 917
Přehled vstupů a výstupů v C++ 918
Toky a vyrovnávací paměti 918
Toky, vyrovnávací paměti a soubor iostream 920

Přesměrování 922
Výstup pomocí objektu cout 924
Přetížený operátor << 924
Výstup a ukazatele 925
Řetězení výstupu 926
Ostatní metody třídy ostream 926
Vyprázdnění výstupní vyrovnávací paměti 929
Formátování pomocí cout 930
Změna číselného základu používaného při výstupu 932
Úprava šířky polí 933
Vyplňovací znaky 935
Nastavení přesnosti výpisu s pohyblivou řádovou čárkou 936
Tisk koncových nul a desetinné tečky 937
Další informace o metodě setf() 938
Standardní manipulátory 943
Hlavičkový soubor iomanip 944
Vstup pomocí cin 946
cin >> a vstup 948
Stavy toků 950
Nastavení stavů 951
V/V a výjimky 951
Účinky tokových stavů 952
Další metody třídy istream 954
Jednoznakový vstup 954
Který tvar použít pro vstup znaku 957
Vstup řetězce: metody getline(), get() a ignore() 957
Neočekávaný vstup řetězce 960
Další metody třídy istream 961
Poznámky k programu 963
Souborové V/V operace 964
Jednoduché souborové V/V operace 965
Kontrola toku a metoda is_open() 968
Otevření více souborů 969
Zpracování příkazového řádku 970
Režimy souborů 972
Přidání dat do souboru 975
Binární soubory 977
Přímý přístup 982
Formátování incore 990
Co dál? 992
Shrnutí 993
Otázky k opakování 994
Programátorská cvičení 995

Kapitola 18

Nový standard C++ 999

Opakování funkcí jazyka C++11 999

Nové typy 999

Jednotná inicializace 999

Zúžení 1000

Deklarace 1001

Návratový typ 1002

Aliasy šablon: using = 1002

Chytré ukazatele 1003

Změny ve specifikaci výjimek 1003

Rozsahové výčty 1004

Změny tříd 1004

Konverzní operátory explicit 1004

Inicializace členů ve třídě 1005

Změny šablon a knihovny STL 1005

Smyčka for s rozsahem 1005

Nové kontejnery STL 1006

Nové metody STL 1006

Upgrade valarray 1006

export 1006

Úhlové závorky 1007

Odkaz r-hodnoty 1007

Přesunová sémantika a odkazy r-hodnot 1008

Proč potřebujeme přesunovou sémantiku 1008

Příklad přesunu 1009

Sledujeme přesunovací konstruktor 1014

Přiřazení 1016

Vynucení přesunu 1016

Nové funkce tříd 1020

Zvláštní funkce členů 1021

Implicitní a odstraněné metody 1021

Delegování konstruktorů 1023

Dědění konstruktorů 1023

Správa virtuálních metod: override a final 1025

Lambda funkce 1026

Jak fungují ukazatele, funktory a lambdy 1026

Proč lambdy 1029

Obálky 1032

Obálka function a nízký výkon šablon 1032

Oprava problému 1034

Další volby 1036

Variadické šablony 1037

Balíčky parametrů šablon a funkcí 1037

Rozbalování balíčků 1038
Rekurze a funkce variadických šablon 1039
Poznámky k programu 1040
Vylepšení 1040
Další funkce jazyka C++11 1042
Souběžné zpracování 1042
Lepší knihovny 1042
Nízkoúrovňové programování 1043
Různé 1044
Změna jazyka 1044
Projekt Boost 1045
TR1 1045
Projekt Boost v praxi 1045
Co nyní? 1046
Shrnutí 1047
Otázky k opakování 1048
Programátorská cvičení 1051

Příloha A
Základy číselných soustav 1053
Desítková čísla (základ 10) 1053
Osmičková celá čísla (základ 8) 1054
Šestnáctková čísla (základ 16) 1054
Dvojková čísla (základ 2) 1055
Dvojková a šestnáctková soustava 1055

Příloha B
Rezervovaná slova v C++ 1057
Klíčová slova C++ 1057
Alternativní symboly 1058
Jména rezervovaná pro knihovny C++ 1059

Příloha C
Znaková sada ASCII 1061

Příloha D
Priorita operátorů 1067

Příloha E

Ostatní operátory 1071

Bitové operátory 1071
Operátory posunu 1071
Bitové logické operátory 1073
Alternativní tvary bitových operátorů 1075
Několik běžných technik s bitovými operátory 1075
Nastavení bitu na 1 1076
Přepnutí bitu 1076
Nulování bitu 1076
Test hodnoty bitu 1076
Operátory dereferencování členů 1077

Příloha F

Šablonová třída string 1081

Třináct typů a jedna konstanta 1082
Datové informace, konstruktory a další 1082
Implicitní konstruktor 1084
Konstruktor používající pole 1085
Konstruktor používající část pole 1085
Kopírovací konstruktor 1086
Konstruktor používající n kopií znaku 1086
Konstruktor pracující s rozsahem 1087
Různé způsoby práce s pamětí 1087
Přístup k řetězci 1088
Základní přiřazení 1089
Prohledávání řetězce 1089
Skupina metod find() 1089
Skupina metod rfind() 1090
Skupina metod find_first_of() 1090
Skupina metod find_last_of() 1090
Skupina metod find_first_not_of() 1091
Skupina metod find_last_not_of() 1091
Metody a funkce pro porovnání řetězců 1092
Řetězcové modifikátory 1093
Připojení a přidání 1093
Další přiřazení 1094
Metody vkládání 1094
Metody pro odstraňování znaků 1094
Metody pro nahrazování 1095
Další upravující metody: copy() a swap() 1095
Výstup a vstup 1096

Příloha G

Metody a funkce knihovny STL 1097

Položky společné všem kontejnerům 1097
Další položky kontejnerů vector, list a deque 1100
Další položky kontejnerů set a map 1102
Funkce knihovny STL 1103

Nemodifikující sekvenční operace 1104
- for_each() 1105
- find() 1105
- find_if() 1105
- find_end() 1105
- find_first_of() 1105
- adjacent_find() 1106
- count() 1106
- count_if() 1106
- mismatch() 1106
- equal() 1107
- search() 1107
- search_n() 1107

Změnové sekvenční operace 1108
- copy() 1109
- copy_backward() 1109
- swap() 1110
- swap_ranges() 1110
- iter_swap() 1110
- transform() 1110
- replace() 1110
- replace_if() 1111
- replace_copy() 1111
- replace_copy_if() 1111
- fill() 1111
- fill_n() 1111
- generate() 1111
- generate_n() 1111
- remove() 1112
- remove_if() 1112
- remove_copy() 1112
- remove_copy_if() 1112
- unique() 1112
- unique_copy() 1113
- reverse() 1113
- reverse_copy() 1113
- rotate() 1113
- rotate_copy() 1113
- random_shuffle() 1114
- partition() 1114
- stable_partition() 1114

Třídicí a relační operace 1114
Třídění 1116
sort() 1116
stable_sort() 1117
partial_sort() 1117
partial_sort_copy() 1117
nth_element() 1117
Binární hledání 1118
lower_bound() 1118
upper_bound() 1118
equal_range() 1118
binary_search() 1119
Slučování 1119
merge() 1119
inplace_merge() 1119
Množinové operace 1120
includes() 1120
set_union() 1120
set_intersection() 1120
set_difference() 1121
set_symmetric_difference() 1121
Operace pracující s haldou 1122
make_heap() 1122
push_heap() 1122
pop_heap() 1122
sort_heap() 1122
Hledání minimálních a maximálních hodnot 1123
min() 1123
max() 1123
min_element() 1123
max_element() 1123
lexicographical_compare() 1123
Změny pořadí 1124
next_permutation() 1124
prev_permutation() 1124
Numerické operace 1125
accumulate() 1125
inner_product() 1125
partial_sum() 1126
adjacent_difference() 1126

Příloha H
Vybraná literatura a internetové prameny 1127
Vybraná literatura 1127
Internetové prameny 1129

Příloha I

Konverze na ANSI/ISO standard C++ 1131

Používejte alternativní direktivy preprocesoru 1131

Konstanty #define nahraďte definicemi const 1131

Definice krátkých funkcí #define nahraďte definicemi inline 1133

Používání funkčních prototypů 1134

Přetypování 1134

Seznamte se s vlastnostmi C++ 1135

Používejte nové uspořádání hlavičkových souborů 1135

Používejte jmenné prostory 1135

Používejte šablonu autoptr 1136

Používejte třídu string 1136

Používejte knihovnu STL 1137

Příloha J

Odpovědi na otázky 1139

Odpovědi na otázky v kapitole 2 1139

Odpovědi na otázky v kapitole 3 1140

Odpovědi na otázky v kapitole 4 1140

Odpovědi na otázky v kapitole 5 1142

Odpovědi na otázky v kapitole 6 1143

Odpovědi na otázky v kapitole 7 1144

Odpovědi na otázky v kapitole 8 1146

Odpovědi na otázky v kapitole 9 1147

Odpovědi na otázky v kapitole 10 1148

Odpovědi na otázky v kapitole 11 1150

Odpovědi na otázky v kapitole 12 1151

Odpovědi na otázky v kapitole 13 1154

Odpovědi na otázky v kapitole 14 1155

Odpovědi na otázky v kapitole 15 1156

Odpovědi na otázky v kapitole 16 1158

Odpovědi na otázky v kapitole 17 1159

Odpovědi na otázky v kapitole 18 1161

Rejstřík 1165

Úvod

V této kapitole najdete:

- Předmluvu ke čtvrtému vydání
- Základní přístup
- Vzory kódu
- Uspořádání knihy
- Poznámky pro lektory
- Konvence použité v této knize
- Systémy, v nichž se vytvářely vzorové příklady
- Poznámku redakce českého vydání

Předmluva ke čtvrtému vydání

Studium C++ je objevné dobrodružství, zejména proto, že tento jazyk se přizpůsobuje několika programátorským přístupům, k nimž patří objektově orientované programování, generické programování a tradiční procedurální programování. Když se jazyk C++ rozšiřoval o nové vlastnosti, byl pohyblivým cílem, nyní však se již stal jazykem stabilizovaným. Současné překladače podporují většinu vlastností vyžadovaných standardem a programátoři měli dostatek času k tomu, aby je ověřili v praxi.

Kniha *Mistrovství v C++* popisuje základní jazyk C a uvádí vlastnosti C++, což přispívá k její celistvosti. Uvádějí se v ní základy C++, jež jsou názorně objasněny na krátkých příkladech, které se snadno dají zkopírovat a lze s nimi dále experimentovat. Naučíte se pracovat se vstupy a výstupy (V/V), programovat opakované úlohy a rozhodnutí, seznámíte se s mnoha způsoby zpracování dat a používání funkcí. Dozvíte se o tom, které nové vlastnosti má jazyk C++ oproti jazyku C, k nimž mimo jiné patří:

- Třídy a objekty
- Dědičnost
- Polymorfie, virtuální funkce a identifikace v době běhu (*runtime type identification*, RTTI)
- Přetěžování funkcí
- Referenční proměnné
- Generické neboli typově nezávislé programování s využitím šablon a Standardní knihovny šablon (*Standard Template Library*, STL)
- Mechanismus výjimek pro zpracování chybových podmínek
- Jmenné prostory pro zpracování jmen funkcí, tříd a proměnných

Základní přístup

Mistrovství v C++ přináší několik vylepšení v prezentaci materiálů a zahrnuje osvědčenou filozofii:

- Kniha by se měla snadno používat a měla by být přátelským průvodcem.
- Kniha nepředpokládá, že byste byli obeznámeni se všemi potřebnými programovacími koncepcemi.
- Kniha chce být pomocníkem, který je neustále po ruce a uvádí stručné, jednoduché příklady, které se snaží v daném okamžiku vysvětlit jeden nebo nanejvýš dva principy.
- K vysvětlování různých koncepcí se v knize používají ilustrace.
- Kniha obsahuje otázky a cvičení, jimž lze zjistit, jak jste zvládli látku, což je vhodné jak pro samouky, tak i pro výuku v učebně.

V souladu s těmito principy se kniha snaží pomoci jednak jazyk zvládnout, a také se jej naučit používat. Například:

- Poskytuje koncepční návody k používání různých možností jazyka, např. kdy používat veřejnou dědičnost k modelu, který je známý jako vztah „*je*“.
- Osvětluje běžné programátorské styly a postupy.
- Obsahuje řadu poznámek jako např. tipy, upozornění, věci k zapamatování, poznámky o kompatibilitě a poznámky o reálném světě.

Autor i redaktoři této knihy se velice snažili, aby text knihy byl k věci, byl čtivý a byl i zábavný. Naším cílem bylo, abyste po jejím přečtení byli schopni psát solidní a efektivní programy a také abyste z toho měli radost.

Vzory kódu

V knize naleznete hojnost vzorového kódu, většinou v podobě kompletních programů. Podobně jako v předchozím vydání používáme obecný jazyk C++ bez vazby na určitý počítač, operační systém nebo překladač. Příklady byly testovány na systémech Windows XP, Macintosh OS X a Linux. Jen velmi málo programů bylo nekonformních s některým překladačem. Shoda kompilátorů se standardem C++ se od předchozích vydání této knihy značně vylepšila. Kódy programů popsaných v této knize jsou k dispozici na internetových stránkách našeho nakladatelství (*http://knihy.cpress.cz/K2080*).

Uspořádání knihy

Kniha je rozdělena do 18 kapitol a 10 příloh. Zde je jejich přehled.

Kapitola 1: Začínáme

Kapitola 1 podává zprávu o tom, jak Bjarne Stroustrup vytvořil z programovacího jazyka C jazyk C++ tím, že do něj přidal podporu objektově orientovaného programování. Dozvíte se o rozdílech mezi procedurálními jazyky typu C a objektově orientovanými

jazyky, např. C++. Přečtete si o zapojení ANSI/ISO do vývoje standardu C++. V kapitole se dále popisují postupy při vytváření programů v C++ a uvádějí se některé rozdíly mezi překladači C++ používanými v současnosti. Nakonec jsou zde popsány konvence používané v této knize.

Kapitola 2: Vzhůru do světa C++

Kapitola 2 vás bude provázet procesem vytváření jednoduchých programů v C++. Dozvíte se o úloze funkce `main()` a o některých typech příkazů v C++. Pro vstupní a výstupní operace budete používat předdefinované objekty `cin` a `cout` a také se naučíte vytvářet a používat proměnné. Nakonec se seznámíte s funkcemi, což jsou programovací moduly v C++.

Kapitola 3: Práce s daty

Pro ukládání dvou typů dat: celých čísel (tj. čísel bez zlomkové části) a čísel v pohyblivé řádové čárce (čísel se zlomkovou částí) používá jazyk C++ vestavěné typy. Pro uspokojení rozličných požadavků nabízí C++ několik typů v každé kategorii. Kapitola 3 uvádí popis těchto typů, vytváření proměnných a zápis konstant různých typů. Také se dozvíte, jak se v C++ provádějí explicitní a implicitní konverze mezi jednotlivými typy.

Kapitola 4: Složené typy

V C++ si můžete ze základních vestavěných typů vytvářet vlastní, složitější typy. Nejvýhodnějším tvarem jsou třídy, které jsou popsány v kapitolách 9 až 13. V této kapitole se dozvíte o jiných formách – mimo jiné o polích, v nichž je uloženo více hodnot stejného typu; strukturách, v nichž je uloženo více hodnot různého typu; a ukazatelích, které ukazují na místo v paměti. Také se zde naučíte vytvářet a ukládat textové řetězce a používat V/V operace, které pracují se znakovými poli typu C a s řetězci typu C++. Nakonec se dozvíte něco o tom, jak se v C++ provádí explicitní správa paměti pomocí operátorů `new` a `delete`.

Kapitola 5: Cykly a relační výrazy

Programy musí často provádět opakované akce. V C++ existují tři struktury pro vytváření cyklů: cyklus typu `for`, typu `while` a typu `do while`. Cykly musí vědět, kdy se mají ukončit, přičemž testování se provádí pomocí relačních operátorů. V kapitole 5 se naučíte vytvářet cykly, které čtou a zpracovávají vstupy znak po znaku. Nakonec se naučíte vytvářet dvourozměrná pole a budete je zpracovávat pomocí vnořených cyklů.

Kapitola 6: Příkazy větvení a logické operátory

Pokud umíte ušít programy na míru, umějí se chovat inteligentně. V kapitole 6 se naučíte řídit programový tok pomocí příkazů `if`, `if else` a `switch` a pomocí podmíněného operátoru. Dozvíte se, jak využívat logické operátory při rozhodovacích testech. Také se setkáte s knihovnou funkcí `cctype` pro vyhodnocování vztahů mezi znaky, např. je-li znak písmeno nebo číslice nebo netisknutelný znak. Nakonec se seznámíte s úvodem do souborových V/V operací.

Kapitola 7: Funkce – programové moduly jazyka C++

Funkce jsou základními stavebními kameny programování v C++. Ústředním tématem kapitoly 7 je sdílení funkcí v C a v C++. Konkrétně si zopakujete obecný tvar definice funkce a vyzkoušíte si, jak lze pomocí prototypů funkcí zvýšit spolehlivost programů. Také se naučíte, jak se mají psát funkce na zpracování polí, znakových řetězců a struktur. Dále si řekneme něco o rekurzi, což znamená, že funkce volá sama sebe, a dozvíte se, jak lze využít rekurze k implementaci strategie rozděl a panuj. Nakonec se seznámíte s ukazateli funkcí, jejichž prostřednictvím lze použít argument funkce k předání informace o tom, jak jedna funkce může použít jinou funkci.

Kapitola 8: Rozšířené možnosti funkcí

Kapitola 8 zkoumá, jaké jsou nové vlastnosti funkcí v C++. Získáte poznatky o vložených (inline) funkcích, které mohou urychlit chod programu za cenu jeho zvětšení. Budete pracovat s referenčními proměnnými, jež poskytují alternativní způsob předávání informací funkcím. Implicitní argumenty automaticky poskytují funkci hodnoty argumentů, jež neuvádíte ve volání funkce. Pomocí přetěžování funkcí můžete vytvářet funkce se stejným jménem, avšak s jinými parametry. Tyto vlastnosti se hojně využívají při psaní tříd. Také se naučíte používat šablony funkcí, jejichž prostřednictvím lze specifikovat návrhy rodin obsahující příbuzné funkce.

Kapitola 9: Paměťové modely a jmenné prostory

V kapitole 9 se budeme věnovat programům, které se skládají z několika souborů. Vyzkoušíte si, jaké jsou možnosti přidělování paměti, seznámíte se s různými způsoby správy paměti a popíšeme si rozsah platnosti, vazbu a jmenné prostory, jež určují, které části programů vědí o dané proměnné.

Kapitola 10: Objekty a třídy

Třída je typ definovaný uživatelem a objekt (např. proměnná) je instancí třídy. V kapitole 10 se seznámíte s objektově orientovaným programováním a s návrhy tříd. Deklarace třídy popisuje data uložená v objektu typu třída a taktéž operace (nazývané metody) přípustné pro objekty typu třída. Některé části objektu jsou viditelné z vnějšího světa (veřejná část), některé jsou skryté (privátní část). Při vytváření a rušení tříd budou ve hře speciální metody tříd (konstruktory a destruktory). Také se v této kapitole seznámíte s dalšími podrobnostmi o třídách a uvidíte, jak lze pomocí tříd implementovat abstraktní datové typy (ADT), například zásobníky.

Kapitola 11: Práce s třídami

V kapitole 11 si prohloubíte znalosti o třídách. Nejdříve si řekneme něco o přetěžování operátorů, což nám umožní pracovat s třídami například pomocí operátorů + apod. Seznámíte se s přátelskými funkcemi, jež mají přístup k takovým datům ve třídě, která jsou pro ostatní svět nedostupná. Uvidíte, jak lze využít určité konstruktory a přetížené operátory členských funkcí ke správě konverzí do a z typů třída.

Kapitola 12: Třídy a dynamické přidělování paměti

Občas je vhodné mít ukazatele na člena třídy v dynamicky přidělované paměti. Použijete-li v konstruktoru třídy k přidělení paměti příkaz `new`, přebíráte zodpovědnost za vytvoření odpovídajícího destruktoru, definici explicitního kopírovacího konstruktoru a definici explicitního přiřazovacího operátoru. Kapitola 12 ukazuje, jak se toto provádí a popisuje chování implicitně vygenerované členské funkce, když takovou funkci nevygenerujete explicitně. Při analýze problému se simulací front s využitím ukazatelů na objekty získáte další zkušenosti s třídami.

Kapitola 13: Dědičnost tříd

Jedním z nejsilnějších nástrojů objektově orientovaného programování je dědičnost, při níž odvozená třída dědí vlastnosti základní třídy, což umožňuje opakované použití kódu základní třídy. V kapitole 13 je popsána veřejná dědičnost s modelem vztahu *je* v tom smyslu, že odvozený objekt je zvláštním případem základního objektu. Fyzik je například zvláštním případem vědce. Některé dědické vztahy jsou polymorfní, což znamená, že můžete psát kód pomocí směšování příbuzných tříd, pro něž stejné jméno metody může vyvolat chování, jež závisí na typu objektu. Implementace tohoto typu chování s sebou přináší nutnost používání nového druhu členských funkcí, jež se nazývají virtuální funkce. Někdy se použití abstraktních základních tříd jeví jako nejlepší přístup k dědickým vztahům. O těchto otázkách a o tom, kdy je vhodné použít dědičnost, se dočtete právě v této kapitole.

Kapitola 14: Opakované používání kódu v C++

Veřejná dědičnost je pouze jedním ze způsobů opakovaného použití kódu. V kapitole 14 se dozvíte i o jiných způsobech. Kompozice je stav, kdy jedna třída obsahuje členy, jež jsou objekty jiné třídy. Kompozice lze využít k modelování vztahu *má*, kde jedna třída má komponenty jiné třídy. Například automobil má motor. Tento vztah můžete modelovat také pomocí privátní a chráněné dědičnosti. V této kapitole je popsáno, jakým způsobem, a oba přístupy se porovnávají. Také se dozvíte něco o šablonách tříd, jejichž prostřednictvím můžete definovat v pojmech nespecifikovaného obecného typu, a pak tuto šablonu používat k vytváření specifických tříd v pojmech specifických typů. Například šablona zásobníku umožňuje vytvořit zásobník celých čísel nebo zásobník řetězců. Nakonec se zmíníme o vícenásobné veřejné dědičnosti, jež umožňuje odvozování třídy z několika tříd.

Kapitola 15: Přátelé, výjimky a další

Kapitola 15 rozšiřuje přátele o přátelské třídy a přátelské členské funkce. Poté uvádí některé novinky v C++ počínaje výjimkami, které poskytují mechanismy zpracování neobvyklých situací v programu, např. nesprávná hodnota argumentu funkce nebo adresování mimo paměť. Poté se dozvíte něco o RTTI, což je způsob identifikace typů objektů. Nakonec si popíšeme bezpečnější alternativy povoleného přetypování.

Kapitola 16: Třída string a standardní knihovna šablon

V kapitole 16 se dozvíte o některých novinkách v oblasti knihoven tříd, o něž byl jazyk C++ doplněn. Třída typu `string` je vhodnou a silnou náhradou řetězců typu C. Třída `auto_ptr` je pomocníkem při správě dynamicky přidělované paměti. Knihovna STL obsahuje několik obecných kontejnerů včetně šablon polí, front, seznamů, množin a map. Taktéž obsahuje rozsáhlou knihovnu obecných algoritmů, jež lze využívat s kontejnery a taktéž i s obyčejnými poli. Šablona `valarray` poskytuje podporu číselných polí.

Kapitola 17: Vstupy, výstupy a soubory

Kapitola 17 podává přehled o vstupech a výstupech v C++ a o jejich formátování. K určování stavu vstupních a výstupních toků budete využívat metody tříd a naučíte se zjišťovat, zda byl vstup ukončen vinou chybných dat, nebo se narazilo na konec souboru. Ke správě vstupů a výstupů v C++ se používají třídy odvozené prostřednictvím dědičnosti. Naučíte se otevírat soubory pro vstup a výstup, připojovat data na konec souboru, používat binární soubory a využívat přímého přístupu k souborům. Nakonec se naučíte aplikovat standardní V/V metody pro čtení a zápis řetězců.

Kapitola 18: Nový standard C++

Poslední kapitola se zaměřuje na změny v jazyce C++ ve verzi C++11. Zopakujete si některé již probrané funkce a podíváte se na další. Seznámíte se s novinkami, které přináší nová verze jazyka a také s doplňky, na které byste měli zaměřit svoje další studium.

Příloha A: Základy číselných soustav

Příloha A popisuje oktalová (osmičková), hexadecimální (šestnáctková) a binární (dvojková) čísla.

Příloha B: Rezervovaná slova v C++

Příloha B obsahuje seznam klíčových slov C++.

Příloha C: Znaková sada ASCII

Příloha C obsahuje množinu znaků ASCII s jejich oktalovou, hexadecimální a binární reprezentací.

Příloha D: Priorita operátorů

Příloha D obsahuje seznam operátorů seřazený od nejvyšší k nejnižší prioritě.

Příloha E: Ostatní operátory

Příloha E shrnuje operátory C++ včetně operátorů bitových, jež nejsou popsány v hlavním textu.

Příloha F: Šablonová třída string

Příloha F podává přehled o řetězcových metodách a funkcích.

Příloha G: Metody a funkce knihovny STL

Příloha G obsahuje přehled kompozičních metod a obecných algoritmů funkcí STL.

Příloha H: Vybraná literatura a internetové prameny

Příloha H uvádí seznam některých knih, které mohou posloužit k prohloubení znalostí o C++.

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

Příloha I je průvodcem pro přechod od C a starších implementací C++ k ANSI/ISO C++.

Příloha J: Odpovědi na otázky

Příloha J obsahuje odpovědi k opakovacím otázkám, které jsou uváděny na konci každé kapitoly.

Poznámky pro lektory

Jedním z cílů tohoto vydání knihy *Mistrovství v C++* je poskytnout knihu, kterou by mohli používat jak samouci, tak i lektoři. Zde jsou některé možnosti, jak lze knihu využít při výuce:

- Kniha popisuje obecný jazyk C++ nezávislý na konkrétní implementaci.
- Obsah reflektuje práci výboru pro standardy ISO/ANSI C++ včetně diskuse o šablonách, STL, řetězcových třídách, výjimkách, RTTI a jmenných prostorech.
- Podmínkou pro zvládnutí knihy není žádná předchozí znalost jazyka C, předpokladem jsou pouze všeobecné vědomosti o programování.
- Témata jsou nicméně uspořádána tak, že ti, kdo znají jazyk C, zvládnou úvodní kapitoly poměrně rychle jako opakovací kapitoly.
- Kapitoly obsahují opakovací otázky a programátorská cvičení. Odpovědi na tyto otázky jsou uvedeny v příloze J. Řešení některých vybraných programátorských cvičení naleznete na stránkách našeho nakladatelství (*http://knihy.cpress.cz*).
- Kniha uvádí několik témat, která jsou vhodná i pro přednášky o informačních technologiích, např. abstraktní datové typy (ADT), zásobníky, fronty, jednoduché seznamy, simulace, generické programování a používání rekurze k implementaci strategie rozděl a panuj.
- Většinu kapitol je možné zvládnout zhruba za týden.
- Určité vlastnosti jsou v knize popisovány tak, že se říká *kdy* se mají používat a také *jak* se mají používat. Například veřejná dědičnost je spojena se vztahem *je* a kompozice a soukromá dědičnost je spojena se vztahem *má*, a dále jsou uvedena doporučení, kdy se mají a kdy se nemají používat virtuální funkce.

Konvence použité v této knize

V této knize používáme pro rozlišení různých typů textů některé typografické konvence:

- Řádky s kódem, povely, příkazy, proměnné, jména souborů a programové výstupy uvádíme neproporcionálním typem písma:

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

- Výplňový text v syntaxi je uveden *neproporcionální kurzívou*. Nahrazuje se skutečným jménem souboru, parametrem nebo jiným prvkem, který představuje.
- Pro nové pojmy se používá *kurzíva*.

V knize jsou také použity prvky, které slouží k osvětlení určité problematiky:

KOMPATIBILITA

Většina překladačů ještě není 100% kompatibilní se standardem ISO/ANSI, můžete tedy narazit na určité nesrovnalosti.

ZAPAMATUJTE SI

Tyto poznámky zvýrazňují témata, která je důležité si zapamatovat.

Z praxe

Poznámky profesionálních programátorů, jež vycházejí z jejich zkušeností.

Označení problému

Podrobnější popis problému nebo jeho dopadu.

TIP

Upozornění na určitou situaci v programu

UPOZORNĚNÍ

Upozornění na určitá úskalí v dané situaci

POZNÁMKA

Ostatní poznámky, které nespadají do žádné z předchozích kategorií

Systémy, v nichž se vytvářely vzorové příklady

Příklady v této knize byly vytvořeny v Microsoft Visual C++ 7.1 (verze, která je dodávaná k Microsoft Visual Studio .NET 2003) a Metrowerks CodeWarrior Development Studio 9 na osobním počítači s procesorem Pentium s pevným diskem vybaveným operačním systémem Windows XP Professional. Většina programů byla také ověřena na řádkovém kompilátoru Borland C++ 5.5 a GNU gpp 3.3.3 na tomtéž systému, s použitím Comeau 4.3.3 a GNU g++ 3.3.1 na osobním počítači s procesorem Pentium kompatibilním s IBM s operačním systémem SuSE 9.0 Linux a Metrowerks Development Studio 9 na počítači Macintosh G4 pod OS 10.3. V knize upozorňujeme na nesrovnalosti, které plynou z opožďování překladačů za standardem, například „starší implementace používají `ios::fixed` místo `ios_base::fixed`". Zmiňujeme se i o některých chybách a výstřednostech ve starších překladačích, jež by mohly způsobit problémy nebo zmatky; většina z nich byla v nejnovějších verzích odstraněna.

C++ nabízí programátorům opravdu mnoho; učte se a radujte se!

Zpětná vazba od čtenářů

Nakladatelství a vydavatelství Computer Press, které pro vás tuto knihu přeložilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

Computer Press
Albatros Media a.s., pobočka Brno
IBC
Příkop 4
602 00 Brno

nebo

sefredaktor.pc@albatrosmedia.cz

Computer Press neposkytuje rady ani jakýkoli servis pro aplikace třetích stran. Pokud budete mít dotaz k programu, obraťte se prosím na jeho tvůrce.

Zdrojové kódy ke knize

Z adresy *http://knihy.cpress.cz/K2080* si po klepnutí na odkaz Soubory ke stažení můžete přímo stáhnout archiv s ukázkovými kódy.

Errata

Přestože jsme udělali maximum pro to, abychom zajistili přesnost a správnost obsahu, chybám se úplně vyhnout nelze. Pokud v některé z našich knih najdete chybu, ať už chybu v textu nebo v kódu, budeme rádi, pokud nám ji oznámíte. Ostatní uživatele tak můžete ušetřit frustrace a pomoci nám zlepšit následující vydání této knihy.

Veškerá existující errata zobrazíte na adrese *http://knihy.cpress.cz/K2080* po klepnutí na odkaz Soubory ke stažení.

KAPITOLA

1

Začínáme

V této kapitole se naučíte:

- Historii a filozofii jazyka C a C++
- Rozdíl mezi procedurálním a objektově orientovaným programováním
- Jak C++ rozšiřuje jazyk C o podporu objektově orientovaného programování
- Jak se C++ opírá o jazyk C při obecném programování
- Standardy programovacího jazyka
- Mechanismy vytváření programu

Vítáme vás v jazyce C++! Tento vzrušující programovací jazyk, který dodává jazyku C podporu objektově orientovaného programování, se stal jedním z nejdůležitějších programovacích jazyků devadesátých let dvacátého století a slibně pokračuje na své cestě i po roce 2000. Jeho předchůdce, jazyk C, přináší do jazyka C++ tradici výkonného, uceleného, rychlého a přenositelného jazyka. Objektově orientované vlastnosti dávají jazyku C++ novou metodiku programování, která je navržena tak, aby si poradila se stupňující se obtížností moderních úkolů programování. Jeho nově rozšířené možnosti šablon přinášejí další metodiku programování, kterou je obecné programování. Toto trojnásobné dědictví představuje opravdové požehnání, ale zároveň i prokletí. Díky němu je jazyk velmi výkonný, ale také to 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 všechny podpůrné prostředky – objekty, třídy, skrývání dat, polymorfismus, dědičnost, až po podporu obecného programování. (Samozřejmě, jak se budete jazyk C++ učit, přejdou tyto výrazy z módních slov do nezbytného slovníku kultivovaného projevu.)

Učíme se C++: Na co se můžete těšit

C++ spojuje tři oddělené programovací tradice – tradici procedurálního jazyka reprezentovanou C; tradici objektově orientovaného jazyka reprezentovaného rozšířením C o třídy C++; a obecné programování podporované šablonami jazyka C++. Tato kapitola do těchto tradic krátce nahlédne. Ale nejprve uvažujme o tom, co z tohoto dědictví vyplývá při učení jazyka C++. Jedním důvodem používání jazyka C++ jsou jeho objektově orientované vlastnosti. Pokud chcete využít této výhody, potřebujete

solidní základ ve standardu C, protože tento jazyk poskytuje základní typy, operátory, řídicí 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++ představuje zhruba tolik práce, jako kdybyste se znovu učili jazyku C. Jestliže tedy znáte jazyk C a přecházíte na C++, musíte se odnaučit některé programovací zvyklosti. Pokud neznáte C a chcete se naučit C++, musíte zvládnout prvky C, prvky OOP a obecné programování, ale zato se alespoň nemusíte odnaučovat některým programovacím návykům. Jestliže se vám začíná zdát, že učení jazyka C++ může představovat z vaší strany jisté úsilí při namáhání mysli, máte pravdu. Tato kniha vás provede tímto procesem jasným a nápomocným způsobem, krok po kroku tak, že namáhání mysli bude přiměřeně pozvolné, aby nezanechalo žádné škody na vašem rozumu.

Mistrovství v C++ přistupuje k jazyku C++ tak, že učí jak základy C, tak nové prvky, takže kniha nepředpokládá, že máte nějaké předběžné znalosti jazyka C. Začnete výukou společných rysů, které C++ sdílí s C. Dokonce i když znáte jazyk C, možná zjistíte, že vám tato část knihy poskytne jeho kvalitní přehled. Navíc zdůrazňuje způsob, který se stane později důležitým a ukazuje, v čem se C++ liší od C. Až budete dostatečně zběhlí v základech C, doplníme nadstavbu v podobě C++. Na tomto místě se začnete učit o objektech a třídách a o tom, jakým způsobem je jazyk C++ implementuje. Dále se dozvíte něco o šablonách.

Tato kniha se nepovažuje za úplný manuál C++; nezkoumá každé zákoutí nebo skulinu tohoto jazyka. Ale naučíte se zde všem hlavním rysům jazyka včetně šablon, výjimek a jmenných prostorů, což jsou jeho rozšíření z nedávné doby. Nyní se stručně podívejme na pozadí jazyka C++.

Počátky C++: Trochu historie

Výpočetní technika se rozvinula během několika posledních desetiletí úžasnou rychlostí. Dnes je laptop schopen pracovat mnohem rychleji a ukládat více informací než sálový počítač před 40 lety. (Pouze několik málo programátorů si ještě pamatuje objemné bedny děrných štítků, které byly dodávány mohutnému, celý sál vyplňujícímu počítačovému systému s majestátními 100 kB paměti – nedostačující dnes ke spuštění dobré hry pro osobní počítač.) Počítačové jazyky se také rozvinuly. Změny možná nejsou tak dramatické, ale jsou důležité. Větší a výkonnější počítače umožňují vznik větších a složitějších programů, které na oplátku způsobují nové problémy ve správě a údržbě programů. V 70. letech pomohly jazyky C a Pascal vzniku období strukturovaného programování, což je filozofie, která vnesla jistý řád a kázeň do oblasti, jež tyto kvality nutně potřebovala. Kromě poskytnutí prostředků pro strukturované programování produkoval jazyk C také robustní, rychle běžící programy, schopné adresovat hardware a řídit tak například komunikační porty a ovladače disků. Tyto dary pomohly jazyku C, aby se stal dominantním programovacím jazykem 80. let. 80. léta se také stala svědkem vývoje 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) poněkud blíže.

Jazyk C

Začátkem 70. let pracoval Dennis Ritchie z Bell Laboratories na svém projektu rozvoje operačního systému UNIX. (Operační systém je sada programů, která spravuje zdroje

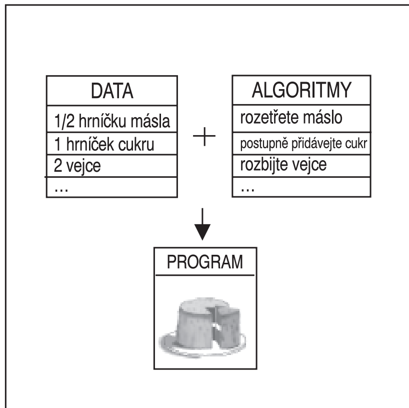
počítače a stará se o komunikaci s uživatelem. Je to například operační systém, kdo vyšle na obrazovku systémovou výzvu a provede váš program.) Na tuto práci potřeboval Ritchie jazyk, který by byl stručný a výstižný, vytvářel robustní a rychlé programy a mohl účinně řídit hardware. Tradičně se programátoři setkávají s těmito potřebami, když používají jazyk assembler, jenž je velmi svázán se strojovým kódem počítače. Avšak assembler je *nízkoúrovňový* jazyk, to znamená, že je závislý na určitém procesoru počítače. Pokud tedy chcete přesunout program napsaný v assembleru na jiný druh počítače, může se stát, že budete muset program úplně přepsat pomocí jiného assembleru. To by bylo trochu jako kdybyste při každé koupi nového auta zjistili, že se konstruktéři rozhodli změnit umístění a funkci ovládacích prvků, což by vás vždy nutilo naučit se znovu řídit. Ale UNIX byl určený pro práci na různých typech počítačů (neboli platformách). To předpokládalo použití vysokoúrovňového programovacího jazyka. *Vysokoúrovňový* programovací jazyk je orientován na řešení problému a nikoli na určitý hardware. Zvláštní programy, které se nazývají *překladače*, převádějí vysokoúrovňový jazyk do strojového kódu určitého počítače. Pokud pro každou platformu použijete jiný překladač, může stejný program napsaný ve vysokoúrovňovém jazyce běžet na různých platformách. Ritchie chtěl jazyk, který by spojoval účinnost strojového jazyka na nízké úrovni a přístup k hardwaru s vyšší úrovní všeobecnosti jazyka a přenositelností. A tak na základě starších programovacích jazyků vytvořil jazyk C.

Filozofie programování v jazyce C

Protože C++ staví nové principy programování na jazyku C, měli bychom se nejprve podívat na starší filozofii jazyka C. Obecně se počítačové jazyky zabývají dvěma pojmy – daty a algoritmy. Data představují informace, které program používá a zpracovává. Algoritmy jsou programem používané metody (viz obrázek 1.1). Jazyk C, podobně jako většina dnešních hlavních jazyků, je *procedurální*. To znamená, že zdůrazňuje hledisko algoritmu programu. Procedurální programování se v podstatě skládá z určení činností, které by měl počítač provádět a poté z jejich implementace pomocí programovacího jazyka. Program popisuje sadu procedur, kterými se počítač řídí při vytváření určitého výstupu, stejně jako recept předepisuje kuchaři postup pečení.

Dřívější procedurální jazyky jako Fortran a Basic narážely ve větších programech na organizační problémy. Programy například často používaly větvení, které směrovalo provádění programu na jednu nebo druhou skupinu instrukcí v závislosti na výsledku určitých testů. Mnoho starších programů mělo tak zamotaný průběh vykonávání (tzv. „špagetové programování"), že bylo prakticky nemožné porozumět programu čtením kódu a požadavek na změnu takového programu se rovnal katastrofě. Počítačoví vědci proto vyvinuli ukázněnější styl programování, který se nazývá *strukturované programování*. Jazyk C má vlastnosti podporující tento přístup. Strukturované programování například omezuje větvení (rozhodnutí, která další instrukce se má vykonat) na malou množinu způsobných konstrukcí. Jazyk C má ve svém slovníku tyto konstrukce – cyklus `for`, cyklus `while`, cyklus `do while` a příkaz `if else`.

Návrh stylem *shora 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.



Obrázek 1.1 Data + algoritmy = program

(Uspořádejte si své studium. Ach! Uspořádejte si svůj pracovní stůl, stolní desku, kartotéku a poličky na knihy. Ech! Začněte pracovním stolem a uspořádejte každou zásuvku a začněte prostřední. Hm, snad mohu ten úkol zvládnout.) Pojetí jazyka C tento přístup usnadňuje, podporuje vás při návrhu programových jednotek nazývaných *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.

Rozšíření C++: Objektově orientované programování

Ačkoli principy strukturovaného programování zlepšily srozumitelnost, spolehlivost a usnadnily údržbu programů, programování velkých celků stále ještě zůstává problémem. *Objektově orientované programování* (OOP) přináší nový přístup řešení tohoto problému. Na rozdíl od procedurálního programování, které zdůrazňuje algoritmy, OOP klade důraz na data. Spíše než se snažit, aby problém vyhovoval procedurálnímu přístupu jazyka, OOP usiluje o to, aby jazyk vyhovoval problému. Myšlenka spočívá v návrhu datových forem, které odpovídají základním rysům problému.

V jazyce C++ existuje slovo *class*, které popisuje specifikaci takovéto nové formy dat a *objekt* představuje určitou datovou strukturu konstruovanou v souladu s daným plánem. Třída by mohla například popisovat obecné vlastnosti manažera společnosti (dejme tomu jméno, titul, plat a neobvyklé schopnosti), zatímco objekt by mohl představovat určitého manažera (Antonín Houžvička, zástupce ředitele, 125 000 Kč, umí obnovit registry ve Windows). Obecně vzato, třída definuje všechny údaje, které slouží k reprezentaci objektu a činnosti, jež mohou být s těmito údaji prováděny. Předpokládejme, že chcete vytvořit počítačový kreslicí program schopný namalovat čtyřúhelník. Mohli byste definovat třídu, která tento čtyřúhelník popisuje. Datová část specifikace by mohla zahrnovat takové informace, jako je umístění rohů, výška a šířka, barva a styl ohraničení, barvu a vzor použitý pro vyplnění čtyřúhelníka. Výkonná část specifikace by mohla obsahovat metody pro posun čtyřúhelníka, změnu jeho velikosti, otočení, změnu barev a vzorů a kopírování čtyřúhelníka na jiné místo. Když pak pomocí svého programu namalujete čtyřúhelník, vytvoří vám objekt podle specifikace této třídy. Tento objekt bude obsahovat všechny datové hodnoty, které popisují čtyřúhelník a pomocí metod

třídy můžete tento čtyřúhelník upravovat. Jestliže namalujete dva čtyřúhelníky, program vytvoří dva objekty, pro každý čtyřúhelník jeden. Při vytváření programů podle pravidel OOP nejprve navrhujete třídy, které přesně popisují to, s čím program pracuje. Kreslicí program může například definovat třídy představující čtyřúhelníky, čáry, kruhy, štětce, pera a podobně. Vzpomeňte si, 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. Takovýto postup, začínající od nejnižší organizační úrovně, jako jsou třídy, po nejvyšší úroveň, jako je návrh programu, se nazývá programování *zdola nahoru*. Programování podle OOP představuje mnohem více než pouhé svázání dat a metod s definicí třídy. OOP například podporuje vytváření znovupoužitelného kódu, který nakonec může ušetřit mnoho práce. Skrývání informací zabezpečuje data proti nevhodnému přístupu. Polymorfismus umožňuje vytvářet vícenásobné definice operátorů a funkcí, přičemž aktuální souvislosti určují, která definice se má použít. Pomocí dědičnosti je možné odvozovat nové třídy od starých. Je zřejmé, že objektově orientované programování zavádí mnoho nových pojmů a vyžaduje odlišný přístup k programování než procedurální programování. Místo na úkoly se můžete soustředit na reprezentaci pojmů. Místo přístupu k programování shora dolů občas použijete přístup zdola nahoru. Kniha vás provede všemi těmito pojmy pomocí množství snadno pochopitelných příkladů.

Návrh užitečné a spolehlivé třídy může být někdy těžkým úkolem. Naštěstí jazyky OOP umožňují do vašeho programu jednoduše začlenit již existující třídy. Prodejci poskytují různé užitečné knihovny tříd včetně takových, které jsou navrženy, aby usnadňovaly vytváření programů pro prostředí jako je Windows nebo Macintosh. Jedna ze skutečných výhod C++ je, že vám dovolí jednoduchým způsobem znovu použít a přizpůsobit existující dobře otestovaný kód.

C++ a obecné programování

Obecné programování je dalším programovacím modelem podporovaným jazykem C++. S OOP sdílí společný cíl v podobě snahy o vytváření jednoduššího kódu pro opakované použití a technikou abstrakce obecných návrhů. Zatímco OOP zdůrazňuje datové hledisko programování, generické programování zdůrazňuje hledisko algoritmické. Ale jeho zaměření je jiné. OOP je prostředkem pro správu velkých projektů, zatímco generické programování poskytuje nástroje pro vykonávání běžných úkolů jako je třídění dat nebo spojování seznamů. Pojem *obecný* vyjadřuje vytváření typově nezávislého kódu. Jazyk C++ má množství datových typů – celá čísla, čísla s desetinnou částí, znaky, řetězce znaků, uživatelsky definované struktury složené z několika typů. Kdybyste například chtěli třídit data těchto různých typů, museli byste pro každý typ vytvořit odlišné třídicí funkce. Obecné programování rozšiřuje jazyk tak, že můžete napsat jedinou funkci pro obecný (tj. neurčený) typ a použít ji pro různé typy. Mechanismem, který popsaný přístup realizuje, jsou šablony jazyka C++.

Zrození C++

Jazyk C++ spatřil světlo světa, podobně jako C, v Bell Labs, kde ho začátkem 80. let vyvinul Bjarne Stroustrup. Podle jeho vlastních slov, „C++ byl původně navržen proto, abychom (mí přátelé a já) nemuseli programovat v assembleru, jazyku C nebo jiných moderních vysokoúrovňových jazycích. Jeho hlavním účelem bylo zjednodušit a zpříjemnit

programátorům psaní dobrých programů“ (Bjarne Stroustrup, *The C++ Programming Language*. Třetí vydání. Reading MA: Addison-Wesley Publishing Company, 1997).

TIP

Domovská stránka Bjarna Stroustrupa

Bjarne Stroustrup je autorem návrhu jazyka C++, implementace a úplných referenčních manuálů *The C++ Programming Language* a *The Design and Evolution of C++*. Jeho osobní stránky na AT&T Labs Research by měly být první, které si dáte do svých oblíbených stránek:

www.research.att.com/~bs

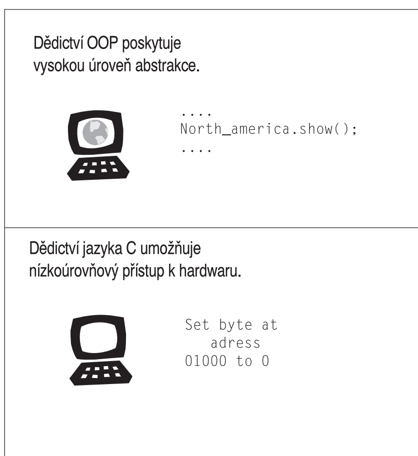
Tyto stránky obsahují zajímavý historický pohled na vznik a důvod existence jazyka C++, Stroustrupovy biografické materiály a časté otázky k C++. Kupodivu vůbec nejčastěji kladenou otázkou je, jak se vyslovuje jméno *Bjarne Stroustrup*. Stáhněte si příslušný soubor .wav a poslechněte si jej.

Stroustrup se více soustředil na vytvoření užitečného jazyka, než na prosazení určitých programovacích postupů nebo stylů. Při stanovování vlastností jazyka jsou skutečné programovací potřeby důležitější než teoretická čistota. Stroustrup založil C++ na stručnosti C, jeho vhodnosti pro systémové programování, široce rozšířené dostupnosti a úzké vazbě na operační systém UNIX. Hledisko OOP v C++ bylo inspirováno počítačovým simulačním jazykem, který se nazýval Simula67. Stroustrup přidal do C vlastnosti OOP, aniž by významně změnil složku C. Jazyk C++ je tedy nadstavbou C, což zajišťuje, že každý platný program v C je také platným programem v C++. Je zde sice několik drobných nesrovnalostí, ale nic rozhodujícího. Programy psané v C++ mohou používat softwarové knihovny jazyka C. *Knihovny* jsou kolekce programových modulů, které můžete volat z programu. Poskytují prověřená řešení mnoha běžných programovacích problémů, proto vám šetří mnoho času a úsilí. To pomohlo i rozšíření jazyka C++.

Jméno C++ pochází z přírůstkového operátoru ++, který přičítá k hodnotě proměnné 1. Jméno jazyka tedy vychází ze jména jazyka C s odkazem na jeho rozšíření.

Počítačový program transformuje problém skutečného života do řady činností, které mají být provedeny počítačem. Zatímco objektový aspekt dává jazyku C++ schopnost uvést do spojitosti představy, které jsou zahrnuty v problému, C část C++ umožňuje jazyku dostat se blíže k hardwaru (viz obrázek 1.2). Toto spojení schopností napomohlo úspěšnému rozšíření jazyka C++. Možná je to také dáno pomyslným přeřazením rychlosti na vyšší stupeň při změně jednoho pohledu na program na pohled jiný. (Vskutku, někteří puritáni OOP pohlížejí na rysy OOP přidané do C jako na přidání křídel praseti, třebaže má sklon být výkonným žroutem.) Protože C++ roubuje principy OOP na jazyk C, můžete objektově orientované rysy C++ ignorovat. Pokud to ovšem uděláte, přijdete o mnoho.

Jakmile dosáhl jazyk C++ jistého úspěchu, dodal Stroustrup šablony umožňující obecné programování. Teprve až se začaly používat a rozšiřovat, zdálo se být zřejmé, že jsou stejně důležitým dodatkem jako OOP. Někteří dokonce tvrdí, že ještě důležitějším. Skutečnost, že C++ zahrnuje jak OOP, tak obecné programování, ukazuje, že C++ zdůrazňuje užitkovost před ideovým přístupem, což je jedním z důvodů úspěchu tohoto jazyka.



Obrázek 1.2 Dualita C++

Přenositelnost a standardy

Představte si, že jste v práci napsali užitečný program v jazyce C++ pro starší 286 PC AT a vedení se rozhodlo nahradit tento stroj pracovní stanicí Sun, počítačem, který používá jiný procesor a jiný operační systém. Můžete spustit svůj program na nové platformě? Samozřejmě, ale musíte tento program přeložit pomocí překladače navrženého pro novou platformu. A co když jste nuceni udělat nějaké změny do kódu, který jste napsali? Pokud můžete přeložit program bez jakýchkoliv změn, a ten běží bez problémů, říkáme, že je takový program *přenositelný*.

Přenositelnosti brání množství překážek; první z nich je hardware. Program závislý na hardwaru pravděpodobně přenositelný nebude. Například program, který má přímo ovládat video kartu VGA na počítači IBM, bude pro Sun nesrozumitelný. (Problémy s přenositelností můžete minimalizovat umístěním částí závislých na hardwaru do funkčních modulů; potom budete muset přepisovat pouze tyto specifické moduly.) V naší knize se takovémuto druhu programování vyhneme.

Druhou překážkou přenositelnosti je jazyková odlišnost. I v mluveném jazyce mohou velmi snadno vznikat problémy. Popis denních událostí člověkem z Yorkshiru nemusí být přenositelný do Brooklynu, i když se v obou oblastech mluví anglicky. V počítačových jazycích mohou také vznikat nářečí. Je implementace jazyka C++ na IBM PC stejná jako implementace na Sunu? Ačkoli by většina implementátorů ráda vytvořila své verze C++ kompatibilní s ostatními, je to obtížné udělat bez zveřejnění standardů, které přesně popisují, jak jazyk pracuje. Proto Americký národní institut pro standardy (American National Standard Institute, ANSI) vytvořil v roce 1990 výbor (ANSI X3J16), aby vyvinul standard pro jazyk C++. (ANSI již vyvinul standard pro C.) Mezinárodní organizace pro standardy (International Standard Organization, ISO) brzy spojila tento proces se svým vlastním výborem (ISO-WG-21) a rozvoj standardů C++ se tak stal společným úsilím ANSI/ISO. Tyto výbory se setkávají společně třikrát do roka a vžilo se tak pro ně označení výbor ANSI/ISO. Rozhodnutí ANSI/ISO vytvořit standardy zdůrazňuje, že se C++ stal důležitým a široce rozšířeným jazykem. Také to indikuje, že C++ dosáhl jisté úrovně zralosti,

navzdory tomu, že není produktivní zavádět standardy, když se jazyk tak rychle vyvíjí. Přesto jazyk C++ od doby, kdy výbor započal svou práci, prodělal významné změny.

Práce na standardech ANSI/ISO C++ začala v roce 1990. V následujících letech vydal výbor několik prozatímních pracovních dokumentů. V dubnu 1995 byl vydán koncept výboru (Committee Draft, CD) k veřejnému připomínkování. V prosinci 1996 uvolnil druhou verzi (CD2) pro další veřejné posouzení. Tyto dokumenty nejen vylepšily popis existujících vlastností jazyka C++, ale také ho rozšířily o výjimky (RTTI) šablony a standardní knihovnu šablon (Standard Template Library, STL). Závěrečný mezinárodní standard (ISO/IEC 14882:1998) byl organizacemi ISO, IEC (International Electrotechnical Committee) a ANSI přijat v roce 1998. V roce 2003 vyšlo druhé vydání standardu C++ (IOS/IEC 14882:2003); jde o technickou revizi, což znamená, že je jen úpravou prvního vydání, – jsou odstraněny překlepy, nejednoznačnosti apod. – avšak jazyk samotný se nemění. Naše kniha je založena na tomto standardu.

Standard ANSI/ISO C++ navíc vychází ze standardu ANSI C, protože jazyk C++ má být v maximální možné míře nadmnožinou jazyka C. To znamená, že jakýkoli platný program v jazyce C by měl být i platným programem v C++. Mezi ANSI C a odpovídajícími pravidly C++ existuje několik rozdílů, ty jsou ale méně podstatné. ANSI C zapracovává některé vlastnosti, jež byly poprvé představeny v C++, jako jsou vytváření prototypů funkcí a kvalifikátor typu const.

Před vydáním ANSI C byla skupinou programátorů v jazyce C za takzvaný standard považována kniha *The C Programming Language* autorů Kernighana a Ritchieho (Addison-Wesley Publishing Company Reading, MA. 1978); tato kniha vyšla v češtině pod názvem *Programovací jazyk C* (Computer Press, 2006). Tento standard byl často označován jako K&R C; po zveřejnění standardu ANSI C je nyní jednodušší K&R C nazýván *klasickým C*.

Standard ANSI C definuje nejen jazyk C, ale také standardní knihovnu C, kterou musí implementace ANSI C podporovat. Jazyk C++ tuto knihovnu používá také; naše kniha se na ni bude odkazovat jako na standardní knihovnu C nebo standardní knihovnu. Standard ANSI/ISO C++ navíc definuje standardní knihovnu tříd jazyka C++.

Nedávno byl standard ANSI C přepracován; nový standard, někdy nazývaný C99, byl přijat ISO v roce 1999 a ANSI v roce 2000. Tento standard přidává jazyku C další vlastnosti, jako je nový celočíselný typ podporovaný některými překladači jazyka C++. I když tyto vlastnosti nejsou součástí nynějšího standardu jazyka C++, mohou být obsaženy v příštím.

Než výbor ANSI/ISO C++ zahájil svou práci, mnoho lidí uznávalo za standard nejnovější verze C++ od Bell Labs. Překladač mohl být například považován za kompatibilní s vydáním C++ verze 2.0 nebo 3.0.

Jazyk C++ se dále vyvíjí, byly zahájeny práce na novém standardu. Nová verze se pracovně označuje C++0X, neboť dokončení se očekává někdy na konci dekády, kolem roku 2009.

V této knize je popsáno druhé vydání standardu ISO/ANSI C++ (ISO/IEC 14882:2003), takže příklady by měly vyhovovat všem překladačům kompatibilním s tímto standardem. Standard C++ je však prozatím poměrně nový, takže může obsahovat určité nesrovnalosti. Starší verze překladačů mohou například postrádat jmenný prostor nejnovějších šablon. Podpora STL popsaná v kapitole 16 „Třída `string` a standardní knihovna šablon“ je v případě starších kompilátorů neúplná. Některé starší unixové systémy používají frontendový překladač, který předává přeložený kód překladači C, jenž není plně ANSI kom-

patibilní, což má za následek, že některé možnosti jazyka se nevyužijí a nejsou podporovány některé standardní knihovní funkce ANSI a hlavičkové soubory. Dokonce i když překladače odpovídají standardu, určité prvky jsou závislé na překladu – například počet bajtů pro zobrazení celého čísla.

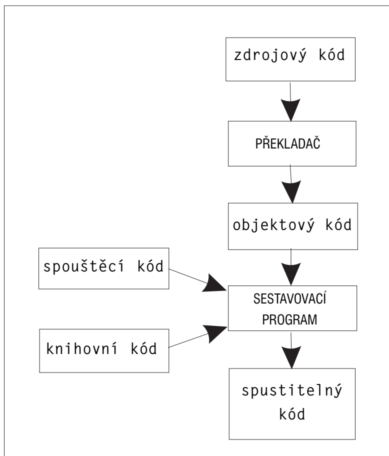
Než se pustíme do jazyka C++, zastavíme se na chvíli u základů vytváření programů a používání této knihy.

Postup vytváření programu

Předpokládejme, že jste napsali program v jazyce C++. Jak ho spustíte? Přesný postup závisí na prostředí počítače a na použitém překladači C++, ale bude se podobat následujícím krokům (viz obrázek 1.3):

1. Pomocí jakéhokoliv textového editoru napište program a uložte ho do souboru. Tento soubor představuje *zdrojový kód (source code)* vašeho programu.
2. Přeložte zdrojový kód. To znamená, že spustíte program, který překládá zdrojový kód do interního jazyka zvaného *strojový jazyk (machine language)*, jenž se používá na daném počítači. Soubor, který obsahuje přeložený program, je *objektový kód (object code)* vašeho programu.
3. Sestavte objektový kód s dalším kódem. Programy psané v jazyce C++ například používají *knihovny (libraries)*. Knihovny C++ obsahují objektový kód pro sadu počítačových rutin nazývaných *funkce (function)*, které provádějí úlohy, jako je zobrazení informace na monitor nebo výpočet druhé odmocniny čísla. Sestavením dojde ke spojení vašeho objektového kódu s objektovým kódem použitých funkcí a s určitým standardním spouštěcím kódem, čímž je vytvořena spustitelná verze vašeho programu. Soubor s tímto výsledným produktem se nazývá *spustitelný kód (executable code)*.

S pojmem zdrojový kód se budete setkávat v celé knize, určitě si jej zapište do paměti s přímým přístupem.



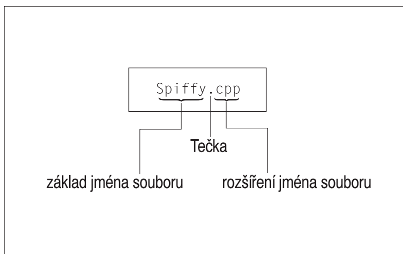
Obrázek 1.3: Postup vytváření programu

Programy v této knize jsou obecně použitelné a měly by běžet na libovolném systému, který podporuje současný jazyk C++. (Možná budete potřebovat některou z posledních verzí, abyste získali podporu jmenných prostorů a nejnovějších vlastností šablon.) Postupy vytváření programu se mohou lišit. Na kroky, které k tomu vedou, se nyní podíváme trochu podrobněji.

Vytváření souboru se zdrojovým kódem

V celém zbytku knihy budeme pracovat se zdrojovými soubory; v této části si řekneme, jak se vytvářejí. Některé implementace jazyka C++, jako je Microsoft Visual C++, Borland C++ (různé verze), Watcom C++, Symantec C++ a Metrowerks CodeWarrior, poskytují *integrovaná vývojová prostředí* (*integrated development environments, IDE*), která umožňují provádět z jednoho hlavního programu všechny kroky vývoje programu včetně editace. Jiné implementace, jako je AT&T C++ nebo GNU C++ na Unixu nebo Linuxu, vykonávají pouze etapy překladu a sestavování a očekávají, že napíšete všechny příkazy v příkazové řádce systému. V takových případech můžete použít libovolný textový editor, pomocí kterého vytvoříte nebo upravíte zdrojový kód. Na Unixu je to například vi, ed, ex nebo emacs. V systému DOS můžete použít edlin nebo edit či některý z dostupných editorů programů. Dokonce je možné použít libovolný textový editor za předpokladu, že soubor uložíte ve standardní znakové sadě ASCII systému DOS namísto speciálního formátu textového editoru.

Při pojmenování zdrojového souboru musíte zvolit správnou příponu, která označuje soubor jako zdroj jazyka C++. To je důležité nejen pro vaši orientaci, ale také pro překladač. (Když si překladač na Unixu stěžuje na „chybné magické číslo“, jde právě o jeho roztomile nesrozumitelný způsob vyjádření toho, že jste použili chybnou příponu.) Přípona se skládá z tečky následované znakem nebo skupinou znaků, která se nazývá rozšíření (viz obrázek 1.4).



Obrázek 1.4: Rozšíření jména zdrojového souboru

Které rozšíření použijete, závisí na implementaci jazyka C++. Tabulka 1.1 ukazuje některé běžné volby. Například `spiffy.C` je platné jméno zdrojového souboru překladače AT&T jazyka C++. Všimněte si, že UNIX rozlišuje velikosti písmen, což znamená, že byste měli použít velké písmeno C. Ve skutečnosti je platné i malé písmeno c, ale toto rozšíření používá standard jazyka C. Abyste se tedy na systémech UNIX vyhnuli nedorozuměním, používejte c v programech jazyka C a C v programech jazyka C++. Pokud vám nevadí psaní jednoho nebo dvou písmen navíc, můžete na některých systémech UNIX použít rozšíření `cc` a `cxx`. V porovnání s Unixem poněkud jednodušší DOS nerozlišuje velikosti

písmen, a tak dosové implementace používají pro odlišení programů C a C++ dodatečná písmena, jak je vidět v tabulce 1.1.

Tabulka 1.1: Rozšíření jména zdrojového souboru

Implementace C++	Rozšíření jména zdrojového souboru
UNIX	C, cc, cxx, c
GNU C++	C, cc, cxx, cpp, c++
Digital Mars	cpp, cxx
Borland C++	cpp
Watcom	cpp
Microsoft Visual C++	cpp, cxx, cc
Metrowerks CodeWarrior	cpp,cp, cc, cxx, c++

Překlad a sestavení

Stroustrup původně implementoval překladač jazyka C++ jako program převádějící C++ na C místo přímého překladače jazyka C++ do objektového kódu. Tento program, který se jmenoval cfront (podle C front end), převáděl zdrojový kód C++ do zdrojového kódu C, jenž potom mohl být zpracován standardním překladačem C. Tento přístup zjednodušil zavedení C++ mezi komunitu zvyklou na jazyk C a některé implementace ho používaly i pro přenesení C++ na jiné platformy. Jak se jazyk C++ rozvíjel a získával na popularitě, pouštělo se stále více a více implementátorů do vytváření překladačů jazyka C++, které generovaly objektový kód přímo ze zdrojového kódu C++. Tento přímý přístup urychluje proces překladu a klade důraz na to, že C++ je samostatným, i když podobným jazykem.

Pro uživatele je rozdíl mezi převodníkem `cfront` a překladačem často téměř nepostřehnutelný. Například na systému UNIX může příkaz `CC` nejprve předat program převodníku `cfront` a jeho výstup potom automaticky překladači jazyka C, který se nazývá `cc`. Napříště budeme používat výraz *překladač* i pro nástroje provádějící kombinaci převodu a překladu. Mechanismus překladu závisí na implementaci a v následujících kapitolách nastíníme několik společných vlastností. Tyto přehledy nám ukáží základní kroky, nenahradí však nutnost prostudování dokumentace systému.

Pokud máte převodník `cfront` a znáte jazyk C, můžete se podívat na převody vašich programů v jazyce C++ do C a získat tak přehled o tom, jak jsou některé vlastnosti jazyka C++ implementovány.

Překlad a sestavení pod Unixem

Klasický unixový překladač C++ se spouští příkazem `CC`. Můžete však narazit i na systémy, které překladač C++ vůbec nemají, anebo mají proprietární překladač, popřípadě jiný, importovaný překladač na komerční bázi nebo překladač volně šiřitelný, například GNU `g++`. Příkaz `CC` většinou funguje i v těchto případech, tedy je-li v systému jiný než klasický překladač (nikoli pochopitelně, když tam překladač není), avšak záleží na konkrétním systému, který překladač je tímto příkazem spuštěn. Pro jednoduchost budeme

v dalším textu používat příkaz CC, avšak je nutno si uvědomit, že v některých systémech by bylo nutno jej nahradit jiným příkazem.

Program tedy překládáme příkazem CC, kde obě písmena jsou velká. Tím jej odlišíme od příkazu cc, který slouží ke spuštění standardního unixového překladače jazyka C. Překladač CC lze použít i na unixovém příkazovém řádku.

Abyste například přeložili zdrojový kód C++ ze souboru spiffy.C, napíšete za výzvu Unixu tento příkaz:

```
CC spiffy.C
```

Jestliže díky vaší dovednosti, nadšení nebo štěstí nemá program žádné chyby, překladač vygeneruje soubor objektového kódu s rozšířením o. V tomto případě by měl překladač vytvořit soubor spiffy.o.

Následně překladač automaticky předá soubor objektového kódu systémovému sestavovacímu programu, tj. programu, který spojuje váš kód s knihovním kódem a vytváří spustitelný soubor. Implicitně se spustitelný 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 překládáte nový program, vytváří se nový spustitelný soubor a.out, který nahrazuje předchozí a.out. (Je tomu tak proto, že spustitelné soubory zabírají mnoho prostoru, takže přepisování starých spustitelných souborů pomáhá snižovat nároky na úložný prostor.) Pokud ale vyvíjíte spustitelný program, který chcete zachovat, můžete použít příkaz Unixu mv, kterým změníte jméno spustitelného souboru.

V C++, podobně jako v C, můžete program rozložit do více než jednoho souboru. (Mnoho programů v této knize, počínaje kapitolou 8 a konče 16, tak činí.) V takovém případě můžete překládat program vypsáním jmen všech souborů na příkazovou řádku:

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

Uvedete-li více souborů zdrojového kódu, překladač nemaže soubory objektových kódů. Pokud změníte pouze soubor my.C, můžete znovu přeložit program příkazem:

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

Tento příkaz znovu přeloží soubor my.C a výsledek spojí s dříve přeloženým souborem precious.o.

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

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

Překlad a sestavení pod Linuxem

Linuxové systémy obvykle používají g++, což je překladač GNU C++ od Free Software Foundation. Tento překladač obsahuje většina distribucí Linuxu, ale nemusí být vždy nainstalován. g++ pracuje obdobně jako standardní překladač Unixu. Například:

```
g++ spiffy.cxx
```

Tento příkaz vytvoří spustitelný soubor a.out.

Některé verze mohou vyžadovat připojení knihovny jazyka C++:

```
g++ spiffy.cxx -lg++
```

Pokud chcete přeložit více zdrojových souborů, uveďte jejich seznam na příkazové řádce:

```
g++ my.cxx precious.cxx
```

Tento příkaz vytvoří spustitelný soubor `a.out` a dva soubory objektových kódů `my.o` a `precious.o`. Jestliže potom změníte pouze jeden ze zdrojových souborů, řekněme `my.cxx`, můžete znovu přeložit soubor `my.cxx` a spojit ho se souborem `precious.o` následujícím příkazem:

```
g++ my.cxx precious.o
```

Překladač Comeau C++ (viz *www.comeaucomputing.com*) je další možností; vyžaduje však přítomnost překladače GNU. Překladač Comeau je nicméně nejúplnější a nejpřesnější implementací standardu C++.

Překladač GNU je k dispozici pro mnoho systémů včetně řádkových režimů v systémech Windows a Unixu na různých platformách.

Překladače pro Windows ovládané z příkazového řádku

Nejlevnějším způsobem překladu programů napsaných v jazyce C++ na počítačích s operačními systémy Windows je stažení překladače ovládaného z příkazového řádku, který běží v okně MS-DOS. Verze překladače GNU C++ pro MS-DOS se nazývá gpp a můžete ji zdarma získat na adrese *www.delorie.com/djgpp/*. Borland bezplatně poskytuje překladač ovládaný z příkazového řádku na adrese *www.borland.com/freecompiler/*. U obou si musíte přečíst instalační pokyny, protože některé části instalačních procesů nejsou automatické.

Před použitím překladače gpp nejprve otevřete okno MS-DOS. Chcete-li přeložit soubor se zdrojovým kódem `great.cpp`, napište následující příkaz:

```
gpp great.cpp
```

Jestliže překlad proběhne úspěšně, je výsledný spustitelný soubor pojmenován `a.exe`.

Před použitím překladače od Borlandu nejprve otevřete okno MS-DOS. Chcete-li přeložit soubor se zdrojovým kódem `great.cpp`, napište příkaz:

```
bcc32 great.cpp
```

Jestliže překlad proběhne úspěšně, je výsledný spustitelný soubor pojmenován `great.exe`.

Překladače pro systémy Windows

Produktů určených pro operační systémy Windows je tak velké množství a jsou tak často aktualizovány, že by nebylo rozumné popisovat je jednotlivě. Nejoblíbenější jsou Microsoft, Borland, Metrowerks a Digital Mars. I navzdory rozdílným návrhům a zaměřením mají řadu společných vlastností.

Obvykle musíte pro váš program vytvořit projekt a přidat do něj soubor nebo soubory, které váš program vytváří. Každý dodavatel poskytuje integrované vývojové prostředí (Integrated Development Environment, IDE) se zadáváním voleb z nabídky a pravděpodobně i s podporou automatického vytváření projektu. Velmi důležité je, abyste správně nastavili druh programu, který vytváříte. Obvykle vám překladač nabídne mnoho možností, jako je aplikace Windows, aplikace MFC pro Windows, dynamicky připojovaná knihovna, ovládací prvek ActiveX, soubor spustitelný v dosovém nebo znakovém režimu,

statická knihovna nebo konzolová aplikace. Některé z těchto voleb jsou k dispozici pro 16bitové i 32bitové verze.

Protože programy této knihy jsou obecné, měli byste se vyhnout volbám, které vyžadují kód závislý na platformě, jako je aplikace Windows. Místo toho byste měli spouštět program ve znakovém režimu. Tato volba bude záviset na překladači. V prostředí Microsoft Visual C++ zadejte volbu Win32 Console Application. Překladače od Metrowerku nabízejí volby Win32 Console C++ App a Win32 WinSIOUX C++ App, obě jsou vhodné. (První spouští přeložený program v okně Dosu a druhá ve standardním okně Windows.) Některé verze od Borlandu zdůrazňují volbu EasyWin, která emuluje práci v Dosu; jiné verze nabízejí volbu Console. Obecně platí, že se podíváte, zda nenaleznete možnost označenou Console, znakový režim (character-mode) nebo soubor spustitelný v Dosu (DOS executable) a vyzkoušíte ji.

Jakmile jste nakonfigurovali projekt, musíte váš program přeložit a sestavit. IDE vám obvykle nabídne několik možností jako je Compile, Build, Make, Build All, Link, Execute a Run (všechny ale nemusejí být ve stejném IDE!).

- *Compile* obvykle znamená přeložit kód právě otevřeného souboru.
- *Build* nebo *Make* obvykle znamená přeložit zdrojové kódy všech souborů v projektu, a to většinou přírůstkovým způsobem. To znamená, že když projekt obsahuje tři soubory a vy změníte pouze jeden, tak se znovu přeloží pouze ten jeden.
- *Build All* obvykle znamená přeložit všechny zdrojové soubory znovu od začátku.
- *Link* znamená (jak bylo popsáno dříve) sestavení přeloženého zdrojového kódu spolu s nezbytným kódem knihoven.
- *Run* nebo *Execute* znamená vykonat program. Pokud jste ještě neprovedli předchozí kroky, příkaz Run je před pokusem o spuštění programu obvykle také provede.

Porušíte-li pravidla jazyka, překladač vygeneruje chybové hlášení a označí řádky, ve kterých problém nastal. Pokud s jazykem začínáte, zřejmě pro vás bude obtížné této zprávě porozumět. Občas se může vlastní chyba vyskytnout před identifikovaným řádkem a někdy jednoduchá chyba generuje celou řadu chybových hlášení.

TIP

Při opravování chyb opravte nejprve tu první. Jestliže ji nemůžete na označeném řádku nalézt, zkontrolujte předchozí řádek.

Je nutno si uvědomit, že přijme-li určitý překladač nějaký program, nemusí to ještě znamenat, že jde o platný program v jazyce C++ a naopak, odmítne-li překladač nějaký program, nemusí to být neplatný program v jazyce C++. Současné kompilátory mnohem více vyhovují standardu než jejich předchůdci před několika lety. V současné době se standardu nejvíce blíží překladač Comeau (a další, které využívají frontendu skupiny Edison Design Group).

TIP

Občas může být překladač po neúplném vytvoření programu zmatený a odpovídá nesmyslnými chybovými hlášeními, která nelze opravit. V takovém případě můžete vše napravit zadáním příkazu Build All, jenž znovu provede celý proces od začátku. Bohužel je velmi obtížné tuto situaci rozpoznat od běžného případu, kdy jsou chybové zprávy nesmyslné pouze zdánlivě.

Prostředí IDE obvykle umožňuje spuštění programu v pomocném okně. Některá IDE toto okno po skončení programu zavírají, jiná ho nechávají otevřené. Jestliže váš překladač okno zavře, stěží si stihnete prohlédnout výstup, pokud nemáte bystré oči a fotografickou paměť. Chcete-li si výstup prohlédnout, musíte na konec programu umístit dodatečný kód:

```
        cin.get(); // přidejte tento příkaz
        cin.get(); // a možná i tento
        return 0;
    }
```

Příkaz `cin.get()` načítá další stisk klávesy, takže tento příkaz způsobí, že program čeká, dokud nestisknete klávesu Enter. (Do programu se neposílá žádný stisk klávesy, dokud nestisknete Enter, takže stisk jiné klávesy se neprojeví.) Druhý příkaz je nutný, jestliže program nechá po normálním načtení vstupu určitou klávesu nezpracovanou. Když například zadáváte číslo, napíšete číslo a stisknete klávesu Enter. Program přečte číslo, avšak zanechá stisk klávesy Enter nezpracován a ten je potom přečten prvním příkazem `cin.get()`.

Překladač Borland C++ Builder se od tradičního návrhu poněkud odchyluje. Jeho hlavním zaměřením je programování pro operační systémy Windows. Abyste ho mohli použít pro obecné programování, vyberte příkaz New z nabídky File. Potom zvolte Console App. Okno, které se otevře, obsahuje základní kostru funkce `main()`. Některé položky můžete vymazat, ale měli byste zachovat dvě následující nestandardní řádky:

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

Jazyk C++ v operačních systémech Macintosh

Předním překladačem jazyka C++ pro operační systémy Macintosh je Metrowerks CodeWarrior. Nabízí v základních ohledech projektově orientované IDE podobné překladačům pro Windows. Začnete výběrem příkazu File, New Project. Tím získáte výběr typu projektu. Pro CodeWarrior zvolte ve starých verzích MacOs:C/C++:ANSI C++ Console, ve středně starých verzích MacOS:C/C++:Standard Console:Std C++ Console resp. MacOS C++ Stationery:Mac OS Carbon:Standard Console:C++ Console Carbon. (Existují i jiné možnosti: místo Carbon můžete zvolit Classic nebo místo prostého C++ Console Carbon můžete zvolit C++ Console Carbon Altivec.

CodeWarrior dodá jako součást počátečního projektu malý zdrojový soubor. Tento program můžete zkusit přeložit a spustit, abyste zjistili, zda máte svůj systém správně nastavený. Nicméně po napsání vlastního kódu, byste měli tento soubor z projektu vymazat. Provedete to označením souboru v okně projektu a výběrem příkazů Project, Remove.

Dále musíte do projektu dodat zdrojový kód. Můžete použít příkaz New z nabídky File pro vytvoření nového souboru nebo příkaz Open z nabídky File pro otevření existujícího souboru. Použijte správnou příponu, jako je .cp nebo .cpp a prostřednictvím nabídky Project vložte zvolený soubor do seznamu projektu. Některé programy v této knize vyžadují vložení více než jednoho zdrojového souboru. Nakonec vyberte příkaz Project, Run.

> **TIP**
>
> **Abyste si ušetřili čas, můžete použít pro všechny vzorové programy pouze jeden projekt. Vymažte ze seznamu projektu předchozí zdrojové programy příkladu a přidejte další zdrojový kód. Tím ušetříte i prostor na disku.**

Tento překladač obsahuje ladicí program, který vám pomůže nalézt příčiny problémů za běhu programu.

Shrnutí

Jak se zvyšoval výkon počítačů, programy se zvětšovaly a byly složitější. V důsledku toho směřoval vývoj počítačových jazyků k jednodušší správě programovacího procesu. Do jazyka C přibyly takové prvky jako řídicí struktury a funkce, které usnadnily sledování běhu programu a zjednodušily možnost strukturovaného a modulárního přístupu. Přínosem jazyka C++ je podpora objektově orientovaného a generického programování, jež nabízí další možnosti modularity programů a zjednodušuje tvorbu opakovaně použitelného kódu, který šetří čas a zvyšuje spolehlivost programů.

Oblíbenost C++ vyústila do velkého počtu implementací na mnoha platformách: standard ISO/ANSI je základem pro jejich vzájemnou kompatibilitu. Jsou jím stanoveny vlastnosti, které by měl jazyk mít, určuje jeho chování a také chování standardních knihoven funkcí, tříd a šablon. Standard také nabízí jazyk přenositelný mezi různými počítačovými platformami a různými implementacemi.

Program v C++ vytvoříme tak, že vytvoříme jeden nebo více zdrojových souborů, které obsahují program vyjádřený prostředky jazyka C++. Zdrojové soubory jsou vlastně textové soubory, které je nutno přeložit a sestavit do strojového jazyka, z něhož se skládá spustitelný program. Často k těmto úkonům používáme IDE, jehož součástí je editor, kterým vytváříme zdrojový text, překladač a sestavovací program, jenž vytváří spustitelné soubory, a další prostředky, např. pro řízení projektu a ladění programů. Stejné úlohy můžeme ovšem zadávat prostřednictvím řádkových příkazů, jimiž se samostatně spouštějí jednotlivé nástroje.

KAPITOLA 2

Vzhůru do světa C++

Témata kapitoly:

- Vytvořit program v jazyce C++
- Znát obecný formát programu C++
- Používat direktivu #include
- Psát funkci main()
- Vypisovat výstup pomocí objektu cout
- Psát komentáře v jazyce C++
- Používat znak nového řádku endl
- Deklarovat a používat proměnné
- Zadávat vstup pomocí objektu cin
- Definovat a používat jednoduché funkce

Když stavíte nový dům, začínáte základy a hrubou stavbou. Pokud nemáte od začátku pevnou konstrukci, nastanou později potíže se zabudováním doplňků, jako jsou okna, dveře, rámy, pozorovací kopule a parketové taneční sály. Podobně byste měli při studiu jazyka počítačů začít pochopením základní struktury programu. Potom se můžete zaměřit na podrobnosti, jako jsou cykly a objekty. Tato kapitola poskytuje přehled základní struktury programu C++ a dotýká se některých důležitých témat – zejména funkcí a tříd – které kniha později vysvětluje mnohem důkladněji. (Myšlenka spočívá v postupném zavádění těch nejzákladnějších pojmů na cestě k obecnému pochopení, které přijde později.)

Úvod do C++

Začneme jednoduchým programem, který zobrazuje zprávu. Výpis 2.1 používá prostředek jazyka C++ `cout` (vyslovujte sí aut), který vytváří znakový výstup. Do zdrojového kódu je pro čtenáře vloženo několik poznámek; tyto řádky začínají znaky `//` a překladač je ignoruje. Jazyk C++ rozlišuje velké a malé znaky. To znamená, že musíte pečlivě dodržovat stejnou velikost písmen, jako je v příkladech.

V našem programu se například vyskytuje slovo `cout`. Jestliže napíšete `Cout` nebo `COUT`, překladač toto slovo nepřijme a nařkne vás z používání neznámých identifikátorů. (Překladač je také citlivý na pravopis, a tak vám neprojde `kout` ani `coot`.) Přípona souboru `cpp` běžně označuje program jazyka C++; může se ale stát, že musíte použít jinou příponu, jak je popsáno v kapitole 1 „Začínáme“.

Výpis 2.1: myfirst.cpp

```
// myfirst.cpp -- zobrazuje zprávu
#include <iostream>       // direktiva
preprocesoru
int main()                // hlavička funkce
{                         // začátek těla funkce
```

```
    using namespace std;                         // zpřístupní definice
    cout << "Pojdme se chvili venovat C++.";     // výstup
    cout << endl;                                // začátek nové řádky
    cout << "Nebudes litovat!" << endl;          // další výstup
    return 0;                                    // ukončuje main()
}                                                // konec těla funkce
```

KOMPATIBILITA

Pokud používáte starší překladač, možná budete nuceni napsat `#include <iostream.h>` místo `#include <iostream>`; v takovém případě byste měli vynechat i řádek `using namespace std`; To znamená, nahraďte

```
#include <iostream>          // budoucí způsob
```

zápisem

```
#include <iostream.h>        // pokud budoucnost ještě nenastala
```

a úplně vynechte řádek

```
using namespace std;         // také budoucnost
```

(Některé velmi staré překladače vyžadují `#include <stream.h>` místo `#include <iostream.h>`; jestliže máte takovýto starý překladač, měli byste si pořídit buďto novější překladač nebo starší knihu.) Změna z `iostream.h` na `iostream` je téměř současná a můžete narazit na překladače, v nichž ještě není implementována.

Některá prostředí, která využívají principu oken, vykonávají program v samostatném okně, které potom automaticky zavírají, jakmile program skončí. Jak jsme si řekli v kapitole 1, můžete donutit okno, aby zůstalo otevřené, dokud nestisknete libovolnou klávesu tím, že přidáte následující řádek kódu před příkaz `return`:

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

Do některých programů musíte vložit takovéto řádky dva. Více se o tomto kódu dozvíte v kapitole 4 „Složené typy".

Úpravy programu

Možná zjistíte, že musíte příklady této knihy upravovat, aby běžely na vašem systému. Dvě nejběžnější změny jsou zmíněny v první Poznámce ke kompatibilitě této kapitoly. První změna souvisí se standardy jazyka; pokud nemáte aktuální překladač, musíte vkládat iostream.h místo iostream a vynechat řádek namespace. Druhá je záležitostí programového prostředí; možná budete přidávat jeden nebo dva příkazy `cin.get()`, aby výstup programu zůstal na obrazovce a mohli jste si ho prohlédnout. Protože tyto úpravy platí stejně pro všechny příklady knihy, je výše uvedená Poznámka ke kompatibilitě jediným upozorněním. Nezapomeňte na ni! Následující Poznámky ke kompatibilitě vás upozorní na další možné úpravy, které možná budete muset udělat.

Pro napsání tohoto programu použijte vámi zvolený editor (nebo zdrojové kódy ze souborů dostupných na webu *www.samspublishing.com*) a pomocí překladače vytvořte spustitelný soubor, jak bylo popsáno v kapitole 1. Po spuštění přeloženého programu z výpisu 2.1 byste měli uvidět následující výstup:

```
Pojdme se chvili venovat C++.
Nebudes litovat!
```

Vstup a výstup v jazyce C

Pokud jste zvyklí programovat v jazyce C, možná vás překvapí přítomnost `cout` na místě funkce `printf()`. Jazyk C++ ve skutečnosti může používat `printf()`, `scanf()` a všechny další standardní funkce vstupu a výstupu jazyka C, pokud vložíte obvyklý hlavičkový soubor jazyka C `stdio.h`. Ale toto je kniha o C++, proto používejte nové prostředky vstupu a výstupu jazyka C++, které jsou v mnoha směrech oproti verzi C vylepšeny.

Program v jazyce C++ vytváříte ze stavebních bloků, které se nazývají *funkce*. Obvykle uspořádáváte program do hlavních úloh a potom navrhujete samostatné funkce, které tyto úlohy vykonávají. Příklad uvedený ve výpisu 2.1 je natolik jednoduchý, že se skládá pouze z jedné funkce nazvané `main()`. Příklad `myfirst.cpp` má následující prvky:

- Poznámky uvozené znaky `//`
- Direktivu preprocesoru `#include`
- Hlavičku funkce: `int main()`
- Direktivu `using namespace`
- Tělo funkce ohraničené složenými závorkami `{` a `}`
- Příkaz, který používá prostředek jazyka C++ `cout` pro zobrazení zprávy
- Příkaz return pro ukončení funkce `main()`

Nyní se podívejme na tyto různorodé prvky trochu podrobněji. Je dobré začít od funkce `main()`, protože některé prvky uvedené před touto funkcí, jako je direktiva preprocesoru, snáze pochopíte, když budete vědět, jak funkce `main()` pracuje.

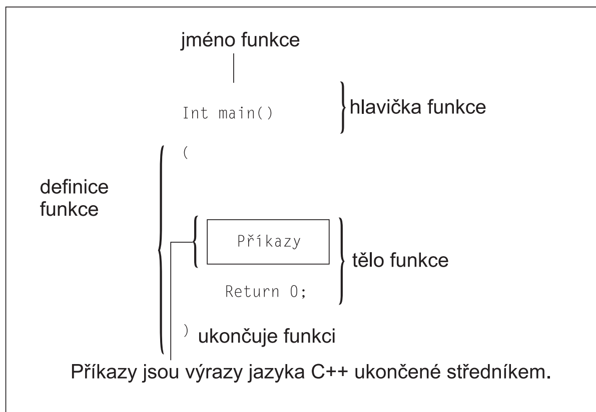
Funkce main()

Vzorový program uvedený ve výpisu 2.1 má následující zjednodušenou strukturu:

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

Tyto řádky říkají, že máte funkci, která se jmenuje `main()` a popisují její chování. Společně tvoří *definici funkce*. Tato definice má dvě části: První řádek `int main()`, kterému říkáme *hlavička funkce* a část uzavřenou do složených závorek (`{` a `}`), jež představuje *tělo funkce*. Funkci `main()` znázorňuje obrázek 2.1. Hlavičku funkce si můžeme představit jako stručný popis rozhraní funkce a zbytku programu. Tělo funkce představuje vaše instrukce pro počítač, které říkají, co by tato funkce měla provádět. V jazyce C++ se každá úplná instrukce nazývá *příkaz*. Každý příkaz musíte zakončit středníkem, takže při psaní příkladů nesmíte středníky vynechávat.

Poslední příkaz ve funkci `main()`, který se nazývá *příkaz návratu*, ukončuje funkci. O příkazu return se dozvíte více při čtení této kapitoly.



Obrázek 2.1: Funkce `main()`

Příkazy a středníky

Příkaz představuje úplnou počítačovou instrukci. Aby překladač vašemu kódu rozuměl, musí vědět, kde jeden příkaz končí a další začíná. Některé jazyky používají oddělovače příkazů. Fortran například odděluje příkazy koncem řádku. Pascal používá středník. V Pascalu můžete v určitých případech středník vynechat, například po příkazu bezprostředně před END, když vlastně dva příkazy neoddělujete. (Pragmatici a minimalisté by nesouhlasili s tím, že *může* znamená *měl by*.) Nicméně C++ stejně jako C používá spíše ukončovací znak než oddělovač. Ukončovacím znakem je zde středník, který označuje konec příkazu, přičemž se stává jeho součástí a nepředstavuje jen značku oddělující příkazy. V jazyce C++ platí, že středník nikdy nemůžete vynechat.

Hlavička funkce jako rozhraní

Nyní je nejdůležitější si zapamatovat, že syntaxe jazyka C++ vyžaduje, abyste definici funkce `main()` začínali takovouto hlavičkou: `int main()`. Tato kapitola se podrobnou syntaxí hlaviček funkcí zabývá později, ale zvědavější z vás si mohou přečíst následující předběžný výklad.

Obecně je funkce C++ aktivována nebo *volána* jinou funkcí a hlavička funkce popisuje rozhraní mezi funkcí a funkcí, která ji volá. Část uvedená před jménem funkce se nazývá *návratový typ funkce* a popisuje informační tok z volané do volající funkce. Část mezi závorkami za jménem funkce se nazývá *seznam argumentů* nebo *parametrů*, který popisuje informační tok z volající do volané funkce. Tento obecný formát je trochu zavádějící, pokud ho aplikujete na funkci `main()`, protože `main()` obvykle z jiné části vašeho programu nevoláte. Ve skutečnosti je funkce `main()` volána startovacím kódem, který přidává do vašeho programu překladač, jenž dělá prostředníka mezi programem a operačním systémem (UNIX, Windows XP nebo jiným). Hlavička funkce zde vlastně popisuje rozhraní mezi funkcí `main()` a operačním systémem.

Nejprve se podívejte na část int rozhraní funkce `main()`. Funkce jazyka C++ volaná jinou funkcí může této aktivující (volající) funkci vracet hodnotu, které říkáme *návratová hodnota*. V tomto případě může `main()` vracet celočíselnou hodnotu, jak je naznačeno klíčo-

vým slovem int. Dále si všimněte prázdných závorek. Obecně může funkce jazyka C++ předávat informaci jiné funkci, kterou volá. Část hlavičky funkce uzavřená v závorkách takovouto informaci popisuje. V našem případě prázdné závorky znamenají, že funkce `main()` nedostává žádnou informaci nebo, v obvyklé terminologii, `main()` nepřebírá žádné argumenty. (Když řekneme, že `main()` nebere žádné argumenty, neznamená to, že se jedná o nesmyslnou, autoritářskou funkci. Výraz *argument* používají počítačoví nadšenci, když mluví o informaci předávané z jedné funkce do druhé.)

Krátce řečeno, hlavička

```
int main()
```

říká, že funkce `main()` může vracet celočíselnou hodnotu funkci, která ji volá a že od ní nedostává žádné informace.

Mnoho existujících programů stále používá klasickou hlavičku jazyka C:

```
main()          // původní styl C
```

V jazyce C znamená vynechání návratového typu to samé, jako když se řekne, že funkce je typu int. Jazyk C++ však takovýto zápis postupně přestává používat.

Možná je i takováto varianta:

```
int main(void)  // přesně popisující styl
```

Použití klíčového slova `void` v závorkách je přesný způsob vyjádření toho, že funkce nedostává žádné argumenty. Pokud necháme v jazyce C++ (ne v C) závorky prázdné, je to totéž, jako bychom do nich vložili void. (Prázdné závorky v jazyce C znamenají, že se nezmiňujeme o tom, zda má funkce parametry nebo ne.)

Někteří programátoři používají tuto hlavičku a vynechávají příkaz `return`:

```
void main()
```

Je to logicky správné, protože návratový typ `void` vyjadřuje, že funkce nemá návratovou hodnotu. Tato varianta pracuje na mnoha systémech, ale protože není zakotvena v současných standardech, na některých systémech nemusí fungovat. Raději ji nepoužívejte a používejte standardní tvar; ruce vám kvůli tomu neupadnou.

Závěrem se ještě zmíníme, že standard ANSI/ISO jazyka C++ poskytuje těm, které psaní příkazu return na konec programu `main()` unavuje, určitou úlevu. Když překladač dojde na konec `main()` a nenajde tam příkaz `return`, tváří se, jakoby byl `main()` ukončen příkazem:

```
return 0;
```

Implicitní doplnění příkazu return platí pouze pro `main()`, nikoli pro ostatní funkce.

Proč se funkce main() nemůže jmenovat jinak

Existuje velmi pádný důvod proč pojmenovat funkci v programu `myfirst.cpp` jako `main()`: musí to tak být. Každý program jazyka C++ vyžaduje funkci, která se nazývá `main()`. (A nemůže to být ani `Main()` nebo `MAIN()` nebo `mane()`. Vzpomeňte si na nutnost dodržování správné velikosti a pořadí písmen.) Protože program myfirst.cpp má pouze jednu funkci, musí na sebe vzít tuto zodpovědnost ona a být funkcí `main()`. Vykonávání spuštěného programu jazyka C++ začíná vždy na začátku funkce `main()`. Bez funkce `main()` tedy není žádný program úplný a překladač vás upozorní, 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 i ostatní programy Windows. Protože modul DLL není nezávislý program, nepotřebuje funkci `main()`. Také programy pro specializovaná prostředí, jako jsou ovládací čipy robota, nemusí potřebovat funkci `main()`. Ale vaše samostatné programy funkci `main()` potřebují a právě takovýmito typy programů se zabývá tato kniha.

Komentáře v jazyce C++

V jazyce C++ je komentář uvozen dvojitým lomítkem (`//`). *Komentář* představuje poznámku programátora, která obvykle identifikuje oblast programu nebo vysvětluje určitý kód. Překladač komentáře ignoruje. Vždyť přece zná C++ alespoň tak dobře jako vy, ale komentářům rozumět nemůže. Pro překladač tedy výpis 2.1 vypadá, jako by byl napsán bez komentářů:

```
#include <iostream>
int main()
{
   using namespace std;
   cout << "Pojdme se chvili venovat C++.";
   cout << endl;
   cout << "Nebudes litovat!" << endl;
   return 0;
}
```

Komentáře začínají v C++ znaky `//` a pokračují do konce řádku. Komentář může být na samostatném řádku nebo na stejném řádku jako kód. Mimochodem, všimněte si prvního řádku výpisu 2.1:

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

V této knize začínají všechny programy komentářem, který udává jméno souboru zdrojového kódu a stručný popis programu. Jak bylo zmíněno v kapitole 1, přípona souboru zdrojového kódu závisí na vašem systému jazyka C++. Některé systémy mohou používat pojmenování `myfirst.C` nebo `myfirst.cxx`.

TIP

Pro dokumentování vašeho programu byste měli používat komentáře. Čím je program složitější, tím se tyto poznámky stávají cennějšími. Nejenže pomáhají jiným pochopit váš kód, ale také mohou být užitečné i vám, zvláště když jste delší dobu program neviděli.

Komentáře ve stylu jazyka C

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

```
#include <iostream> /* komentář ve stylu jazyka C */
```

Protože komentář stylu C je ukončen `*/` a ne koncem řádku, může se nacházet na více než jednom řádku. Ve vašich programech můžete používat oba styly. Snažte se však držet stylu C++, který způsobuje méně problémů, protože není nutné myslet na správné spárování začátečního znaku s koncovým. Není divu, že nový standard C99 přidal komentáře typu `//` i do jazyka C.

Preprocesor jazyka C++ a soubor iostream

Zde je uveden stručný popis toho, co potřebujete vědět. Bude-li váš program používat obvyklé prostředky vstupu nebo výstupu jazyka C++, vložte tyto dva řádky:

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

Pokud se vašemu překladači tyto řádky nelíbí (například když si stěžuje, že nemůže nalézt soubor `iostream`), zkuste místo nich napsat následující řádek:

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

To je opravdu vše, co potřebujete vědět, aby váš program pracoval, ale nyní se podíváme na věc trochu hlouběji.

Jazyk C++, stejně jako C, používá *preprocesor*. To je program, který zpracovává zdrojový soubor, dříve než přijde na řadu hlavní překlad. (Některé implementace jazyka C++, jak si možná pamatujete z kapitoly 1, používají transformační program, který převádí kód jazyka C++ do C. Ačkoli transformační program také představuje jistou formu preprocesoru, takovýmto preprocesorem se zde nebudeme zabývat a místo toho se zaměříme na preprocesor, který zpracovává direktivy, jejichž jména začínají znakem #.) Pro vyvolání tohoto preprocesoru nemusíte udělat nic zvláštního. Spustí se automaticky před překladem programu.

Výpis 2.1 používá direktivu `#include`:

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

Tato direktiva říká preprocesoru, aby do vašeho programu přidal obsah souboru `iostream`. Přidání nebo záměna textu zdrojového kódu před jeho překladem patří mezi typické akce preprocesoru.

Možná se ptáte, proč byste měli vkládat obsah souboru `iostream` do programu. Odpověď souvisí s komunikací programu s okolním světem. Předpona `io` v `iostream` označuje *vstup (input)*, což jsou informace dodávané programu, a *výstup (output)*, informace předávané z programu ven. Váš první program tyto definice potřebuje, aby mohl zobrazit zprávu pomocí objektu `cout`. Direktiva `#include` zajistí, aby byl obsah souboru `iostream` dodán spolu s obsahem vašeho souboru překladači. Obsah souboru iostream prakticky v programu nahradí řádku `#include <iostream>`. Váš původní soubor se nezmění, ale složený soubor vytvořený z vašeho zdrojového souboru a souboru iostream postupuje do další etapy překladu.

ZAPAMATUJTE SI

Do zdrojového kódu programu, který používá `cin` a `cout` pro vstup a výstup, musí být vložen soubor `iostream` (nebo v některých systémech `iostream.h`).

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 souboru). Překladače jazyka C++ jsou distribuovány s mnoha hlavičkovými soubory, z nichž každý podporuje určitou skupinu prostředků. V jazyce C bylo zvykem označovat hlavičkové soubory příponou `h` kvůli snadnému určení typu souboru podle jeho jména. Například

hlavičkový soubor math.h podporuje různé matematické funkce jazyka C. Původně se jazyk C++ choval stejně. Například hlavičkový soubor podporující vstup a výstup se jmenoval iostream.h. Nicméně nedávno se zvyklosti jazyka C++ změnily. Nyní je přípona h vyhrazena pro staré hlavičkové soubory jazyka C (které programy C++ mohou i nadále používat), zatímco hlavičkové soubory jazyka C++ jsou bez přípony. Některé hlavičkové soubory jazyka C byly převedeny na hlavičkové soubory jazyka C++. Tyto soubory vznikly vynecháním přípony h (což vytvořilo jméno stylu C++) a přidáním předpony c (ta naznačuje původ souboru v jazyce C). Například C++ verzi math.h představuje hlavičkový soubor cmath. Někdy jsou C a C++ verze hlavičkových souborů jazyka C shodné, zatímco v jiných případech může mít novější verze několik změn. U hlavičkových souborů určených pouze pro jazyk C++, jako je iostream, představuje vynechání přípony h mnohem více než jen kosmetickou úpravu, protože hlavičkové soubory bez přípony h navíc zavádějí jmenné prostory, jimiž se budeme zabývat v následujícím tématu tohoto přehledu. Tabulka 2.1 shrnuje konvence pojmenování hlavičkových souborů.

Tabulka 2.1 Konvence pojmenování hlavičkových souborů

Druh hlavičky	Konvence	Příklad	Poznámky
starý styl C++	končí na .h	iostream.h	Použitelné v programech jazyka C++
starý styl C	končí na .h	math.h	Použitelné v programech jazyků C, C++
nový styl C++	bez přípony	iostream	Použitelné v programech jazyka C++, používá jmenný prostor std
převedené C	předpona c	cmath	Použitelné v programech jazyka C++, bez přípony může používat rysy nepatřící C, jako je jmenný prostor std

Z hlediska zvyklostí jazyka C, který různé typy souborů označuje odlišnými příponami, se zdá rozumné mít nějakou specifickou příponu pro hlavičkové soubory jazyka C++. Komise standardu ANSI/ISO měla stejný názor. Problém spočíval v tom dohodnout se, jaká přípona to bude, a tak se nakonec shodli na souborech bez přípony.

Jmenné prostory

Vkládáte-li soubor iostream místo iostream.h, měli byste použít níže uvedenou direktivu jmenného prostoru, abyste vašemu programu zpřístupnili definice z iostream:

```
using namespace std;
```

Tomuto příkazu říkáme *direktiva using*. Prozatím bude nejjednodušší, když ji přijmete tak, jak je a její bližší prozkoumání necháte na později (například v kapitole 9 „Paměťové modely a jmenné prostory“). Jako základní informaci si můžete přečíst následující přehled.

Podpora jmenných prostorů je zcela novým rysem jazyka C++, který byl navržen pro zjednodušení psaní programů spojujících již existující kód několika dodavatelů. Problém nastane, když použijete dva produkty, které mají stejně pojmenovanou funkci, například wanda(). Jestliže takovou funkci zavoláte, překladač nebude vědět, kterou verzi potřebujete. *Jmenný prostor* umožňuje dodavateli zabalit své zboží tak, abyste pomocí jména jmenného prostoru mohli určit, od kterého dodavatele výrobek chcete. Například spo-

lečnost Microflop Industries by mohla umístit své definice do jmenného prostoru nazvaného `Microflop`. Potom by se výraz `Microflop::wanda()` mohl stát úplným jménem pro její funkci `wanda()`. Podobně by `Piscine::wanda()` označoval verzi funkce `wanda()` společnosti Piscine Corporation. Takže nyní by váš program mohl rozlišovat různé verze pomocí jmenných prostorů:

```
Microflop::wanda("tancujete?");          // použití verze jmenného prostoru
                                         // Microflop
Priscine::wanda("ryba jménem Touha");    // použití verze jmenného prostoru
                                         // Priscine
```


Do jmenného prostoru nazvaného `std` jsou stejným způsobem umístěny třídy, funkce a proměnné, které představují standardní komponenty překladačů jazyka C++. To platí pro hlavičkové soubory bez přípony `h`. To znamená, že například proměnná `cout`, která je používána pro výstup a definována v `iostream`, se tedy ve skutečnosti nazývá `std::cout` a že `endl` je skutečně `std::endl`. Můžete tedy vynechat direktivu `using` a program napsat takto:

```
std::cout << "Pojdme se chvili venovat C++.";
std::cout << std::endl;
```

Avšak většina uživatelů si nechce přidělávat starosti s převáděním kódu z doby před zavedením jmenných prostorů používajícího `iostream.h` a cout na kód se jmennými prostory, který pracuje s `iostream` a `std::cout`. Zde nastupuje direktiva `using`. Následující řádek zajistí, že můžete použít jména definovaná ve jmenném prostoru `std` bez předpony `std::`.

```
using namespace std;
```

Takto napsaná direktiva `using` zpřístupní všechna jména jmenného prostoru `std`. V současné praxi je takovýto přístup považován za poněkud pohodlný. Prostřednictvím deklarací `using` by měla být zpřístupněna pouze ta jména, která v kódu potřebujete:

```
using std::cout;                         // zpřístupní cout
using std::endl;                         // zpřístupní endl
using std::cin;                          // zpřístupní cin
```

Pokud umístíte tyto řádky na začátek souboru namísto

```
using namespace std;  // pohodlný způsob, jsou dostupná všechna jména
```

můžete používat `cin` a `cout` bez předpony `std::`. Pokud ale potřebujete z hlavičky `iostream` i jiná jména, musíte je jednotlivě doplnit do seznamu direktiv `using`. Z určitých důvodů v této knize nejprve používám pohodlný způsob. Jednak u malých programů není důležité, jaký použijete způsob správy jmenného prostoru, a zadruhé bych při výuce C++ chtěl zdůraznit především základní aspekty jazyka. Později budu pro správu jmenného prostoru používat i jiné mechanismy.

Výstup jazyka C++ pomocí cout

Nyní se podíváme na to, jak je možné zobrazit zprávu. Program `myfirst.cpp` používá následující příkaz jazyka C++:

```
cout << "Pojdme se chvili venovat C++.";
```

Část uzavřená do dvojitých uvozovek je zpráva pro tisk. V C++ se jakákoli skupina znaků uzavřená do dvojitých uvozovek nazývá *řetězec znaků* zřejmě proto, že obsahuje několik znaků zřetězených do většího celku. Zápis `<<` naznačuje, že příkaz posílá řetězec na `cout`; tyto symboly ukazují směr toku informací. A co je `cout`? Za tímto slovem se skrývá

předdefinovaný objekt, který ví, jak zobrazit různé věci včetně řetězců, čísel a jednotlivých znaků. (Objektem, jak si můžete pamatovat z kapitoly 1, je určitá instance třídy a třída definuje uložení a použití dat.)

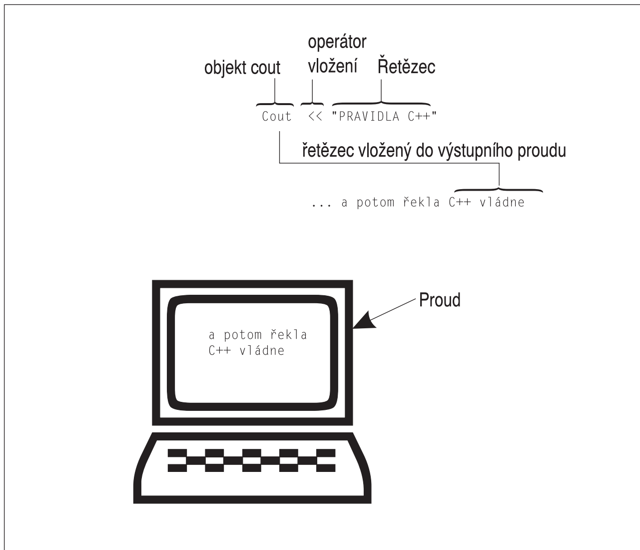
Prozatím to může být trochu obtížněji pochopitelné. Objekty budeme probírat až po několika kapitolách, nicméně jeden zde použít musíte. Tato skutečnost ovšem odhaluje jednu ze silných stránek objektů. Nemusíte vědět, jak objekt pracuje, abyste ho mohli používat. Potřebujete pouze znát jeho rozhraní – to znamená, jak ho používat. Objekt `cout` má jednoduché rozhraní. Jestliže retezec reprezentuje řetězec, můžete ho zobrazit následujícím příkazem:

```
cout << retezec;
```

Pro zobrazení řetězce nic víc vědět nepotřebujete. Ale nyní se podívejme na to, jakými pojmy jazyk C++ tento proces popisuje. Z tohoto pohledu je výstupem proud – to znamená řada znaků, která teče z programu. Objekt `cout`, jehož vlastnosti jsou definovány v souboru iostream, tento proud reprezentuje. Vlastnosti objektu `cout` obsahují operátor vložení (<<), který vkládá informaci na své pravé straně do proudu. Tedy příkaz (všimněte si ukončujícího středníku)

```
cout << "Pojdme se chvili venovat C++.";
```

vkládá řetězec „Pojdme se chvili venovat C++." do výstupního proudu. Pokud se tedy chcete vyjadřovat přesněji, můžete říci, že váš program vkládá řetězec do výstupního proudu, a ne že zobrazuje zprávu. Zní to jaksi působivěji (viz obrázek 2.2).



Obrázek 2.2 Zobrazení řetězce pomocí cout

První setkání s přetíženým operátorem

Pokud přecházíte do C++ z jazyka C, pravděpodobně jste si všimli, že operátor vložení (<<) vypadá stejně jako operátor bitového posuvu doleva (<<). Toto je příklad *přetížení operátoru*, podle kterého může mít stejný symbol operátoru různé významy. Překladač rozpozná, o který význam se jedná z aktuálních souvislostí. I jazyk C má některé operátory přetížené. Například symbol & představuje jak operátor adresování, tak bitový operátor AND. Symbol * zastupuje násobení a také dereferencuje ukazatel. Důležitá zde není přesná funkce těchto operátorů, ale to, že stejný symbol může mít více než jeden význam, jehož přesný význam určí překladač ze souvislostí. (I vy vlastně děláte to samé, když například určujete význam slova „špička" jako ostrý konec nože, zatížení telefonní sítě nebo stav vyvolaný nemírnou konzumací alkoholických nápojů.) Jazyk C++ rozšiřuje přetěžování operátorů tím, že vám umožňuje předefinovat významy operátorů pro uživatelsky definované typy zvané třídy.

Manipulační symbol endl

Nyní se podíváme na podivně vypadající zápis, který se vyskytuje ve druhém příkazu výstupu ve výpisu 2.1:

```
cout << endl;
```

`endl` představuje zvláštní zápis jazyka C++, jenž reprezentuje důležitý pojem, kterému říkáme *začátek nového řádku*. Když vsuneme `endl` do výstupního toku, kurzor na obrazovce se přemístí na začátek následujícího řádku. Speciální zápisy jako `endl`, které mají určitý význam pro `cout`, nazýváme *manipulačními symboly*. Podobně jako `cout` je `endl` definován v hlavičkovém souboru `iostream` jako součást jmenného prostoru `std`.

Všimněte si, že při tisku řetězce neprovádí `cout` přesun na další řádek automaticky, takže první příkaz `cout` ve výpisu 2.1 zanechá kurzor umístěný hned za tečkou na konci výstupního řetězce. Výstup příkazu `cout` začíná tam, kde skončil poslední výstup, takže kdybychom vynechali `endl`, výstup by vypadal takto:

```
Pojdme se chvili venovat C++.Nebudes litovat!
```

Všimněte si, že písmeno N následuje bezprostředně za tečkou. Podívejme se na jiný příklad. Spustíte-li program:

```
cout << "Dobry nastroj, spatny";
cout << "nastroj ";
cout << "a ukulele";
cout << endl;
```

vypíše:

```
Dobry nastroj, spatnynastroj a ukulele
```

Znovu si všimněte, že začátek nového řetězce následuje bezprostředně za koncem předchozího řetězce. (Poznamenejme, že kdybyste chtěli tento příklad na výstupy vyzkoušet, museli byste z něj udělat úplný program s funkční hlavičkou `main()` a levou a pravou složenou závorkou.)

Znak nového řádku

V C++ existuje i starší způsob ukončení řádku na výstupu – zápis pochází z jazyka C a má tvar `\n`:

```
cout << "Co bude dál?\n";
```

Kombinace znaků `\n` se považuje za jeden znak a nazývá se *nový řádek* (*new line*).

Při psaní řetězce má toto ukončení řádku méně znaků než `endl`:

```
cout << "Jupiter je velka planeta.\n";       // Vypiš větu a jdi na nový řádek
cout << "Jupiter je velka planeta." << endl;// Vypiš větu a jdi na nový řádek
```

Na druhé straně, chcete-li generovat samostatný nový řádek, v obou případech musíte zadat stejný počet znaků. Většina lidí považuje zadávání konce řádku ve tvaru `endl` za pohodlnější.

```
cout << "\n";                                // Začni nový řádek
cout << endl;                                // Začni nový řádek
```

V naší knize obvykle používáme nový řádek (`\n`), je-li součástí uzávorkovaného řetězce, jinak v ostatních případech používáme `endl`.

Znak nový řádek je jedním z příkladů na speciální kombinaci znaků, kterou nazýváme „escape sekvence“; podrobněji si tyto kombinace probereme v kapitole 3, „Práce s daty“.

Formátování zdrojového kódu jazyka C++

Některé jazyky, jako je Fortran, jsou řádkově orientované, což znamená, že mají na řádku jen jeden příkaz. U takovýchto jazyků odděluje příkazy znak návratu vozíku (carriage return). Ale v jazyce C++ označuje konec každého příkazu středník. To jazyku C++ umožňuje nakládat se znakem návratu vozíku stejně jako s mezerou nebo tabulátorem. Což znamená, že v C++ můžete udělat mezeru tam, kde byste ukončili řádek a naopak. Jeden příkaz tedy můžete rozepsat na několik řádků nebo umístit několik příkazů na jeden řádek. Soubor myfirst.cpp byste například mohli naformátovat následovně:

```
#include <iostream>
      int
main
() {  using
   namespace
         std; cout
            <<
"Pojdme se chvili venovat C++."
;     cout <<
endl; cout <<
"Nebudete litovat" <<
endl;return 0; }
```

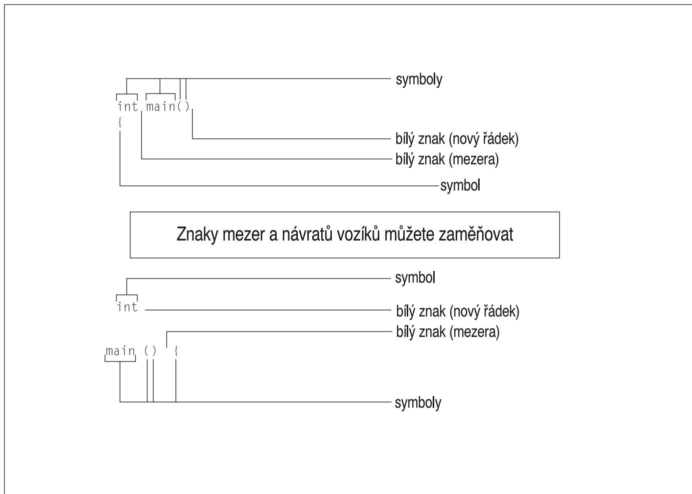
Toto je na pohled hrozný, ale platný kód. Nicméně z něj lze vypozorovat některá pravidla. V jazycích C a C++ nemůžete vložit mezeru, tabulátor nebo nový řádek doprostřed prvku, jako je jméno a stejně tak nemůžete umístit znak návratu vozíku do řetězce. V následujícím příkladu jsou chyby, kterých se nesmíte dopustit:

```
int ma in() // NEPLATNÉ -- mezera ve jménu
re
turn 0;      // NEPLATNÉ -- slovo na více řádcích
cout << "Pohledte na zarodky
krasy! ";    // NEPLATNÉ -- řetězec na více řádcích
```

Symboly a bílé znaky

Nedělitelné prvky v řádku kódu se nazývají *symboly* (tokens) (viz obrázek 2.3). Obecně platí, že musíte oddělit jeden symbol od dalšího mezerou, tabulátorem nebo návratem vozíku, které se společně nazývají *bílé znaky*. Některé samostatné znaky, například závorky a čárky, jsou symboly, jež nemusí být oddělovány bílými znaky.

```
return0;        // NEPLATNÉ, musí být return 0;
return(0);      // PLATNÉ, bílý znak je vynechán
return (0);     // PLATNÉ, bílý znak je použit
int main()      // PLATNÉ, bílý znak je vynechán v ()
int main ( )    // TAKÉ PLATNÉ, bílý znak je použit v ( )
```



Obrázek 2.3 Symboly a bílé znaky

Styl zdrojového kódu jazyka C++

Ačkoli vám jazyk C++ poskytuje mnoho volnosti při formátování zdrojového kódu, vaše programy budou čitelnější, pokud budete dodržovat rozumný styl. S platným, ale na pohled hrozným, kódem byste neměli být spokojeni. Většina programátorů používá styl předvedený ve výpisu 2.1, který se řídí následujícími pravidly:

- Jeden příkaz na řádku.
- Otevírací a uzavírací složená závorka funkce jsou na samostatném řádku.
- Příkazy ve funkcích jsou odsazené od složených závorek.
- Kolem závorek, které patří ke jménu funkce, nejsou bílé znaky.

První tři pravidla si kladou za cíl udržet kód čistý a čitelný. Čtvrté pomáhá odlišit funkce od některých vestavěných struktur jazyka C++, jako jsou cykly, jež také používají závorky. Kniha vás upozorní na ostatní pravidla, jakmile se objeví.

Příkazy jazyka C++

Program jazyka C++ se skládá z funkcí a každá funkce obsahuje příkazy. C++ má několik druhů příkazů a tak se podívejme na některé možnosti. Výpis 2.2 ukazuje dva druhy příkazů. První, deklarační, příkaz vytváří proměnnou. Druhý, přiřazovací, příkaz dává této proměnné hodnotu. Program také předvádí nové možnosti cout.

Výpis 2.2 carrots.cpp

```
// carrots.cpp .. food processing program
// uses and displays a variable

#include <iostream>

int main()
{
   using namespace std;

   int carrots;                  // deklarace celočíselné proměnné

   carrots = 25;                 // přiřazení hodnoty proměnné
   cout << "Mam ";
   cout << carrots;              // výpis hodnoty proměnné
   cout << " mrkvi.";
   cout << endl;
   carrots = carrots - 1;        // modifikace proměnné
   cout << "Krup, krup. Ted mam " << carrots << " mrkvi." << endl;
   return 0;
}
```

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

```
Mam 25 mrkvi.
Krup, krup. Ted mam 24 mrkvi.
```

Tímto programem se budeme zabývat na následujících stránkách.

Deklarační příkazy a proměnné

Počítače jsou přesné, systematické stroje. Chcete-li do počítače uložit nějakou informaci, musíte zadat umístění úložiště a množství paměťového prostoru, který daná informace vyžaduje. Poměrně jednoduchým způsobem lze zadat typ úložiště a označit jeho umístění pomocí *deklaračního příkazu*. Například výše uvedený program obsahuje následující deklarační příkaz (všimněte si středníku):

```
int carrots;
```

Tento příkaz deklaruje, že program poskytne dostatečné úložiště pro to, co je nazváno slovem `int`. O podrobnosti přidělení a označení paměti se postará překladač. Jazyk C++ může pracovat s několika druhy nebo typy dat a `int` je nejzákladnějším datovým typem. Představuje celé číslo bez desetinné čárky. V jazyce C++ může být typ `int` kladný nebo záporný, ale rozsah velikosti závisí na implementaci. Kapitola 3, „Práce s daty“, popisuje `int` a další základní typy podrobněji.

Kromě přidělení typu deklarační příkaz prohlašuje, že od nynějška bude program používat jméno `carrots` pro označení hodnoty uložené na daném místě. Výrazu `carrots` říkáme *proměnná*, protože může měnit svou hodnotu. Kdybyste v souboru `carrots.cpp` vynechali deklaraci, překladač by ohlásil chybu, jakmile by se program pokusil s proměnnou `carrots` pracovat. (Můžete si vyzkoušet vynechat deklaraci, abyste viděli, jak překladač odpovídá. Pokud někdy takovouto odpověď v budoucnosti uvidíte, budete umět zkontrolovat chybějící deklarace.)

Proč musí být proměnné deklarovány

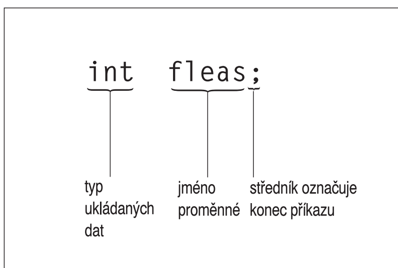
Některé jazyky, zejména Basic, vytvářejí proměnné bez explicitní deklarace, jakmile použijete nové jméno. To může být pro uživatele příjemné, a také za určitých podmínek je. Problém nastane, když se zmýlíte ve jménu proměnné. Nedopatřením můžete vytvořit novou proměnnou, aniž si to uvědomíte. To znamená, že v Basicu můžete udělat něco podobného, co popisuje následující kód:

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

Protože proměnná CastleDank je špatně napsána (písmeno *r* bylo zaměněno za *n*), změny, které s ní provedete, zanechají proměnnou CastleDark nezměněnou. Tento druh chyby se bude velmi těžko hledat, protože neporušuje žádná pravidla Basicu. V C++ však podobný kód poruší pravidlo o potřebě deklarovat proměnnou před jejím použitím, takže překladač tuto chybu zachytí a možný problém označí.

Obecně tedy deklarace určuje typ ukládaných dat a jméno, které pro ně bude program používat. V tomto případě program vytváří proměnnou se jménem carrots, do níž může uložit celé číslo. (Viz obrázek 2.4.)

Deklarační příkaz se v programu nazývá *definičním deklaračním* příkazem nebo krátce *definicí*. To znamená, že jeho výskyt způsobí, že překladač přidělí proměnné paměťový prostor. Ve složitějších případech můžete také narazit na *referenční deklarace*. Ty říkají počítači, aby použil proměnnou, která již byla definována někde jinde. Obecně platí, že deklarace nemusí být definicí, ale v tomto příkladu jí je.



Obrázek 2.4 Deklarace proměnné

Pokud znáte jazyk C nebo Pascal, tak jste se již jistě setkali s deklaracemi proměnných a možná budete trochu překvapeni. V C a Pascalu se deklarace proměnných obvykle umisťují na začátek funkcí nebo procedur, ale C++ podobná omezení nemá. Běžným programovacím stylem v jazyce C++ je deklarovat proměnné až před jejich prvním použitím. Potom nemusíte zpětně prohledávat program, abyste se dozvěděli, jakého je daná proměnná typu. Příklad na toto téma uvidíte v kapitole později. Tento styl má i nevýhodu. Protože nemáte všechny proměnné na jednom místě, nemůžete jednoduše zjistit, které proměnné vaše funkce používá. (Mimochodem, v C99 se teď zavádí téměř stejná pravidla pro deklaraci jako v C++.)

TIP

V jazyce C++ jsou proměnné deklarovány co nejblíže k jejich prvnímu použití.

Přiřazovací příkazy

Přiřazovací příkaz přiřazuje úložišti hodnotu. Například příkaz

```
carrots = 25;
```

přiřazuje celé číslo 25 místu, které je reprezentováno proměnnou carrots. Symbolu = říkáme *operátor přiřazení*. Neobvyklým rysem jazyka C++ (i C) je, že můžete operátor přiřazení použít za sebou. To znamená, že následující kód je platný:

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

Přiřazení probíhá zprava doleva. Nejprve je hodnota `88` přiřazena proměnné `steinwey`; potom hodnota `steinway`, která je nyní `88`, proměnné `baldwin` a nakonec je hodnota `88` proměnné `baldwin` přiřazena proměnné `yamaha`. (Jazyk C++ pokračuje v slabosti jazyka C pro povolení zvláštně vypadajícího kódu.)

Druhý přiřazovací příkaz ve výpisu 2.2 ukazuje, že hodnotu proměnné můžete změnit:

```
carrots = carrots - 1;   // změň hodnotu proměnné
```

Výraz napravo od operátoru přiřazení (`carrots - 1`) je aritmetický příklad – od čísla 25, což je hodnota proměnné `carrots`, se odečte 1 a výsledek je 24. Tato nová hodnota se uloží do `carrots`.

Nový trik objektu cout

Doposud byly v příkladech objektu `cout` předávány k tisku pouze řetězce. Výpis 2.2 navíc předává objektu `cout` proměnnou, jejíž hodnota je celočíselná.

```
cout << carrots;
```

Program netiskne slovo carrots, ale hodnotu 25 v proměnné carrots uloženou. Ve skutečnosti jsou zde provedeny dva triky v jednom příkazu. Nejprve objekt `cout` nahradí proměnnou `carrots` její aktuální číselnou hodnotou 25. Potom tuto hodnotu převede na správné výstupní znaky.

Jak jste si jistě všimli, objekt `cout` umí pracovat s řetězci i s celými čísly. Možná se vám to nezdá moc zajímavé, ale musíte si uvědomit, že celé číslo `25` představuje něco zcela

jiného než řetězec „25“. Řetězec obsahuje znaky, pomocí kterých číslo píšete, to znamená znak 2 a 5. Program vnitřně ukládá kódy znaků 2 a 5. Při tisku řetězce objekt `cout` jednoduše tiskne každý znak řetězce. Ale číslo `25` je uloženo jako numerická hodnota. Počítač neukládá každou číslici jednotlivě, ale má číslo 25 uloženo binárně. (Viz Příloha A, „Číselné soustavy“.) Důležité je, že `cout` musí přeložit číslo v celočíselném tvaru do znakové podoby předtím, než ho vytiskne. Navíc je objekt `cout` natolik chytrý, aby poznal, že se v proměnné `carrots` nachází celé číslo vyžadující konverzi.

Možná vám důmyslnost objektu `cout` přiblíží porovnání s kódem ve starším jazyce C. Pro vytisknutí řetězce „25“ a celého čísla 25 v jazyce C můžete použít víceúčelovou funkci výstupu `printf()`:

```
printf("Tisk retezce: %s\n", ""25");
printf("Tisk celeho cisla: %d\n", 25);
```

Aniž byste se zabývali podrobnostmi funkce `printf()`, je zřejmé, že musíte použít zvláštní kódy (`%s` a `%d`), které určují, zda se chystáte tisknout řetězec nebo celé číslo. Funkce `printf()` není natolik propracovaná, aby si všimla chyby, když ji požádáte o vytištění řetězce a omylem jí dodáte celé číslo. Pouze provede svůj úkol a zobrazí nesmysly.

Inteligentní způsob, jakým se chová objekt `cout`, má svůj původ v objektově orientovaných vlastnostech jazyka C++. Operátor vložení (<<) jazyka C++ v podstatě přizpůsobuje své chování typu dat, nad kterými pracuje. Toto je příklad přetížení operátoru. V dalších kapitolách, které popisují přetěžování funkcí a operátorů, se naučíte, jak můžete takovéto elegantní návrhy implementovat sami.

cout a printf()

Jestliže jste zvyklí na jazyk C a jeho funkci printf(), může se vám zdát chování objektu cout trochu podivné. Možná budete dokonce dávat přednost pracně nabytému ovládnutí funkce printf(). Ale cout ve skutečnosti nevypadá divněji než funkce printf() se všemi svými specifikacemi konverze. Důležitější je, že objekt cout poskytuje i značné výhody. Jeho schopnost rozeznat typy odráží inteligentnější a spolehlivější návrh. Navíc je rozšiřitelný. To znamená, že můžete operátor << předefinovat tak, aby objekt cout rozeznával a zobrazoval i nové vámi navržené datové typy. Pokud si pochvalujete pěkné ovládání, které poskytuje funkce printf(), můžete dosáhnout stejného výsledku pomocí pokročilých technik použití objektu cout (viz kapitola 17, „Vstup, výstup a soubory").

Další příkazy jazyka C++

Podívejme se na pár dalších příkladů příkazů. Program na výpisu 2.3 rozšiřuje předchozí příklad tím, že umožňuje zadávat hodnotu za běhu programu. K tomu používá objekt `cin` (vyslovujte sí in), což je protějšek objektu cout určený pro zadávání vstupu. Program také ukazuje další použití všestranného objektu `cout`.

Výpis 2.3 getinfo.cpp

```
// getinfo.cpp -- vstup a vystup
#include <iostream>

int main()
```

```
{
   using namespace std;

   int carrots;

   cout << "Kolik mas mrkvi?" << endl;
   cin >> carrots;                      // Vstup v C++
   cout << "Tady ti dve pridam. ";
   carrots = carrots + 2;
// v následujícím řádku je výstup zřetězen
   cout << "Ted mas " << carrots << " mrkvi." << endl;
   return 0;
}
```

Zde je příklad výstupu z programu ve výpisu 2.3:

```
Kolik mas mrkvi?
12
Tady ti dve pridam. Ted mas 14 mrkvi.
```

Program představuje dva nové prvky: používá `cin` ke čtení vstupu z klávesnice a spojuje čtyři výstupní příkazy do jednoho. Podívejme se na to.

Použití objektu cin

Jak ukazuje výstup, hodnota zadaná z klávesnice (12) je nakonec přiřazena proměnné `carrots`. Zde je příkaz, který tento zázrak provádí:

```
cin >> carrots;
```

Když si výše uvedený příkaz prohlédnete, můžete téměř vidět informaci tekoucí z objektu `cin` do proměnné `carrots`. Existuje přirozeně i poněkud oficiálnější popis tohoto procesu. Stejně jako jazyk 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 tento proud reprezentuje. Při výstupu vkládá operátor << znaky do výstupního proudu. Při vstupu používá objekt `cin` pro získání znaků ze vstupního proudu operátor >>. Na pravou stranu operátoru obvykle umisťujeme proměnnou, do které je získaná informace uložena. (Symboly << a >> byly vybrány tak, aby vizuálně připomínaly směr toku informací.)

Podobně jako `cout` je i `cin` chytrým objektem. Převádí vstup, jenž je pouze řadou znaků zadaných z klávesnice, do tvaru, který odpovídá proměnné přijímající informaci. Program v tomto příkladě deklaroval celočíselnou proměnnou `carrots`, takže je vstup převeden do číselného tvaru, který počítač používá pro ukládání celých čísel.

Zřetězení s cout

Druhým novým prvkem programu `getinfo.cpp` je spojení čtyř výstupních příkazů do jednoho. Soubor `iostream` definuje operátor << tak, abyste mohli výstup takto sloučit (zřetězit):

```
cout << "Ted mas ", " << carrots << " mrkvi." << endl;
```

Tím vám umožňuje spojit výstup řetězce a celého čísla do jediného příkazu. Výsledný výstup je stejný, jako při použití následujícího kódu:

```
cout << "Ted mas ";
cout << carrots;
```

```
cout << " mrkvi."
cout << endl;
```

Máte-li ještě náladu na radu týkající se objektu `cout`, můžete zřetězenou verzi přepsat tak, že jeden příkaz rozložíte na čtyři řádky:

```
cout  << "Ted mas "
      << carrots
      << " mrkvi."
      << endl;
```

Je to možné proto, že pravidla volného formátování jazyka C++ považují nové řádky a mezery mezi symboly za zaměnitelné. Tento poslední postup je vhodný tehdy, když je omezena šířka řádku.

Za zmínku stojí i to, že

```
Ted mas 14 mrkvi.
```

je na stejném řádku jako

```
Tady ti dve pridam.
```

Důvodem je skutečnost, jak už jsme se zmínili shora, že výstup příkazu `cout` následuje bezprostředně za předchozím výstupem `cout`, což platí, i když jsou mezi příkazy `cout` jiné příkazy.

cin a cout: První seznámení s třídami

O objektech cin a cout jste se toho již dozvěděli dost, takže si můžeme objekty probrat trochu podrobněji. Především se více zaměříme na pojem třída, který představuje jeden ze základů objektově orientovaného programování (OOP) v jazyce C++.

Třída je uživatelem definovaný datový typ. Chcete-li definovat třídu, musíte popsat, jaký druh informací zastupuje a co můžete s těmito daty dělat. Třída má stejný vztah k objektu jako typ k proměnné. To znamená, že definice třídy popisuje formát dat a jejich použití, zatímco objekt představuje entitu vytvořenou podle specifikace formátu dat. Pomocí nepočítačových výrazů můžeme říci, že pokud bude třída obdobou kategorie, jako jsou známí herci, potom může objekt představovat některý zástupce této kategorie, například žába Kermit. Abychom rozšířili toto přirovnání, třída reprezentující herce by měla obsahovat definice jejich možných činností, jako je studium textu, vyjádření smutku, zdůraznění nebezpečí, přijetí ocenění a jim podobných. Jestliže jste se setkali s různou terminologií OOP, možná vám pomůže, když budete vědět, že třídy jazyka C++ odpovídají tomu, co jiné jazyky nazývají *objektovým typem* a objekty jazyka C++ představují instanci objektu nebo instanci proměnné.

Buďme nyní více konkrétní. Vzpomeňte si na tuto deklaraci proměnné:

```
int carrots;
```

Výše uvedený příkaz vytváří proměnnou (`carrots`), která má vlastnosti typu `int`. To znamená, že proměnná `carrots` může uchovávat celé číslo a může být používána určitými způsoby, například při sčítání a odčítání. Nyní se podívejme na `cout`. To je objekt vytvořený tak, aby měl vlastnosti třídy `ostream`. Definice třídy `ostream` (v souboru `iostream`) popisuje druh dat, která reprezentuje objekt `ostream` a operace, jež s ním nebo na něm můžeme provádět, jako jsou vložení čísla nebo řetězce do výstupního proudu. Podobně je objekt `cin` vytvořený s vlastnostmi třídy `istream`, která je také definována v `iostream`.

ZAPAMATUJTE SI

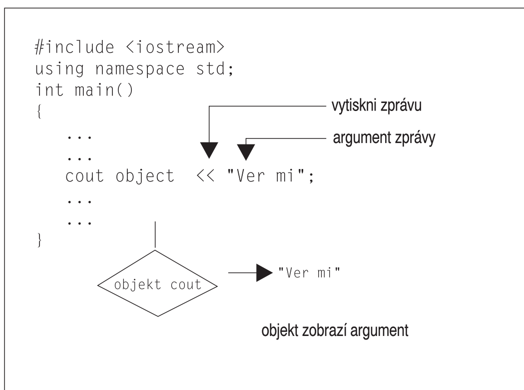
Třída popisuje všechny vlastnosti datového typu a objekt je entita vytvořená podle tohoto popisu.

Dozvěděli jste se, že třídy jsou uživatelsky definované typy, ale jako uživatelé jistě nebudete navrhovat třídy `ostream` a `istream`. Stejně jako jsou funkce distribuovány v knihovnách funkcí, mohou být třídy dodávány v knihovnách tříd. To je případ tříd `ostream` a `istream`. Ty nejsou technicky vestavěny do jazyka C++, ale představují příklady tříd, které jsou dodávány spolu s jazykem. Definice tříd jsou rozmístěny v souboru `iostream`, ale nejsou vestavěny do překladače. Pokud chcete, můžete tyto definice tříd i upravovat, to však není dobrý nápad. (Popravdě řečeno, je to hrozný nápad.) Rodina tříd `iostream` a příbuzná rodina `fstream` (nebo soubor I/O) jsou jedinou sadou definic tříd, která byla dodávána se všemi ranými implementacemi jazyka C++. Avšak výbor ANSI/ISO pro jazyk C++ přidal ke standardu ještě několik tříd. Většina implementací poskytuje další dodatečné definice tříd jako součást dodávky programového balíku. Oblíbenost jazyka C++ spočívá velkou měrou v existenci rozsáhlých a užitečných knihoven, které podporují programování pro operační systémy UNIX, Macintosh a Windows.

Popis třídy specifikuje všechny operace, které mohou být nad objekty dané třídy vykonány. Abyste s určitým objektem provedli některou povolenou činnost, pošlete mu zprávu. Když například chcete, aby objekt `cout` zobrazil řetězec, pošlete mu zprávu, která ve skutečnosti říká „Objekte! Zobraz toto!". C++ poskytuje několik způsobů zasílání zpráv. Jeden způsob, nazvaný použití metody třídy, je v podstatě voláním funkce, které jste již viděli. Druhý, používaný pro `cin` a `cout`, představuje předefinování operátoru. Tedy příkaz

```
cout << "Nejsem lump."
```

používá předefinovaný operátor << pro poslání zprávy „zobraz zprávu" objektu `cout`. V tomto případě zpráva přichází spolu s argumentem, kterým je řetězec, jenž se má zobrazit. (Viz obrázek 2.5.)



Obrázek 2.5 Poslání zprávy objektu

Funkce

Protože funkce představují moduly, ze kterých se skládají programy jazyka C++ a protože jsou důležité pro OOP definice C++, měli byste se s nimi podrobně seznámit. Jelikož některé stránky funkcí patří do pokročilejších témat, hlavní diskuse o funkcích přijde později v kapitole 7 „Funkce – programové moduly C++“ a v kapitole 8 „Rozšířené možnosti funkcí“. Nicméně, seznámíte-li se s některými základními charakteristikami funkcí již nyní, budete později ve výhodě. Zbytek kapitoly vás právě s těmito základy funkcí seznámí.

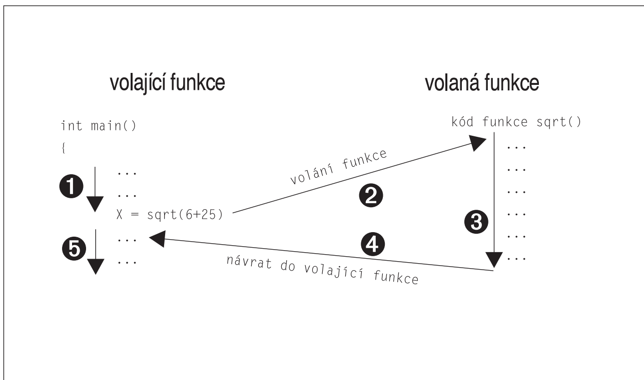
Existují dva druhy funkcí jazyka C++: s návratovou hodnotou a bez návratové hodnoty. Ve standardní knihovně funkcí jazyka C++ naleznete funkce obou druhů a můžete je i sami vytvářet. Podívejte se na knihovní funkci s návratovou hodnotu a potom vyzkoušejte, zda umíte napsat své vlastní jednoduché funkce.

Použití funkce, která má návratovou hodnotu

Funkce s návratovou hodnotu vytváří 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 přiřadit ji proměnné `x`. Ve svém programu můžete použít následující příkaz:

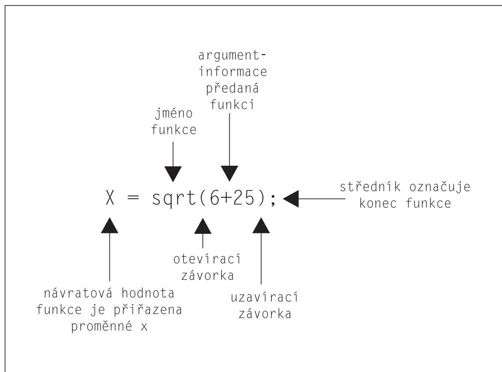
```
x = sqrt(6.25); // vrací hodnotu 2,5 a přiřazuje ji proměnné x
```

Výraz `sqrt(6.25)` vyvolává nebo *volá* funkci `sqrt()`. Výraz `sqrt(6.25)` nazýváme *volání funkce*, vyvolané funkci říkáme *volaná funkce* a funkci obsahující volání funkce označujeme termínem *volající funkce*. (Viz obrázek 2.6.)



Obrázek 2.6 Volání funkce

Hodnota v závorkách (v tomto příkladu 6.25) je informace posílaná funkci; říká se, že má být *předána* funkci. Tímto způsobem funkci posílaná hodnota se nazývá *argument* nebo *parametr*. (Viz obrázek 2.7.) Funkce `sqrt()` vypočítá odpověď, kterou je 2,5 a pošle tuto hodnotu zpět volající funkci; zpět poslaná hodnota se nazývá *návratová hodnota* funkce. Návratovou hodnotu si představte jako náhradu volání funkce v příkazu poté, co tato funkce skončí svou práci. Výše uvedený příklad tedy přiřazuje návratovou hodnotu proměnné `x`. Stručně řečeno, argument je informace předaná funkci a návratová hodnota je hodnota vrácená funkcí zpět.



Obrázek 2.7 Syntaxe volání funkce

To je k této záležitosti prakticky všechno, kromě toho, že překladač jazyka C++ musí před použitím funkce vědět, jaký druh argumentů funkce používá a jaký má druh návratové hodnoty. To znamená, vrací funkce celé číslo? Znak? Číslo s desetinnou částí? Rozsudek o vině? Nebo něco jiného? Pokud tato informace chybí, překladač neví, jak interpretovat návratovou hodnotu? Jazyk C++ tuto informaci vyjadřuje funkčním prototypem.

ZAPAMATUJTE SI

Program v jazyce C++ by měl ke každé funkci použité v programu poskytnout funkční prototyp.

Funkční prototyp představuje pro funkci to samé, co deklarace pro proměnnou – popisuje požadované typy. Knihovna jazyka C++ například definuje funkci `sqrt()`, která přebírá číslo (může být i) s desetinnou částí (zde je to 6,25) jako argument a vrací číslo stejného typu. Některé jazyky označují taková čísla jako reálná čísla, avšak jazyk C++ používá typ double. (O typu `double` se více dozvíte v kapitole 3.) Funkční prototyp funkce `sqrt()` vypadá následovně:

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

První double říká, že funkce `sqrt()` vrací hodnotu typu double. Typ `double` v závorkách udává, že funkce `sqrt()` očekává argument typu `double`. Tento prototyp tedy popisuje funkci `sqrt()` přesně tak, jak je použita v následujícím kódu:

```
double x;                    // deklaruje x jako proměnnou typu double
x = sqrt(6.25);
```

Ukončovací středník za prototypem označuje příkaz, čili vytvoří místo hlavičky funkce prototyp. Pokud vynecháte středník, překladač interpretuje řádek jako hlavičku funkce a očekává, že bude následovat tělo funkce, které funkci definuje.

Použijete-li funkci `sqrt()` v programu, musíte také dodat prototyp. To můžete udělat dvěma způsoby:

- Můžete sami napsat prototyp funkce do zdrojového souboru.
- Můžete vložit hlavičkový soubor `cmath` (`math.h` pro starší systémy), který potřebný prototyp obsahuje.

Druhý způsob je lepší, protože hlavičkový soubor dodá s větší pravděpodobností než vy správný prototyp funkce. Každá funkce knihovny C++ má prototyp v jednom nebo více hlavičkových souborech. Proto ověřte popis funkce v příslušné příručce nebo v nápovědě, pokud ji máte. Zde zjistíte, který hlavičkový soubor máte použít. V popisu funkce `sqrt()` byste se například dozvěděli, že máte použít hlavičkový soubor cmath. (Znovu připomínáme, že možná budete muset použít starší hlavičkový soubor `math.h`, který pracuje pro programy napsané v jazyce C i C++.)

Nezaměňujte funkční prototyp a definici funkce. Prototyp, jak jste viděli, pouze určuje rozhraní funkce. To znamená, že popisuje informaci předávanou funkci a informaci získávanou zpět. Naproti tomu definice obsahuje kód práce funkce – například kód pro výpočet druhé odmocniny čísla. Jazyky C a C++ v knihovnách funkcí tyto dva pojmy – prototyp a definice – od sebe oddělují. Knihovní soubory obsahují přeložený kód funkcí, zatímco hlavičkové soubory obsahují prototypy.

Funkční prototyp byste měli umístit před první použití funkce. Obvyklá praxe je vložit prototyp právě před funkci `main()`. Výpis 2.4 demonstruje použití knihovní funkce `sqrt()`; prototyp poskytuje vložením souboru `cmath`.

Výpis 2.4 sqrt.cpp

```
// sqrt.cpp -- použití funkce sqrt()
#include <iostream>
#include <cmath>      // nebo math.h

int main()
{
   using namespace std;

   double area;
   cout << "Zadejte vymeru podlahy sveho domu ve ctverecnich metrech: ";
   cin >> area;
   double side;
   side = sqrt(area);
   cout << "To je ekvivalent ctverce o strane " << side
        << " metru." << endl;
   cout << "Uzasne!" << endl;
   return 0;
}
```

KOMPATIBILITA

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

Použití knihovních funkcí

Funkce knihovny jazyka C++ jsou uloženy v knihovních souborech. Překladač musí při překladu programu v knihovních souborech hledat vámi použité funkce. Překladače se liší v tom, které knihovní soubory prohledávají automaticky. Když zkusíte spustit výpis 2.4 a obdržíte zprávu, že `_sqrt` je nedefinovaná vnější proměnná, překladač pravděpodobně neprohledal matematickou knihovnu automaticky. (Překladače přidá-

vají ke jménu funkce předponu ve tvaru znaku podtržítko – další jemná připomínka toho, že mají poslední slovo.) Pokud dostanete takovou zprávu, zkontrolujte dokumentaci, zda překladač prohledává správnou knihovnu. Obvyklá implementace na Unixu například vyžaduje, abyste na konci příkazové řádky zadali volbu `-lm` (zkratka slov *l*ibrary *m*ath):

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

Podobně vypadá i příkaz pro spuštění GNU překladače v Linuxu:

```
g++ sqrt.C -lm
```

Pouhé vložení hlavičkového souboru `cmath` poskytne prototyp, ale nemusí nutně přesvědčit překladač, aby prohledal správný knihovní soubor.

Program uvedený ve výpisu 2.4 po spuštění vypíše:

```
Zadejte vymeru podlahy sveho domu ve ctverecnich metrech: : 1536
To je ekvivalent ctverce o strane 39.1918 metru.
Uzasne!
```

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

```
jméno_typu jméno_proměnné;
```

Typ double umožňuje proměnným `area` a `side`, aby obsahovaly hodnotu s desetinnou částí, například 1536 a 39,1918. V proměnné typu `double` je i celé číslo 1536 uloženo jako reálná hodnota s desetinnou částí 0. Jak uvidíte v kapitole 3, typ `double` obsáhne mnohem větší rozsah hodnot než typ `int`.

Jazyk C++ umožňuje deklarovat nové proměnné kdekoli v programu, takže `sqrt.cpp` nedeklaruje proměnnou `area` dříve, než ji použijete. Dále vám jazyk C++ dovoluje přiřadit proměnné hodnotu při jejím vytváření, takže byste také mohli napsat:

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

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

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

Varianty funkcí

Některé funkce vyžadují více než jednu informaci. Tyto funkce používají vícenásobné argumenty oddělené čárkami. Například funkce `pow()` z knihovny math přebírá dva argumenty a vrací hodnotu, která je rovna prvnímu argumentu umocněnému hodnotou druhého.

Má tento prototyp:

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

Chcete-li například vypočítat 5^8 (5 na osmou), zavoláte funkci takto:

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

Jiné funkce nemají žádné argumenty. Například jedna z knihoven jazyka C (spojená s hlavičkovým souborem `cstdlib` nebo `stdlib.h`) obsahuje funkci `rand()`, která nemá žádné argumenty a vrací náhodné celé číslo. Její prototyp vypadá takto:

```
int rand(void);    // prototyp funkce, která nepřebírá žádné argumenty
```

Klíčové slovo `void` explicitně oznamuje, že funkce nemá žádné parametry. Jestliže vynecháte `void` a necháte závorky prázdné, jazyk C++ to interpretuje jako implicitní vyjádření funkce bez argumentů. Tuto funkci můžete volat následujícím způsobem:

```
myGuess = rand();  // volání funkce bez argumentů
```

Všimněte si, že na rozdíl od některých programovacích jazyků musíte napsat při volání funkce závorky, i když tato funkce nemá žádné parametry.

Také existují funkce, které nemají návratovou hodnotu. Předpokládejme, že jste například napsali funkci, která zobrazuje číslo ve formátu měny jako dolary a centy. Této funkci předáte argument, řekněme 23,5 a ona by měla na obrazovce zobrazit $23,50. Protože funkce posílá hodnotu na obrazovku místo volajícímu programu, nejedná se o návratovou hodnotu. V prototypu tuto situaci znázorníte tak, že místo typu návratové hodnoty napíšete klíčové slovo `void`:

```
void bucks(double);          // prototyp funkce bez návratové hodnoty
```

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

```
bucks(1234.56);              // volání funkce bez návratové hodnoty
```

Některé jazyky používají výraz *funkce* pro funkce s návratovou hodnotou a výrazy *procedura* nebo *subrutina* pro funkce bez návratové hodnoty, ale C++, podobně jako C, označuje výrazem *funkce* obě varianty.

Funkce definované uživatelem

Standardní knihovna jazyka C poskytuje více než 140 předdefinovaných funkcí. Pokud některá vyhovuje vašim požadavkům, rozhodně ji použijte. Avšak často musíte napsat svoji vlastní, zvláště když navrhujete třídy. Mimochodem, při návrhu vlastních funkcí si užijete mnohem více zábavy, takže si nyní tento proces můžete vyzkoušet. Již jste používali několik uživatelsky definovaných funkcí, ale všechny se jmenovaly `main()`. Každý program napsaný v jazyce C++ musí mít definovanou funkci `main()`, kterou musí definovat uživatel. Předpokládejme, že chcete přidat další vlastní funkci. Stejně jako u knihovní funkce můžete funkci definovanou uživatelem volat pomocí jejího jména a také musíte před jejím použitím uvést funkční prototyp, což obvykle uděláte umístěním prototypu před definici `main()`. Novým prvkem je, že také musíte dodat zdrojový kód nové funkce. Nejjednodušším způsobem je umístění kódu do stejného souboru za kód `main()`. Výpis 2.5 tyto prvky objasňuje.

Výpis 2.5 ourfunct.cpp

```
// ourfunc.cpp -- definování vlastní funkce
#include <iostream>
void simon(int);            // funkční prototyp pro simon()

int main()
{
   using namespace std;
```

```
    simon(3);                 // volání funkce simon()
    cout << "Vyberte si cele cislo: ";
    int count;
    cin >> count;
    simon(count);             // opětovné volání
    cout << "Hotovo!" << endl;
    return 0;
}

void simon(int n)             // definice funkce simon()
{
    using namespace std;
    cout << "Simon rika, abyste se dotknul prstu u nohou " << n << " krat. "
    << endl;
}                             // funkce void nemusí být ukončeny příkazem
return
```

Funkce `main()` volá dvakrát funkci `simon()`, jednou s argumentem 3 a jednou s argumentem reprezentovaným proměnnou `count`. Mezitím uživatel zadává celé číslo, které se používá pro nastavení hodnoty proměnné `count`. Příklad nepoužívá ve výzvě znak nového řádku. To má za následek, že se uživatelský vstup nachází na stejném řádku jako výzva. Program uvedený ve výpisu 2.5 po spuštění vypíše:

```
Simon rika, abyste se dotknul prstu u nohou 3 krat.
Vyberte si cele cislo: 512
Simon rika, abyste se dotknul prstu u nohou 512 krat.
Hotovo!
```

Tvar funkce

Definice funkce `simon()` má stejný obecný tvar jako definice funkce `main()`. Nejprve je uvedena hlavička funkce, potom v závorkách uzavřené tělo funkce. Tvar definice funkce můžeme zobecnit následovně:

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

Všimněte si, že zdrojový kód funkce `simon()` následuje za uzavírací složenou závorkou funkce `main()`. Podobně jako v C, a na rozdíl od Pascalu, C++ nedovoluje vkládat jednu definici funkce do druhé. Každá definice funkce je oddělená od ostatních; všechny funkce jsou vytvořeny jako rovnocenné. (Viz obrázek 2.8.)

Hlavičky funkcí

Funkce `simon()` ve výpisu 2.5 má tuto hlavičku:

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

Počáteční `void` naznačuje, že funkce `simon()` nemá návratovou hodnotu. Takže zavolání funkce `simon()` nevytvoří číslo, které byste mohli přiřadit proměnné ve funkci `main()`. Tedy první volání funkce vypadá takto:

```
simon(3);                     // v pořádku pro funkce s návratovým typem void
```

Protože chudák `simon()` nemá návratovou hodnotu, nemůžete použít tento způsob:

```
simple = simon(3);            // není povoleno pro funkce s návratovým typem void
```

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

prototypy funkcí

```
void simon(int);
double taxes(double);
```

funkce č.1

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

funkce č.2

```
void simon(int);
{
  ...
}
```

funkce č.3

```
double taxes(double t)
{
  ...
  return 2 * t;
}
```

Obrázek 2.8 Definice funkcí se nachází v souboru za sebou

Výraz `int n` v závorkách vyjadřuje, že máte volat funkci `simon()` s jediným argumentem typu `int`. Proměnné `n` se přiřadí nová hodnota předaná během volání funkce. Tedy volání funkce

```
simon(3);
```

přiřadí proměnné n definované v hlavičce `simon()` hodnotu 3. Když příkaz `cout` použije v těle funkce proměnnou `n`, pracuje vlastně s hodnotou předanou při volání funkce. Proto tedy `simon(3)` zobrazí na výstupu číslici 3. Volání `simon(count)` v příkladu běhu programu způsobí, že funkce zobrazí číslo 512, protože je to hodnota dodaná do `count`. Krátce řečeno, hlavička `simon()` říká, že funkce přebírá jediný argument typu `int` a že nemá návratovou hodnotu.

Podívejme se na hlavičku funkce `main()`:

```
int main()
```

Počáteční int vyjadřuje, že funkce `main()` vrací celočíselnou hodnotu. Prázdné závorky (které by nepovinně mohly obsahovat `void`) říkají, že `main()` nemá žádné parametry. Funkce, které mají návratové hodnoty, by měly vracet hodnotu a ukončovat funkci pomocí klíčového slova `return`. Proto jste na konci funkce `main()` psali následující příkaz:

```
return 0;
```

To je logicky správné: od funkce `main()` se očekává, že vrátí hodnotu typu `int` a vy ji přimějete vrátit celé číslo 0. Ale možná jste zvědaví, komu tuto hodnotu vracíte? Vždyť jste v žádném ze svých programů neviděli volání funkce `main()`:

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

Odpověď spočívá v tom, že si můžete operační systém vašeho počítače (řekněme UNIX nebo DOS) představit, jak volá váš program. Takže návratová hodnota `main()` není vrácena jiné části programu, ale operačnímu systému. Mnoho operačních systémů může využít návratovou hodnotu programu. Například příkazové soubory Unixu a dávkové soubory Dosu mohou být navrženy tak, aby vykonávaly programy a testovaly jejich návratové hodnoty, kterým se obvykle říká *výstupní hodnoty*. Podle běžné konvence nulová výstup-

ní hodnota vyjadřuje úspěšné proběhnutí programu, zatímco nenulová hodnota signalizuje nějaký problém. Můžete tedy navrhnout váš program v jazyce C++ tak, aby vracel nenulovou hodnotu, řekněme tehdy, když selže otevření souboru. Dále můžete navrhnout příkazový nebo dávkový soubor, který tento program vykoná a provede jisté alternativní akce, jestliže program signalizuje selhání.

Klíčová slova

Klíčová slova jsou slovníkem počítačového jazyka. V této kapitole jste použili čtyři klíčová slova jazyka C++: `int`, `void`, `return` a `double`. Protože tato klíčová slova mají pro jazyk C++ zvláštní význam, 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`. Příloha B, „Klíčová slova v jazyce C++", poskytuje úplný seznam klíčových slov C++. Mimochodem, `main` nepředstavuje klíčové slovo, protože není součástí jazyka. Jedná se o jméno požadované funkce. Takže `main` můžete použít jako jméno proměnné. (To však může způsobit problém za okolností, které jsou těžko srozumitelné, než abychom je zde popisovali, a protože je to v každém případě matoucí, raději to nedělejte.) Podobně nejsou klíčovými slovy ostatní jména funkcí a objektů. Avšak použití stejného jména, například `cout`, pro objekt i pro proměnnou v programu mate překladač. To znamená, že můžete použít `cout` jako jméno proměnné ve funkci, která nepoužívá objekt `cout` pro výstup, ale nemůžete použít `cout` oběma způsoby ve stejné funkci.

Jak se používá funkce definovaná uživatelem, která má návratovou hodnotu

Nyní se posuneme o krok dále a napíšeme funkci, která používá příkaz `return`. Funkce `main()` již přece ukazuje postup pro funkci s návratovou hodnotou: Dodejte hlavičce funkce návratový typ a na konci těla funkce napište příkaz `return`. Tento způsob můžete použít při řešení problému s váhovými jednotkami pro návštěvníky Spojeného království. Ve Spojeném království je mnoho osobních vah cejchováno ve váhových jednotkách kameny (*stone*), na rozdíl od USA, kde se používají libry nebo mezinárodně kilogramy. Jeden kámen váží 14 liber a program ve výpisu 2.6 používá funkci, která tuto konverzi provádí.

Výpis 2.6 convert.cpp

```
// convert.cpp -- konvertuje kameny na libry
#include <iostream>
int stonetolb(int);      // funkční prototyp
int main()
{
   using namespace std;
   int stone;
   cout << "Zadejte vahu v kamenech: ";
   cin >> stone;
   int pounds = stonetolb(stone);
   cout << stone << " kamenu je ";
   cout << pounds << " liber." << endl;
```

```
   return 0;
}

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

Program uvedený ve výpisu 2.6 po spuštění vypíše:

```
Zadejte vahu v kamenech: 14
14 kamenu je 196 liber.
```

Ve funkci `main()` používá program objekt `cin`, aby dodal celočíselné proměnné `stone` hodnotu. Tato hodnota je předána funkci `stonetolb()` jako argument a v této funkci je přiřazena proměnné `sts`. Funkce `stonetolb()` vrací pomocí klíčového slova `return` funkci `main()` hodnotu `14 * sts`. Výraz ukazuje, že za příkazem `return` nemusí vždy následovat pouze jednoduché číslo. Zde se použitím složitějšího výrazu vyhnete nutnosti vytvořit novou proměnnou a před jejím návratem jí přiřadit hodnotu. Program vypočítá hodnotu tohoto výrazu (v tomto případě 196) a vrátí výslednou hodnotu. Pokud se vám vracení hodnoty výrazu nelíbí, můžete se vydat delší cestou:

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

Obě verze vytvářejí stejný výsledek, ale druhé to trvá o něco déle.

Obecně platí, že funkci s návratovou hodnotou můžete umístit všude, kde byste mohli použít jednoduchou konstantu stejného typu. Funkce `stonetolb()` například vrací hodnotu typu `int`. To znamená, že tuto funkci můžete použít následujícími způsoby:

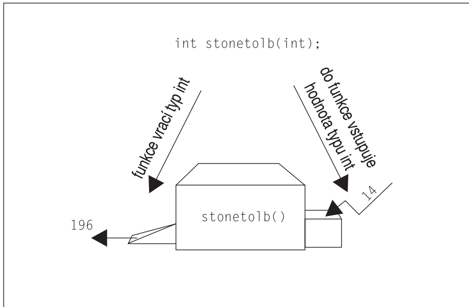
```
int aunt = stonetolb(20);
int aunts = aunt + stonetolb(10);
cout << "Ferda vazi " << stonetolb(16) << " liber." << endl;
```

V každém případě program spočítá návratovou hodnotu a potom toto číslo v uvedených příkazech použije.

Jak výše uvedené příklady ukazují, prototyp funkce popisuje rozhraní funkce, to znamená, jak funkce spolupracuje se zbytkem programu. Seznam argumentů ukazuje, jaký druh informace vstupuje do funkce a typ funkce udává typ návratové hodnoty. Programátoři často popisují funkce jako *černé skříňky* (výraz z elektroniky) specifikované tokem informací dovnitř a ven. Funkční prototyp tento pohled dokonale popisuje (viz obrázek 2.9).

Funkce `stonetolb()` je krátká a jednoduchá, přesto v ní nalezneme vše, co může funkce obsahovat:

- Má hlavičku a tělo.
- Přijímá argument.
- Vrací hodnotu.
- Vyžaduje prototyp.



Obrázek 2.9 Prototyp funkce a funkce jako černá skříňka

Funkci `stonetolb()` můžeme považovat za standardní tvar funkce. Podrobněji se budeme funkcemi zabývat v kapitolách 7 a 8. Do té doby by vám měla látka této kapitoly poskytnout dobrý přehled o tom, jak funkce pracují a jaké je jejich místo v jazyce C++.

Umístění direktivy using v multifunkčních programech

Ve výpisu 2.5 si všimněme, že direktivu `using` obsahují obě funkce:

```
using namspace std;
```

Důvodem je skutečnost, že obě funkce používají `cout` a musí tedy mít přístup k definici `cout` ze jmenného prostoru `std`.

Existuje i jiný způsob, jak zpřístupnit jmenný prostor `std` oběma funkcím uvedeným ve výpisu 2.5, a to umístit direktivu mimo obě funkce, konkrétně před ně:

```
// ourfunc1.cpp -- premistení direktivy using
#include <iostream>
using namespace std;     // definice všech funkcí jsou v tomto souboru
void simon(int);

int main()
{
   simon(3);
   cout << "Vyberte si cele cislo: ";
   int count;
   cin >> count;
   simon(count);
   cout << "Hotovo!" << endl;
   return 0;
}

void simon(int n)
{
   cout << "Simon rika, abyste se dotknul prstu u nohou " << n << " krat."
         << endl;
}
```

V současné době převládá názor, že je lepší funkce rozlišit a přístup k adresovému prostoru `std` omezit pouze na ty funkce, které jej využijí. Ve výpisu 2.6 používá cout pouze

main(), takže je zbytečné zpřístupňovat prostor std funkci stonetolb(). Direktiva using je tedy umístěna pouze ve funkci main() a přístup k ní je omezen jen na ni.

Shrneme-li to, můžeme si vybrat z několika možností, jak programům zpřístupnit prvky z adresového prostoru std. Zde jsou některé z nich:

- Direktiva

  ```
  using std namespace;
  ```

  může být v souboru umístěna před definicemi funkcí, čímž se celý obsah adresového prostoru std zpřístupní všem funkcím.

- Direktiva

  ```
  using std namespace;
  ```

  může být umístěna v definici určité funkce, takže adresový prostor std bude přístupný pouze této funkci.

- Místo direktivy

  ```
  using std namespace;
  ```

  lze v určité funkci použít direktivu např.

  ```
  using std::cout;
  ```

  čímž této funkci zpřístupníme z adresového prostoru std pouze určitý prvek, v tomto případě cout.

- Direktivu using můžeme zcela vynechat a prvek ze jmenného prostoru std označíme předponou std::

  ```
  std::cout << "I'm using cout and endl from the std namespace" << std::endl;
  ```


Z praxe: Konvence přidělování jmen

Programátoři jazyka C++ mají to štěstí (nebo smůlu), že si při pojmenovávání funkcí, tříd a proměnných mohou vybrat z velkého množství možností. Programátoři většinou mívají rozmanité a hluboce zakořeněné názory na styl programování, jejichž obhajování a prosazování bývá leckdy příčinou ostrých střetů na veřejných diskusních fórech. I když programátor vychází ze stejné základní myšlenky, může funkci pojmenovat některým z následujících způsobů:

```
MojeFunkce()
mojefunkce()
mojeFunkce()
moje_funkce()
moje_fun()
```

Výběr jména závisí na týmu vývojářů, zvláštností použitých technologií a knihoven, stejně jako na vkusu a volbě jednotlivých programátorů. Můžete zůstat klidní, protože, pokud jde o jazyk C++, jsou všechny povolené styly správné a mohu být použity podle vašeho uvážení.

Ponecháme-li stranou omezení jazyka, je dobré dodržovat svůj osobní styl pojmenovávání – musí být ovšem takový, aby pomáhal dodržovat důslednost a přesnost kódu. Přesné a zřetelné osobní konvence pojmenovávání jsou známkou dobrého návrhu softwaru a jistě budou přínosem pro vaši kariéru programátora.

Shrnutí

Program napsaný v jazyce C++ se skládá z jednoho nebo více modulů nazývaných funkce. Programy jsou vykonávány od začátku funkce `main()` (všechna písmena jsou malá), takže funkce tohoto jména by vám rozhodně neměla chybět. Funkci pak tvoří hlavička a tělo. Hlavička funkce říká, jaký druh návratové hodnoty, je-li nějaký, funkce vrací a jaký druh informace očekává, že jí bude předán prostřednictvím argumentů. Tělo funkce obsahuje příkazy jazyka C++ uzavřené do spárovaných složených závorek: (`{}`).

Typy příkazů jazyka C++ zahrnují:

- **Deklarační příkaz** – Příkaz pro zadání jména a typu proměnné ve funkci.
- **Přiřazovací příkaz** – Příkaz pro přiřazení hodnoty proměnné. Přiřazovacím operátorem je rovnítko (=).
- **Příkaz pro poslání zprávy** – Tímto příkazem se pošle určitému objektu zpráva, čímž se vyvolá nějaká činnost.
- **Volání funkce** – Předání řízení funkci. Po ukončení vrátí tato funkce řízení zpět do programu, odkud byla volána, na příkaz bezprostředně následující za voláním funkce.
- **Prototyp funkce** – Deklarace návratového typu funkce, počtu parametrů a jejich typu.
- **Příkaz návratu** – Vrácení hodnoty z volané funkce volající funkci.

Třída je uživatelsky definovaná specifikace datového typu. Tato specifikace podrobně popisuje, jak má být informace reprezentována a které operace mohou být s daty prováděny. Objekt je entita vytvořená podle popisu třídy, podobně jako je jednoduchá proměnná entitou vytvořenou podle popisu datového typu.

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

Jazyk C++ může používat množství knihovních funkcí jazyka C. Abyste mohli používat funkci z knihovny, měli byste vložit hlavičkový soubor s prototypem funkce.

Nyní, když máte celkový přehled o jednoduchých programech jazyka C++, můžete pokračovat následujícími kapitolami, abyste si doplnili podrobnosti a rozšířili obzory.

Otázky k opakování

Odpovědi na tyto a následující otázky můžete nalézt v příloze J, „Odpovědi na opakovací otázky".

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

```
#include <iostream>
```

3. Co provádí následující příkaz?

```
using namespace std;
```

4 Jakým příkazem byste vytiskli výraz „Nazdar, světe“ a nastavili nový řádek?
5. Jakým příkazem byste vytvořili celočíselnou proměnnou se jménem `cheeses`?
6. Jakým příkazem byste přiřadili hodnotu 32 proměnné `cheeses`?
7. Jakým příkazem byste načetli vstupní hodnotu z klávesnice do proměnné `cheeses`?
8. Jakým příkazem byste vytiskli „Máme X druhů sýra“, kde hodnota proměnné `cheeses` nahradí proměnnou `X`?
9. Co vám o funkci říká následující prototyp?

```
int froop(double t);
void rattle(int n);
int prune(void);
```

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

Programátorská cvičení

1. Napište program v jazyce C++, který vypíše vaše jméno a adresu.
2. Napište program v jazyce C++, který se vás zeptá na vzdálenost v jednotkách furlong a zadanou hodnotu převede na yardy (jeden furlong je 220 yardů).
3. Napište program v jazyce C++ se třemi uživatelsky definovanými funkcemi (`main()` se počítá za jednu), jehož výstup je následující:

```
Tri slepe mysi
Tri slepe mysi
Divejte se, jak utikaji
Divejte se, jak utikaji
```

   První funkce, která je volána dvakrát, by měla vypsat první dva řádky a druhá funkce, která je také volána dvakrát, by měla obstarat zbývající výstup.
4. Napište program, v jehož funkci `main()` je volána uživatelsky definovaná funkce, která jako argument přijímá teplotu ve stupních Celsia a vrací stejnou hodnotu ve stupních Fahrenheita. Program by měl požádat uživatele o vstup hodnoty ve stupních Celsia a zobrazit výsledek, viz následující kód:

```
Zadejte prosim hodnotu ve stupnich Celsia: 20
20 stupnu Celsia je 68 stupnu Fahrenheita.
```

   Zde je vzorec pro převod:

   Fahrenheit = 1.8 x stupně Celsia + 32.0
5. Napište program, v jehož funkci `main()` je volána uživatelsky definovaná funkce, která jako argument přijímá vzdálenost ve světelných letech a vrací vzdálenost v astronomických jednotkách. Program by měl požádat uživatele o vstup hodnoty ve světelných letech a zobrazit výsledek, jak je ukázáno v následujícím kódu:

```
Zadejte vzdalenost ve svetelnych letech: 4.2
4.2 svetelnych let je 265608 astronomickych jednotek.
```

   Astronomická jednotka odpovídá průměrné vzdálenosti Země od Slunce (přibližně 150 000 000 km nebo 93 000 000 mil) a světelný rok představuje vzdálenost,

kterou světlo urazí za jeden rok (přibližně 10 bilionů km nebo 6 bilionů mil). (Nejbližší hvězda po Slunci je vzdálená 4,2 světelných let.) Použijte typ `double` (jako ve výpisu 2.4) a následující převodní vzorec:

1 světelný rok = 63 240 astronomických jednotek

6. Napište program, který si vyžádá hodnotu hodiny a hodnotu minuty. Funkce `main()` pak tyto hodnoty předá funkci typu `void`, která je vypíše v následujícím tvaru:

```
Enter the number of hours: 9
Enter the number of minutes: 28
Time: 9:28
```

KAPITOLA

3

Práce s daty

Podstata objektově orientovaného programování (OOP) spočívá v navrhování a rozšiřování vašich vlastních datových typů. Vlastní datové typy by měly v co největší míře odpovídat datům. Pokud tuto práci odvedete správně, zjistíte, že se s daty později pracuje mnohem lépe. Dříve než budete umět vytvářet své vlastní typy, musíte znát a rozumět vestavěným typům jazyka C++, protože tyto typy budou vašimi stavebními bloky.

Vestavěné typy jazyka C++ můžeme rozdělit do dvou skupin: základní typy a složené typy. V této kapitole se seznámíte se základními typy, které reprezentují celá čísla a čísla s pohyblivou desetinnou čárkou. Může se zdát, že se jedná pouze o dva typy; avšak jazyk C++ uznává, že žádný celočíselný typ, ani žádný typ s pohyblivou desetinnou čárkou, neodpovídá všem programovým požadavkům, proto nabízí několik variant těchto dvou datových témat. Další kapitola 4, „Složené typy“, popisuje několik typů postavených na základních typech; tyto složené typy obsahují pole, řetězce, ukazatele a struktury.

Programy také nepochybně potřebují prostředky pro identifikaci uložených dat. Prozkoumáme jednu metodu, která to provádí – pomocí proměnných. Dále se podíváme, jak se v jazyce C++ provádějí aritmetické operace. Nakonec si ukážeme, jak C++ převádí hodnoty z jednoho typu na druhý.

V této kapitole se naučíte:

- Pravidla pro pojmenování proměnných v jazyce C++
- Celočíselné typy vestavěné do jazyka C++: unsigned long, long, unsigned int, int, unsigned short, short, char, unsigned char, signed char a bool
- Soubor climits, který reprezentuje systémové omezení různých celočíselných typů
- Číselné konstanty různých celočíselných typů
- Používat kvalifikátor const pro vytváření symbolických konstant
- Typy s pohyblivou desetinnou čárkou vestavěné do jazyka C++: float, double a long double
- Soubor cfloat, který reprezentuje systémové omezení různých typů s pohyblivou desetinnou čárkou
- Číselné konstanty různých typů s pohyblivou desetinnou čárkou
- Aritmetické operátory jazyka C++
- Automatické typové konverze
- Vynucené typové konverze (přetypování)

Jednoduché proměnné

Program většinou potřebuje ukládat informace – může to být například aktuální cena akcií společnosti IBM, průměrná vlhkost v New Yorku v srpnu, nejčastěji se vyskytující písmeno v textu Ústavy a jeho relativní četnost nebo počet dostupných představitelů Elvise. Pokud chce program uložit informaci do počítače, musí si udržovat přehled o jejích třech základních vlastnostech:

- Kde je informace uložena
- Jakou má hodnotu
- Jaký druh informace je uložen

Doposud se příklady v této knize řídily strategií deklarování proměnné. Typ použitý v deklaraci popisuje druh informace a jméno proměnné reprezentuje symbolickou hodnotu. Předpokládejme, že například zástupce vedoucího laboratoře Igor zadá následující příkazy:

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

Tyto příkazy říkají programu, že je ukládáno celé číslo a jméno braincount v našem případě představuje celočíselnou hodnotu 5. Program v podstatě vymezí dostatečný paměťový prostor pro celé číslo, poznamená si jeho umístění a na určené místo zkopíruje hodnotu 5. Výše uvedené příkazy vám (nebo Igorovi) neříkají, kde je v paměti hodnota uložena, ale program o této informaci ví. Pomocí operátoru & samozřejmě můžete zjistit adresu proměnné braincount v paměti. Tento operátor probereme v následující kapitole, až budeme zkoumat druhou strategii identifikace dat – použití ukazatelů.

Jména proměnných

Jazyk C++ podporuje používání smysluplných jmen proměnných. Jestliže proměnná představuje náklady na výlet, nazvěte ji `naklady_na_vylet` nebo `nakladyNaVylet` (dáváte-li přednost angličtině pak `cost_of_trip` nebo `costOfTrip`) a ne pouze `x` nebo `nav`. Při pojmenovávání byste měli dodržovat několik jednoduchých pravidel jazyka C++:

- Ve jménu můžete použít pouze alfabetické znaky, číslice a podtržítko (_).
- První znak ve jménu nemůže být číslice.
- Rozlišujte velká a malá písmena.
- Pro jména nemůžete používat klíčová slova jazyka C++.
- Jména začínající dvěma nebo jedním podtržítkem následovaná velkým písmenem jsou vyhrazena pro implementaci – tj. pro překladač a prostředky, které využívá. Jména začínající jedním podtržítkem jsou vyhrazena pro implementaci, která je používá jako globální identifikátory.
- Jazyk C++ nemá žádná omezení délky jmen a všechny znaky ve jménu jsou významové.

Předposlední pravidlo se poněkud liší od předcházejících, protože použití názvu, jako je `__time_stop` nebo `_Donut`, nevyvolá chybu překladače, ale způsobí nedefinované chování. Jinými slovy, není možné předem říci, jaký bude výsledek. Důvodem, proč není generována chyba překladače, je skutečnost, že tato jména nejsou neplatná, ale vyhrazená pro implementaci. V kapitole 4 se tomuto tématu budeme znovu věnovat.

Poslední pravidlo odlišuje jazyk C++ od ANSI C (C99), který zaručuje, že významové jsou pouze první 63 znaky. (V ANSI C jsou dvě jména, která mají prvních 63 znaků stejných, považována za stejná, i když se liší v 64. znaku.)

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

```
int pudl;            // platné
int Pudl;            // platné a odlišné od pudl
```

```
int PUDL;             // platné a ještě více odlišné
Int terier;           // neplatné -- má být int, nikoli Int
int moje_hvezdy3;     // platné
int _Mojehvezdy3;     // platné, ale vyhrazené -- začíná podtržítkem
int 1tel;             // neplatné, protože začíná číslicí
int double;           // neplatné -- double je klíčovým slovem C++
int begin;            // platné -- begin je klíčovým slovem Pascalu
int __blazni;         // platné, ale vyhrazené -- začíná dvěma podtržítky
int ta_nejlepsi_promenna_jaka_muze_byt_verze_112; // platné
int honky-tonk;       // neplatné -- pomlčka není povolena
```

Chcete-li vytvářet jméno ze dvou nebo více slov, obvyklá praxe spočívá v oddělení slov podtržítkem jako ve jménu `moje_cibule` nebo napsáním velkého počátečního písmene v každém slově počínaje druhým, například `mojeOcniKapky`. (Veteráni jazyka C mají sklon používat metodu s podtržítkem podle zvyklostí z C, příznivci Pascalu dávají přednost velkým písmenům.) Oba tvary usnadňují čitelnost jednotlivých slov a umožňují jejich rozlišení, například `maLahodnost` a `malaHodnost` nebo `motor_kapsa` a `motorka_psa`.

Z praxe: Jména proměnných

Metody pojmenovávání proměnných a funkcí jsou živnou půdou pro žhavé diskuse. Toto téma bývá zdrojem ostrých rozporů ve světě programování. Obdobně, jako jsme si říkali u funkcí, platí i u proměnných, že překladač se nestará o jména vašich proměnných, pokud dodržují stanovená omezení, a že dodržování pevně stanovených a přesných osobních konvencí pojmenovávání se vždy vyplatí.

Psaní velkých písmen je stejně důležité u jmen proměnných jako u funkcí (viz poznámka v kapitole 2 „Konvence přidělování jmen"), ale mnoho programátorů vkládá do jména proměnných ještě další informaci, tou je předpona, která popisuje typ nebo obsah proměnné. Například celočíselná proměnná `mojeVaha` by mohla být nazvána `nMojeVaha`; zde je předpona `n`, která reprezentuje celé číslo, užitečná, když čtete kód a nemáte okamžitě k dispozici definici této proměnné. Další možností je nazvat tuto proměnnou jménem `intMojeVaha`, které je výstižnější a čitelnější, ačkoliv to znamená přidat dva znaky navíc (k čemuž má mnoho programátorů vysloveně odpor). V podobném stylu jsou běžně používány i další předpony: `str` a `sz` představují nulou ukončený řetězec znaků (zero-terminated string), `b` pravdivostní hodnotu (`boolean`), `p` ukazatel (`pointer`) a `c` jeden znak (`character`).

Při vašem pronikání do světa jazyka C++ se setkáte s mnoha příklady psaní předpon, včetně pěkné přepony m_lpctstr, která označuje členskou proměnnou třídy (class member m_), jíž je ukazatel (long pointer lp) na konstantní, nulou ukončený řetězec znaků (constant, zero-terminated character string ctstr), stejně jako s ještě podivnějšími a někdy i neodhadnutelnými styly, které můžete, ale nemusíte, přijmout za vlastní. Nehledě na osobní preference i zde se vyplatí vsadit na důslednost a přesnost. Dávejte přednost proměnným, které vyhovují vašim potřebám a osobnímu stylu. (Nebo, pokud je to od vás vyžadováno, potřebám a stylu vašeho zaměstnavatele.)

Celočíselné typy

Celá čísla nemají desetinnou část, jsou to například 2, 98, -5286 a 0. Za předpokladu, že je celých čísel nekonečně mnoho, nemůže je žádné konečné množství paměti počítače

obsáhnout všechna. Proto jazyk pracuje pouze s podmnožinou všech celých čísel. Některé jazyky, jako například standardní Pascal, nabízejí pouze jeden typ celého čísla (jeden typ stačí na všechno!), naproti tomu C++ poskytuje několik možností. Můžete si vybrat celočíselný typ, který nejlépe vyhovuje určitým požadavkům programu. Tato snaha o přizpůsobení typu datům je předzvěstí navrhování datových typů v OOP.

Různé celočíselné typy jazyka C++ se liší v množství paměti potřebné pro uložení celého čísla. Větší blok paměti může obsáhnout širší rozsah celočíselných hodnot. Některé typy (znaménkové) mohou ukládat kladné i záporné hodnoty, zatímco jiné (neznaménkové) nemohou představovat záporné hodnoty. Obvyklým termínem popisujícím množství paměti použité pro uložení celého čísla je *délka*. Čím více paměti hodnota využívá, tím je delší. Základní celočíselné typy jazyka C++ seřazené podle délky se nazývají `char`, `short`, `int` a `long`. Všechny mají znaménkovou i neznaménkovou verzi. To znamená, že máte k dispozici osm různých celočíselných typů! Podívejme se na tyto typy podrobněji. Protože typ `char` má jisté zvláštní vlastnosti (nejčastěji se používá pro reprezentaci znaků namísto čísel), bude se tato kapitola nejprve věnovat ostatním typům.

Celočíselné typy short, int a long

Paměť počítače se skládá z jednotek, které se nazývají *bity*. (Podívejte se na poznámku „Bity a bajty" v této kapitole.) Typy jazyka C++ `short`, `int` a `long` mohou použitím odlišného počtu bitů pro uložení hodnoty představovat až tři různé délky celých čísel. Bylo by výhodné, kdyby měl každý typ na všech systémech pevně danou délku, například pro `short` by bylo 16 bitů, pro `int` 32 bitů a tak dále. Ale život není tak jednoduchý. Důvodem je, že žádná volba by nevyhovovala všem typům počítačů. Jazyk C++ nabízí přizpůsobivý standard s jistými zaručenými minimálními velikostmi, které jsou převzaty z jazyka C. Zde jsou jeho pravidla:

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

Bity a bajty

Základní jednotkou počítačové paměti je *bit*. Můžete si ho představit jako elektronický přepínač, který je možné nastavit na vypnuto nebo zapnuto. Vypnuto představuje hodnotu 0 a zapnuto hodnotu 1. 8bitová část paměti může být nastavena 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 × 2 × 2 × 2 × 2 × 2 × 2 × 2, neboli 256. Proto 8bitová jednotka může reprezentovat například hodnoty od 0 do 255 nebo hodnoty od -128 do 127. Každý další bit zdvojnásobuje počet kombinací. To znamená, že 16bitovou jednotku můžete nastavit na 65 536 různých hodnot a 32bitovou jednotku na 4 294 672 296 různých hodnot.

***Bajt* obvykle představuje 8bitovou jednotku paměti. Je v tomto smyslu jednotkou míry, která popisuje objem paměti v počítači, přičemž kilobajt má 1 024 bajtů a megabajt 1 024 kilobajtů. Jazyk C++ však definuje bajt odlišně. *Bajt* se v C++ skládá alespoň z tolika přilehlých bitů, které odpovídají základní znakové sadě implementace. To znamená, že počet možných hodnot musí být roven nebo převyšovat počet jednotlivých**

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 používat 16bitový bajt nebo dokonce 32bitový bajt.

Mnoho systémů v současné době používá minimální zaručené hodnoty, 16bitový `short` a 32bitový `long`. To však otevírá několik možností pro typ `int`. Ten by mohl mít 16, 24 nebo 32 bitů a stále by vyhovoval standardu. Obvykle má `int` na starších implementacích IBM PC 16 bitů (stejně jako short) a 32 bitů (stejně jako `long`) na systémech Windows 98, Windows NT, Macintosh OS X, VAX a řadě dalších implementací minipočítačů. U některých implementací si můžete vybrat, jak budete s typem `int` zacházet. (Co používá vaše implementace? Následující příklad vám ukáže, jak zjistit omezení ve vašem systému, aniž byste museli otevřít manuál.) Různé velikosti typů v implementacích mohou způsobit problémy při přenášení programu napsaného v jazyce C++ z jednoho prostředí na jiné. Avšak trocha pozornosti, jak se dozvíte později v této kapitole, může takovéto problémy minimalizovat.

Jména výše uvedených typů můžete použít pro deklarování proměnných stejně jako `int`:

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

Ve skutečnosti je `short` zkratkou pro `short int` a `long` pro `long int`, ale kdo by používal delší tvary?

Tyto tři typy `int`, `short` a `long` jsou se znaménkem, což znamená, že rozsah je rozdělen téměř shodně mezi kladné a záporné hodnoty. Například 16bitové int může nabývat hodnot od -32 768 do 32 767.

Pokud chcete znát rozsahy celých čísel na vašem systému, jazyk C++ nabízí prostředky, které umožňují zjišťovat velikosti typů v programu. Zaprvé, operátor `sizeof` vrací velikost typu nebo proměnné v bajtech. (**Operátor** je vestavěný prvek jazyka, který pracuje nad jednou nebo více položkami a vytváří hodnotu. Například operátor sčítání, reprezentovaný znakem +, 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. Zadruhé, hlavičkový soubor `climits` (u starších implementací `limits.h`) obsahuje informace o mezích celočíselných typů. Především definuje symbolická jména, která reprezentují různé meze. Například definuje `INT_MAX`, což je největší možná hodnota typu `int` a `CHAR_BIT` jako počet bitů v jednom bajtu. Výpis 3.1 předvádí použití těchto prostředků. Program také ukazuje **inicializaci**, což je využití deklaračního příkazu pro přidělení hodnoty proměnné.

Výpis 3.1 limits.cpp

```
// limits.cpp -- některé hodnoty celočíselných mezí
#include <iostream>
#include <climits>          // pro starší systémy použijte limits.h
int main()
{
   using namespace std;
   int n_int = INT_MAX;     // inicializace n_int na maximální hodnotu
```

```
                                 // proměnné typu 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 ma " << sizeof (int) << " bajty." << endl;
    cout << "short ma " << sizeof n_short << " bajty." << endl;
    cout << "long ma " << sizeof n_long << " bajty." << endl << endl;

    cout << "Maximalni hodnoty:" << endl;
    cout << "int: " << n_int << endl;
    cout << "short: " << n_short << endl;
    cout << "long: " << n_long << endl << endl;

    cout << "Minimalni hodnota typu int = " << INT_MIN << endl;
    cout << "Bitu na bajt = " << CHAR_BIT << endl;
    return 0;
}
```

KOMPATIBILITA

Hlavičkový soubor `climits` je verzí jazyka C++ hlavičkového souboru `limits.h` normy ANSI C. Některé dřívější platformy C++ neměly žádné hlavičkové soubory. Pokud takové systémy používáte, musíte si projít tento příklad pouze v duchu.

Zde je výstup z programu uvedného na výpisu 3.1 a přeloženého systémem Microsoft Visual C++ 7.1:

```
int ma 4 bajty.
short ma 2 bajty.
long ma 4 bajty.

Maximalni hodnoty:
int: 2147483647
short: 32767
long: 2147483647

Minimalni hodnota typu int = -2147483648
Bitu na bajt = 8
```

Zde je výstup z jiného systému (Borland C++ 3.1 pro DOS):

```
int ma 2 bajty.
short ma 2 bajty.
long ma 4 bajty.

Maximalni hodnoty:
int: 32767
short: 32767
long: 2147483647

Minimalni hodnota typu int = -32768
Bitu na bajt = 8
```

Poznámky k programu

V následující části se podíváme na hlavní prvky tohoto programu.

Operátor sizeof a hlavičkový soubor climits

Operátor `sizeof` říká, že typ `int` má na základním systému s 8bitovým bajtem 4 bajty. Operátor `sizeof` můžete použít na jméno typu nebo proměnné. Jméno typu, jako je `int`, musí být uzavřeno v závorkách, ale u jména proměnné, jako je `n_short`, jsou závorky volitelné:

```
cout << "int ma " << sizeof (int) << " bajty.\n";
cout << "short ma " << sizeof n_short << " bajty.\n";
```

Hlavičkový soubor `climits` definuje symbolické konstanty (viz poznámka Symbolické konstanty) odpovídající mezním hodnotám typů. Jak již bylo zmíněno, konstanta `INT_MAX` představuje největší hodnotu, která může být uložena v typu int; jak můžete vidět na výstupu, pro systém DOS by to mělo být číslo 32 767. Výrobce překladače dodává soubor climits, který obsahuje hodnoty odpovídající danému překladači. Například soubor climits pro systém používající 32bitový typ int by měl definovat konstantu `INT_MAX` jako hodnotu 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 souboru `climits`

Symbolická konstanta	Představuje
CHAR_BIT	Počet bitů typu `char`
CHAR_MAX	Maximální hodnota typu `char`
CHAR_MIN	Minimální hodnota typu `char`
SCHAR_MAX	Maximální hodnota typu `signed char`
SCHAR_MIN	Minimální hodnota typu `signed char`
UCHAR_MAX	Maximální hodnota typu `unsigned char`
SHRT_MAX	Maximální hodnota typu `short`
SHRT_MIN	Minimální hodnota typu `short`
USHRT_MAX	Maximální hodnota typu `unsigned short`
INT_MAX	Maximální hodnota typu `int`
INT_MIN	Minimální hodnota typu `int`
UINT_MAX	Maximální hodnota typu `unsigned int`
LONG_MAX	Maximální hodnota typu `long`
LONG_MIN	Minimální hodnota typu `long`
ULONG_MAX	Maximální hodnota typu `unsigned long`

Inicializace

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`. Při inicializaci hodnot je také možné použít obvyklé konstanty. Proměnnou lze inicializovat i jinou dříve definovanou proměnnou. Proměnná může být inicializována dokonce i výrazem, pokud jsou všechny hodnoty výrazu v době překladu známé:

```
int strycove = 5;                  // inicializace strycove na 5
int tety = strycove;               // inicializace tety na 5
int kresla = strycove + tety + 4;  // inicializace kresla na hodnotu 14
```

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

Výše uvedená syntaxe inicializace má původ v jazyce C; C++ nabízí i další syntaxi inicializace, která není součástí jazyka C:

```
int sovy = 101;       // klasická inicializace jazyka C
int strizlici(432);   // jiná syntaxe jazyka C++, nastavuje strizlici na 432
```

ZAPAMATUJTE SI

Pokud neinicializujete proměnnou definovanou uvnitř funkce, její hodnota je **nedefinovaná**. To znamená, že touto hodnotou je to, co se nacházelo v přiřazené paměti před vytvořením proměnné.

Pokud víte, jaká by měla být počáteční hodnota proměnné, inicializujte ji. Oddělení deklarace proměnné od přiřazení hodnoty může způsobit dočasnou nejistotu:

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

Ale inicializace proměnné spojená s deklarací vás chrání před pozdějším opomenutím přiřadit hodnotu.

Symbolické konstanty pomocí preprocesoru

Soubor `climits` obsahuje řádky podobné následujícímu:

```
#define INT_MAX 32767
```

Vzpomeňte si, že proces překladu začíná zpracováním zdrojového kódu preprocesorem. Výrazy `#define` stejně jako `#include` představují direktivy preprocesoru. Výše uvedená direktiva říká preprocesoru: Projdi program a každý výskyt instance `INT_MAX` nahraď hodnotou `32 767`. Takže direktiva `#define` pracuje podobně jako příkaz globálního vyhledávání a nahrazování v editoru nebo v textovém editoru. Po provedení těchto změn je upravený program přeložen. Preprocesor hledá samostatné symboly (oddělená slova) a přeskakuje zabudovaná slova. To znamená, že preprocesor nezaměňuje `PINT_MAXIM` za `P32767IM`. Pomocí direktivy `#define` můžete také definovat své vlastní symbolické konstanty. (Viz výpis 3.2.) Direktiva `#define` je však přežitek z C. C++ má lepší způsob vytváření symbolických konstant (klíčové slovo `const`, o kterém si povíme později), takže direktivu `#define` nebudete používat moc často. Ale některé hlavičkové soubory, zvláště ty, které byly navrženy pro jazyk C i C++, ji používají.

Neznaménkové typy

Všechny tři celočíselné typy se vyskytují i ve variantě bez znaménka, která nemůže obsahovat záporné hodnoty. To má výhodu ve zvýšení největší hodnoty, kterou může proměnná obsahovat. Pokud má například typ short rozsah -32 768 až +32 767, u neznaménkové verze je to 0 až 65 535. Neznaménkové typy můžete samozřejmě používat pouze pro hodnoty, které nejsou nikdy záporné, jako je například populace, stav zásob a počet šťastných tváří. Ze základních celočíselných typů vytvoříte neznaménkové verze jednoduchým přidáním klíčového slova `unsigned` do deklarace:

```
unsigned short zmena;    // typ unsigned short
unsigned int rovert;     // typ unsigned int
unsigned zadak;          // také typ unsigned int
unsigned long pryc;      // typ unsigned long
```

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

Výpis 3.2 předvádí použití neznaménkových typů. Dále ukazuje, co by se mohlo stát, kdyby program překročil hranice celočíselných typů. Zde se také naposledy setkáte s direktivou preprocesoru `#define`.

Výpis 3.2 exceed.cpp

```
// exceed.cpp -- překročení některých celočíselných omezení
#include <iostream>
#define ZERO 0          // symbol ZERO bude mít hodnotu 0
#include <climits>      // definuje INT_MAX jako největší celočíselnou hodnotu
int main()
{
   using namespace std;
   short sam = SHRT_MAX;        // inicializuje proměnnou na maximální hodnotu
   unsigned short sue = sam;    // v pořádku, je-li proměnná sam již
                                // definována

   cout << "Sam ma ulozeno " << sam << " dolaru a Sue " << sue;
   cout << " dolaru." << endl
        << "Na kazdy ucet pridame 1 dolar." << endl << "Nyni ";
   sam = sam + 1;
   sue = sue + 1;
   cout << "ma Sam ulozeno " << sam << " dolaru a Sue " << sue;
   cout << " dolaru.\nChudak Sam!" << endl;
   sam = ZERO;
   sue = ZERO;
   cout << "Sam ma ulozeno " << sam << " dolaru a Sue " << sue;
   cout << " dolaru." << endl;
   cout << "Z kazdeho uctu vezmeme 1 dolar." << endl << "Nyni ";
   sam = sam - 1;
   sue = sue - 1;
   cout << "ma Sam ulozeno " << sam << " dolaru a Sue " << sue;
   cout << " dolaru." << endl << "Stastna Sue!" << endl;
   return 0;
}
```

KOMPATIBILITA

Výpis 3.2, podobně jako 3.1, používá soubor `climits`; starší překladače budou možná požadovat soubor `limits.h` a některé velmi staré nemusí mít k dispozici ani jeden z uvedených souborů.

Zde je výstup z programu uvedeného na výpisu 3.2:

```
Sam ma ulozeno 32767 dolaru a Sue 32767 dolaru.
Na kazdy ucet pridame 1 dolar.
Nyni ma Sam ulozeno -32768 dolaru a Sue 32768 dolaru.
Chudak Sam!
Sam ma ulozeno 0 dolaru a Sue 0 dolaru.
Z kazdeho uctu vezmeme 1 dolar.
Nyni ma Sam ulozeno -1 dolaru a Sue 65535 dolaru.
Stastna Sue!
```

Program přiřadí proměnné `sam` typu `short` a proměnné `sue` typu `unsigned short` největší hodnotu typu `short`, která je na našem systému 32 767. Potom ke každé hodnotě přičte 1. To nezpůsobuje žádné problémy pro `sue`, protože nová hodnota je stále mnohem menší, než maximální hodnota celého čísla bez znaménka. Ale `sam` přejde z 32 767 na -32 768! Podobně odečtení 1 od nuly nepředstavuje žádný problém pro `sam`, 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 znovu na druhé straně rozsahu. (Viz obrázek 3.1.) Jazyk C++ zaručuje, že se neznaménkové typy chovají tímto způsobem. Avšak překročení mezí (přetečení a podtečení) u znaménkových typů není jazykem C++ zaručeno, nicméně popsané chování je v současných implementacích nejběžnější.

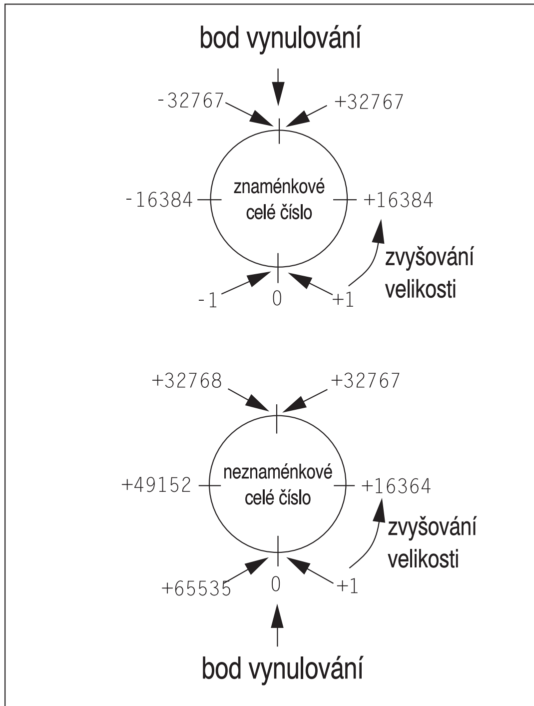
Typ long už nestačí

Ke standardu C99 jazyka C přibylo pár nových typů, které budou s největší pravděpodobností součástí dalšího vydání standardu jazyka C++. Ve skutečnosti je již mnoho překladačů jazyka C++ podporuje. Těmito typy jsou `long long` a `unsigned long long`. U obou je zaručeno, že mají nejméně 64 bitů a jsou minimálně tak velké jako typy `long` a `unsigned long`.

Jaký zvolit celočíselný typ?

Jaký typ byste měli při této bohatosti celočíselných typů jazyka C++ použít? Obecně platí, že typ int je nastavován tak, aby byl pro cílový počítač „nejpřirozenější" velikostí celého čísla. **Přirozená velikost** představuje tvar celého čísla, se kterým počítač pracuje nejefektivněji. Neexistuje-li žádný pádný důvod pro výběr jiného typu, použijte `int`.

Nyní se podíváme na důvody opravňující k použití jiných typů. Jestliže proměnná představuje hodnotu, která nemůže být nikdy záporná, jako je například počet slov v dokumentu, můžete použít typ bez znaménka; proměnná pak může reprezentovat větší hodnoty.



Obrázek 3.1 Typické chování celých čísel při přetečení

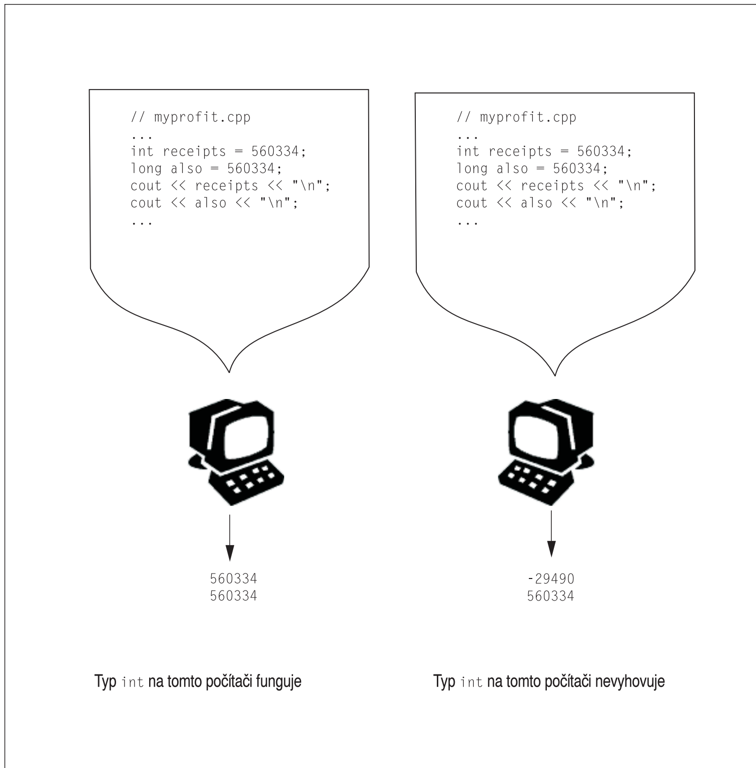
Pokud víte, že proměnná bude muset reprezentovat celočíselné hodnoty, které jsou příliš velké na 16bitové celé číslo, použijte `long`. To platí dokonce i tehdy, když má `int` na vašem systému 32 bitů. Tímto způsobem zaručíte správnou funkčnost programu i po přenesení na systém s 16bitovým `int`. (Viz obrázek 3.2.)

Jestliže je short menší než int, šetří použití typu short 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 uložených v paměti za sebou.) Pokud je ušetřené místo důležité, měli byste použít short namísto int, i když jsou oba typy stejně velké. Předpokládejme například, že přesunete váš program ze systému DOS PC s 16bitovým typem `int` na systém Windows NT, kde je int 32bitový. To zdvojnásobí množství paměti potřebné pro uložení pole typů int, ale neovlivňuje požadavky na pole typů short. Pamatujte, že každý ušetřený bit se počítá.

Potřebujete-li pouze jediný bajt, můžete použít char. Tuto možnost si brzy předvedeme.

Celočíselné konstanty

Celočíselné konstanty píšeme explicitně, například 212 nebo 1776. Jazyk C++, podobně jako C, vám dovoluje zapsat celá čísla pomocí tří různých číselných základů: základ 10 (oblíbený veřejností), základ 8 (používaný na starém Unixu) a základ 16 (oblíbený hardwarovými nadšenci). Příloha A, „Číselné soustavy“, tyto soustavy popisuje; zde se podíváme na jejich zastoupení v jazyce C++. Pro identifikaci základu číselné konstanty pou-



Obrázek 3.2 Z důvodů přenositelnosti použijte pro velká celá čísla typ `long`

žívá jazyk C++ jednu nebo dvě první číslice. Je-li první číslice v rozsahu 1–9, výsledné číslo má základ 10 (desítkový, dekadický); tedy 93 má základ 10. Jestliže je první číslice 0 a druhá v rozsahu 1–7, číslo má základ 8 (osmičkový, oktalový); tedy 076 je osmičkové a rovno 62 desítkově. Pokud jsou první dva znaky 0x nebo 0X, číslo má základ 16 (šestnáctkový, hexadecimální); tedy 0x76 je hexadecimální a je rovno 118 desítkově. Pro hexadecimální hodnoty znaky a–f a A–F reprezentují hexadecimální číslice, které odpovídají hodnotám 10–15. 0xF je 15 a 0xA5 je 165 (10 šestnáctek plus 5 jedniček). Výpis 3.3 obsahuje program, v němž jsou použity všechny tři číselné soustavy.

Výpis 3.3 hexoct1.cpp

```
// hexoct1.cpp - příklady hexadecimálních a oktalových konstant
#include <iostream>
int main()
{
using namespace std;
int chest = 76;            // dekadická celočíselná konstanta
int waist = 0x76;          // hexadecimální celočíselná konstanta
int inseam = 076;          // oktalová celočíselná konstanta
cout << "Pan ma pozoruhodnou postavu!\n";
cout << "hrudnik = " << chest << "\n";
cout << "pas = " << waist << "\n";
```

```
cout << "sed = " << inseam << "\n";
return 0;
}
```

Objekt `cout` zobrazuje celá čísla implicitně v desítkovém tvaru nehledě na to, jak jsou v programu napsána, což dokazuje následující výstup:

```
Pan ma pozoruhodnou postavu!
hrudnik = 76 (76 dekadicky)
pas = 118 (0x76 hexadecimálně)
sed = 62 (076 oktalově)
```

Pamatujte si, že tvar těchto zápisů slouží pouze k zjednodušení psaní programů. Zjistíte-li například, že segment video paměti CGA je `B000` hexadecimálně, nemusíte tuto hodnotu převádět do desítkové soustavy na 45 056. Zadejte prostě `0xB000`. Když napíšete hodnotu deset jako `10`, `012` nebo `0xA`, ukládá se v počítači stejným způsobem – jako binární hodnota (se základem 2).

Mimochodem, chcete-li zobrazit hodnotu hexadecimálně nebo osmičkově, můžete použít některé zvláštní vlastnosti objektu `cout`. Připomeňme si, že hlavičkový soubor `iostream` má manipulační symbol `endl`, který dává objektu `cout` pokyn k tomu, aby začal psát na nový řádek. Podobně jsou k dispozici další manipulační symboly `dec`, `hex` a `oct`, jimiž informujeme `cout`, aby vypsal celé číslo dekadicky, hexadecimálně resp. oktalově. Ve výpisu 3.4 slouží manipulační symboly `hex` a `oct` k tomu, aby program vypsal číslo 76 ve třech tvarech (dekadický tvar je implicitní až do nejbližší změny).

Výpis 3.4 hexoct2.cpp

```
// hexoct2.cpp - výpis hexadecimálních a oktalových hodnot
#include <iostream>
using namespace std;
int main()
{
   using namespace std;
   int chest = 76;
   int waist = 76;
   int inseam = 76;

   cout << "Pan ma pozoruhodnou postavu!" << endl;
   cout << "hrudnik = " << chest << " (dekadicky)" << endl;
   cout << hex;      // manipulační symbol pro změnu základu číselné soustavy
   cout << "pas = " << waist << " hexadecimalne" << endl;
   cout << oct;      // manipulační symbol pro změnu základu číselné soustavy
   cout << "sed = " << inseam << " (oktalove)" << endl;
   return 0;
}
```

Zde je výstup z programu na výpisu 3.4:

```
Pan ma pozoruhodnou postavu!
hrudnik = 76 (dekadicky)
pas = 4c hexadecimalne
sed = 114 (oktalove)
```

Všimněme si, že příkaz

```
cout << hex;
```

nevypíše na obrazovku vůbec nic – pouze změní způsob výpisu celých čísel. Manipulační symbol `hex` je tedy skutečně jen zpráva objektu `cout`, která říká, jak se má chovat. Také si všimněme, že vzhledem k tomu, že `hex` je součástí jmenného prostoru `std` a program jej používá, nemůže tímto symbolem pojmenovat proměnnou. Nicméně pokud jste vynechali direktivu `using` a použili místo ní `std::cout`, `std::endl`, `std::hex` a `std::oct`, symbol `hex` můžete použít jako název proměnné.

Jak C++ určuje typ konstanty

Deklarace programu oznamují překladači jazyka C++ typ určité celočíselné proměnné. Ale co konstanty? Předpokládejme, že v programu reprezentujete číslo konstantou:

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

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

Nejprve se podívejme na přípony. Jsou to písmena umístěná na konci číselné konstanty, která určují její typ. Přípona l nebo L u celého čísla říká, že se jedná o konstantu typu `long`, přípona u nebo U signalizuje konstantu typu `unsigned int` a `ul` (v jakékoli kombinaci pořadí malých a velkých písmen) určuje typ konstanty jako `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` typu `unsigned long`.

Dále se podívejme na velikost. Jazyk C++ má nepatrně jiná pravidla pro celá čísla desítková než pro hexadecimální nebo osmičková. (Výraz desítkový zde vyjadřuje základ 10, stejně jako hexadecimální základ 16.) Desítkové číslo bez přípony je reprezentováno nejmenším z následujících typů, který ho může obsahovat: `int`, `long` nebo `unsigned long`. Na počítačovém systému, který používá 16bitový typ `int` a 32bitový `long`, vyjadřuje číslo `2000` typ int, `40000 long` a `3000000000 unsigned long`. Hexadecimální 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 vyjadřuje `40000` jako `long`, reprezentuje hexadecimální ekvivalent `0x9C40` typem `unsigned int`. To proto, že se hexadecimální číslo používá pro vyjádření paměťových adres, které jsou bez znaménka. Takže pro 16bitové adresy je `unsigned int` mnohem vhodnější než `long`.

Typ char: Znaky a malá celá čísla

Nyní se vrátíme k poslednímu celočíselnému typu `char`. Jak pravděpodobně tušíte podle jeho jména, typ `char` je navržen pro ukládání znaků, jako jsou písmena a číslice. Ukládání čísel není pro počítač žádný problém, ale ukládání písmen je jinou záležitostí. Programovací jazyky to řeší jednoduchým přidělováním číselného kódu znakům. Typ `char` tedy představuje pouze jiný typ celého čísla. Musí být dostatečně velký, aby obsáhl celý rozsah 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. Ačkoliv se `char` nej-

častěji používá pro práci se znaky, můžete ho také využít jako celočíselný typ, který je menší než `short`.

Nejběžnější sadou symbolů ve Spojených státech je znaková sada ASCII, jejíž popis naleznete v příloze C, „Znaková sada ASCII“. Jednotlivé znaky této sady reprezentuje číselný kód (kód ASCII). Například 65 je kód pro znak A. Tato kniha ve svých příkladech předpokládá kvůli zjednodušení kód ASCII. Avšak implementace jazyka C++ mohou pracovat s jakýmkoliv nativním kódem hostitelského systému – na sálovém počítači IBM to může být například EBCDIC (vyslovujte eb-se-dik). Jelikož ASCII ani EBCDIC neslouží dobře mezinárodním potřebám, jazyk C++ podporuje také znakový typ `wchar_t`, který má větší rozsah hodnot. Používá jej například mezinárodní znaková sada Unicode a dozvíte se o něm dále v této kapitole.

Typ `char` si můžete vyzkoušet v programu uvedeném na výpisu 3.5.

Výpis 3.5 chartype.cpp

```
// chartype.cpp -- the char type
#include <iostream>
int main( )
{
   using namespace std;
   char ch; // deklarace znakove promenne

   cout << "Zadejte znak: " << endl;
   cin >> ch;
   cout << "Hola! ";
   cout << "Za znak " << ch << " vam dekuji." << endl;
   return 0;
}
```

Zde je výstup z programu uvedeného ve výpisu 3.5:

```
Zadejte znak:
M
Hola! Za znak M vam dekuji.
```

Zajímavé je, že zadáváte `M`, nikoli tedy kód odpovídající znaku `77`. Program také tiskne `M`, ne `77`. Avšak když se podíváte do paměti zjistíte, že v proměnné `ch` je uložena hodnota `77`. Toto kouzlo nespočívá v typu `char`, ale v objektech `cin` a `cout`, které za vás provádějí odpovídající převody. Na vstupu převádí objekt `cin` písmeno `M` zadané z klávesnice na hodnotu `77`. Na výstupu převádí objekt `cout` hodnotu `77` na zobrazený znak `M`; `cin` a `cout` se řídí typem proměnné. Pokud umístíte stejnou hodnotu `77` do proměnné `int`, objekt `cout` ji zobrazí jako `77`. (To znamená, že `cout` zobrazí dva znaky `7`.) Výpis 3.5 tuto vlastnost předvádí. Také ukazuje, že znakovou konstantu můžeme v C++ psát uzavřením znaku mezi dvě jednoduché uvozovky, jak vidíte v případě ‘M’. (Všimněte si, že příklad nepoužívá dvojité uvozovky. C++ používá jednoduché uvozovky pro znaky a dvojité uvozovky pro řetězce. Objekt `cout` umí zpracovat obě, ale jak se dozvíte v kapitole 4, jsou obě zcela různé.) Nakonec program představuje vlastnost objektu `cout`, funkci `cout.put()`, která zobrazuje jediný znak.

Výpis 3.6 morechar.cpp

```
// morechar.cpp – typy char versus int
#include <iostream>
int main()
{
   using namespace std;
   char ch = 'M';           // přiřaď ASCII kód M do ch
   int i = ch;              // přiřaď tentýž kód do int
   cout << "The ASCII code for " << ch << " is " << i << endl;

   cout << "Ke kodu znaku prictu jednicku:" << endl;
   ch = ch + 1;             // změň kód znaku v ch
   i = ch;                  // nový znak ulož do i
   cout << "ASCII kod znaku " << ch << " je " << i << endl;

   // pouziti clenske funkce cout.put() k vypisu znaku
   cout << "Znak ch vypisuji pomoci cout.put(ch): ";
   cout.put(ch);

   // pouziti cout.put() k vypisu znakove konstanty
   cout.put('!');
   cout << endl << "Hotovo" << endl;
   return 0;
}
```

Zde je výstup z programu uvedeného ve výpisu 3.6:

```
ASCII kod znaku M je 77
Ke kodu znaku prictu jednicku:
ASCII kod znaku N je 78
Znak ch vypisuji pomoci cout.put(ch): N!
Hotovo
```

Poznámky k programu

V programu uvedeném na výpisu 3.6 reprezentuje zápis ‘M’ číselný kód znaku `M`. Inicializací proměnné `c` typu `char` na hodnotu ‘M’ je tedy do proměnné `c` vložena hodnota `77`. Program potom přiřadí stejnou hodnotu do proměnné `i` typu `int`, takže proměnné `c` a `i` obsahují hodnotu `77`. Dále objekt `cout` zobrazí obsah proměnné `c` jako `M` a `i` jako `77`. Jak jsme si již řekli dříve, formát výstupu řídí u objektu `cout` typ zobrazované hodnoty – je to pouze další příklad činnosti 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. (Stejně jednoduše je možné přičíst 1 k `i`.) Objekt `cout` znovu 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 zobrazuje obsah proměnné `c` a znakovou konstantu pomocí funkce `cout.put()`.

Členská funkce: cout.put()

Co je vlastně `cout.put()` a proč má ve svém jméně tečku? Funkce `cout.put()` je prvním příkladem důležitého pojmu OOP v jazyce C++, kterému říkáme **členská funkce**. Vzpomeňte si, že třída definuje reprezentaci dat a manipulaci s nimi. Členská funkce patří třídě a popisuje metodu pro manipulaci s daty třídy. Třída `ostream` má například členskou funkci `put()`, která je určena pro výstup znaků. Členskou funkci můžete použít pouze s určitým objektem třídy, což je v tomto případě `cout`. Abyste mohli zavolat členskou funkci třídy nad objektem, jako je například `cout`, použijete ke spojení jména objektu (`cout`) se jménem funkce (`put()`) tečku. Tečka představuje **členský operátor**. Zápis `cout.put()` znamená použití členské funkce `put()` nad objektem `cout` třídy `ostream`. V kapitole 10 „Objekty a třídy" se o tomto samozřejmě dozvíte více podrobností. Prozatím jsou vašimi jedinými třídami `istream` a `ostream`, s jejichž členskými funkcemi můžete různě experimentovat, abyste si nové pojmy více osvojili.

Členská funkce `cout.put()` poskytuje jinou možnost zobrazení znaku jako alternativu k použití operátoru <<. Možná by vás zajímalo, proč je vůbec funkce `cout.put()` potřebná. Je to převážně z historických důvodů. Před vydáním verze 2.0 jazyka C++ objekt `cout` zobrazoval znakové proměnné jako znaky, ale znakové konstanty, například 'M' a '\n', jako čísla. Problém spočíval v tom, že dřívější verze C++, podobně jako C, ukládaly znakové konstanty jako typ `int`. To znamená, že kód `77` pro 'M' byl uložen do 16bitové nebo 32bitové jednotky. Zatímco proměnné typu `char` obsazovaly 8 bitů. A příkaz jako

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

kopíroval 8 bitů (důležitých 8 bitů) z konstanty 'M' do proměnné `c`. To bohužel znamenalo, že se konstanta 'M' a proměnná `c` jevily objektu `cout` zcela odlišně, dokonce i když obsahovaly stejnou hodnotu. Takže příkaz jako

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

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

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

vytiskl požadovaný znak. Nyní, po vydání verze 2.0, C++ ukládá jednotlivé znakové konstanty jako typ `char`, ne jako typ `int`. To znamená, že `cout` nyní zachází se znakovými konstantami správně.

Objekt `cin` ovládá několik různých způsobů čtení znaků ze vstupu. Můžete je snadno prozkoumat pomocí programu, který používá cykly pro čtení několika znaků, k tomuto tématu se vrátíme, až budeme probírat cykly v kapitole 5, „Cykly a relační výrazy".

Konstanty typu char

Znakové konstanty můžete v jazyce C++ psát několika způsoby. Nejjednodušší je uzavřít běžné znaky, jako jsou písmena, interpunkční znaménka a číslice, do jednoduchých uvozovek. Tento zápis představuje numerický kód znaku. Systém ASCII rozumí následujícím příkladům:

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

Tento zápis je lepší než explicitní použití numerických kódů. Vypadá srozumitelněji a nepředpokládá určitý kód. Pokud systém používá EBCDIC, potom 65 není kód pro `A`, avšak `'A'` tento znak představuje stále.

Některé znaky nemůžete do programu zapisovat přímo z klávesnice. Nelze například vložit znak nového řádku do řetězce stiskem klávesy Enter; editovací program si místo toho stisk této klávesy vysvětlí jako požadavek přechodu na začátek nového řádku ve vašem zdrojovém souboru. S jinými znaky jsou problémy, protože mají v jazyce C++ zvláštní význam. Například znak dvojité uvozovky ohraničuje řetězce, takže ho nemůžete jen tak vložit doprostřed řetězce. C++ má pro některé z těchto znaků zvláštní označení, která se nazývají *escape sekvence*, jak je uvedeno v tabulce 3.2. Například `\a` představuje výstrahu, která vyšle na reproduktor terminálu zvukový signál nebo zazvoní. A \" symbolizuje běžný znak dvojité uvozovky 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\nwas here!\n";
```

Poslední řádek vypíše:

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

Tabulka 3.2 Kódy escape sekvencí v C++

Jméno proměnné	Symbol ASCII	Kód C++	Desítkový kód ASCII	Hexadecimální kód ASCII
nový řádek	NL (LF)	\n	10	0xA
horizontální tabulátor	HT	\t	9	0x9
vertikální tabulátor	VT	\v	11	0xB
posun doleva	BS	\b	8	0x8
návrat vozíku	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

Všimněte si, že escape sekvenci, např. `\n`, považujete za normální znak, třeba jako `Q`. To znamená, že ho uzavíráte do jednoduchých uvozovek při vytváření znakové konstanty, ale tyto uvozovky nepoužíváte, pokud ho vkládáte do řetězce.

Znakem nový řádek se vkládají nové řádky do výstupu a je alternativou k `endl`. Lze jej zapsat jako znakovou konstantu (`'\n'`) nebo jako znak v řetězci (`"\n"`). Všechny tři následující příkazy posunou kurzor na začátek následujícího řádku.

```
cout << endl; // manipulační symbol endl
cout << '\n'; // znaková konstanta
cout << "\n"; // řetězec
```

Nový řádek může být součástí delšího řetězce; často je tento způsob vhodnější než `endl`. Například výsledný výstup ve dvou následujících příkazech `cout` je shodný:

```
cout << endl << endl << "Co dal?" << endl << "Zadejte cislo:" << endl;
cout << "\n\nCo dal?\nZadejte cislo:\n";
```

Při zobrazování čísla je o něco jednodušší napsat `endl` než `"\n"` nebo `'\n'`, zatímco při zobrazování řetězce je tomu naopak.

```
cout << x << endl;  // jednodussi nez cout << x << "\n";
cout << "Dr. X.\n"; // jednodussi nez cout << "Dr. X." << endl;
```

TIP

Máte-li možnost volby mezi použitím numerické nebo symbolické escape sekvence, například `\0x8` nebo `\b`, použijte symbolický kód. Numerická reprezentace se váže na určitý kód, například ASCII, zatímco symbolická reprezentace pracuje se všemi kódy a je čitelnější. Na výpisu 3.7 naleznete několik escape sekvencí. Používá znak zvukové výstrahy na upoutání pozornosti, znak nového řádku na posunutí kurzoru a znak posun doleva pro posunutí kurzoru o jedno místo zpět.

Výpis 3.7 bondini.cpp

```
// bondini.cpp -- pouziti escape sekvenci
#include <iostream>
int main()
{
    using namespace std;
    cout << "\aOperace \"HyperHype\" je nyni aktivovana!\n";
    cout << "Zadejte svuj tajny kod:________\b\b\b\b\b\b\b\b";
    long code;
    cin >> code;
    cout << "\aZadal jste " << code << "...\n";
    cout << "\aKod overen! Spustte plan Z3!\n";
    return 0;
}
```

KOMPATIBILITA

Některé systémy C++ založené na překladačích jazyka C z doby před normou ANSI C neumí rozpoznat znak `\a`. Na systémech, které používají znakové kódy ASCII, můžete znak `\a` nahradit znakem `\007`. Některé systémy se mohou chovat odlišně, místo posunu doleva zobrazují znak `\b` jako malý obdélník nebo při posunu doleva mažou znaky, v některých případech možná kromě `\a`.

Program z výpisu 3.7 vypíše po spuštění následující text:

```
Operace "HyperHype" je nyni aktivovana!
Zadejte svuj tajny kod:________
```

Po vytisknutí znaků podtržení program použije znak posunu doleva pro nastavení kurzoru na první znak podtržení. Potom můžete zadat svůj tajný kód a pokračovat. Zde je úplný výpis programu:

```
Operace "HyperHype" je nyni aktivovana!
Zadejte svuj tajny kod:42007007
```

```
Zadal jste 42007007...
Kod overen! Spustte plan Z3!
```

Univerzální jména znaků

Implementace jazyka C++ podporují základní znakovou sadu zdroje, to znamená sadu znaků, které můžete použít pro napsání zdrojového kódu. Skládá se z písmen (velkých a malých) a číslic standardní americké klávesnice, symbolů, jako jsou { a =, používaných jazykem C, a z několika dalších znaků, jako jsou nový řádek a mezera. Dále rozlišujeme základní znakovou sadu výstupu (možné znaky výstupu z programu), ta je doplněna několika dalšími znaky, jako jsou posun doleva a zvuková výstraha. Standardní dodatky umožňují aplikacím nabízet rozšířenou znakovou sadu zdroje a rozšířenou znakovou sadu výstupu. Navíc mohou být dodatečné znaky, které splňují podmínky pro písmena, součástí jmen identifikátorů. A tak může německá implementace povolit přehlasované samohlásky a francouzská samohlásky s diakritikou. C++ má mechanismus reprezentace mezinárodních znaků, které jsou nezávislé na konkrétní klávesnici a tím je používání *univerzálních jmen znaků*.

Tento mechanismus je podobný escape sekvencím. Univerzální jméno znaku je uvozeno znaky `\u` nebo `\U`. Tvar s `\u` na začátku pokračuje čtyřmi hexadecimálními číslicemi a tvar s `\U` osmi hexadecimálními číslicemi. Tyto číslice představují kód standardu ISO 10646 daného znaku. (ISO 10646 je mezinárodní standard ve vývoji, který stanovuje číselné kódy pro široký rozsah znaků. Viz poznámka „Unicode a ISO 10646“ v této kapitole.)

Jestliže vaše implementace podporuje rozšířené znaky, můžete používat univerzální jména znaků v identifikátorech jako znakové konstanty a v řetězcích. Podívejte se například na následující kód:

```
int k\u00F6rper;
cout << "Let then eat g\u00E2teau.\n";
```

Kód znaku ö ve standardu ISO 10646 je 00F6 a znaku â 00E2. Výše uvedený zdrojový kód tedy přidělí proměnné jméno `körper` a vypíše následující výstup:

```
Let them eat gâteau.
```

Nepodporuje-li váš systém ISO 10646, může místo â vypsat něco jiného, anebo vypíše jen `gu00E2teau`.

Unicode a ISO 10646

Unicode přiděluje číselné kódy velkému množství znaků a symbolů seskupených podle typu, čímž řeší reprezentaci různých znakových sad. Kódy ASCII například představují podmnožinu sady Unicode, takže latinské znaky používané ve Spojených státech, jako jsou A a Z, mají stejné zastoupení v obou systémech. Unicode ale navíc obsahuje další latinské znaky používané v evropských jazycích; dále znaky jiných abeced včetně řecké, cyrilické, hebrejské, arabské, thajské a bengálské; a ideogramy čínštiny a japonštiny. Unicode doposud zahrnuje přes 96 000 symbolů a 49 skriptů a stále se rozvíjí. Pokud se chcete dozvědět více, podívejte se na web konsorcia Unicode *www.unicode.org*.

Mezinárodní organizace pro standardy (International Standard Organization, ISO) zřídila pracovní skupinu pro vývoj standardu ISO 10646 a standardu pro kódování vícejazyčného textu. Skupiny ISO 10646 a Unicode spolupracují od roku 1991 na synchronizaci obou standardů.

Typ signed char a unsigned char

Narozdíl od typu int, není char implicitně znaménkový ani neznaménkový. Která volba bude vybrána, záleží pouze na implementaci jazyka C++, aby mohli vývojáři překladače tento typ co nejlépe přizpůsobit vlastnostem hardwaru. Pokud je pro vás důležité, aby měl typ char určité chování, můžete použít typ signed char nebo unsigned char explicitně:

```
char fodo;                  // může být znaménkový i neznaménkový
unsigned char bar;          // vždy znaménkový
signed char snark;          // vždy neznaménkový
```

Zmiňované odlišnosti jsou zvláště důležité, používáte-li `char` jako číselný typ. Typ unsigned char obvykle reprezentuje rozsah 0 až 255 a `signed char` -128 až 127. Předpokládejme, že chcete proměnnou typu `char` použít pro ukládání hodnot do velikosti 200. Potom zjistíte, že na některých systémech toto funguje a na některých ne. Pro dané účely ovšem můžete na všech systémech s úspěchem používat typ unsigned char. Pokud ale chcete v proměnné `char` ukládat standardní znaky ASCII, vůbec nezáleží na tom, zda je typ `char` znaménkový nebo ne, takže můžete bez obav použít typ char.

Když potřebujete větší velikost: wchar_t

Programy někdy musí pracovat se znakovými sadami, které přesahují hranice 8bitového bajtu; například v japonském systému kanji. Jazyk C++ to může řešit několika způsoby. Zaprvé, jestliže je velká sada znaků základní znakovou sadou implementace, dodavatel překladače může definovat typ `char` jako 16bitový nebo větší bajt. Zadruhé, implementace může podporovat jak malou základní znakovou sadu, tak větší, rozšířenou znakovou sadu. Obvyklý 8bitový `char` představuje základní znakovou sadu a nový typ nazvaný `wchar_t` (pro široký datový typ – *w*ide *char t*ype) může reprezentovat rozšířenou znakovou sadu. Typ `wchar_t` je celočíselný typ s dodatečným prostorem pro vyjádření největší rozšířené znakové sady použité v daném systému. Tento typ má stejnou velikost a znaménko jako některý z celočíselných typů, jemuž říkáme *základní* typ. Volba základního typu závisí na implementaci, takže to může být například `unsigned short` na jednom systému a `int` na jiném.

Rodina objektů `cin` a `cout` předpokládá vstup a výstup jako proud typů `char`. Tyto objekty tudíž nejsou vhodné pro obsluhu typu `wchar_t`. Poslední verze hlavičkového souboru iostream poskytuje obdobné možnosti ve formě objektů `wcin` a `wcout` pro práci s proudy typů `wchar_t`. Rozšířenou znakovou konstantu nebo řetězec můžete tedy vyjádřit předponou `L`. Následující kód uloží verzi písmene P, které je ve verzi `wchart_t` do proměnné `bob` a vypíše slovo `tall` ve verzi `wchart_t`.

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

Na systému s dvoubajtovým typem `wchar_t` ukládá výše uvedený kód každý znak do dvoubajtové jednotky paměti. Tato kniha sice nepoužívá dlouhý znakový typ, ale měli byste o něm vědět. Budete jej potřebovat, pokud se někdy budete účastnit mezinárodního programování nebo budete používat některou ze sad Unicode nebo ISO 10646.

Nový typ bool

Standard ANSI/ISO jazyka C++ zavedl nový typ (to znamená nový pro C++) nazvaný `bool`. Takto byl pojmenován na počest anglického matematika George Boolea, který položil

matematické základy principům logiky. Ve výpočtech může **booleovská proměnná** nabývat hodnot `true` (pravda) nebo `false` (nepravda). V minulosti jazyky C++ ani C booleovské typy neměly. Místo toho, jak podrobněji uvidíte v kapitolách 5 „Cykly a relační výrazy“ a 6 „Příkazy větvení a logické operátory“, C++ interpretoval nenulovou hodnotu jako pravdu a nulovou hodnotu jako nepravdu. Avšak nyní můžete na vyjádření pravdy a nepravdy používat typ `bool` a předdefinované konstanty `true` a `false`, které tyto hodnoty představují. To jest, můžete vytvořit například takovéto příkazy:

```
bool isready = true;
```

Konstanty `true` a `false` lze převést na typ `int`, přičemž hodnota `true` je převedena na 1 a `false` na 0.

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

Všechny numerické a ukazatelové hodnoty mohou také být implicitně převedeny na hodnotu typu `bool` (to znamená bez explicitního přetypování). Všechny nenulové hodnoty jsou převedeny na `true` a nulové na `false`.

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

Až probereme příkaz `if` (v kapitole 6), budeme typ `bool` běžně používat v příkladech.

Kvalifikátor const

Vraťme se nyní k symbolickým jménům konstant. Symbolické jméno může naznačovat, co konstanta představuje. Používá-li program konstantu na několika místech a vy potřebujete změnit její hodnotu, můžete jednoduše změnit pouze jednu definici symbolu. Poznámka o příkazech `#define` uvedená dříve v této kapitole (Symbolické konstanty pomocí preprocesoru) slíbila, že jazyk C++ má lepší způsob zacházení se symbolickými konstantami. Tento způsob spočívá v použití klíčového slova const na úpravu deklarace proměnné a inicializaci. Předpokládejme, že chcete symbolickou konstantu například na počet měsíců v roce. Vložte do programu tento řádek:

```
const int MESICE = 12;      // MESICE je symbolická konstanta pro 12
```

Nyní můžete konstantu `MESICE` v programu používat místo čísla 12. (Samotná číslice 12 může v programu reprezentovat počet palců ve stopě nebo počet koblih v tuctu, ale jméno `MESICE` srozumitelně říká, co znamená.) Po inicializaci konstanty, jako je `MESICE`, je nastavena její hodnota. Potom vám již překladač nepovolí tuto hodnotu změnit. Například Borland C++ vypisuje chybové hlášení, že je požadována `l-hodnota` (`lvalue`). Stejné hlášení dostanete, když se třeba pokusíte přiřadit hodnotu `4` do `3`. (`l-hodnota` je hodnota, jako například proměnná, která se vyskytuje na levé straně přiřazovacího operátoru.) Klíčové slovo `const` se nazývá *kvalifikátor*, protože blíže určuje (kvalifikuje) význam deklarace.

V praxi se obvykle pro jména konstant používají velká písmena, podle nichž poznáte, že `MESICE` je konstanta. Toto není v žádném případě univerzální konvence, ale pomáhá odlišit konstanty od proměnných při čtení programu. Dalšími zvyklostmi je psát pouze první znak jména velkým písmenem nebo začínat konstantu písmenem k, například `kmesice`. Existují i další konvence. Mnoho organizací má vlastní konvence psaní kódu a očekává, že se jich budou jejich programátoři držet.

Obecný tvar pro vytvoření konstanty je tento:

```
const typ jmeno = hodnota;
```

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

```
const int prsty;   // hodnota konstanty prsty není na tomto místě definována
prsty = 10;        // příliš pozdě!
```

Nezadáte-li hodnotu při deklaraci konstanty, bude mít nespecifikovanou hodnotou, kterou již není možné změnit.

Máte-li základy z jazyka C, možná se domníváte, že příkaz `#define`, který jsme si popsali dříve, je pro tuto práci dostatečný. Ale `const` je lepší. Zaprvé umožňuje explicitní určení typu. Zadruhé je možné využít pravidla rozsahu platnosti v jazyce C++ pro omezení definic na určité funkce nebo soubory. (Pravidla rozsahu platnosti popisují, jak široce je jméno známé v různých modulech; o tom se podrobněji dozvíte v kapitole 9 „Paměťové modely a jmenné prostory".) Zatřetí, const můžete použít i pro složitější typy, jako jsou pole a struktury, které přijdou na řadu v kapitole 4.

TIP

Pokud přecházíte na jazyk C++ z jazyka C a chystáte se definovat symbolické konstanty pomocí příkazu `#define`, použijte raději kvalifikátor `const`.

Standard ANSI C také používá kvalifikátor const, který si vypůjčil z C++. Znáte-li verzi ANSI C, měli byste vědět, že verze C++ je trochu odlišná. Jeden rozdíl se vztahuje k pravidlům rozsahu platnosti, tomuto se věnuje kapitola 9. Dalším hlavním rozdílem je, že v C++ (ale ne v C) můžete použít hodnotu označenou jako const při deklaraci velikosti pole. Příklady uvidíte v kapitole 4.

Čísla s pohyblivou desetinnou čárkou

Po přehledu úplné řady celočíselných typů jazyka C++ se podíváme na typy s pohyblivou desetinnou čárkou, které tvoří druhou hlavní skupinu základních typů C++. Tyto typy umožňují vyjadřovat čísla s desetinnou částí, jako například průměrnou rychlost (71,3 km/h). Poskytují také mnohem větší rozsah hodnot. Je-li číslo příliš velké na to, aby ho bylo možné vyjádřit typem `long`, například počet hvězd v naší galaxii (odhadováno 400 miliard), můžete použít některý z typů s pohyblivou desetinnou čárkou.

Typy s pohyblivou desetinnou čárkou mohou představovat čísla, jako jsou 2,5 a 3,14159 nebo 122442,32 – to znamená čísla s desetinnou částí. Počítač takové hodnoty ukládá do dvou částí. Jedna část reprezentuje hodnotu a druhá měřítko, které zvyšuje nebo snižuje hodnotu. Jako příklad si představte dvě čísla 34,1245 a 34124,5. Tato čísla jsou stejná až na měřítko. První můžete vyjádřit jako 0,341245 (základní hodnota) a 100 (měřítko). Druhé jako 0,341245 (stejná základní hodnota) a 100 000 (větší měřítko). Měřítko slouží k posunu desetinné čárky, odtud pochází pojem **pohyblivá desetinná čárka**. Jazyk C++ používá pro interní reprezentaci čísel s pohyblivou desetinnou čárkou podobné metody, kromě toho, že je založena na binárních číslech, takže měřítkem jsou násobky 2 místo 10. Naštěstí toho o vnitřní reprezentaci nemusíte mnoho vědět. Důležité je, že čísla s pohyblivou desetinnou čárkou umožňují vyjadřovat zlomkové, velmi velké a velmi malé hodnoty a mají značně rozdílnou vnitřní reprezentaci oproti celočíselným typům.

Zápis čísel s pohyblivou desetinnou čárkou

Jazyk C++ má dva způsoby zápisu čísel s pohyblivou desetinnou čárkou. Prvním je standardní notace s desetinnou tečkou běžně používanou v anglicky mluvících zemích:

```
12.34          // číslo s pohyblivou desetinnou čárkou
939001.32      // číslo s pohyblivou desetinnou čárkou
0.00023        // číslo s pohyblivou desetinnou čárkou
8.0            // stále číslo s pohyblivou desetinnou čárkou
```

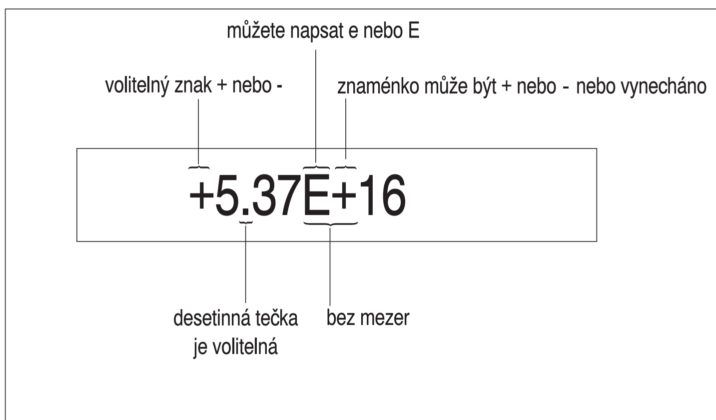
Dokonce i když je desetinná část rovna 0, například 8.0, zaručuje desetinná tečka uložení čísla ve formátu s pohyblivou desetinnou čárkou a ne jako celé číslo. (Standard jazyka C++ neumožňuje implementacím použití různých národních lokalizací, například evropské desetinné čárky místo tečky. Nicméně tyto zvyklosti určují, jak se mohou čísla objevovat na vstupu a výstupu, ale ne v kódu.)

Druhým způsobem zápisu hodnot s pohyblivou desetinnou čárkou je takzvaná E notace, která vypadá například takto: 3.45E6. To znamená, že hodnota 3,45 se násobí 1 000 000; E6 vyjadřuje 10 na 6, což je číselně jednička se šesti nulami. Proto má číslo 3.45E6 hodnotu 3 450 000. V tomto případě je číslo 6 **exponent** a 3.45 **mantisa**. 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      // před číslem může mít 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, E notace je mnohem vhodnější pro velmi velká a velmi malá čísla.

E notace zaručuje, že je číslo uloženo ve formátu s pohyblivou desetinnou čárkou, dokonce i když nebyla použita desetinná tečka. Všimněte si, že můžete použít jak `E`, tak `e` a exponent může mít kladné i záporné znaménko (viz obrázek 3.3). V čísle se však nesmí vyskytovat mezery: Číslo `7.2 E6` je neplatné.



Obrázek 3.3 E notace

Použití záporného exponentu představuje dělení mocninou 10 namísto násobení mocninou 10. Takže 8.33E-4 znamená 8,33 1044 nebo 0,000833. Obdobně hmotnost elektronu 9.11e-31 kg znamená 0.000000000000000000000000000000911 kg. Takže si můžete vybrat. (Mimochodem, všimněte si, že 911 je ve Spojených státech obvyklé telefonní číslo pro případ nouze a že telefonní zprávy jsou přenášeny elektrony. Shoda okolností, nebo vědecké spiknutí? Rozhodnutí necháme na vás.) Dále si všimněte, že -8.33E4 znamená -83300. Znaménko před číslem se vztahuje k hodnotě čísla, zatímco znaménko u exponentu k měřítku.

ZAPAMATUJTE SI

Tvar d.dddE+n představuje posun desetinné čárky o n míst doprava a d.dddE-n posun o n míst doleva.

Typy s pohyblivou desetinnou čárkou

Podobně jako ANSI C, má i jazyk C++ tři typy s pohyblivou desetinnou čárkou: `float`, `double` a `long double`. Tyto typy popisuje počet významových číslic a minimální rozsah exponentu. **Významové číslice** jsou pro číslo důležité. Například zápis výšky hory Shasta v Kalifornii 4 317 metrů používá čtyři významové číslice, ale napíšeme-li, že je Shasta asi 4 000 metrů vysoká, použijeme pouze jednu významovou číslici, protože výsledek je zaokrouhlen na nejbližších tisíc metrů. V tomto případě tři zbývající číslice pouze vymezují místo. Počet významových číslic nezávisí na umístění desetinné čárky. Danou výšku můžeme například zapsat jako 4,317 kilometru. Zde jsou opět použity čtyři významové číslice a tato hodnota je vyjádřena s přesností čtyř číslic.

Požadavky jazyků C a C++ na množství významových číslic jsou u typu `float` alespoň 32 bitů, pro `double` alespoň 48 bitů a ne méně než u typu `float`, a pro `long double` alespoň tolik jako pro `double`. Všechny tři typy mohou mít stejnou velikost. Avšak `float` má obvykle 32 bitů, double 64 bitů a `long double` 80, 96 nebo 128 bitů. Také rozsah exponentů je pro všechny tři typy přinejmenším -37 až +37. Chcete-li zjistit meze vašeho systému, můžete se podívat do hlavičkových souborů `cfloat` nebo `float.h` (Soubor `cfloat` je C++ verze souboru `float.h` jazyka C.) Zde jsou například poznámkami opatřené ukázky ze souboru `float.h` pro Borland C++ Builder:

```
// následující definice představují minimální počty významových 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 exponentů
#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

Všechny implementace jazyka C++ ještě nejsou opatřeny hlavičkovým souborem `cfloat` a některé implementace založené na překladačích jazyka C z doby před normou ANSI C nemají hlavičkový soubor `float.h`.

Výpis 3.8 zkoumá typy `float` a `double` a různou přesnost jejich vyjadřování čísel (počet významových číslic). Program předvádí metodu `setf()` třídy `ostream`, se kterou se blíže seznámíte v kapitole 17. Uvedené volání této funkce nutí výstup, aby ponechal zápis desetinných čísel ve formátu s desetinnou tečkou, takže můžete vidět jejich přesnost. Program potom nemůže výstup velkých čísel přepnout na E notaci a zobrazuje vpravo od desetinné tečky šest číslic. Parametry `ios_base::fixed` a `ios_base::floatfield` jsou konstanty dodané vloženým souborem `iostream`.

Výpis 3.8 floatnum.cpp

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

   cout << "skopek = " << tub;
   cout << ", milion skopku = " << million * tub;
   cout << ",\na deset milionu skopku = ";
   cout << 10 * million * tub << endl;

   cout << "ranec = " << mint << " a milion rancu = ";
   cout << million * mint << endl;
   return 0;
}
```

Zde je výstup z programu uvedeného ve výpisu 3.8:

```
skopek = 3.333333, milion skopku = 3333333.250000,
a deset milionu skopku = 33333332.000000
ranec = 3.333333 a milion rancu = 3333333.333333
```

KOMPATIBILITA

Standard jazyka C++ nahradil konstantu `ios::fixed` konstantou `ios_base::fixed` a konstantu `ios::floatfield` konstantou `ios_base::floatfield`. Pokud váš překladač nepracuje s tvary začínajícími na `ios_base`, zkuste místo toho `ios`; to znamená vyměňte `ios_base::fixed` za `ios::fixed` atd. Starší verze jazyka C++ zobrazují implicitně hodnoty s pohyblivou desetinnou čárkou s šesti číslicemi napravo od desetinné

tečky, například `2345.831541`. Standard jazyka C++ zobrazuje implicitně šest číslic celkově (`2345.83`), pokud hodnota dosáhne milionu a více, přepne se do `E` notace (`2.34583E+06`). Avšak explicitní zobrazovací režimy, jako například `fixed` v příkladu, vypisují napravo od desetinné tečky šest číslic ve starých i nových verzích.

Implicitní nastavení také potlačuje koncové nuly, zobrazuje `23.4500` jako `23.45`. Implementace se liší v tom, jak odpovídají na použití příkazu `setf()`, který ruší implicitní nastavení. Starší verze, jako je například Borland C++ 3.1 pro DOS, potlačují koncové nuly i v tomto režimu. Novější verze, jako například Microsoft Visual C++ 7.0, Metrowerks CodeWarrior 9, Gnu GCC 3.3 a Borland C++ 5.5, zobrazují nuly tak, jak je uvedeno na výpisu 3.8.

Poznámky k programu

Objekt cout koncové nuly obvykle nevypisuje. Například číslo 3333333.250000 by zobrazil jako 3333333.25. Volání `cout.setf()` toto chování alespoň v nových implementacích potlačuje. Nejdůležitější je, abyste si všimli, že `float` má menší přesnost než `double`. Proměnné `tub` i `mint` jsou inicializovány výrazem 10.0/3.0. Ten by měl být vyhodnocen jako 3.33333333333333333... (atd.). Protože `cout` tiskne napravo od desetinné tečky šest číslic, vidíte, že `tub` i `mint` mají až sem stejnou přesnost. Ale po vynásobení každého čísla milionem zjistíte, že se `tub` odchyluje od přesné hodnoty po sedmé číslici 3. Proměnná `tub` má tedy přesnost na sedm významových číslic. (Tento systém zaručuje šest významových číslic, ale jen v nejhorším případě.) Proměnná typu `double` vypisuje třináct číslic 3, takže její přesnost je alespoň třináct významových číslic. Protože systém zaručuje patnáct, nemělo by vás to překvapit. Všimněte si také, že vynásobením milionu `tub` deseti neobdržíte zcela správný výsledek; to opět poukazuje na omezení přesnosti typu `float`.

Třída `ostream`, ke které objekt `cout` patří, má členské funkce umožňující přesné ovládání formátu výstupu – šířky polí, místa napravo od desetinné tečky, desítkový nebo E tvar atd. Tyto volby popisuje kapitola 17. Příklady knihy výstupy zjednodušují a obvykle používají pouze operátor <<. Tento přístup občas zobrazí více číslic, než je nezbytně nutné, ale jedná se pouze o estetický nedostatek. Chcete-li příklady vylepšovat, projděte si kapitolu 17, kde uvidíte použití formátovacích metod. Avšak nečekejte, že nyní výkladu plně porozumíte.

Z praxe

Direktivy vkládání souborů, které se nacházejí na začátku zdrojových souborů jazyka C++ často působí dojmem magického zaklínadla; začínající programátoři jazyka C++ zjistí čtením a zkušenostmi, které hlavičkové soubory dodávají určitou funkčnost a vkládají je pouze, aby jejich programy fungovaly. Nespoléhejte se na hlavičkové soubory pouze jako na zdroj záhadných a tajemných znalostí; klidně je otevřete a čtěte. Jsou to textové soubory, takže jejich čtení je snadné. Všechny soubory, které vkládáte do svých programů, se nachází buďto přímo v počítači nebo na místě počítači dostupném. Zkuste najít vámi vkládané soubory a podívat se, co obsahují. Brzy zjistíte, že používané zdrojové a hlavičkové soubory jsou výborným zdrojem znalostí a informací – v mnoha případech také představují nejlepší dostupnou dokumentaci. Až později přikročíte ke složitějšímu vkládání a začnete používat další nestandardní knihovny, bude se vám tento zvyk velmi hodit.

Konstanty s pohyblivou desetinnou čárkou

Když napíšete v programu konstantu s pohyblivou desetinnou čárkou, v jakém typu ji program uloží? Standardně jsou konstanty s pohyblivou desetinnou čárkou, jako například 8.24 a 2.4E8, typu `double`. Chcete-li, aby měla konstanta typ `float`, použijte příponu `f` nebo `F`. Pro typ `long double` použijte příponu `l` nebo `L`. (Protože malé `l` vypadá jako číslice `1`, je lépe použít velké `L`). Zde je několik příkladů:

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

Výhody a nevýhody typů s pohyblivou desetinnou čárkou

Čísla s pohyblivou desetinnou čárkou mají oproti celým číslům dvě výhody. Zaprvé mohou reprezentovat čísla, která leží mezi celými čísly. Zadruhé, díky násobení měřítkem mají mnohem větší rozsah hodnot. Na druhé straně jsou operace s pohyblivou desetinnou čárkou pomalejší než operace s celými čísly, alespoň na počítačích bez matematického koprocesoru, a navíc může docházet ke ztrátám přesnosti. Výpis 3.9 ukazuje naposledy zmíněný problém.

Výpis 3.9 fltadd.cpp

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

   cout << "a = " << a << endl;
   cout << "b - a = " << b - a << endl;
   return 0;
}
```

KOMPATIBILITA

Některé staré implementace C++ založené na překladačích jazyka C z doby před vydáním normy ANSI C nepodporují pro konstanty s pohyblivou desetinnou čárkou příponu `f`. Zjistíte-li tento problém, můžete konstantu 2.34E+22f nahradit 2.34E+22 a `1.0f` pomocí (`float`) `1.0`.

Program na výpisu 3.9 přičte k číslu hodnotu 1 a od výsledku odečte původní číslo. Na výstupu bychom měli dostat hodnotu 1. Je to pravda? Zde je výstup na jednom systému:

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

Problém spočívá v tom, že výraz 2.34+22 představuje číslo s 23 číslicemi vlevo od desetinné čárky. Přičtením hodnoty 1 vlastně přičítáte jedničku k 23. číslici tohoto čísla. Ale typ float rozlišuje pouze 6 nebo 7 prvních číslic, takže snaha změnit 23. číslici nemá žádný vliv na výslednou hodnotu.

Klasifikace datových typů

V jazyce C++ jsou základní typy rozděleny do skupin. Typy `signed char`, `short`, `int` a `long` jsou označovány jako *celočíselné znaménkové* typy. Jejich verzím bez znamének říkáme *celočíselné neznaménkové* typy. `bool`, `char`, `wchar_t`, celočíselné znaménkové a celočíselné neznaménkové typy se dohromady označují jako *integrální* nebo *celočíselné* typy. `float`, `double` a `long double` jsou typy *s pohyblivou desetinnou čárkou*. Celočíselné typy a typy s pohyblivou desetinnou čárkou souhrnně označujeme jako *aritmetické* typy.

Aritmetické operátory jazyka C++

Možná máte stále v živé paměti početní cvičení ze základní školy. Stejné potěšení můžete dopřát i vašemu počítači. Jazyk C++ provádí aritmetické výpočty pomocí operátorů. Poskytuje operátory pro pět základních aritmetických výpočtů: sčítání, odčítání, násobení, dělení a výpočet zbytku po celočíselném dělení. Každý z těchto operátorů potřebuje pro vypočítání výsledku dvě hodnoty (takzvané **operandy**). Operátor tvoří spolu se svými operandy **výraz**. Podívejte se například na následující příkaz:

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

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

Zde je pět základních aritmetických operátorů jazyka C++:

- Operátor `+` sčítá operandy. Například výraz `4 + 20` je vyhodnocen jako `24`.
- Operátor `-` odečítá druhý operand od prvého. Například výraz `12 - 3` je vyhodnocen jako `9`.
- Operátor `*` násobí operandy. Například výraz `28 * 4` je vyhodnocen jako `112`.
- Operátor `/` dělí první operand druhým. Například výraz `1000 / 5` je vyhodnocen jako `200`. Jsou-li oba operandy celá čísla, výsledkem je celá část podílu. Například výraz `17 / 3` má hodnotu `5` a desetinná část je odříznuta.
- Operátor `%` zjistí zbytek po celočíselném dělení jeho prvního operandu s ohledem na druhý. To znamená, že výsledkem je zbytek po dělení prvého operandu druhým. Například hodnota výrazu `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.

Za operandy můžete samozřejmě dosadit proměnné stejně jako konstanty, což právě předvádí výpis 3.10. Protože operátor `%` pracuje pouze s celými čísly, použijeme ho v dalším příkladu.

Výpis 3.10 arith.cpp

```
// arith.cpp -- nekteré aritmeticke vypocty v C++
#include <iostream>
int main()
{
```

```
    using namespace std;
    float klobouky, hlavy;
    cout.setf(ios_base::fixed, ios_base::floatfield); // pevná řádová tečka
    cout << "Zadejte cislo: ";
    cin >> klobouky;
    cout << "Zadejte dalsi cislo: ";
    cin >> hlavy;

    cout << "klobouky = " << klobouky << "; hlavy = " << hlavy << endl;
    cout << "klobouky + hlavy = " << klobouky + hlavy << endl;
    cout << "klobouky - hlavy = " << klobouky - hlavy << endl;
    cout << "klobouky * hlavy = " << klobouky * hlavy << endl;
    cout << "klobouky / hlavy = " << klobouky / hlavy << endl;
    return 0;
}
```

KOMPATIBILITA

Jestliže váš překladač neumí pracovat ve funkci `setf()` s tvary začínajícími na `ios_base`, zkuste místo toho starší `ios`, to znamená napište místo `ios_base::fixed` výraz `ios::fixed` atd.

Zde je příklad výstupu programu z výpisu 3.10. Jak vidíte, jazyku C++ můžete při provádění jednoduchých aritmetických výpočtů důvěřovat:

```
Zadejte cislo: 50.25
Zadejte dalsi cislo: 11.17
klobouky = 50.250000; hlavy = 11.170000
klobouky + hlavy = 61.419998
klobouky - hlavy = 39.080002
klobouky * hlavy = 561.292480
klobouky / hlavy = 4.498657
```

Dobře, možná mu nemůžete důvěřovat úplně. Součet hodnot 11,17 a 50,25 by měl vrátit 61,42, ale na výstupu je 61,419998. To není chyba ve výpočtu; tento problém je způsoben omezenou kapacitou významových číslic typu `float`. Pamatujte, že jazyk C++ zaručuje pro typ `float` pouze šest významových číslic. Zaokrouhlíte-li 61,419998 na šest číslic, dostanete 61,4200, což je správná hodnota v zaručené přesnosti. Pokud potřebujete větší přesnost, použijte `double` nebo `long double`.

Jaké pořadí: Priorita a asociativita operátorů

Můžete jazyku C++ důvěřovat při provádění složitých aritmetických výpočtů? Ano, ale musíte znát jeho pravidla. Mnoho výrazů například obsahuje více než jeden operátor. To může vyvolat otázku, který operátor je vykonán první. Podívejte se například na tento příkaz:

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

Zdá se, že číslo 4 je operandem obou operátorů `+` i `*`. Pokud může být stejný operand zpracován více než jedním operátorem, jazyk C++ rozhodne, který operátor bude použit první, podle pravidel **priority**. Aritmetické operátory dodržují obvyklé algebraické pořadí. Násobení, dělení a zjišťování zbytku po celočíselném dělení se provádí před sčítáním a odčítáním. Proto `3 + 4 * 5` znamená `3 + (4 * 5)`, nikoli `(3 + 4 ) * 5`. Takže výsled-

kem je 23, ne 35. Vaše vlastní priority si samozřejmě můžete vynutit pomocí závorek. Příloha D „Priorita operátorů“ shrnuje priority všech operátorů v jazyku C++. Zde si můžete všimnout, že operátory *, / a % jsou uvedeny na stejném řádku. To znamená, že mají stejnou prioritu. Podobně sčítání a odčítání sdílejí nižší prioritu.

V některých případech s prioritami nevystačíme. Představte si následující příkaz:

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

Číslo 4 je opět operand 4 se dvěma operátory. Ale operátory / a * mají stejnou prioritu, takže program neví, zda má nejprve 120 dělit 4 nebo 4 násobit 5. Protože první možnost má výsledek 150 a druhá 6, je toto rozhodnutí důležité. U dvou operátorů se stejnou prioritou jazyk C++ zjišťuje, zda mají **asociativitu** zleva doprava nebo zprava doleva. Asociativita zleva doprava říká, že pokud mají dva operátory jednoho operandu stejnou prioritou, je nejprve použit levý operátor. Při asociativitě zprava doleva se aplikuje první operátor na pravé straně. Informace o asociativitě jsou shrnuty v Příloze D. Zde se dozvíte, že násobení a dělení jsou asociativní zleva doprava. To znamená, že operand 4 použijete nejprve s levým operátorem. Čili 120 dělíte 4, dostanete 30 jako výsledek a ten potom násobíte 5, čímž získáte 150.

Všimněte si, že pravidla o prioritě a asociativitě se uplatňují pouze tehdy, když dva operátory sdílejí stejný operand. Ve výrazu:

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

priorita operátorů říká, že program musí před sčítáním vyhodnotit výrazy 20 * 5 a 24 * 6. Ale ani priorita ani asociativita neurčuje, které násobení má být první. Možná se domníváte, že podle asociativity by mělo mít přednost levé násobení, ale v tomto případě operátory nesdílejí společný operand, takže toto pravidlo nelze použít. Ve skutečnosti jazyk C++ nechává na implementaci, aby rozhodla, které pořadí pracuje nejlépe na daném systému. V našem příkladu mají obě pořadí stejný výsledek, ale v některých případech může mít zvolené pořadí na konečný výsledek vliv. Jeden uvidíte v kapitole 5 při probírání operátoru inkrementace.

Odlišnosti dělení

Ještě se musíme zastavit u operátoru dělení (/). Jeho chování závisí na typu operandů. Jsou-li oba operandy celočíselné, C++ provádí celočíselné dělení. To znamená, že zlomková část výsledku je oddělena a výsledkem je celé číslo. Jsou-li jeden nebo oba operandy typy s pohyblivou desetinnou čárkou, desetinná část zůstává a výsledkem je hodnota s pohyblivou desetinnou čárkou. Výpis 3.11 ukazuje, jak dělení v C++ pracuje s různými typy hodnot. Stejně jako ve výpisu 3.10 volá členskou funkci setf() pro nastavení zobrazování výsledků.

Výpis 3.11 divide.cpp

```
// divide.cpp -- deleni celociselne a v poh. des. carce
#include <iostream>
int main()
{

    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Celociselne deleni: 9/5 = " << 9 / 5 << endl;
```

```
    cout << "Deleni v poh. des. carce: 9.0/5.0 = ";
    cout << 9.0 / 5.0 << endl;
    cout << "Kombinovane deleni: 9.0/5 = " << 9.0 / 5 << endl;
    cout << "konstanty typu double: 1e7/9.0 = ";
    cout << 1.e7 / 9.0 << endl;
    cout << "konstanty typu float: 1e7f/9.0f = ";
    cout << 1.e7f / 9.0f << endl;
    return 0;
}
```

KOMPATIBILITA

Jestliže váš překladač neumí pracovat ve funkci `setf()` s tvary začínajícími na `ios_base`, zkuste místo toho starší `ios`.

Některé staré implementace C++ založené na překladačích jazyka C z doby před vydáním normy ANSI C nepodporují pro konstanty s pohyblivou desetinnou čárkou příponu `f`. Zjistíte-li tento problém, můžete výraz `1.e7f / 9.0f` nahradit výrazem `(float) 1.e7/(float) 9.0`.

Některé implementace potlačují koncové nuly.

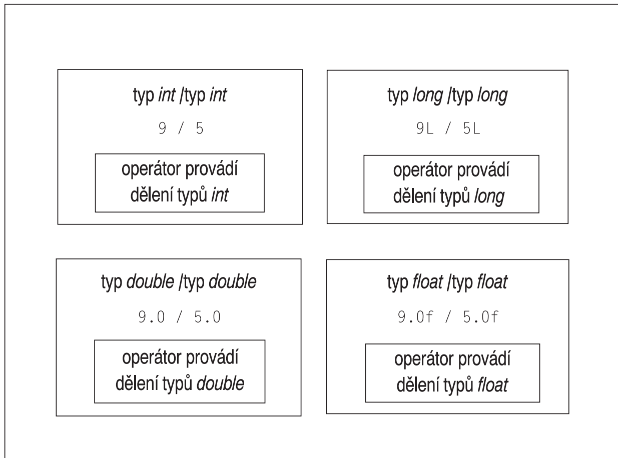
Zde je výstup programu uvedeného na výpisu 3.11 pro jednu implementaci:

```
Celociselne deleni: 9/5 = 1
Deleni v poh. des. carce: 9.0/5.0 = 1.800000
Kombinovane deleni: 9.0/5 = 1.800000
konstanty typu double: 1e7/9.0 = 1111111.111111
konstanty typu float: 1e7f/9.0f = 1111111.125000
```

První řádek výstupu ukazuje, že výsledkem dělení celého čísla 9 celým číslem 5 je celé číslo 1. Zlomková část 4 / 5 (nebo 0.8) se odděluje. Praktické použití tohoto druhu dělení uvidíte, až se budete učit o operátoru zbytku po celočíselném dělení. Následující dva řádky dokazují, že pokud má alespoň jeden z operandů typ s pohyblivou desetinnou čárkou, dostanete výsledek s pohyblivou desetinnou čárkou 1.8. Jestliže se pokusíte kombinovat různé typy, C++ ve skutečnosti převede všechny typy, kterých se to týká, na stejný typ. O těchto automatických převodech si více povíme později v této kapitole. Relativní přesnost posledních dvou řádků dokazuje, že výsledkem je typ `double`, pokud jsou oba operandy `double`, ale jestliže jsou oba operandy `float`, výsledek má typ `float`. Pamatujte si, že konstanty s pohyblivou desetinnou čárkou jsou implicitně typu `double`.

Letmý pohled na přetěžování operátorů

Ve výpisu 3.11 představuje operátor dělení tři různé operace: dělení typů `int`, `float` a `double`. C++ určuje podle souvislostí, v tomto případě typů operandů, který operátor je vyvolán. Procesu používání stejného symbolu pro více než jednu operaci říkáme *přetížení operátoru*. C++ má několik příkladů přetížení vestavěných do jazyka. Navíc umožňuje rozšiřovat přetížení operátoru na uživatelsky definované třídy, takže zde vlastně vidíte předchůdce důležité vlastnosti OOP (viz obrázek 3.4).



Obrázek 3.4 Různá dělení

Operátor modulo (zbytek po celočíselném dělení)

Většina lidí zná spíše sčítání, odčítání, násobení a dělení než operátor modulo, proto si tento operátor předvedeme v příkladu. Operátor modulo vrací zbytek po celočíselném dělení. Spolu s celočíselným dělením je operátor modulo velmi užitečný při dělení množství na různé celočíselné jednotky, jako je například převod palců na stopy nebo dolarů na čtvrťáky, desetníky, pěticenty a centy. V kapitole 2 „Vzhůru do světa C++" výpis 2.6 převáděl váhu v britských kamenech na libry. Výpis 3.12 tento proces obrací, převádí váhu v librách na kameny. Kámen, jak si pamatujete, je 14 liber a většina britských osobních vah má tyto jednotky. Program používá celočíselné dělení pro nalezení největšího čísla v celých kamenech ve váze a operátorem modulo zjistí počet zbývajících liber.

Výpis 3.12 modulus.cpp

```
// modulus.cpp -- použití operátoru % při konverzi liber na kameny
#include <iostream>
int main()
{
   using namespace std;
   const int Lbs_per_stn = 14;
   int lbs;

   cout << "Zadejte svoji vahu v librach: ";
   cin >> lbs;
   int stone = lbs / Lbs_per_stn;  // celé kameny
   int pounds = lbs % Lbs_per_stn; // zbytek v librách
   cout << lbs << " liber je " << stone
        << " kamenu, " << pounds << " liber.\n";
   return 0;
}
```

Zde je příklad běhu programu z výpisu 3.12:

```
Zadejte svoji vahu v librach: 177
177 liber je 12 kamenu, 9 liber.
```

Ve výrazu `lbs / Lbs_per_stn` jsou oba operandy typu `int`, takže počítač provádí celočíselné dělení. Pokud má proměnná `lbs` hodnotu 177, je výraz vyhodnocen jako 12. Součin 12 x 14 je 168, takže zbytek po dělení 177 / 14 je 9 a to je právě hodnota výrazu `lbs % Lbs_per_stn`. Nyní jste technicky, i když možná ne psychicky, připraveni odpovídat na otázky týkající se vaší váhy při cestách do Velké Británie.

Typové konverze

Množství typů jazyka C++ umožňuje výběr v souladu s vašimi potřebami. Tato skutečnost ale komplikuje život počítačům. Například sečtení dvou hodnot typu `short` může vyžadovat jiné hardwarové instrukce než sečtení dvou hodnot typu `long`. S jedenácti celočíselnými typy a třemi s pohyblivou desetinnou čárkou může počítač zpracovávat mnoho různých případů, zvláště při míchání typů. Jazyk C++ provádí množství typových konverzí automaticky, aby pomohl zvládnout tento možný zmatek:

- C++ konvertuje hodnoty, když přiřazujete hodnotu jednoho aritmetického typu proměnné s jiným typem.
- C++ konvertuje hodnoty, pokud použijete ve výrazu různé typy.
- C++ konvertuje hodnoty při předávání argumentů funkcím.

Pokud nevíte, co se v těchto automatických konverzích děje, mohou se vám zdát některé výsledky programů záhadné, a tak se na tato pravidla podíváme podrobněji.

Konverze během přiřazování

Jazyk C++ je docela velkorysý, umožňuje přiřazování číselné hodnoty jednoho typu do proměnné jiného typu. Kdykoli tak učiníte, je přiřazovaná hodnota převedena na typ přijímající proměnné. Předpokládejme, že proměnná `so_long` má typ `long`, `thirty` typ `short` a v programu se nachází následující příkaz:

```
so_long = thirty;     // přiřadí proměnnou typu short do proměnné typu long
```

Hodnota proměnné `thirty` (obvykle 16bitová) je rozšířena během přiřazení na hodnotu typu `long` (obvykle 32bitová). Pamatujte si, že toto rozšíření vytváří novou hodnotu, která je vložena do proměnné `so_long`; obsah proměnné thirty zůstává nezměněn.

Přiřazení hodnoty do typu s větším rozsahem bývá obvykle bez problémů. Například přiřazení hodnoty typu `short` do proměnné typu `long` nemění hodnotu, pouze jí dodává několik nevyužitých bajtů navíc. Avšak přiřazení velké hodnoty `long`, například 2111222333, do proměnné `float` má za následek určitou ztrátu přesnosti. Protože typ `float` může mít pouze šest významových číslic, může být tato hodnota zaokrouhlena na 2.11122E9. Tabulka 3.3 ukazuje některé možné konverzní problémy.

Tabulka 3.3 Možné konverzní problémy

Konverze	Možné problémy
Větší typ s pohyblivou desetinnou čárkou na menší typ s pohyblivou desetinnou čárkou, například `double` na `float`	Ztráta přesnosti (významné číslice), hodnota může být mimo rozsah cílového typu, v takovém případě je výsledek nedefinovaný
Typ s pohyblivou desetinnou čárkou na celočíselný typ	Ztráta zlomkové části, původní hodnota může být mimo rozsah cílové hodnoty, v takovém případě je výsledek nedefinovaný

Konverze	Možné problémy
Větší celočíselný typ na menší celočíselný typ, například `long` na `short`	Původní hodnota může být mimo rozsah cílové hodnoty, obvykle jsou kopírovány pouze bajty nižšího řádu

Nulová hodnota přiřazená do proměnné `bool` je převedena na false a nenulová hodnota na `true`.

Přiřazení hodnoty s pohyblivou desetinnou čárkou do celočíselného typu představuje několik problémů. Zaprvé, konvertování hodnoty s pohyblivou desetinnou čárkou na celé číslo má za následek oříznutí čísla (zrušení desetinné části). Zadruhé, hodnota typu `float` může být příliš velká a nevejde se do proměnné typu int. V tomto případě jazyk C++ nedefinuje výsledek; to znamená, že různé implementace mohou odpovídat různě. Výpis 3.13 předvádí několik konverzí během přiřazení.

Výpis 3.13 assign.cpp

```
// assign.cpp - zmena typu pri prirazeni
#include <iostream>
int main()
{
    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield);
    float tree = 3;              // int se konvertuje na float
    int guess = 3.9832;          // float se konvertuje na int
    int debt = 7.2E12;           // výsledek není v C++ definován
    cout << "tree = " << tree << endl;
    cout << "guess = " << guess << endl;
    cout << "debt = " << debt << endl;
    return 0;
}
```

Zde je výstup programu z výpisu 3.13 na jednom systému:

```
tree = 3.00000
guess = 3
debt = 1634811904
```

Proměnné `three` je přiřazena hodnota s pohyblivou desetinnou čárkou 3,0. Ale protože objekt `cout` nevypisuje koncové nuly, je hodnota 3,0 zobrazena jako 3. Přiřazení hodnoty 3,9832 do proměnné `guess` typu `int` způsobuje oříznutí této hodnoty na 3; jazyk C++ používá při převodu typů s pohyblivou desetinnou čárkou na celočíselné zkrácení (odříznutí zlomkové části) a nikoli zaokrouhlení (nalezení nejbližší celočíselné hodnoty). Nakonec si všimněte, že proměnná `debt` typu int nemůže obsahovat hodnotu 3.0E12 a výsledek takového přiřazení není v C++ definován. Na tomto systému má proměnná `debt` hodnotu 1634811904, což je přibližně 1.6E09. Hle, jak snadno lze snížit tak velký dluh.

Některé překladače varují před možnou ztrátou dat u příkazů, které inicializují celočíselné proměnné hodnotami s pohyblivou desetinnou čárkou. Také hodnota proměnné `debt` se mění od překladače k překladači. Kdybychom spustili program 3.13 na jiném systému, hodnota proměnné `debt` může nabýt hodnoty například 2147483647.

Konverze ve výrazech

Dále se podíváme na to, co se stane, když vložíme do jednoho výrazu dva různé aritmetické typy. Jazyk C++ v takovém případě provádí dva druhy automatických konverzí. První, některé typy jsou konvertovány automaticky při jejich výskytu. Druhý, některé typy jsou konvertovány, když se vyskytnou ve výrazu spolu s jinými typy.

Nejprve si vyzkoušíme automatické konverze. Při vyhodnocování výrazů převádí jazyk C++ hodnoty typů `bool`, `char`, `unsigned char`, `signed char` a `short` na `int`. Přičemž je hodnota `true` převedena na 1 a `false` na 0. Těmto konverzím říkáme *celočíselná povýšení*. Podívejme se například na následující příkazy sčítání drůbeže:

```
short kurata = 20;                    // řádek 1
short kachny = 35;                    // řádek 2
short drubez = kurata + kachny;       // řádek 3
```

Při vykonávání příkazu na řádku 3 program jazyka C++ převede hodnoty proměnných kurata a kachny na `int`. Výsledek součtu poté převede zpět na typ `short`, protože je přiřazen proměnné typu `short`. Tento postup se vám může zdát poněkud zdlouhavý, ale má svůj význam. Typ `int` je obecně považován za nejpřirozenější počítačový typ, což znamená, že počítač bude s tímto typem pravděpodobně provádět výpočty nejrychleji.

Existuje několik dalších celočíselných povýšení: typ `unsigned short` je konvertován na `int`, jestliže je `short` menší než `int`. Pokud mají oba typy stejnou velikost, je `unsigned short` konvertován na `unsigned int`. Toto pravidlo zaručuje, že při povýšení typu `unsigned short` nedojde ke ztrátě dat. Obdobně je povýšen typ `wchar_t` na první z následujících typů, který je dostatečně veliký na uložení jeho celého rozsahu: `int`, `unsigned int`, `long` nebo `unsigned long`.

Dále existují konverze, ke kterým dochází při aritmetické kombinaci různých typů, například sčítání typu `int` a `float`. Pokud některá operace obsahuje dva různé typy, je menší převeden na větší. Například program ve výpisu 3.11 dělí hodnotu 9.0 hodnotou 5. Protože 9.0 má typ `double`, program před vlastním dělením konvertuje číslo 5 na typ `double`. Obecně platí, že překladač provádí konverze v aritmetickém výrazu podle následujícího seznamu (v uvedeném pořadí):

1. Pokud je některý z operandů typu `long double`, druhý je převeden také na `long double`.
2. Jinak je-li některý z operandů typu `double`, druhý je převeden také na `double`.
3. Jinak je-li některý z operandů typu `float`, druhý je převeden také na `float`.
4. Jinak mají-li operandy celočíselné typy, je provedeno celočíselné povýšení.
5. V takovém případě, je-li některý z operandů typu `unsigned long`, druhý je převeden také na `unsigned long`.
6. Jinak je-li jeden operand typu `long int` a druhý `unsigned int`, konverze závisí na relativní velikosti obou typů. Jestliže může `long` reprezentovat možné hodnoty typu `unsigned int`, je `unsigned int` převeden na `long`.
7. Jinak jsou oba operandy převedeny na `unsigned long`.
8. Jinak je-li některý z operandů `long`, druhý je převeden také na `long`.
9. Jinak je-li některý z operandů `unsigned int`, druhý je převeden také na `unsigned int`.
10. Když překladač dojde v seznamu až sem, oba operandy by měly být typu `int`.

ANSI C pracuje podle stejných pravidel jako C++, ale klasický jazyk K&R C měl pravidla trochu odlišná. Klasický jazyk C například vždy konvertoval float na double, i když byly oba operandy typu float.

Konverze při předávání argumentů

Typové konverze při předávání argumentů obvykle řídí prototypy funkcí jazyka C++, jak se dozvíte v kapitole 7 „Funkce – programové moduly C++". Nicméně je možné, i když obvykle nerozumné, vzdát se řízení předávání argumentů prototypy. V takovém případě jazyk C++ provádí celočíselné povýšení typů `char` a `short` (`signed` a `unsigned`). Z důvodů zachování kompatibility s obrovským množstvím programového kódu napsaného v klasickém C, jazyk C++ také povyšuje argumenty typu `float` na `double`, když je předává funkcím bez prototypů.

Přetypování

Jazyk C++ také umožňuje explicitní vynucení konverze pomocí mechanismu přetypování. (C++ uznává nutnost typových pravidel, ale připouští možnost potlačení těchto pravidel.) Přetypování se vyskytuje ve dvou tvarech. Například pro převod hodnoty typu `int` uložené v proměnné `trn` na typ `long` můžete použít oba následující výrazy:

```
(long) trn                  // vytváří z hodnoty proměnné trn typ long
long (trn)                  // vytváří z hodnoty proměnné trn typ long
```

Přetypování nemění vlastní proměnnou `trn`; ale vytváří novou hodnotu požadovaného typu, kterou lze využít ve výrazu, například takto:

```
cout << int('Q');  // vypíše celočíselný kód písmene 'Q'
```

Obecně platí, že můžete udělat následující:

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

První tvar je pravé C, druhý čisté C++. Nový tvar byl navržen tak, aby se přetypování podobalo volání funkce. Přetypování vestavěných typů tak vypadají stejně jako typové konverze navržené pro uživatelsky definované třídy.

V C++ jsou také čtyři operátory pro přetypování, které jsou poněkud restriktivnější. Jsou popsány v kapitole 15, „Přátelé, výjimky a další". Z těchto čtyř operátorů lze operátor `static_cast<>` použít pro konverzi hodnot z jednoho číselného typu do druhého. Například převod proměnné `trn` na typ `long value` se provede takto:

```
static_cast<long> (trn)     // konverze proměnné trn na typ long
```

Obecně vyjádřeno:

```
static_cast<jménoTypu> (hodnota)  // konverze hodnoty na typ typeName
```

Jak se zmiňujeme také v kapitole 15, Stroustrup cítil, že klasické přetypování, jak je známe z C, má nebezpečně málo omezení.

Výpis 3.14 názorně předvádí oba tvary. Představte si, že první část tohoto výpisu je součástí výkonného programu modelování ekologie, který provádí výpočty s pohyblivou desetinnou čárkou a ty převádí na celé počty ptáků nebo zvířat. Výpočet pro alky nejprve sečte hodnoty s pohyblivou desetinnou čárkou a potom tento součet převede na typ `int` během přiřazení. Ale výpočty pro netopýry a lysky nejprve provádějí přetypování hodnoty s pohyblivou desetinnou čárkou na `int` a teprve potom součet konvertovaných

hodnot. Poslední část programu ukazuje použití přetypování pro zobrazení ASCII kódu hodnoty typu `char`.

Výpis 3.14 typecast.cpp

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

   // v těchto příkazech se hodnoty sčítají jako double,
   // a výsledek se převede na int
   alky = 19.99 + 11.99;

   // v těchto příkazech se hodnoty sčítají jako celá čísla
   netopyri = (int) 19.99 + (int) 11.99;          // původní syntaxe jazyka C
   lysky = int (19.99) + int (11.99);             // nová syntaxe jazyka C++
   cout << "alky = " << alky << ", netopyri = " << netopyri;
   cout << ", lysky = " << lysky << endl;
   char ch = 'Z';
   cout << "Kod znaku " << ch << " je ";          // tiskni jako znak
   cout << int(ch) << endl;                       // tiskni jako číslo
   return 0;
}
```

Zde je výsledek programu na výpisu 3.14:

```
alky = 31, netopyri = 30, lysky = 30
Kod znaku Z je 90
```

Výsledkem prvního součtu 19,99 a 11,99 je 31,98. Přiřazení této hodnoty do proměnné `alky` typu `int` ji ořízne na 31. Ale přetypování před sečtením ořízne tytéž hodnoty na 19 a 11 a do proměnných `netopyri` i `lysky` je vložen výsledek 30. Poslední příkaz s objektem `cout` provádí před zobrazením výsledku přetypování hodnoty typu `char` na `int`. Objekt `cout` potom vytiskne místo znaku celé číslo.

Program předvádí dva důvody pro použití přetypování. Můžete mít hodnoty, které jsou uloženy jako `double`, ale používají se pro výpočet hodnoty typu `int`. Například určování polohy vzhledem k mřížce nebo modelování celočíselných hodnot, jako je populace, pomocí čísel s pohyblivou desetinnou čárkou. V takových případech možná budete chtít, aby výpočty zacházely s těmito hodnotami jako s typy `int`. Pomocí přetypování to můžete provádět přímo. Všimněte si, že alespoň pro výše uvedené hodnoty dostáváte různé výsledky, konvertujete-li napřed 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 plnit jiné požadavky. V tomto výpisu například proměnná `ch` typu `char` obsahuje kód písmena `Z`. Použitím objektu `cout` na proměnnou `ch` dojde k zobrazení znaku Z, protože `cout` ví, že proměnná `ch` má typ `char`. Ale přetypováním `ch` na `int` přinutíte objekt `cout`, aby se přepnul do režimu `int` a vytiskl kód ASCII uložený v proměnné `ch`.

Shrnutí

Základní typy jazyka C++ dělíme do dvou skupin. Jedna skupina představuje hodnoty, které jsou ukládány jako celá čísla. Do druhé skupiny patří hodnoty ukládané ve tvaru s pohyblivou desetinnou čárkou. Celočíselné typy se liší velikostí paměti, kterou používají pro 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 int`, `long` a `unsigned long`. Dále se můžeme setkat také s typem `wchar_t`, jehož umístění ve výše uvedené posloupnosti závisí na implementaci. Jazyk C++ zaručuje, že typ `char` je dostatečně velký na uložení jakéhokoli znaku základní znakové sady systému, `wchar_t` může obsahovat libovolný znak rozšířené znakové sady systému, `short` je nejméně 16bitový, `long` je alespoň 32bitový a přinejmenším tak velký jako `int`. Přesné velikosti závisí na implementaci.

Znaky reprezentují číselné kódy. Vstupně/výstupní systém určuje, zda je kód interpretován jako znak nebo jako číslo.

Typy s pohyblivou desetinnou čárkou mohou představovat zlomkové hodnoty nebo hodnoty mnohem větší než celá čísla. Tři typy s pohyblivou desetinnou čárkou se nazývají `float`, `double` a `long double`. Jazyk C++ zaručuje, že `float` není větší než `double` a že `double` není větší než `long double`. Typ `float` obvykle používá 32 bitů paměti, double 64 bitů a `long double` 80 až 128 bitů.

Nabídkou množství typů různých velikostí a ve variantách se znaménkem a bez znaménka umožňuje jazyk C++ zvolit typ podle konkrétních datových požadavků.

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, která operace je provedena dříve.

Jazyk C++ konvertuje hodnoty z jednoho typu na jiný během přiřazování hodnoty do proměnné, smíchání různých typů v aritmetických operacích a přetypování při nucené konverzi typů. Mnoho typových konverzí je „bezpečných", což znamená, že je můžete provádět bez ztráty nebo změny dat. Bez problémů například můžete konvertovat hodnotu `int` na hodnotu `long`. Jiné konverze, jako je třeba převod typů s pohyblivou desetinnou čárkou na celočíselné typy, vyžadují více pozornosti.

Ze začátku se vám možná bude zdát velké množství základních typů jazyka C++ poněkud nepřiměřené, zvláště když vezmete v úvahu různá pravidla konverze. Ale nakonec se s největší pravděpodobností setkáte s případy, kdy bude některý z typů přesně vyhovovat vašim potřebám a budete rádi, že ho jazyk C++ má.

Otázky k opakování

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

   a) celé číslo typu `short` s hodnotou 80

   b) celé číslo typu `unsigned int` s hodnotou 42 110

   c) celé číslo s hodnotou 3 000 000 000

3. Jaké bezpečnostní opatření má jazyk C++ proti překročení mezí celočíselných typů?
4. Jaký je rozdíl mezi konstantou 33L a 33?
5. Podívejte se na dva následující příkazy v jazyce C++.

   ```
   char znamka = 65;
   char znamka = 'A';
   ```

   Jsou stejné?
6. Jak byste pomocí jazyka C++ zjistili, jakou znakovou hodnotu reprezentuje kód 88? Navrhněte alespoň dva způsoby.
7. Přiřazení hodnoty `long` do `float` může mít za následek chybu zaokrouhlení. A co když přiřadíte `long` do `double`?
8. Vyhodnoťte následující výrazy tak, jak by to udělal program jazyka 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 pro to příkaz v C++.

Programátorská cvičení

1. Napište krátký program, který se zeptá na vaši výšku v celých palcích, a potom ji konvertuje na stopy a palce. Podtržítky vyznačte místa pro vepsání zadání. Pro konverzní faktor použijte symbolickou konstantu uvozenou klíčovým slovem `const`.
2. Napište krátký program, který se zeptá na vaši výšku ve stopách a palcích a váhu v librách. (Zadané informace uložte do tří proměnných.) V programu vypočítejte a vypište váš BMI (Body Mass Index – index tělesné hmotnosti). Abyste mohli vypočítat BMI, nejprve převeďte výšku ve stopách a palcích pouze na palce. Potom převeďte palce na metry vynásobením 0,0254. Dále konvertujte váhu v librách na kilogramy dělením 2,2. Nakonec vypočítejte BMI dělením váhy v kilogramech výškou v metrech na druhou. Pro různé konverzní faktory použijte symbolické konstanty.
3. Napište program, který si vyžádá vstup zeměpisné šířky ve stupních, minutách a vteřinách a poté ji zobrazí v dekadickém tvaru. Úhlová minuta má 60 úhlových vteřin, úhlový stupeň má 60 úhlových minut; tyto hodnoty zadejte jako symbolické konstanty. Pro každou vstupní hodnotu použijte jinou proměnnou. Po spuštění by se mělo vypsat:

   ```
   Zadejte zeměpisnou šířku ve stupních, minutách a vteřinách:
   Nejdříve stupně: 37
   Dále úhlové minuty: 51
   Nakonec úhlové vteřiny: 19
   37 stupňů, 51 minut a 19 vteřin = 37,8553 stupňů.
   ```

4. Napište program, který si vyžádá vstup počtu sekund jako celočíselnou hodnotu (použijte typ `long`) a pak vypíše ekvivalent ve dnech, hodinách, minutách a sekundách. Počty hodin ve dni, minut v hodině a sekund v minutě nechť jsou dány symbolickými konstantami. Po spuštění by se mělo vypsat:

```
Zadejte počet sekund: 31600000
31600000 sekund = 365 dní, 46 minut, 40 sekund
```

5. Napište program, který se zeptá, kolik mil jste ujeli a kolik galonů benzinu jste spotřebovali a potom vypíše zprávu o spotřebě vašeho auta v mílích na galon. Nebo se program může dotázat na vzdálenost v kilometrech a na benzin v litrech a potom vypsat zprávu o výsledku v evropském stylu, v litrech na 100 kilometrů.
6. Napište program, který vás požádá o zadání spotřeby benzinu v evropském stylu (litry na 100 kilometrů) a převede ji na americký styl v mílích na galon. Všimněte si, že kromě použití jiných jednotek je americký styl (vzdálenost / palivo) obrácený oproti evropskému (palivo / vzdálenost). Sto kilometrů je 62,14 mil a jeden galon má 3,875 litru. Z čehož vyplývá, že 19 mpg (miles per gallon – mil na galon) je zhruba 12,4 l/100 km a 27 mpg je asi 8,7 l/100 km.

Složené typy

KAPITOLA

4

V této kapitole se naučíte:

- Vytvářet a používat pole
- Vytvářet a používat řetězce v C
- Používat metody getline() a get() pro načítání řetězců
- Vytvářet a používat string – řetězce jako třída
- Zpracovat řetězcový i číselný vstup
- Vytvářet a používat struktury
- Vytvářet a používat operátory union
- Vytvářet a používat výčty
- Vytvářet a používat ukazatele
- Dynamickou správu paměti pomocí klíčových slov new a delete
- Vytvářet dynamická pole
- Vytvářet dynamické struktury
- Automatická, statická a dynamická úložiště

Představte si, že jste vyvinuli počítačovou hru nazvanou Uživatelsky nepřátelský, ve které zkoušejí své štěstí hráči proti tajuplnému a nepřátelskému počítačovému rozhraní. Nyní potřebujete napsat program na sledování měsíční prodejnosti hry po dobu pěti let. Nebo chcete udržovat seznam počtu obchodních karet nazvaných Hacker-hrdina. Brzy zjistíte, že na pokrytí těchto datových požadavků potřebujete trochu víc než jednoduché základní typy a jazyk C++ vám to nabízí ve formě složených typů. Tyto typy jsou vytvářené ze základních celočíselných typů a typů s pohyblivou desetinnou čárkou. Nejpropracovanějším složeným typem je třída, která představuje základ OOP a postupně se k ní dopracujeme. Ale jazyk C++ podporuje také několik jednodušších typů převzatých z C. Například pole může obsahovat několik hodnot stejného typu. Do určitého druhu pole můžeme ukládat řetězce, což jsou řady znaků. Také struktury mohou obsahovat několik hodnot různých typů. Dále se nabízí využití ukazatelů. Tyto proměnné říkají počítači, kde jsou data umístěna. V této kapitole prozkoumáme všechny výše uvedené složené tvary (kromě tříd) a také se podíváme na klíčová slova `new` a `delete` a na jejich využití při správě dat. Nakonec se seznámíte s úvodem do třídy `string` v jazyce C++, což je alternativní způsob práce s řetězci.

Úvod do polí

Pole je forma dat, která může obsahovat několik hodnot stejného typu. Do pole je možné uložit například 60 hodnot typu `int` pro data za pět let prodeje hry, 12 hodnot typu `short` reprezentujících počet dnů v každém měsíci nebo 365 hodnot typu `float`, které zaznamenávají výdaje na potraviny pro všechny dny v roce. Každá hodnota představuje samostatný prvek pole a počítač všechny prvky ukládá za sebou do paměti.

Pole je vytvářeno pomocí deklaračního příkazu. Deklarace pole by měla obsahovat tři informace:

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

V jazyce C++ to provedete úpravou deklarace jednoduché proměnné přidáním hranatých závorek, které obsahují počet prvků. Například deklarace

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

vytváří pole, které se jmenuje `months` a 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:

```
jménoTypu jménoPole [velikostPole];
```

Výraz *`velikostPole`*, 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 překladu známé. Výraz *`velikostPole`* nemůže být proměnnou, jejíž hodnota je nastavena během vykonávání programu. V této kapitole si ale později ukážeme, jak je možné uvedené omezení obejít pomocí operátoru `new`.

Pole jako složený typ

Pole nazýváme *složeným typem*, protože je založeno na jiném typu. (Jazyk C používá výraz „odvozený typ", ale jelikož jazyk C++ vyjadřuje pomocí výrazu „odvozený" vztahy mezi třídami, musel přijít s novým termínem – „složený typ".) Nemůžete jednoduše deklarovat, že něco je polem; vždy to musí být pole nějakého určitého typu. Neexistuje žádný obecný typ pole. Místo toho máme mnoho konkrétních typů polí, jako například pole typů `char` nebo pole typů `long`. Podívejme se například na tuto deklaraci:

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

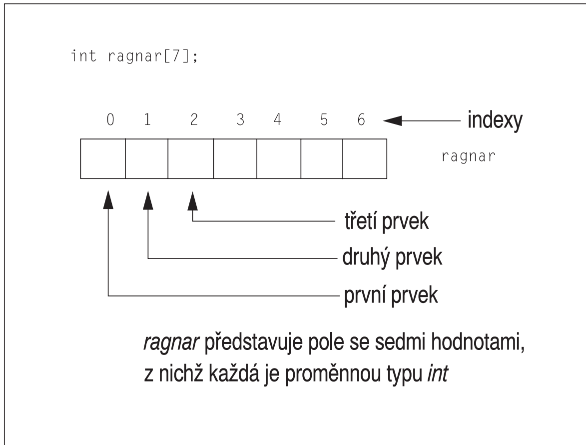
Typem proměnné `loans` není „pole", ale „pole typů `float`". Tím je zdůrazněno, že pole `loans` je postaveno na typu `float`.

Mnoho užitečných vlastností polí vyplývá ze skutečnosti, že k prvkům pole je možné přistupovat jednotlivě. Toto je realizováno pomocí *indexu*, kterým jsou jednotlivé prvky očíslovány. Číslování pole v jazyce C++ začíná hodnotou nula. (Začátek od čísla nula nelze měnit. Uživatelé Pascalu a Basicu se musí přizpůsobit.) C++ používá pro určení prvku pole označení pomocí hranatých závorek s indexem. Například `months[0]` je prvním prvkem pole months a `months[11]` je posledním. Všimněte si, že index posledního prvku je o jedničku menší než velikost pole. (Viz obrázek 4.1.) Deklarace pole tedy dovoluje vytvořit mnoho proměnných jedinou deklarací. Identifikaci a přístup k jednotlivým prvkům pak obstarávají indexy.

Hodnoty indexů musí být platné

Překladač nekontroluje, zda jste použili platný index. Nebude si stěžovat, ani když přiřadíte hodnotu neexistujícímu prvku `months[101]`. Ale takovéto přiřazení může způsobit problémy za běhu programu, možné je poškození dat nebo kódu a může dokonce dojít i k ukončení programu. Takže za používání platných indexů je zodpovědný programátor.

Program na analýzu sladkých brambor (yams) ve výpisu 4.1 ukazuje několik vlastností polí včetně deklarování pole, přiřazení hodnot prvkům pole a inicializování pole.



Obrázek 4.1 Vytvoření pole

Výpis 4.1 arrayone.cpp

```
// arrayone.cpp -- malá pole celých čísel
#include <iostream>
int main()
{
   using namespace std;
   int yams[3];                    // vytváří pole se třemi prvky
   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 překladač nebo převodník C++ neumí
// inicializovat toto pole, použijte static int yamcosts[3] namísto
// int yamcosts[3]

   cout << "Celkovy pocet sladkych brambor = ";
   cout << (yams[0] + yams[1] + yams[2]) << "\n";
   cout << "Balicek s " << yams[1] << " sladkymi bramborami stoji ";
   cout << yamcosts[1] << " centu za jeden sladky brambor.\n";
   int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
   total = total + yams[2] * yamcosts[2];
   cout << "Celkove vydaje za sladke brambory jsou " << total << " centu.\n";

   cout << "\nVelikost pole yams (sladke brambory) = " << sizeof yams;
   cout << " bajtu.\n";
   cout << "Velikost jednoho prvku = " << sizeof yams[0];
   cout << " bajty.\n";
   return 0;
}
```

KOMPATIBILITA

Současné verze C++ stejně jako ANSI C umožňují inicializaci běžných polí definovaných ve funkci. Avšak v některých starších implementacích, které používají převodník jazyka C++ místo pravého překladače, vytváří převodník C++ programový kód v C pro překladač jazyka C, jenž zcela nevyhovuje normě ANSI C. V takových 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
```

Řešením tohoto problému je použití klíčového slova `static` v deklaraci pole:

```
// tvar inicializace před vydáním ANSI C
static int yamcosts[3] = {20, 30, 5};
```

Klíčové slovo `static` řekne překladači, aby používal pro uložení pole jiné paměťové schéma, které umožňuje inicializaci i pro překladače z doby před ANSI C. V kapitole 9 „Paměťové modely a jmenné prostory" se o použití klíčového slova `static` dozvíte více.

Zde je výstup pogramu z výpisu 4.1:

```
Celkovy pocet sladkych brambor = 21
Balicek s 8 sladkymi bramborami stoji 30 centu za jeden sladky brambor.
Celkove vydaje za sladke brambory jsou 410 centu.
Velikost pole yams (sladke brambory) = 12 bajtu.
Velikost jednoho prvku = 4 bajty.
```

Poznámky k programu

Program nejprve vytváří tříprvkové pole nazvané `yams`. Jeho tři prvky jsou očíslovány od 0 do 2 a programový kód v souboru `arrayone.cpp` používá indexy 0 až 2 během přiřazování hodnot těmto třem jednotlivým prvkům. Každý prvek pole `yams` je typu `int` se všemi právy a výhodami typu `int`, takže `arrayone.cpp` může provádět a také provádí přiřazení hodnoty prvkům, sčítání, násobení a zobrazení prvků.

Program používá pro přiřazení hodnot prvkům pole `yams` zdlouhavý způsob. Jazyk C++ umožňuje inicializaci prvků pole také v deklaračním příkazu. Výpis 4.1 používá pro přiřazení hodnot prvkům pole `yamcosts` následující zkrácený zápis:

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

Jednoduše poskytuje seznam hodnot oddělených čárkou (inicializační seznam) uzavřený do složených závorek. Mezery v seznamu jsou volitelné. Pokud neinicializujete pole, které je definováno uvnitř funkce, hodnoty prvků zůstanou nedefinované. To znamená, že prvek může obsahovat jakoukoli hodnotu, která se dříve nacházela na odpovídajícím místě v paměti.

Dále program používá hodnoty polí v několika výpočtech. Tato část programu vypadá se všemi indexy a závorkami poněkud neupraveně. Cyklus `for`, na který přijde řada v kapitole 5 „Cykly a relační výrazy“, nabízí výkonný způsob práce s poli. Pomocí něho se zbavíme nutnosti psát kód pro každý prvek zvlášť. Nicméně prozatím zůstaneme u malých polí.

Vzpomeňte si, že operátor `sizeof` vrací 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 dokazuje, že jméno `yams` zastupuje pole, ale `yams[1]` pouze typ int.

Pravidla inicializace polí

Jazyk C++ má pro inicializace polí několik pravidel. Určují, kdy je můžete provádět a co se stane, pokud počet prvků pole neodpovídá počtu hodnot v inicializátoru. Pojďme se na tato pravidla podívat.

Pole je možné inicializovat pouze v definici. Nelze ji provést později nebo přiřadit jedno celé pole druhému:

```
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 pro přiřazení hodnot jednotlivým prvkům pole můžete vždy použít indexy.

Při inicializaci lze dodat méně hodnot než je prvků pole. Například následující příkaz inicializuje pouze první dva prvky pole `hotelTips`:

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

Při částečné inicializaci překladač nastaví zbývající prvky na nulu. Je tedy snadné inicializovat všechny prvky pole nulovými hodnotami – inicializujete explicitně pouze první prvek pole na nulu a potom necháte překladač inicializovat nulou zbývající prvky.

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

Jestliže jsou při inicializaci pole hranaté závorky ponechány prázdné, překladač C++ sám spočítá prvky pole. Předpokládejme, že jste napsali následující deklaraci:

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

Překladač vytvoří pole `things` se čtyřmi prvky.

Nechte to na překladači

Obvykle není dobrým zvykem nechat určení počtu prvků na překladači, protože jeho výpočet se může lišit od vašeho. Nicméně tento přístup bývá bezpečnější při inicializaci znakového pole řetězcem, což si brzy ukážeme. Pokud vás znepokojuje, že nevíte, jak velké je pole, můžete napsat kód podobný následujícímu:

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

Zda je tento kód užitečný nebo napsaný z lenosti, záleží na okolnostech.

Knihovna standardních šablon v C++ (Standard Template Library, STL) obsahuje alternativu k polím, která se nazývá *třída šablon* `vector` (`vector` *template class*). Tato možnost je sofistikovanější a flexibilnější než vestavěný složený typ. Další podrobnosti o STL a třídě šablon `vector` naleznete v kapitole 16, „Třída `string` a standardní knihovna šablon“.

Řetězce

Řetězec je řada znaků uložená v za sebou jdoucích bajtech paměti. Jazyk C++ používá dva způsoby práce s řetězci. První, převzatý z jazyka C a často označovaný jako *řetězec stylu C*, představuje metodu, kterou se naučíme zde. Dále se v této kapitole zmíníme o alternativní metodě založené na knihovně tříd `string`.

Zatím nám myšlenka řady znaků uložených v za sebou jdoucích bajtech napovídá, že můžeme ukládat řetězce do pole typů `char`, kde má každý znak vlastní prvek pole. Řetězce nabízejí vhodný způsob ukládání textových informací, jako jsou například zprávy pro uživatele (*„Prosím, sdělte mi vaše tajné číslo účtu ve švýcarské bance!“*) nebo jeho odpovědi (*„Děláte si legraci?“*). Řetězce stylu C mají zvláštní vlastnost: Posledním znakem každého řetězce je *nulový znak*. Tento znak, jehož zápis je \0, má hodnotu kódu ASCII 0 a slouží pro označení konce řetězce. Podívejte se na následující dvě deklarace:

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

Obě pole obsahují typy char, ale pouze druhé představuje řetězec. Nulový znak má v řetězcích stylu C důležitou úlohu. Jazyk C++ nabízí mnoho funkcí pro práci s řetězci, včetně těch, které používají objekt `cout`. Všechny zpracovávají řetězce znak po znaku, dokud nenarazí na nulový znak. Požádáte-li objekt `cout`, aby zobrazil řetězec uložený v poli `cat`, vypíše první čtyři znaky, detekuje nulový znak a zastaví se. Avšak budete-li natolik neslušní a řeknete objektu `cout`, aby zobrazil pole `dog`, které není řetězcem, `cout` vytiskne pět znaků pole a potom bude postupovat pamětí dál bajt po bajtu, přičemž vyhodnotí každý bajt jako znak vhodný pro tisk, dokud nedosáhne nulového znaku. Protože se nulové znaky, což jsou ve skutečnosti bajty nastavené na hodnotu nula, v paměti vyskytují poměrně často, škody lze obvykle rychle napravit; přesto byste neměli s neřetězcovými znakovými poli pracovat jako s řetězci.

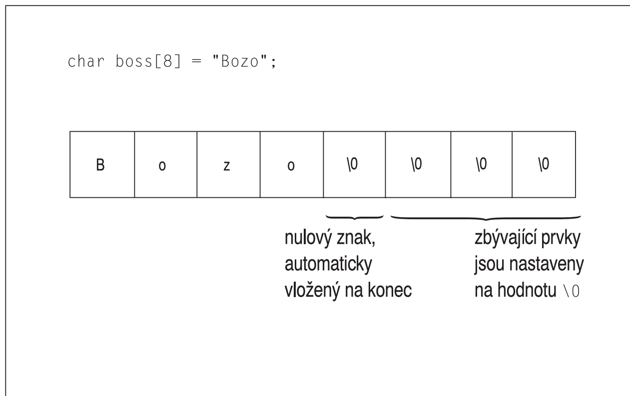
Podle příkladu pole `cat` vypadá inicializace pole řetězcem obtížně – všechny ty jednoduché uvozovky a nezapomenout na nulový znak. Nedělejte si z toho těžkou hlavu. Existuje lepší způsob inicializace znakových polí řetězcem. Stačí jednoduše použít řetězec v uvozovkách, který se nazývá *řetězcová konstanta* resp. *řetězcový literál*, jak je vidět v následujících příkladech:

```
char bird[10] = "Pan Levny";            // \0 se rozumí samo sebou
char fish[] = "Bubliny";                // velikost spočítá překladač
```

Řetězcům v uvozovkách je vždy implicitně doplněn koncový nulový znak, takže ho nemusíte vypisovat. (Viz obrázek 4.2.) Také různé služby jazyka C++ pro načítání řetězce z klávesnicového vstupu do pole typů char automaticky přidávají ukončovací nulový znak. (Jestliže za běhu programu z výpisu 4.1 zjistíte, že musíte použít klíčové slovo `static`, aby byla provedena inicializace pole, jeho použití bude nutné také u těchto polí typů `char`.)

Musíte si být samozřejmě jistí, že je pole dostatečně veliké pro uložení všech znaků řetězce včetně ukončující nuly. Inicializace znakového pole řetězcovou konstantou je jedním z případů, kdy může být bezpečnější nechat určení počtu prvků na překladači. Pokud vytvoříte pole větší než je řetězec, kromě plýtvání pamětí neuděláte žádnou jinou chybu. Je to proto, že se funkce, které pracují s řetězci, neřídí velikostí

pole, ale umístěním nulového znaku. Jazyk C++ žádným způsobem neomezuje délku řetězců.



Obrázek 4.2 Inicializace pole řetězcem

> **ZAPAMATUJTE SI**
>
> **Při určování minimální velikosti pole pro uložení řetězce nezapomeňte na koncový nulový znak.**

Všimněte si, že řetězcová konstanta (dvojité uvozovky) je nezaměnitelná se znakovou konstantou (jednoduché uvozovky). Znaková konstanta, jako je `'S'`, představuje zkrácený zápis kódu znaku. Na systému se znakovou sadou ASCII je `'S'` pouze jiným způsobem zápisu čísla 83. Proto příkaz

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

přiřazuje hodnotu 83 proměnné `shirt_size`. Ale řetězcová konstanta `"S"` reprezentuje řetězec složený ze dvou znaků S a \0. Horší je, že `"S"` ve skutečnosti představuje adresu paměti, na které je řetězec uložen. Takže příkaz

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

se pokouší proměnné `shirt_size` přiřadit paměťovou adresu! Protože má adresa v jazyce C++ samostatný typ, překladač takovéto přiřazení nepřipustí. (K tomuto problému se vrátíme později, až zvládneme ukazatele.)

Spojování řetězců

Občas mohou být řetězce příliš dlouhé a nevejdou se na jeden řádek kódu. C++ umožňuje spojování řetězcových konstant, to znamená sloučení dvou řetězců v uvozovkách do jednoho. Libovolné dvě řetězcové konstanty oddělené pouze bílými znaky (mezery, tabulátory a znaky nového řádku) jsou ve skutečnosti automaticky spojeny do jedné. Všechny následující příkazy výstupu jsou tedy vlastně stejné:

```
cout << "Obetoval bych pravou ruku, kdybych" " mohl byt slavnym
houslistou.\n";
cout << "Obetoval bych pravou ruku, kdybych mohl byt slavnym houslistou.\n";
cout << "Obetoval bych pravou ru"
"ku, kdybych mohl byt slavnym houslistou.\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 nepočítá. Znak `\0` prvního řetězce je nahrazen prvním znakem druhého řetězce.

Použití řetězců v polích

Dva nejběžnější způsoby vložení řetězce do pole jsou inicializace pole řetězcovou konstantou a načtení vstupu do pole z klávesnice nebo ze souboru. Výpis 4.2 názorně ukazuje tyto přístupy prostřednictvím inicializace jednoho pole řetězcem v uvozovkách a umístění vstupního řetězce do druhého pole pomocí objektu `cin`. Program také používá standardní knihovní funkci `strlen()` na zjištění délky řetězce. Standardní hlavičkový soubor `cstring` (nebo `string.h` u starších implementací) dodává potřebné deklarace pro tento příklad i pro mnoho dalších funkcí, které pracují s řetězci.

Výpis 4.2 strings.cpp

```
// strings.cpp -- uklada retezce do pole
#include <iostream>
#include <cstring>                       // pro funkci strlen()
int main()
{
   using namespace std;
   const int Size = 15;
   char name1[Size];                     // prázdné pole
   char name2[Size] = "C++owboy";  // inicializované pole
   // Poznámka: některé implementace mohou vyžadovat
   // pro inicializaci pole name2 klíčové slovo static

   cout << "Ahoj! Jmenuji se " << name2;
   cout << "! Jak se jmenujete vy?\n";
   cin >> name1;
   cout << "Dobre, " << name1 << ", vase jmeno obsahuje ";
   cout << strlen(name1) << " znaku a je ulozeno\n";
   cout << "v poli o velikosti " << sizeof name1 << " bajtu.\n";
   cout << "Pocatecni pismeno vaseho jmena je " << name1[0] << ".\n";
   name2[3] = '\0';                      // nulový znak
   cout << "Toto jsou prvni tri pismena meho jmena: ";
   cout << name2 << endl;
   return 0;
}
```

KOMPATIBILITA

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

Zde je ukázka běhu programu uvedeného na výpisu 4.2:

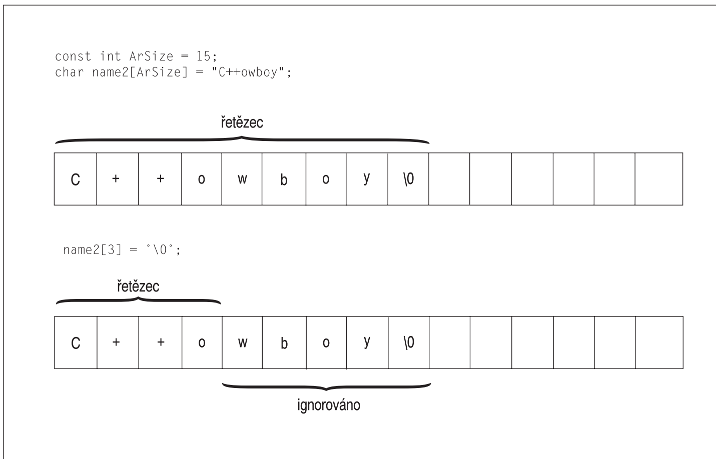
```
Dobry den! Jmenuji se C++owboy! Jak se jmenujete vy?
Basicman
Dobre, Basicman, vase jmeno obsahuje 8 znaku a je ulozeno
v poli o velikosti 15 bajtu.
Pocatecni pismeno vaseho jmena je B.
Toto jsou prvni tri pismena meho jmena: C++
```

Poznámky k programu

Co se můžete z tohoto příkladu naučit? Nejprve si všimněte, že operátor `sizeof` dodává velikost celého pole 15 bajtů, naproti tomu funkce `strlen()` vrací velikost v poli uloženého řetězce a ne velikost vlastního pole. Funkce `strlen()` počítá pouze viditelné znaky bez nulového. Proto vrací pro délku řetězce `Basicman` hodnotu 8 a ne 9. Jestliže je `cosmic` řetězec, zjistíme minimální velikost pole pro uložení tohoto řetězce příkazem `strlen(cosmic) + 1`.

Protože `name1` a `name2` jsou pole, můžeme ke konkrétnímu znaku pole přistupovat pomocí indexu. Náš program například používá pro zjištění prvního znaku pole výraz `name1[0]`. Program také nastavuje `name2[3]` na nulový znak. To způsobí ukončení řetězce po třech znacích, i když v poli ostatní znaky zůstávají. Viz obrázek 4.3.

Všimněte si, že program na výpisu 4.2 používá pro velikost pole symbolickou konstantu. Velikost pole se často vyskytuje v několika příkazech programu. Vyjádření velikosti pole prostřednictvím symbolické konstanty zjednodušuje úpravy programu při změnách velikosti pole; je nutná pouze oprava jedné hodnoty v místě definice konstanty.



Obrázek 4.3 Zkrácení řetězce pomocí znaku \0

Rizika spojená se vstupem řetězce

Program `string.cpp` obsahuje chybu, která se neprojevila díky vhodně vybranému vstupu. Výpis 4.3 tento nedostatek předvádí a ukazuje, že vstup řetězce nemusí být jednoduchý.

Výpis 4.3 instr1.cpp

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

```
    cout << "Zadejte vase jmeno:\n";
    cin >> name;
    cout << "Zadejte vas oblibeny zakusek:\n";
    cin >> dessert;
    cout << "Mam vyborny " << dessert;
    cout << ", ktery si zaslouzi jen " << name << ".\n";
    return 0;
}
```

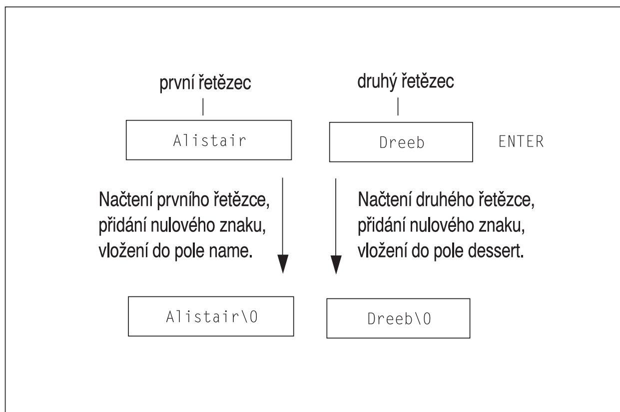
Zadání programu ve výpisu 4.3 je jednoduché. Přečtěte z klávesnice jméno uživatele a oblíbený zákusek a potom tuto informaci vypište. Zde je příklad běhu programu:

```
Zadejte vase jmeno:
Alistair Dreeb
Zadejte vas oblibeny zakusek:
Mam vyborny Dreeb, ktery si zaslouzi jen Alistair.
```

V tomto případě jsme ani neměli možnost zadat oblíbený zákusek! Program sice zobrazil odpovídající dotaz, ale potom okamžitě vypsal poslední řádek.

Problém spočívá v tom, jak objekt `cin` rozeznává ukončení zadávání řetězce. Z klávesnice nelze zadat nulový znak, takže `cin` potřebuje jiný způsob určení konce řetězce. Objekt `cin` proto používá pro vymezení řetězce bílé znaky – mezery, tabulátory a znaky nového řádku. To znamená, že `cin` při získávání vstupních údajů pro znakové pole načte pouze jedno slovo. Po načtení tohoto slova `cin` automaticky přidá během vkládání řetězce do pole ukončovací nulový znak.

V praxi to znamená, že `cin` přečte `Alistair` jako první celý řetězec a umístí ho do pole `name`. Slovo `Dreeb` tedy zůstane ve vstupní frontě. Když `cin` hledá ve vstupní frontě odpověď na otázku o oblíbeném zákusku, nalezne tam `Dreeb`. Potom objekt `cin` spolkne `Dreeb` a vloží ho do pole `dessert`. Viz obrázek 4.4.



Obrázek 4.4 Pohled objektu `cin` na vstupní řetězec

Další problém, který se v ukázkovém běhu programu nevyskytl, spočívá v tom, že by vstupní řetězec mohl být delší, než je cílové pole. Použití objektu `cin` způsobem předvedeným v tomto příkladu nezabrání vložení 30znakového řetězce do 20znakového pole.

Na vstupních řetězcích je závislé množství programů, takže si tuto problematiku probereme podrobněji. Budeme k tomu ale potřebovat některé pokročilejší vlastnosti objektu `cin`, které jsou popsány v kapitole 17 „Vstup, výstup a soubory“.

Vstup řetězce z celého řádku

Často nám přečtení celého jednoho slova nemusí vyhovovat. Představte si, že si program vyžádá jméno města a uživatel zadá `New York` nabo `Sao Paulo`. Program by měl číst a ukládat celá jména, nikoli jen `New` nebo `Sao`. Aby se místo jednotlivých slov četly celé věty, musíme jít na čtení řetězců jinak. Přesně řečeno, místo slovně orientované procedury potřebujeme řádkově orientovanou proceduru. A máme štěstí, neboť třída `istream`, do níž patří například i `cin`, obsahuje další členské funkce: `getline()` a `get()`. Obě čtou celý vstupní řádek až po znak nový řádek. Procedura `getline()` pak znak nový řádek zruší, zatímco `get()` jej ponechá ve vstupní frontě. Podívejme se na obě procedury podrobněji, začneme s `getline()`.

Vstup řádku pomocí getline()

Funkce `getline()` čte celý řádek a pro označení konce vstupu používá znak nového řádku zadaného klávesou Enter. Tuto metodu vyvoláte zápisem volání funkce `cin.getline()`. Funkce `getline()` přebírá dva argumenty. Prvním je jméno pole pro uložení vstupního řádku a druhý argument omezuje počet načtených znaků. Je-li tato hranice například 20, funkce načte nejvíce 19 znaků a ponechá místo pro automatické přidání nulového znaku na konec. Členská funkce `getline()` ukončí načítání vstupu, když dosáhne zadané číselné hranice, nebo přečte znak nového řádku, podle toho, která situace nastane dříve.

Předpokládejme, že chcete pomocí funkce `getline()` načíst jméno do pole name o 20 prvcích. Můžete to udělat pomocí následujícího volání:

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

Tento příkaz načte celý řádek do pole `name` za předpokladu, že řádek obsahuje 19 nebo méně znaků. (Členská funkce `getline()` má také volitelný třetí parametr, který popisuje kapitola 17.)

Výpis 4.4 upravuje výpis 4.3 voláním členské funkce `cin.getline()` namísto použití objektu `cin`. Jinak je program beze změn.

Výpis 4.4 instr2.cpp

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

    cout << "Zadejte vase jmeno:\n";
    cin.getline(name, ArSize); // čte celý řádek
    cout << "Zadejte vas oblibeny zakusek:\n";
    cin.getline(dessert, ArSize);
```

```
    cout << "Mam vyborny " << dessert;
    cout << ", ktery si zaslouzi jen " << name << ".\n";
    return 0;
}
```

KOMPATIBILITA

Některé rané verze jazyka C++ neimplementují všechny vlastnosti současného balíku vstupně/výstupních operací C++. Zvláště členská funkce `getline()` není vždy k dispozici. Pokud máte tyto problémy, příklad si pouze přečtěte a pokračujte dalším, který používá členskou funkci, jež předchází `getline()`. První verze Turbo C++ implementují `getline()` trochu jinak a do řetězce ukládají i znak nového řádku. Microsoft Visual C++ 5.0 a 6.0 má ve funkci `getline()` chybu, která se nachází v hlavičkovém souboru `iostream`, ale ve verzi `ostream.h` není; Service Pack 5 pro Microsoft Visual C++ 6.0 dostupný na adrese `msdn.microsoft.com/vstdio` tuto chybu opravuje.

Zde je několik ukázek výstupu programu z výpisu 4.4:

```
Zadejte vase jmeno:
Dirk Hammernose
Zadejte vas oblibeny zakusek:
redkvickovy dort
Mam vyborny redkvickovy dort, ktery si zaslouzi jen Dirk Hammernose
```

Program nyní načítá celá jména a uživateli vypíše jeho oblíbený zákusek! Funkce `getline()` výhodně načítá celý řádek najednou. Přečte vstup až do znaku nového řádku, který označuje jeho konec. Znak nového řádku nahradí při ukládání nulovým znakem. Viz obrázek 4.5.

Kód:

```
char name[10];
cout << "Zadejte vase jmeno: ";
cin.getline(name, 10);
```

Uživatel odpoví vypsáním jména **Jud** a stlačením klávesy ENTER

```
Zadejte vase jmeno: Jud
```
ENTER

Funkce cin.getline() načte řetězec Jud i se znakem nového řádku vygenerovaným klávesou Enter a tento znak nahradí nulovým znakem.

J	u	d	\0						

nový řádek tento znak nahradí nulovým znakem.

Obrázek 4.5 Funkce `getline()` čte a nahrazuje znak nového řádku

Vstup řádku pomocí get()

Zkusme to teď jinak. Třída `istream` obsahuje další členskou funkci `get()`, která má několik variant. Jedna pracuje téměř podobně jako `getline()`. Přebírá stejné argumenty, interpretuje je stejným způsobem a čte do konce řádku. Ale místo načtení a zahození znaku nového řádku ho nechá ve vstupní frontě. Předpokládejme, že jsme použili dvě volání funkce `get()` za sebou:

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

Protože první volání nechá znak nového řádku ve vstupní frontě, tento znak je prvním znakem, který vidí druhé volání. Funkce `get()` si tedy myslí, že dosáhla konce řádku aniž by nalezla něco ke čtení. Bez pomoci se funkce `get()` přes tento znak nového řádku nedostane.

Naštěstí si můžeme pomoci variantou funkce `get()`. Volání `cin.get()` (bez argumentů) načte jeden následující znak, i když je to znak nového řádku. Tuto funkci tedy můžeme použít pro odstranění znaku nového řádku a přípravu na další řádek vstupu. Následující řada příkazů již bude fungovat správně:

```
cin.get(name, ArSize);             // čte první řádek
cin.get();                         // čte znak nového řádku
cin.get(dessert, Arsize);          // čte druhý řádek
```

Dalším možným způsobem použití funkce `get()` je *zřetězení* neboli spojení dvou členských funkcí třídy:

```
cin.get(name, ArSize).get();       // Zřetězení členských funkcí
```

Tento zápis umožňuje skutečnost, že funkce `cin.get(name, ArSize)` vrací objekt `cin`, jenž je použit jako objekt, který vyvolá funkci `get()`. Obdobně příkaz

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

načítá dva za sebou jdoucí vstupní řádky do polí name1 a name2; tento zápis provede stejnou činnost jako dvě po sobě jdoucí samostatná volání funkce `cin.getline()`.

Výpis 4.5 používá zřetězení. V kapitole 11 „Práce s třídami" se naučíte, jak můžete tuto vlastnost vložit i do vašich definic tříd.

Výpis 4.5 instr3.cpp

```
// instr3.cpp -- čtení více než jednoho slova pomocí get() & get()
#include <iostream>
int main()
{
   using namespace std;
   const int ArSize = 20;
   char name[ArSize];
   char dessert[ArSize];

   cout << "Zadejte vase jmeno:\n";
   cin.get(name, ArSize).get();    // čte řetězec, nový řádek
   cout << "Zadejte vas oblibeny zakusek:\n";
   cin.get(dessert, ArSize).get();
   cout << "Mam vyborny " << dessert;
   cout << ", ktery si zaslouzi jen " << name << ".\n";
   return 0;
}
```

KOMPATIBILITA

Některé starší verze C++ neimplementují variantu get() bez argumentů. Nicméně implementují jinou variantu `get()`, která přijímá jeden argument typu `char`. Pokud ji použijete místo varianty bez argumentů, musíte nejprve deklarovat proměnnou typu `char`:

```
char ch;
cin.get(name, ArSize).get(ch);
```

Tento kód můžete použít místo kódu z výpisu 4.5. Kapitoly 6 a 17 se zabývají dalšími variantami funkce `get()`.

Tady je příklad běhu programu z výpisu 4.5:

```
Zadejte vase jmeno:
Mai Parfait
Zadejte vas oblibeny zakusek:
Cokoladova pena
Mam vyborny Cokoladova pena, ktery si zaslouzi jen Mai Parfait.
```

Všimněte si, že jazyk C++ povoluje vícenásobné verze funkcí, pokud mají různý seznam argumentů. Jestliže například zavoláte funkci `cin.get(name, ArSize)`, překladač si všimne, že jste použili tvar, který vkládá řetězec do pole a vyvolává vhodnou členskou funkci. Když zavoláte funkci `cin.get()`, překladač zjistí, že chcete tvar, který čte jeden znak. Kapitola 8 bude tuto vlastnost, která se jmenuje *přetěžování funkcí*, zkoumat.

Proč bychom měli používat funkci `get()` místo `getline()`? Zaprvé, ve starších implementacích funkce `getline()` nemusí být. Zadruhé, funkce `get()` má více možností. Předpokládejme, že načítáte pomocí funkce `get()` řádek do pole. Jak zjistíte, zda přečetla celý řádek a nezastavila se dříve z důvodů naplněnosti pole? Podíváte se na další vstupní znak. Je-li tam znak nového řádku, pak byl načten celý řádek. Pokud tam není znak nového řádku, pak jsou na řádku ještě další vstupní data. Kapitola 16 tuto techniku probírá podrobněji. Stručně řečeno, funkce `getline()` je trochu jednodušší na použití, ale `get()` zjednodušuje ošetření chyb. Pro načtení řádku vstupu můžete použít kteroukoliv z těchto dvou funkcí; pouze musíte myslet na jejich trochu odlišné chování.

Prázdné řádky a další problémy

Co se stane, když funkce `getline()` nebo `get()` přečte prázdný řádek? Původní postup byl takový, že následující příkaz vstupu začal tam, kde poslední funkce `getline()` nebo `get()` skončila. Avšak v současnosti je po přečtení prázdného řádku funkcí `get()` (ale ne `getline()`) nastaven takzvaný *chybový bit* (*failbit*). Potom je další vstup zablokován, ale můžete ho odblokovat následujícím příkazem:

```
cin.clear();
```

Další problém nastává, když je vstupní řetězec delší než alokovaný prostor. Je-li vstupní řádek delší než počet stanovených znaků, obě funkce `getline()` i `get()` nechávají zbylé znaky ve vstupní frontě. Funkce `getline()` navíc nastaví chybový bit a vypne další vstup.

Kapitoly 5, 6 a 16 tyto vlastnosti zkoumají podrobněji a zabývají se jejich využitím v programech.

Smíchání řetězcového a číselného vstupu

Smíchání číselného vstupu s řádkově orientovaným vstupem řetězce může způsobit problémy. Podívejte se na jednoduchý program z výpisu 4.6.

Výpis 4.6 numstr.cpp

```
// numstr.cpp -- řádkový vstup následující vstup čísla
#include <iostream>
int main()
{
   using namespace std;
   cout << "V kterem roce byl vas dum postaven?\n";
   int year;
   cin >> year;
   cout << "Jaka je jeho adresa?\n";
   char address[80];
   cin.getline(address, 80);
   cout << "Rok vystavby: " << year << endl;
   cout << "Adresa: " << address << endl;
   cout << "Konec!\n";
   return 0;
}
```

Spuštěný program z výpisu 4.6 by vypadal asi takto:

```
V kterem roce byl vas dum postaven?
1966
Jaka je jeho adresa?
Rok vystavby: 1966
Adresa:
Konec!
```

Nikdy nedostanete příležitost zadat adresu. Tento problém je způsoben tím, že když `cin` načte rok, 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 zahození znaku nového řádku před čtením adresy. Může to být provedeno několika způsoby, včetně použití `get()` bez argumentu nebo s argumentem typu `char`, jak je popsáno 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` vrací objekt `cin`:

```
(cin >> year).get();    // nebo (cin >> year).get(ch);
```

Pokud provedete některou z těchto oprav ve výpisu 4.6, program bude pracovat správně:

```
V kterem roce byl vas dum postaven?
1966
Jaka je jeho adresa?
43821 Unsigned Short Street
Rok vystavby: 1966
Adresa: 43821 Unsigned Short Street
Konec!
```

Programy jazyka C++ často používají pro práci s řetězci ukazatele místo polí. Tuto vlastnost řetězců si probereme, až se o ukazatelích něco málo naučíme. Mezitím se podíváme na modernější způsob zpracování řetězců: třída `string`.

Úvod do třídy string

Standardem ISO/ANSI byla knihovna C++ rozšířena o třídu `string`. Nyní tedy místo do znakového pole můžeme ukládat řetězec do proměnné typu `string` (resp. do objektu, abychom použili terminologii C++). Jak uvidíme, třída string se používá snadněji než pole a také reprezentace řetězce jako typu je věrohodnější.

Má-li program využívat třídy `string`, musí obsahovat hlavičkový soubor `string`. Třída `string` je součástí jmenného prostoru `std`, takže je nutno zadat direktivu `using`, nebo se na třídu odkázat pomocí `std::string`. Definicí třídy se u řetězce skryje charakter pole a pracuje se s ním jako s běžnou proměnnou. Ve výpisu 4.7 jsou znázorněny podobnosti a rozdíly mezi řetězcovými objekty a znakovými poli.

Výpis 4.7 strtype1.cpp

```
// strtype1.cpp -- using the C++ string class
#include <iostream>
#include <string>                        // make string class available
int main()
{
    using namespace std;
    char charr1[20];                     // vytvoř prázdné pole
    char charr2[20] = "jaguar";          // vytvoř a inicializuj pole
    string str1;                         // vytvoř prázdný řetězcový objekt
    string str2 = "panter";              // vytvoř a inicializuj řetězec

    cout << "Zadej kockovitou selmu: ";
    cin >> charr1;
    cout << "Zadej jinou kockovitou selmu: ";
    cin >> str1; // use cin for input
    cout << "Zde jsou kockovite selmy:\n";
    cout << charr1 << " " << charr2 << " "
         << str1 << " " << str2          // výstup prostřednictvím cout
         << endl;
    cout << "Treti pismeno v " << charr2 << " je "
         << charr2[2] << endl;
    cout << "Treti pismeno v " << str2 << " je "
         << str2[2] << endl;             // zápis typu pole

    return 0;
}
```

Příklad výstupu programu z výpisu 4.7:

```
Zadej kockovitou selmu: ocelot
Zadej jinou kockovitou selmu: tygr
Zde jsou kockovite selmy:
ocelot jaguar tygr panter
Treti pismeno v jaguar je g
Treti pismeno v panter je n
```

Z tohoto příkladu by mělo být jasné, že řetězcový objekt lze v mnoha ohledech používat stejně jako znakové pole:

- Řetězcový objekt lze inicializovat jako řetězec v C.
- Vstup z klávesnice lze do řetězcového objektu ukládat pomocí `cin`.
- Řetězcový objekt lze vypisovat pomocí `cout`.
- Na jednotlivé znaky uložené v řetězcovém objektu se lze odkazovat stejným zápisem jako na prvky v poli.

Nejdůležitějším rozdílem mezi řetězcovými objekty a znakovými poli ve výpisu 4.7 je skutečnost, že řetězcový objekt se deklaruje jako jednoduchá proměnná, nikoli jako pole.

```
string str1;                // vytvoř prázdný objekt typu string
string str2 = "panther";    // vytvoř řetězec s počátečním nastavením
```

Třída je vytvořena tak, že program může pracovat s velikostí automaticky. Například při deklaraci `str1` se vytvoří řetězcový objekt délky nula, avšak přičtení dat do `str1` program automaticky změní velikost tohoto objektu:

```
cin >> str1;       // velikost str1 se změní podle vstupu
```

Práce s řetězcovými objekty je pohodlnější a bezpečnější než práce s poli. Pojmově si lze představit pole znaků jako množinu znakových jednotek, do nichž se ukládá řetězec, zatímco proměnná třídy `string` je samostatná entita, která reprezentuje řetězec.

Přiřazování, zřetězování a připojování

S třídou `string` jsou některé operace jednodušší než v případě polí. Například není možné jednoduše přiřadit jedno pole druhému, zatímco v případě řetězcového objektu to možné je:

```
char charr1[20];                // vytvoř prázdné pole
char charr2[20] = "jaguar";     // vytvoř inicializované pole
string str1;                    // vytvoř prázdný řetězcový objekt
string str2 = "panter";         // vytvoř inicializovaný řetězec
charr1 = charr2;                // NEPLATNÉ, pole nelze přiřazovat
str1 = str2;                    // PLATNÉ, přiřazení objektu je ok
```

Třída string zjednodušuje kombinování řetězců. Pomocí operátoru + spojíme dva řetězcové objekty do jednoho, operátorem += „přišpendlíme" řetězec ke konci existujícího řetězcového objektu. Budeme-li pokračovat v předchozím kódu, nabízejí se nám tyto možnosti:

```
string str3;
str3 = str1 + str2;             // proměnné str3 přiřaď spojené řetězce
str1 += str2;                   // přidej str2 na konec str1
```

Ve výpisu 4.8 je uveden příklad použití. Všimněme si, že můžeme připojovat jak klasické řetězce používané v C, tak i objekty typu `string`.

Výpis 4.8 strtype2.cpp

```
// strtype2.cpp -- prirazovani, pridavani a pripojovani
#include <iostream>
#include <string>               // zpřístupnění třídy string
int main()
{
   using namespace std;
```

```
    string s1 = "tucnak";
    string s2, s3;

    cout << "Retezcove objekty muzete prirazovat: s2 = s1\n";
    s2 = s1;
    cout << "s1: " << s1 << ", s2: " << s2 << endl;
    cout << "Retezce v C muzete prirazovat retezcovym objektum.\n";
    cout << "s2 = \"tucnak\"\n";
    s2 = "tucnak";
    cout << "s2: " << s2 << endl;
    cout << "Promenne typu string muzete zretezovat: s3 = s1 + s2\n";
    s3 = s1 + s2;
    cout << "s3: " << s3 << endl;
    cout << "Retezce muzete pripojovat.\n";
    s1 += s2;
    cout <<"Vysledkem s1 += s2 je s1 = " << s1 << endl;
    s2 += " na den";
    cout <<"Vysledkem s2 += \" na den\" je s2 = " << s2 << endl;

    return 0;
}
```

Připomeňme si, že escape sekvence `\"` představuje dvojité uvozovky použité jako literál, nikoli tedy hranice řetězce. Zde je výstup programu z výpisu 4.8.

```
Retezcove objekty muzete prirazovat: s2 = s1
s1: tucnak, s2: tucnak
Retezce v C muzete prirazovat retezcovym objektum.
s2 = "krkavec"
s2: krkavec
Promenne typu string muzete zretezovat: s3 = s1 + s2
s3: tucnakkrkavec
Retezce muzete pripojovat.
Vysledkem s1 += s2 je s1 = tucnakkrkavec
Vysledkem s2 += " na den" je s2 = krkavec na den
```

Další operace s třídou string

Programátoři potřebovali přiřazovat řetězce už i dřív, než byla do C++ zavedena třída `string`. Používali k tomu funkce z knihovny C, které podporuje hlavičkový soubor `cstring` (dříve `string.h`). Například ke zkopírování řetězce do znakového pole můžete použít funkci `strcpy()`, k připojení řetězce ke znakovému poli zase `strcat()`:

```
strcpy(charr1, charr2);     // kopíruj charr2 do charr1
strcat(charr1, charr2);     // připoj obsah charr2 k char1
```

Ve výpisu 4.9 můžete porovnat způsob zpracování řetězcových objektů a znakových polí.

Výpis 4.9 strtype3.cpp

```
// strtype3.cpp -- dalsi vlastnosti tridy string
#include <iostream>
#include <string>           // zpřístupnění třídy string
#include <cstring>          // knihovna C pro zpracování řetězců
int main()
{
```

```
    using namespace std;
    char charr1[20];
    char charr2[20] = "jaguarovy";
    string str1;
    string str2 = "panterove";

    // prirazovani retezcovym objektum a znakovym polim
    str1 = str2;                // zkopíruj str2 do str1
    strcpy(charr1, charr2);     // zkopíruj charr2 do charr1

    // pripojovani k retezcovym objektum a ke znakovym polim
    str1 += " testo";           // ke konci str1 pripoj testo
    strcat(charr1, " dzus");    // ke konci charr1 pripoj dzus

    // zjistovani delky retezcoveho objektu a klasickeho retezce
    int len1 = str1.size();     // delka str1
    int len2 = strlen(charr1);  // delka charr1

    cout << "Retezec " << str1 << " obsahuje "
         << len1 << " znaku.\n";
    cout << "Retezec " << charr1 << " obsahuje "
         << len2 << " znaku.\n";

    return 0;
}
```

Zde je výstup z programu na výpisu 4.9:

```
Retezec panterove testo obsahuje 15 znaku.
Retezec jaguarovy dzus obsahuje 14 znaku.
```

Práce s řetězcovými objekty je jednodušší než s řetězcovými funkcemi v C. To platí zejména u složitějších operací. Například knihovní ekvivalent

```
str3 = str1 + str2;
```

je:

```
strcpy(charr3, charr1);
strcat(charr3, charr2);
```

Navíc, u polí vždy hrozí nebezpečí, že cílové pole bude menší než objem ukládaných dat, například zde:

```
char site[10] = "dum";
strcat(site, " plny livancu");  // problem s pameti
```

Funkce `strcat()` se pokusí zkopírovat všech 12 znaků do prostoru pole, čímž přepíše přilehlou paměť. To může způsobit buď havárii programu, anebo může program sice pokračovat ve výpočtu, avšak s nesprávnými daty. U třídy `string` takový problém nemůže nastat, neboť délka proměnné se mění automaticky.

Knihovna C také obsahuje programy `strncat()` a `strncpy()`, které jsou bezpečnější než `strcat()` a `strcpy()`. Oproti svým příbuzným mají jeden argument navíc, který indikuje maximální povolenou délku cílového pole, avšak program je tak zbytečně složitější.

Syntaxe při zjišťování počtu znaků v řetězci je také odlišná:

```
int len1 = str1.size();       // délka str1
int len2 = strlen(charr1);    // délka charr1
```

Funkce `strlen()` je běžná funkce s argumentem, kterým je řetězec v jazyce C. Funkce vrací počet znaků v tomto řetězci. Funkce `size()` v zásadě dělá totéž, avšak má odlišnou syntaxi. Jméno proměnné `str1` zde není argumentem, nýbrž je uvedeno před jménem funkce a je k němu připojeno tečkou. Jak bylo uvedeno u metody `put()` v kapitole 3 „Práce s daty", syntaxe říká, že `str1` je objekt a `size()` je metoda třídy. Metoda je funkce, která může být vyvolána pouze objektem, který patří do téže třídy jako tato metoda. V tomto konkrétním případě je `str1` řetězcový objekt a `size()` je řetězcová metoda. Stručně řečeno, funkce C využívají k identifikaci řetězce argument a objekty řetězcové třídy v C++ využívají k identifikaci řetězce jméno objektu a operátor tečku.

Vstupní a výstupní operace třídy string

Už shora jsme uvedli, že řetězcový objekt lze číst pomocí cin s operátorem >> a vypisovat objekt pomocí cout s operátorem >> , přičemž syntaxe je stejná jako v případě řetězců v C. Při čtení celého řádku se však syntaxe liší, rozdíl je znázorněn v programu na výpisu 4.10.

Výpis 4.10 strtype4.cpp

```
// strtype4.cpp -- radkovy vstup
#include <iostream>
#include <string>              // zpřístupnění třídy string
#include <cstring>             // knihovna řetězců v C
int main()
{
   using namespace std;
   char charr[20];
   string str;

   cout << "Delka retezce v charr pred vstupem: "
         << strlen(charr) << endl;
   cout << "Delka retezce v str pred vstupem: "
         << str.size() << endl;
   cout << "Zadejte textovy radek:\n";
   cin.getline(charr, 20);     // indikace maximální délky
   cout << "Zadali jste: " << charr << endl;
   cout << "Zadejte dalsi radek:\n";
   getline(cin, str);          // cin je nyní argumentem; délka není
                               // uvedena
   cout << "Zadali jste: " << str << endl;
   cout << "Delka retezce v charr po vstupu je: "
         << strlen(charr) << endl;
   cout << "Delka retezce v str po vstupu je: "
         << str.size() << endl;

   return 0;
}
```

Příklad spuštění programu uvedeného ve výpisu 4.10:

```
Delka retezce v charr pred vstupem: 27
Delka retezce v str pred vstupem: 0
Zadejte textovy radek:
oriskove maslo
```

```
Zadali jste: oriskove maslo
Zadejte dalsi radek:
brusinkovy dzem
Zadali jste: brusinkovy dzem
Delka retezce v charr po vstupu je: 14
Delka retezce v str po vstupu je: 15
```

Všimněte si, že před vstupem je zadána délka řetězce v poli `charr` 27, což je víc než velikost pole! To má dva důvody. První je ten, že obsah neinicializovaného pole je nedefinovaný. Druhým důvodem je skutečnost, že funkce `strlen()` počítá prvky od prvního po bajtech, dokud nenarazí na nulový znak. V tomto případě bude první nulový znak až několik bajtů za koncem pole. Kde se v neinicializovaných datech nacházejí nulové znaky, závisí na náhodě, takže by se snadno mohlo stát, že by tento program poskytl jiný výsledek.

Také si všimněte, že délka řetězce ve `str` před vstupem je 0. To proto, že délka neinicializovaného řetězcového objektu je automaticky nastavena na 0.

Kód, který přečte řádek do pole, vypadá takto:

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

Notace s tečkou říká, že funkce `getlin()` je metoda třídy `istream`. (Vzpomeňte si, že `cin` je objekt `istream`.) Jak už jsme se zmínili dříve, první argument označuje cílové pole a druhý udává jeho velikost. Funkce `getlin()` musí znát velikosti pole kvůli ochraně paměti proti přepsání.

Kód, který přečte řádek do řetězcového objektu, vypadá takto:

```
getline(cin,str);
```

Není použita notace s tečkou, což znamená, že *tato* funkce `getline()` *není* metodou třídy. Považuje tedy `cin` za argument, který označuje vstup. Neexistuje argument pro velikost řetězce, neboť ta se automaticky mění tak, aby odpovídala řetězci.

Proč tedy jedna funkce `getline()` je metodou třídy `istream` a druhá není? Třída `istream` byla součástí C++ dávno před zavedením řetězcových tříd. Zná tedy základní typy C++ jako `double` nebo `int`, avšak typ `string` nezná. Proto existují ve třídě istream metody pro zpracování double, int a dalších základních typů, avšak neexistují zde metody pro zpracování řetězcových objektů.

Vzhledem k tomu, že v třídě `istream` neexistují metody na zpracování řetězcových objektů, může se zdát s podivem, že funguje kód:

```
cin >> str;           // přečti slovo do řetězcového objektu str
```

Ukazuje se, že program:

```
cin >> x;             // přečti slovo do základního typu C++
```

využívá (v přestrojení) členskou funkci třídy `istream`. Avšak řetězcový ekvivalent třídy využívá přátelskou funkci (také v přestrojení) pro třídu `string`. Budete si muset počkat do kapitoly 11, kde se dozvíte, co je to přátelská funkce a jak pracuje. Do té doby můžete využívat funkce `cin` a `cout` s řetězcovými objekty a nestarat se o to, proč fungují.

Úvod do struktur

Předpokládejme, že chcete ukládat informace o hráči košíkové. Možná budete chtít uložit jeho nebo její jméno, plat, výšku, váhu, průměrný počet košů, procento volných hodů, nahrávky a tak dále. Určitě by se vám líbil jistý druh datové formy, která by mohla všechny potřebné informace ukládat na jednom místě. Pole se pro to nehodí. Ačkoli může obsahovat několik položek, každá položka musí být stejného typu. To znamená, že jedno pole může obsahovat 20 celých čísel a jiné 10 čísel s pohyblivou desetinnou čárkou, ale jediné pole nemůže v některých prvcích ukládat celá čísla a v jiných čísla s pohyblivou desetinnou čárkou.

Odpovědí na váš požadavek (uložení informací o hráči košíkové) je *struktura* jazyka C++. Struktura představuje univerzálnější datovou formu než pole, protože jedna struktura může obsahovat položky více než jednoho datového typu. Umožňuje sjednotit datovou reprezentaci uložením všech informací souvisejících s košíkovou do jediné strukturní proměnné. Chcete-li mít přehled o celém týmu, můžete použít pole struktur. Struktura je také odrazovým můstkem pro základ OOP v jazyce C++, který představuje třída. Podrobnější studium struktur nás přivede blíže k podstatě OOP v jazyce C++.

Struktura je uživatelsky definovaným typem, jehož datové vlastnosti popisuje deklarace struktury. Po definování typu můžete vytvářet jeho proměnné. Proces vytváření struktury se tedy skládá ze dvou částí. Nejprve definujete popis struktury. Ten popisuje a označuje různé typy dat, které mohou být ve struktuře uloženy. Potom vytvoříte proměnné struktury nebo obecněji, datové objekty struktury odpovídající danému popisu.

Předpokládejme, že společnost Bloataire, Inc. chce vytvořit typ, který by popisoval parametry jejich výrobků řady značkového nafukovacího zboží. Tento typ by měl obsahovat jméno výrobku, jeho objem v krychlových stopách a prodejní cenu. Zde je popis struktury, která odpovídá zadaným požadavkům:

```
struct inflatable           // popis struktury
{
   char name[20];
   float volume;
   double price;
};
```

Klíčové slovo `struct` určuje, že daný kód definuje rozvržení struktury. Identifikátor `inflatable` představuje jméno nebo značku (*tag*) tohoto tvaru; jméno `inflatable` tedy označuje nový typ. Nyní můžete vytvářet proměnné typu `inflatable` stejně jako proměnné typu `char` nebo `int`. Mezi složenými závorkami se nachází seznam datových typů obsažených ve struktuře. Zde může být uveden libovolný z typů jazyka C++ včetně polí nebo jiných struktur. Výše uvedený příklad používá pole typů `char` vhodné pro uložení řetězce, typ `float` a typ `double`. Každá položka tohoto seznamu je členem struktury, takže struktura `inflatable` má tři členy. (Viz obrázek 4.6.)

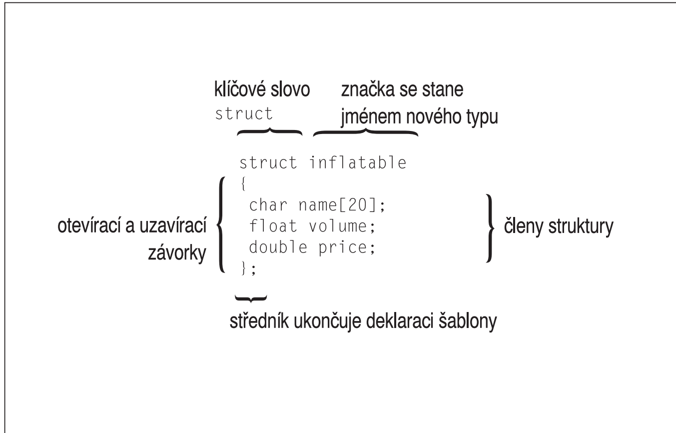
Pokud máte šablonu, můžete vytvářet proměnné odpovídajícího typu:

```
inflatable hat;             // hat je strukturní proměnná typu inflatable
inflatable woopie_cushion;  // proměnná typu inflatable
inflatable mainframe;       // proměnná typu inflatable
```

Znáte-li struktury v C, určitě si všimnete (pravděpodobně s potěšením), že C++ dovoluje vynechat klíčové slovo `struct` při deklaraci strukturní proměnné:

```
struct inflatable goose; // klíčové slovo struct je vyžadováno v C
inflatable vincent;      // klíčové slovo struct není vyžadováno v C++
```

V jazyce C++ je značka struktury používána stejně jako jméno základního typu. Tato změna zdůrazňuje, že deklarace struktury definuje nový typ. Vynechání klíčového slova `struct` je také vyjmuto ze seznamu chybových hlášení.



Obrázek 4.6 Části popisu struktury

Pokud je proměnná `hat` typu `inflatable`, zajišťuje přístup k jednotlivým členům operátor příslušnosti (`.`). Například výraz `hat.volume` odkazuje na člen `volume` odpovídající struktury a `hat.price` na člen `price`. Obdobně představuje `vincent.price` člen `price` proměnné `vincent`. Stručně řečeno, jména členů umožňují přístup ke členům struktury podobně jako indexy k prvkům pole. Protože člen `price` je deklarován jako typ `double`, `hat.price` a `vincent.price` jsou proměnné typu `double` a mohou být použity stejným způsobem jako obvyklé proměnné typu `double`. Proměnná `hat` představuje strukturu, ale `hat.price` je proměnná typu `double`. Mimochodem, metoda použitá pro přístup ke členským funkcím třídy, jako je například `cin.getline()`, má svůj původ v metodě přístupu k členským proměnným struktury, jako `je vincent.price`.

Použití struktury v programu

Nyní, když už jsme si řekli něco o nejdůležitějších vlastnostech struktur, mohli bychom si na příkladech ukázat, jak se využívají v programech. Výpis 4.7 ukazuje výše popsané vlastnosti struktury. Dále předvádí její inicializaci.

Výpis 4.11 structur.cpp

```
// structur.cpp -- jednoduchá struktura
#include <iostream>
struct inflatable        // šablona struktury
{
   char name[20];
   float volume;
   double price;
};
```

```
int main()
{
   using namespace std;
   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 << "Rozsirte vas seznam hostu o " << guest.name;
   cout << " a " << pal.name << "!\n";
// zápis pal.name znamená, že name je členem proměnné pal
   cout << "Oba ziskate za ";
   cout << (guest.price + pal.price) << " Kc!\n";
   return 0;
}
```

KOMPATIBILITA

Stejně jako některé starší verze jazyka C++ neimplementovaly možnost inicializace běžného pole definovaného ve funkci, také neimplementovaly inicializaci běžné struktury ve funkci. I nyní je řešením použití klíčového slova `static` v deklaraci.

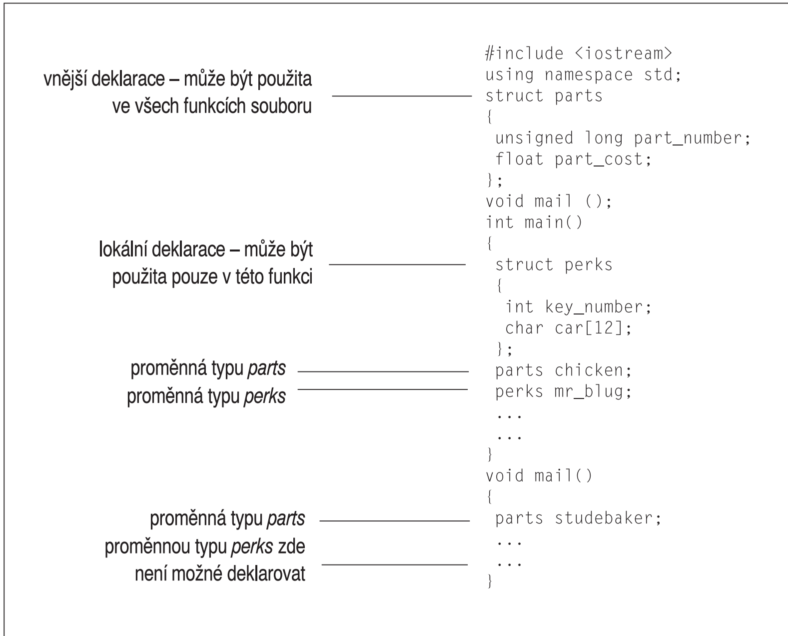
Zde je výstup programu na výpisu 4.11:

```
Rozsirte vas seznam hostu o Glorious Gloria a Audacious Arthur!
Oba ziskate za 62.98 Kc!
```

Poznámky k programu

Důležitou otázkou je umístění deklarace struktury. Pro soubor `structur.cpp` existují dvě možnosti. Deklaraci lze vložit do funkce `main()` hned za otevírací složenou závorku. Druhou zde realizovanou možností je umístit ji mimo a před funkci `main()`. Nachází-li se deklarace vně funkce, říkáme jí *vnější deklarace*. V tomto programu není mezi uvedenými možnostmi žádný praktický rozdíl. Ale u programů, které se skládají ze dvou nebo více funkcí, může být tento rozdíl důležitý. Vnější deklarace mohou využívat všechny následující funkce, zatímco interní deklarace je platná pouze ve funkci, ve které se nachází. Nejčastěji potřebujete vnější deklaraci struktury proto, aby všechny funkce mohly používat strukturu tohoto typu. (Viz obrázek 4.7.)

Proměnné mohou být také deklarovány jako vnitřní nebo vnější a vnější proměnné jsou sdílené mezi funkcemi. (V kapitole 9 se na toto téma podíváme blíže.) Praktické zkuše-



Obrázek 4.7 Lokální a vnější deklarace struktury

nosti z jazyka C++ nedoporučují používání vnějších proměnných, ale podporují vnější deklarace struktur. Často bývá také rozumné deklarovat symbolické konstanty jako vnější.

Dále si všimněte inicializační procedury:

```
inflatable guest =
{
"Glorious Gloria",    // hodnota name
1.88,                 // hodnota volume
29.99                 // hodnota price
};
```

Stejně tak jako u polí je seznam hodnot oddělený čárkami uzavřený do páru složených závorek. V příkladu je každá hodnota na samostatném řádku, ale mohou být i na jednom. Pouze nesmíte zapomenout oddělovat položky čárkami:

```
inflatable duck = {"Daphne", 0.12, 9.98};
```

Každý člen struktury může být inicializován odpovídajícím druhem údaje. Například člen `name` je znakové pole, takže ho můžete inicializovat řetězcem.

Každý člen struktury je zpracováván jako proměnná odpovídajícího typu. Výraz `pal.price` tedy představuje proměnnou typu `double` a `pal.name` pole typů `char`. Pokud program zobrazí `pal.name` pomocí objektu `cout`, je vypsán řetězec. Protože `pal.name` představuje znakové pole, můžete pro přístup k jeho jednotlivým znakům použít indexy. Například `pal.name[ 0]` je znak A. Avšak výraz `pal[0]` nemá význam, protože `pal` je strukturou, nikoli polem.

Může používat struktura člena třídy string?

Můžeme použít objekt třídy `string` místo znakového pole jako pojmenovaného člena struktury? Tj. můžeme deklarovat takovouto strukturu?

```
#include <string>
struct inflatable // structure template
{
   std::string name;
   float volume;
   double price;
};
```

V principu zní odpověď ano. V praxi odpověď závisí na tom, jaký máme překladač, neboť některé (např. Borland C++ 5.5 nebo Microsoft Visual C++ před verzí 7.1) inicializaci struktur členy třídy `string` nepodporují.

Pokud váš překladač takovéto použití třídy nepodporuje, přesvědčte se, zda definice struktur mají přístup ke jmennému prostoru `std`. Direktiva `using` by měla být nad definicí struktury. Alternativně, jako v předchozím příkladu, lze deklarovat `name`, jako by mělo typ `std::string`.

Další vlastnosti struktur

Jazyk C++ vytváří uživatelsky definované typy tak, aby byly co nejvíce podobné vestavěným typům. Struktury můžete například předávat funkcím jako argumenty a také můžete napsat funkci, která vrací strukturu jako návratovou hodnotu. Přiřazovacím operátorem (=) můžete dokonce přiřadit jednu strukturu druhé stejného typu. Tak nastavíte všem členům jedné struktury hodnoty odpovídajících členů druhé struktury, dokonce i když jsou těmito členy pole. Takovéto přiřazení se nazývá *členským přiřazením*. Předávání a vracení struktur odložíme do doby, kdy budeme probírat funkce v kapitole 7 „Funkce – programové moduly jazyka C++“, ale letmo se můžeme na přiřazení struktur podívat již nyní. Výpis 4.12 uvádí příklad.

Výpis 4.12 assgn_st.cpp

```
// assgn_st.cpp -- přiřazení struktur
#include <iostream>
struct inflatable
{
   char name[20];
   float volume;
   double price;
};
int main()
{
   using namespace std;
   inflatable bouquet =
   {
      "slunecnice",
      0.20,
      12.49
   };
   inflatable choice;
```

```
        cout << "kytice: " << bouquet.name << " za ";
        cout << bouquet.price << " Kc" << endl;

        choice = bouquet;     // přiřadí jednu strukturu druhé
        cout << "vyber: " << choice.name << " za ";
        cout << choice.price << " Kc" << endl;
        return 0;
    }
```

Zde je výstup z programu uvedeného ve výpisu 4.12:

```
kytice: slunecnice za 12.49 Kc
vyber: slunecnice za 12.49 Kc
```

Vidíte sami, že *členské přiřazení* funguje, protože členům struktury choice jsou přiřazeny hodnoty uložené ve struktuře bouquet.

Definici tvaru struktury a vytvoření strukturní proměnné můžete spojit do jednoho příkazu. Stačí, když za uzavírací složené závorky uvedete jméno nebo jména proměnných:

```
struct perks
{
   int key_number;
   char car[12];
} mr_smith, ms_jones;    // dvě proměnné typu perks
```

Tímto způsobem můžete proměnné také 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 oddělením definice struktury od deklarací proměnných se obvykle program stane snáze čitelným a pochopitelným.

Dále můžete vytvářet struktury bez jména typu. Uděláte to vynecháním jména značky v definici tvaru struktury a proměnné:

```
struct                    // bez značky
{
   int x;                 // 2 členy
   int y;
} position;               // strukturní proměnná
```

Takto je vytvořena jediná strukturní proměnná position. K jejím členům můžete přistupovat pomocí operátoru příslušnosti, jako například v position.x, ale žádné obecné jméno tohoto typu neexistuje. Později nemůžete vytvářet další proměnné stejného typu. Naše kniha nebude tento omezený tvar struktury používat.

Kromě skutečnosti, že program jazyka C++ může používat značku struktury jako jméno typu, struktury jazyka C mají všechny doposud probrané vlastnosti struktur jazyka C++. Avšak struktury jazyka C++ jdou dále. Například struktury v C++ mohou mít, na rozdíl od

struktur v C, spolu s členskými proměnnými i členské funkce. Ale tyto pokročilejší vlastnosti jsou většinou používány třídami, takže si o nich povíme více při probírání tříd.

Pole struktur

Struktura `inflatable` obsahuje pole (`name`). Je také možné vytvářet pole, jejichž prvky jsou struktury. Postup je zcela stejný jako při vytváření polí základních typů. Například pro vytvoření pole o 100 strukturách inflatable napíšete následující příkaz:

```
inflatable gifts[100];          // pole o 100 strukturách inflatable
```

Tímto způsobem vznikne pole `gifts` typů `inflatable`. Každý prvek tohoto pole, například `gifts[0]` nebo `gifts[99]`, je objektem `inflatable` a můžete na něj použít operátor příslušnosti:

```
cin >> gifts[0].volume;           // použije člen volume první struktury
cout << gifts[99].price << endl;  // zobrazí poslední člen struktury price
```

Pamatujte, že vlastní proměnná `gifts` je polem, ne strukturou, takže konstrukce jako `gifts.price` jsou nepřípustné.

Při inicializování pole struktur spojujeme pravidlo inicializace polí (složenými závorkami uzavřený, čárkami oddělený seznam hodnot pro každý prvek) s pravidlem pro struktury (složenými závorkami uzavřený, čárkami oddělený seznam hodnot pro každý člen). Protože každý prvek pole je strukturou, jeho hodnota je reprezentována inicializací struktury. Výsledkem je tedy složenými závorkami uzavřený, čárkami oddělený seznam hodnot, kde každá hodnota je také složenými závorkami uzavřený, čárkami oddělený seznam hodnot:

```
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 si můžete zvolit formát zápisu podle vašich potřeb. Obě inicializace mohou být na stejném řádku nebo každá inicializace samostatného členu struktury například na vlastním řádku.

Ve výpisu 4.13 je uveden krátký příklad, který využívá pole struktur. Poznamenejme, že vzhledem k tomu, že `guests` je pole `inflatable`, typ `guest[0]` je také `inflatable`, takže pro přístup ke členu struktury `inflatable` můžeme v tomto případě použít operátor tečka.

Výpis 4.13 arrstruc.cpp

```
// arrstruc.cpp -- pole struktur
#include <iostream>
struct inflatable
{
   char name[20];
   float volume;
   double price;
};
int main()
{
   using namespace std;
   inflatable guests[2] =         // inicializace pole struktur
```

```
    {
       {"Bambi", 0.1, 21.99},          // první struktura v poli
       {"Godzilla", 56, 565.99}        // další struktura v poli
    };

    cout << "Hoste " << guests[0].name << " a " << guests[1].name
          << "\nmaji celkovy objem "
          << guests[0].volume + guests[1].volume << " krychlovych metru.\n";
    return 0;
}
```

Zde je výstup z programu uvedeného na výpisu 4.13:

```
Hoste Bambi a Godzilla
maji celkovy objem 56.1 krychlovych metru.
```

Bitová pole ve strukturách

Jazyk C++, podobně jako C, umožňuje specifikovat členy struktury, které zabírají určitý počet bitů. Tato vlastnost se může hodit například při vytváření datové struktury, která má odpovídat registru nějakého hardwarového zařízení. Typ umístěný do bitového pole by měl být celočíselný nebo výčtový (výčtové typy budeme probírat v této kapitole později) následovaný dvojtečkou s číslem, které určuje skutečný počet použitých bitů. Nepojmenovaná pole představují vloženou mezeru. Každý člen nazýváme *bitové pole*. Zde je příklad:

```
struct torgle_register
{
   unsigned int SN : 4;          // 4 bity pro hodnotu SN
   unsigned int : 4;             // 4 nevyužité bity
   bool goodIn : 1;              // platný vstup (1 bit)
   bool goodTorgle : 1;          // úspěšné kroucení
};
```

Pole můžete inicializovat obvyklým způsobem a pro přístup k bitovým polím se používá standardní zápis jako u struktur:

```
torgle_register tr;
...
if (tr.goodIn)                   // příkaz if je popsán v kapitole 6
...
```

Bitová pole jsou obvykle využívána v nízkoúrovňovém programování. Alternativním přístupem bývá použití celočíselného typu a bitových operátorů z Přílohy E „Další operátory“.

Uniony

Datový formát *union* může ukládat různé datové typy, ale pouze jeden typ současně. To znamená, že zatímco struktura obsahuje řekněme `int`, `long` a `double`, v unionu může být buďto `int` nebo `long` nebo `double`. Syntaxe je podobná jako u struktury, ale význam je různý. Podívejme se například na následující deklaraci:

```
union one4all
{
```

```
      int int_val;
      long long_val;
      double double_val;
   };
```

Proměnnou `one4all` můžete využít na uložení typu `int`, `long` nebo `double` podle toho, který typ potřebujete v různých situacích:

```
   one4all pail;
      pail.int_val = 15;          // uloží typ int
      cout << pail.int_val;
      pail.double_val = 1.38;     // uloží typ double, hodnota typu int je
   zapomenuta
   cout << pail.double_val;
```

Proměnná `pail` tedy může jednou posloužit jako typ `int` a jindy jako `double`. Jméno členu určuje kapacitu, se kterou tato proměnná pracuje. Protože union vždy obsahuje pouze jednu hodnotu, musí mít dostatek místa pro uložení největšího členu. Z toho důvodu je velikost unionu rovna velikosti jeho největšího členu.

Použití unionu tedy šetří místo, pokud datová položka může mít dva nebo více formátů, které se nikdy nevyskytnou současně. Předpokládejme, že spravujete inventář různých věcí, které mají někdy celočíselný identifikátor a jindy řetězcový. Potom můžete napsat následující kód:

```
   struct veci
   {
   char znacka[20];
   int typ;
   union id                       // formát závisí na typu věci
   {
      long id_num;                // první typ věcí
      char id_char[20];           // ostatní věci
   } id_val;
   };
   ...
   veci cena;
   ...
   if (cena.typ == 1)             // příkaz if-else kapitola 6
      cin >> cena.id_val.id_num;  // pro určení režimu použijte jméno členu
   else
      cin >> cena.id_val.id_char;
```

Anonymní union nemá jméno; v podstatě se jeho členy stávají proměnnými, které sdílejí stejnou adresu. V daném okamžiku může být samozřejmě aktuální pouze jeden člen:

```
   struct veci
   {
      char znacka[20];
      int typ;
      union                       // anonymní union
   {
         long id_num;             // typ veci 1
         char id_char[20];        // ostatni veci
      };
   };
   ...
```

```
veci cena;
...
if (cena.typ == 1)
   cin >> cena.id_num;
else
   cin >> cena.id_char;
```

Protože je union anonymní, `id_num` a `id_char` jsou považovány za dva členy struktury `veci`, které sdílejí stejnou adresu. Tím je odstraněna nutnost přechodného identifikátoru `id_val`. Je na programátorovi, aby sledoval, která z uvedených voleb je právě aktivní.

Enumerace (výčtové typy)

Klíčové slovo `enum` poskytuje v jazyce C++ alternativní prostředek ke `const` pro vytváření symbolických konstant. V omezeném rozsahu také umožňuje definovat nové typy. Syntaxe použití `enum` se podobá syntaxi struktury. Podívejte se například na následující příkaz:

```
enum spektrum {cervena, oranzova, zluta, zelena, modra, fialova, indigo,
ultrafialova};
```

Tento příkaz provádí dvě činnosti:

- Z identifikátoru `spektrum` vytváří jméno nového typu; typ `spektrum` je označován jako *výčtový typ* nebo *enumerace*, podobně jako proměnná typu `struct` je nazývána strukturou.
- Stanoví `cervena`, `oranzova`, `zluta` a tak dále symbolickými konstantami pro celá čísla 0–7. Tato čísla se nazývají *enumerátory*.

Implicitně se enumerátorům přiřazují celočíselné hodnoty počínaje nulou pro první enumerátor, 1 pro další atd. Implicitní hodnoty můžete přepsat přiřazením celočíselných hodnot. Jak se to dělá, si ukážeme později v této kapitole.

Jméno výčtu můžete použít na deklaraci proměnné tohoto typu:

```
spektrum skupina;
```

Výčtová proměnná má některé zvláštní vlastnosti, na které se nyní podíváme.

Jediné platné hodnoty, které můžete přiřadit proměnné enumerace bez přetypování, jsou hodnoty enumerátorů použité v definici odpovídajícího typu. Proto platí následující:

```
skupina = modra;   // platné, modra je enumerátorem
skupina = 2000;    // neplatné, 2000 není enumerátorem
```

Tedy proměnná spektrum je omezena pouze na osm možných hodnot. Některé překladače vypisují při pokusu přiřadit neplatnou hodnotu chybové hlášení, jiné pouze varování. Z důvodů zachování maximální přenositelnosti byste měli považovat přiřazení nevýčtové hodnoty výčtové proměnné za chybu.

Pro výčtový typ je definován pouze přiřazovací operátor. Konkrétně nejsou definovány aritmetické operace:

```
skupina = oranzova;            // platné
++skupina;                     // neplatné, ++ viz kapitola 5
skupina = oranzova + cervena;  // neplatné, avšak poněkud nejasné
...
```

Avšak některé implementace toto omezení nedodržují. To může způsobit porušení mezí typu. Má-li proměnná skupina například hodnotu ultrafialova, nebo 7, potom výraz ++skupina, pokud je platný, inkrementuje hodnotu proměnné skupina na 8, což není platnou hodnotou pro typ spektrum. Znovu platí, že kvůli maximální přenositelnosti byste se měli řídit výše uvedeným přísnějším omezením.

Enumerátory mají celočíselný typ a mohou být povýšeny na typ int, ale typy int nejsou automaticky převáděny na výčtový typ:

```
int barva = modra;            // platné, typ spektrum je povýšen na int
skupina = 3;                  // neplatné, int není převoditelný na spektrum
barva = 3 + cervena;          // platné, hodnota cervena je převedena na int
...
```

V tomto příkladu si všimněte, že i když hodnota 3 odpovídá enumerátoru zelena, je přiřazení hodnoty 3 do proměnné skupina typovou chybou. Některé implementace si však dodržování tohoto omezení nevynucují. Ve výrazu 3 + cervena není sčítání pro enumerátory definováno. Avšak hodnota cervena je převedena na typ int a výsledkem je typ int. Díky převodu výčtového typu na typ int v této situaci můžete výčtový typ používat v aritmetických výrazech spolu s obvyklými celými čísly, i když pro výčty samotné nejsou definovány aritmetické operace.

Shora uvedený příklad

```
skupina = oranzova + cervena;  // neplatné, avšak poněkud nejasné
```

zhavaruje z poněkud nejasného důvodu. Je pravda, že operátor + není pro výčtový typ definován. Avšak také je pravda, že výčtové typy jsou v aritmetických výrazech převedeny na celá čísla, takže výraz oranzova + cervena lze převést na 1 + 0, což je platný výraz. Je však typu int, takže jej nelze přiřadit do proměnné skupina typu spektrum.

Hodnotu typu int můžete přiřadit typu enum za předpokladu, že je hodnota platná a použijete explicitní přetypování:

```
skupina = spektrum(3);        // přetypování 3 na typ spektrum
```

A co když se pokusíte přetypovat nevhodnou hodnotu? V takovém případě není výsledek definován, což znamená, že tento pokus nebude označen za chybu, ale na hodnotu jeho výsledku se nemůžete spolehnout:

```
skupina = spektrum(40003);    // nedefinováno
```

(Podívejte se na podkapitolu „Rozsahy hodnot enumerací“ později v této kapitole, která rozebírá vhodné a nevhodné hodnoty.)

Jak sami vidíte, pravidla pro výčtové proměnné jsou velmi omezující. V praxi se enumerace používají častěji jako způsob definování příbuzných symbolických konstant než jako prostředek pro definování nového typu. Výčtový typ můžete například použít na definování symbolických konstant pro příkaz switch. (V kapitole 6 uvidíte příklad.) Chcete-li používat pouze konstanty a nevytvářet proměnné výčtového typu, můžete vynechat jeho jméno, jako v tomto příkladu:

enum {cervena, oranzova, zluta, zelena, modra, fialova, indigo, ultrafialova};

Nastavení hodnot enumerátorů

Hodnoty enumerátorů můžete nastavit explicitně pomocí přiřazovacího operátoru:

```
enum bity{jedna = 1, dve = 2, ctyri = 4, osm = 8};
```

Přiřazované hodnoty musí být celá čísla. Explicitně můžete definovat pouze některé enumerátory:

```
enum velkykrok{prvni, druhy = 100, treti};
```

V tomto případě je enumerátoru `prvni` implicitně přiřazena hodnota 0. Následující neinicializované enumerátory jsou o jedničku větší než jejich předchůdci. Takže enumerátor `treti` by měl mít hodnotu 101.

Dokonce můžete vytvořit více než jeden enumerátor se stejnou hodnotou:

```
enum {zero, null = 0, jedna, cislo_jedna = 1};
```

Zde má enumerátor `zero` i `null` hodnotu 0 a `jedna` i `cislo_jedna` hodnotu 1. V dřívějších verzích jazyka C++ jste mohli enumerátorům přiřazovat pouze hodnoty typu `int` (nebo hodnoty, které se dají na `int` povýšit), ale toto omezení bylo odstraněno, takže můžete používat i hodnoty typu `long`.

Rozsahy hodnot výčtového typu

Původně byly platnými hodnotami enumerací pouze hodnoty uvedené v deklaraci. Ale nyní jazyk C++ podporuje vylepšený způsob, pomocí kterého můžete přiřazovat hodnoty přetypováním na enumerační proměnnou. Každá enumerace má *rozsah* a v tomto rozsahu můžete přiřadit libovolnou celočíselnou hodnotu, i když není enumerační hodnotou, přetypováním na enumerační proměnnou. Předpokládejme, že jména `mujpriznak` a `bity` jsou definována takto:

```
enum bity{jedna = 1, dve = 2, ctyri = 4, osm = 8};
bity mujpriznak;
```

Potom je následující příkaz platný:

```
mujpriznak = bity(6); // platný, protože 6 je v rozsahu enumerace bity
```

Zde 6 není jednou z enumerací, ale leží v enumeracemi definovaném rozsahu.

Rozsah je definován následovně. Abyste zjistili horní mez, vezměte největší hodnotu enumerátoru. Zjistěte nejmenší mocninu dvou, která je větší než tato největší hodnota, odečtěte jedničku a toto číslo je horní hranicí rozsahu. Například největší hodnota dříve definovaného enumerátoru velkykrok je 101. Nejmenší mocnina dvou větší než tato hodnota je 128, takže horní hranicí rozsahu je 127. Abyste nalezli spodní hranici, zjistěte nejmenší hodnotu enumerátoru. Pokud je to nula nebo větší číslo, spodní hranicí rozsahu je 0. Jestliže je nejmenší enumerátor záporný, použijte stejný postup jako při hledání horní hranice, ale odmyslete si záporné znaménko. Například je-li nejmenší enumerátor –6, následující mocnina dvou (násobte záporným znaménkem) je –8, a spodní hranice je –7.

Překladač totiž může určovat, kolik prostoru bude pro uložení enumerace potřebovat. Může použít jeden bajt nebo méně na enumerace s malým rozsahem a čtyři bajty pro enumerace s hodnotami typu `long`.

Ukazatele a volná paměť

Na začátku kapitoly 3 „Práce s daty“ jsme si uvedli tři základní vlastnosti, které musí počítačový program sledovat při ukládání dat. Abyste si příliš neopotřebovali knihu případným listováním zpět k této kapitole, uvedeme si zmiňované vlastnosti znovu:

- Kde je informace uložena
- Jakou má hodnotu
- Jaký druh informace je uložen

Pro splnění výše uvedených požadavků jsme používali strategii definování jednoduché proměnné. Deklarační příkaz poskytuje typ a symbolické jméno hodnoty. Také řekne programu, aby pro danou hodnotu alokoval paměť a vnitřně si udržoval o tomto místě přehled.

Podívejme se nyní na druhou strategii, která bude velmi důležitá při vývoji tříd jazyka C++. Tato strategie je založena na ukazatelích, což jsou proměnné, které místo vlastní hodnoty ukládají její adresu. Ale dříve, než si budeme povídat o ukazatelích, se podíváme, jak je možné zjišťovat adresy obyčejných proměnných. Chcete-li získat umístění proměnné, můžete jednoduše použít adresový operátor, který představuje znak &; pokud je například home proměnná, výrazem &home získáte její adresu. Výpis 4.9 tento operátor demonstruje.

Výpis 4.14 address.cpp

```
// address.cpp -- použití operátoru & na zjištění adresy
#include <iostream>
int main()
{
   using namespace std;
   int donuts = 6;
   double cups = 4.5;

   cout << "hodnota promenne donuts = " << donuts;
   cout << " a adresa promenne donuts = " << &donuts << endl;
// Poznámka: možná budete muset použít unsigned (&donuts)
// a unsigned (&cups)
   cout << "hodnota promenne cups = " << cups;
   cout << " a adresa promenne cups = " << &cups << endl;
   return 0;
}
```

KOMPATIBILITA

`cout` je chytrý objekt, ale některé verze jsou chytřejší než jiné. Proto některé implementace nemusejí rozpoznat typ ukazatele. V takovém případě musíte adresu přetypovat, aby se stala rozpoznatelným typem, jako je například `unsigned int`. Vhodné přetypování závisí na paměťovém modelu. Standardní paměťový model operačního systému DOS používá 2bajtové adresy, zde je `unsigned int` správným přetypováním. Některé paměťové modely DOS však používají 4bajtové adresy, které vyžadují přetypování na `unsigned long`.

Zde je výstup z programu z výpisu 4.14 na jednom systému:

```
hodnota promenne donuts = 6 a adresa promenne donuts = 0x0065fd40
hodnota promenne cups = 4.5 a adresa promenne cups = 0x0065fd44
```

Objekt cout používá při výpisu adres hexadecimální čísla, protože tato notace je pro popis paměti obvyklá. Naše implementace ukládá proměnnou `donuts` na nižší místo v paměti než `cups`. Rozdíl mezi oběma adresami je 0x0065fd44 – 0x0065fd40, neboli 4. To dává smysl, protože `donuts` je typu `int`, který používá čtyři bajty. Různé systémy budou samozřejmě mít jiné hodnoty adres. Některé také mohou ukládat nejprve proměnnou `cups` a teprve potom `donuts`, čímž získáme rozdíl 8, protože proměnná `cups` je typu `double`. Některé implemetace dokonce nemusí používat souvislou paměť.

Při použití obyčejných proměnných se s hodnotou pracuje jako s pojmenovaným množstvím a s umístěním jako s odvozeným množstvím. Nyní se podíváme na strategii ukazatelů, která představuje základ filozofie správy paměti při programování v jazyce C++. (Viz následující poznámka Ukazatele a filozofie jazyka C++.)

Ukazatele a filozofie jazyka C++

Objektově orientované programování se liší od tradičního procedurálního programování v důrazu, který klade na rozhodování za běhu programu místo v době překladu. Doba běhu programu (runtime) označuje čas vykonávání programu a doba překladu časový úsek sestavování programu překladačem. Rozhodování za běhu programu je, jako byste si na dovolené vybírali místa, která chcete navštívit, podle momentálního počasí a nálady, zatímco rozhodnutí v době překladu můžeme přirovnat k dodržování předem dohodnutého rozvrhu bez ohledu na změny podmínek.

Rozhodování za běhu poskytuje pružnost v přizpůsobení se aktuálním okolnostem. Podívejme se například na alokování paměti pro pole. Tradiční způsob představuje deklarace pole. Při deklaraci pole v C++ se musíte rozhodnout pro určitou velikost pole. Tato velikost je tedy určena při překladu programu; to znamená rozhodnutí v době překladu. Můžete mít pole, o kterém si myslíte, že 80 % času bude dostatečných 20 prvků, ale příležitostně bude program muset zpracovat 200 prvků. Z důvodů bezpečnosti použijete pole s 200 prvky. To má za následek plýtvání pamětí po většinu času běhu programu. OOP se pokouší vytvořit program mnohem pružnější tím, že ponechává takováto rozhodnutí na dobu běhu. Tak můžete běžícímu programu říci, že potřebujete v jednom okamžiku pouze 20 prvků a v jiném 205 prvků.

Zkrátka rozhodnutí o velikosti pole ponecháte na dobu běhu programu. Aby byl tento přístup možný, musí jazyk poskytovat vytváření pole – nebo jeho ekvivalentu – za běhu programu. Metoda jazyka C++, kterou brzy uvidíte, používá na vyžádání správného množství paměti klíčové slovo `new` a pro sledování umístění nově alokované paměti ukazatele.

Nová strategie správy uložených dat mění význam práce s umístěním jako s pojmenovaným množstvím a s hodnotou jako s odvozeným množstvím. Zvláštní typ proměnné – *ukazatel* – uchovává adresu hodnoty. Tedy jméno ukazatele reprezentuje umístění. Použití operátoru `*`, kterému říkáme *nepřímá hodnota* nebo *dereferenční operátor*, vrací hodnotu uloženou na daném místě. (Ano, jde o stejný symbol `*` používaný i pro násobení; jazyk C++ určuje podle souvislostí, zda máte na mysli násobení nebo dereferenci.) Předpokládejme, že proměnná `manly` je ukazatel. Potom výraz `manly` představuje adresu a `*manly` hodnotu uloženou na této adrese. Spojení `*manly` se stává obdobou obyčejného typu proměnné `int`. Výpis 4.10 tyto vlastnosti ukazuje. Také předvádí deklaraci ukazatele.

Výpis 4.15 pointer.cpp

```
// pointer.cpp -- naše první ukazatelová proměnná
#include <iostream>
int main()
{
   using namespace std;
   int updates = 6;         // deklaruje proměnnou
   int * p_updates;         // deklaruje ukazatel na int

   p_updates = &updates; // přiřazuje adresu int ukazateli

// vyjadřuje hodnoty dvěma způsoby
   cout << "Hodnoty: updates = " << updates;
   cout << ", *p_updates = " << *p_updates << endl;

// vyjadřuje hodnoty dvěma způsoby
   cout << "Adresy: &updates = " << &updates;
   cout << ", p_updates = " << p_updates << endl;

// používá ukazatel na změnu hodnoty
   *p_updates = *p_updates + 1;
   cout << "Promenna updates nyni = " << updates << endl;
   return 0;
}
```

Zde je výstup z programu na výpisu 4.15:

```
Hodnoty: updates = 6, *p_updates = 6
Adresy: &updates = 0x0065fd48, p_updates = 0x0065fd48
Promenna updates nyni = 7
```

Jak vidíte, proměnná `updates` typu `int` a ukazatelová proměnná `p_updates`, jsou pouze dvě strany jedné mince. Hlavním úkolem proměnné `updates` je reprezentace hodnoty a operátorem `&` lze získat adresu, zatímco hlavním úkolem proměnné `p_updates` je reprezentace adresy a operátorem `*` lze získat hodnotu. (Viz obrázek 4.8.) Protože `p_updates` ukazuje na `updates`, `*p_updates` a `updates` jsou zcela ekvivalentní. Výraz `*p_updates` můžete použít úplně stejně jako proměnnou typu `int`. Program také ukazuje, že je dokonce možné přiřazovat `*p_updates` hodnoty. Pokud to uděláte, změní se hodnota proměnné `updates`, na kterou `p_updates` ukazuje.

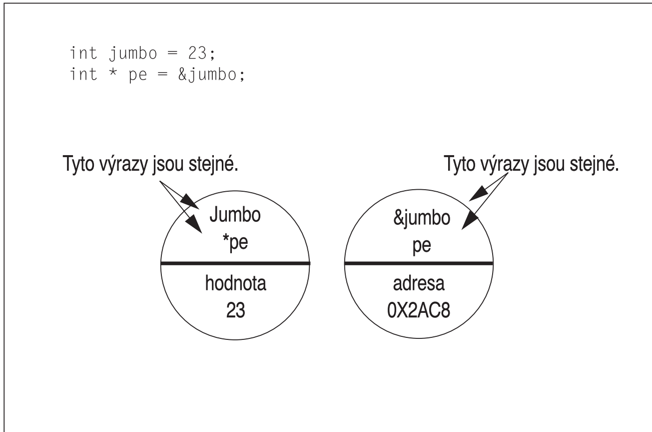
Deklarace a inicializace ukazatelů

Podívejme se na postup deklarace ukazatelů. Počítač musí sledovat typ proměnné, na kterou se ukazatel odvolává. Například adresa typu char vypadá stejně jako adresa typu `double`, ale `char` a `double` používají různý počet bajtů a různé vnitřní formáty pro uložení hodnot. Proto musí deklarace ukazatele určovat typ dat, na který ukazatel ukazuje.

Poslední příklad má tuto deklaraci:

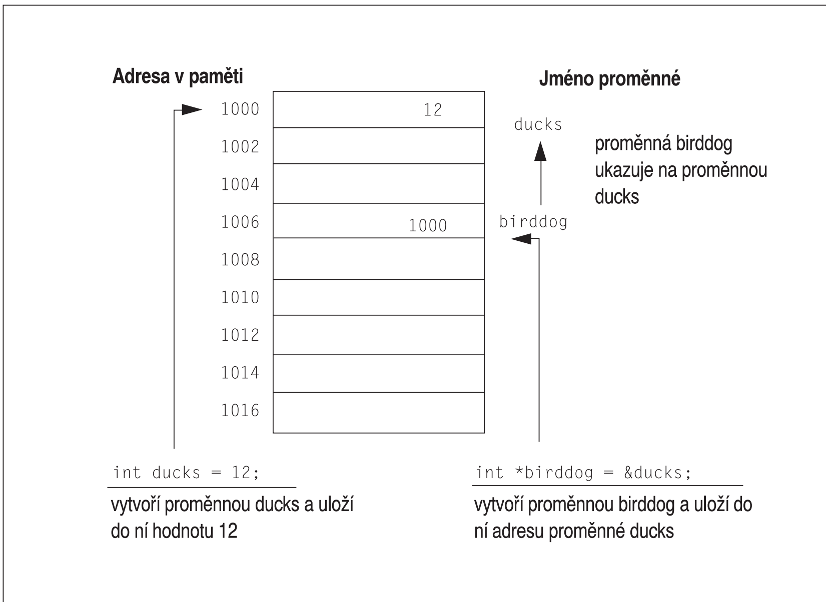
```
int * p_updates;
```

Ta stanovuje, že spojení `* p_update` je typu `int`. Protože je operátor `*` používán s ukazatelem, musí být vlastní proměnná `p_update` ukazatel. Říkáme, že `p_update` ukazuje na typ `int`. A dále říkáme, že typ proměnné `p_update` je ukazatel na `int` nebo stručněji `int *`.



Obrázek 4.8 Dvě strany jedné mince

Pro zopakování: `p_updates` představuje ukazatel (adresu) a `*p_updates` typ `int`, nikoli ukazatel. (Viz obrázek 4.9.)



Obrázek 4.9 Ukazatele uchovávají adresy

Mimochodem, umístění mezer kolem operátoru `*` je volitelné. Programátoři v jazyce C používali tradičně tento tvar:

```
int *ptr;
```

To zdůrazňuje myšlenku, že spojení `*ptr` představuje hodnotu typu `int`. Na druhou stranu, mnoho programátorů v jazyce C++ používá následující formu:

```
int* ptr;
```

Tento zápis vyzdvihuje myšlenku, že `int*` je typ ukazatel na `int`. Překladač nedělá v umístění mezer žádné rozdíly. Mějte však na vědomí, že deklarace

```
int* p1, p2;
```

vytváří jeden ukazatel (`p1`) a jednu obyčejnou proměnnou (`p2`) typu `int`. Pro každé jméno ukazatelové proměnné potřebujete znak `*`.

ZAPAMATUJTE SI

V jazyce C++ je kombinace `int *` složeným typem, ukazatel na `int`.

Při deklaraci ukazatelů na ostatní typy 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 typ `double`, překladač ví, že `*tax_ptr` obsahuje hodnotu typu `double`. Ví tedy, že `*tax_ptr` reprezentuje číslo uložené ve formátu s pohyblivou desetinnou čárkou, které zabírá (na většině systémů) osm bajtů. Ukazatelová proměnná není nikdy pouze ukazatelem. Vždy se jedná o ukazatel na určitý typ. `tax_ptr` má typ ukazatel na `double` (nebo typ `double *`) a `str` má typ ukazatel na `char` (nebo `char *`). Ačkoli jde v obou případech o ukazatele, jsou to ukazatele na dva různé typy. Ukazatele jsou podobně jako pole založeny na jiných typech.

Všimněte si, že zatímco `tax_ptr` a `str` ukazují na datové typy různých velikostí, vlastní proměnné `tax_ptr` a `str` mají stejné velikosti. To znamená, že adresa typu `char` má stejnou velikost jako adresa typu `double`, obdobně by 1016 mohlo být číslem adresy obchodního domu, zatímco 1024 číslem adresy malé chaloupky. Velikost nebo hodnota adresy vám ve skutečnosti nic neříká o velikosti nebo druhu proměnné nebo budovy, kterou na dané adrese najdete. Adresa obvykle vyžaduje dva nebo čtyři bajty, to záleží na počítačovém systému. (Některé systémy mohou mít větší adresy a systém může používat různé velikosti adres pro různé typy.)

Pro inicializaci ukazatele můžete použít deklarační příkaz. V takovém případě je inicializován ukazatel, nikoli hodnota, na kterou ukazuje. To znamená, že příkazy:

```
int higgens = 5;
int * pt = &higgens;
```

nastavují `pt` a ne `*pt` na hodnotu `&higgens`.

Výpis 4.16 ukazuje inicializaci ukazatele adresou.

Výpis 4.16 init_ptr.cpp

```
// init_ptr.cpp -- inicializuje ukazatel
#include <iostream>
int main()
{
   using namespace std;
   int higgens = 5;
   int * pt = &higgens;

   cout << "Hodnota higgens = " << higgens
         << "; Adresa higgens = " << &higgens << "\n";
```

```
    cout << "Hodnota *pt = " << *pt
          << "; Hodnota pt = " << pt << "\n";
    return 0;
}
```

Zde je výstup z programu na výpisu 4.16:

```
Hodnota higgens = 5; Adresa higgens = 0012FED4
Hodnota *pt = 5; Hodnota pt = 0012FED4
```

Jak vidíte, program adresou `higgens` inicializuje `pt`, nikoli `*pt`.

Nebezpečné ukazatele

Nepozorné používání ukazatelů je velmi nebezpečné. Je důležité vědět, že když v C++ vytvoříte ukazatel, počítač alokuje paměť pro uložení adresy, ale nealokuje paměť pro uložení dat, na která adresa ukazuje. Vytvoření prostoru pro data vyžaduje samostatný krok. Opomenutí tohoto kroku, podobně jako v následující ukázce, může skončit pohromou:

```
long * fellow;          // vytvoření ukazatele na long
*fellow = 223323;       // umístění hodnoty do země nikoho
```

Jistě, proměnná `fellow` je ukazatel. Ale kam vlastně ukazuje? Ve výše uvedeném kódu chybí přidělení adresy ukazateli `fellow`. Kam je tedy uložena hodnota `223323`? Na to nelze jednoduše odpovědět. Protože ukazatel `fellow` nebyl inicializován, může mít jakoukoliv hodnotu. Tuto blíže neurčenou hodnotu program interpretuje jako adresu, na kterou uloží hodnotu `223323`. Kdyby měl ukazatel `fellow` hodnotu `1200`, potom by se program pokusil umístit data na adresu `1200`, i když by to byla adresa uprostřed programového kódu. Je velice pravděpodobné, že ať už `fellow` ukazuje kamkoli, není to místo, kam chcete uložit číslo `223323`. Tento druh chyb patří mezi nejzáludnější a jejich odhalování není snadné.

UPOZORNĚNÍ

Zlaté pravidlo ukazatelů: VŽDY inicializujte ukazatel jednoznačnou a vhodnou adresou před tím, než na něj použijete dereferenční operátor (`*`).

Ukazatele a čísla

Ukazatele nejsou celočíselné typy, i když počítače obvykle pracují s adresami jako s celými čísly. Pojmově jsou ukazatele jinými typy než celá čísla. Celá čísla můžeme sčítat, odčítat, dělit atd. Ale ukazatele popisují umístění a například nemá smysl násobit jedno umístění druhým. Ukazatele a celá čísla se tedy od sebe liší v operacích, které s nimi můžeme provádět. Proto nemůžeme jednoduše přiřadit ukazateli celé číslo:

```
int * pt;
pt = 0xB8000000;        // neshoda 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. Jazyk C před vydáním nového standardu C99 takováto přiřazení umožňoval. Avšak jazyk C++ vyžaduje shodu typů mnohem přísněji a překladač vypisuje chybové hlášení o neshodě typů. Chcete-li jako adresu uvést číselnou hodnotu, měli byste toto číslo převést na vhodný adresový typ pomocí přetypování:

```
int * pt;
pt = (int *) 0xB8000000; // nyní se typy shodují
```

Nyní obě strany přiřazovacího příkazu reprezentují adresy celých čísel, takže je přiřazení platné. Všimněte si, že i když se jedná o adresu typu `int`, nemusí mít vlastní proměnná `pt` typ `int`. Například ve velkém paměťovém modelu na IBM PC s operačním systémem DOS je typ `int` 2bajtovou hodnotou, zatímco adresy jsou 4bajtové. Ukazatele mají některé další zajímavé vlastnosti, které si při vhodné příležitosti také probereme. Mezitím se podíváme na využití ukazatelů při alokaci paměťového prostoru za běhu programu.

Alokace paměti pomocí operátoru new

Protože již nyní máme o práci ukazatelů jistou představu, podíváme se, jak můžeme s jejich pomocí implementovat důležitou techniku OOP, kterou je alokace paměti za běhu programu. Doposud jsme ukazatele inicializovali adresami proměnných; proměnné představují *pojmenovanou* paměť alokovanou v době překladu a ukazatele pouze poskytují druhé jméno pro tuto paměť, ke které byste stejně mohli přistupovat pod jménem proměnné. Pravá cena ukazatelů se projeví při alokování *nepojmenované* paměti za běhu programu pro ukládání hodnot. K takto alokované paměti je možné přistupovat pouze prostřednictvím ukazatelů. V jazyce C se alokace paměti provádí pomocí knihovní funkce `malloc()`. I v jazyce C++ je tento postup možný, ale C++ má také lepší způsob, který představuje operátor `new`.

Vyzkoušejme si tento nový postup vytvořením nepojmenované paměti za běhu programu pro hodnotu typu `int` a přístupem k odpovídající hodnotě pomocí ukazatele. Operátor jazyka C++ `new` má při této činnosti klíčovou úlohu. Operátoru `new` musíte sdělit, pro jaký datový typ dat paměť chcete; `new` nalezne blok správné velikosti a vrátí jeho adresu. Tuto adresu pouze přiřadíte ukazateli a to je vše. Zde je příklad postupu:

```
int * pn = new int;
```

Část `new int` říká programu, že chcete jistou novou (`new`) paměť vhodnou pro uložení typu `int`. Operátor `new` používá předaný typ k určení počtu potřebných bajtů. Potom nalezne vhodnou paměť a vrátí její adresu. Dále dojde k přiřazení vrácené adresy proměnné `pn`, která je deklarována jako typ ukazatele na `int`. Nyní představuje `pn` adresu a `*pn` hodnotu na této adrese uloženou. Porovnejte tento postup s přiřazením adresy proměnné ukazateli:

```
int higgens;
int * pt = &higgens;
```

V obou případech (pn a pt) přiřazujete ukazateli adresu typu `int`. Ve druhém případě můžete přistupovat k typu `int` i prostřednictvím jména `higgens`. V prvním případě je typ `int` přístupný pouze pomocí ukazatele. To vyvolává otázku: Protože paměť, na kterou `pn` ukazuje, nemá jméno, jak ji budeme nazývat? Říkáme, že `pn` ukazuje na *datový objekt*. Není to „objekt" ve smyslu „objektově orientovaného programování"; je to pouze „objekt" ve smyslu „věci". Výraz „datový objekt" představuje obecnější výraz než „proměnná", protože znamená jakýkoliv blok paměti alokovaný pro datovou položku. Tedy proměnná je také datovým objektem, ale paměť, na kterou `pn` ukazuje, není proměnnou. Metoda správy datových objektů pomocí ukazatelů může vypadat zpočátku nešikovně, ale nabízí lepší možnosti práce s pamětí.

Obecný tvar získání a přiřazení paměti pro jednoduchý datový objekt, který může být strukturou stejně jako základním typem, je:

```
jménoTypu jménoUkazatele = new jménoTypu;
```

Datový typ je použit dvakrát: jednou pro určení druhu požadované paměti a podruhé pro deklaraci vhodného ukazatele. Pokud jste již ukazatel správného typu deklarovali, můžete ho samozřejmě použít místo deklarace nového. Výpis 4.17 ukazuje použití klíčového slova `new` se dvěma různými typy.

Výpis 4.17 use_new.cpp

```
// use_new.cpp -- použití operátoru new
#include <iostream>
int main()
{
   using namespace std;
   int * pt = new int;                    // alokuje prostor pro int
   *pt = 1001;                            // uloží tam hodnotu

   cout << "int ";
   cout << "hodnota = " << *pt << ": umisteni = " << pt << endl;

   double * pd = new double;              // alokuje prostor pro double
   *pd = 10000001.0;                      // uloží tam double

   cout << "double ";
   cout << "hodnota = " << *pd << ": umisteni = " << pd << endl;
   cout << "velikost pt = " << sizeof pt;
   cout << ": velikost *pt = " << sizeof *pt << endl;
   cout << "velikost pd = " << sizeof pd;
   cout << ": velikost *pd = " << sizeof *pd << endl;
   return 0;
}
```

Zde je výstup z programu uvedeného na výpisu 4.17:

```
int hodnota = 1001: umisteni = 0x004301a8
double hodnota = 1e+007: umisteni = 0x004301d8
velikost pt = 4: velikost *pt = 4
velikost pd = 4: velikost *pd = 8
```

Přesné hodnoty umístění v paměti jsou samozřejmě na různých systémech odlišné.

Poznámky k programu

Program na výpisu 4.17 alokuje za běhu paměť pro datové objekty typu `int` a `double` pomocí operátoru `new`. Ukazatele `pt` a `pd` na tyto dva datové objekty ukazují. Bez ukazatelů není možné k těmto paměťovým místům přistupovat. S nimi můžete použít `*pt` a `*pn` stejně jako proměnné. Přiřazení hodnot do `*pt` a `*pd` způsobí přiřazení hodnot novým datovým objektům. Obdobně je možné vytisknout `*pt` a `*pd` a tímto způsobem hodnoty zobrazit.

Program také předvádí jeden z důvodů nutnosti deklarace typu, na který ukazatel ukazuje. Vlastní adresa odhaluje pouze počáteční adresu uložení objektu, nikoli jeho typ nebo počet použitých bajtů. Podívejte se na adresy obou hodnot. Jsou to pouze čísla bez informací o typu nebo 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 hodnota typu double o 8 bajtech, zatímco `*pt` je hodnotou typu `int` o 4 bajtech. Když `use_new.cpp` tiskne hodnotu `*pd`, cout může říci, kolik bajtů má přečíst a jak je má interpretovat.

Nedostatek paměti

Může se stát, že počítač nemá dostatek dostupné paměti, aby vyhověl požadavku operátoru `new`. V takovém případě vrací `new` hodnotu 0. V jazyce C++ se ukazatel s hodnotou 0 nazývá *nulovým ukazatelem*. C++ zaručuje, že nulový ukazatel nikdy neukazuje na platná data, takže se často používá k označení selhání operátorů nebo funkcí, které jinak vracejí použitelné ukazatele. Až se naučíte používat příkaz `if` (v kapitole 6), můžete kontrolovat, zda `new` vrací nulový ukazatel a tím váš program chrání před pokusem o překročení vymezených hranic. Kromě vrácení prázdného ukazatele po selhání alokace paměti může operátor `new` vyhodit výjimku `bad_alloc`. Kapitola 15 „Přátelé, výjimky a další" popisuje mechanismy výjimek.

Uvolnění paměti operátorem delete

Použití operátoru `new` pro vyžádání potřebné paměti představuje pouze atraktivnější půlku vybavení jazyka C++ pro správu paměti. Druhou půlkou je operátor `delete`, který umožňuje vrácení nepotřebné paměti systému. To je důležitým krokem co nejefektivnějšího využití paměti. Vrácená nebo také *uvolněná* paměť může být znovu použita jinými částmi programu. Pro tuto činnost je používáno klíčové slovo `delete` následované ukazatelem na blok paměti, který byl původně alokován pomocí `new`:

```
int * ps = new int;         // alokace paměti pomocí new
. . .                       // použití paměti
delete ps;                  // uvolnění paměti pomocí delete
```

Výše uvedený příkaz ruší paměť, na kterou ukazuje `ps`; ale neodstraňuje vlastní ukazatel. Můžete ho znovu použít, například aby ukazoval na další nové místo. Použití operátorů `new` a `delete` by mělo být v rovnováze, jinak může docházet k *úniku paměti*, což je označení alokované paměti, která nemůže být dále použita. Jestliže se únik paměti příliš zvětší, může dojít k zastavení programu hledajícího další paměť.

Dříve uvolněný blok paměti by již neměl být znovu uvolňován. Výsledek takovéto činnosti není definován. Operátor delete také nemůže být použit pro uvolnění paměti vytvořené deklarací proměnných:

```
int * ps = new int;         // v pořádku
delete ps;
// v pořádku
delete ps;
// toto již není v pořádku
int jugs = 5;               // v pořádku
int * pi = & jugs;          // v pořádku
delete pi;                  // není povoleno, paměť nebyla alokována
                               operátorem new
```

UPOZORNĚNÍ

Operátor `delete` byste měli používat pouze pro uvolňování paměti alokované operátorem `new`. Nicméně je bezpečné použít operátor `delete` na nulový ukazatel.

Všimněte si, že rozhodujícím kritériem možnosti použití operátoru `delete` je to, zda se jedná o paměť alokovanou operátorem `new`. Nemusíte ale použít stejný ukazatel, kterému jste přiřadili výsledek operátoru `new`; musí se však jednat o stejnou adresu:

```
int * ps = new int;            // alokujte paměť
int * pq = ps;                 // nastavte druhý ukazatel na stejný blok
delete pq;                     // vymažte pomocí druhého ukazatele
```

Obvykle nebudete chtít vytvářet dva ukazatele na stejný blok paměti, protože tím vzrůstá pravděpodobnost chybného uvolnění stejného bloku dvakrát. Ale jak brzy uvidíte, použití druhého ukazatele má smysl, pokud pracujete s funkcí, která vrací ukazatel.

Vytváření dynamických polí pomocí operátoru new

Jestliže program potřebuje pouze jednu hodnotu, měli byste deklarovat jednoduchou proměnnou, protože je to jednodušší, i když méně působivé, než použít na správu malého datového objektu operátor `new` a ukazatel. Operátor `new` obvykle přijde ke slovu při velkém množství dat, které se nachází například v polích, řetězcích a strukturách. V takových případech je použití operátoru `new` užitečné. Předpokládejme, že píšete program, jenž může ale nemusí potřebovat pole, což závisí na informaci, která je programu předána za běhu. Vytvoříte-li pole deklarací, prostor je alokován v době překladu. I když program nakonec pole využije nebo ne, toto pole je vytvořené a zabírá paměť. Alokace pole během překladu se nazývá *statickou vazbou*, to znamená, že takovéto pole je do programu začleněno během překladu. Ale pomocí operátoru `new` můžete pole vytvořit za běhu programu, když ho potřebujete, nebo vytvoření přeskočit, pokud ho nepotřebujete. Za běhu také můžete zadat velikost pole. Vytváření pole za běhu programu je označováno jako *dynamická vazba*. Takto vytvořené pole se nazývá *dynamickým polem*. U statické vazby musíte zadat velikost pole při psaní programu. V případě dynamické vazby může program určit velikosti pole až za běhu.

Nyní se podíváme na dvě základní otázky, které se týkají dynamických polí: jak vytvořit pole pomocí operátoru new jazyka C++ a jak přistupovat k prvkům pole prostřednictvím ukazatelů.

Vytvoření dynamického pole operátorem new

V C++ je jednoduché vytvořit dynamické pole; stačí sdělit operátoru `new` typ prvků pole a jejich požadovaný počet. Syntaxe vyžaduje, že za jménem pole následuje počet prvků v hranatých závorkách. Požadavek na pole o 10 prvcích typu `int` zapíšete následovně:

```
int * psome = new int [10];    // získáte blok o 10 prvcích typu int
```

Operátor `new` vrací adresu prvního prvku bloku. V tomto příkladě je tato hodnota přiřazena ukazateli `psome`. Až program přestane používat alokovaný blok paměti, musíte vyrovnat volání operátoru `new` zavoláním operátoru `delete`.

Když pole vytvoříte pomocí operátoru `new`, měli byste použít alternativní tvar `delete`, který označuje uvolňování pole:

```
delete [] psome;               // uvolnění dynamického pole
```

Přítomnost hranatých závorek říká programu, že má uvolnit celé pole, ne pouze prvek, na který ukazuje ukazatel. Všimněte si, že hranaté závorky jsou mezi `delete` a ukazatelem. Použijete-li `new` bez hranatých závorek, uveďte bez nich i `delete`. Použijete-li `new` s hranatými závorkami, uveďte je i u `delete`. Dřívější verze C++ nemusely rozpoznat notaci s hranatými závorkami. Nicméně ve standardu ANSI/ISO není výsledek záměny tvarů `new` a `delete` definován, což znamená, že se nemůžete spolehnout na předepsané chování.

```
int * pt = new int;
short * ps = new short [500];
delete [] pt;                 // výsledek není definován, nedělejte to
delete ps;                    // výsledek není definován, nedělejte to
```

Zkrátka, používáte-li operátory `new` a `delete`, zachovávejte tato pravidla:

- Nepoužívejte `delete` na uvolnění paměti, kterou nealokoval operátor `new`.
- Nepoužívejte `delete` na uvolnění stejného bloku paměti dvakrát za sebou.
- Použijte `delete []`, pokud jste paměť alokovali pomocí `new []`.
- Použijte `delete` (bez hranatých závorek), pokud jste alokovali jednoduchou entitu pomocí `new`.
- Je bezpečné aplikovat `delete` na nulový ukazatel (nestane se nic).

Nyní se vrátíme k dynamickému poli. Všimněte si, že `psome` představuje ukazatel na jeden typ `int`, kterým je první prvek bloku. Programátor je zodpovědný za udržování přehledu o počtu prvků v bloku. To znamená, že jelikož překladač nesleduje skutečnost, že `psome` ukazuje na první z deseti celých čísel, musíte napsat váš program tak, aby si počet prvků hlídal vlastními prostředky.

Ve skutečnosti program sleduje množství alokované paměti tak, aby mohla být později správně uvolněna pomocí operátoru `delete []`. Ale tato informace není veřejně přístupná; nemůžete použít operátor `sizeof` například pro nalezení počtu bajtů v dynamicky alokované paměti.

Obecný tvar alokování a přiřazení paměti poli je tento:

```
jménoTypu jménoUkazatele new jménoTypu [početPrvků];
```

Vyvolání operátoru `new` zajistí dostatečně velký blok paměti pro uložení *početPrvků* prvků typu *jménoTypu*, přičemž *jménoUkazatele* ukazuje na první prvek. Jak za chvíli uvidíte, *jménoUkazatele* můžete používat mnoha podobnými způsoby jako jméno pole.

Použití dynamického pole

Jak ale vytvořené dynamické pole použijete? Nejprve se na tento problém podíváme teoreticky. Příkaz

```
int * psome = new int [10];    // získáte blok o 10 prvcích typu int
```

vytváří ukazatel `psome`, který ukazuje na první prvek bloku o 10 hodnotách typu `int`. Tento ukazatel si můžete představit jako prst ukazující na první prvek. Předpokládejme, že typ `int` zabírá čtyři bajty. Potom náš pomyslný prst posunutý o čtyři bajty správným směrem bude ukazovat na další prvek v pořadí. Dohromady existuje 10 prvků, které představují rozsah možného pohybu prstu. Příkaz `new` tedy poskytuje všechny informace potřebné k identifikaci každého prvku bloku.

Nyní se podíváme na daný problém prakticky. Jak přistoupíte k některému z těchto prvků? U prvního prvku je to jednoduché. Protože psome ukazuje na první prvek pole, *psome představuje hodnotu prvního prvku. Zbývá tedy zpřístupnit dalších devět prvků. Následující způsob, pokud neznáte jazyk C, vás může trochu překvapit: S ukazatelem lze pracovat, jako by představoval jméno pole. To znamená, že pro první prvek můžete použít psome[0] namísto *psome, pro druhý psome[1] atd. Použití ukazatele pro přístup k dynamickému poli se ukazuje jako velmi jednoduché, i když nemusí být na první pohled zřejmé, proč tato metoda funguje. Můžete takto postupovat, protože jazyk C i C++ ve skutečnosti pracuje vnitřně s poli pomocí ukazatelů. Tato vnitřní podobnost polí a ukazatelů představuje jednu z krás jazyků C a C++. Zmíněnou podobností se budeme zabývat o chvíli později. Nejprve se podíváme na výpis 4.13, který ukazuje, jak můžeme pomocí operátoru new vytvářet dynamická pole a potom pro přístup k jednotlivým prvkům použít zápis známý z polí. Předvádí také základní rozdíl mezi ukazatelem a pravým jménem pole.

Výpis 4.18 arraynew.cpp

```
// arraynew.cpp -- použití operátoru new pro pole
#include <iostream>
int main()
{
   using namespace std;
   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] je " << p3[1] << ".\n";
   p3 = p3 + 1;                            // inkrementuje ukazatel
   cout << "Nyni je p3[0] " << p3[0] << " a ";
   cout << "p3[1] " << p3[1] << ".\n";
   p3 = p3 - 1;                            // ukazuje zpět na začátek
   delete [] p3;                           // uvolňuje paměť
   return 0;
}
```

Zde je výstup z programu na výpisu 4.18:

```
p3[1] je 0.5.
Nyni je p3[0] 0.5 a p3[1] 0.8.
```

Jak vidíte, v programu arraynew.cpp používáme ukazatel p3, jako by se jednalo o jméno pole, kde p3[0] představuje první prvek atd. Základní rozdíl mezi jménem pole a ukazatelem je zřejmý z následujícího řádku:

```
p3 = p3 + 1;       // v pořádku pro ukazatele, chybně pro jména polí
```

Hodnotu jména pole nelze měnit. Ale ukazatel je proměnná, proto její hodnotu změnit můžeme. Všimněte si výsledku přičtení hodnoty 1 k ukazateli p3. Výraz p3[0] nyní odkazuje na bývalý druhý prvek pole. Po přičtení hodnoty 1 tedy ukazatel p3 ukazuje místo na první prvek na druhý. Odečtení jedničky vrací ukazatel zpět na původní hodnotu, takže program může dodat operátoru delete [] správnou adresu.

Skutečná adresa za sebou jdoucích prvků typu `double` se obvykle liší o osm bajtů, takže si jistě říkáte, že pokud přičtením hodnoty 1 k ukazateli `p3` dostaneme adresu následujícího prvku, musí se aritmetika ukazatelů řídit zvláštními pravidly. Ano, je tomu tak.

Ukazatele, pole a aritmetika ukazatelů

Velká podobnost ukazatelů a jmen polí má svůj původ v aritmetice ukazatelů a ve vnitřním zpracování polí jazykem C++. Nejprve se podíváme na aritmetiku. Přičtení 1 k celočíselné proměnné zvyšuje její hodnotu o 1, ale přičtení 1 k ukazatelové proměnné zvyšuje její hodnotu o počet bajtů typu, na který ukazuje. Přičtení 1 k ukazateli na typ `double` přidává 8 k číselné hodnotě na systému s 8bajtovým `double`, zatímco přičtení 1 k ukazateli na typ `short` přidává k hodnotě ukazatele 2, pokud má `short` 2 bajty. Výpis 4.19 tuto zajímavou vlastnost předvádí. Ukazuje také další důležitou vlastnost: jazyk C++ interpretuje jméno pole jako adresu.

Výpis 4.19 addpntrs.cpp

```
// addpntrs.cpp -- přičítání k ukazatelům
#include <iostream>
int main()
{
   using namespace std;
   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
// s prvkem pole
   cout << "pw = " << pw << ", *pw = " << *pw << endl;
   pw = pw + 1;
   cout << "k ukazateli pw pricteme 1:\n";
   cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";

   cout << "ps = " << ps << ", *ps = " << *ps << endl;
   ps = ps + 1;
   cout << "k ukazateli ps pricteme 1:\n";
   cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";

   cout << " zpristupneni dvou prvku pomoci zapisu pole \n";
   cout << "stacks[0] = " << stacks[0]
        << ", stacks[1] = " << stacks[1] << endl;
   cout << " zpristupneni dvou prvku pomoci zapisu ukazatele \n";
   cout << "*stacks = " << *stacks
        << ", *(stacks + 1) = " << *(stacks + 1) << endl;

   cout << sizeof wages << " = velikost pole wages\n";
   cout << sizeof pw << " = velikost ukazatele pw\n";
   return 0;
}
```

Výstup z programu na výpisu 4.19:

```
pw = 0012FEBC, *pw = 10000
k ukazateli pw pricteme 1:
pw = 0012FEC4, *pw = 20000

ps = 0012FEAC, *ps = 3
k ukazateli ps pricteme 1:
ps = 0012FEAE, *ps = 2

zpristupneni dvou prvku pomoci zapisu pole
stacks[0] = 3, stacks[1] = 2
zpristupneni dvou prvku pomoci zapisu ukazatele
*stacks = 3, *(stacks + 1) = 2
24 = velikost pole wages
4 = velikost ukazatele pw
```

Poznámky k programu

Ve většině případů jazyk C++ interpretuje jméno pole jako adresu prvního prvku tohoto pole. Tedy příkaz

```
double * pw = wages;
```

deklaruje `pw` jako ukazatel na typ `double`, jenž poté inicializuje hodnotou `wages`, což je adresa prvního prvku pole `wages`. Pro `wages` i ostatní pole platí následující rovnost:

```
wages = &wages[0] = adresa prvního prvku pole
```

Program explicitně používá ve výrazu `&stack[0]` adresový operátor pro inicializaci ukazatele `ps` prvním prvkem pole `stacks` pouze proto, abychom si dokázali platnost výše uvedené rovnosti.

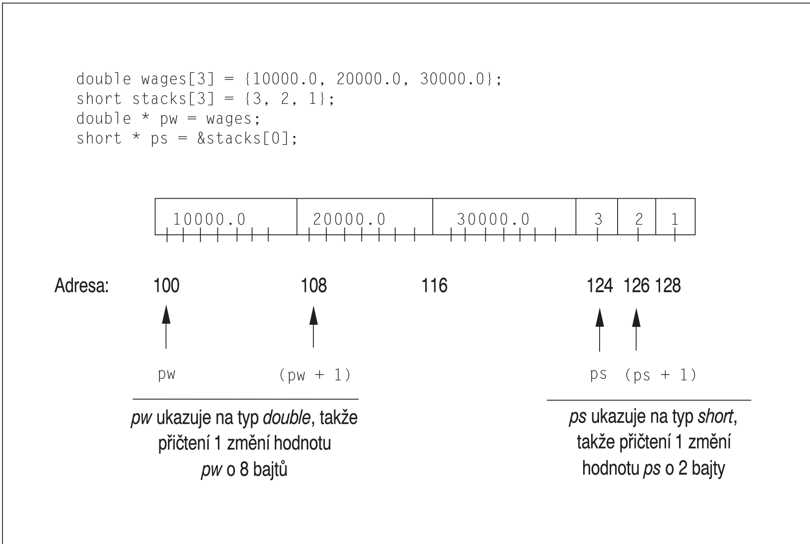
V programu si dále všímáme hodnot `pw` a `*pw`. První je adresa, druhá představuje hodnotu na této adrese uloženou. Protože `pw` ukazuje na první prvek, zobrazenou hodnotou `*pw` je hodnota prvního prvku 10 000. Potom program přičte k ukazateli `pw` hodnotu 1. Jak jsme si již říkali, to způsobí přičtení 8 (fd24 + 8 = fd2c hexadecimálně) k číselné hodnotě adresy. Po přičtení je hodnota ukazatele `pw` rovna adrese druhého prvku. Takže `*pw` má nyní hodnotu druhého prvku, která je 20 000. (Viz obrázek 4.10.) (Adresové hodnoty na obrázku jsou upravené, aby byl obrázek srozumitelnější.)

Poté program provádí obdobné kroky i s ukazatelem `ps`. Protože `ps` ukazuje na typ `short`, a ten má 2 bajty, přičtení 1 k tomuto ukazateli tentokrát zvýší jeho hodnotu o 2. Ukazatel `ps` pak také ukazuje na další prvek pole.

ZAPAMATUJTE SI

Přičtení hodnoty 1 k ukazatelové proměnné zvyšuje její hodnotu o počet bajtů typu, na který ukazuje.

Nyní se podíváme na výraz pole `stacks[1]`. Překladač jazyka C++ s ním pracuje naprosto stejně jako s výrazem `*(stacks + 1)`. Druhý výraz říká, vypočítejte adresu druhého prvku pole a potom zjistěte hodnotu, která je tam uložena. Konečný výsledek je stejný jako u `stacks[1]`. (Priorita operátorů vyžaduje, abyste použili závorky. Bez nich by byla hodnota 1 přičtena k `*stacks` namísto k `stacks`.)



Obrázek 4.10 Sčítání ukazatelů

Výstup programu dokazuje, že výrazy `*(stacks + 1)` a `stacks[1]` jsou stejné. Obdobně jsou stejné výrazy `*(stacks + 2)` a `stacks[2]`. Obecně platí, kdykoli použijete zápis pole, C++ vytvoří následující konverzi:

```
jmenopole[i] se stává *(jmenopole + i)
```

A pokud použijete místo jména pole ukazatel, jazyk C++ vytvoří stejnou konverzi:

```
jmenoukazatele[i] se stává *(jmenoukazatele + 1)
```

V mnoha případech tedy můžete používat jména ukazatelů a jména polí stejným způsobem. S oběma můžete používat zápis pole s hranatými závorkami i dereferenční operátor (`*`). Ve většině výrazů představují adresu. Jedním rozdílem je to, že hodnotu ukazatele můžete měnit, zatímco jméno pole je konstanta:

```
jmenoukazatele = jmenoukazatele + 1;  // platné
jmenopole = jmenopole + 1;            // neplatné
```

Druhým rozdílem je, že aplikace operátoru sizeof na jméno pole vrací velikost pole, ale aplikace sizeof na ukazatel vrací velikost ukazatele, dokonce i když ukazuje na pole. Například ve výpisu 4.19 se jak `pw` tak `wages` odkazují na stejné pole. Ale použijeme-li operátor `sizeof`, dostaneme následující výsledky:

```
24 = velikost pole wages             // zobrazení velikosti wages
4 = velikost ukazatele pw            // zobrazení velikosti pw
```

To je jeden případ, kdy jazyk C++ neinterpretuje jméno pole jako adresu.

Stručně řečeno, použití operátoru `new` pro vytvoření pole a následné přistupování k různým prvkům pomocí ukazatele je jednoduché. Stačí pouze pracovat s ukazatelem stejně jako se jménem pole. Avšak pochopení, proč to funguje, je trochu složitější. Chcete-li opravdu rozumět polím a ukazatelům, měli byste se ujistit, že znáte jejich vzájemné vztahy. Proto si ještě jednou zopakujeme, co jsme se o polích a ukazatelích dozvěděli doposud.

Shrnutí vlastností ukazatelů

Konec předchozího textu obsahoval velké množství informací, takže si látku stručně shrneme.

Deklarace ukazatelů

Pro deklaraci ukazatele určitého typu používáme následujíc tvar:

```
jmenoTypu * jmenoUkazatele
```

Zde je několik příkladů:

```
double * pn;            // pn ukazuje na hodnotu double
char * pc;              // pc ukazuje na hodnotu char
```

Zde představují proměnné `pn` a `pc` ukazatele a výrazy `double *` a `char *` jsou zápisy typů ukazatel na `double` a ukazatel na `char` v jazyce C++.

Přiřazování hodnot ukazatelům

Ukazatelům přiřazujeme paměťové adresy. Můžeme použít operátor `&` na jméno proměnné pro získání adresy pojmenované paměti. Operátor `new` vrací adresu nepojmenované paměti.

Zde je několik příkladů:

```
double * pn;            // pn může ukazovat na hodnotu double
double * pa;            // pa také
char * pc;              // pc může ukazovat na hodnotu char
double bubble = 3.2;
pn = &bubble;           // přiřaď adresu bubble do pn
pc = new char;          // přiřaď adresu nově alokované paměti char do pc
pa = new double[30];    // přiřaď adresu pole 30 double do pa
```

Dereferencování ukazatelů

Dereferencováním ukazatele se odkazujeme na hodnotu, na kterou ukazatel ukazuje. Ukazatele resp. nepřímou hodnotu dereferencujeme pomocí operátoru *. Pokud je `pn` ukazatelem na proměnnou `bubble`, jako v posledním příkladu, potom `*pn` představuje hodnotu, v tomto případě 3.2, na kterou ukazatel `pn` ukazuje.

Zde je několik příkladů:

```
cout << *pn;            // tiskne hodnotu proměnné bubble
*pc = 'S';              // uloží znak 'S' na místo v paměti, jehož adresa je pc
```

Zápis polí představuje druhý způsob dereference ukazatelů, například zápis `pn[0]` je stejný jako `*pn`. Nikdy nesmíme dereferencovat ukazatel, který nebyl inicializován řádnou adresou.

Rozlišování mezi ukazatelem a hodnotou, na kterou ukazatel ukazuje

Nezapomeňte, že pokud je `pt` ukazatel na typ `int`, potom výraz `*pt` nepředstavuje ukazatel na `int`, ale úplný ekvivalent proměnné typu `int`.

Zde je několik příkladů:

```
int * pt = new int;     // přiřadí adresu ukazateli pt
*pt = 5;                // uloží hodnotu 5 na tuto adresu
```

Jména polí

Ve většině případů pracuje jazyk C++ se jménem pole jako s adresou prvního prvku pole. Zde je příklad:

```
int tacos[10];  // tacos je to samé jako &tacos[0]
```

Výjimkou je použití jména pole s operátorem `sizeof`. V tomto případě `sizeof` vrací velikost celého pole v bajtech.

Aritmetika ukazatelů

Jazyk C++ dovoluje přičítat k ukazatelům celá čísla. Výsledkem přičtení hodnoty 1 je původní hodnota adresy plus počet bajtů objektu, na který ukazatel ukazuje. Také můžeme od ukazatelů celá čísla odečítat a zjišťovat tak rozdíl mezi dvěma ukazateli. Naposledy popsaná operace, která vrací celé číslo, má význam pouze tehdy, když oba ukazatele ukazují do stejného pole (ukazovat na jednu pozici za koncem je také povoleno); tímto způsobem můžeme získat rozestup mezi dvěma prvky.

Zde je několik příkladů:

```
int tacos[10] = {5,2,8,4,1,2,2,4,6,8};
int * pt = tacos;           // předpokládejme, že pt i tacos mají adresu 3000
pt = pt + 1;                // nyní je hodnota pt 3004, pokud má int čtyři
bajty
int *pe = &tacos[9];        // hodnota pe je 3036, pokud má int čtyři bajty
pe = pe -1;                 // nyní je hodnota pe 3032, adresa tacos[8]
int diff = pe - pt;         // rozdíl je 7, rozestup mezi prvky
                            // tacos[8] a tacos[1]
```

Dynamická a statická vazba polí

Deklarací vytváříme pole se statickou vazbou, to znamená pole, jejichž velikost je určena v době překladu:

```
int tacos[10];  // statická vazba, velikost určena v době překladu
```

Pomocí operátoru `new []` vytváříme pole s dynamickou vazbou (dynamické pole), to znamená pole, která jsou alokována a jejichž velikost může být určena za běhu programu. Paměť nepotřebných polí uvolňujeme prostřednictvím operátoru `delete []`:

```
int size;
cin >> size;
int * pz = new int [size];  // dynamická vazba, velikost pole určena za běhu
...
delete [] pz;               // uvolnění nepotřebné paměti
```

Zápis polí a ukazatelů

Použití zápisu pole s hranatými závorkami je ekvivalentní dereferencování ukazatele:

```
tacos[0] je stejné jako *tacos a znamená hodnotu na adrese tacos
tacos[3] je stejné jako *(tacos + 3) a znamená hodnotu na adrese tacos + 3
```

To platí pro jména polí i pro ukazatelové proměnné, takže na ukazatele a jména polí můžete použít zápis ukazatelů i zápis polí.

Zde je několik příkladů:

```
int * pt = new int [10];    // pt ukazuje na blok 10 int
*pt = 5;                    // nastavení nulového prvku na 5
```

```
pt[0] = 6;                      // přenastavení nulového prvku na 6
pt[9] = 44;                     // nastavení desátého prvku na 44
int coats[10];
*(coats + 4) = 12;              // nastavení prvku coats[4] na 12
```

Ukazatele a řetězce

Zvláštní vztah mezi poli a ukazateli se vztahuje i na řetězce. Podívejme se na následující kód:

```
char flower[10] = "ruze";
cout << flower << " jsou cervene\n";
```

Jméno pole představuje adresa jeho prvního 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 nacházející se na této adrese, a potom pokračuje tisknutím znaků, dokud nenarazí na nulový znak (`\0`). Zkrátka, pokud předáte objektu `cout` adresu znaku, vytiskne vše od předaného znaku až po první následující nulový znak.

Není důležité, že proměnná `flower` představuje jméno pole, ale že se chová jako adresa typu `char`. To znamená, že ukazatel na proměnnou typu `char` můžeme používat jako parametr objektu `cout`, protože je také adresou typu `char`. Ukazatel by měl samozřejmě ukazovat na začátek řetězce. Vyzkoušíme si to za chvíli.

Ale nejprve si vysvětlíme, co znamená závěrečná část výše uvedeného příkazu `cout`. Je-li `flower` skutečně adresou prvního znaku řetězce, co je výraz `" jsou cervene\n"`? Abychom zachovali shodu v řízení výstupu řetězce objektem `cout`, měl by být tento řetězec v uvozovkách také adresou. A on také je, protože řetězec v uvozovkách v jazyce C++, podobně jako jméno pole, reprezentuje adresa jeho prvního prvku. Předcházející kód ve skutečnosti neposílá do objektu `cout` celý řetězec, ale pouze adresu řetězce. To znamená, že řetězce v poli, řetězcové konstanty v uvozovkách a řetězce popsané pomocí ukazatelů, jsou zpracovány stejně. Každý z těchto druhů řetězců je ve skutečnosti předáván jako adresa. To zcela určitě představuje méně práce než předávání všech znaků v řetězci.

ZAPAMATUJTE SI

Pro objekt `cout` a většinu výrazů jazyka C++ představují jména polí typů `char`, ukazatele na `char` a řetězcové konstanty v uvozovkách adresy prvního znaku řetězce.

Výpis 4.20 ukazuje použití různých tvarů řetězců. Volá dvě funkce z řetězcové knihovny. Funkce `strlen()`, kterou jsme již používali, vrací délku řetězce. Funkce `strcpy()` kopíruje řetězec z jednoho místa na druhé. Obě mají funkční prototypy v hlavičkovém souboru `cstring` (nebo `string.h` ve starších implementacích). Program také předvádí některá nesprávná použití ukazatelů, kterým byste se měli vyhnout.

Výpis 4.20 ptrstr.cpp

```
// ptrstr.cpp -- použití ukazatelů na řetězce
#include <iostream>
#include <cstring>                  // deklaruje strlen(), strcpy()
int main()
{
    using namespace std;
    char animal[20] = "medved";     // animal obsahuje medved
```

```
    const char * bird = "strizlik";    // bird obsahuje adresu řetězce
    char * ps;                         // neinicializováno

    cout << animal << " a ";           // zobrazí medved
    cout << bird << "\n";              // zobrazí strizlik
    // cout << ps << "\n";          // může zobrazit nesmysl nebo způsobit pád
                                       programu

    cout << "Zadejte druh zvirete: ";
    cin >> animal;                     // v pořádku pokud je vstup kratší než
                                          20 znaků
    // cin >> ps; Velmi hrubá chyba;
    // ps neukazuje na alokovaný prostor

    ps = animal;                       // nastaví ps, aby ukazoval na řetězec
    cout << ps << "!\n";               // v pořádku, stejné jako použití animal
    cout << "Pred pouzitim strcpy():\n";
    cout << animal << " je na adrese " << (int *) animal << endl;
    cout << ps << " je na adrese " << (int *) ps << endl;

    ps = new char[strlen(animal) + 1]; // získá novou paměť
    strcpy(ps, animal); // kopíruje řetězec do nové paměti
    cout << "Po pouziti strcpy():\n";
    cout << animal << " je na adrese " << (int *) animal << endl;
    cout << ps << " je na adrese " << (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 příklad běhu programu z výpisu 4.20:

```
medved a strizlik
Zadejte druh zvirete: liska
liska!
Pred pouzitim strcpy():
liska je na adrese 0x0065fd30
liska je na adrese 0x0065fd30
Po pouziti strcpy():
liska je na adrese 0x0065fd30
liska je na adrese 0x004301c8
```

Poznámky k programu

Program na výpisu 4.20 vytváří jedno pole typů `char` (`animal`) a dvě proměnné typu ukazatel na `char` (`bird` a `ps`). Program začíná inicializací pole animal řetězcem „`medved`", stejně jako jsme je inicializovali dříve. Potom program předvádí něco nového, inicializuje ukazatel na `char` řetězcem:

```
const char * bird = "strizlik";       // bird obsahuje adresu řetězce
```

Pamatujte si, že „strizlik" ve skutečnosti představuje adresu řetězce, takže tento příkaz přiřazuje adresu řetězce „strizlik" ukazateli bird. (Překladač si obvykle rezervuje oblast v paměti na uložení všech řetězců v uvozovkách použitých ve zdrojovém kódu programu a každému uloženému řetězci přiřadí adresu.) To znamená, že můžete použít ukazatel bird stejně, jako byste použili řetězec „strizlik", například takto

```
cout << "Znepokojeny " << bird << " mluvi\n".
```

Řetězce v uvozovkách jsou konstanty, a proto programový kód používá v deklaraci klíčové slovo const. Použití const v této podobě znamená, že můžete pomocí ukazatele bird k řetězci přistupovat, ale nemůžete ho měnit. V kapitole 7 se budeme tématu konstantních ukazatelů věnovat mnohem podrobněji. Ukazatel ps zůstává neinicializován, takže neukazuje na žádný řetězec. (Neinicializovaný ukazatel, jak si jistě vzpomenete, může způsobit problémy a náš příklad není výjimkou.)

Dále program ukazuje, že s objektem cout můžete použít stejným způsobem jméno pole animal i ukazatel bird. Oba jsou přece adresami řetězců a objekt cout zobrazuje řetězce („medved" a „strizlik") na těchto adresách uložené. Pokud odstraněním znaků komentáře zpřístupníte kód, který se chybně pokouší zobrazit ukazatel ps, můžete dostat prázdný řádek, nesmyslný výpis nebo můžete způsobit pád programu. Vytvoření neinicializovaného ukazatele je něco jako předání nevyplněného 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. Pole animal můžeme bezpečně použít pro vstup pouze pokud se vstupní řetězec do tohoto pole vejde. Avšak použít pro vstupní data ukazatel bird by nebylo správné:

- Některé překladače považují řetězce v uvozovkách za konstanty určené pouze ke čtení a pokus přepsat je novými daty vede k chybě za běhu programu. Konstantní řetězce v uvozovkách předepisuje jazyk C++, ale všechny překladače se takto ještě nechovají.
- Některé překladače používají pro uložení všech řetězců v uvozovkách obsažených v programu pouze jednu kopii řetězce.

Druhý bod si vysvětlíme podrobněji. Jazyk C++ nezaručuje jedinečné uložení řetězců v uvozovkách. To znamená, že pokud použijeme řetězec „strizlik" v programu několikrát, překladač může uložit několik kopií tohoto řetězce nebo pouze jednu. Nastane-li druhý případ, potom ukazatel bird nastavený na řetězec „strizlik" ukazuje pouze na jedinou kopii tohoto řetězce. Načtení hodnoty do jednoho řetězce by mohlo ovlivnit i jiný řetězec, který se zdá být zcela jiným nezávislým řetězcem. Protože je ukazatel bird deklarován jako konstanta, překladač nedovolí měnit obsah paměti, na kterou bird ukazuje.

Ještě horší je snaha o načtení informací na místo označené ukazatelem ps. Protože ukazatel ps není inicializován, nemůžeme vědět, kam se informace uloží. Takto dokonce můžeme přepsat informaci již v paměti uloženou. Naštěstí je snadné se těmto problémům vyhnout, musíme pouze použít dostatečně velké pole typů char pro přijetí vstupních dat. Anebo se můžete všem těmto problémům vyhnout tak, že místo polí použijete objekty std::string.

UPOZORNĚNÍ

Když do programu načítáme řetězec, je nutno použít adresu paměti, která už byla dříve přidělena. Adresa může mít podobu jména pole nebo ukazatele inicializovaného pomocí `new`.

Dále se podíváme, co dělá následující kód:

```
ps = animal;                    // nastaví ps tak, aby ukazoval na řetězec
...
cout << animal << " je na adrese " << (int *) animal << endl;
cout << ps << " je na adrese " << (int *) ps << endl;
```

Vytváří následující výstup:

```
liska je na adrese 0x0065fd30
liska je na adrese 0x0065fd30
```

Když objektu `cout` předáte ukazatel, obvykle vytiskne jeho adresu. Ale pokud má ukazatel typ `char *`, `cout` zobrazí řetězec, na který tento ukazatel ukazuje. Chcete-li vidět adresu řetězce, musíte ukazatel přetypovat například `na int *`, což právě výše uvedený programový kód dělá. Objekt `cout` tedy zobrazuje `ps` jako řetězec „`liska`“, ale (`int *`) `ps` jako adresu, na které tento řetězec leží. Všimněte si, že přiřazení `animal` do `ps` nekopíruje řetězec, ale adresu. Výsledkem jsou dva ukazatele (`animal` a `ps`) na stejné místo v paměti a řetězec.

Pro vytvoření kopie řetězce toho musíte udělat více. Nejprve je třeba alokovat paměť pro uložení řetězce. To můžete udělat deklarací dalšího pole nebo pomocí klíčového slova `new`. Druhý přístup umožňuje přizpůsobení paměti řetězci:

```
ps = new char[strlen(animal) + 1]; // získá novou pamě_
```

Řetězec „`liska`“ nevyužívá celé pole `animal`, takže plýtváme pamětí. Tento kousek programového kódu zjišťuje délku řetězce pomocí funkce `strlen()` a přičítá 1 pro nulový znak. Potom alokuje dostatečné množství paměti pro řetězec pomocí `new`.

Dále potřebujeme zkopírovat řetězec z pole `animal` do nově alokovaného prostoru. Přiřazení `animal` do `ps` nefunguje, protože tak pouze změníme adresu v `ps` uloženou a ztratíme jediný způsob přístupu nově alokované paměti, kterou program měl. Správným řešením je použití knihovní funkce `strcpy()`:

```
strcpy(ps, animal);             // kopíruje řetězec do nové paměti
```

Funkce `strcpy()` má dva argumenty. Prvním je cílová adresa a druhým adresa řetězce, který se má zkopírovat. Je na vás, abyste zajistili, že cíl byl skutečně alokován a má dostatek místa pro uložení kopie. Toho je v příkladu dosaženo pomocí funkce `strlen()`, která zjišťuje správnou velikost klíčového slova new pro alokování volné paměti.

```
liska je na adrese 0x0065fd30
liska je na adrese 0x004301c8
```

Také si všimněte, že operátor `new` alokoval nové úložiště v paměti na místě dosti vzdáleném od pole `animal`.

S potřebou umístit řetězec do paměti se setkáte poměrně často. Při inicializaci pole použijte operátor =; v ostatních případech funkci `strcpy()` nebo `strncpy()`. Funkci `strcpy()` jste již viděli v podobném kódu:

```
char food[20] = "mrkve";        // inicializace
strcpy(food, "ovocny kolac");   // ostatní případy
```

Zapamatujte si, že takovýto kód:

```
strcpy(food, "piknikovy kosik naplneny dobrotami");
```

může způsobit problémy, protože pole `food` je menší než řetězec. V takovém případě funkce zkopíruje zbytek řetězce do paměťových bajtů následujících bezprostředně za polem, čímž může přepsat jinou paměť používanou vaším programem. Tomuto problému se můžete vyhnout pomocí funkce `strncpy()`. Třetí argument této funkce představuje maximální počet kopírovaných znaků. Mějte ale na paměti, že pokud je hodnota maximálního počtu kopírovaných znaků menší nebo rovna délce kopírovaného řetězce, funkce nepřidá nulový znak. Proto byste ji měli používat následovně:

```
strncpy(food, "piknikovy kosik naplneny dobrotami", 19);
food[19] = '\0';
```

Tento kód zkopíruje 19 znaků do pole a potom poslednímu prvku nastaví nulový znak. Je-li řetězec kratší než 19 znaků, funkce `strncpy()` vloží nulový znak označující správný konec řetězce dříve.

ZAPAMATUJTE SI

Pro přiřazení řetězce do pole používejte funkce `strcpy()` a `strncpy()`, nikoli operátor přiřazení.

Nyní, když jste se seznámili s tím, jak se používají řetězce typu C a s knihovnou `cstring`, jistě oceníte, jak jednoduchá je práce s řetězci typu C++. Nemusíte se (většinou) starat o přetečení pole a místo `strcpy()` nebo `strncpy()` se používá přiřazovací příkaz.

Vytváření dynamických struktur pomocí new

Viděli jste, proč může být výhodnější vytvářet pole za běhu programu místo při překladu. To samé platí i pro struktury. Je nutné alokovat prostor pouze pro tolik struktur, kolik jich program za určitého běhu potřebuje. K těmto účelům opět používáme operátor `new`. S jeho pomocí můžeme vytvářet dynamické struktury. Pro zopakování, slovem *dynamická* označujeme paměť alokovanou za běhu programu, nikoli během překladu. Mimochodem, protože třídy jsou velmi podobné strukturám, budeme moci zde naučené postupy se strukturami využít také u tříd.

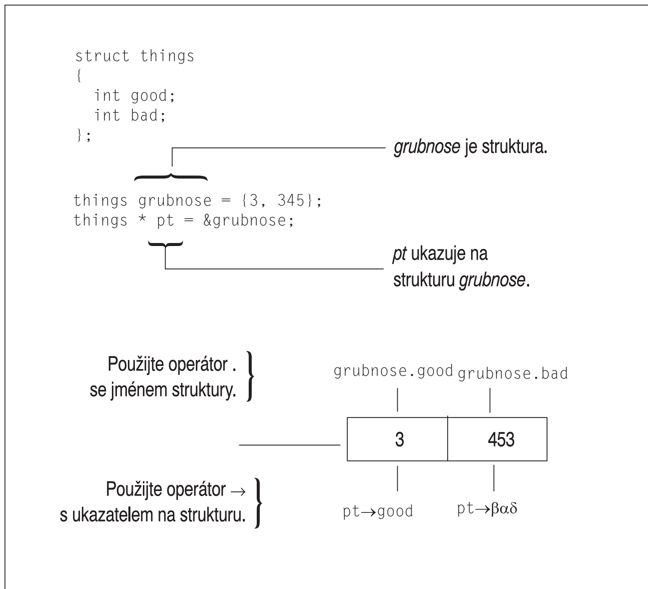
Použití operátoru `new` se strukturami má dvě části: vytvoření struktury a přístup k jejím členům. Při vytváření struktury napíšeme typ struktury za klíčové slovo `new`. Například pro vytvoření nepojmenované struktury typu `inflatable` a přiřazení její adresy vhodnému ukazateli můžeme napsat následující kód:

```
inflatable * ps = new inflatable;
```

Ten přiřadí ukazateli `ps` adresu dostatečně velké paměti pro uložení struktury typu `inflatable`. Všimněte si, že uvedená syntaxe je přesně stejná jako pro vestavěné typy jazyka C++.

Přístup ke členům je složitější. Při vytváření dynamické struktury nemůžeme použít členský operátor tečka se jménem struktury, protože tato struktura nemá jméno. Známe pouze její adresu. Jazyk C++ má právě pro tyto případy členský operátor šipka (->). Tento operátor, který se skládá ze symbolu pomlčka a větší než, představuje pro ukazatele na struktury totéž, co tečka pro jména struktur. Ukazuje-li například `ps` na strukturu typu `infla-`

`table`, potom výraz `ps->price` zastupuje člen `price` struktury, na kterou se ukazuje. (Viz obrázek 4.11.)



Obrázek 4.11 Identifikace členů struktury

ZAPAMATUJTE SI

Občas bývají noví uživatelé C++ na pochybách, kdy mají pro určení členu struktury použít operátor tečka a kdy operátor šipka. Pravidlo je jednoduché. Je-li identifikátorem struktury její jméno, použijte operátor tečka. Je-li identifikátorem ukazatel na strukturu, použijte operátor šipka.

Druhý, méně pěkný přístup můžeme použít, když si uvědomíme, že pokud je `ps` ukazatelem na strukturu, potom `*ps` představuje hodnotu, na kterou tento ukazatel ukazuje, tedy vlastní strukturu. Protože `*ps` reprezentuje strukturu, `(*ps).price` představuje člen struktury `price`. Pravidla priority v C++ vyžadují v této konstrukci použití závorek.

Ve výpise 4.21 si předvedeme vytvoření nepojmenované struktury pomocí operátoru `new` a oba zápisy s ukazateli pro přístup ke členům struktury.

Výpis 4.21 newstrct.cpp

```
// newstrct.cpp -- použití new se strukturou
#include <iostream>
struct inflatable     // šablona struktury
{
char name[20];
float volume;
double price;
};
int main()
{
```

```
    using namespace std;
    inflatable * ps = new inflatable;     // alokuje prostor pro strukturu
    cout << "Zadejte jmeno nafukovaciho predmetu: ";
    cin.get(ps->name, 20);                // metoda 1 pro přístup ke členu
    cout << "Zadejte objem v litrech: ";
    cin >> (*ps).volume;                  // metoda 2 pro přístup ke členu
    cout << "Zadejte cenu v Kc: ";
    cin >> ps->price;
    cout << "Jmeno: " << (*ps).name << endl;        // metoda 2
    cout << "Objem: " << ps->volume << " litru\n"; // metoda 1
    cout << "Cena: " << ps->price << endl;          // metoda 1
    delete ps;                            // uvolnění paměti použité strukturou
    return 0;
}
```

Zde je ukázka běhu programu uvedeného na výpisu 4.21:

```
Zadejte jmeno nafukovaciho predmetu: Bajecny Frodo
Zadejte objem v litrech: 1.4
Zadejte cenu v Kc: 179
Jmeno: Bajecny Frodo
Objem: 1.4 litru
Cena: 179 Kc
```

Příklad použití operátoru new a delete

Podívejme se na příklad demonstrující ukládání vstupu řetězců z klávesnice pomocí operátorů `new` a `delete`. Výpis 4.17 definuje funkci, která vrací ukazatel na vstupní řetězec. Tato funkce načítá vstup do velkého dočasného pole a potom alokuje operátorem `new []` paměť podle velikosti vstupního řetězce. Nakonec funkce vrací ukazatel na alokovanou paměť. Tento přístup by mohl ušetřit spoustu paměti programům, které načítají velké množství řetězců.

Předpokládejme, že náš program musí načíst 1000 řetězců a že největší řetězec může být 79 znaků dlouhý, ale většina řetězců je mnohem kratších. Pro ukládání řetězců pomocí znakových polí typu `char` bychom potřebovali 1000 polí po 80 znacích. To znamená 80 tisíc bajtů a většina této paměti by zůstala nevyužitá. Lepším řešením je vytvoření pole o 1000 ukazatelích na `char` a alokování potřebné paměti pro každý řetězec operátorem `new`. Tak můžeme ušetřit desítky tisíc bajtů. Místo použití velkého pole pro každý řetězec přizpůsobíme paměť vstupním datům. Navíc můžeme pomocí `new` vytvořit prostor pouze pro potřebné množství ukazatelů. Nyní by bylo toto řešení příliš náročné. I jednodušší použití pole o 1000 ukazatelích je prozatím trošku předčasné, ale výpis 4.22 předvádí některé techniky. Příklad také na uvolňování paměti pro opětovné použití předvádí práci operátoru `delete`.

Výpis 4.22 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 nealokuje paměť

    name = getname();         // přiřazuje adresu řetězce do name
    cout << name << " na adrese " << (int *) name << "\n";
    delete [] name;           // uvolnil paměť

    name = getname();         // nové použití uvolněné paměti
    cout << name << " na adrese " << (int *) name << "\n";
    delete [] name;           // opět uvolnil paměť
    return 0;
}

char * getname()              // vrací ukazatel novému řetězci
{
    char temp[80];            // dočasná paměť
    cout << "Zadejte prijmeni: ";
    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 na výpisu 4.22:

```
Zadejte prijmeni: Fredeldumpkin
Fredeldumpkin na adrese 0x004326b8
Zadejte prijmeni: Pook
Pook na adrese 0x004301c8
```

Poznámky k programu

Nejprve se podíváme na funkci `getname()`. Pro umístění vstupního řetězce do pole `temp` používá objekt `cin`. Potom alokuje novou paměť pro uložení tohoto řetězce pomocí operátoru `new`. Program potřebuje `strlen(temp) + 1` znaků pro uložení řetězce včetně nulového znaku, což je hodnota předávaná operátoru `new`. Po zpřístupnění paměti funkce `getname()` kopíruje řetězec z pole `temp` do nového bloku paměti standardní knihovní funkci `strcpy()`. Funkce `getname()`nekontroluje, zda se řetězec vejde nebo ne, ale místo toho požaduje přesný počet bajtů od operátoru `new`. Nakonec funkce vrací v ukazateli `pn` adresu kopie řetězce.

Ve funkci `main()` je návratová hodnota (adresa) přiřazena ukazateli `name`. Tento ukazatel je definován ve funkci `main()`, ale ukazuje na blok paměti alokované ve funkci `getname()`. Program potom tiskne řetězec a jeho adresu.

Dále, po uvolnění bloku, na který `name` ukazuje, `volá main()` funkci `getname()` podruhé. Jazyk C++ nezaručuje, že nově uvolněná paměť bude při dalším použití operátoru `new` vybrána jako první a ani v této ukázce běhu programu tomu tak není.

Všimněme si v tomto příkladu, že `getname()` paměť přiděluje a `main()` ji uvolňuje. Obvykle není vhodné dávat `new` a `delete` do různých funkcí, neboť tak snáze zapomeneme na `delete`. V tomto příkladě to tak máme jenom proto, aby bylo vidět, že je to možné.

Abyste ocenili některé důmyslnější vlastnosti programu, musíte vědět trochu více o tom, jak C++ pracuje s pamětí. Proto si nyní ukážeme přehled některých pojmů, které jsou probrány podrobněji v kapitole 9.

Automatické, statické a volné úložiště

Jazyk C++ podporuje při práci s daty tři způsoby správy paměti, jež jsou závislé na použitém způsobu alokace paměti: *automatické*, *statické* a *volné úložiště*, kterému také říkáme *volné úložiště* nebo *hromada* (*heap*). Datové objekty alokované těmito třemi způsoby se liší délkou existence. Nyní se na každý typ v rychlosti podíváme.

Automatické úložiště

Běžným proměnným definovaným uvnitř funkce říkáme *automatické proměnné*. Vznikají automaticky při vyvolání funkce, která je obsahuje a jejich platnost vyprší s ukončením funkce. Například pole `temp` ve výpisu 4.22 existuje pouze, pokud je funkce `getname()` aktivní. Při vrácení řízení funkci `main()` je použitá paměť pole `temp` automaticky uvolněna. Kdyby funkce `getname()` vrátila adresu pole `temp`, ukazatel `name` by ve funkci `main()` ukazoval na paměť, která by mohla být brzy použita znovu k jiným účelům. To je jeden z důvodů, proč jsme ve funkci `getname()` museli použít operátor `new`.

Automatické hodnoty jsou platné pouze v bloku, který je obsahuje. Blok představuje úsek kódu uzavřený mezi složenými závorkami. Doposud byly našimi bloky jen celé funkce. Ale jak uvidíte v následující kapitole, bloky se mohou vyskytovat i uvnitř funkcí. Proměnná definovaná uvnitř některého z těchto bloků existuje pouze, když program provádí příkazy uvnitř bloku.

Statické úložiště

Statické úložiště je platné po dobu provádění celého programu. Existují dva způsoby vytvoření statické proměnné. Jedním je vnější definice vně funkce a druhým použití klíčového slova `static` v deklaraci proměnné:

```
static double fee = 56.50;
```

V jazyce C podle K&R můžeme pouze inicializovat statická pole a struktury, zatímco jazyk C++ verze 2.0 (a pozdější) a ANSI C nám dovolují inicializovat také automatická pole a struktury. Avšak, jak již někteří z vás možná zjistili, některé implementace jazyka C++ inicializaci automatických polí a struktur ještě nepodporují.

V kapitole 9 se o statickém úložišti dozvíte další podrobnosti. Důležité je, že automatická a statická úložiště pevně definují dobu života proměnné. Proměnná buďto existuje během celého trvání programu (statická proměnná) nebo při vykonávání určité funkce (automatická proměnná).

Dynamické úložiště

Operátory `new` a `delete` však poskytují mnohem pružnější přístup. Řídí paměť, kterou jazyk C++ popisuje jako volnou paměť. Tato volná paměť je oddělená od paměti používané pro statické a automatické proměnné. Jak ukazuje výpis 4.22, operátory `new` a `delete` umožňují alokaci paměti v jedné funkci a její uvolnění v jiné. Tedy doba života dat není svázána s životem programu nebo funkce. Operátory `new` a `delete` umožňují daleko lepší správu využití paměti než použití obvyklých proměnných.

Z praxe: Stack, heap a úniky paměti

Co se stane, když po vytvoření proměnné ve volném úložišti (`heap`) operátorem `new` *nezavoláte* operátor `delete`? Pokud není zavolán operátor `delete`, proměnná nebo konstrukce dynamicky alokovaná ve volném úložišti existuje dál, i když byla podle pravidel rozsahu platnosti a doby života objektů uvolněna paměť s odpovídajícím ukazatelem. Potom v podstatě nemáte žádnou možnost přístupu k vaší konstrukci ve volném úložišti, protože neexistuje ukazatel na paměť, která ji obsahuje. Tímto způsobem jste právě vytvořili ***únik paměti***. Uniklá paměť zůstává nevyužitá po celou dobu života programu; byla alokována a nemůže být uvolněna. V krajních (ale ne neobvyklých) případech mohou být úniky paměti tak vážné, že spotřebují veškerou aplikaci dostupnou paměť a způsobí její pád na chybu nedostatku paměti. Úniky paměti mohou mít navíc negativní vliv na některé operační systémy nebo ostatní aplikace běžící ve stejném paměťovém prostoru a způsobit i jejich selhání.

Úniky paměti se podaří vytvořit dokonce i nejlepším programátorům a softwarovým společnostem. Abyste se jim vyhnuli, bude nejlepší, když si zvyknete psát operátory `new` a `delete` zároveň, čímž si naplánujete a vložíte uvolnění vašich konstrukcí okamžitě po jejich dynamické alokaci ve volném úložišti.

POZNÁMKA

Ukazatele patří mezi nejmocnější nástroje jazyka C++. Jsou také nejnebezpečnější, protože umožňují provádět nekorektní akce, jako je například použití neinicializovaného ukazatele na přístup do paměti nebo pokus o uvolnění stejného bloku paměti dvakrát. Dokud si praktickými zkušenostmi neosvojíte zápis a logiku ukazatelů, mohou na vás působit poněkud zmateně. V této knize se budeme k ukazatelům několikrát vracet, abyste si na ně zvykli a jejich používání pro vás nebylo nepříjemné.

Shrnutí

Pole, struktury a ukazatele představují tři složené typy jazyka C++. Pole může v jednom datovém objektu uchovávat několik hodnot stejného typu. K jednotlivým prvkům pole přistupujeme pomocí indexu nebo ukazatele.

Struktura může v jednom datovém objektu ukládat několik hodnot různých typů. Pro přístup k jednotlivým prvkům struktury používáme operátor příslušnosti (.). Prvním krokem při používání struktur je vytvoření šablony struktury, která definuje členy struktury. Jméno, neboli značka, této šablony se potom stává novým identifikátorem typu. Potom můžeme deklarovat strukturní proměnné tohoto typu.

Union obsahuje pouze jedinou hodnotu, ale ta může mít různé typy, přičemž aktuální režim určuje jméno členu.

Ukazatele jsou proměnné navržené pro ukládání adres. Říkáme, že ukazatel ukazuje na adresu, kterou má v sobě uloženou. Deklarace ukazatele vždy stanoví, na který typ objektu ukazatel ukazuje. Dereferenční operátor (*) aplikovaný na ukazatel vrací hodnotu uloženou v místě, na které ukazatel ukazuje.

Řetězec je řada znaků ukončená nulovým znakem. Řetězec může reprezentovat řetězcová konstanta uzavřená do dvojitých uvozovek, v takovém případě je ukončovací nulový znak doplněn implicitně. Řetězec můžeme uložit do pole typu `char` a odkazovat se na něj pomocí ukazatele na `char` inicializovaného tak, aby na tento řetězec ukazoval. Funkce `strlen()` vrací délku řetězce bez nulového znaku. Funkce `strcpy()` kopíruje řetězec z jednoho místa na druhé. Chceme-li tyto funkce použít, musíme vložit hlavičkový soubor `cstring` nebo `string.h`.

Pro práci s řetězci v C++ nabízí třída `string`, společně s hlavičkovým souborem `string`, alternativu s uživatelsky přátelštějšími prostředky. Konkrétně se podle ukládaných řetězců automaticky mění velikost řetězcových objektů a pro kopírování řetězce lze použít přiřazovací operátor.

Operátor `new` umožňuje vyžádání paměti pro datový objekt za běhu programu. Tento operátor vrací adresu obdržené paměti, kterou můžeme přiřadit ukazateli. Jediným prostředkem pro přístup k takto získané paměti je ukazatel. Je-li datovým objektem jednoduchá proměnná, můžeme získat její hodnotu pomocí dereferenčního operátoru (`*`). Pokud je datovým objektem pole, můžeme k jeho jednotlivým prvkům přistupovat prostřednictvím ukazatele, jakoby se jednalo o jméno pole. Jestliže je datovým objektem struktura, můžeme pro přístup ke členům struktury použít ukazatelový dereferenční operátor (`->`).

Ukazatele a pole mají velmi těsnou vazbu. Pokud představuje `ar` jméno pole, potom je výraz `ar[i]` interpretován jako `*(ar + i)`, kde jméno pole vyjadřuje adresu prvního prvku pole. Jméno pole má tedy stejný význam jako ukazatel. Pro přístup k prvkům pole alokovaným operátorem `new` můžeme i obráceně použít jméno ukazatele se zápisem pro pole.

Operátory `new` a `delete` umožňují explicitní alokaci paměti pro datové objekty a její vrácení systému k dalšímu použití. Automatické proměnné, deklarované ve funkci, a statické proměnné, definované mimo funkci nebo pomocí klíčového slova `static`, jsou méně přizpůsobivé. Doba života automatické proměnné začíná vstupem do bloku, který ji obsahuje (obvykle definice funkce) a končí při opuštění bloku. Statická proměnná existuje po celou dobu trvání programu.

Otázky k opakování

1. Jak byste deklarovali následující proměnné?
   a) `actors` je pole o 30 prvcích typu `char`
   b) `betsie` je pole o 100 prvcích typu `short`
   c) `chuck` je pole o 13 prvcích typu `float`
   d) `dipsea` je pole o 64 prvcích typu `long double`
2. Deklarujte pole, které obsahuje pět prvků typu `int` a inicializujte ho prvními pěti kladnými lichými celými čísly.
3. Napište příkaz, který přiřadí součet prvního a posledního prvku pole v otázce 2 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 gramech a délku (`length`) v desetinách centimetrů.
7. Deklarujte proměnnou typu definovaného v otázce 6 a inicializujte ji.
8. Pomocí klíčového slova `enum` definujte typ `Response` s možnými hodnotami `Ano`, `Ne` a `Snad`. `Ano` by mělo být `1`, `Ne 0` a `Snad 2`.
9. Předpokládejte, že `ted` je proměnná typu `double`. Deklarujte ukazatel, který na proměnnou `ted` ukazuje a použijte ho pro zobrazení její hodnoty.
10. Předpokládejte, že `treacle` je pole o 10 prvcích typu `float`. Deklarujte ukazatel, který ukazuje na první prvek pole `treacle` a použijte ho pro zobrazení jeho prvního a posledního prvku tohoto pole.
11. Napište část programového kódu, který požádá uživatele o zadání kladného celého čísla a potom vytvořte dynamické pole s tolika prvky typu `int`.
12. Je následující programový kód platný? Jestliže ano, co vytiskne?

```
cout << (int *) "Domov veselych bajtu";
```

13. Napište část programového kódu, který dynamicky alokuje strukturu typu popsaného v otázce 6 a přečtěte hodnotu členu druh (kind) této struktury.
14. Výpis 4.6 předvádí problém číselného vstupu následovaného řádkově orientovaným vstupem řetězce. Jak by nahrazení příkazu

```
cin.getline(address, 80);
```

příkazem

```
cin >> address;
```

ovlivnilo činnost programu?

Programátorská cvičení

1. Napište program v C++, který požádá o informaci a zobrazí ji, jak je ukázáno v následujícím příkladu výstupu.

```
Jake je vase krestni jmeno? Betty Sue
Jake je vase prijmeni? Yew
Jakou znamku si zaslouzite? B
Kolik je vam let? 22
Jmeno: Yew, Betty Sue
Znamka: C
Vek: 22
```

Všimněte si, že by si program měl poradit s křestními jmény, která se skládají z více než jednoho slova. Také si všimněte, že program snižuje známku směrem dolů, což znamená o jedno písmeno nahoru. Předpokládejte, že uživatel požádá o známky A, B nebo C, takže si nemusíte dělat starosti s mezerou mezi D a F.

2. Přepište výpis 4.4., místo polí `char` použijte třídy `string` jazyka C++.
3. Napište program, který si vyžádá od uživatele křestní jméno a příjmení a poté vytvoří, uloží a vypíše třetí řetězec, který se bude skládat z příjmení, čárky, meze-

ry a křestního jména. Použijte k tomu pole `char` a funkce z hlavičkového souboru `cstring`. Výpis po spuštění by mohl vypadat například takto:

```
Zadejte krestni jmeno: Flip
Zadejte prijmeni: Fleming
Zde jsou oba udaje spojeny do jednoho retezce: Fleming, Flip
```

4. Napište program, který si vyžádá od uživatele křestní jméno a příjmení a poté vytvoří, uloží a vypíše třetí řetězec, který se bude skládat z příjmení, čárky, mezery a křestního jména. Použijte k tomu objekty `string` a metody z hlavičkového souboru `string`. Výpis po spuštění by mohl vypadat například takto:

```
Zadejte krestni jmeno: Flip
Zadejte prijmeni: Fleming
Zde jsou oba udaje spojeny do jednoho retezce: Fleming, Flip
```

5. Struktura `SladkaTycinka` má tři členy. První člen uchovává značku sladké tyčinky, druhý člen váhu (která může mít desetinnou část) a třetí člen počet kalorií (celé číslo). Napište program, který takovouto strukturu deklaruje a vytváří proměnnou typu `SladkaTycinka` nazvanou `svacina`. Inicializujte každý její člen postupně na „`Mocha Munch`“, `2.3` a `350`. Inicializace by měla být součástí deklarace proměnné `snack`. Nakonec by měl program obsah proměnné `svacina` vypsat.
6. Struktura `SladkaTycinka` obsahuje tři členy, jak je popsáno v programovém cvičení 5. Napište program, který vytvoří pole tří struktur `SladkaTycinka`, inicializuje je hodnotami podle vašeho výběru, a potom zobrazí obsah každé struktury.
7. William Wingate poskytuje službu analýzy pizz. Pro každou pizzu musí zaznamenat následující informaci:
   - Jméno společnosti vyrábějící pizzu, které se může skládat z více než jednoho slova
   - Průměr pizzy
   - Váha pizzy

   Navrhněte strukturu pro uložení požadovaných informací a proměnnou tohoto typu použijte v programu. Tento program by měl požádat uživatele o zadání všech informací a poté by je měl zobrazit. Použijte objekty `cin` (nebo jeho metody) a `cout`.
8. Proveďte programové cvičení 4, ale místo deklarace strukturní proměnné alokujte paměť pro strukturu pomocí operátoru `new`. Program by měl nejprve požádat o poloměr pizzy a potom o jméno společnosti, která pizzu vyrábí.
9. Proveďte programové cvičení 3, ale místo deklarace pole tří struktur `SladkaTycinka` alokujte paměť pro pole dynamicky pomocí operátoru `new`.

Cykly a relační výrazy

KAPITOLA

5

V této kapitole se naučíte:

- Cyklus for
- Výrazy a příkazy
- Operátory inkrementace a dekrementace: ++ a —
- Spojovat přiřazovací operátory
- Složené příkazy (bloky)
- Operátor čárka
- Relační operátory: >, >=, ==, <=, < a !=
- Cyklus while
- Příkaz typedef
- Cyklus do while
- Metodu vstupu znaku get()
- Podmínku konec souboru
- Vnořené cykly a dvojrozměrná pole

Počítače kromě ukládání dat provádějí i jiné činnosti. Analyzují, spojují, přeskupují, extrahují, mění, extrapolují, syntetizují a jiným způsobem manipulují s daty. Občas také data komolí a ztrácejí, ale takovémuto chování se budeme snažit vyhnout. Aby mohly programy provádět své manipulační zázraky, potřebují nástroje na vykonávání opakujících se činností a rozhodování. Jazyk C++ samozřejmě takové nástroje poskytuje. Ve skutečnosti používá stejné cykly for, while, do while, příkazy if a switch jako standardní jazyk C, takže pokud jazyk C znáte, můžete tuto a kapitolu 6, „Příkazy větvení a logické operátory" pouze prolistovat. (Ale nepřehlédněte popis toho, jak objekt `cin` zpracovává vstup znaků!) Tyto rozmanité řídicí příkazy programu často využívají pro rozhodování o svém chování relační a logické výrazy. Tato kapitola pojednává o cyklech a relačních operátorech a následující se zabývá příkazy větvení a logickými výrazy.

Úvod do cyklu for

Okolnosti často vyžadují, aby program prováděl opakující se úkoly, jako je například sečtení prvků pole jeden po druhém nebo vytištění textu opěvujícího produktivitu 20krát. Cyklus `for` jazyka C++ provádění takovýchto úkolů velmi zjednodušuje. Podívejme se na cyklus ve výpisu 5.1, nejprve zjistíme, co dělá a potom si vysvětlíme, jak pracuje.

Výpis 5.1 forloop.cpp

```
// forloop.cpp -- představení cyklu for
#include <iostream>
int main()
{
   using namespace std;
   int i;          // vytvoří počitadlo
// inicializace; test; aktualizace
   for (i = 0; i < 5; i++)
      cout << "C++ umi cykly.\n";
   cout << "C++ vi, kdy skoncit.\n";
   return 0;
}
```

Zde je výstup z programu na výpisu 5.1:

```
C++ umi cykly.
C++ umi cykly.
C++ umi cykly.
C++ umi cykly.
C++ umi cykly.
C++ vi, kdy skoncit.
```

Cyklus začíná nastavením celočíselné proměnné i na hodnotu 0:

```
i = 0
```

Tato část se nazývá *inicializace cyklu*. Potom v *testu cyklu* program zjišťuje, zda je hodnota proměnné i menší než 5:

```
i < 5
```

Je-li tomu tak, program provede následující příkaz, kterému říkáme *tělo cyklu*:

```
cout << "C++ umi cykly.\n";
```

Dále je prostřednictvím *aktualizace cyklu* zvýšena hodnota proměnné `i` o 1:

```
i++
```

Zde je použit operátor ++, kterému říkáme *operátor inkrementace*. Ten inkrementuje hodnotu svého operandu o 1. (Operátor inkrementace není vyhrazený pouze pro cykly `for`. Příkaz `i++;` můžeme například použít místo příkazu `i = i + 1;` v programu.) Inkrementace proměnné `i` zakončuje první průchod cyklem.

Další průchod cyklem začíná porovnáním nové hodnoty proměnné `i` s hodnotou 5. Protože je nová hodnota (1) také menší než 5, cyklus vytiskne další řádek a nakonec znovu inkrementuje proměnnou `i`. Tím jsou připraveny výchozí hodnoty pro nový cyklus testování, provedení příkazu a aktualizaci hodnoty proměnné `i`. Proces pokračuje, dokud cyklus nepřiřadí proměnné `i` hodnotu 5. Následující test selže a program předá řízení dalšímu příkazu za cyklem.

Části cyklu for

Cyklus `for` tedy poskytuje popis kroků provádění opakovaných činností. Podívejme se nyní podrobněji, jak pracuje. Obvyklé části cyklu for obsluhují tyto kroky:

1. Nastavení počáteční hodnoty.
2. Provedení testu, který rozhodne, zda má cyklus pokračovat.
3. Vykonání příkazů cyklu.
4. Aktualizace hodnoty (hodnot) použitých v testu.

Návrh tohoto cyklu v jazyce C++ umisťuje zmíněné prvky tak, že jsou na první pohled zřejmé. Inicializace, test a aktualizace tvoří tři části řídicí sekce, která je uzavřena do závorek. Každá část představuje výraz a jednotlivé výrazy jsou od sebe odděleny středníky. Příkaz umístěný za řídicí sekcí se nazývá *tělo* cyklu a provádí se tak dlouho, dokud je testovací výraz pravdivý:

```
for (inicializace; testovací-výraz; aktualizační-výraz)
    tělo
```

Syntaxe jazyka C++ považuje celý cyklus `for` za jeden příkaz, i když tělo může obsahovat jeden nebo více příkazů. (Více příkazů vyžaduje použití složeného příkazu v bloku, o čemž si povíme více později v této kapitole.)

Inicializace cyklu se provádí pouze jednou. Program obvykle používá tento výraz pro nastavení proměnné na počáteční hodnotu a takto inicializovaná proměnná bývá využita pro počítání průchodů cyklem.

testovací-výraz určuje, zda bude vykonáno tělo cyklu. Tento výraz bývá obvykle relačním výrazem, to znamená, že porovnává dvě hodnoty. Náš příklad porovnává hodnotu proměnné i s číslem 5 a zjišťuje, zda je `i` menší než 5. Je-li porovnání pravdivé, program vykoná tělo cyklu. Jazyk C++ ve skutečnosti neomezuje *testovací-výraz* na porovnání pravda-nepravda. Můžeme použít libovolný výraz a jazyk C++ ho převede na typ `bool`. Výraz vyhodnocený jako 0 je převeden na hodnotu `false` typu `bool` a cyklus končí. Výraz vyhodnocený jako nenulový je převeden na hodnotu `true` typu `bool` a cyklus pokračuje. Ve výpisu 5.2 je toto předvedeno použitím výrazu i jako testovací podmínky. (V aktualizační části je `i--` podobné `i++` s tím rozdílem, že při každém použití snižuje hodnotu i o 1.)

Výpis 5.2 num_test.cpp

```
// num_test.cpp -- použijte numerický test v cyklu for
#include <iostream>
int main()
{
   using namespace std;
   cout << "Zadejte pocatecni odpocitavaci hodnotu: ";
   int limit;
   cin >> limit;
   int i;
   for (i = limit; i; i--)  // konec, když je i rovno 0
      cout << "i = " << i << "\n";
   cout << "Hotovo, nyni je i = " << i << "\n";
   return 0;
}
```

Zde je výstup z programu na výpisu 5.2:

```
Zadejte pocatecni odpocitavaci hodnotu: 4
i = 4
i = 3
i = 2
i = 1
Hotovo, nyni je i = 0
```

Všimněte si, že cyklus končí, když proměnná i dosáhne hodnoty 0.

Jak relační výrazy, jako je například `i < 5`, zapadají do konstrukce ukončení cyklu hodnotou 0? Před zavedením typu `bool` byly pravdivé relační výrazy vyhodnocovány jako 1 a nepravdivé jako 0. Tedy hodnota výrazu `3 < 5` byla 1 a hodnota `5 < 5` byla 0. Nyní C++ přidal typ `bool` a relační výrazy jsou vyhodnocovány řetězcovými konstantami `true` a `false` typu `bool` namísto hodnot 1 a 0. Tato změna nezpůsobuje nekompatibility, protože když jazyk C++ očekává v programu celočíselné hodnoty, konvertuje `true` a `false` na 0 a 1, pokud předpokládá hodnotu typu `bool`, konvertuje 0 na `false` a nenulovou hodnotu na `true`.

Cyklus `for` má podmínku na vstupu. To znamená, že testovací výraz je vyhodnocen *před* každým průchodem cyklem. Cyklus nikdy nevykoná tělo cyklu, když je testovací výraz nepravdivý. Předpokládejme, že například znovu spustíme program z výpisu 5.2 a zadá-

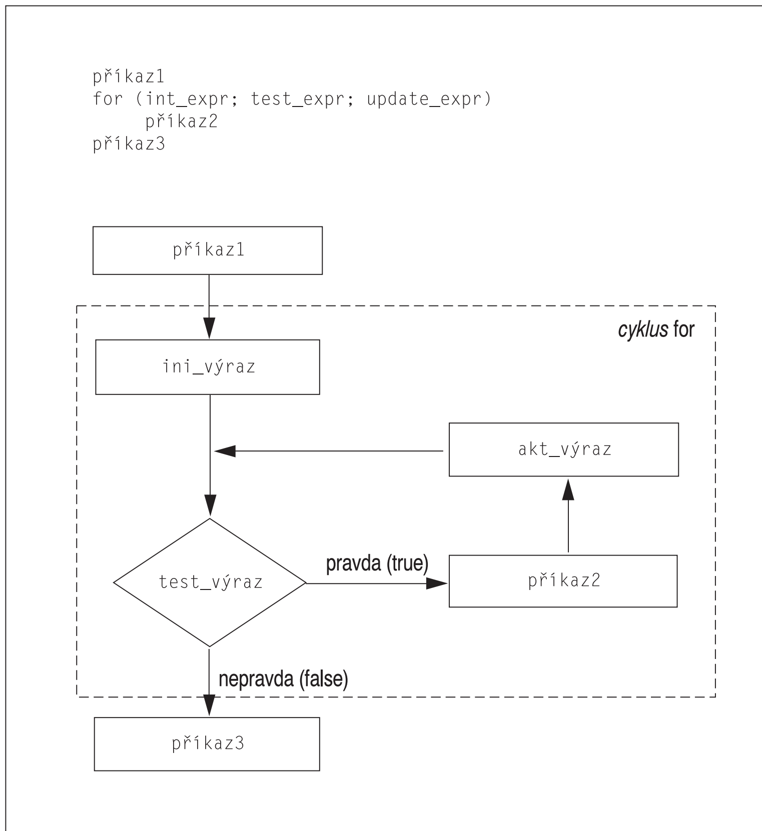
me počáteční hodnotu 0. Protože testovací podmínka selže už při prvním vyhodnocení, tělo cyklu nebude nikdy vykonáno:

```
Zadejte pocatecni odpocitavaci hodnotu: 0
Hotovo, nyni je i = 0
```

Takovýto přístup, kdy je testovací podmínka prověřena před průchodem cyklem, může pomoci bezproblémovému chodu programu.

aktualizační-výraz je vyhodnocován na konci cyklu, až po provedení těla cyklu. Obvykle se používá pro zvýšení nebo snížení hodnoty proměnné, která sleduje počet průchodů cyklem. Může jím být libovolný platný výraz jazyka C++, stejně jako u ostatních řídicích příkazů. To dává cyklu `for` mnohem větší možnosti než pouhé počítání od 0 do 5, jak ukazuje první příklad cyklu. Několik příkladů si ukážeme později.

Tělo cyklu for se skládá z jediného příkazu, ale brzy si předvedeme rozšíření tohoto pravidla. Na obrázku 5.1 je znázorněno schéma cyklu `for`.



Obrázek 5.1 Cyklus `for`

Cyklus `for` se podobá volání funkce, protože používá jméno následované párem závorek. Ale protože `for` představuje v jazyce C++ klíčové slovo, nemůže si ho překladač splést se jménem funkce. Překladač také nedovolí pojmenovat funkci slovem `for`.

TIP

Běžným programovacím stylem v jazyce C++ je umístění mezery mezi klíčovým slovem `for` a následující závorkou a vynechání mezery mezi jménem funkce a následující závorkou:

```
for (int i = 6; i < 10; i++)
          chytra_funkce(i);
```

U ostatních řídicích příkazů, jako jsou například `if` a `while`, je to podobné jako u `for`. To slouží k vizuálnímu posílení rozdílu mezi řídicím příkazem a voláním funkce. Velmi obvyklé je také odsazení těla příkazu `for`, který pak více vynikne.

Výrazy a příkazy

Řídicí sekce cyklu for používá tři výrazy. V rámci vlastních omezení syntaxe jsou vyjadřovací schopnosti jazyka C++ velké. Jakákoliv hodnota nebo platná kombinace hodnot a operátorů vytváří výraz. Například 10 je výraz s hodnotou 10 (což jistě nikoho nepřekvapí) a 28 * 20 je výraz s hodnotou 560. V jazyce C++ má každý výraz hodnotu. Často je hodnota zřejmá. Například výraz

```
22 + 27
```

se skládá ze dvou hodnot a operátoru sčítání a jeho výsledná hodnota je 49. Občas bývá výsledek méně zřejmý. Například

```
x = 20
```

je výraz, protože se skládá ze dvou hodnot a operátoru přiřazení. Jazyk C++ definuje, že výslednou hodnotou přiřazovacího výrazu je hodnota jeho levého členu, takže výše uvedený výraz má hodnotu 20. Skutečnost, že přiřazovací výrazy mají hodnoty, dovoluje psát příkazy, jako je například následující:

```
maids = (cooks = 4) + 3;
```

Výraz `cooks = 4` má hodnotu 4, takže proměnné `maids` je přiřazena hodnota 7. I když jazyk C++ takovéto chování připouští, neznamená to, že bychom ho měli využívat. Ale stejné pravidlo, které umožňuje zmíněný nezvyklý příkaz, nám dovoluje psát užitečné příkazy podobné následujícímu:

```
x = y = z = 0;
```

Toto je rychlý způsob nastavení několika proměnných na stejnou hodnotu. Tabulka priorit (Příloha D) říká, že přiřazení je prováděno zprava doleva, takže hodnota 0 je přiřazena proměnné `z`, potom je hodnota výrazu `z = 0` přiřazena proměnné `y` atd.

Jak jsme si již říkali, relační výrazy, jako je například `x < y`, jsou vyhodnocovány jako hodnoty `true` nebo `false` typu `bool`. Krátký program ve výpisu 5.3 ilustruje některé vlastnosti hodnot výrazů. Operátor << má vyšší prioritu než operátory použité ve výrazech, takže si programový kód vynucuje správné pořadí pomocí závorek.

Výpis 5.3 express.cpp

```
// express.cpp -- hodnoty výrazů
#include <iostream>
int main()
{
   using namespace std;
```

```
    int x;

    cout << "Vyraz x = 100 ma hodnotu ";
    cout << (x = 100) <<  endl;
    cout << "Nyni je x = " << x <<  endl;
    cout << "Vyraz x < 3 ma hodnotu ";
    cout << (x < 3) <<  endl;
    cout << "Vyraz x > 3 ma hodnotu ";
    cout << (x > 3) <<  endl;
    cout.setf(ios_base::boolalpha);
    cout << "Vyraz x < 3 ma hodnotu ";
    cout << (x < 3) <<  endl;
    cout << "Vyraz x > 3 ma hodnotu ";
    cout << (x > 3) <<  endl;
    return 0;
}
```

KOMPATIBILITA

Starší implementace C++ mohou jako argument funkce `cout.setf()` vyžadovat konstantu `ios::boolalpha` namísto `ios_base::boolalpha`. Ještě starší implementace možná nerozpoznají žádnou z těchto konstant.

Zde je výstup z programu na výpisu 5.3:

```
Vyraz x = 100 ma hodnotu 100
Nyni je x = 100
Vyraz x < 3 ma hodnotu 0
Vyraz x > 3 ma hodnotu 1
Vyraz x < 3 ma hodnotu false
Vyraz x > 3 ma hodnotu 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 objektu `cout` přikazuje, aby zobrazoval slova `true` a `false` namísto čísel 1 a 0.

ZAPAMATUJTE SI

V jazyce C++ je výrazem hodnota nebo spojení hodnot a operátorů a každý výraz v C++ má hodnotu.

Aby jazyk C++ vyhodnotil výraz `x = 100`, musí přiřadit hodnotu 100 proměnné `x`. Když vyhodnocení výrazu změní hodnotu dat v paměti, říkáme, že vyhodnocení má *vedlejší účinek*. Tedy vyhodnocení výrazu přiřazení má vedlejší účinek změny hodnoty proměnné, do které je hodnota přiřazována. O přiřazení můžete uvažovat jako o zamýšleném účinku, ale z hlediska návrhu jazyka C++ je primární vyhodnocení výrazu. Ne všechny výrazy mají vedlejší účinky. Například vyhodnocení výrazu `x + 15` vypočítá novou hodnotu, ale nemění hodnotu proměnné `x`. Avšak vyhodnocení `++x + 15` má vedlejší účinek, protože obsahuje inkrementaci proměnné `x`.

Od výrazu k příkazu je jen malý krůček; stačí přidat středník. Tedy

```
age = 100
```

je výraz, zatímco

```
age = 100;
```

představuje příkaz. Libovolný výraz se může stát příkazem, přidáme-li středník, ale výsledek nemusí mít programovací smysl. Například, je-li `rodents` proměnná, potom

```
rodents + 6;              // platný, ale zbytečný příkaz
```

je v C++ platným příkazem. Překladač ho povolí, ale tento příkaz neprovede nic užitečného. Program pouze vypočítá součet, nic s ním neudělá a pokračuje dalším příkazem. (Chytrý překladač dokonce může takovéto příkazy přeskakovat.)

Nevýrazy a příkazy

Některé dovednosti, jako je například znalost struktury cyklu `for`, jsou pro zvládnutí jazyka C++ zásadní. Ale také existují relativně méně důležitá hlediska syntaxe, která vás mohou potrápit právě v okamžiku, kdy si myslíte, že jazyku již rozumíte. Na pár z nich se nyní podíváme.

Ačkoliv je pravda, že přidáním středníku k libovolnému výrazu vytváříme příkaz, opačné tvrzení neplatí. To znamená, že odstranění středníku z příkazu nemusí nezbytně vést ke vzniku výrazu. Z doposud použitých příkazů nevyhovují tvaru *příkaz* = *výraz* + *středník* `return`, deklarační příkazy a příkazy `for`. I když je například

```
int toad;
```

příkaz, část `int toad` nepředstavuje výraz a nemá hodnotu. Může dokonce způsobit neplatnost kódu, což vidíme v následující ukázce:

```
eggs = int toad * 1000;   // neplatné, není výrazem
cin >> int toad;          // nemůžeme kombinovat deklaraci s objektem cin
```

Podobně nemůžeme přiřadit cyklus `for` proměnné. V následujícím příkladu cyklus `for` není výraz, takže nemá hodnotu a nemůžeme ho přiřazovat

```
int fx = for (int i = 0; i< 4; i++)
    cout >> i;            // není možné
```

Přizpůsobování pravidel

Jazyk C++ přidává cyklům jazyka C vlastnost, která vyžaduje jisté úpravy syntaxe cyklu `for`. Toto byla původní syntaxe:

```
for (výraz; výraz; výraz)
    příkaz
```

Jak jsme si již řekli dříve, řídicí část struktury cyklu for se skládá ze tří výrazů oddělených středníkem. Cykly jazyka C++ nám však dovolují provádět například následující činnosti:

```
for (int i = 0; i < 5; i++)
```

To znamená, že v inicializační oblasti cyklu `for` můžeme deklarovat proměnnou. Takováto možnost je výhodná, ale odporuje původní syntaxi, protože deklarace není výrazem. Toto nepovolené chování bylo původně upraveno definováním nového druhu výrazu nazvaného *deklaračně-příkazový výraz*, který byl vlastně deklarací bez středníku a mohl se vyskytovat pouze v příkazech cyklu `for`. Avšak od této úpravy bylo později upuštěno. Místo toho byla syntaxe příkazu `for` pozměněna následovně:

```
for (inicializační-příkaz-for podmínka; výraz)
    příkaz
```

Tento zápis vypadá na první pohled podivně, protože obsahuje pouze jeden středník místo dvou. To je ale v pořádku, protože *inicializační-příkaz-for* je definován jako příkaz a příkaz má svůj vlastní středník. *inicializační-příkaz-for* je identifikován jako výrazový příkaz nebo deklarace. Toto syntaktické pravidlo nahrazuje výraz následovaný středníkem příkazem, který má svůj vlastní středník. Důvodem této změny je, že programátoři jazyka C++ chtějí deklarovat a inicializovat proměnnou v inicializační části cyklu `for` a udělají všechno pro to, aby jim to syntaxe jazyka C++ a anglický jazyk dovolily.

Je zde ještě praktický důvod deklarace proměnné v *inicializačním-příkazu-for*, o kterém byste měli vědět. Takto deklarovaná proměnná existuje pouze uvnitř příkazu `for`. To znamená, že jakmile program opustí cyklus, proměnná je odstraněna:

```
for (int i = 0; i < 5; i++)
      cout << "C++ umi cykly.\n";
cout << i << endl;          // chyba! proměnná i už není definována
```

Dále byste měli vědět, že některé implementace jazyka C++ dodržují dřívější pravidlo a k výše uvedenému cyklu se chovají, jako kdyby byla proměnná i deklarována *před* cyklem, a tedy zůstává dostupná i po ukončení cyklu. Použití nové možnosti deklarace proměnné v inicializaci cyklu `for` má alespoň prozatím za následek odlišné chování na různých systémech.

UPOZORNĚNÍ

V době psaní této knihy ne všechny překladače dodržovaly aktuální pravidlo, podle kterého proměnná deklarovaná v řídicí sekci cyklu `for` zaniká s ukončením cyklu. Například Microsoft Visual C++ do verze 7.1 implicitně dodržuje původní pravidlo, a to zejména proto, že ve stávajících knihovnách firmy Microsoft je mnoho programů napsaných ještě před přijetím nového pravidla. (Verze 7.1 je statečný, avšak nestandardní pokus žít s oběma pravidly současně, tedy akceptovat programy napsané oběma způsoby.

Zpět k cyklu for

Buďme na cykly trochu náročnější. Výpis 5.4 používá cyklus na výpočet a uložení prvních 16 faktoriálů. Faktoriály, které jsou užitečné při výpočtech pravděpodobnosti, se počítají následujícím způsobem. Nulový faktoriál, psáno jako 0!, je definován jako rovný 1. Potom 1! vypočítáme 1 * 0!, což je 1. Dále 2! vypočítáme 2 * 1!, čili 2. Potom 3! vypočítáme 3 * 2!, čili 6 atd. Faktoriál každého celého čísla je součin tohoto čísla s faktoriálem předchozího celého čísla. (Jeden z nejznámějších monologů pianisty Victora Borge charakterizuje vlastnosti hlasové interpunkce, ve které je vykřičník vyslovován jako phffft pptz s vlhkým přízvukem. Avšak v tomto případě se „!“ vyslovuje jako „faktoriál“.) Program používá jeden cyklus na výpočet hodnot po sobě jdoucích faktoriálů a výsledky ukládá do pole. Ve druhém cyklu vypočítané hodnoty zobrazuje. Program také předvádí použití hodnot vnějších deklarací.

Výpis 5.4 formore.cpp

```
// formore.cpp -- více cyklů for
#include <iostream>
using namespace std;
const int ArSize = 16;      // příklad vnější 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] << endl;
   return 0;
}
```

Zde je výstup z programu ve výpisu 5.4:

```
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 na výpisu 5.4 vytváří pole pro uložení hodnot faktoriálů. Prvek 0 je 0!, prvek 1 je 1! atd. Protože první dva faktoriály jsou rovny 1, program nastaví první dva prvky pole `factorials` na hodnotu 1.0. (Vzpomeňte si, že první prvek pole má hodnotu indexu 0.) Potom program v cyklu nastavuje hodnotu každého faktoriálu na součin indexu s předchozím faktoriálem. Cyklus ukazuje možnost použití počítadla cyklu jako proměnné v těle cyklu.

Program předvádí spolupráci cyklu `for` s polem, která poskytuje vhodný prostředek pro postupný přístup ke každému členu pole. Výpis `formore.cpp` také pomocí klíčového slova `const` vytváří symbolickou reprezentaci (`ArSize`) velikosti pole. Konstanta `ArSize` je potom použita všude, kde je potřeba velikost pole, například v definici pole a pro meze cyklů, které s polem pracují. Pokud budeme nyní chtít rozšířit program například na 20 faktoriálů, musíme pouze přiřadit konstantě `ArSize` hodnotu 20 a znovu přeložit program. Použitím symbolické konstanty se vyhneme nutnosti měnit všechny výskyty čísla 16 na 20.

TIP

Obvykle se vyplatí definovat pomocí klíčového slova const konstantu pro vyjádření počtu prvků v poli. Tuto konstantu použijeme v deklaraci pole a ve všech dalších odkazech na velikost pole, například v cyklu `for`.

Výraz `limit i < ArSize` vyjadřuje skutečnost, že index pole s `ArSize` prvky má rozsah od jedné do `ArSize - 1`, takže by se index tohoto pole měl zastavit na hodnotě o 1 menší než `ArSize`. Můžeme také použít test `i <= ArSize - 1`, ale ten vypadá v porovnání s předchozím poněkud neohrabaně.

Zajímavou vlastností tohoto programu je, že deklaruje proměnnou `ArSize` typu `const int` mimo tělo funkce `main()`. Jak jsme si řekli na konci kapitoly 4 „Složené typy", tímto způsobem vytváříme z `ArSize` vnější datovou položku. Takto deklarovaná konstanta `ArSize` bude existovat po celou dobu trvání programu a budou ji moci využívat všechny funkce ze souboru kódu programu. V tomto konkrétním případě má program pouze jednu funkci, takže vnější deklarace konstanty `ArSize` nemá velký význam. Ale v programech s více funkcemi bývá sdílení vnějších konstant užitečné, takže si nyní procvičíme jejich používání.

Změna velikosti kroku

V příkladech cyklů bylo počitadlo doposud zvyšováno nebo snižováno při každém průchodu pouze o 1. Toto můžeme změnit úpravou aktualizačního výrazu. Program ve výpisu 5.5 například zvyšuje počitadlo cyklů o velikost kroku zvolenou uživatelem. Místo aktualizačního výrazu `i++`, tento program používá výraz `i = i + by`, kde `by` je uživatelem zvolená velikost kroku.

Výpis 5.5 bigstep.cpp

```
// bigstep.cpp -- přičítání podle zadání
#include <iostream>
int main()
{
    using namespace std;
    cout << "Zadejte cele cislo: ";
    int by;
    cin >> by;
    cout << "Pricitani po " << by << ":\n";
    for (int i = 0; i < 100; i = i + by)
        cout << i << endl;
    return 0;
}
```

Zde je ukázka běhu programu na výpisu 5.5:

```
Zadejte cele cislo: 17
Pricitani po 17:
0
17
34
51
68
85
```

Jakmile proměnná i dosáhne hodnoty 102, cyklus končí. Důležité je, abychom si uvědomili, že aktualizačním výrazem může být libovolný platný výraz. Chcete-li v každém průchodu například umocnit proměnnou i na druhou a přičíst 10, můžete použít výraz

```
i = i * i + 10.
```

Procházení řetězců pomocí cyklu for

Cyklus for poskytuje přímý způsob postupného přístupu ke každému znaku v řetězci. Výpis 5.6 například umožňuje zadání řetězce a potom jeho zobrazení znak po znaku v obráceném pořadí. V tomto příkladu byste měli použít buď objekt třídy `string` nebo pole `char`, neboť v obou případech můžete při odkazech na jednotlivé znaky použít stejný zápis jako při odkazech do polí. Metoda `size()` třídy `string` vrací počet znaků v řetězci; cyklus používá tuto hodnotu ve svém inicializačním výrazu pro nastavení proměnné i na index posledního znaku řetězce, když nepočítáme nulový znak. Na odpočítávání směrem dolů používáme v programu operátor dekrementace (--), který snižuje index pole o 1 při každém průchodu cyklem. Ve výpisu 5.6 také používáme pro otestování, zda cyklus dosáhl prvního prvku, relační operátor větší nebo rovno (>=). Všechny relační operátory si brzy shrneme.

Výpis 5.6 forstr1.cpp

```
// forstr1.cpp -- použití cyklu for s řetězcem
#include <iostream>
#include <cstring>
const int ArSize = 20;
int main()
{
   using namespace std;
   cout << "Zadejte slovo: ";
   string word;
   cin >> word;

   // zobrazuje písmena v opačném pořadí
   for (int i = word.size() - 1; i >= 0; i--)
      cout << word[i];
   cout << "\nNashledanou.\n";
   return 0;
}
```

Zde je ukázka běhu programu z výpisu 5.6:

```
Zadejte slovo: zvire
erivz
Nashledanou.
```

Je zřejmé, že program umí vytisknout slovo zvire pozpátku; zvolené testovací slovo zvire mnohem jasněji ukazuje smysl tohoto programu než například slovo `oko` nebo `bob`.

Operátor inkrementace (++) a dekrementace (–)

Jazyk C++ má několik operátorů, které se často vyskytují v cyklech a právě jim se budeme nyní chvíli věnovat. Dva z nich jsme si již ukázali: operátor inkrementace (++), jenž inspiroval jméno jazyka C++ a operátor dekrementace (--). Oba provádějí velmi časté operace cyklů: zvyšování a snižování počitadla o 1. Musíme si však o nich povědět více než doposud. Tyto operátory se vyskytují ve dvou variantách. *Prefixová* verze je vkládána před operand, jako ve výrazu `++x`. *Postfixová* verze je vkládána za operand, jako ve výrazu `x++`. Obě verze mají na operand stejný vliv, ale liší se v okamžiku jejich provedení. Je to jako zaplatit za posekání trávníku předem nebo až po vykonání práce; obě meto-

dy mají stejný konečný vliv na naši peněženku, ale liší se v tom, kdy musíme peníze vydat. Výpis 5.7 ukazuje tento rozdíl pro operátor inkrementace.

Výpis 5.7 plus_one.cpp

```
// plus_one.cpp -- oprátor inkrementování
#include <iostream>
int main()
{
   using namespace std;
   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 z programu na výpisu 5.7:

```
a   = 20:   b = 20
a++ = 20: ++b = 21
a   = 21:   b = 21
```

Stručně řečeno, zápis `a++` znamená „použij aktuální hodnotu `a` a vyhodnoť výraz, potom hodnotu `a` inkrementuj“. Podobně zápis `++b` znamená „nejprve inkrementuj hodnotu `b` a potom použij novou hodnotu na vyhodnocení výrazu“. Máme například následující vztahy:

```
int x = 5;
int y = ++x;               // změní hodnotu x a potom ji přiřadí do y
                           // y je 6, x je 6

int z = 5;
int y = z++;               // přiřadí hodnotu z do y a potom ji změní
                           // y je 5, z je 6
```

Operátory inkrementace a dekrementace představují jednoduchý a stručný způsob zápisu zvyšování a snižování hodnoty o jedničku.

Operátory inkrementace a dekrementace jsou šikovné, ale nenechte se unést a neprovádějte inkrementaci nebo dekrementaci stejné hodnoty v jednom příkazu více než jednou. Problémem je, že pravidla použij-potom-změň a změň-potom-použij se mohou stát nejednoznačnými. To znamená, že například následující příkaz

```
x = 2 * x++ * (3 - ++x);    // nedělejte to
```

může mít na různých systémech zcela odlišné výsledky. Jazyk C++ nedefinuje přesné chování pro takovýto typ příkazů.

Vedlejší účinky a sekvenční ukazatele

Podívejme se trochu podrobněji, co C++ říká a co neříká o tom, co se děje, když použijeme inkrementační operátory. Nejdříve si vzpomeňme, že *vedlejší účinek* je nějaká změna, která nastane při vyhodnocování výrazu, například uložení hodnoty do proměnné. *Sekvenční bod* je místo v programu, v němž je zaručeno, že všechny vedlejší účinky

budou provedeny před přechodem na další krok. V C++ je sekvenční bod označen středníkem. To znamená, že všechny změny v příkazu způsobené přiřazovacím operátorem, inkremetačními a dekremetačními operátory musí nastat před zpracováním následujícího příkazu. Některé operátory, o nichž budeme hovořit v následujících kapitolách, mají sekvenční bod. Konec všech úplných výrazů je tedy sekvenčním bodem.

Co je to úplný výraz? Je to výraz, který není podvýrazem většího výrazu. Příklady úplných výrazů obsahují výrazovou část příkazu a výraz, který tvoří testovací podmínku cyklu `while`.

Sekvenční body nám pomohou objasnit, kdy nastává postfixová inkrementace. Předpokládejme například, že máme takovýto kód:

```
while (guests++ < 10)
      printf("%d \n", guests);
```

Začátečníci v C++ někdy mají za to, že „použít hodnotu a pak ji inkrementovat" znamená v tomto kontextu inkrementovat proměnnou `guests` až poté, co je použita v příkazu `printf()`. Avšak výraz `guests++ < 10` je úplný výraz, neboť je to testovací podmínka cyklu `while`, takže konec tohoto výrazu je sekvenčním bodem. Proto C++ zaručuje, že vedlejší účinek (inkrementace proměnné `guests`) nastane dřív, než program přejde na `prinf()`. V postfixovém tvaru však zaručuje, že `guests` se zvýší až po porovnání s číslem 10.

Uvažujme nyní tento příkaz:

```
y = (4 + x++) + (6 + x++);
```

Výraz `4 + x++` není úplným výrazem, takže C++ nezaručuje, že x bude inkrementováno ihned po vyhodnocení podvýrazu `4 + x++`. Zde je úplným výrazem celý přiřazovací příkaz (sekvenční bod je označen středníkem), takže dokud program nepřejde na následující příkaz, C++ zaručuje pouze dvojitou inkrementaci `x`. Není určeno, zda `x` je inkrementováno po vyhodnocení každého podvýrazu nebo až po vyhodnocení celého výrazu. A právě to je důvodem, proč je nutno se takovým příkazům vyhnout.

Prefix a postfix

Je zjevné, že je rozdíl mezi tím, zda použijete prefix nebo postfix, pokud je hodnota použita pro určitý účel, např. jako argument funkce nebo pro přiřazení hodnoty proměnné. Ale co když se hodnota inkrementu resp. dekrementu nepoužije? Je nějaký rozdíl mezi

```
x++;
```

a

```
++x;
```

Liší se nějak? Anebo je mezi

```
for (n = lim; n > 0; --n)
    ...;
```

a

```
for (n = lim; n > 0; n--)
      ...;
```

nějaký rozdíl?

V těchto dvou situacích logicky není mezi prefixem a postfixem žádný rozdíl. Hodnoty výrazů se nepoužijí, takže nedojde k ničemu jinému než k vedlejším účinkům. Oba výra-

zy, které používají operátory, jsou úplné, takže je zaručeno, že vedlejší účinky inkremetace x a dekremetace n jsou provedeny do té doby, než program přejde na další krok; prefix i postfix tedy povedou ke stejnému výsledku.

Nicméně, i když na chování programu nebude mít žádný vliv, zda vybereme prefix nebo postfix, může mít výběr určitý vliv na rychlost provádění programu. U vestavěných typů a moderních překladačů v tomto problém nebude. Avšak v C++ můžeme definovat tyto operátory jako třídy. V takovém případě uživatel definuje prefixovou funkci, která inkrementuje hodnotu a vrátí ji, zatímco postfixová verze nejdříve uloží kopii hodnoty, inkrementuje hodnotu a pak vrátí uloženou hodnotu. V případě tříd je tedy prefixová verze o trochu efektivnější než postfixová.

Stručně řečeno, u vestavěných typů je většinou jedno, jaký tvar použijete. V případě typů definovaných uživatelem, které mají uživatelsky definované inkremetační a dekrementační operátory, je prefixový tvar lepší.

Operátory ++ / -- a ukazatele

Inkrementační operátory můžete používat s ukazateli podobně jako se základními proměnnými. Připomeňme si, že počet bajtů, o nějž se zvýší hodnota ukazatele přičtením, odpovídá typu, na nějž ukazuje. Stejné pravidlo platí i pro inkrementaci a dekrementaci ukazatelů:

```
double arr[5] = {21.1, 32.8, 23.4, 45.2, 37.4};
double *pt = arr;     // pt ukazuje na arr[0], tj. na 21,1
++pt;                 // pt ukazuje na arr[1], tj. na 32,8
```

Tyto operátory lze ve spojení s operátorem * také využít ke změně hodnoty, na niž ukazatel ukazuje. Vyvstává otázka, co se vlastně aplikací obou operátorů * a ++ na ukazatel dereferencuje a co se inkrementuje. Rozhodující je umístění a priorita operátorů. Operátory pro prefixový inkrement, prefixový dekrement a dereference mají stejnou prioritu a vyhodnocují se zprava doleva. Operátory pro postfixový inkrement a dekrement mají také stejnou prioritu, avšak vyšší než má prefix, a vyhodnocují se naopak zleva doprava.

Z asociačního pravidla (tj. zprava doleva) pro prefixové operátory plyne, že ve výrazu *++pt se nejdříve uplatní operátor ++ na pt (neboť ++ je napravo od *) a teprve pak operátor * na novou hodnotu pt:

```
*++pt;          // inkrement ukazatele, vezmi hodnotu; tj. arr[2] resp. 23,4
```

Naopak ++*pt znamená vezmi hodnotu, na kterou ukazuje pt a inkrementuj ji:

```
++*pt;          // inkrement hodnoty, na kterou ukazuje ukazatel;
                // tj. změň 23,4 na 24,4
```

Po provedení bude pt i nadále ukazovat na arr[2].

Dále uvažujme o této kombinaci:

```
(*pt)++;        // inkrement hodnoty, na kterou ukazuje ukazatel
```

Závorky říkají, že nejdříve se dereferencuje odkaz, z čehož plyne 24,4 a poté operátor ++ zvýší tuto hodnotu na 25,4; pt bude i nadále ukazovat na arr[2].

A nakonec uvažujme o kombinaci:

```
*pt++;          // dereference ukazatele, poté jeho inkrement
```

Vyšší priorita postfixového operátoru ++ znamená, že se vztahuje k `pt`, nikoli k `*pt`, takže se inkrementuje ukazatel. Avšak skutečnost, že je použit postfixový operátor, znamená, že je dereferencována původní adresa `&arr[2]`, nikoli nová adresa. Hodnota `*pt++` je `arr[2]` resp. 25,4, avšak hodnota `pt` po dokončení příkazu bude adresa `arr[3]`.

ZAPAMATUJTE SI

Inkrementace a dekrementace ukazatelů dodržuje pravidla aritmetiky ukazatelů. Tedy jestliže `pt` ukazuje na první člen pole, `++pt` změní `pt` tak, že ukazuje na druhý člen.

Sdružování přiřazovacích operátorů

Výpis 5.5 používá na změnu počitadla cyklu následující výraz:

```
i = i + by
```

Jazyk C++ má sdružený operátor sčítání a přiřazení, který dosáhne stejného výsledku zkráceným zápisem:

```
i += by
```

Operátor += sečte hodnotu svých dvou operandů a výsledek přiřadí levému operandu. Levému operátoru tedy musíme být schopni přiřadit hodnotu. Může jím například být proměnná, prvek pole, člen struktury nebo data identifikovaná dereferencí ukazatele:

```
int k = 5;
k += 3;                    // v pořádku, k přiřadí hodnotu 8
int *pa = new int[10];// pa ukazuje na pa[0]
pa[4] = 12;
pa[4] += 6;                // v pořádku, pa[4] přiřadí hodnotu 18
*(pa + 4) += 7;            // v pořádku, pa[4] přiřadí hodnotu 25
pa += 2;                   // v pořádku, pa ukazuje na předchozí prvek pa[2]
34 += 10;                  // zcela chybné
```

Každý aritmetický operátor má odpovídající přiřazovací operátor, jak ukazuje tabulka 5.1. Každý operátor pracuje podobně jako +=. Tedy příkaz

```
k *= 10;
```

nahradí současnou hodnotu k hodnotou 10krát větší.

Tabulka 5.1 Sdružené přiřazovací operátory

Operátor	Výsledek (L = levý operand, P = pravý operand)
+=	Přiřadí L + P do L
-=	Přiřadí L – P do L
*=	Přiřadí L * P do L
/=	Přiřadí L / P do L
%=	Přiřadí L % P do L

Složené příkazy neboli bloky

Formát nebo syntaxe psaní příkazu `for` v jazyce C++ se může zdát omezující, protože tělo cyklu může obsahovat pouze jeden příkaz. To je nešikovné, pokud potřebujeme v cyklu

provádět více příkazů. Naštěstí jazyk C++ poskytuje syntaxi, s jejíž pomocí můžeme do těla cyklu vložit libovolné množství příkazů. Tento trik spočívá ve vytvoření *složeného příkazu*, častěji označovaného jako *blok*, pomocí páru složených závorek. Blok, který je z důvodů syntaxe považován za jediný příkaz, se tedy skládá z páru složených závorek a v nich uzavřených příkazů. Například program ve výpisu 5.8 používá složené závorky pro sloučení tří samostatných příkazů do jednoho bloku. To umožňuje v těle cyklu vyzvat uživatele, přečíst vstup a provést výpočet. Program sčítá zadaná čísla, což nám dává přirozenou možnost použít operátor +=.

Výpis 5.8 block.cpp

```
// block.cpp -- použití příkazů v bloku
#include <iostream>
int main()
{
   using namespace std;
   cout << "Bajecny Accounto vam secte";
   cout << " a vypocita prumer z peti cisel.\n";
   cout << "Prosim, zadejte pet hodnot:\n";
   double number;
   double sum = 0.0;
   for (int i = 1; i <= 5; i++)
   {                                      // začátek bloku
      cout << "Hodnota " << i << ": ";
      cin >> number;
      sum += number;
   }                                      // konec bloku
   cout << "Vybral jste pet opravdu skvelych cisel! ";
   cout << "Jejich soucet je " << sum << endl;
   cout << "a prumer " << sum / 5 << ".\n";
   cout << "Bajecny Accounto se s vami louci!\n";
   return 0;
}
```

Zde je příklad běhu programu na výpisu 5.8:

```
Bajecny Accounto vam secte a vypocita prumer z peti cisel.
Prosim, zadejte pet hodnot:
Hodnota 1: 1942
Hodnota 2: 1948
Hodnota 3: 1957
Hodnota 4: 1974
Hodnota 5: 1980
Vybral jste pet opravdu skvelych cisel! Jejich soucet je 9801
a prumer 1960.2.
Bajecny Accounto se s vami louci!
```

Předpokládejme, že ponecháme odsazení, ale vynecháme složené závorky:

```
for (int i = 1; i <= 5; i++)
   cout << "Hodnota " << i << ": ";      // cyklus končí zde
   cin >> number;                         // zde už jsme za cyklem
   sum += number;
cout << "Vybral jste pet opravdu skvelych cisel! ";
```

Překladač ignoruje odsazení, takže v cyklu by byl pouze první příkaz. Tedy cyklus by vytiskl pět výzev a nic víc. Po ukončení cyklu by se program přesunul na následující řádky, a potom by přečetl a sečetl pouze jedno číslo.

Složené příkazy mají další zajímavou vlastnost. Definujete-li uvnitř bloku novou proměnnou, tato proměnná existuje pouze tak dlouho, dokud program provádí příkazy uvnitř bloku. Jakmile výpočet opouští blok, proměnná je zrušena. To znamená, že takováto proměnná je známa pouze uvnitř bloku:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 20;
    {                               // začátek bloku
        int y = 100;
        cout << x << "\n";          // v pořádku
        cout << y << "\n";          // v pořádku
    }                               // konec bloku
    cout << x << "\n";              // v pořádku
    cout << y << "\n";              // neplatné, nelze přeložit
    return 0;
}
```

Všimněte si, že proměnná definovaná ve vnějším bloku je platná i ve vnitřním bloku.

Co se stane, když v bloku deklarujeme proměnnou se stejným jménem, jako je ve vnějším bloku? Nová proměnná překryje starou od začátku své existence do konce bloku. Potom je opět viditelná stará proměnná.

```
int main()
{
    int x = 20;                     // původní proměnná x
    {                               // začátek bloku
        cout << x << endl;          // použití původní proměnné x
        int x = 100;                // nová proměnná x
        cout << x << endl;          // použití nové proměnné x
    }                               // konec bloku
    cout << x << endl;              // použití původní proměnné x
    return 0;
}
```

Operátor čárka (další syntaktické triky)

Blok, jak jsme si ukázali, umožňuje vložit dva nebo více příkazů na místo, kde syntaxe jazyka C++ povoluje pouze jeden příkaz. Operátor čárka poskytuje to samé pro výrazy, umožňuje vložit dva výrazy na místo, kde syntaxe C++ dovoluje pouze jeden. Předpokládejme, že máme cyklus, kde je v každém průchodu jedna proměnná o 1 zvyšována a druhá snižována. Bylo by výhodné provádět oba výrazy v aktualizační části řídicí sekce cyklu for, ale syntaxe cyklu zde umožňuje vložit pouze jediný výraz. Řešením je použití operátoru čárka ke spojení dvou výrazů do jednoho:

```
j++, i--     // pro syntaktické účely se tyto dva výrazy počítají za jeden
```

Čárka není vždy operátorem. Například čárka v deklaraci

```
int i, j;    // čárka je zde oddělovačem, nikoli operátorem
```

slouží k oddělení sousedních jmen v seznamu proměnných.

Ve výpisu 5.9, který převrací obsah pole znaků, je operátor čárka použit dvakrát. Všimněte si, že výpis 5.6 zobrazuje obsah pole v obráceném pořadí, ale výpis 5.9 skutečně přesunuje znaky v poli. Program také používá blok pro seskupení několika příkazů do jednoho.

Výpis 5.9 forstr2.cpp

```
// forstr2.cpp -- obrácení pole
#include <iostream>
#include <cstring>
const int ArSize = 20;
int main()
{
   using namespace std;
   cout << "Zadejte slovo: ";
   string word;
   cin >> word;

   // fyzicky modifikuje pole
   char temp;
   int i, j;
   for (j = 0, i = word.size() - 1; j < i; --i, ++j)
   {                                     // začátek bloku
      temp = word[i];
      word[i] = word[j];
      word[j] = temp;
   }                                     // konec bloku
   cout << word << "\nKonec\n";
   return 0;
}
```

Zde je příklad výstupu programu z výpisu 5.9:

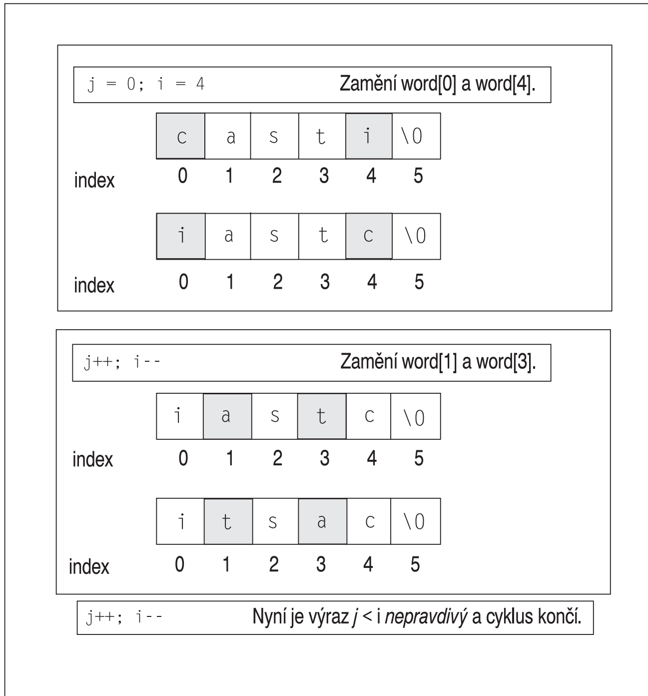
```
Zadejte slovo: casti
itsac
Konec
```

Poznámky k programu

Podívejme se na řídicí sekci cyklu `for`:

Nejprve pomocí operátoru čárka vložíme dvě inicializace do jednoho výrazu v první části řídicí sekce. Potom opět použijeme tento operátor na sloučení dvou řídicích údajů do jediného výrazu v poslední části řídicí sekce.

Dále se podíváme na tělo cyklu. Program používá složené závorky na sloučení několika příkazů do jednoho. V těle program obrací pořadí ve slově záměnou prvního a posledního znaku pole. Potom inkrementuje `j` a dekrementuje `i`, takže se nyní odkazují na prvek následující za prvním prvkem a prvek před posledním prvkem. Jakmile je to hotovo, program tyto prvky zamění. Všimněte si testovací podmínky `j < i`, která způsobuje ukončení cyklu při dosažení středu pole. Kdyby program pokračoval za tento bod, začal by vracet přehozené prvky do jejich původní pozice. (Viz obrázek 5.2.)



Obrázek 5.2 Obrácení řetězce

Další věc, která stojí za povšimnutí, je umístění deklarace proměnných `temp`, `i` a `j`. Programový kód deklaruje `i` a `j` před cyklem, protože operátorem čárka nemůžeme spojovat dvě deklarace. Deklarace již používají čárku pro jiný účel – oddělení položek v seznamu. Pro vytvoření a inicializaci dvou proměnných můžeme použít jeden výraz obsahující deklaraci i příkaz, ale ten je na pohled poněkud matoucí:

```
int j = 0, i = word.size() - 1;
```

V tomto případě je čárka pouze oddělovačem položek seznamu, nikoli operátorem, takže výraz deklaruje a inicializuje jak proměnnou `j`, tak `i`. Ale vypadá to, jako by deklaroval pouze proměnnou `j`.

Proměnnou `temp` můžete také deklarovat uvnitř cyklu `for`:

```
int temp = word[i];
```

To má za následek alokaci a dealokaci proměnné `temp` při každém průchodu cyklem, což může být trochu pomalejší než deklarace proměnné `temp` jednou před cyklem. Ale na druhou stranu je proměnná deklarovaná uvnitř cyklu na jeho konci zrušena.

Zvláštnosti použití operátoru čárka

Zdaleka nejběžnějším použitím operátoru čárka je vložení dvou nebo více výrazů do jediného výrazu cyklu `for`. Avšak jazyk C++ dává tomuto operátoru další dvě vlastnosti. Zaprvé zaručuje, že je první výraz vyhodnocen před druhým. (Jinými slovy, operátor čárka je sekvenční bod.) Například výrazy podobné následujícímu jsou bezpečné:

```
i = 20, j = 2 * i     // proměnná i nastavena na hodnotu 20, j na 40
```

Zadruhé jazyk C++ stanovuje, že hodnotou výrazu s operátorem čárka je hodnota jeho druhé části. Například hodnotou předchozího výrazu je 40, protože to je výsledek výrazu `j = 2 * i`.

Operátor čárka má mezi všemi operátory nejnižší prioritu. Například příkaz

```
cats = 17, 240;
```

se čte jako

```
(cats = 17), 240;
```

To znamená, že proměnné `cats` je přiřazena hodnota 17 a 240 nic nedělá. Ale protože závorky mají vyšší prioritu, příkaz

```
cats = (17,240);
```

způsobí nastavení proměnné `cats` na 240, což je hodnota napravo od čárky.

Relační výrazy

Počítače toho umí více než neúnavné polykání čísel. Mají schopnost porovnávat hodnoty a to je základem počítačového rozhodování. V jazyce C++ je tato schopnost dána relačními operátory. C++ disponuje šesti relačními operátory pro porovnávání čísel. Protože jsou znaky reprezentovány pomocí odpovídajících ASCII kódů, můžeme na ně tyto operátory také použít, ale nepracují s řetězci stylu C. Každý relační výraz je zredukován na hodnotu `true` typu `bool`, pokud je porovnání pravdivé a na `false`, jestliže je nepravdivé, takže se velmi dobře uplatní v testovacích výrazech cyklů. (Starší implementace vyhodnocují pravdivé relační výrazy hodnotou 1 a nepravdivé 0.) Přehled těchto operátorů naleznete v tabulce 5.2.

Tabulka 5.2 Relační operátory

Operátor	Význam
<	Je menší než
<=	Je menší nebo rovno
==	Je rovno
>	Je větší než
>=	Je větší nebo rovno
!=	Není rovno

Těchto šest relačních operátorů představuje všechna porovnání, která jazyk C++ umožňuje provádět s čísly. Chcete-li porovnat dvě hodnoty, abyste se dozvěděli, která je hezčí nebo šťastnější, musíte se poohlédnout po něčem jiném.

Zde je několik příkladů 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 výrazu

```
(x + 3) > (y - 2)        // výraz 2
```

nikoli však výrazu:

```
x + (3 > y) - 2          // výraz 3
```

Protože výraz (`3 > y`) má po povýšení hodnoty typu `bool` na `int` buďto hodnotu 1 nebo 0, jsou oba výrazy 2 a 3 platné. Ale většina z nás by jistě chtěla, aby význam výrazu 1 znamenal to samé co význam výrazu 2, a to právě C++ dělá.

Chyba, kterou možná uděláte

Nespleťte si při testování operátor je rovno (==) s přiřazovacím operátorem (=). Výraz

```
muzikanti == 4           // porovnání
```

se ptá na hudební otázku, jsou muzikanti rovni 4? Tento výraz má hodnotu `true` nebo `false`. Výraz

```
muzikanti = 4            // přiřazení
```

přiřadí hodnotu 4 proměnné muzikanti. Celý výraz má v tomto případě hodnotu 4, protože to je hodnota levé strany.

Pružně navržený cyklus `for` vytváří zajímavou příležitost pro vznik chyby. Pokud náhodou v operátoru == vynecháte jeden znak rovná se (=) a tak v testovací části cyklu for použijete přiřazovací výraz místo relačního, bude stále výsledkem platný kód. Pro testovací podmínku cyklu for je možné použít jakýkoli platný výraz jazyka C++. Pamatujte si, že nenulové hodnoty jsou vyhodnocovány jako `true` a nulové jako `false`. Výraz, který přiřazuje 4 proměnné muzikanti, má hodnotu 4 a je považován za `true`. Přecházíte-li z jazyků např. Pascal nebo Basic, které používají pro testování rovnosti operátor =, pravděpodobně tuto chybu někdy uděláte.

Výpis 5.10 ukazuje situaci, ve které můžete tento druh chyby vytvořit. Program se pokouší prověřit pole výsledků kvizu a zastaví se, když dosáhne prvního výsledku, jenž není roven 20. Předvádí cyklus, který správně používá porovnání a potom druhý, jenž používá omylem přiřazení v testovací podmínce. Tento program má ještě jednu zřejmou chybu v návrhu, jejíž odstranění si ukážeme později. (Chybami se učíme a výpis 5.10 k těmto účelům dobře poslouží.)

Výpis 5.10 equal.cpp

```
// equal.cpp -- rovnost proti přiřazení
#include <iostream>
int main()
{
   using namespace std;
   int quizscores[10] =
      {20, 20, 20, 20, 20, 19, 20, 18, 20, 20};

   cout << "Provedeno spravne:\n";
   int i;
   for (i = 0; quizscores[i] == 20; i++)
      cout << "kviz " << i << " je 20\n";

   cout << "Provedeno nebezpecne spatne:\n";
   for (i = 0; quizscores[i] = 20; i++)
      cout << "kviz " << i << " je 20\n";
```

```
    return 0;
}
```

Protože tento program má závažný problém, měli byste si ho před spuštěním prostudovat. Zde je jedna ukázka výstupu z programu:

```
Provedeno spravne:
kviz 0 je 20
kviz 1 je 20
kviz 2 je 20
kviz 3 je 20
kviz 4 je 20
Provedeno nebezpecne spatne:
kviz 0 je 20
kviz 1 je 20
kviz 2 je 20
kviz 3 je 20
kviz 4 je 20
kviz 5 je 20
kviz 6 je 20
kviz 7 je 20
kviz 8 je 20
kviz 9 je 20
kviz 10 je 20
kviz 11 je 20
kviz 12 je 20
kviz 13 je 20
...
```

První cyklus se správně zastaví po zobrazení prvních pěti výsledků kvizu. Ale druhý projde celé pole. Horší ovšem je, že podle něj mají všechny položky hodnotu 20. A ještě horší je, že se na konci pole nezastaví!

Chyba je samozřejmě v následující testovací podmínce:

```
quizscores[i] = 20
```

Zaprvé, tento výraz je vždy nenulový a tudíž pravdivý jednoduše proto, že přiřazuje nenulovou hodnotu prvku pole. Zadruhé, protože výraz přiřazuje hodnoty prvkům pole, skutečně mění data. Zatřetí, jelikož testovací výraz zůstává pravdivým, program mění data i za koncem pole. Prostě pokračuje v ukládání dalších hodnot 20 do paměti! A to není vůbec dobré.

Obtížnost tohoto druhu chyby spočívá v tom, že kód je syntakticky správný, takže ho překladač neoznačí jako chybný. (Jenomže programátoři v C a C++ tuto chybu již dělají spoustu let, což nakonec přesvědčilo mnoho překladačů k tomu, aby vypisovaly varování, které se ptá, zda takovouto činnost opravdu míníte vykonat.)

UPOZORNĚNÍ

Pro porovnávání nepoužívejte =, ale ==.

Jazyk C i C++ poskytují více volnosti než většina programovacích jazyků. Je to ovšem za cenu větší zodpovědnosti na straně programátorů. Pouze dobře napsaný program nepřekročí hranice standardního pole jazyka C++. Nicméně pomocí tříd jazyka C++ je možné

navrhnout chráněný typ pole, který takovéto chyby nedovolí. Příklad si ukážeme v kapitole 13 „Dědičnost tříd". Prozatím byste měli v případě potřeby zabudovat podobnou ochranu do vašich programů. Například cyklus by měl obsahovat test, který by zabránil přístupu za poslední člen. To může být užitečné dokonce i pro dobře napsaný cyklus. Kdyby měly všechny výsledky hodnotu 20, také by došlo k překročení hranice pole. Zkrátka, cyklus by měl testovat hodnoty pole a index pole. Kapitola 6 „Příkazy větvení a logické operátory" ukazuje spojení takovýchto dvou testů do jediné podmínky pomocí logických operátorů.

Porovnávání řetězců

Předpokládejme, že chceme vědět, zda znakové pole obsahuje řetězec mate. Pokud word reprezentuje jméno pole, potom následující test zřejmě nebude dělat to, co bychom potřebovali:

```
word == "mate"
```

Vzpomeňte si, že jméno pole je synonymem své adresy. Podobně představuje synonymum své adresy i znaková konstanta v uvozovkách. A tak předchozí relační výraz nezjišťuje, jestli jsou uvedené řetězce stejné, ale zda jsou uloženy na stejných adresách. Odpovědí na tuto otázku je ne, dokonce i když oba řetězce obsahují stejné znaky.

Pokud se pokusíme porovnávat řetězce pomocí relačních operátorů, nezískáme uspokojivé výsledky, protože jazyk C++ zachází s řetězci jako s adresami. Místo toho můžeme řetězce porovnávat pomocí funkce `strcmp()` z řetězcové knihovny ve stylu jazyka C. Tato funkce přebírá jako argumenty dvě adresy řetězců. To znamená, že argumenty mohou být ukazatele, řetězcové konstanty nebo jména znakových polí. Jsou-li oba řetězce stejné, funkce `strcmp()` vrací nulovou hodnotu. Jestliže je první řetězec v abecedě před druhým, vrací zápornou hodnotu, a pokud je první řetězec v abecedě za druhým, vrací kladnou hodnotu. Ve skutečnosti je „srovnávané pořadí na systému" přesnější než „abecední". To znamená, že znaky jsou porovnávány podle jejich systémových kódů. Například v kódu ASCII mají všechna velká písmena menší kód než malá, takže při porovnávání pořadí velká písmena předcházejí malá. Proto řetězec „`Zoo`" předchází řetězec „`aviary`". Porovnání podle hodnoty kódu také způsobuje odlišnost velkých a malých písmen, takže například řetězce „`FOO`" a „`foo`" nejsou považovány za stejné.

V některých jazycích, jako je například Basic a standardní Pascal, se řetězce uložené v polích o různé velikosti nemohou rovnat. Ale řetězce stylu C jsou definovány ukončovacím znakem null, nikoli velikostí pole, ve kterém jsou uloženy. To znamená, že dva řetězce mohou být stejné, i když jsou uloženy v různě velkých polích:

```
char big[80] = "Daffy";      // 5 písmen plus \0
char little[6] = "Daffy";    // 5 písmen plus \0
```

Mimochodem, ačkoli nemůžeme používat relační operátory na porovnání řetězců, můžete je použít na porovnání znaků, protože znaky jsou ve skutečnosti celočíselnými typy.

Takže

```
for (ch = 'a'; ch <= 'z'; ch++)
      cout << ch;
```

je platným programovým kódem, alespoň pro znakovou sadu ASCII, pro zobrazení znaků abecedy.

Výpis 5.11 volá v testovací podmínce cyklu for funkci `strcmp()`. Tento program zobrazí slovo, změní jeho první písmeno, zobrazí ho znovu a tak pokračuje, dokud funkce `strcmp()` neurčí, že je slovo stejné jako řetězec „`mate`“. Všimněte si, že výpis vkládá `cstring`, protože tento soubor dodává prototyp funkce `strcmp()`.

Výpis 5.11 compstr.cpp

```
// compstr.cpp -- porovnání řetězců
#include <iostream>
#include <cstring> // prototyp pro strcmp()
int main()
{
   using namespace std;
   char word[5] = "?ate";

   for (char ch = 'a'; strcmp(word, "mate"); ch++)
   {
      cout << word << endl;
      word[0] = ch;
   }
   cout << "Po skonceni cyklu je slovo " << word << endl;
   return 0;
}
```

KOMPATIBILITA

Možná budete muset vložit soubor `string.h` místo cstring. Programový kód ve výpisu 5.11 také předpokládá, že systém používá znakovou kódovou sadu ASCII. V této sadě jsou kódy pro písmena `a` až `z` za sebou a kód znaku `?` je umístěn bezprostředně před kódem znaku `a`.

Zde je výstup z programu na výpisu 5.11:

```
?ate
aate
bate
cate
date
eate
fate
gate
hate
iate
jate
kate
late
Po skonceni cyklu je slovo mate
```

Poznámky k programu

Program na výpisu 5.11 má několik zajímavých vlastností. Jednou z nich je samozřejmě testování. Chceme, aby cyklus pokračoval tak dlouho, dokud pole word neobsahuje řetě-

zec `mate`. To znamená, že chceme, aby test pokračoval, dokud funkce `strcmp()` neřekne, že jsou oba řetězce stejné. Zde se přímo nabízí podmínka:

```
strcmp(word, "mate") != 0   // řetězce nejsou stejné
```

Výše uvedený výraz má hodnotu 1 (`true`), jestliže se řetězce nerovnají, a 0 (`false`), pokud se rovnají. Ale co samostatný výraz `strcmp(word, "mate")`? Ten má nenulovou hodnotu (`true`), jestliže se řetězce nerovnají, a hodnotu 0 (`false`), pokud se rovnají. V podstatě funkce vrací `true`, jestliže se řetězce nerovnají, a `false`, pokud jsou stejné. Místo celého relačního výrazu tedy můžeme použít pouze funkci. Tak získáme stejné chování a kratší zápis. Tímto způsobem také programátoři jazyka C a C++ obvykle funkci `strcmp()` používají.

ZAPAMATUJTE SI

Pro testování rovnosti a pořadí řetězců používejte funkci strcmp(). Výraz

```
strcmp(str1, str2) == 0
```

je pravdivý, jestliže jsou řetězce `str1` a `str2` stejné. Výrazy

```
strcmp(str1, str2) != 0
```

a

```
strcmp(str1, str2)
```

jsou pravdivé, jestliže řetězce `str1` a `str2` stejné 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` následuje řetězec `str2`. Funkce `strcmp()` tedy může hrát roli operátorů ==, !=, < a > podle toho, jak vytvoříte testovací podmínku.

Program compstr.cpp používá pro postupné vkládání za sebou jdoucích znaků v abecedě do proměnné ch operátor inkrementace:

```
ch++
```

Operátory inkrementace a dekrementace můžeme použít se znakovými proměnnými, protože typ char je ve skutečnosti celočíselným typem, takže operace vlastně mění celočíselný kód uložený v proměnné. Také si všimněte, že pomocí indexu pole lze jednoduše měnit jednotlivé znaky v řetězci:

```
word[0] = ch;
```

Porovnávání řetězců třídy string

S třídou `string` je život o něco jednodušší než s řetězci ve stylu C, neboť třída umožňuje porovnávání pomocí relačních operátorů. To je možné proto, že si definujete funkce třídy, které „převáží" neboli předefinují operátory. V kapitole 12, „Třídy a dynamické přidělování paměti" je uvedeno, jak se tyto vlastnosti začleňují do tříd, avšak z praktického hlediska zatím stačí, když víte, že s objekty třídy `string` můžete používat relační operátory. Na výpisu 5.12 je tatáž úloha jako na výpisu 5.11, avšak znaková pole (typu `char`) jsou nahrazena objekty `string`.

Výpis 5.12 compstr2.cpp

```
// compstr2.cpp -- comparing strings using arrays
#include <iostream>
#include <string>  // třída string
int main()
{
    using namespace std;
    string word = "?ate";

    for (char ch = 'a'; word != "mate"; ch++)
    {
        cout << word << endl;
        word[0] = ch;
    }
    cout << "Po skonceni cyklu je slovo " << word << endl;
    return 0;
}
```

Výstup z programu na výpisu 5.12 je stejný jako výstup z programu na výpisu 5.11.

Poznámky k programu

Testovací podmínka ve výpisu 5.12

```
word != "mate"
```

používá na levé straně s řetězcovým objektem relační operátor a na pravé straně řetězec typu C. Operátor `!=` předefinovaný třídou `string` můžeme používat, dokud je alespoň jedním operátorem objekt typu `string`; druhý operand může být jak objekt `string`, tak i řetězec typu C.

Ve třídě je možno při porovnávání v testovacím výrazu používat objekt typu `string` jako samostatnou entitu, anebo jako agregovaný objekt, v němž můžete jednotlivé znaky zpracovávat pomocí zápisu jako u běžného pole.

Na závěr je dobré podotknout, že na rozdíl od většiny doposud uvedených cyklů `for`, dva poslední cykly nejsou počítacími cykly. To znamená, že neopakují provádění bloku příkazů v daném počtu průchodů. Místo toho cykly probíhají, dokud nenastanou určité podmínky (v poli `word` je řetězec „`mate`“). Programy jazyka C++ používají častěji pro tento druh testů cykly `while`, které si probereme nyní.

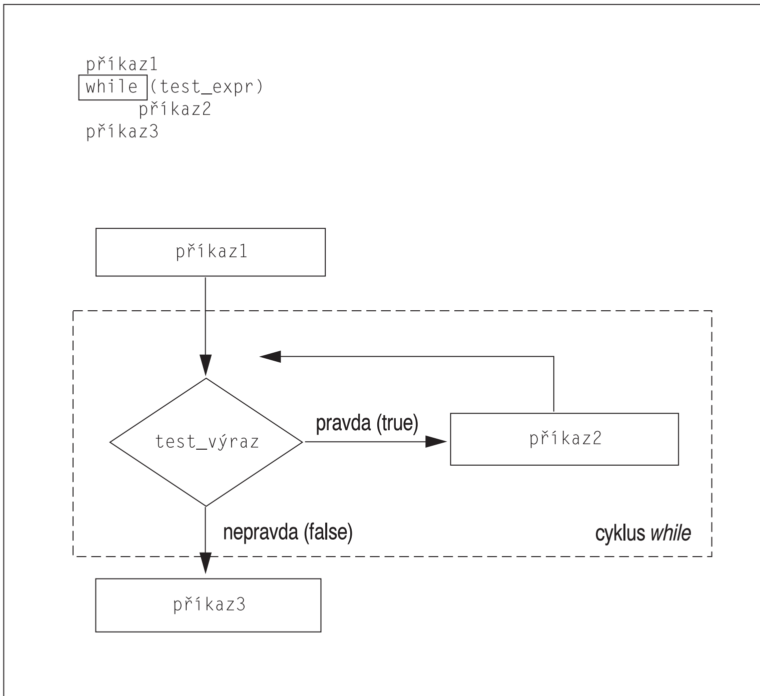
Cyklus while

Cyklus `while` je vlastně cyklus `for` zbavený inicializační a aktualizační části; má pouze testovací podmínku a tělo.

```
while (testovací-podmínka)
    tělo
```

Program nejprve vyhodnotí výraz *testovací-podmínka*. Pokud má tento výraz hodnotu `true`, dojde k vykonání příkazu(ů) v těle. Stejně jako u cyklu `for` tělo obsahuje jeden příkaz nebo blok definovaný pomocí složených závorek. Po ukončení těla se program vrátí na testovací podmínku a znovu ji vyhodnotí. Jestliže je podmínka nenulová, program opět vykoná tělo. Tento cyklus testování a vykonávání pokračuje, dokud nemá testovací podmínka hodnotu `false`. (Viz obrázek 5.3.) Je zřejmé, že pokud chceme, aby cyklus někdy

skončil, musíme v těle cyklu provádět činnosti ovlivňující výraz *testovací-podmínka*. Cyklus může například inkrementovat proměnnou, která je použita v testovací podmínce nebo přečíst novou hodnotu ze vstupu z klávesnice. Podobně jako cyklus `for` je i cyklus `while` cyklem se vstupní podmínkou. Tedy jestliže je *testovací-podmínka* na začátku vyhodnocena jako `false`, program nikdy neprovede tělo cyklu.



Obrázek 5.3 Cyklus while

Ve výpisu 5.13 můžeme vidět příklad práce cyklu `while`. Tento cyklus prochází všechny znaky řetězce a zobrazuje aktuální znak a jeho kód ASCII. Cyklus je ukončen při dosažení nulového znaku. Tato technika procházení řetězce znak po znaku až do nulového znaku představuje standardní metodu jazyka C++ pro zpracovávání řetězců. Protože řetězce obsahují ukončovací značku, programy většinou nepotřebují informaci o délce řetězce.

Výpis 5.13 while.cpp

```
// while.cpp - úvod do cyklu while
#include <iostream>
const int ArSize = 20;
int main()
{
   using namespace std;
   char name[ArSize];

   cout << "Vase krestni jmeno, prosim: ";
```

```
    cin >> name;
    cout << "Zde je vase jmeno svisle s ASCII kody:\n";
    int i = 0;                    // start na začátku řetězce
    while (name[i] != '\0')  // zpracování do konce řetězce
    {
        cout << name[i] << ": " << int(name[i]) << endl;
        i++;                      // na tento krok nezapomeňte
    }
    return 0;
}
```

Zde je příklad běhu programu na výpisu 5.13:

```
Vase krestni jmeno, prosim: Muffy
Zde je vase jmeno svisle s ASCII kody:
M: 77
u: 117
f: 102
f: 102
y: 121
```

(Reálná slova ani pseudoslova nepíšeme s kódy ASCII svisle, ale takovýto výstup vypadá mile technicky.)

Poznámky k programu

Podmínka while vypadá následovně:

```
while (name[i] != '\0')
```

Testuje, zda je určitý znak pole roven nulovému znaku. Aby byl tento test nakonec úspěšný, musí tělo cyklu měnit hodnotu proměnné i. Což dělá pomocí inkrementace i na konci těla cyklu. Při vynechání tohoto kroku by cyklus pracoval se stejným prvkem pole a tiskl by stále stejný znak a jeho kód, dokud by se vám nepodařilo program ukončit. Takovýto nekonečný cyklus je jedním z nejběžnějších problémů cyklů. Často ho můžete vytvořit opomenutím změny nějaké hodnoty uvnitř těla cyklu.

Řádek s podmínkou `while` můžeme přepsat tímto způsobem:

```
while (name[i])
```

S takto změněným kódem program pracuje stejně jako dříve. Je tomu tak proto, že pokud `name[i]` obsahuje obvyklý znak, má hodnotu kódu znaku, který je nenulový, neboli `true`. Ale jestliže `name[i]` obsahuje nulový znak, hodnota kódu tohoto znaku je 0, neboli `false`. Takovýto zápis je mnohem zhuštěnější (a častěji používaný), ale oproti předchozímu méně názorný. Jednodušší překladače mohou generovat rychlejší kód pro druhou verzi, ale chytré překladače vytvářejí pro obě verze stejný kód.

Aby program vytiskl kód ASCII znaku, přetypujeme hodnotu proměnné `name[i]` na celé číslo. Objekt `cout` tiskne takto získanou hodnotu jako celé číslo a neinterpretuje ji jako kód znaku.

Na rozdíl od řetězce typu C třída `string` nepoužívá k identifikaci konce řetězce nulový znak, takže výpis 5.13 můžete snadno změnit tak, že znakové pole nahradíte řetězcovým objektem. Postupy pro identifikaci posledního znaku v řetězci v objektech typu `string` jsou popsány v kapitole 16.

Porovnání cyklů for a while

V jazyce C++ jsou cykly `for` a `while` v podstatě totožné. Například cyklus `for`

```
for (inicializační-výraz; testovací-výraz; aktualizační-výraz)
{
   příkaz(y)
}
```

lze zapsat takto:

```
inicializační-výraz;
while (testovací-výraz)
{
   příkaz(y)
   aktualizační-výraz;
}
```

Podobně lze tento cyklus `while`

```
while (testovací-výraz)
      tělo
```

zapsat takto:

```
for ( ;testovací-výraz;)
      tělo
```

Cyklus for vyžaduje tři výrazy (nebo techničtěji řečeno, jeden příkaz následovaný dvěma výrazy), ale mohou to být prázdné výrazy (příkazy). Povinné jsou pouze dva středníky. Abychom nezapomněli, chybějící testovací výraz v cyklu `for` je považován za pravdivý (`true`), takže následující cyklus se nikdy neukončí:

```
for ( ; ; )
      tělo
```

Protože jsou cykly `while` a `for` téměř totožné, závisí pouze na vašem programovacím stylu, který z nich použijete. (Je zde jemný rozdíl, jestliže tělo obsahuje příkaz `continue`, o kterém si povíme v kapitole 6.) Programátoři většinou používají cyklus `for` pro čítací cykly, protože jeho formát umožňuje umístit všechny důležité informace – počáteční hodnotu, koncovou hodnotu a způsob aktualizace počitadla – na jedno místo. Cyklus `while` používají častěji tehdy, když předem přesně nevědí, kolikrát cyklus proběhne.

TIP

Při návrhu cyklu je dobré dodržovat následující zásady:

- **Určete podmínku ukončující cyklus.**
- **Tuto podmínku inicializujte před prvním testem.**
- **Aktualizujte vybranou podmínku v každém průchodu cyklem před dalším testem.**

U cyklu `for` je příjemné, že jeho struktura umožňuje implementaci těchto tří zásad, čímž přispívá k jejich dodržování.

Chybné rozvržení

Těla obou cyklů `for` a `while` se skládají z jediného příkazu umístěného za výrazy v závorkách. Jak jste již viděli, tímto jediným příkazem může být blok několika příkazů. Myslete na to, že blok definují složené závorky, nikoli odsazení. Podívejme se například na následující cyklus:

```
i = 0;
while (name[i] != '\0')
  cout << name[i] << endl;
  i++;
cout << "Hotovo\n";
```

Odsazení nám napovídá, že autor programu chtěl, aby příkaz `i++;` byl součástí těla cyklu. Nepřítomnost složených závorek však říká překladači, že tělo obsahuje pouze první příkaz `cout`. Tento program tedy stále tiskne první znak pole a nikdy nedojde k příkazu `i++;`, který leží mimo cyklus.

Následující příklad ukazuje další skryté nebezpečí:

```
i = 0;
while (name[i] != '\0'); // sem středník nepatří
{
            cout << name[i] << endl;
            i++;
}
cout << "Hotovo\n";
```

Tentokrát jsou v kódu složené závorky umístěny dobře, ale je zde navíc středník. Vzpomeňte si, že středníky označují konec příkazu, takže tento středník vlastně ukončuje cyklus while. Jinými slovy, tělem cyklu je *prázdný příkaz*, to znamená nic následované středníkem. Kód ve složených závorkách se nachází až za cyklem a nikdy na něj nepřijde řada. Místo toho cyklus stále probíhá a nic nedělá. Na takovéto zbloudilé středníky si musíte dávat pozor.

Okamžik – jak se dělá cyklus s časovým zpožděním

Občas je užitečné zabudovat do programu časové zpoždění. Možná jste se setkali s programy, které na obrazovku vypíší zprávu a okamžitě pokračují ve vykonávání něčeho jiného dříve, než si zprávu můžete přečíst. Takto nemáte jistotu, zda jste nepropásli nějakou důležitou informaci. Bylo by daleko lepší, kdyby se program před pokračováním na pět sekund zastavil. Pro realizaci takovéhoto požadavku je vhodný cyklus `while`. Jednou z dřívějších technik bylo vytvoření čítače, který spotřebovával čas:

```
long wait = 0;
while (wait < 10000)
    wait++;         // tiché počítání
```

Problém takového přístupu spočívá v nutnosti měnit při změně rychlosti procesoru počítače mezní hodnoty počitadla. Různé hry napsané například pro původní IBM PC přestaly být na rychlejších procesorech ovladatelné. Lepším přístupem je přenechat časování systémovým hodinám.

Knihovny ANSI C a C++ nabízejí užitečnou funkci, která se jmenuje `clock()` a vrací systémový čas uplynulý od spuštění programu. Je zde však několik problémů. Zaprvé, funkce `clock()` nemusí vracet čas v sekundách. Zadruhé, typ návratové hodnoty může být na některých systémech `long`, na jiných `unsigned long` nebo na některých i jiný.

Avšak hlavičkový soubor `ctime` (`time.h` na starších implementacích) poskytuje řešení těchto problémů. Zaprvé, definuje symbolickou konstantu `CLOCKS_PER_SEC`, která se rovná počtu systémových časových jednotek za sekundu. Takže vydělením systémového času touto hodnotou získáme sekundy. Nebo můžeme sekundy násobit `CLOCKS_PER_SEC` a obdržet tak čas v systémových jednotkách. Zadruhé, ctime stanovuje `clock_t` jako zástupné jméno pro návratový typ funkce `clock()`. (Viz poznámka o zástupných jménech typů.) To znamená, že můžeme deklarovat proměnnou typu `clock_t` a překladač ji převede na `long` nebo `unsigned int` nebo na jakýkoliv jiný pro odpovídající systém správný typ.

KOMPATIBILITA

Systémy, které nedodávají hlavičkový soubor `ctime`, mohou místo toho použít `time.h`. Některé implementace mohou mít s programem `waiting.cpp` problémy, pokud knihovna implementace není plně kompatibilní s ANSI C. Je tomu tak proto, že funkce `clock()` je dodatkem ANSI do tradiční knihovny C. Také některé nezralé implementace ANSI C používaly `CLK_TCK` nebo `TCK_CLK` namísto delší symbolické konstanty `CLOCKS_PER_SEC`. Některé starší verze C++ nerozeznávají žádnou z těchto definovaných konstant. Některá prostředí (jako například MSVC++ 1.0, ale ne MSVC++ 5.0) mají potíže se znakem alarm `\a` a sladěním obrazovky s časovým zpožděním.

Výpis 5.14 předvádí použití funkce `clock()` a hlavičky `ctime` při vytváření zpožďovací smyčky.

Výpis 5.14 waiting.cpp

```
// waiting.cpp -- použití funkce clock() ve zpožďovací smyčce
#include <iostream>
#include <ctime> // popisuje funkci clock() a typ clock_t
int main()
{
   using namespace std;
   cout << "Zadejte zpozdeni v sekundach: ";
   float secs;
   cin >> secs;
   clock_t delay = secs * CLOCKS_PER_SEC;   // převede na hodinové takty
   cout << "zacatek\a\n";
   clock_t start = clock();
   while (clock() - start < delay )            // čekání do vypršení času
   ;                                           // všimněte si středníku
cout << "konec \a\n";
return 0;
}
```

Výpočtem času zpoždění v systémových jednotkách místo v sekundách se program na výpisu 5.14 vyhýbá nutnosti převádět systémový čas na sekundy v každém průchodu cyklem.

Zástupná jména typů

C++ má dva způsoby stanovení nového jména jako zástupného jména typu. První spočívá v použití preprocesoru:

```
#define BYTE char     // preprocesor nahrazuje konstantu BYTE typem char
```

Při překladu programu preprocesor nahrazuje všechny výskyty konstanty BYTE typem `char`, tedy vytváří pro `char` zástupné jméno `BYTE`.

Druhým způsobem vytvoření zástupného jména je v C++ (a C) použití klíčového slova `typedef`. Pokud například chceme, aby byte bylo zástupným jménem typu `char`, napíšeme následující kód:

```
typedef char byte;    // typ char má zástupné jméno byte
```

Zde je obecný tvar:

```
typedef jménoTypu zástupnéJméno;
```

Jinými slovy, pokud chceme, aby *zástupnéJméno* zastupovalo určitý typ, deklarujeme *zástupnéJméno* jako by to byla proměnná tohoto typu a před deklaraci napíšeme klíčové slovo `typedef`. Jestliže například chceme, aby `byte_pointer` zastupoval ukazatel na typ `char`, deklarujeme `byte_pointer` jako ukazatel na `char` a dopředu přidáme `typedef`:

```
typedef char * byte_pointer;       // ukazatel na typ char
```

Pomocí `#define` můžeme zkusit něco podobného, ale při použití seznamu proměnných to nebude fungovat. Podívejme se například na následující kód:

```
#define FLOAT_POINTER float *
FLOAT_POINTER pa, pb;
```

Preprocesor nahradí deklaraci takto:

```
float * pa, pb;       // pa je ukazatel na float, pb pouze float
```

Při použití klíčového slova `typedef` tento problém odpadá. Vzhledem k tomu, že `typedef` umí zpracovat složitější typy zástupných jmen, je vhodnější než `#define` – a někdy je možností jedinou.

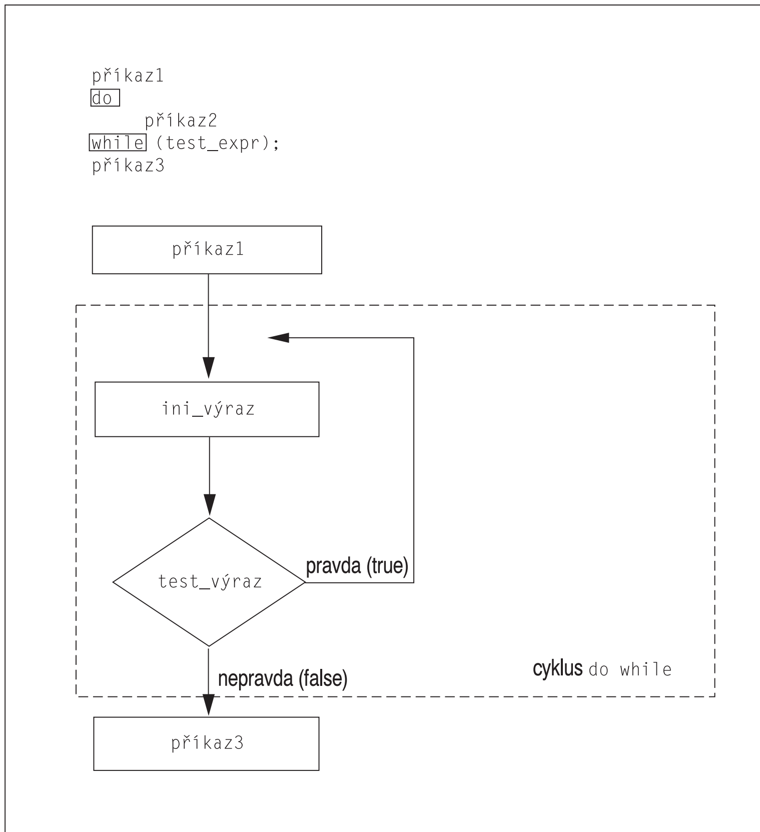
Všimněte si, že `typedef` nevytváří nový typ. Definuje pouze pro starý typ nové jméno. Je-li `word` zástupným jménem typu `int`, objekt `cout` bude s hodnotou typu `word` zacházet jako s `int`, kterou ve skutečnosti vlastně je.

Cyklus do while

Doposud jste viděli cykly `for` a `while`. Třetím cyklem jazyka C++ je `do while`. Liší se od obou předchozích, protože je cyklem s výstupní podmínkou. To znamená, že tento bezstarostný cyklus nejprve provede své tělo a teprve potom vyhodnotí testovací výraz, aby věděl, zda má pokračovat dalším průchodem. Když je podmínka vyhodnocena jako `false`, cyklus končí; jinak začíná nový cyklus provedení a testování. Takovýto cyklus je vždy vykonán alespoň jednou, protože jeho programový tok musí před testem projít tělo cyklu. Zde je syntaxe cyklu `do while`:

```
do
   tělo
while (testovací-výraz);
```

Tělo může být jediný příkaz nebo blok příkazů ohraničený složenými závorkami. Obrázek 5.4 shrnuje tok programu cyklem `do while`.



Obrázek 5.4 Struktura cyklů `do while`

Obvykle je zvolení cyklu se vstupní podmínkou lepší než cyklu s výstupní podmínkou, protože vstupní podmínka je ověřována před průběhem cyklu. Předpokládejme, že bychom například ve výpisu 5.13 použili `do while` namísto `while`. Potom by cyklus vytiskl nulový znak a jeho kód předtím, než by zjistil, že již dosáhl konce řetězce. Ale občas má použití cyklu `do while` svůj význam. Když například žádáme uživatele o vstup, program ho musí před otestováním získat. Výpis 5.15 předvádí použití cyklu `do while` v takové situaci.

Výpis 5.15 dowhile.cpp

```
// dowhile.cpp -- výstupní podmínka cyklu
#include <iostream>
int main()
{
   using namespace std;
   int n;
```

```
    cout << "Zadavejte cisla v rozmezi 1-10, abyste nasli ";
    cout << "me oblibene cislo\n";
    do
    {
       cin >> n;           // provádí tělo
    } while (n != 7);  // potom testuje
    cout << "Ano, 7 je me oblibene cislo.\n" ;
    return 0;
}
```

Zde je ukázka běhu programu z výpisu 5.15:

```
Zadavejte cisla v rozmezi 1-10, abyste nasli me oblibene cislo
9
4
7
Ano, 7 je me oblibene cislo.
```

Z praxe: Podivné cykly for

Není to příliš běžné, ale občas se můžete setkat s kódem, který se podobá následujícímu:

```
for(;;)      // někdy mu říkáme "nekonečný cyklus"
{
  I++;
  // nějaká činnost ...
  if (30 >= I) break;
}
```

nebo této variantě:

```
for(;;I++)
{
  if (30 >= I) break;
             // nějaká činnost ...
}
```

Výše uvedený kód je založený na skutečnosti, že prázdná testovací podmínka je ve smyčce `for` považována za pravdivou. Žádný z těchto příkladů není snadné číst a neměl by být používán jako obecný model psaní cyklů. Funkčnost prvního příkladu můžeme mnohem srozumitelněji vyjádřit pomocí cyklu `do while`:

```
do {
  I++;
  // nějaká činnost ...
} while (30 < I);
```

Obdobně může být druhý příklad srozumitelněji vyjádřen cyklem `while`:

```
while (I < 30)
{
  // nějaká činnost ...
  I++;
}
```

Obecně platí, že psaní srozumitelného, snadno pochopitelného, kódu je mnohem užitečnější než přehlídka schopností používat málo známé vlastnosti jazyka.

Cykly a vstup textu

Když už nyní víte, jak cykly pracují, podívejme se na jednu z nejběžnějších a nejdůležitějších úloh, která se k nim vztahuje: čtení textu znak po znaku ze souboru nebo z klávesnice. Můžete například chtít napsat program, který zjišťuje počet znaků, řádků a slov na vstupu. Jazyky C++ i C tradičně používají pro tento druh úlohy cyklus `while`. Nyní se podíváme, jak to dělají. I přes to, že jazyk C již znáte, neprocházejte tuto část knihy příliš rychle. Ačkoli je cyklus `while` v C++ stejný jako v C, vlastnosti V/V jsou v C++ jiné. To může dodat cyklu napsanému v jazyce C++ poněkud odlišný vzhled. Objekt `cin` ve skutečnosti podporuje tři různé režimy vstupu po jednom znaku, z nichž má každý jiné uživatelské rozhraní. Podívejme se, jak můžeme tyto možnosti použít v cyklech `while`.

Použití prostého objektu cin pro vstup

Chceme-li pro načítání textového vstupu z klávesnice použít cyklus, musíme vědět, kdy máme skončit. Jak se to ale dozvíme? Jedním způsobem je výběr určitého speciálního znaku, kterému někdy říkáme *indikátor*, jenž bude sloužit jako znamení zastav. Například ve výpisu 5.16 je načítání vstupu zastaveno, když program narazí na znak #. Tento program pročítá množství znaků, které načítá a zobrazuje. To znamená, že znovu zobrazuje znaky, které byly načteny. (Stisknutí klávesy klávesnice neumisťuje znaky na obrazovku automaticky, tuto práci musí vykonávat programy. Obvykle se o to postará operační systém. V našem případě vypisuje znaky operační systém i testovaný program.) Před ukončením vypisuje program zprávu o počtu zpracovaných znaků. Tento program se nachází ve výpisu 5.16.

Výpis 5.16 textin1.cpp

```
// textin1.cpp -- čtení znaků cyklem while
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;              // použije základní vstup
   cout << " Zadejte znaky; zadejte # pro ukonceni:\n";
   cin >> ch;                  // získá znak
   while (ch != '#')           // test znaku
   {
      cout << ch;              // zobrazení znaku
      ++count;                 // přičtení znaku
      cin >> ch;               // získá další znak
   }
   cout << endl << count << " znaku nacteno\n";
   return 0;
}
```

Zde je ukázka běhu programu z výpisu 5.16:

```
Zadejte znaky; zadejte # pro ukonceni:
videl jsem Kena bezet#velmi rychle
videljsemKenabezet
18 znaku nacteno
```

Ken zřejmě běží tak rychle, že sám vymaže mezery nebo alespoň znaky mezer na vstupu..

Poznámky k programu

Nejprve se podíváme na strukturu programu na výpisu 5.16. Tento program načte první vstupní znak před začátkem cyklu, aby ho mohl otestovat. To je důležité, protože prvním znakem může být #. Jelikož textin1.cpp používá cyklus se vstupní podmínkou, program v takovém případě správně přeskočí celý cyklus. A protože byla proměnná count dříve nastavena na nulu, count bude mít 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, ale s jeho pomocí pokračuje dalším znakem. Všimněte si, že program dodržuje dříve uvedené zásady. Podmínka ukončující cyklus nastane, pokud je posledním přečteným znakem #. Inicializace této podmínky proběhne před začátkem cyklu a její aktualizace načtením dalšího znaku na konci cyklu.

Tento postup vypadá správně. Tak proč program vynechává na výstupu mezery? Může za to objekt `cin`. Při čtení hodnot typu `char`, stejně jako dalších základních typů, přeskakuje mezery a znaky nového řádku. Mezery na vstupu nejsou vypisovány ani počítány.

Aby se to celé ještě více zkomplikovalo, vstup do objektu `cin` je nejprve ukládán do vyrovnávací paměti. To znamená, že zadané znaky jsou odeslány programu až po nestisknutí klávesy ENTER. Proto můžeme zadávat další znaky i po znaku #. Po stisknutí klávesy ENTER je celá řada znaků odeslána programu, ale program ukončí zpracování vstupu po dosažení znaku #.

Řešení je v cin.get(char)

Programy, které čtou znak po znaku, obvykle potřebují otestovat každý znak včetně mezer, tabulátorů a znaku nového řádku. Třída `istream` (definovaná v `iostream`), do které `cin` patří, obsahuje členské funkce, jež tyto potřeby splňují. Konkrétně členská funkce `cin.get(ch)` načte ze vstupu následující znak, i když jím je mezera, a přiřadí ho proměnné `ch`. Nahrazením `cin>>ch` tímto voláním funkce můžeme opravit výpis 5.16. Na výpisu 5.17 vidíme výsledek.

Výpis 5.17 textin2.cpp

```
//textin2.cpp -- použití funkce cin.get(char)
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;

   cout << "Zadejte znaky; zadejte # pro ukonceni:\n";
   cin.get(ch);              // použijte funkci cin.get(ch)
   while (ch != '#')
   {
      cout << ch;
      ++count;
      cin.get(ch);           // použijte ji znovu
   }
   cout << endl << count << " znaku nacteno\n";
   return 0;
}
```

Zde je ukázka běhu programu z výpisu 5.17:

```
Pouzil jste #2 tuzku?
Pouzil jste
12 znaku nacteno
```

Nyní program zobrazuje a počítá všechny znaky včetně mezer. Vstup se ukládá do vyrovnávací paměti, takže je možné zadat více vstupních dat, než program nakonec zpracuje.

Pokud znáte jazyk C, může se vám zdát, že je tento program napsaný špatně! Volání `cin.get(ch)` vkládá hodnotu do proměnné `ch`, což znamená, že se její hodnota mění. Chceme-li měnit hodnotu proměnné v jazyce C, je nutné předat funkci adresu této proměnné. Avšak volání `cin.get()` ve výpisu 5.17 předává `ch`, nikoli `&ch`. V jazyce C nebude podobně napsaný programový kód fungovat. Ale v C++ bude pracovat správně, pokud funkce deklaruje odpovídající argument jako *odkaz (referenci)*. To je nový typ jazyka C++. Hlavičkový soubor `iostream` deklaruje argument funkce `cin.get(ch)` jako referenční typ, takže tato funkce může měnit hodnotu svého parametru. Podrobnosti si vysvětlíme v kapitole 8 „Rozšířené možnosti funkcí". Do té doby mohou zůstat odborníci na jazyk C klidní, předávání argumentů v C++ pracuje obvykle stejně jako v C. Avšak pro `cin.get(ch)` to neplatí.

Která verze cin.get()?

Ve výpisu 4.5 v kapitole 4 jsme použili tento programový kód:

```
char name[ArSize];
...
cout << "Zadejte vase jmeno:\n";
cin.get(name, ArSize).get();
```

Poslední řádek můžeme přepsat pomocí těchto dvou volání:

```
cin.get(name, ArSize);
cin.get();
```

Jedna verze `cin.get()` přebírá dva argumenty: jméno pole, což je adresa řetězce (typ `char *`), a `ArSize`, jenž představuje celé číslo typu `int`. (Vzpomeňte si, že jméno pole je adresou jeho prvního prvku, takže jméno znakového pole je typu `char *`.) Program potom používá `cin.get()` bez argumentů. A docela nedávno jsme funkci `cin.get()` volali také následujícím způsobem:

```
char ch;
cin.get(ch);
```

Tentokrát má funkce `cin.get()` jeden argument typu `char`.

Znalci jazyka C mohou znovu žasnout nebo pochybovat. Pokud přebírá funkce v jazyce C argumenty ukazatel na `char` a `int`, nemůžeme stejnou funkci volat s jedním argumentem jiného typu. Ale v C++ to možné je, protože jazyk podporuje vlastnost OOP, která se nazývá *přetěžování funkcí*. Přetěžování funkcí umožňuje vytvářet různé funkce se stejným jménem za předpokladu, že mají různé argumenty. Pokud například v C++ zavoláme `cin.get(name, ArSize)`, překladač nalezne verzi `cin.get()`, která pracuje s argumenty `char *` a `int`. Ale když použijeme `cin.get(ch)`, překladač dodá verzi s jedním argumentem typu `char`. A jestliže programový kód neposkytne žádný argument, překladač zvolí verzi funkce `cin.get()` bez argumentů. Přetěžování funkcí umožňuje použít stejné jméno pro příbuzné funkce, jež provádějí tentýž základní úkol různými způsoby nebo pro různé

typy. To je další téma, které na nás čeká v kapitole 8. Mezitím si přetěžování funkcí předvedeme na příkladech zabývajících se třídou `istream`. Pro odlišení různých verzí této funkce budeme uvádět seznam argumentů. Tedy `cin.get()` označuje verzi bez argumentů a `cin.get(ch)` verzi, která má jeden argument.

Podmínka konce souboru

Jak ukazuje výpis 5.17, použití symbolu, jako je například `#`, pro označení konce vstupu, není vždy vhodné, protože tento symbol může být součástí vstupu. To platí i pro jiné libovolně zvolené symboly, například `@` nebo `%`. Nachází-li se vstup v souboru, můžeme použít mnohem výkonnější techniku – nalezení konce souboru (end-of-file – EOF). Prostředky vstupu jazyka C++ ve spolupráci s operačním systémem rozpoznají, kdy vstup dosáhl konce souboru a oznamují tuto informaci zpět programu.

Na první pohled se může zdát, že čtení informací ze souborů nemá mnoho společného s objektem `cin` a vstupem z klávesnice, ale jsou zde dvě souvislosti. Zaprvé, mnoho operačních systémů, včetně Unixu a MS-DOS, podporuje *přesměrování*, které umožňuje nahradit vstup z klávesnice souborem. Pokud máme například v MS-DOS spustitelný program `gofish.exe` a textový soubor `fishtale`, potom můžeme za výzvu Dosu napsat tento příkaz:

```
gofish < fishtale
```

Tak řekneme programu, aby bral vstup ze souboru fishtale místo z klávesnice. Symbol `<` je operátorem přesměrování pro UNIX i DOS.

Zadruhé, mnoho operačních systémů dovoluje simulovat konec souboru z klávesnice. Na Unixu tak učiníme stisknutím Ctrl+D na začátku řádku. V Dosu stiskneme kdekoli na řádku Ctrl+Z, Enter. Některé implementace podporují podobné chování, dokonce i když toto chování není součástí operačního systému. Používání konce souboru při vstupu z klávesnice je ve skutečnosti odkazem prostředí s příkazovým řádkem. Nicméně Symantec C++ pro Mac imituje UNIX a rozpoznává Ctrl+D jako simulovaný EOF. Metrowerks Codewarrior rozeznává Ctrl+Z v prostředích Macintosh a Windows. Microsoft Visual C++ 7.0, Borland C++ 5.5 a GNU C++ pro PC rozpoznají Ctrl+Z, když je prvním znakem na řádku, avšak musí následovat Enter. Stručně řečeno, řada programových prostředí na PC rozpoznává Ctrl+Z jako simulovaný EOF, avšak přesné detaily (zda kdekoli na řádku nebo jen jako první znak, zda musí být následován znakem Enter) se mohou lišit.

Pokud programovací prostředí umí EOF, můžeme u programů podobných programu na výpisu 5.17 používat přesměrování souborů a vstup z klávesnice se simulovaným koncem souboru. To vypadá užitečně, a tak se podíváme na podrobnosti.

Když objekt `cin` zjistí konec souboru (EOF), nastaví dva bity (*eofbit* a *failbit*) na hodnotu 1. Zjistit, zda byl `eofbit` nastaven, můžeme pomocí členské funkce `eof()`; volání `cin.eof()` vrací hodnotu `true` typu `bool`, pokud byl detekován EOF, a jinak `false`. Obdobně vrací hodnotu `true` členská funkce `fail()`, jestliže byl na hodnotu 1 nastaven *eofbit* nebo *failbit*, jinak vrací `false`. Všimněte si, že metody `eof()` a `fail()` oznamují výsledek posledního pokusu o čtení; to znamená, že informují o minulosti a nehledí dopředu. Takže za pokusem o čtení by měl vždy následovat test `cin.eof()` nebo `cin.fail()`. Návrh výpisu 5.17 se tímto pravidlem řídí, ale místo `eof()` používá `fail()`, protože posledně jmenovaná metoda pracuje na více implementacích.

KOMPATIBILITA

Některé systémy nepodporují EOF simulovaný z klávesnice. Jiné systémy, včetně Microsoft Visual C++ 6.0, Metrowerks Codewarrior a Borland C++ Builder ho podporují nedokonale. Pokud používáte při zastavení výpisu na obrazovce pro její lepší čtení metodu `cin.get()`, tak vám to zde nebude fungovat, protože detekování EOF ruší další pokusy o čtení ze vstupu. Nicméně pro pozdržení viditelnosti výpisu na obrazovce můžete použít časovou smyčku jako ve výpisu 5.14.

Výpis 5.18 textin3.cpp

```
// textin3.cpp -- čtení znaků do konce souboru
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;
   cin.get(ch);                        // pokus přečíst znak
   while (cin.eof() == false)          // test na EOF
   {
      cout << ch;                      // echo znaku na displeji
      ++count;
      cin.get(ch);                     // pokus o čtení dalšího znaku
   }
   cout << endl << count << " znaku nacteno\n";
   return 0;
}
```

Zde je ukázka výstupu programu z výpisu 5.18:

```
Zeleny ptak zpiva v zime.<ENTER>
Zeleny ptak zpiva v zime.
Ano, ale vrana leta za usvitu.<ENTER>
Ano, ale vrana leta za usvitu.
<CTRL><Z>
57 znaku nacteno
```

Protože jsme program spustili pod systémem Windows XP, simulovali jsme konec souboru stisknutím kláves Ctrl+Z. Uživatelé Unixu a Linuxu by měli místo toho stisknout Ctrl+D.

Pomocí přesměrování můžeme tento program využít pro zobrazení textového souboru a vypsání počtu znaků. Níže je uvedena ukázka programu, který čte, zobrazuje a počítá znaky dvouřádkového souboru na systému UNIX ($ je unixová výzva):

```
$ textin3 < stuff
Ja jsem UNIXovy soubor
a jsem na to hrdy.
40 znaku nacteno
$
```

EOF ukončuje vstup

Vzpomeňte si, že metoda objektu cin po detekci konce souboru nastaví příznak v objektu cin, který indikuje nalezení konce souboru. Po nastavení tohoto příznaku přestane objekt cin načítat vstup a jeho další volání nemají žádný účinek. Pro vstup ze souboru to má smysl, protože za koncem souboru bychom už neměli číst. Avšak při vstupu z klávesnice můžeme pro ukončení cyklu použít simulovaný konec souboru, ale později budeme chtít číst další vstupní údaje. Tento problém řeší metoda cin.clear(), která vymaže příznak konce souboru a umožní další zpracovávání vstupu. Více si o této problematice povíme v kapitole 17 „Vstup, výstup a soubory". Mějte ale na paměti, že v některých systémech stisknutí Ctrl+Z ukončuje vstup i výstup bez možnosti jejich obnovení metodou cin.clear().

Běžné postupy při vstupu znaků

Následující program je základní konstrukcí cyklu, který čte text po znacích až po EOF:

```
cin.get(ch);                        // pokus o přečtení znaku
while (cin.eof() == false)          // test na EOF
{
    ...                             // zpracování příkazů
    cin.get(ch);                    // pokus o přečtení dalšího znaku
}
```

Některé příkazy můžeme zkrátit. Kapitola 6 představuje operátor !, který mění true na false a naopak. S jeho pomocí přepsaný test while pak bude vypadat takto:

```
while (!cin.fail())                 // když vstup nezhavaroval
```

Návratovou hodnotu metody cin.get(char) představuje objekt cin. Avšak třída istream poskytuje funkci, která konvertuje objekt istream, jako je cin, na hodnotu typu bool; tato konverzní funkce je volána, když se objekt cin vyskytuje na místě, kde je očekávána hodnota typu bool, například v testovací podmínce cyklu while. Takto konvertovaná hodnota je true, pokud byl poslední pokus o čtení úspěšný a v opačném případě false. To znamená, že test while můžeme přepsat následujícím způsobem:

```
while (cin)    // pokud je vstup úspěšný!cin.fail()
```

Tento zápis je poněkud obecnější než použití !cin.fail() nebo !cin.eof(), protože detekuje další možné příčiny neúspěšného čtení, jako je například selhání disku.

Protože metoda cin.get(char) vrací objekt cin, můžeme nakonec podmínku cyklu stěsnat do tohoto formátu:

```
while (cin.get(ch))                 // když je vstup úspěšný
{
... // zpracování příkazů
}
```

Aby mohl program vyhodnotit test cyklu, musí nejprve vykonat volání cin.get(ch), které, pokud je úspěšné, umístí hodnotu do proměnné ch. Program tak získá návratovou hodnotu volání funkce typu cin. Potom konvertuje cin na bool, čímž získá hodnotu true, jestliže vstup proběhl v pořádku, jinak false. Tři zásady navrhování cyklu (určení ukončovací podmínky, inicializace podmínky a aktualizace podmínky) jsou všechny vtěsnány do jedné testovací podmínky.

Toto je pouze náhled elektronické knihy. Zakoupení její plné verze je možné v elektronickém obchodě společnosti eReading.