2. aktualizované vydání bestselleru

Jesse Liberty

Naučte se C++ za 21 dní

Zvládněte principy a techniky programování v C++ za pouhých 21 dní

Uplatněte získané dovednosti při programování reálných aplikací

CD obsahuje:
prostředí pro vývoj aplikací v C++ a lokalizované příklady z knihy

computer press
CPRESS
SAMS

Jesse Liberty

Naučte se C++ za 21 dní

2. aktualizované vydání

Computer Press, a.s.
Brno
2007

Naučte se C++ za 21 dní

2. aktualizované vydání

Jesse Liberty

Vydalo nakladatelství Computer Press, a.s. jako svou 2568. publikaci.

Vydavatelství a nakladatelství Computer Press, a.s.,
Holandská 8, 639 00 Brno, knihy.cpress.cz

ISBN 978-80-251-1583-1

Prodejní kód: K1455

Překlad: Josef Pojsl, Karel Voráček
Odborná korektura: Karel Voráček
Jazyková korektura: Petra Láníčková
Vnitřní úprava: Petr Klíma
Sazba: Vladimír Ludva
Rejstřík: Ctibor Foltýn

Obálka: Martin Sodomka
Komentář na zadní straně obálky: Václav Kadlec
Technická spolupráce: Jiří Matoušek, Iva Vilímská
Odpovědný redaktor: Václav Kadlec
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Computer Press, a.s., Holandská 8, 639 00 Brno
tel.: 546 122 111, fax: 546 122 112

Objednávejte na: **knihy.cpress.cz**
distribuce@cpress.cz

Bezplatná telefonní linka: **800 555 513**

Dotazy k vydavatelské činnosti směřujte na: **knihy@cpress.cz**

Máte-li zájem o pravidelné zasílání informací o knižních novinkách do Vaší e-mailové schránky, zašlete nám zprávu, obsahující váš souhlas se zasíláním knižních novinek, na adresu **novinky@cpress.cz.**

Stručný obsah

Obsah

O autorovi

Jesse Liberty je autorem tuctu knih věnovaných C++, C# a objektově orientované analýze a návrhu. Je prezidentem společnosti Liberty Associates, Inc. (`http://www.LibertyAssociates.com`), která nabízí služby týkající se vývoje pro platformu .NET, programování na zakázku, konzultační a poradenské činnosti, vzdělávání a školení.

Jesse patřil mezi významné softwarové inženýry společnosti AT&T, pracoval jako architekt pro software u firem Xerox a LinkNet (PBS) a byl viceprezidentem divize vývoje u Citibank. Žije se ženou Stacey a dvěma dcerami Rachel a Robin na předměstí Cambridge v americkém státě Massachusetts. Informace o této knize najdete na webové stránce `http://www.libertyassociates.com`, kde stačí klepnout na „Books and Resources“.

Věnování

Tato kniha je věnována památce Davida Levinea.

Poděkování

Využívám čtvrtého vydání této knihy, abych znovu poděkoval všem, bez jejichž pomoci a podpory by tato kniha nemohla nikdy vzniknout. V první řadě to jsou Stacey, Rachel a Robin Libertyovy.

Musím také poděkovat redaktorům nakladatelství Sams za jejich vysoce profesionální přístup. Zvláště zde musím poděkovat a zmínit Carol Ackermanovou, George Nedeffa, Kima Cofera, Eriku Millenovou, Howarda Lee Harknesse, Benjamina Berga a Candice Hightowera.

Rád bych poděkoval těm, kteří mě naučili programovat: Skippu Gilbrechovi a Davidu McCunovi, a také těm, kteří mě naučili C++, mezi nimi Stephenu Zagieboylovi. Mé poděkování také patří všem těm čtenářům, kteří mě u dřívějších vydání této knihy upozornili na chyby a překlepy.

Nakonec bych rád poděkoval paní Kalishové, která mě v roce 1965 v šesté třídě naučila binární aritmetiku, aniž bychom já nebo ona věděli proč.

Napište nám, co si myslíte

Jako čtenáři této knihy jste našimi nejdůležitějšími kritiky a komentátory. Velmi si vážíme vašich názorů a rádi bychom věděli, co děláme dobře, co by mohlo být lepší a o jakých oblastech byste rádi našli další publikace. Rádi proto uvítáme každou radu či připomínku, kterou nám pošlete.

Vaše připomínky jsou vítány. Můžete mi faxovat i psát elektronickou nebo běžnou poštou. Rád se dozvím, co se vám na této knize líbilo či nelíbilo a také co můžeme udělat, aby byla naše kniha ještě lepší.

Rád bych vás upozornil, že vám nemůžu pomoci s technickými obtížemi, které souvisí s tématy této knihy. Kvůli velkému množství pošty, kterou dostávám, se mi možná nepodaří odpovědět všem. Na webových stránkách `http://www.LibertyAssociates.com` *však k této knize najdete veškerou podporu.*

Jestliže se nám rozhodnete napsat, nezapomeňte prosím připsat název této knihy a jméno autora spolu s vaším jménem a telefonním či faxovým číslem. Všem komentářům bude věnována patřičná pozornost a podělím se o ně s autorem a redaktory, kteří na této knize pracovali.

Fax: 317-581-4770

E-mail: `Feedback@samspublishing.com`

Adresa: Michael Stephens
Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Poznámka redakce českého vydání: I nakladatelství 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
redakce počítačové literatury
Holandská 8
639 00 Brno

nebo

knihy@cpress.cz.

Další informace a případné opravy českého vydání knihy najdete na internetové adrese *http://knihy.cpress.cz/k1455*. Prostřednictvím uvedené adresy můžete též naší redakci zaslat komentář nebo dotaz týkající se knihy. Na vaše reakce se srdečně těšíme.

Úvod

S pomocí této knihy se můžete sami naučit programovat v jazyce C++. I když se nikdo nemůže opravdovému programovacímu jazyku naučit za pouhé tři týdny, seznámí vás tato kniha se základními principy jazyka C++. Každá z lekcí je sestavena tak, abyste ji byli schopni zvládnout za jediný den.

V pouhých 21 kapitolách se naučíte základům, jakými jsou smyčky, pole, šablony, řízení vstupu a výstupu, objektově orientované programování a tvorba aplikací C++. Vše je podáno přístupnou formou v přehledně sestavených lekcích. Pro ilustraci probíraného tématu najdete v každé lekci vzorové výpisy programů, doplněné o jejich výstupy a rozbory kódu.

Abyste se stali skutečně zběhlí v probírané látce, uzavírá se každá lekce sadou otázek a odpovědí, testem a souborem cvičení. V dodatku D „Odpovědi" si můžete zkontrolovat výsledky testů a cvičení a zjistit, jakého pokroku jste dosáhli.

Komu je tato kniha určena

Jazyku C++ se s touto knihou naučíte i bez předchozích zkušeností s programováním. Začíná totiž od základů a seznámí vás s jazykem i s koncepty, které se při programování v C++ používají. Naleznete zde četné příklady syntaxe a podrobné rozbory kódu vám budou na začátku cesty ke zvládnutí jazyka C++ vynikajícím průvodcem. Ať jste úplní začátečníci nebo už máte s programováním jisté zkušenosti, přehledná organizace této knihy vám studium jazyka C++ usnadní a urychlí.

Konvence

POZNÁMKA V těchto rámečcích najdete zvýrazněné informace, které mohou přispět k vyšší efektivitě programování v jazyce C++.

Často kladené otázky

K čemu často kladené otázky slouží?

Odpověď: Často kladené otázky nabízejí hlubší pohled do struktury jazyka a vyjasní možná nedorozumění.

UPOZORNĚNÍ V takových rámečcích vás upozorníme na problémy nebo vedlejší účinky, ke kterým může dojít ve specifických případech.

V tomto rámečku najdete srozumitelné definice základních pojmů.

TIP Touto ikonou jsou v knize označovány tipy, které obsahují užitečné rady související s projednávaným tématem.

ANO	NE
Zde si **najděte** stručná shrnutí základních pravidel každé lekce.	**Nepřehlédněte** informace v těchto rámečcích, mohou se vám hodit.

V této knize se používají různé typy písma, což vám pomůže v rozlišování kódu C++ od běžného textu. Vlastní kód C++ je vysázen typem písma s konstantní šířkou. Slova nebo znaky, které se používají dočasně pro reprezentaci skutečných slov nebo znaků v kódu, jsou vysázeny písmem s konstantní šířkou a kurzivou. Nové nebo důležité pojmy jsou vysázeny *kurzivou*.

Řádky u výpisu každého skutečného kódu jsou v této knize očíslovány. Jestliže se ve výpisu setkáte s neočíslovaným řádkem, bude se jednat o pokračování předcházejícího očíslovaného řádku kódu (některé řádky kódu jsou totiž na šířku stránky této knihy příliš dlouhé). V takovém případě byste měli oba dva řádky zapsat v programu jako jeden a nerozdělovat je.

První týden

Na první pohled

Než se pustíte do prvního týdne studia programování v C++, bude třeba si obstarat několik věcí: překladač, editor a tuto knihu. I bez překladače (kompilátoru) C++ a editoru bude stále možné tuto knihu používat, ale nevytěžíte z ní tolik, jako kdybyste prováděli všechna cvičení.

Programování se nejlépe naučíte tak, že budete programovat. Každý den je zakončen částí, která obsahuje test a několik cvičení. Udělejte si čas a všechny otázky v klidu zodpovězte; potom se pokuste své výsledky co nejobjektivněji zhodnotit. Kapitoly na sebe navazují, proto nepokračujte s další kapitolou, dokud si nejste jisti, že látku stávající kapitoly zvládáte.

Poznámka pro programátory v jazyce C

Obsah látky v prvních pěti dnech vám bude patrně známý. Určitě si však i tuto část alespoň zběžně projděte a proveďte cvičení, abyste byli správně připraveni na den 6., který má název „Objektově orientované programování".

Další postup

První týden se budeme zabývat obecnou tematikou, která je nezbytná, abyste mohli začít s programováním obecně a zejména v jazyce C++. První den – „Začínáme" – a druhý den – „Anatomie programu C++" – se seznámíte se základními pojmy z programování a toku programu. Třetího dne se v kapitole s názvem „Proměnné a konstanty" dozvíte něco o proměnných a konstantách i o tom, jak se v programech pracuje s daty. V kapitole čtvrté s názvem „Výrazy a příkazy" uvidíte, jak se programy větví podle dat a podmínek, na které za běhu narazí. Pátého dne – „Funkce"– vás seznámíme s tím, co to funkce jsou a jak se používají. V kapitole šesté se dozvíte něco více o třídách a objektech. Den sedmý – „Více o toku programu" – je věnovaný dalšímu rozšíření tematiky toku programu. Na konci prvního týdne budete psát skutečné objektově orientované programy.

První týden

den 1

Začínáme

Úvod

Vítejte při čtení knihy *Naučte se C++ za 21 dní.* Dnes se pouštíte do samostatného studia, díky kterému se stanete zdatnými programátory v jazyce C++.

Dnes se dozvíte a naučíte:

- Proč je C++ nově vznikající standard v oblasti vývoje softwaru.
- Postup, jak vyvinout program v C++.
- Způsob, jak zapsat, zkompilovat a sestavit svůj první fungující program v C++.

Stručná historie jazyka C++

Od dob prvních elektronických počítačů určených k výpočtům dráhy dělové koule během druhé světové války prodělaly počítačové jazyky dramatický vývoj. Zpočátku pracovali programátoři s nejprimitivnějšími instrukcemi počítače, s tak zvaným strojovým jazykem. Tyto instrukce měly tvar dlouhých řetězců jedniček a nul. Brzy vznikly as-

semblery, které převáděly pokyny pro počítač do podoby dobře čitelných a ovladatelných mnemotechnických pomůcek, jako například ADD a MOV.

Časem byly vyvinuty jazyky vyšší úrovně, jako BASIC a COBOL. Tyto jazyky umožnily programátorům pracovat s kódem, který se podobal (většinou anglickým) slovům a větám, například Let I = 100. Tyto věty byly pak prostřednictvím interpreterů a kompilátorů přeloženy opět do strojového jazyka.

Interpreter program přeloží při čtení a mění instrukce programového kódu přímo v konkrétní akce. Kompilátor kód přeloží do podoby tak zvaného objektového souboru, který ještě není spustitelný; tento krok se nazývá kompilace. Pak kompilátor spustí sestavovací program (linker), který objektové soubory sestaví do spustitelného programu.

Vzhledem k tomu, že interpreter kód čte a ihned jej provádí, je pro programátory snadné s interpretery pracovat. Dnes se většina interpretovaných programů označuje jako skripty a interpreter samotný se pak označuje jako skriptové jádro (Script Engine).

V některých jazycích, například ve Visual Basicu, se interpreter nazývá běhová (runtimová) knihovna. V Javě se pak označuje termínem virtuální stroj (Virtual Machine, VM), ale v tomto případě je virtuální stroj součástí prohlížeče (například Internet Exploreru nebo Netscapu).

Kompilátory s sebou přinášejí krok navíc, a tím je kompilace zdrojového kódu (který je lidem srozumitelný) do objektového kódu (který je srozumitelný stroji). Tento krok navíc je nepohodlný, ale zkompilované programy běží velmi rychle, protože časově náročný úkol přeložení zdrojového kódu do strojového jazyka se provede jednou (v době kompilace) a při provádění programu se už nevyžaduje.

K dalším výhodám kompilovaných jazyků, jako například C++, patří, že je možné šířit spustitelný program i mezi uživatele, kteří žádný kompilátor nemají. Interpretované jazyky ke spuštění programu vyžadují interpreter.

C++ je většinou kompilovaný jazyk, ovšem existují také interpretery C++. Jako řada jiných kompilovaných jazyků má i C++ pověst generování rychlých, ale přesto velmi silných programů.

Dlouhá léta bylo nejdůležitějším úkolem programátora psát krátké úseky kódu, které se provedou co možná nejrychleji. Program musel být malý, protože paměť byla drahá, a zároveň musel být rychlý, protože strojový čas byl také drahý. Jak se počítače zmenšovaly, zlevňovaly a zrychlovaly a náklady na paměť se snižovaly, došlo v těchto prioritách k posunu. V dnešní době cena času programátora zdaleka převyšuje cenu většiny počítačů, které se používají v komerční sféře. Kód, který je dobře napsaný a snadno se udržuje, je ceněn nejvíce. Pod snadnou údržbou kódu rozumíme to, že program lze v případě změny požadavků snadno a bez velkých nákladů rozšířit a vylepšit.

POZNÁMKA Slovo program se používá ve dvou významech: jako označení jednotlivých instrukcí vytvořených programátorem (neboli zdrojového kódu) a jako označení celého úseku spustitelného kusu softwaru. Tento rozdíl může vést k velkým nedorozuměním, proto se budeme snažit rozlišovat mezi zdrojovým kódem na jedné straně a spustitelným programem na straně druhé.

Řešení problémů

Problémy, které dnes před programátory stojí, jsou zcela odlišné od problémů, které měli za úkol řešit před dvaceti lety. V osmdesátých letech 20. století se programy psaly proto, aby

zpracovávaly ohromná množství neutříděných dat. Lidé, kteří programy psali, i ti, kteří je používali, byli všichni odborníci. Dnes počítače používá mnohem více lidí a většina z nich toho ví jen velmi málo o tom, jak programy a počítače fungují. Lidem dnes počítače slouží jako nástroj, který jim má pomoci při řešení jejich pracovních problémů, a nemají zájem s nimi zápasit.

Ironií je, že jak se programy stále více přizpůsobují tomuto novému publiku, stávají se samy mnohem propracovanější a složitější. Pryč jsou časy, kdy na těžko srozumitelnou výzvu uživatel odpověděl magickým příkazem a výsledkem byl pouze proud nečitelných dat. Dnešní programátoři používají „uživatelsky příjemná rozhraní“, jako jsou okna, nabídky, dialogy a nesčetné množství metafor, na které jsme si všichni zvykli.

S rozvojem webu vstoupily počítače do nové éry, ve které stále více pronikají na trh. Používá je stále více lidí a jejich očekávání jsou velmi vysoká. Za několik let od prvního vydání této knihy se programy staly rozsáhlejší a komplexnější a vyvstala potřeba programovacích technik, které by tuto složitost pomohly zvládnout.

V několika posledních letech se aplikace rozšířily také na jiná zařízení. Dnes již není jejich jediným vážným cílem stolní počítač. Stále významnějšími cíli moderních aplikací jsou mobilní telefony, osobní digitální asistenti (PDA), stroje Tablet PC a další zařízení.

Během několika let od prvního vydání této knihy zareagovali programátoři na požadavky uživatelů a jejich programy byly stále větší a komplexnější. Jasně se následně projevila potřeba programovacích technik pomáhajících tuto složitost zvládnout.

Se změnou nároků na programování se změnily a rozvinuly i jazyky a techniky, které se při psaní programů používají. Ačkoli je celá historie této proměny nesmírně zajímavá, v této knize se zaměříme pouze na přechod od procedurálního programování k objektově orientovanému programování.

Procedurální, strukturované a objektově orientované programování

Až donedávna se za program považoval sled procedur, které zpracovávaly data. Procedurou nebo funkcí rozumíme sadu přesně stanovených instrukcí, které se mají provést jedna po druhé. Data byla od procedur zcela oddělena a kouzlo programování spočívalo v udržování přehledu o tom, které funkce volaly které jiné funkce a u kterých dat došlo ke změně. Snaha o zvládnutí této potenciálně nepřehledné situace vedla ke vzniku strukturovaného programování.

Základní myšlenka, která se za strukturovaným programováním skrývá, se dá přirovnat ke známému rčení „rozděl a panuj“. Na počítačový program se můžeme podívat jako na sadu úkolů. Každý úkol, který je příliš náročný na to, aby se dal jednoduše popsat, budeme dělit na menší části, a to tak dlouho, dokud nebudou úkoly natolik malé a samostatné, aby se daly snadno pojmout.

Jako příklad nám poslouží výpočet průměrného platu všech zaměstnanců firmy, což je poměrně složitý úkol. Lze jej však rozložit na následující podúkoly:

1. Zjištění, kolik každá osoba vydělává.
2. Určení počtu zaměstnanců.
3. Sečtení všech platů.
4. Vydělení výsledku součtu počtem zaměstnanců.

Stanovení součtu všech platů můžeme rozložit na následující kroky:

1. Získání záznamu o každém zaměstnanci.
2. Získání platu.
3. Přičtení platu k průběžnému součtu.
4. Získání záznamu dalšího zaměstnance.

Získání záznamu každého zaměstnance lze dále rozložit takto:

1. Otevření souboru zaměstnanců.
2. Přechod na správný záznam.
3. Přečtení dat z disku.

Strukturované programování zůstává nesmírně účinným přístupem při řešení složitých problémů. Koncem osmdesátých let se však začaly stále více projevovat jeho nedostatky.

Zaprvé, je přirozené o datech (například záznamech o zaměstnanci) a o tom, co je možné s nimi provádět (třídění, editace atd.), uvažovat uceleně. Strukturované programy však bohužel datové struktury oddělují od funkcí, které s nimi pracují, a neexistuje zde žádný přirozený způsob, jak data s funkcemi propojit. Strukturované programování se často označuje termínem procedurální programování, protože se soustřeďuje na procedury (spíše než na „objekty“).

Zadruhé, programátoři často potřebovali opakovaně používat určité funkce. Ovšem funkce fungující s jedním typem dat často nebylo možné aplikovat ve spojení s jiným typem dat, takže dosažené přínosy byly minimální.

Objektově orientované programování (OOP)

Objektově orientované programování se pokouší vyjít těmto požadavkům vstříc a nabízí techniky, díky kterým lze zvládnout i velmi složité problémy. Účelem je dosáhnout opakované použitelnosti softwarových komponent a propojit data s úkoly, jež data zpracovávají.

Podstata objektově orientovaného programování spočívá ve vymodelování „objektů“ (tedy reálných věcí) spíše než „dat“. Objekty, které modelujete, mohou být drobné prvky na obrazovce počítače jako tlačítka či okna se seznamem nabízených hodnot, nebo jimi mohou být reálné předměty jako kola, letadla, kočky či voda.

Objekty mají charakteristiky (rychlá, prostorná, černá, mokrá) a schopnosti (zrychlit, letět, příst, bublat). Úkolem objektově orientovaného programování je tyto objekty vytvořit v programovacím jazyce.

C++ a objektově orientované programování

C++ plně podporuje objektově orientované programování včetně jeho tří pilířů: zapouzdření, dědičnost a mnohotvarost (polymorfismus).

Zapouzdření

Když technik potřebuje rezistor do zařízení, na kterém pracuje, zpravidla si jej nesestavuje sám. Zajde si k přihrádce s rezistory a podle barevného proužku označujícího jeho vlastnosti si vybere ten, který odpovídá jeho aktuální potřebě. Pokud se technika týče, je pro něj re-

zistor „černou skříňkou“, a pokud vyhovuje jeho potřebám, příliš jej nezajímá, jak funguje. Nemusí se dívat dovnitř, aby jej mohl použít pro svou práci.

Tuto vlastnost, tedy existenci samostatné fungující jednotky, nazýváme zapouzdření. S pomocí zapouzdření je možné dosáhnout tak zvaného skrývání dat. Jedná se o vysoce ceněnou charakteristiku, kdy lze objekt použít bez toho, aby uživatel věděl nebo se staral o to, jak uvnitř funguje. Stejně jako můžete používat ledničku, aniž byste věděli, jak funguje její kompresor, můžete používat dobře navržený objekt, aniž byste cokoli věděli o jeho vnitřních datových členech.

Podobně nemusí technik nic vědět o vnitřním stavu rezistoru, který používá. Všechny vlastnosti rezistoru jsou zapouzdřeny do objektu rezistoru a nešíří se za hranice jeho pouzdra. Není třeba rozumět tomu, jak rezistor funguje, abychom jej mohli účinně používat. Jeho vlastnosti jsou ukryty v pouzdře rezistoru.

V C++ je zapouzdření podporováno prostřednictvím uživatelsky definovaných typů, kterým se říká třídy. V kapitole šesté „Objektově orientované programování“ se seznámíte s tím, jak se dají třídy vytvořit. Jakmile je jednou třída vytvořena, může fungovat jako plně zapouzdřená entita, která se používá jako jeden celek. Vlastní vnitřní fungování třídy by mělo zůstat skryto. Uživatelé dobře navržené třídy nemusí vědět, jak funguje, stačí jim vědět, jak mají třídu používat.

Dědičnost a opakované používání

Když budou chtít inženýři ve Škodě Mladá Boleslav postavit nové auto, mají dvě možnosti. Mohou začít úplně od začátku nebo mohou upravit stávající model. Řekněme, že jejich stávající model Star je téměř dokonalý, ale chtěli by, aby měl turbokompresor a převodovku se šesti rychlostmi. Hlavnímu inženýrovi se nechce začínat zase od začátku, a proto raději prohlásí: „Postavme další Star a přidejme k němu tyto dodatečné vlastnosti. Můžeme mu říkat Quasar.“ Bude se v podstatě jednat o model Star, ale specializovaný a s jednou vlastností navíc. (Podle NASA jsou kvasary extrémně svítící tělesa, která vydávají neuvěřitelné množství energie.)

V C++ je dědičnost podporována. Je možné deklarovat nový typ, který je rozšířením existujícího typu. Lze říci, že nová podtřída je odvozena od stávajícího typu, a proto se někdy označuje jako odvozený typ. Model Quasar je odvozený od modelu Star, a proto dědí všechny jeho vlastnosti, ale může k nim přidávat další nebo je podle potřeby modifikovat. Dědičnost a její aplikace v C++ se probírají ve dni 12. – „Dědičnost“ – a dni 16. – „Pokročilá dědičnost“.

Mnohotvarost

Když šlápnete na pedál plynu nového modelu Quasar, může se jeho reakce lišit od reakce modelu Star. Quasar by uvedl do chodu palivovou trysku a turbokompresor, zatímco Star by jednoduše umožnil přívod paliva do karburátoru. Ten, kdo auto řídí, však o těchto rozdílech nic neví. Stačí, když na to pořádně šlápne, a podle toho, jaké auto řídí, bude provedena ta správná akce.

C++ podporuje myšlenku, že různé objekty dělají vždy tu „správnou věc“, prostřednictvím tak zvaného polymorfismu (mnohotvarosti) funkcí a tříd. „Poly“ znamená mnoho a „morf“ je zkráceně forma. Polymorfismem rozumíme, že stejné jméno může mít mnoho forem. Podrobně se s mnohotvarostí seznámíte ve dnu 10. – „Pokročilé funkce“ – a dnu 14. – „Mnohotvarost“.

Vývoj jazyka C++

Když se objektově orientovaná analýza, návrh a programování stávaly stále více populární, vzal Bjarne Stroustrup nejoblíbenější jazyk pro vývoj komerčního softwaru, C, a rozšířil jej o nezbytné vlastnosti umožňující objektově orientované programování.

Ačkoli je pravda, že jazyk C++ je nadmnožinou jazyka C a že každý platný program v C je platným programem i v C++, je přechod od C k C++ velmi významný. Po dlouhá léta profitoval jazyk C++ ze své příbuznosti s jazykem C, protože programátorům v C mohla jeho znalost usnadnit přechod k používání jazyka C++. Když však chtěli z jazyka C++ vytěžit maximum, zjistili, že se musí odnaučit mnohému, co znali, a naučit se novému způsobu koncepčního myšlení a řešení programátorských problémů.

Není lepší naučit se nejdříve C?

Nevyhnutelně vyvstává otázka: „Jestliže je jazyk C++ nadmnožinou jazyka C, neměl bych se nejdříve naučit C?“ Stroustrup a mnozí další se shodují v tom, že to nezbytné není, ale že navíc může být dokonce výhodou, když tak neučiníte.

Programování v C vychází z pojetí strukturovaného programování, zatímco C++ je založeno na objektově orientovaném programování. Je tedy chybou se nejdříve naučit C, protože se pak budete muset odnaučit špatné návyky odvozené z jazyka C.

Tato kniha nepředpokládá, že čtenář má předchozí zkušenosti s programováním. Pokud však už programujete v C, prvních několik kapitol bude pro vás spíše opakováním známé látky. Šestým dne začíná práce na skutečném objektově orientovaném vývoji softwaru.

C++, Java a C#

Vývoji komerčního softwaru dnes dominuje jazyk C++. Před několika lety se jazyk Java pokusil tuto dominanci zpochybnit, ale kyvadlo se vrací zpět a mnozí z programátorů, kteří kvůli Javě opustili C++, se k němu v poslední době začali vracet. V každém případě se jedná o tak podobné jazyky, že když se naučíte jednomu, budete z devadesáti procent znát i ten druhý.

Microsoft pro svou platformu .Net vyvinul i nový jazyk C#. Ten používá v zásadě stejnou syntaxi jako C++. Přestože se oba jazyky v některých důležitých vlastnostech od sebe odlišují, znalost jazyka C++ vám zaručí, že zároveň budete znát i devadesát procent z jazyka C#. Pokud se později rozhodnete naučit jazyk C#, úsilí, které vložíte do studia C++, bude vynikající investicí.

Standard ANSI

Komise Accredited Standards Committee, která pracuje pod hlavičkou amerického ústavu pro standardizaci American National Standards Institue (ANSI), vytvořila pro jazyk C++ mezinárodní standard.

Na tento standard C++ se nyní odkazuje i jako na standard ISO (International Standards Organization), NCITS (National Committee for Information Technology Standards), X3 (starý název pro NCITS) a ANSI/ISO. V této knize budeme používat označení „standard ANSI“, protože se jedná o běžněji používaný termín.

POZNÁMKA Zkratku ANSI obvykle vyslovujeme s němým „t“ jako „antsy“.

Standard ANSI považujeme za pokus o zaručení toho, že kód C++ bude přenositelný. To například znamená, že když napíšete kód pro kompilátor od Microsoftu, který je v souladu se standardem ANSI, tak se bezchybně zkompiluje, i když použijete kompilátor od jakéhokoli jiného výrobce. Vzhledem k tomu, že kód v této knize je se standardem ANSI ve shodě, měl by se bez problémů zkompilovat na počítačích Mac, v systémech PC Windows i Alpha.

Pro většinu studentů C++ zůstane standard ANSI „neviditelnou“ samozřejmostí. Poslední verzí tohoto standardu je ISO/IEC 14882-2003. Předchozí standard ISO/IEC 14882-1998 byl stabilní a všichni významní výrobci jej podporují. Vynaložili jsme všechno úsilí, abychom zaručili, že veškerý kód v tomto vydání knihy je se standardem ANSI ve shodě.

Příprava na psaní programu

Pro jazyk C++ platí snad více než pro ostatní programovací jazyky, že programátor by měl před samotným napsáním programu vypracovat jeho návrh. U triviálních problémů, které uvádíme jako příklady v prvních několika kapitolách této knihy, většinou programátor mnoho času návrhem nestráví. Složité problémy, se kterými jsou profesionální programátoři ve své každodenní praxi konfrontováni, však už návrh vyžadují. Bude-li návrh dostatečně důkladný, je velmi pravděpodobné, že problém bude úspěšně vyřešen včas a s daným rozpočtem. Dobrý návrh se může významně podepsat na tom, zda bude program více méně bez chyb a zda se bude dobře udržovat. Odhaduje se, že plných devadesát procent ceny softwaru připadne na odstraňování chyb a jeho údržbu. Z tohoto hlediska pak může mít dobrý návrh programu opravdu zásadní vliv na konečnou cenu celého projektu.

První otázka, kterou si při přípravě návrhu každého programu musíte položit, zní následovně: „Jaký problém se pokouším vyřešit?“ U každého programu by měl být jasný, srozumitelně formulovaný cíl a brzy poznáte, že i ty nejjednodušší programy v této knize je mají.

Druhá otázka, kterou si každý dobrý programátor položí, zní: „Není možné úkol provést, aniž bych musel psát úplně nový software na zakázku?“ Často se jako lepší řešení ukáže použití starého programu, tužky a papíru, nebo zakoupení už hotového softwaru v obchodě. Programátor, který navrhne tyto alternativy, se nemusí obávat, že by trpěl nedostatkem práce. Budou-li nalezena méně nákladná řešení dnešních problémů, může to v budoucnosti přinést nové příležitosti.

Dojdete-li k závěru, že problému rozumíte a že vyžaduje nový program, jste připraveni začít s jeho návrhem.

Analýza problému, tedy jeho úplné pochopení, a tvorba návrhu, tedy nalezení řešení, jsou nezbytnými předpoklady úspěšně napsané komerční aplikace.

Vývojové prostředí

V této knize vycházíme z předpokladu, že váš kompilátor má režim, ve kterém můžete psát přímo na obrazovku, aniž byste se museli starat o grafické prostředí, jakým jsou systémy Windows nebo Macintosh. Hledejte v kompilátoru volby jako *console* nebo *easy window* anebo nahlédněte do dokumentace ke kompilátoru.

Váš kompilátor může mít vlastní vestavěný textový editor nebo můžete používat nějaký komerční textový editor nebo procesor, který umí vytvářet textové soubory. Ať už píšete programy v čemkoli, důležité je, aby se programy ukládaly do prostých textových souborů bez jakýchkoli příkazů ke zpracování či formátování textu. Jako příklad bezpečných editorů mohou sloužit editory: Windows Notepad, příkaz Dosu Edit, Brief, Epsilon, EMACS a vi. Také mnohé komerční textové procesory jako například WordPerfect, Word a tucet dalších umožňují ukládat prosté textové soubory.

Soubory vytvořené v editoru se nazývají zdrojové soubory a pro jazyk C++ je typické, že se k jejich názvům přidává přípona `.cpp`, `.cp` nebo `.c`. V této knize budeme k názvům všech zdrojových souborů přidávat příponu `.cpp`; pro jistotu si ověřte, jaké požadavky na názvy souborů má váš kompilátor.

POZNÁMKA U většiny kompilátorů C++ nezáleží na tom, jakou příponu zdrojovému souboru udělíte, ale jestliže neurčíte jinak, bude většina z nich standardně používat `.cpp`. Dejte si však pozor, protože některé kompilátory zachází se soubory s příponou `.c` jako s kódem v C a se soubory s příponou `.cpp` jako s kódem v C++. V každém případě si to ověřte v dokumentaci.

Ano	Ne
Ano, k vytvoření zdrojového kódu používejte jednoduchý textový editor nebo zabudovaný editor, který je součástí vašeho kompilátoru.	**Ne**, nepoužívejte textový procesor, který ukládá speciální formátovací znaky. Jestliže textový procesor používáte, ukládejte soubory jako text ASCII.
Ano, ukládejte své soubory s příponami `.c`, `.cp` a `.cpp`. Doporučujeme používat příponu `.cpp`.	**Ne**, nepoužívejte příponu `.c`, pokud váš kompilátor pracuje s takovým kódem jako s jazykem C, a nikoli C++.
Ano, nahlédněte do dokumentace a ověřte si, zda váš kompilátor nebo linker nevyžaduje nějaká specifika. Budete pak s jistotou vědět, jak programy kompilovat a sestavovat.	

Vytvoření programu

Prvním krokem tvorby nového programu je zápis příslušných příkazů do zdrojového souboru. Ačkoli se vám může zdrojový kód v souboru zdát poněkud tajemný, a kdo o jazyku C++ nic neví, bude mu jen stěží rozumět, stále je možné říci, že je zapsán v lidsky čitelné podobě. Soubor se zdrojovým kódem není program a nelze jej provést ani spustit, jako je to možné udělat s programem.

Vytvoření objektového souboru kompilátorem

Kompilátor slouží k převedení zdrojového souboru na program. Způsob spuštění kompilátoru (překladače) a předání informace o tom, kde se zdrojový soubor nalézá, se u různých kompilátorů liší. Nahlédněte proto do dokumentace.

Výsledkem kompilace zdrojového programu je objektový soubor. K názvu tohoto souboru se obvykle přidává přípona `.obj` nebo `.o`. Stále se však nejedná o spustitelný program. Ten získáte až po spuštění sestavovacího programu (linkeru).

Vytvoření spustitelného programu linkerem

Pro programy v C++ je typické, že se vytváří propojením jednoho nebo více objektových souborů `.obj` s jednou nebo více knihovnami. Knihovnou rozumíme sbírku propojitelných souborů, které jsou buď součástí vašeho kompilátoru, nebo jež jste zakoupili zvlášť nebo jste je sami vytvořili a zkompilovali. Každý kompilátor C++ je opatřen knihovnou praktických funkcí (nebo procedur) a tříd, které se mohou stát součástí vašeho programu. O funkcích a třídách budeme mluvit podrobněji v příštích kapitolách.

Postup při vytvoření spustitelného programu je následující:

1. Vytvořte zdrojový soubor s příponou `.cpp`.
2. Tento zdrojový soubor zkompilujte do souboru s příponou `.obj`.
3. Propojte soubor `.obj` se všemi potřebnými knihovnami, čímž vytvoříte spustitelný program.

Vývojový cyklus

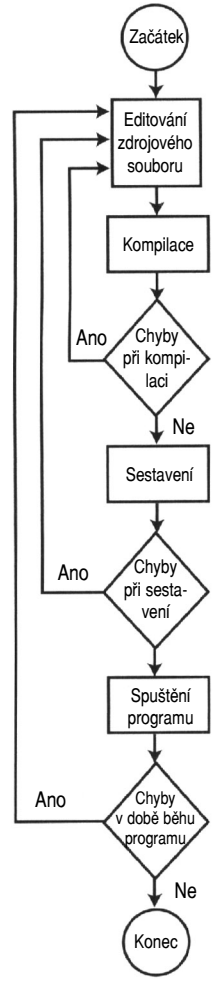
Kdyby každý program fungoval hned napoprvé, kdy jej vyzkoušíte, byl by to úplný vývojový cyklus: program je napsán, zdrojový kód se zkompiluje, program se sestaví a spustí. Bohužel téměř každý program, jakkoli triviální, může a bude mít chyby. Některé z nich budou příčinou chyb při kompilaci, jiné způsobí chyby při sestavení a další se projeví, až když se program spustí.

Bez ohledu na typ nalezené chyby ji musíte opravit, což zahrnuje úpravu zdrojového kódu, opětovné zkompilování, sestavení a spuštění programu. Na obrázku 1.1 si můžete prohlédnout diagram, který popisuje kroky vývojového cyklu.

`NAZDAR.cpp` – první program v C++

V každé učebnici programování se začíná programem, který má na obrazovku napsat slova „Nazdar lidi“ (v anglických textech je to „Hello World“) nebo nějakou obměnu této věty. I my zde budeme v této časem prověřené tradici pokračovat.

Napište svůj první program do editoru přesně tak, jak ho vidíte ve výpisu. Když si budete jisti, že je vše v pořádku, soubor uložte, zkompilujte, sestavte a spusťte. Na vaší obrazovce se objeví slova Nazdar lidi. Nedělejte si hlavu s tím, jak program funguje. Záměrem tohoto programu je, abyste si prakticky procvičili vývojový cyklus. K tomuto programu se ještě během příštích dnů vrátíme a rozebereme si jej ze všech stran.



Obrázek 1.1
Kroky ve vývojovém cyklu programu C++

UPOZORNĚNÍ Řádky v následujícím výpisu jsou zleva očíslovány. Tato čísla slouží pro odkazy v rámci knihy a neměli byste je psát do editoru. Například na řádku 0 výpisu 1.1 byste měli zadat:

```
#include <iostream>
```

Výpis 1.1

NAZDAR.cpp, program Nazdar lidi

```
0:  #include <iostream>
1:
2:  int main()
3:  {
4:       std::cout << "Nazdar lidi!\n";
5:       return 0;
6:  }
```

Dbejte na to, abyste vše zadali přesně tak, jak vidíte. Věnujte pozornost zvláště interpunkci. Na řádku 4 se jedná o symbol přesměrování <<, který u většiny klávesnic získáte podržením klávesy Shift a dvojím stisknutím klávesy s čárkou (na české klávesnici stiskem pravého Alt a čárky). Na řádku 4 jsou mezi slovy `std` a `cout` dvě dvojtečky (:). Řádky 4 a 5 jsou zakončeny středníkem (;).

Měli byste se také náležitě řídit pokyny kompilátoru. Většina kompilátorů sestavuje program automaticky, ale pro jistotu se podívejte do dokumentace. Pokud narazíte na chybu, pečlivě si prohlédněte kód a pokuste se zjistit, v čem se liší od našeho. Jestliže na řádku 1 uvidíte chybu, jako například `cannot find file iostream`, vyhledejte si v dokumentaci ke svému kompilátoru pokyny týkající se nastavení cesty k souborům pro operaci „include“ nebo nastavení proměnných prostředí. Jestliže obdržíte chybu oznamující, že pro `main` neexistuje žádný prototyp, přidejte před řádek 2 řádek obsahující `int main();`. V takovém případě pak budete muset ke každému programu v této knize přidat před začátek hlavní funkce tento řádek. Většina kompilátorů to nevyžaduje, ale existují výjimky.

Váš program by pak měl vypadat takto:

```
0:  #include <iostream>
1:  int main(); // Většina kompilátorů tento řádek nevyžaduje
2:  int main()
3:  {
4:      std::cout << "Nazdar lidi!\n";
5:      return 0;
6:  }
```

POZNÁMKA Může být obtížné si pro sebe číst program, jestliže nevíte, jak se zvláštní znaky nebo klíčová slova vyslovují. První řádek přečtěte „haš inklúd nebo křížek inklúd aj-ou-strím“. Řádek 4 můžete číst následovně: „es-tí-dí-sí-aut, nazdar lidi“.

Pokuste se spustit program `NAZDAR.exe`. Přímo na obrazovce byste měli dostat text:

```
Nazdar lidi!
```

(Toto zadání platí pro systém Windows. Obecně je zapotřebí spustit vykonatelný soubor, který ovšem na unixovém systému nemá uvedenou příponu a spouští se prostým zadáním NAZDAR.)

Je-li tomu tak, přijměte naši gratulaci. Úspěšně jste zapsali, zkompilovali a spustili svůj první program v C++. Možná se vám to nezdá, ale skoro každý profesionální programátor v C++ začínal s podobným programem.

Programátoři, kteří využívají prostředí IDE (jako je Visual Studio nebo Borland C++ Builder), zjistí, že jejich program jen krátce otevře nové okno, které ovšem zase ihned zmizí, aniž by měli šanci podívat se na výsledky programu. Pokud je to právě vaše situace, doplňte do svého kódu těsně před příkaz `return` následující řádky:

```
char reakce;
std::cin >> reakce;
```

Uvedené řádky způsobí pozastavení programu až do zápisu nějakého znaku (nebo kupříkladu stisku klávesy Enter). Budete tedy mít šanci podívat se na výsledek zkušebního spuštění programu. Pokud musíte uvedené řádky zadat v případě souboru `nazdar.cpp`, pak je zřejmě bude zapotřebí zadat také ve většině ostatních programů naší knihy.

Používání standardních knihoven

Jestliže je váš kompilátor příliš starý, nebude výše uvedený program fungovat. Nebude totiž možné nalézt nové standardní knihovny ANSI. V tom případě je třeba program změnit takto:

```
0:  #include <iostream.h>
1:
2:  int main()
3:  {
4:      cout << "Nazdar lidi!\n";
5:      return 0;
6:  }
```

Všimněte si, že jméno knihovny teď končí příponou `.h` (tečka-h) a že před `cout` na řádku 4 se nadále nepoužívá předpona `std::`. Jedná se o starý způsob zápisu pro hlavičkové soubory z dob před ANSI. Jestliže váš kompilátor pracuje s touto, a ne s původní verzí programu, jedná se o zastaralý kompilátor. Vystačíte s ním v prvních kapitolách, ale jakmile se dostaneme k šablonám a výjimkám, nebude váš kompilátor s naším kódem fungovat.

Úvod do práce s kompilátorem

Tato kniha *není závislá* na žádném konkrétním kompilátoru. To znamená, že programy v této knize uvedené by měly fungovat s *libovolným* kompilátorem pro C++, který je ve shodě se standardy ANSI, a na libovolné platformě (Windows, Mac, UNIX, Linux atd.).

I přes tuto nezávislost je zřejmé, že velká většina programátorů pracuje v prostředí Windows a velká většina profesionálních programátorů používá kompilátory Microsoftu. Není možné, abychom zde pro každý myslitelný kompilátor uvedli podrobný popis kompilace a sestavení, ale jako dobrý výchozí bod vám můžeme předvést, jak začít v prostředí Dev-C++ 5.0, který je umístěn na doprovodném CD, takže si následující návod můžete ihned vyzkoušet. Postup se bude v dalších kompilátorech v detailech lišit, ale princip zůstane vždy stejný.

Kompilátory se však mohou odlišovat, proto v každém případě nahlédněte do dokumentace.

Vytvoření projektu Nazdar lidi

Chcete-li vytvořit a otestovat program Nazdar lidi, řiďte se následujícími kroky:

1. Po instalaci spusťte prostředí Dev-C++. Prostředí Dev-C++ sice existuje i v českém jazyce, ale protože jiná prostředí lokalizována být nemusí, budeme v následujícím popisu uvádět anglické příkazy z nabídek.
2. Z hlavní nabídky vyberte File, New – Project.
3. Zvolte Console Application, zadejte název projektu, například Priklad1, a klepněte na OK.
4. Vyberte umístění projektu a potvrďte.
5. Otevře se soubor, do něho přesně přepište (nebo zkopírujte z doprovodného CD) zdrojový kód.
6. Vyberte Execute, Compile & Run nebo stiskněte klávesu F9.
7. Zadejte název spustitelného programu, který bude vytvořen.
8. Program bude přeložen a spuštěn. Pokud se okno se spuštěným programem ihned zavře a vy si ani nestihnete prohlédnout, co program udělal a vypsal, čtěte dál – tento problém ihned vyřešíme.

Často kladené otázky

Podařilo se mi program spustit, ale mihne se tak rychle, že ho sotva stačím přečíst. V čem je chyba?

Odpověď: Nahlédněte do dokumentace ke kompilátoru. Měli byste tam najít způsob, jak program přimět, aby se po provedení zastavil. U kompilátorů Microsoftu spočívá tento trik v použití klávesové zkratky Control+F5.

U každého kompilátoru můžete před příkaz návratu (tedy mezi řádky 4 a 5 ve výpisu 1.1) přidat následující řádky:

```
char reakce;
std::cin >> reakce;
```

Ty způsobí, že se program zastaví a bude čekat, dokud nezadáte hodnotu. Budete-li chtít program ukončit, zadáte číslo nebo znak (například 1) a pak stisknete Enter.

S významem `std::cin` a `std::cout` se blíže seznámíte v příštích dnech. Prozatím je používejte jako kouzelná zaříkávadla.

Chyby při kompilaci

K chybám v době kompilace může dojít z mnoha důvodů. Obvykle se jedná o překlep nebo nějakou jinou neúmyslnou chybu. Dobré kompilátory vám nejen oznámí, co jste udělali špatně, ale také vám v kódu ukáží místo, kde k chybě dochází. Ty nejlepší vám dokonce nabídnou způsob, jak ji opravit.

Můžete si to vyzkoušet tak, že do svého programu úmyslně zanesete chybu. Jestliže program NAZDAR.cpp běžel bez problémů, upravte jej a z řádku 6 odstraňte závorku. Váš program pak bude vypadat jako ve výpisu 1.2.

Výpis 1.2

Ukázka chyby při kompilaci

```
 0:  #include <iostream>
 1:
 2:  int main()
 3:  {
 4:       std::cout << "Nazdar lidi!\n";
 5:       return 0;
```

Znovu program zkompilujte a měli byste vidět chybu, která se podobá této:

```
Nazdar.cpp(7) : fatal error C1004: unexpected end of file found
```

Tato chyba vás informuje o souboru a čísle řádku, na kterém je problém. Také se snaží problém pojmenovat (i když je to v tomto případě poněkud tajemné). V tomto případě vám kompilátor oznamuje, že dosáhl konce souboru zdrojového kódu, aniž by nalezl uzavírací složenou závorku.

Někdy vám chyba pouze všeobecně ukáže na problematickou oblast. Kdyby kompilátor uměl dokonale identifikovat každý problém, opravoval by si kód sám.

Souhrn

Po přečtení této kapitoly byste měli mít představu o tom, jak se jazyk C++ vyvíjel a k řešení jakých problémů je určen. Měli byste být klidní, protože jazyk C++ je pro vás tou správnou volbou, neboť se jedná o jazyk příštího desetiletí. Jazyk C++ nabízí nástroje pro objektově orientované programování a výkon na úrovni systémového jazyka. To všechno mluví pro volbu C++ jako jazyka, ve kterém budete vyvíjet své projekty.

Dnes jste se naučili zapsat, zkompilovat, sestavit a spustit program v C++. Seznámili jste se s tím, co rozumíme pod pojmem vývojový cyklus. Dozvěděli jste se také něco málo o objektově orientovaném programování. V následujících třech týdnech se k těmto tématům znovu vrátíme.

Otázky a odpovědi

Otázka: **Jaký je rozdíl mezi textovým editorem a textovým procesorem?**

Odpověď: Textový editor vytváří soubory, které obsahují obyčejný text. Konkrétní textový procesor nevyžaduje obvykle žádné příkazy pro formátování nebo jiné zvláštní symboly. Textové soubory nemají žádné automatické zalomení slov, tučné písmo, kurzivu atd.

Otázka: **Jestliže má kompilátor vestavěný editor, musím jej používat?**

Odpověď: Téměř všechny kompilátory zkompilují kód, který je výsledkem editace v libovolném textovém editoru. Používání vestavěného textového editoru má obvykle tu výhodu, že je možné rychle se přesouvat mezi kroky vývojového cyklu, zvláště mezi editováním a kompilací. Důmyslnější kompilátory nabízí plně integrované vývojové prostředí, což programátorovi usnadní přístup k souborům s nápovědou a editování i kompilování kódu na stejném místě. Bude také možné řešit chyby při kompilování nebo sestavování bez nutnosti vývojové prostředí opouštět.

Otázka: **Mohu ignorovat varování kompilátoru?**

Odpověď: Kompilátory obecně vydávají varování a chyby. Pokud se objeví nějaké chyby, pak se program nesestaví. V případě varování bude všem kompilátor obvykle v překladu pokračovat a program vytvoří. Mnohé knihy se této otázce vyhýbají, ale my si zde budeme stát za svým a odpovíme: Ne! Zvykejte si hned od prvního dne zacházet s varováními při kompilaci jako s chybami. C++ používá kompilátor k upozorněním, že možná děláte něco, co nemáte v úmyslu. Dbejte těchto upozornění a udělejte vše, co se po vás žádá, abyste se jich zbavili.

Otázka: **Co je to doba kompilace?**

Odpověď: Dobou kompilace rozumíme dobu, kdy máte spuštěný kompilátor, v kontrastu s dobou sestavení (kdy máte spuštěný sestavovací program) nebo s dobou běhu (kdy máte spuštěný výsledný program). Jedná se o programátorskou zkratku, která slouží k identifikaci těchto tří období, ve kterých se mohou vynořit na povrch chyby.

Úkoly pro vás

Testovací otázky v této části vám pomohou upevnit znalosti, které jste v této kapitole získali. Cvičení slouží k tomu, aby vám nabídlo praktickou zkušenost s tím, co jste se naučili. Pokuste se na otázky v testu a cvičení odpovědět dříve, než se podíváte do klíče v dodatku D. Než přejdete k další kapitole, ujistěte se, že všem odpovědím rozumíte.

Test

1. Jaký je rozdíl mezi interpreterem a kompilátorem?
2. Jak s pomocí kompilátoru (překladače) zkompilujete zdrojový kód?
3. Co je úkolem sestavovacího programu (linkeru)?
4. Jaké jsou jednotlivé kroky běžného vývojového cyklu?

Cvičení

1. Podívejte se na následující program, a aniž byste jej spustili, pokuste se odhadnout, co dělá.

```
 1: #include <iostream>
 2: int main()
 3: {
 4:   int x = 5;
 5:   int y = 7;
 6:   std::cout << endl;
 7:   std::cout << x + y << " " << x * y;
 8:   std::cout << end;
 9:   return 0;
10: }
```

2. Zapište v editoru program ze cvičení 1 a pak jej zkompilujte a sestavte. Co dělá? Odhadli jste to správně?
3. Napište následující program a zkompilujte jej. Jakou chybu obdržíte?

```
1: include <iostream>
2: int main ()
3: {
4:     std::cout << "Nazdar lidi\n";
5:     return 0;
6: }
```

4. Ve cvičení 3 opravte v programu chybu a znovu jej zkompilujte a spusťte. Co dělá?

den 2

Anatomie programu C++

Programy C++ jsou složeny z objektů, funkcí, proměnných a dalších komponent. Tato kniha je z větší části věnována podrobným rozborům těchto komponent. Chcete-li však pochopit, jak do sebe jednotlivé komponenty programu zapadají, musíte se podívat na celý fungující program.

Dnes se naučíte a dozvíte:

- Z jakých částí se skládá program C++.
- Jak tyto části spolupracují.
- Co jsou to funkce a k čemu slouží.

Jednoduchý program

Dokonce i v tak jednoduchém programu, jakým je program `NAZDAR.cpp` z prvního dne „Začínáme“, můžeme najít mnoho zajímavých prvků. V následující části se na tento program podíváme znovu, a to mnohem podrobněji. Určitě vám přijde vhod výpis 2.1, kde je přetištěna původní verze programu `NAZDAR.cpp`.

Výpis 2.1

Na programu `NAZDAR.cpp` demonstrujeme části programu C++

```
0:  #include <iostream>
1:
2:  int main()
3:  {
4:      std::cout << "Nazdar lidi!\n";
5:      return 0;
6:  }
```

Výstup

```
Nazdar lidi!
```

Analýza Na řádku 0 je do aktuálního souboru zahrnut soubor `iostream`. Jak program funguje? Prvním znakem je #, což je signál pro preprocesor. Preprocesor se spustí při každém spuštění kompilátoru. Projde si zdrojový kód a najde řádky, které začínají symbolem křížku (#). S těmito řádky pracuje ještě před spuštěním samotného kompilátoru. Preprocesorem se budeme podrobně zabývat v den 21. – „Co dál?".

Příkaz `include` je instrukce pro preprocesor, která mu říká: „To, co následuje, je název souboru. Najdi tento soubor a vlož jeho obsah přímo do tohoto místa programu." Lomené závorky kolem názvu souboru preprocesoru říkají, aby tento soubor hledal na všech obvyklých místech. Pokud je váš kompilátor správně nastaven, lomené závorky znamenají, že preprocesor bude soubor `iostream` hledat v adresáři, který pro váš kompilátor uchovává všechny soubory, jež jsou obvykle předmětem operace zahrnutí. Soubor `iostream` (Input-Output-Stream, proud vstupu a výstupu) používá objekt `cout`, který zajišťuje psaní na obrazovku. Výsledkem řádku 0 bude zahrnutí souboru `iostream` do tohoto programu, jako kdybyste jeho obsah do programu napsali sami.

POZNÁMKA Preprocesor běží vždy před tím, než dojde k zavolání samotného kompilátoru. Každý řádek, který začíná symbolem #, přeloží preprocesor do zvláštního příkazu a připraví soubor s výsledným kódem pro kompilátor. Ne všechny kompilátory podporují direktivy `#include` bez zahrnutých přípon stejným způsobem. Objeví-li se nějaká chybová hlášení, pak může být zapotřebí upravit prohledávanou cestu v nastavení kompilátoru nebo za `#include` vložit také příponu souboru.

Na řádku 2 začíná vlastní program, a to funkcí s názvem `main()`. Každý program C++ má funkci `main()`. Funkcí rozumíme úsek kódu, který provádí jednu nebo více akcí. Funkce jsou obvykle spuštěny nebo zavolány jinými funkcemi, ale funkce `main()` má zvláštní význam: je zavolána automaticky ihned po spuštění programu jako první.

Podobně jako v případě jiných funkcí musí být i u funkce `main()` stanoveno, jaký typ hodnoty bude vracet. Typ vrácené hodnoty funkce `main()` v programu `NAZDAR.cpp` je `int`, což znamená, že tato funkce vrátí operačnímu systému po svém ukončení celočíselnou hodnotu. V našem případě vrací celočíselnou hodnotu 0, jak můžeme vidět na řádku 5. Vracení hodnoty operačnímu systému je relativně nepodstatná a málo používaná vlastnost, ale standard C++ vyžaduje, aby byla funkce `main()` deklarována výše uvedeným způsobem.

UPOZORNĚNÍ **Některé kompilátory dovolují funkci `main()` deklarovat tak, že vrací hodnotu `void`. To však není v C++ nadále přípustné a neměli byste si zbytečně osvojovat špatné návyky. Proto jako návratový typ funkce `main()` používejte `int` a na posledním řádku `main()` jednoduše vracejte hodnotu 0.**

POZNÁMKA **Některé operační systémy umožňují testovat hodnotu vrácenou programem. Podle úmluvy je návratová hodnota 0 signálem normálního ukončení programu.**

Všechny funkce jsou ohraničeny počáteční ({) a koncovou (}) složenou závorkou. Na řádcích 3 a 6 jsou závorky k funkci `main()`. Vše, co je mezi těmito závorkami, se považuje za součást dotyčné funkce.

Nejpodstatnější část celého programu najdeme na řádku 4.

Pro tisk na obrazovku se používá objekt `cout`. Objekty se budeme zabývat v den 6. – „Objektově orientované programování“ a s objektem `cout` a s ním souvisejícím objektem `cin` se pak podrobněji seznámíme v den 17. – „Proudy“. V jazyce C++ se tyto dva objekty, `cin` a `cout`, používají pro zajištění vstupu (například z klávesnice) a výstupu (například na obrazovku).

Objekt `cout` je nabízen standardní knihovnou. Knihovnou rozumíme sbírku tříd a standardní knihovna je standardní sbírka, která je součástí každého kompilátoru splňujícího standard ANSI.

Specifikátor oboru názvů `std` kompilátoru oznamuje, že objekt `cout`, který chcete použít, je součástí standardní knihovny, jejíž název je právě `std`. Protože v různých knihovnách či programových modulech mohou existovat objekty se stejným názvem, rozděluje C++ vše do tak zvaných „oborů názvů“ (neboli názvových prostorů – anglicky „namespace“). Kompilátoru se jeho použitím oznamuje: „Následující jméno `cout` je jméno, které je součástí standardního oboru názvů `std` a žádného jiného oboru názvů.“ Syntaxe vypadá tak, že před `cout` jsou umístěny znaky `std`, za nimiž následují dvě dvojtečky. V příštích dnech se o oborech názvů dozvíte více.

Objekt `cout` se používá takto: Napíšete slovo `cout` a za ně operátor přesměrování výstupu (<<). Cokoli následuje za operátorem přesměrování výstupu, objeví se na obrazovce. Jestliže chcete, aby se vypsal řetězec znaků, nezapomeňte jej dát do uvozovek ("), kterých si můžete všimnout i na našem řádku 4.

Poznámka **Všimněte si, že operátor přesměrování má formu dvou znaků „menší než“, mezi nimiž není mezera.**

Textovým řetězcem rozumíme posloupnost znaků, které je možné vytisknout.

Poslední dva znaky `\n` objektu `cout` říkají, aby za slovy Nazdar lidi! zalomil řádek. S touto zvláštní částí kódu se podrobněji seznámíte v den 18. – „Obory názvů“, která je věnována objektu `cout`.

Na řádku 6 je závorkou ukončena funkce `main()`.

Krátký pohled na objekt `cout`

V den 17. se dozvíte, jak s pomocí objektu `cout` tisknout data na obrazovku. Prozatím můžete tento objekt používat, aniž byste zcela chápali, jak funguje. Budete-li chtít na obrazovku vytisknout hodnotu, napíšete slovo `cout`, za nímž bude následovat operátor vložení (<<), tedy dvakrát za sebou zapsaný znak „menší než“ (<). Ačkoli se jedná o dva znaky, C++ s nimi zachází jako s jedním.

Za znak vložení zapíšete data. Ve výpisu 2.2 se můžete podívat, jak se objekt `cout` používá. Příklad zadejte přesně tak, jak vidíte, pouze s tou výjimkou, že namísto Jesse Liberty zadáte své jméno (pokud se ovšem nejmenujete Jesse Liberty).

Výpis 2.2

Používání objektu `cout`

```
 0: // Výpis 2.2 Používání objektu std::cout
 1: #include <iostream>
 2: int main()
 3: {
 4:      std::cout << "Dobry den.\n";
 5:      std::cout << "Zde je cislo 5: " << 5 << "\n";
 6:      std::cout << "Manipulacni objekt std::endl ";
 7:      std::cout << "zpusobi zalomeni radku na obrazovce.";
 8:      std::cout << std::endl;
 9:      std::cout << "Zde je velke cislo:\t\t" << 70000;
10:      std::cout << std::endl;
11:      std::cout << "Tady je soucet 8 a 5:\t\t";
12:      std::cout << 8+5 << std::endl;
13:      std::cout << "A tady mame zlomek:\t\t";
14:      std::cout << (float) 5/8 << std::endl;
15:      std::cout << "Zde je velmi velke cislo:\t";
16:      std::cout << (double) 7000 * 7000 << std::endl;
17:      std::cout << "Nezapomente zmenit Jesse Liberty ";
18:      std::cout << "na vase vlastni jmeno...\n";
19:      std::cout << "Jesse Liberty je programator C++!\n";
20:      return 0;
21: }
```

Výstup

```
Dobry den.
Zde je cislo 5: 5
Manipulacni objekt std::endl zpusobi zalomeni radku na obrazovce.
Zde je velke cislo:             70000
Tady je soucet 8 a 5:           13
A tady mame zlomek:             0.625
Zde je velmi velke cislo:       4.9e+007
Nezapomente zmenit Jesse Liberty na vase vlastni jmeno...
Jesse Liberty je programator C++!
```

UPOZORNĚNÍ Některé kompilátory budou považovat za chybu, když kolem výrazu pro součet nedáte kulaté závorky. V takovém případě by se řádek 12 změnil takto:

```
12: cout << (8+5) << std::endl;
```

Analýza Příkaz `#include <iostream>` na řádku 1 způsobí, že se soubor `iostream` přidá ke zdrojovému souboru. To je nutné vždy, když se dále používá objekt `cout` a s ním související funkce.

Řádek 4 ukazuje nejjednodušší použití objektu `cout`: tisk řetězce čili posloupnosti znaků. Symbol `\n` je zvláštní znak pro formátování, který objekt `cout` interpretuje tak, že na obrazovku vypíše znak pro nový řádek. Tento symbol vyslovujeme jako „zpětné lomítko en“ nebo „znak nového řádku“.

Na řádku 5 se objektu `cout` předají tři hodnoty, které jsou navzájem odděleny operátorem vložení. První hodnota je řetězec `"Zde je cislo 5: "`. Za dvojtečkou si všimněte mezery; ta je součástí řetězce. Pak se předá operátoru vložení hodnota 5 a znak nového řádku (vždy musí být v apostrofech nebo uvozovkách). Tím dojde k vytištění řádku:

```
Zde je cislo 5: 5
```

na obrazovku. Protože za prvním řetězcem není žádný znak nového řádku, bude se další hodnota tisknout bezprostředně za něj. Označujeme to zřetězení hodnot.

Na řádku 6 se vytiskne informativní zpráva a pak se použije manipulační objekt `endl`. Jeho účelem je napsat na obrazovku nový řádek. (Více se o používání `endl` dozvíte v den 16.) Všimněte si, že `endl` je také součástí standardní knihovny.

POZNÁMKA Manipulátor `endl` znamená konec řádku (z anglického „*end* line“), tedy l je písmeno el, nikoli jednička. Obyčejně se vyslovuje „end-el“. Místo `\n` je lepší používat `endl`, protože tento manipulátor se přizpůsobí používanému operačnímu systému, zatímco `\n` nemusí být na některých platformách znakem dostačujícím pro správné vytvoření nového řádku.

Na řádku 9 se poprvé setkáte s formátovacím znakem `\t`. Ten vloží na výstup znak tabulátoru a používá se na řádcích 9 až 15, aby byl výstup zarovnaný. Řádek 9 ukazuje, že je možné vedle malých celočíselných hodnot tisknout i velká čísla. Řádek 12 demonstruje, že objektu `cout` je možné předložit jednoduché sčítání. Objektu `cout` je předána hodnota 8+5, ale vytiskne se výsledný součet 13.

Na řádku 14 se do objektu `cout` vloží 5/8. Výraz `(float)` říká, že tato hodnota má být vyhodnocena jako desetinné číslo, a tak se vytiskne hodnota zlomku. Na řádku 16 je objektu `cout` předána hodnota 7000*7000 a výraz `(double)` způsobí, že ji objekt `cout` vyhodnotí jako číslo s plovoucí desetinnou čárkou. V den 3. – „Proměnné a konstanty“ – se s tímto vším znovu setkáte a získáte podrobnější informace, protože se zde budeme věnovat typům dat.

Na řádcích 17 a 20 jste měli doplnit své jméno a na výstupu tak potvrdit, že jste skutečně programátorem či programátorkou C++. A to musí být pravda, protože to řekl počítač!

Používání standardního oboru názvů

Možná po čase zjistíte, že používat `std::` před každým výskytem `cout` a `endl` začíná být trochu únavné. Ačkoli se jedná o přesný způsob stanovení oborů názvů, může vás jejich neustálé zadávání zdržovat. Standard ANSI nabízí dvě řešení tohoto drobného problému.

První možností je kompilátoru na začátku programu oznámit, že budete používat objekty `cout` a `endl` ze standardní knihovny, čehož příklad můžete vidět ve výpisu 2.3.

Výpis 2.3

Použití klíčového slova `using`

```
 0: // Výpis 2.3 Používání klíčového slova "using"
 1: #include <iostream>
 2: int main()
 3: {
 4:      using std::cout;
 5:      using std::endl;
 6:
 7:      cout << "Dobry den.\n";
 8:      cout << "Zde je cislo 5: " << 5 << "\n";
 9:      cout << "Manipulacni objekt endl ";
10:     cout << "zpusobi zalomeni radku na obrazovce.";
11:     cout << endl;
12:     cout << "Zde je velke cislo:\t\t" << 70000;
13:     cout << endl;
14:     cout << "Tady je soucet 8 a 5:\t\t";
15:     cout << 8+5 << endl;
16:     cout << "A tady mame zlomek:\t\t";
17:     cout << (float) 5/8 << endl;
18:     cout << "Zde je velmi velke cislo:\t";
19:     cout << (double) 7000 * 7000 << endl;
20:     cout << "Nezapomente zmenit Jesse Liberty ";
21:     cout << "na vase vlastni jmeno...\n";
22:     cout << "Jesse Liberty je programator C++!\n";
23:     return 0;
24: }
```

Výstup

```
Dobry den.
Zde je cislo 5: 5
Manipulacni objekt endl zpusobi zalomeni radku na obrazovce.
Zde je velke cislo:             70000
Tady je soucet 8 a 5:           13
A tady mame zlomek:             0.625
Zde je velmi velke cislo:       4.9e+007
Nezapomente zmenit Jesse Liberty na vase vlastni jmeno...
Jesse Liberty je programator C++!
```

Analýza Můžete si všimnout, že výstup je zcela identický. Jediný rozdíl mezi výpisy programů 2.3 a 2.2 jsou informace na řádcích 4 a 5, ve kterých kompilátoru sdělujeme, že budeme používat dané dva objekty ze standardní knihovny. Provedeme to s pomocí klíčového slova `using`. Díky tomu není nadále nutné kvalifikovat názvový prostor objektů `cout` a `endl`.

Druhý způsob, kterým se můžeme vyhnout nepraktickému psaní `std::` před objekty `cout` a `endl`, spočívá ve specifikaci celého standardního oboru názvů. To znamená, že u každého objektu, pokud nebude určeno jinak, se přepokládá, že pochází ze standardního oboru názvů. V tomto případě dáme tedy před zápisem `using std::cout;` přednost příkazu `using namespace std;`, který si můžete prohlédnout ve výpisu 2.4.

Výpis 2.4

Použití klíčového slova `namespace`

```
 0: // Výpis 2.4 Používání klíčového slova "namespace"
 1: #include <iostream>
 2: int main()
 3: {
 4:      using namespace std;
 5:
 6:      cout << "Dobry den.\n";
 7:      cout << "Zde je cislo 5: " << 5 << "\n";
 8:      cout << "Manipulacni objekt endl ";
 9:      cout << "zpusobi zalomeni radku na obrazovce.";
10:     cout << endl;
11:     cout << "Zde je velke cislo:\t\t" << 70000;
12:     cout << endl;
13:     cout << "Tady je soucet 8 a 5:\t\t";
14:     cout << 8+5 << endl;
15:     cout << "A tady mame zlomek:\t\t";
16:     cout << (float) 5/8 << endl;
17:     cout << "Zde je velmi velke cislo:\t";
18:     cout << (double) 7000 * 7000 << endl;
19:     cout << "Nezapomente zmenit Jesse Liberty ";
20:     cout << "na vase vlastni jmeno...\n";
21:     cout << "Jesse Liberty je programator C++!\n";
22:     return 0;
23: }
```

Analýza Výstup je opět identický s dřívějšími verzemi tohoto programu. Používání `using namespace std;` má tu výhodu, že nemusíte výslovně stanovit objekty, které skutečně používáte (například `cout` a `endl`). Na druhé straně se vystavujete riziku, že nechtěně použijete objekt z nesprávné knihovny.

Puristé upřednostňují psaní `std::` před každou instancí `cout` nebo `endl`. Ti línější dají přednost příkazu `using namespace std;` a jsou se vším ihned hotoví. V této knize to vyřešíme kompromisem a většinou uvedeme, které objekty používáme, ale občas si jen tak pro změnu vyzkoušíme i jiné způsoby.

Komentáře

Když program píšete, je vám naprosto jasné, co činíte. Je zajímavé, že už o měsíc později, když se ke svému programu vrátíte, se vám tak jasný a čitelný zdát nebude. Zůstává záhadou, jak se tyto nejasnosti do programu vloudí, ale dojde k tomu vždy.

Abyste se vyhnuli takovým zmatkům a pomohli i ostatním s porozuměním vašemu kódu, stačí začít používat komentáře. Jedná se o text, který kompilátor ignoruje, ale který čtenáře informuje o tom, co se v tom kterém místě programu děje.

Typy komentářů

Komentáře jsou v jazyce C++ dvojího druhu: jednořádkové a víceřádkové.

Jednořádkové komentáře se vytvářejí pomocí dvou lomítek (//). Kompilátor pak bude ignorovat vše, co následuje za tímto komentářem až do konce řádku.

Víceřádkové komentáře začínají lomítkem, za nímž následuje hvězdička (/*). Tato značka komentáře říká kompilátoru, aby přeskočil vše další až po znaky komentáře */. Uvedené znaky se mohou nacházet na jednom řádku, ale mohou mezi sebou mít také několik řádků – po každém otevíracím /* ovšem musí následovat uzavírací */.

Mnozí programátoři běžně používají jednořádkové komentáře a víceřádkové komentáře si ponechávají v rezervě pro „vykomentování“ (tedy zablokování) rozsáhlejších úseků programu. Do bloku víceřádkového vykomentovaného kódu můžete vložit jednořádkové komentáře a všechno mezi značkami pro víceřádkový komentář, včetně jednořádkových komentářů, bude ignorováno.

Poznámka **Víceřádkový komentář byl označován za komentář ve stylu jazyka C, protože byl zaveden a používán v jazyce C. Jednořádkové komentáře byly původní součástí C++, a nikoli jazyka C, takže se označovaly za komentáře ve stylu C++. Aktuální standard C i C++ nyní zahrnuje oba styly komentářů.**

Používání komentářů

Někdy se doporučuje uvést ke každé funkci komentář s vysvětlením, co funkce dělá a jaké hodnoty vrací.

Funkce by se měly pojmenovávat tak, aby nedocházelo k žádným sporům o tom, co je jejich účelem. Zavádějící a podivné úseky kódu by se měly předělat, aby bylo na první pohled zřejmé, o co v nich jde. Komentáře by rozhodně neměly řešit nesrozumitelnost kódu způsobenou líným programátorem.

Nechceme tím ale naznačit, že se komentáře nemají používat vůbec, rádi bychom jen podotkli, že byste se na ně u nesrozumitelného kódu neměli spoléhat. Místo toho kód raději upravte. Zkrátka, pište kód dobře a komentáře používejte jen jako doplněk.

Výpis 2.5 ilustruje použití komentářů a ukazuje, že nemají vliv na zpracování programu ani na jeho výstup.

Výpis 2.5

Program `POMOC.cpp` ilustruje používání komentářů

```
0: #include <iostream>
1:
2: int main()
3: {
4:     using std::cout;
5:
6:     /* toto je komentář
7:     a sahá až po ukončující
8:     značku hvězdičky a lomítka */
```

```
 9:     cout << "Nazdar lidi!\n";
10:     // tento komentář končí s koncem řádku
11:     cout << "Komentar skoncil!\n";
12:
13:     // komentáře s dvojitým lomítkem mohou být na samostatném řádku
14:     /* stejně jako komentáře s lomítkem a hvězdičkou */
15:     return 0;
16: }
```

Výstup

```
Nazdar lidi!
Komentar skoncil!
```

Analýza Komentáře na řádcích 6 až 8 jsou kompilátorem zcela ignorovány a podobně je tomu u komentářů na řádcích 10, 13 a 14. Komentář na řádku 10 skončil s koncem řádku, u komentářů na řádcích 6 a 14 je však třeba použít značku pro ukončení komentáře.

Poznámka Existuje ještě třetí styl komentáře, jak jej podporují některé kompilátory jazyka C++. Tyto komentáře se označují jako dokumentační a vyznačují se třemi lomítky (///). Kompilátory zahrnující podporu tohoto stylu komentářů vám umožňují generovat z nich dokumentaci programu. Protože ovšem nejsou v současné době součástí standardu C++, podrobněji se jimi zde nezabýváme.

A na co si dát u komentářů pozor

O komentářích, které popisují něco zřejmého, se dá jen stěží prohlásit, že jsou k něčemu dobré. Ve skutečnosti mohou být spíše kontraproduktivní, protože kód se může měnit a programátor může opomenout komentáře aktualizovat. Avšak co je zřejmé někomu, může být pro někoho jiného tajemné, takže je třeba jistého úsudku.

Podstatou zůstává, že komentáře by neměly říkat *co* s děje, ale *proč* se to děje.

Funkce

Ačkoli je `main()` funkcí, jedná se o funkci neobvyklou. Má-li být funkce užitečná, musí být možné ji v průběhu programu zavolat. Funkci `main()` volá operační systém.

Řádky programu jsou postupně prováděny v pořadí, v jakém se objevují ve zdrojovém kódu, dokud není zavolána funkce. V té chvíli se program rozvětví, aby funkci provedl. Když je funkce dokončena, vrátí řízení na řádek kódu, který bezprostředně následuje za voláním této funkce.

Jako analogie nám může dobře posloužit ořezávání tužky. Když kreslíte obrázek a zlomí se vám tužka, asi přestanete kreslit a jdete si tužku ořezat. Pak se vrátíte na místo, kde jste předtím přerušili činnost. Když program potřebuje provést nějakou službu, může zavolat funkci, aby tuto službu vykonala. Když pak funkce svou práci ukončí, vrátí se program za místo, kde funkci zavolal. Výpis 2.6 nám to ilustruje.

Výpis 2.6

Volání funkce

```
 0: #include <iostream>
 1:
 2: // funkce IlustracniFunkce
 3: // vytiskne velmi užitečnou zprávu
 4: void IlustracniFunkce()
 5: {
 6:     std::cout << "Uvnitr Ilustracni funkce\n";
 7: }
 8:
 9: // funkce main - vytiskne zprávu, pak zavolá
10: // ilustrační funkci a nakonec vytiskne
11: // druhou zprávu
12: int main()
13: {
14:     std::cout << "Ve funkci main\n";
15:     IlustracniFunkce();
16:     std::cout << "Zpet ve funkci main\n";
17:     return 0;
18: }
```

Výstup

```
Ve funkci main
Uvnitr Ilustracni funkce
Zpet ve funkci main
```

Analýza Funkce `IlustracniFunkce()` je definována na řádcích 5–7. Když je zavolána, vytiskne na obrazovku zprávu a pak vrátí řízení.

Řádkem 12 začíná vlastní program. Na řádku 14 funkce `main()` vytiskne zprávu, že řízení je ve funkci `main()`. Po vytištění zprávy je na řádku 15 zavolána funkce `IlustracniFunkce()`. Toto volání způsobí, že se provede příkaz v těle `IlustracniFunkce()`. V tomto případě se celá funkce skládá z kódu na řádku 6, který vytiskne další zprávu. Když je `IlustracniFunkce()` u konce (řádek 7), vrátí se řízení za místo, odkud byla zavolána. V tomto případě se řízení vrátí na řádek 16, kde funkce `main()` vytiskne svůj poslední řádek.

Používání funkcí

Funkce vrací buď skutečnou hodnotu, nebo prázdnou hodnotu typu `void`, tedy nic. Funkce, která má sečíst dvě celočíselné hodnoty, vrátí patrně součet; proto by měla být deklarována s celočíselnou návratovou hodnotou. Funkce, která má pouze vytisknout zprávu, nemá co vrátit a měla by být deklarována tak, že vrátí hodnotu typu `void`.

Funkce se skládá z hlavičky a těla. Hlavička pak obsahuje návratový typ, název funkce a parametry, které k této funkci příslušejí. Parametry umožňují předání hodnot funkci. Kdyby tedy měla funkce sečíst dvě čísla, byly by sčítance parametry funkce. Zde je příklad typické hlavičky funkce:

```
int Soucet(int a, int b)
```

Parametrem rozumíme deklaraci typu hodnoty, která se bude funkci předávat. Skutečná hodnota předávaná funkci při zavolání se nazývá argument. Mnoho programátorů termíny argument a parametr používá jako synonyma. Jiní jsou si naopak odlišnosti mezi nimi vědomi. V této knize se budou oba termíny používat zaměnitelně.

Tělo funkce se skládá z počáteční složené závorky, žádného nebo více příkazů a závěrečné složené závorky. Příkazy tvoří samotnou činnost funkce. Funkce může vracet specifickou hodnotu, k čemuž slouží příkaz `return`. Tento příkaz zároveň funkci ukončuje. Jestliže ve funkci příkaz `return` vynecháte, bude při ukončení automaticky vrácena hodnota `void`. Vrácená hodnota musí být stejného typu, jaký je deklarován v hlavičce funkce.

POZNÁMKA Funkcemi se budeme podrobněji zabývat v den 5. – „Funkce“. Typy, které mohou být funkcí vráceny, probíráme v den 3. – „Proměnné a konstanty“. Informace, které uvádíme dnes, jsou pouze základního charakteru; funkce totiž budete používat téměř v každém programu C++.

Výpis 2.7 ilustruje funkci, která přijímá dva celočíselné parametry a vrací celočíselnou hodnotu. Prozatím se neznepokojujte syntaxí ani podrobnostmi týkajícími se práce s celočíselnými hodnotami (například `int x`). Podrobněji se tomuto tématu budeme věnovat v den 3.

Výpis 2.7

Program `FUNKCE.cpp` ukazuje příklad jednoduché funkce

```
 0: #include <iostream>
 1: int Soucet(int x, int y)
 2: {
 3:     std::cout << "Ve funkci Soucet(), prijato " << x << " a " << y << "\n";
 4:     return (x+y);
 5: }
 6:
 7: int main()
 8: {
 9:     using std::cout;
10:     using std::cin;
11:
12:
13:     cout << "Ve funkci main()!\n";
14:     int a, b, c;
15:     cout << "Vlozte dve cisla: ";
16:     cin >> a;
17:     cin >> b;
18:     cout << "\nVolani funkce Soucet()\n";
19:     c=Soucet(a,b);
20:     cout << "\nZpet ve funkci main().\n";
21:     cout << "c bylo nastaveno na " << c;
22:     cout << "\nKonec...\n\n";
23:     return 0;
24: }
```

Výstup

```
Ve funkci main()!
Vlozte dve cisla: 3 5

Volani funkce Soucet()
Ve funkci Soucet(), prijato 3 a 5

Zpet ve funkci main().
c bylo nastaveno na 8
Konec...
```

Analýza Na řádku 1 je definována funkce `Soucet()`. Ta přijímá dva celočíselné parametry a vrací celočíselnou hodnotu. Na řádku 7 pak začíná samotný program. Ten uživatele vyzve, aby zadal dvě čísla (řádky 15 až 17). Zadávaná čísla uživatel oddělí mezerou a pak stiskne klávesu Enter. Funkce `main()` na řádku 19 předá uživatelem zadané hodnoty funkci `Soucet()` jako argumenty.

Řízení programu se předá funkci `Soucet()`, která začíná na řádku 1. Nejdříve se parametry `a` a `b` vytisknou a pak se sečtou. Na řádku 4 dojde k navrácení výsledku a funkce se ukončí.

Na řádcích 16 a 17 slouží objekt `cin` k získání číselných hodnot pro proměnné `a` a `b`, poté objekt `cout` k vypsání hodnot na obrazovku. V příštích kapitolách se k tomuto programu vrátíme a podrobněji si rozebereme proměnné a ostatní jeho aspekty.

Metody a funkce

I když budete funkci říkat jinak, stále se jedná o funkci. Stojí zde za zmínku, že různé programovací jazyky a různé programovací metodiky mohou označovat funkce odlišným termínem. Jedním z často používaných označení je metoda. Metoda je jen jiným označením pro funkci, která je součástí nějaké třídy.

Poznámky pro českého čtenáře

S používáním jazyka C++ v českém prostředí souvisí určité problémy, kterých si musíte být vědomi. Tento krátký oddíl představuje důležité informace především pro uživatele, kteří zatím nemají zkušenosti s prací v žádném z programovacích jazyků. Dále uvedená pravidla mají totiž celkem obecnou platnost.

Desetinná tečka

Základní věcí, na kterou nesmíte nikdy zapomínat, je skutečnost, že při zadávání desetinných čísel do programového kódu se vždy používá jako oddělovač desetinných míst **tečka**, a nikoli čárka. To je dáno tím, že programovací jazyky vycházejí z anglických konvencí zadávání čísel, kde se čárka používá k oddělování tisíců, milionů atd. (u nás je to mezera) a tečka označuje konec celočíselné a začátek zlomkové části hodnoty (u nás se používá čárka).

Jak uvidíte, čárka má v jazyce C++ naprosto odlišný význam. Důležité je, že čísla v kódu musíte vždy zapisovat s desetinnou tečkou. Pokud na to zapomenete, program se nepřeloží (nezkompiluje). Desetinná tečka je přímo součástí definice jazyka a její funkci tedy neovlivňuje ani místní nastavení aktuálního systému Windows, jako je tomu v jiných programech, například v sadě Microsoft Office.

Diakritika

Druhou problematickou oblastí je diakritika neboli znaky s háčky a čárkami. Potíže v tomto ohledu lze rozdělit do tří kategorií:

- Jako **názvy proměnných, funkcí** atd. lze používat jen dolní část tabulky ASCII. To znamená, že vaše vlastní názvy elementů v jazyce C++ nemohou obsahovat žádné „české“ znaky, tedy písmena s diakritickými znaménky. Pokud nějaký náhodou použijete, kompilátor bude hlásit chybu. Při vytváření názvů se musíte omezit na standardní malá a velká písmena anglické abecedy a další povolené znaky, jak bude popsáno dále. (To neplatí např. v Javě, která pracuje s 16bitovými znaky sady Unicode a plně podporuje kompletně české názvy čehokoli – proměnných, funkcí, tříd atd.)
- Protože kompilátor ignoruje veškerý text mezi znaky komentáře na více řádcích nebo od symbolu komentáře do konce řádku, lze diakritiku bez nepříznivých následků na „přeložitelnost“ programu používat v **komentářích kódu** (jako je tomu v této knize). Jestliže se vám znaky s diakritikou ve vývojovém prostředí nezobrazují správně, často lze tuto chybu napravit změnou nastavení vývojového prostředí. V nabídce hledejte postupně položky jako Tools (Nástroje), Options (Možnosti), Environment (Prostředí) a Font (Písmo). Musíte si však být vědomi toho, že díky nekompatibilitě znakových sad se české komentáře vytvořené třeba v prostředí Windows nezobrazí správně v Dosu a na Unixu, takže se vlastně omezuje přenositelnost programového kódu.
- S nekompatibilními znakovými sadami souvisí rovněž potíže s výstupem českých znaků z programů. Například všechny ukázkové programy v této knize jsou napsané a zkompilované v prostředí Windows, **textový výstup** je však směrován do okna konzoly neboli systému DOS. Protože kódování „nadstandardních“ znaků (horní polovina tabulky ASCII) se v systémech DOS a Windows liší, nezobrazí se správná písmena s diakritickými znaménky. Tento problém by se neprojevil, pokud byste zůstávali čistě v prostředí Windows. Protože však všechny příklady v této knize používají okno konzoly, pracujeme jen se standardními znaky. Totéž platí i pro řetězce v jazyce C++.

Souhrn

Obtížnost studia tak komplexního předmětu, jakým programování bezesporu je, spočívá v tom, že právě probíraná látka je mnohdy závislá na znalostech, které se máte teprve naučit. V této kapitole jsme se seznámili se základními částmi jednoduchého programu C++. Popsali jsme také vývojový cyklus a zavedli několik nových důležitých pojmů.

Otázky a odpovědi

Otázka: **K čemu slouží `#include`?**

Odpověď: Jedná se o pokyn preprocesoru, který se sám spustí, když zavoláte kompilátor. Tento specifický pokyn způsobí, že soubor uvedený za slovem `include` je načten do zdrojového souboru, jako byste jej na toto místo celý opsali.

Otázka: **Jaký je rozdíl mezi komentáři stylu // a stylu /*?**

Odpověď: Platnost komentáře ve tvaru dvojitého lomítka „vyprší“ na konci řádku. Komentáře ve tvaru lomítka s hvězdičkou (/*) končí, až když program narazí na ukončující kombinaci znaků (*/). Pamatujte si, že ani konec funkce neukončí tento typ komentáře. Musíte tedy vždy takový komentář ukončit, jinak dojde k chybě při kompilaci.

Otázka: **Jak se odlišuje špatný komentář od dobrého komentáře?**

Odpověď: Dobrý komentář svému čtenáři sděluje, proč konkrétní úsek kódu dělá to, co dělá, nebo vysvětluje, co jistý úsek kódu činí. Špatný komentář jen znovu opakuje, co konkrétní řádek kódu dělá. Řádky kódu by se měly psát tak, aby hovořily samy za sebe. Když si přečtete řádek kódu, měli byste bez komentáře pochopit, co se na něm děje.

Úkoly pro vás

Testovací otázky v této části vám pomohou upevnit znalosti, které jste v této kapitole získali. Cvičení slouží k tomu, aby vám nabídlo praktickou zkušenost s tím, co jste se naučili. Pokuste se na otázky v testu a cvičení odpovědět dříve, než se podíváte do klíče v dodatku D. Než přejdete k další kapitole, ujistěte se, že každé odpovědi rozumíte.

Test

1. V čem spočívá rozdíl mezi kompilátorem a preprocesorem?
2. Čím je funkce `main()` zvláštní?
3. Jaké dva typy komentářů znáte a jak se od sebe odlišují?
4. Je možné komentáře vnořovat?
5. Mohou být komentáře delší než jeden řádek?

Cvičení

1. Napište program, který na obrazovku napíše „C++ se mi moc líbí“.
2. Napište nejmenší program, který je možné zkompilovat, sestavit a spustit.
3. ODHALENÍ CHYBY: Zadejte tento program a zkompilujte jej. Proč selhal? Jak jej můžete opravit?

```
1: #include <iostream>
2: main()
3: {
4:       std::cout << Je zde chyba?";
5: }
```

4. Opravte chybu v programu z cvičení 3, program znovu zkompilujte, sestavte a spusťte.
5. Upravte výpis 2.7 tak, aby zahrnoval funkci odečítání. Nazvěte ji `Rozdil()` a použijte ji stejným způsobem jako funkci `Soucet()`. Předejte jí stejné hodnoty jako funkci sčítání.

den 3

Proměnné a konstanty

Programy potřebují nějakým způsobem uchovávat data, která používají. Různé způsoby reprezentace, uchování a manipulace s daty nabízí proměnné a konstanty.

Dnes se naučíte a dozvíte:

- Jak se deklarují a definují proměnné a konstanty.
- Jak se proměnným přiřazují hodnoty a jak se s těmito hodnotami zachází.
- Jak se na obrazovku vypíše hodnota proměnné.

Co je to proměnná?

Proměnná je v programu C++ místo, kde se uchovává informace. Rozumíme tím místo v paměti počítače, na které můžete hodnotu uložit a ze kterého ji můžete později znovu získat.

Proměnné poskytují pouze dočasné uchování informací. Když počítač vypnete, tyto hodnoty se ztratí. Trvalé uchování hodnot je jiná záleži-

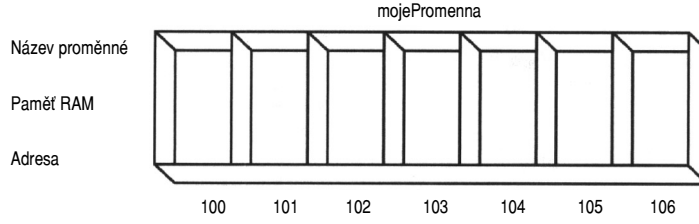
tost a obvykle se realizuje v databázi nebo v souboru na disku. Ukládání hodnot do souborů na disku se budeme věnovat v den 16. – „Pokročilá dědičnost“.

Reprezentace dat v paměti

Paměť počítače si můžeme představit jako řadu přihrádek, které jsou seřazeny vedle sebe. Každé přihrádce – neboli umístění v paměti – je postupně přiřazeno číslo. Těmto číslům říkáme adresy v paměti. Proměnná si vyhradí jednu nebo více přihrádek, do kterých může uložit hodnotu.

Názvem proměnné (například `mojePromenna`) rozumíme jmenovku na přihrádce, díky které ji můžete snadno najít, aniž byste museli znát skutečnou adresu v paměti. Na obrázku 3.1 si můžete prohlédnout schematické znázornění paměti. Povšimněte si, že proměnná `mojePromenna` začíná na paměťové adrese s číslem 103. Podle velikosti pak může proměnná `mojePromenna` zabrat jednu nebo více paměťových adres.

Obrázek 3.1
Schematické znázornění paměti



POZNÁMKA Zkratka *RAM* pochází z angličtiny a znamená „Random Access Memory“, tedy paměť s přímým (náhodným) přístupem. Program se po svém spuštění načte ze souboru na disku do paměti RAM. Zde se také vytvoří všechny proměnné. Když programátoři hovoří o paměti, mají obvykle na mysli právě paměť RAM.

Vymezení paměti

Když definujete v C++ proměnnou, musíte kompilátoru oznámit, o jaký druh proměnné se jedná: zda je to celé číslo, znak a podobně. Podle této informace vyhradí kompilátor v paměti příslušné místo a zároveň ví, jaký druh hodnoty chcete v proměnné uchovávat.

Kompilátor také získává možnost varovat vás nebo zobrazit chybu, když se náhodou pokusíte uložit do takové proměnné hodnotu nesprávného typu. (Programovací jazyk s touto charakteristikou se označuje za „silně typový“.)

Každá přihrádka má velikost jeden bajt. Jestliže typ proměnné, kterou vytvoříte, bude mít velikost čtyři bajty, vyhradí se čtyři bajty paměti neboli čtyři přihrádky. Prostřednictvím typu proměnné (například celé číslo) kompilátoru sdělíte, kolik paměti (kolik přihrádek) si má pro tuto proměnnou vyhradit.

Kdysi musel programátor bitům a bajtům dobře rozumět, jedná se nakonec o základní jednotky uložení informací. Počítačové programy se zlepšily a dnes není nutné znát všechny detaily. Stále je však užitečné vědět, jak se data ukládají. Krátký přehled základů binární matematiky najdete v dodatku A „Binární a hexadecimální aritmetika“.

POZNÁMKA Jestliže se vám při slově matematika zachtělo knihu rychle zavřít, pak se s dodatkem A nezatěžujte, nebudete jej nezbytně potřebovat. Je skutečností, že v dnešní době už programátor nemusí být zároveň matematikem, ačkoli logické a racionální uvažování by mu přesto mělo být vlastní.

Velikost celých čísel

Na každém počítači zabere každý typ proměnné v paměti jednotnou a neměnnou velikost. To znamená, že celé číslo může mít na jednom počítači velikost dva bajty a na druhém čtyři, ale na každém z nich to bude za všech okolností stejně.

Proměnná typu `char` (používá se k ukládání znaků) má většinou velikost jednoho bajtu.

POZNÁMKA Vedou se nekonečné debaty o tom, jak by se mělo vyslovovat slovo `char`. Někteří jej vyslovují „kar“, jiní „char“, další zase „kér“. Jisté je, že výslovnost „kar“ je správná, protože ji používám *já*, ale vy si samozřejmě můžete vybrat podle svého.

Celé číslo typu `short` má na většině počítačů velikost dva bajty, celé číslo typu `long` má obvykle velikost čtyři bajty a celé číslo (bez klíčových slov `short` nebo `long`) může mít dva nebo čtyři bajty. Asi byste očekávali, že to jazyk bude specifikovat přesněji, ale není tomu tak. Pouze říká, že typ `short` musí mít velikost menší nebo rovnu velikosti typu `int`, která dále musí být menší nebo rovna velikosti typu `long`.

S největší pravděpodobností pracujete právě s počítačem, jehož celočíselné hodnoty typu `short` mají velikost dva bajty, u typu `int` jsou to čtyři bajty a v případě typu `long` také čtyři bajty.

Velikost celého čísla určuje procesor (16bitový nebo 32bitový) a kompilátor, který používáte. U moderních 32bitových počítačů (Pentium), které používají moderní kompilátory (například Visual C++ 4 nebo novější), mají celočíselné hodnoty velikost *čtyři* bajty.

Pozor Při vytváření programů nikdy nesmíte předpokládat, kolik paměti jednotlivé typy využívají.

Na svém počítači nyní zkompilujte a spusťte program z výpisu 3.1. Řekne vám, jaká je ve vašem případě přesná velikost jednotlivých datových typů.

Výpis 3.1

Určení velikosti typů proměnných na vašem počítači

```
0: #include <iostream>
1:
2: int main()
3: {
4:     using std::cout;
5:
6:     cout << "Velikost typu int je:\t\t"
7:          << sizeof(int) << " bajtu.\n";
8:     cout << "Velikost typu short je:\t\t"
9:          << sizeof(short) << " bajtu.\n";
```

```
10:     cout << "Velikost typu long je:\t\t"
11:          << sizeof(long) << " bajtu.\n";
12:     cout << "Velikost typu char je:\t\t"
13:          << sizeof(char) << " bajtu.\n";
14:     cout << "Velikost typu float je:\t\t"
15:          << sizeof(float) << " bajtu.\n";
16:     cout << "Velikost typu double je:\t"
17:          << sizeof(double) << " bajtu.\n";
18:     cout << "Velikost typu bool je:\t\t"
19:          << sizeof(bool) << " bajtu.\n";
20:
21:     return 0;
22: }
```

Výstup

```
Velikost typu int je:           4 bajtu.
Velikost typu short je:         2 bajtu.
Velikost typu long je:          4 bajtu.
Velikost typu char je:          1 bajtu.
Velikost typu float je:         4 bajtu.
Velikost typu double je:        8 bajtu.
Velikost typu bool je:          1 bajtu.
```

POZNÁMKA Na vašem počítači se uvedené počty bajtů mohou lišit.

Většina kódu, který najdete ve výpisu 3.1, by vám už neměla činit problémy. Některé řádky jsme rozdělili, aby odpovídaly velikosti stránky v knize. Například obsah řádků 6 a 7 by ve skutečnosti mohl být na jediném řádku. Kompilátor prázdné prostory ignoruje (mezery, tabulátory, znaky konce řádku), takže s řádky 6 a 7 bude zacházet jako s jedním řádkem.

Tento program uvádí novinku, kterou je na řádcích 6 až 19 operátor `sizeof()`. Je součástí kompilátoru a vrací velikost objektu, který mu předáte jako parametr. Například na řádku 7 se operátoru `sizeof()` předá klíčové slovo `int`. Dnes se ještě dozvíte, že se `int` používá k popisu standardní celočíselné proměnné. Když použijete `sizeof` na počítači s Pentiem 4 a systémem Windows XP, pak má `int` velikost čtyři bajty, což je náhodou stejný počet jako má typ `long int` na témže počítači.

Další řádky výpisu 3.1 ukazují velikosti jiných datových typů. Více se o možném obsahu těchto datových typů a rozdílech mezi nimi dozvíte za chvíli.

Celá čísla se znaménkem (`signed`) a bez znaménka (`unsigned`)

U všech celočíselných typů rozlišujeme dvě varianty: čísla se znaménkem (`signed`) a bez znaménka (`unsigned`). Podstata spočívá v tom, že za jistých okolností můžete potřebovat pracovat se zápornými čísly a jindy ne. Celá čísla (všech typů, i `short` a `long`), která nejsou explicitně označena jako „`unsigned`“, tedy bez znaménka, se považují za „`signed`“, tedy se znaménkem. Celá čísla se znaménkem mohou nabývat kladných i záporných hodnot. Celá čísla bez znaménka jsou vždy kladná.

Protože pro obě varianty celých čísel, `signed` i `unsigned`, je vyhrazen stejný počet bajtů, bude největší číslo, které můžete uložit v celočíselné proměnné bez znaménka, dvakrát větší než největší kladné číslo, které můžete uložit do celočíselné proměnné se znaménkem. Celá čísla typu `unsigned short` mohou tedy nabývat hodnot od 0 do 65 535. Poloviční počet

možných hodnot připadajících na typ celého čísla `signed short` je záporný. Celá čísla typu `signed short` mohou proto nabývat hodnot od –32 768 do 32 767. Další informace o přednostech operátorů najdete v příloze C.

Základní typy proměnných

Do jazyka C++ je zabudováno několik dalších typů proměnných. Je možné je rozdělit na celočíselné proměnné (těmi jsme se dosud zabývali), proměnné s pohyblivou řádovou (desetinnou) čárkou a znakové proměnné.

Proměnné s pohyblivou řádovou čárkou nabývají hodnot, které lze vyjádřit ve tvaru zlomku; jedná se tedy o reálná čísla. Znakové proměnné zabírají jeden bajt a používají se k uchování 256 znaků a symbolů ze sady znaků ASCII nebo rozšířené sady ASCII.

Poznámka Sadou znaků ASCII rozumíme sadu znaků standardizovaných pro používání na počítačích. Jedná se o zkratku názvu „American Standard Code for Information Interchange“ (americký standardní kód pro výměnu informací). Sada ASCII je podporována téměř všemi počítačovými operačními systémy, ačkoli mnohé z nich podporují také mezinárodní sady znaků.

V tabulce 3.1 najdete přehled typů proměnných, které se v programech C++ používají. U každého typu proměnné v tabulce najdete jeho předpokládanou velikost v paměti a druh hodnot, které lze do proměnné tohoto typu uložit. Mezní hodnoty jsou však určeny velikostí typu v paměti, proto raději zkonzultujte váš výstup z programu 3.1. Je pravděpodobné, že zaznamenáte shodné velikosti, pokud ovšem nepoužíváte počítač se 64bitovým procesorem.

Tabulka 3.1 Typy proměnných

Typ	Velikost	Hodnoty
`bool`	1 bajt	`true` nebo `false`
`unsigned short int`	2 bajty	0 až 65 535
`short int`	2 bajty	-32 768 až 32 767
`unsigned long int`	4 bajty	0 až 4 294 967 295
`long int`	4 bajty	-2 147 483 648 až 2 147 483 647
`int` (16 bitů)	2 bajty	-32 768 až 32 767
`int` (32 bitů)	4 bajty	-2 147 483 648 až 2 147 483 647
`unsigned int` (16 bitů)	2 bajty	0 až 65 535
`unsigned int` (32 bitů)	4 bajty	0 až 4 294 967 295
`char`	1 bajt	256 znakových hodnot
`float`	4 bajty	1,2e-38 až 3,4e38
`double`	8 bajtů	2,2e-308 až 1,8e308

POZNÁMKA Velikost proměnných se může od hodnot uvedených v tabulce 3.1 lišit, a to v závislosti na typu kompilátoru a počítače, který používáte. Jestliže jste dostali stejný výstup, jaký je uveden u výpisu 3.1, měla by tabulka 3.1 platit i pro váš kompilátor. Jestliže se výstup u výpisu 3.1 lišil, měli byste nahlédnout do příručky ke kompilátoru a zjistit hodnoty, kterých mohou proměnné nabývat.

Definice proměnné

Již jste viděli vytvoření a používání řady proměnných. Nyní se dozvíte, jak vytvářet své vlastní. Proměnná se zavádí uvedením jejího typu, za nímž následuje jedna nebo více mezer, název proměnné a středník. Název proměnné může tvořit prakticky jakákoli kombinace písmen a číslic, ale nesmí obsahovat žádné mezery. Povolené názvy proměnných jsou `x`, `J23qrsnf` a `mujVek`. Názvy proměnných obvykle zároveň informují, k čemu slouží. Vhodně zvolené názvy proměnných usnadní porozumění programu. Následující příkaz definuje celočíselnou proměnnou s názvem `mujVek`:

```
int mujVek;
```

POZNÁMKA Když deklarujete proměnnou, dojde k alokaci (vyhrazení) paměti pro tuto proměnnou. Hodnota proměnné v paměti bude v tomto okamžiku náhodná. Za chvíli uvedeme, jak se do této paměti přiřadí nová hodnota.

K dobrým zásadám programování patří, že se snažíme vyhýbat názvům proměnných, jako je `J23qrsnf`. V případě proměnných, které budeme používat jen krátce, se omezujeme na jednopísmenné názvy (například `x` nebo `i`). Snažte se také používat názvy jako `mujVek` nebo `mnozstvi`. Takové názvy vám mohou pomoci, když si o tři týdny později budete lámat hlavu, co jste jistým řádkem kódu vlastně mysleli.

Proveďte následující pokus: Zkuste u uvedených programů uhodnout, co dělají, na základě prvních několika řádků:

Příklad 1

```
int main()
{
       unsigned short x;
       unsigned short y;
       unsigned short z;
       z = x * y;
       return 0;
}
```

Příklad 2

```
int main()
{
       unsigned short Sirka;
       unsigned short Delka;
       unsigned short Plocha;
       Plocha = Sirka * Delka;
       return 0;
}
```

POZNÁMKA Jestliže tento program zkompilujete, dostanete od kompilátoru upozornění, že nedošlo k inicializaci hodnot. Za chvíli uvedeme, jak tento problém odstranit.

Je zřejmé, že v druhém případě asi uhodnete snáze, k čemu je program určen. Nepohodlné psaní delších názvů proměnných se více než vyplatí, protože takový program se mnohem lépe čte i udržuje.

Rozlišování velkých a malých písmen

Jazyk C++ rozlišuje mezi velkými a malými písmeny. Jinými slovy, proměnná s názvem `vek` není totožná s proměnnou, jejíž název je `Vek`, a to je zase jiná proměnná než `VEK`.

POZNÁMKA U některých kompilátorů je možné rozlišování mezi velkými a malými písmeny vypnout. Raději to však nezkoušejte; vaše programy pak přestanou pracovat v prostředí jiných kompilátorů a kód se stane pro ostatní programátory C++ zavádějící.

Názvy proměnných

Můžete se setkat s různými způsoby pojmenování proměnných. Příliš nezáleží na tom, který si zvolíte, je však důležité v celém programu tento způsob konzistentně dodržovat. V opačném případě se budou v kódu ostatní programátoři jen obtížně orientovat.

Mnozí programátoři v názvech proměnných raději používají malá písmena. Jestliže je vhodné v názvu proměnné použít dvě slova (například „moje auto“), jsou rozšířeny dva způsoby pojmenování: `moje_auto` nebo `mojeAuto`. Pro druhý případ se vžilo označení velbloudí zápis, protože velká písmena uvnitř názvu připomínají velbloudí hrby.

Některým se lépe čte název s podtržítkem (_), ale jiní se mu snaží vyhýbat, protože jeho psaní je poněkud nepohodlné. V této knize budeme používat ten druhý typ zápisu, ve kterém všechna následující slova v názvu proměnné začínají velkým písmenem: `mojeAuto`, `rychlaHnedaLiska` a podobně.

Mnozí zkušení programátoři používají styl zápisu, kterému se říká maďarský. U tohoto typu zápisu se před název proměnné přidá předpona, která se skládá ze sady znaků popisujících typ proměnné. Zápis názvu celočíselné proměnné by mohl začínat malým písmenem i, v případě typu `long` malým písmenem l a podobně. Dále by se podobným způsobem označily například konstanty, ukazatele, globální proměnné a tak dále. Většina z těchto úmluv má své opodstatnění při programování v C, v této knize je však používat nebudeme.

Poznámka Tento typ zápisu se označuje jako maďarský podle jeho autora. Jedná se o Charlese Simonyiho z Microsoftu, který pochází z Maďarska. Jeho původní monografii můžete najít na `http://www.strangecreations.com/library/c/naming.txt`**.**

Microsoft od tohoto způsobu zápisu nedávno upustil a v doporučeních pro syntaxi v C# se výslovně radí maďarský způsob zápisu *nepoužívat*. Jejich zdůvodnění pro C# platí i pro C++.

Klíčová slova

V jazyce C++ jsou některá slova vyhrazena a vy je nesmíte používat jako názvy proměnných. Jedná se o názvy příkazů, které kompilátor interpretuje jako pokyny pro řízení programu. Mezi klíčová slova patří `if`, `while`, `for` a `main`. V příručce ke kompilátoru najdete jejich úplný seznam, ale obecně se dá říci, že žádný vhodný název proměnné není klíčovým slovem. Seznam klíčových slov jazyka C++ najdete v dodatku B a také v tabulce 3.2.

Tabulka 3.2 Klíčová slova jazyka C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Dále jsou rezervována následující slova:

and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

ANO	NE
ANO, definujte proměnnou tak, že napíšete její typ a pak název.	**NE**, nepoužívejte klíčová slova jazyka C++ jako názvy proměnných.
ANO, používejte smysluplné názvy proměnných.	**NE**, nevytvářejte programy tak, aby závisely na počtu bajtů používaných k uložení nějaké proměnné.
ANO, pamatujte na to, že C++ rozlišuje malá a velká písmena	**NE**, nepoužívejte pro záporná čísla proměnné typu `unsigned`.
ANO, snažte se uvědomit si, jaký počet bajtů každá proměnná v paměti spotřebuje a jaké hodnoty lze do proměnných určitého typu uložit.	

Vytvoření více proměnných najednou

V jednom výrazu je možné definovat i více proměnných najednou. Napíšete nejdříve typ a pak názvy proměnných, které oddělíte čárkami. Příklad:

```
unsigned int mujVek, mojeVaha;  // dvě celočíselné proměnné bez znaménka
long int plocha, sirka, delka;  // tři proměnné typu long
```

Jak vidíte, proměnné `mujVek` i `mojeVaha` jsou obě deklarovány jako celočíselné proměnné typu `unsigned`, bez znaménka. Na druhém řádku je deklarace tří samostatných proměnných typu `long` s názvy `plocha`, `sirka` a `delka`. Typ `long` je přiřazen ke všem třem proměnným; v jednom příkazu pro definici není možné typy proměnných míchat.

Přiřazení hodnoty do proměnné

Hodnota se do proměnné přiřazuje s pomocí operátoru přiřazení (=). Proměnné `Sirka` přiřadíte hodnotu 5 takto:

```
unsigned short Sirka;
Sirka = 5;
```

POZNÁMKA `long` **je zkrácená verze pro** `long int` **a** `short` **je zkrácená verze pro** `short int`**.**

Oba tyto kroky můžete spojit a proměnnou `Sirka` inicializovat přímo v definici:

```
unsigned short Sirka = 5;
```

Inicializace se velmi podobá přiřazení a u celočíselných proměnných je rozdíl minimální. Později, když budeme probírat konstanty, uvidíte, že některé hodnoty je nutné inicializovat, protože není možné do nich později přiřadit hodnotu. Rozdíl spočívá především v tom, že k inicializaci dochází ve stejném okamžiku, kdy proměnná vznikne.

Podobně jako je možné definovat více proměnných najednou, je možné inicializovat více než jednu proměnnou zároveň. Například:

```
// vytvoření více proměnných typu long a jejich inicializace
long sirka = 5, delka = 7;
```

V tomto případě se inicializuje celočíselná proměnná `sirka` typu `long` na hodnotu `5` a celočíselná proměnná `delka` typu `long` na hodnotu `7`. Je dokonce možné kombinovat definice s inicializacemi:

```
int mujVek = 39, tvujVek, jehoVek = 40;
```

V tomto případě se vytvoří tři proměnné typu `int` a první a třetí se zároveň inicializují.

Výpis 3.2 ukazuje hotový program připravený ke kompilaci, který vypočítá plochu obdélníka a na obrazovku napíše výsledek.

Výpis 3.2

Ukázka použití proměnných

```
 0: // Ukázka použití proměnných
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:      using std::cout;
 6:      using std::endl;
 7:
 8:      unsigned short int Sirka = 5, Delka;
```

```
 9:     Delka = 10;
10:
11:     // vytvoření proměnné typu unsigned short a její inicializace
12:     // na výsledek násobení šířky délkou
13:     unsigned short int Plocha = (Sirka * Delka);
14:
15:     cout << "Sirka: " << Sirka << "\n";
16:     cout << "Delka: " << Delka << endl;
17:     cout << "Plocha: " << Plocha << endl;
18:     return 0;
19: }
```

Výstup

```
Sirka: 5
Delka: 10
Plocha: 50
```

Analýza Řádek 1 obsahuje příkaz `include` pro knihovnu `iostream` a objekt `cout` bude tedy fungovat. Na řádku 3 začíná program. Na řádcích 5 a 6 se definují objekty `cout` a `endl` jako součásti standardního oboru názvů (`std`).

Na řádku 8 se definuje proměnná `Sirka` jako celé číslo typu `unsigned short` a její hodnota se inicializuje na `5`. Další celé číslo typu `unsigned short`, `Delka`, se také definuje, ale neinicializuje. Na řádku 9 se proměnné `Delka` přiřadí hodnota `10`.

Na řádku 13 se definuje celočíselná proměnná typu `unsigned short`, `Plocha`, a inicializuje se na hodnotu, která se získá vynásobením proměnných `Sirka` a `Delka`. Na řádcích 15 až 17 se hodnoty všech proměnných vytisknou na obrazovku. Všimněte si, že speciální slovo `endl` vytvoří nový řádek.

Příkaz `typedef`

Opakované psaní `unsigned short int` může být únavné, a co je důležitější, náchylné k chybám. Jazyk C++ umožňuje pro celou tuto frázi vytvořit s pomocí klíčového slova `typedef` zkratku neboli alias. Slovo `typedef` pochází z anglického „type definition“ (definice typu).

Prakticky tak můžete vytvářet synonyma a je důležité, abyste tuto operaci rozlišovali od tvorby nových typů (což je téma šestého dne „Objektově orientované programování“). Za klíčové slovo `typedef` se píše stávající název typu a pak nový název. Výraz se ukončí středníkem. Příklad:

```
typedef unsigned short int USHORT;
```

Vzniká zde nový název typu `USHORT`, který budete moci použít všude tam, kde byste jinak museli psát `unsigned short int`. Výpis 3.3 je obdobou výpisu 3.2, ale místo `unsigned short int` se zde používá zkratka `USHORT`.

Výpis 3.3

Ukázka použití klíčového slova `typedef`

```
0: // ****************
1: // Ukázka použití klíčového slova typedef
2: #include <iostream>
```

```
 3:
 4: typedef unsigned short int USHORT;    // definice USHORT
 5:
 6: int main()
 7: {
 8:
 9:     using std::cout;
10:     using std::endl;
11:
12:     USHORT Sirka = 5;
13:     USHORT Delka;
14:     Delka = 10;
15:     USHORT Plocha = Sirka * Delka;
16:     cout << "Sirka: " << Sirka << "\n";
17:     cout << "Delka: " << Delka << endl;
18:     cout << "Plocha: " << Plocha << endl;
19:     return 0;
20: }
```

Výstup

```
Sirka: 5
Delka: 10
Plocha: 50
```

POZNÁMKA **Znak * označuje násobení.**

Analýza Na řádku 4 se definuje typ `USHORT` jako synonymum pro `unsigned short int`. Program i výstup je v podstatě stejný jako u výpisu 3.2.

Kdy používat `short` a kdy `long`

Někdy může být zdrojem nejasností nejistota, kdy proměnnou deklarovat jako typ `long` a kdy použít spíše typ `short`. Pravidlo, vykládá-li se správně, je poměrně přímočaré: Jestliže existuje sebemenší šance, že hodnota, kterou chcete do proměnné vložit, bude pro její typ příliš velká, použijte větší typ.

Jak ukazuje tabulka 3.1, proměnné typu `unsigned short` mohou nabývat celočíselných hodnot pouze do 65 535 (za předpokladu, že mají dva bajty). Proměnné typu `signed short` své hodnoty rozdělují mezi kladná a záporná čísla, a proto je zde maximální kladná hodnota, které mohou nabývat, pouze poloviční oproti typu `unsigned`.

Přestože celá čísla typu `unsigned long` mohou nabývat velmi vysokých hodnot (4 294 967 295), jedná se stále o konečné číslo. Budete-li potřebovat číslo větší, budete muset použít typ `float` nebo `double`, ale bude to vždy na úkor přesnosti. Typy `float` a `double` mohou nabývat extrémně vysokých hodnot, ale většina počítačů bere v úvahu pouze prvních 7 nebo 9 platných číslic. To znamená, že na tento počet číslic se hodnoty zaokrouhlují.

Kratší proměnné vyžadují méně paměti. V dnešní době je paměť levná a její životnost je krátká. Používejte bez obav typ `int`, který bude mít na vašem počítači pravděpodobně velikost čtyři bajty.

Přetečení celočíselného typu bez znaménka (unsigned)

U proměnných typu unsigned long existuje limit, kterého mohou maximálně dosáhnout jejich hodnoty. Jen zřídka se tento fakt stane problémem; k čemu však dojde v případě, že celé číslo tento limit překročí?

Když celé číslo typu unsigned dosáhne své maximální hodnoty, „přetočí" se zpět na nulu a počítání začíná znovu, podobně jako je tomu u tachometru v autě. Výpis 3.4 ilustruje, co se stane, když se pokusíte do celočíselné proměnné typu short vložit příliš velkou hodnotu.

Výpis 3.4

Ukázka vložení příliš velké hodnoty do celočíselné proměnné typu unsigned

```
 0: #include <iostream>
 1: int main()
 2: {
 3:
 4:      using std::cout;
 5:      using std::endl;
 6:
 7:      unsigned short int maleCislo;
 8:      maleCislo = 65535;
 9:      cout << "male cislo: " << maleCislo << endl;
10:     maleCislo++;
11:     cout << "male cislo: " << maleCislo << endl;
12:     maleCislo++;
13:     cout << "male cislo: " << maleCislo << endl;
14:     return 0;
15: }
```

Výstup

```
male cislo: 65535
male cislo: 0
male cislo: 1
```

Analýza Na řádku 7 se deklaruje maleCislo jako proměnná typu unsigned short int, což na typickém počítači znamená proměnnou o velikosti dvou bajtů, která může nabývat hodnot mezi 0 a 65 535. Na řádku 8 se do proměnné maleCislo přiřadí maximální možná hodnota a ta se vytiskne na řádku 9.

Na řádku 10 se hodnota proměnné maleCislo zvýší o jedničku. Symbol pro operaci, při které se hodnota proměnné zvýší o jedničku, je ++ (stejně jako C++ je vyšší verze jazyka než C). Hodnota v proměnné maleCislo by teď měla být 65 536. Celá čísla typu unsigned short však nemohou nabývat hodnot vyšších než 65 535. Hodnota se tedy vrátí zpět na 0, která se vytiskne na řádku 11.

Na řádku 12 se hodnota proměnné maleCislo znovu zvýší o jedničku a pak se tato nová hodnota 1 vytiskne.

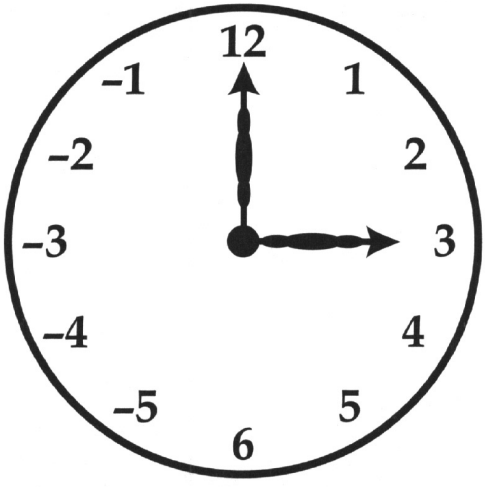
Přetečení celočíselného typu se znaménkem (signed)

U celých čísel typu signed se situace od celých čísel typu unsigned liší v tom, že polovina možných hodnot představuje záporná čísla. Místo tachometru si tentokrát raději představte hodiny, které se podobají těm z obrázku 3.2. Když se na nich pohybujeme ve směru hodi-

nových ručiček, čísla se zvyšují. Když jdeme proti směru hodinových ručiček, čísla se snižují. Čísla se pak střetnou ve spodní části hodin (v 6 hodin).

Obrázek 3.2

Kdyby se na hodinách používala čísla se znaménkem...



Čísla vedle 0 jsou buď 1 (ve směru hodinových ručiček), nebo –1 (proti směru hodinových ručiček). Když překročíme všechna kladná čísla, dostaneme se k největším záporným číslům a pak počítáme zpět k 0. Výpis 3.5 ukazuje, co se stane, když u proměnné typu `short integer` k maximálnímu kladnému číslu přičtete jedničku.

Výpis 3.5

Ukázka vložení příliš velké hodnoty do celočíselné proměnné typu `signed`

```
 0: #include <iostream>
 1: int main()
 2: {
 3:      using std::cout;
 4:      using std::endl;
 5:      short int maleCislo;
 6:      maleCislo = 32767;
 7:      cout << "male cislo: " << maleCislo << endl;
 8:      maleCislo++;
 9:      cout << "male cislo: " << maleCislo << endl;
10:      maleCislo++;
11:      cout << "male cislo: " << maleCislo << endl;
12:      return 0;
13: }
```

Výstup

```
male cislo: 32767
male cislo: -32768
male cislo: -32767
```

Analýza Na řádku 5 se deklaruje `maleCislo`, tentokrát jako proměnná typu `signed short` (jestliže výslovně neuvedete, že se jedná o `unsigned`, předpokládá se `signed`). Program pokračuje podobně jako předcházející, ale výstup se bude lišit. Chcete-li mu plně

porozumět, musíte být obeznámeni s tím, jak jsou čísla typu `signed` reprezentována jako bity v celém čísle o velikosti dvou bajtů.

Podstata však spočívá v tom, že stejně jako u typu celého čísla `unsigned`, i u celého čísla typu `signed` dochází k přetečení z nejvyšší možné kladné hodnoty na nejnižší možnou (tentokrát zápornou) hodnotu.

Znaky

Proměnná typu znak (typ `char`) má obvykle velikost 1 bajt, což je dost na to, aby pojala 256 hodnot (viz dodatek C). Proměnnou typu `char` můžeme interpretovat jako malé číslo (0 až 255) nebo jako prvek sady ASCII (American Standard Code for Information Interchange). Sada znaků ASCII a její obdoba ISO (International Standards Organization) nabízí způsob kódování všech písmen, číslic a interpunkčních znamének.

POZNÁMKA Počítače neznají písmena, věty nebo interpunkční znaménka. Jediné, čemu rozumí, jsou čísla. Ve skutečnosti to, co opravdu poznají, je, zda v konkrétním místě křížení spojů je dostatečné množství elektrické energie. Je-li tomu tak, bude tento stav symbolizovat 1, opačný stav pak symbolizuje 0. Podle seskupení jedniček a nul je počítač schopen generovat vzory, které je možné interpretovat jako čísla, a těm se pak mohou přiřadit písmena a interpunkční znaménka.

Podle kódu ASCII se hodnotě `97` přiřazuje malé písmeno „a“. Podobně jsou všechna malá a velká písmena, všechna čísla a interpunkční znaménka přiřazena hodnotám mezi 1 a 128. Zbývajících 128 značek a symbolů je vyhrazeno pro výrobce počítače, i když dnes se stala už téměř standardem rozšířená sada znaků od IBM.

POZNÁMKA Zkratka ASCII se obvykle vyslovuje „aski“.

Znaky a čísla

Když do proměnné typu `char` vložíte například hodnotu „a“, bude tam ve skutečnosti číslo mezi 0 a 255. Kompilátor však mezi sebou umí znaky (reprezentované apostrofem, pak písmenem, číslicí nebo interpunkčním znaménkem, následovaným opět apostrofem) a hodnoty sady ASCII tam i zpět překládat.

Funkce převádějící hodnoty na písmena a zpět může být libovolná. Neexistuje žádný zvláštní důvod, proč se malému písmenu „a“ přiřazuje právě hodnota 97. Pokud se na tom všichni (vaše klávesnice, kompilátor i obrazovka) shodnou, nedojde k žádným problémům. Je však nutné si uvědomit, že mezi hodnotou 5 a znakem „5“ existuje velký rozdíl. Skutečná hodnota znaku „5“ je 53, podobně jako písmeno „a“ je vyhodnoceno jako 97. Výpis 3.6 nám to ilustruje.

Výpis 3.6

Tisk znaků podle čísel

```
0: #include <iostream>
1: int main()
```

```
 2: {
 3:      for (int i = 32; i<128; i++)
 4:          std::cout << (char) i;
 5:      return 0;
 6: }
```

Výstup

```
 !"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuv
wyz{|}~_
```

Tento jednoduchý program vytiskne hodnoty znaků pro celá čísla od 32 do 127. Výpis využívá k dosažení tohoto efektu na řádku 3 celočíselnou proměnnou i. Na řádku 4 je číslo v proměnné i přinuceno k zobrazení ve formě znaku.

Bylo by možné použít také znakovou proměnnou, a to způsobem zachyceným ve výpisu 3.7, jehož výstup je totožný.

Výpis 3.7

Tisk znaků podle čísel, druhá varianta

```
 0: #include <iostream>
 1: int main()
 2: {
 3:    for (unsigned char i = 32; i<128; i++)
 4:      std::cout << i;
 5:    return 0;
 6: }
```

Jak vidíte, na řádku 3 je použit znak bez znaménka. Protože se místo číselné proměnné používá znaková proměnná, dokáže příkaz cout na řádku 4 přímo zobrazit znakovou hodnotu.

Speciální znaky

Kompilátor C++ rozeznává několik speciálních znaků určených k formátování. V tabulce 3.3 jsou uvedeny ty nejběžnější. Do kódu je zapíšete tak, že zadáte zpětné lomítko, za nímž bude následovat znak. Chcete-li například do kódu vložit znak tabulátoru, zadáte apostrof, zpětné lomítko, písmeno t a pak znovu apostrof:

```
char znakTab = '\t';
```

V tomto příkladě se deklaruje proměnná typu `char` (`znakTab`) a inicializuje se hodnotou znaku `\t`, který je interpretován jako tabulátor. Tyto speciální znaky pro formátování se používají při tisku na obrazovku, do souboru nebo na jiné výstupní zařízení.

Zpětné lomítko mění význam znaku, který za ním následuje. Například znak n normálně znamená písmeno n, ale když mu bude předcházet zpětné lomítko (\), bude se jednat o znak nového řádku.

Tabulka 3.3 Speciální znaky

Znak	Význam
`\a`	Pípnutí
`\b`	Smazání předchozího znaku (Backspace)

Znak	Význam
\f	Posun o stránku
\n	Nový řádek
\r	Návrat vozíku
\t	Tabulátor

Znak	Význam
\v	Vertikální tabulátor
\'	Apostrof
\"	Uvozovky
\?	Otazník
\\	Zpětné lomítko
\000	Oktalový (osmičkový) zápis
\xhhh	Hexadecimální (šestnáctkový) zápis

Konstanty

Podobně jako proměnné slouží i konstanty jako místa, kde se uchovávají data. Jak už sám název napovídá, konstanty na rozdíl od proměnných nemohou změnit hodnotu. Když definujete konstantu, musíte ji inicializovat a později jí už nemůžete přiřadit jinou hodnotu.

Literální konstanty

V jazyce C++ jsou dva typy konstant: literální a symbolické.

Literální konstantou rozumíme hodnotu, která se zadá přímo do programu. Například:

```
int mujVek = 39;
```

Proměnná `mujVek` je typu `int` a `39` je literální (doslovná) konstanta. Nemůžete 39 přiřadit hodnotu ani tuto hodnotu změnit.

Symbolická konstanta

Symbolickou konstantou rozumíme konstantu reprezentovanou názvem, podobně jako je názvem reprezentována proměnná. Na rozdíl od proměnné však nelze hodnotu konstanty po její inicializaci změnit.

Jestliže má program jednu celočíselnou proměnnou s názvem `studenti` a další s názvem `tridy`, můžete vypočítat, kolik máte při daném počtu tříd studentů; zde předpokládáme, že každá třída má 15 studentů.

```
studenti = tridy * 15;
```

V tomto případě je 15 literální konstantou. Váš kód by se snadněji četl a udržoval, kdybyste tuto hodnotu nahradili symbolickou konstantou:

```
studenti = tridy * studentiNaTridu;
```

Kdybyste se později rozhodli počet studentů v každé třídě změnit, stačilo by udělat to pouze v místě definice konstanty `studentiNaTridu`. Není nutné provádět jakékoli změny u žádného dalšího výskytu této konstanty. Symbolickou konstantu lze v C++ definovat dvěma

způsoby. Tradiční způsob, který je dnes už poněkud zastaralý, používá direktivu preprocesoru `#define`. Druhý a vhodnější způsob spočívá v aplikaci klíčového slova `const`.

Definice konstanty s pomocí `#define`

Protože direktivu preprocesoru `#define` používá mnoho již existujících programů, musíte dobře pochopit její skutečný způsob použití. Chcete-li konstantu definovat tradičním způsobem, zadejte:

```
#define studentiNaTridu 15
```

Všimněte si, že u konstanty `studentiNaTridu` není určen žádný typ (`int`, `char` a podobně). Příkaz `#define` znamená pouhou substituci textu. Pokaždé, když preprocesor narazí na slovo `studentiNaTridu`, vloží do textu místo něj hodnotu `15`.

Protože preprocesor běží před kompilátorem, kompilátor nikdy žádnou konstantu neuvidí, narazí pouze na číslo 15.

Pozor: I když se vám může zdát použití klíčového slova `#define` velmi jednoduché, měli byste se mu vyhýbat, protože je současný standard C++ považuje za již překonaný způsob deklarace.

Definice konstanty s pomocí `const`

Ačkoli konstanty definované s pomocí `#define` fungují dobře, existuje v C++ i lepší způsob:

```
const unsigned short int studentiNaTridu = 15;
```

V tomto příkladě se opět deklaruje symbolická konstanta s názvem `studentiNaTridu`, tentokrát se však jedná o konstantu typu `unsigned short int`. Výhodou této metody je snadnější údržba kódu a zlepšená prevence chyb. Největším rozdílem je však to, že konstanta má pevný typ a kompilátor si může vynutit, aby se používala v souladu se svým typem.

POZNÁMKA Konstanty není možné v době běhu programu měnit. Budete-li například chtít změnit konstantu `studentiNaTridu`, budete muset provést změnu v kódu a ten pak znovu zkompilovat.

ANO	NE
ANO, dávejte si pozor, aby velikost celých čísel nepřekročila dané meze a nevrátila se na nesprávnou hodnotu.	**NE**, nepoužívejte klíčová slova jako názvy proměnných.
ANO, dávejte svým proměnným smysluplné názvy, které vypovídají o jejich účelu.	**NE**, nepoužívejte direktivu preprocesoru `#define` k definování konstant. Pracujte raději s `const`.

Výčtové konstanty

Výčtové konstanty dovolují vytvořit nové typy a proměnné, jejichž hodnoty se omezují na množinu uvedených hodnot. Můžete například výčtem deklarovat typ `BARVA`, který bude nabývat následujících hodnot: `CERVENA`, `MODRA`, `ZELENA`, `BILA` a `CERNA`.

Pro syntaxi výčtových konstant platí, že začínají klíčovým slovem `enum`, za nímž následuje název typu, levá složená závorka, povolené hodnoty oddělené čárkou a nakonec pravá složená závorka se středníkem. Zde je příklad:

```
enum BARVA { CERVENA, MODRA, ZELENA, BILA, CERNA };
```

Tento příkaz způsobí:

1. Vytvoření nového výčtu `BARVA`, čímž dojde k založení nového typu.
2. Vytvoření symbolické konstanty `CERVENA` s hodnotou `0`, symbolické konstanty `BILA` s hodnotou `1`, symbolické konstanty `ZELENA` s hodnotou `2` a tak dále.

Každá výčtová konstanta má celočíselnou hodnotu. Jestliže neurčíte jinak, bude mít první konstanta hodnotu 0 a další vždy o jedničku větší. Každou z těchto konstant je však možné inicializovat na konkrétní hodnotu a těm, které inicializovány nejsou, se přiřadí hodnota podle předcházejících. Prohlédněte si tento příkaz:

```
enum BARVA { CERVENA=100, MODRA, ZELENA=500, BILA, CERNA=700 };
```

Konstanta `CERVENA` bude mít hodnotu `100`, `MODRA` bude mít hodnotu `101`, `ZELENA` bude mít hodnotu `500`, `BILA` hodnotu `501` a nakonec `CERNA 700`.

Nyní můžete definovat proměnné typu `BARVA`, ale bude možné jim přiřadit pouze jednu z hodnot výčtu (v tomto případě `CERVENA`, `MODRA`, `ZELENA`, `BILA` nebo `CERNA`, čili `100`, `101`, `500`, `501` nebo `700`). Proměnné typu `BARVA` tedy můžete přiřadit libovolnou hodnotu barvy. Ve skutečnosti jí můžete přiřadit libovolnou celočíselnou hodnotu, i když se nebude jednat o povolenou barvu, ačkoli dobrý kompilátor vás na to upozorní. Je důležité vědět, že výčtové proměnné mají ve skutečnosti typ `unsigned int` a že výčtové konstanty jsou vlastně celá čísla. Je však velmi praktické, když je možné při práci s barvami, dny v týdnu a dalšími typy jejich hodnoty pojmenovávat. Ve výpisu 3.8 najdete program, který používá výčtový typ.

Výpis 3.8

Ukázka výčtových konstant

```
 0: #include <iostream>
 1: int main()
 2: {
 3:     enum Dny { Pondeli, Utery, Streda,
 4:         Ctvrtek, Patek, Sobota, Nedele };
 5:
 6:     Dny dnes;
 7:     dnes = Pondeli;
 8:
 9:     if ( dnes == Sobota || dnes == Nedele )
10:         std::cout << "\nMiluji vikendove dny!\n";
11:     else
12:         std::cout << "\nVzhuru do prace.\n";
13:
14:     return 0;
15: }
```

Výstup

```
Vzhuru do prace.
```

Analýza Na řádcích 3 a 4 se s pomocí sedmi hodnot definuje výčtový typ `Dny`. Každá jeho konstanta má celočíselnou hodnotu, kterou počítáme od nuly nahoru; `Utery` má tedy hodnotu 1.

Vytvoříme nyní proměnnou typu `Dny`, která bude uchovávat jednu z platných hodnot ze seznamu uvedených konstant. Této proměnné pak přiřadíme na řádku 7 výčtovou hodnotu `Pondeli` a tuto hodnotu podrobíme na řádku 9 testu.

Výčtový typ z tohoto programu by mohl být nahrazen řadou celočíselných konstant, jak ilustruje výpis 3.9.

Výpis 3.9

Stejný program, který používá řadu celočíselných konstant

```
 0: #include <iostream>
 1: int main()
 2: {
 3:      const int Pondeli = 0;
 4:      const int Utery = 1;
 5:      const int Streda = 2;
 6:      const int Ctvrtek = 3;
 7:      const int Patek = 4;
 8:      const int Sobota = 5;
 9:      const int Nedele = 6;
10:
11:      int dnes;
12:      dnes = Pondeli;
13:
14:      if ( dnes == Sobota || dnes == Nedele )
15:           std::cout << "\nMiluji vikendove dny!\n";
16:      else
17:           std::cout << "\nVzhuru do prace.\n";
18:
19:      return 0;
20: }
```

Výstup

```
Vzhuru do prace.
```

Analýza Výstup bude u výpisu 3.9 identický s výstupem programu 3.8. V programu 3.9 se každá z konstant explicitně definuje (`Pondeli`, `Utery` atd.) a žádný výčtový typ `Dny` zde nenajdeme. Výčtové konstanty mají tu výhodu, že dokumentují samy sebe, čímž je záměr výčtového typu `Dny` okamžitě zřejmý.

Pozor **Řada proměnných deklarovaných v tomto programu se nepoužívá. Proto může při jeho překladu kompilátor zobrazit varování.**

Souhrn

V této kapitole jsme se seznámili s číselnými a znakovými proměnnými a konstantami, které se v C++ používají k uchovávání dat během provádění programu. Číselné proměnné jsou buď celočíselné (`char`, `short` a `long int`), nebo s plovoucí řádovou čárkou (`float` a `double`). U celočíselných proměnných dále rozeznáváme typy `signed` a `unsigned`, tedy se znaménkem a bez znaménka. Přestože se velikosti jednotlivých typů mohou od sebe na různých počítačích lišit, má typ u daného počítače přesně specifikovanou velikost.

Dříve než je možné proměnnou použít, je třeba ji deklarovat a pak do ní uložit správná data toho typu, se kterým byla tato proměnná deklarována. Jestliže do celočíselné proměnné vložíte příliš velké číslo, dojde k jejímu „přetečení“, takže její hodnota se vrátí na nejnižší možnou hodnotu a jako důsledek získáte nesprávné výsledky.

V této kapitole jsme také zavedli pojmy literální, symbolické a v neposlední řadě i výčtové konstanty. Ukázali jsme dva způsoby, kterými lze symbolickou konstantu deklarovat: s pomocí příkazu preprocesoru `#define` a klíčového slova `const`.

Otázky a odpovědi

Otázka: **Jestliže u proměnné typu `short int` může dojít v paměti k přetečení a následnému vrácení hodnoty zpět, proč se vždy nepoužívají celá čísla typu `long`?**

Odpověď: K přetečení v paměti dochází jak u proměnných typu `short`, tak u proměnných typu `long`, ale u posledně uvedených k tomu dochází až v případě mnohem větších čísel. Například proměnná typu `unsigned short int` se vrátí zpět na nulu po dosažení hodnoty 65 535, zatímco u proměnné typu `unsigned long int` k tomu dojde až při hodnotě 4 294 967 295. U většiny počítačů zabírá každá celočíselná proměnná typu `long` dvojnásobné množství paměti oproti typu `short` (čtyři bajty oproti dvěma bajtům). Program se stem takových proměnných spotřebuje o dvě stě bajtů paměti RAM navíc. Upřímně řečeno, dnes se jedná o mnohem menší problém než dříve, protože většina osobních počítačů má miliony (ne-li miliardy) bajtů paměti. Používání větších než nutných typů může rovněž vyžadovat další čas vašeho procesoru.

Otázka: **Co se stane, když desetinné číslo přiřadím do celočíselné proměnné místo do proměnné typu `float`? Představte si následující řádek kódu:**

```
int aCislo = 5.4;
```

Odpověď: Dobrý kompilátor vás upozorní, ale přiřazení je zcela v pořádku. Číslo, které jste přiřadili, bude zarovnáno na celočíselnou hodnotu. Jestliže celočíselné proměnné přiřadíte hodnotu 5,4, bude hodnota této proměnné nadále 5. Dojde však k nenávratné ztrátě informace, a jestliže se později pokusíte přiřadit hodnotu z této celočíselné proměnné do proměnné typu `float`, bude výsledkem pouze hodnota 5,0.

Otázka: **Proč se nepoužívají literální konstanty? Proč se mám zatěžovat s používáním symbolických konstant?**

Odpověď: Jestliže jistou hodnotu používáte na mnoha místech programu, umožní vám symbolická konstanta změnit všechny tyto výskyty najednou pouhou změnou definice konstanty. Symbolické konstanty také mluví samy za sebe. Může být jen obtížně pochopitelné, proč se číslo násobí hodnotou 360; mnohem srozumitelnější je, když na tomto místě figuruje konstanta `stupnuVKruznici`.

Otázka: **Co se stane, když do proměnné typu `unsigned` přiřadím zápornou hodnotu? Představte si následující řádek kódu:**

```
unsigned int aKladneCislo = -1;
```

Odpověď: Dobrý kompilátor vás upozorní, ale jedná se o povolené přiřazení. K zápornému číslu se bude přistupovat jako k jeho bitové reprezentaci a ta se uloží do proměnné. Hodnota této proměnné se bude interpretovat jako číslo typu `unsigned`. Číslu -1 odpovídá bitová reprezentace 11111111 11111111 (hexadecimálně 0xFF); takové číslo je u typu `unsigned` reprezentováno hodnotou 65 535.

Otázka: **Mohu pracovat s C++, aniž bych musel rozumět bitovým reprezentacím, binární a hexadecimální aritmetice?**

Odpověď: Samozřejmě ano, ale nevytěžíte z jazyka tolik, jako kdybyste těmto otázkám rozuměli. Jazyk C++ vás na rozdíl od jiných programovacích jazyků příliš neochrání před tím, co počítač doopravdy dělá. Ve skutečnosti je to výhoda, protože tím získáte obrovskou sílu, kterou jiné jazyky nenabízí. Ale jako u každého silného nástroje, chcete-li z něj dostat maximum, musíte rozumět tomu, jak funguje. Programátoři, kteří se snaží programovat v C++ bez pochopení základů binárního systému, jsou často zmateni výsledky, které dostávají.

Úkoly pro vás

Testovací otázky v této části vám pomohou upevnit znalosti, které jste v této kapitole získali. Cvičení slouží k tomu, aby vám nabídlo praktickou zkušenost s tím, co jste se naučili. Pokuste se na otázky v testu a cvičení odpovědět dříve, než se podíváte do klíče v dodatku D. Než přejdete k další kapitole, ujistěte se, že každé odpovědi rozumíte.

Test

1. Jaký je rozdíl mezi celočíselnou proměnnou a proměnnou s plovoucí (pohyblivou) řádovou čárkou?
2. Jaký je rozdíl mezi proměnnými typu `unsigned short int` a `unsigned long int`?
3. Jaké výhody má používání symbolické konstanty místo literální konstanty?
4. Jaké výhody má používání klíčového slova `const` místo `#define`?
5. Podle čeho posuzujeme, zda je název proměnné vhodný nebo nevhodný?
6. U daného výrazu `enum` rozhodněte, jaká je skutečná hodnota konstanty `MODRA`.

```
enum BARVA {BILA, CERNA=100, CERVENA, MODRA, ZELENA=300} ;
```

7. Které z následujících názvů proměnných považujete za vhodné, které za nevhodné a které jsou neplatné?
   a/ `Vek`
   b/ `!ex`
   c/ `R79J`
   d/ `CelkovyPrijem`
   e/ `_Neplatne`

Cvičení

1. Jaký je správný typ proměnné pro uložení následujících informací?
   a/ Váš věk.
   b/ Výměra vaší zahrady.
   c/ Počet hvězd v galaxii.
   d/ Průměrné srážky v měsíci lednu.
2. Vytvořte k výše uvedeným informacím vhodné názvy proměnných.
3. Deklarujte konstantu pro Ludolfovo číslo (pí) jako 3,14159 (pozor na desetinný oddělovač).
4. Deklarujte proměnnou typu `float` a inicializujte ji s pomocí vaší konstanty pí.

den 4

Výrazy a příkazy

Ve své podstatě je program sadou příkazů, které se mají provést v určitém pořadí. Síla programu spočívá v možnosti provést jednu nebo druhou sadu příkazů podle toho, je-li konkrétní podmínka splněna či ne.

Dnes se naučíte a dozvíte:

- Co jsou to příkazy.
- Co jsou to bloky.
- Co jsou to výrazy.
- Jak se dá kód rozvětvit v závislosti na podmínce.
- Co je pravdivost a jak s ní zacházet.

Příkazy

Příkazy v C++ dovolují řídit provádění kódu, vyhodnocovat výrazy a nebo nedělat vůbec nic (prázdný příkaz). Všechny příkazy v C++ jsou ukončeny středníkem, dokonce i prázdný příkaz, což je vlastně středník a nic jiného. Jeden z nejběžnějších příkazů, příkaz přiřazení, má následující podobu:

```
x = a + b;
```

Na rozdíl od algebraického výrazu tento příkaz neznamená, že `x` se rovná `a+b`. Čteme jej: „Proměnné x přiřaď hodnotu součtu a a b“ nebo „Přiřaď a+b do x“ nebo „Nastav x rovno a plus b“. I když se v tomto příkazu provedou dvě věci, jedná se o jeden příkaz, a proto je zde pouze jeden středník.

Poznámka **Příkaz přiřazení uloží do toho, co je na levé straně rovnítka, hodnotu výrazu, který je na pravé straně.**

Prázdné znaky

Prázdné znaky (neboli tabulátory, mezery a nové řádky) se obecně ve výrazech ignorují. Výše uvedený příkaz přiřazení bychom mohli zapsat takto:

```
x=a+b;
```

nebo takto:

```
x                               = a
+           b                 ;
```

Ačkoli i poslední možnost je povolená, je zcela nesmyslná. Prázdné znaky lze použít k lepší čitelnosti a snadnější údržbě kódu, stejně jako k napsání kódu, ve kterém se nikdo nevyzná. Jazyk C++ vám nabízí veškerou svou sílu, ale vy musíte použít vlastní úsudek.

Prázdné (bílé) znaky nejsou vidět. Jestliže se tyto znaky vytisknou, uvidíte pouze bílou barvu papíru.

Bloky a složené příkazy

Všude tam, kde můžete použít jeden příkaz, je možné vložit i složený příkaz, který se označuje pojmem blok. Blok začíná levou složenou závorkou ({) a končí pravou složenou závorkou (}). Ačkoli každý příkaz bloku musí být ukončen středníkem, blok sám se středníkem neukončuje. Prohlédněte si následující příklad:

```
{
      pom = a;
      a = b;
      b = pom;
}
```

Tento blok se provede jako jeden příkaz, přičemž se v něm vymění hodnoty v proměnných `a` a `b`.

ANO	NE
ANO, ukončete každý výraz středníkem.	**NE**, nezapomeňte na uzavírací složenou závorku v případě použití otevírací složené závorky.
ANO, používejte prázdné znaky s rozmyslem, aby byl kód srozumitelnější.	

Výrazy

Vše, co se vyhodnotí na nějakou hodnotu, se v C++ označuje pojmem výraz. Říkáme, že výraz *vrací* hodnotu. Například `3+2` vrací hodnotu `5`, proto se jedná o výraz. Všechny výrazy jsou zároveň příkazy.

Množství úseků kódu, které lze označit jako výraz, vás možná překvapí. Uvedeme tři příklady:

```
3.2                     // vrací hodnotu 3,2
PI                      // konstanta typu float, která vrací hodnotu 3,14
SekundZaMinutu          // konstanta typu int, která vrací hodnotu 60
```

Předpokládáme, že `PI` je konstanta, která byla dříve vytvořena a inicializována na hodnotu 3,14, a `SekundZaMinutu` je konstanta, jejíž hodnota je rovna 60. Za těchto okolností jsou na všech třech řádcích správné výrazy.

Trochu komplikovanější je výraz:

```
x = a + b;
```

V něm se nejen sečtou hodnoty `a` a `b` a výsledek tohoto součtu se přiřadí proměnné `x`, ale vrátí se také hodnota tohoto přiřazení (hodnota `x`). Protože se jedná o výraz, může být sám na pravé straně operátoru přiřazení:

```
y = x = a + b;
```

Tento řádek se vyhodnotí v následujícím pořadí:

Sečte se hodnota proměnných `a` a `b`.

Výsledek výrazu `a + b` se přiřadí proměnné `x`.

Výsledek výrazu přiřazení `x = a + b` se přiřadí proměnné `y`.

Jestliže jsou proměnné `a`, `b`, `x` a `y` všechny typu `int` a jestliže `a` má hodnotu `2` a `b` má hodnotu `5`, bude oběma proměnným `x` i `y` přiřazena hodnota `7`. Tento příklad je ilustrován v programu ve výpisu 4.1.

Výpis 4.1

Vyhodnocování složitých výrazů

```
 0: #include <iostream>
 1: int main()
 2: {
 3:     using std::cout;
 4:     using std::endl;
 5:
 6:     int a=0, b=0, x=0, y=35;
 7:     cout << "a: " << a << " b: " << b;
 8:     cout << " x: " << x << " y: " << y << endl;
 9:     a = 9;
10:     b = 7;
11:     y = x = a+b;
12:     cout << "a: " << a << " b: " << b;
13:     cout << " x: " << x << " y: " << y << endl;
14:     return 0;
15: }
```

Výstup

```
a: 0 b: 0 x: 0 y: 35
a: 9 b: 7 x: 16 y: 16
```

Analýza Na řádku 6 se inicializují a deklarují čtyři proměnné. Na řádcích 7 a 8 se vytisknou jejich hodnoty. Na řádku 9 se proměnné `a` přiřadí hodnota 9. Na řádku 10 se proměnné `b` přiřadí hodnota 7. Na řádku 11 se sečtou hodnoty proměnných `a` a `b` a výsledek tohoto součtu se přiřadí proměnné `x`. Tento výraz (`x = a + b`) se vyhodnotí na hodnotu (součet `a + b`) a tato hodnota se pak přiřadí proměnné `y`.

Operátory

Operátor je symbol, který přiměje kompilátor, aby provedl nějakou akci. Operátory pracují nad operandy a všechny operandy v C++ jsou výrazy. V jazyce C++ existuje několik kategorií operátorů. Dvě z nich jsou:

- Operátory přiřazení
- Matematické operátory

Operátor přiřazení

Operátor přiřazení způsobí, že operand na levé straně operátoru změní svou hodnotu na hodnotu pravé strany tohoto operátoru. Výraz

```
x = a + b;
```

přiřadí operandu `x` hodnotu, která je výsledkem součtu proměnných `a` a `b`.

L-hodnoty a p-hodnoty

Operand, který může být umístěn na levou stranu operátoru přiřazení, se nazývá l-hodnota. Operátor, který se může vyskytovat na pravé straně operátoru přiřazení, se nazývá (ano, uhádli jste) p-hodnota (nebo také r-hodnota).

Konstanty jsou p-hodnoty, které nemohou být l-hodnotami. Můžete tedy napsat

```
x = 35;                    // v pořádku
```

ale nemůžete napsat

```
35 = x;                    // chyba, nejedná se o l-hodnotu!
```

L-hodnota je operand, který může být na levé straně výrazu. P-hodnota je operand, který může být na pravé straně výrazu. Všimněte si, že všechny l-hodnoty jsou p-hodnotami, ale ne všechny p-hodnoty jsou l-hodnotami. Příklad p-hodnoty, která není l-hodnotou, je prostý. Můžete napsat `x = 5;`, ale nemůžete napsat `5 = x;` (`x` může být l-hodnotou i p-hodnotou, ale `5` může být pouze p-hodnotou).

Poznámka Konstanty jsou p-hodnoty. Jelikož jejich hodnotu nelze změnit, nemohou se nacházet na levé straně operátoru přiřazení, takže nemohou být l-hodnotami.

Matematické operátory

K pěti matematickým operátorům patří sčítání (`+`), odečítání (`-`), násobení (`*`), dělení (`/`) a dělení se zbytkem (`%`).

Sčítání a odečítání funguje tak, jak ho znáte: Dvě čísla oddělená znakem plus nebo minus se sečtou, resp. odečtou. Násobení funguje stejně, jen jeho operátorem je hvězdička (*). Dělení se provádí lomítkem (/). Dále jsou uvedeny příklady výrazů využívajících jednotlivé zmíněné operátory. Výsledek se vždy přiřadí proměnné vysledek. Komentáře vpravo ukazují právě aktuální hodnotu výsledku.

```
vysledek = 56 + 32   // vysledek = 88
vysledek = 12 + 10   // vysledek = 2
vysledek = 21 / 7    // vysledek = 3
vysledek = 12 * 4    // vysledek = 48
```

Potíže s odečítáním

Odečítání u celočíselných proměnných typu `unsigned` (bez znaménka) může vést k překvapivým výsledkům, pokud je výsledkem operace záporné číslo. S něčím podobným jste se již setkali v minulé kapitole, když jste viděli, jak může dojít k přetečení proměnné. Výpis 4.2 nám ukazuje, co se stane, když odečtete velké číslo bez znaménka od malého čísla bez znaménka.

Výpis 4.2

Ukázka přetečení celočíselné proměnné při odečítání

```
 0: // Výpis 4.2 - demonstruje odečítání a
 1: // přetečení celočíselné proměnné
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:     using std::cout;
 7:     using std::endl;
 8:
 9:     unsigned int rozdil;
10:     unsigned int velkeCislo = 100;
11:     unsigned int maleCislo = 50;
12:     rozdil = velkeCislo - maleCislo;
13:     cout << "Rozdil je: " << rozdil;
14:     rozdil = maleCislo - velkeCislo;
15:     cout << "\nRozdil nyni: " << rozdil <<endl;
16:     return 0;
17: }
```

Výstup

```
Rozdil je: 50
Rozdil nyni: 4294967246
```

Analýza Na řádku 12 je zavolán operátor odečítání a výsledek vytištěný na řádku 13 je takový, jaký bychom asi čekali. Na řádku 14 se znovu volá operátor odečítání, ale tentokrát se větší číslo typu `unsigned` odečte od menšího čísla typu `unsigned`. Výsledek by měl být záporný, ale protože se vyhodnocuje (a vytiskne) jako číslo typu `unsigned`, tedy bez znaménka, výsledkem je přetečení, které jsme popsali včera. Touto problematikou se znovu zabýváme v Dodatku C – „Přednost operátorů“.

Celočíselné dělení a dělení se zbytkem

S celočíselným dělením jste se setkali už ve druhé třídě, když jste měli číslo 21 dělit číslem 4 (21/4) a vy jste odpověděli 5 (se zbytkem).

Operátor dělení se zbytkem vrací zbytek po celočíselném dělení. Chcete-li spočítat, jaký je zbytek po dělení čísla 21 číslem 4 (21%4), získáte výsledek 1.

Zbytek po celočíselném dělení se vám může hodit v mnoha situacích. Můžete například chtít vytisknout při každé desáté operaci nějakou větu. Čísla, která dávají při celočíselném dělení 10 zbytek 0, jsou právě násobky deseti. Tedy 1 % 10 je 1, 2 % 10 je 2 atd., dokud se nedojde k 10 % 10, kde je výsledkem 0. 11 % 10 je znovu 1, a tak můžeme pokračovat k dalšímu násobku 10, což je 20. Znovu pak platí, že 20 % 10 = 0. K této metodě se ještě vrátíme v den 7. – „Více o toku programu“, kde se budeme věnovat smyčkám.

Často kladené otázky

Při dělení 5/3 dostávám výsledek 1. Co není v pořádku?

Odpověď: Když dělíte jedno celé číslo jiným, dostanete jako výsledek celé číslo. Proto 5/3 je 1. (Skutečná odpověď by zněla 1 se zbytkem 2. Abyste dostali zbytek, zkuste 5%3, což jsou 2.)

Chcete-li dostat výsledek ve tvaru desetinného čísla, musíte používat proměnné typu float. 5.0/3.0 vám vrátí 1,66667.

Je-li buď čitatel nebo jmenovatel číslem s pohyblivou řádovou čárkou, vygeneruje kompilátor podíl, který bude číslem s pohyblivou řádovou (desetinnou) čárkou.

Kombinace operátoru přiřazení s matematickými operátory

Není neobvyklé, když se k hodnotě proměnné přidá jiná hodnota a výsledek se pak přiřadí zpět do původní proměnné. Máte-li proměnnou `mujVek` a chcete-li zvýšit její hodnotu o 2, můžete napsat:

```
int mujVek = 5;
int zvys;
zvys = mujVek + 2;   // součet 5 + 2 a uložení do zvys
mujVek = zvys;       // vložení zpět do mujVek
```

Jedná se však o značně komplikovanou a neúspornou metodu. V C++ je možné dát stejnou proměnnou na obě strany operátoru přiřazení a v našem případě pak dojde k následující změně:

```
mujVek = mujVek + 2;
```

To je mnohem lepší. V algebře by byl tento výraz považován za nesmyslný, ale v C++ se bude číst jako „k hodnotě proměnné `mujVek` přičti dvě a výsledek přiřaď do proměnné `mujVek`“.

Existuje dokonce ještě jednodušší forma zápisu, která se však tak dobře nečte:

```
mujVek += 2;
```

Operátor (+=) sám přičte p-hodnotu k l-hodnotě a pak výsledek znovu přiřadí do l-hodnoty. Název tohoto operátoru se vyslovuje „plus-rovná-se“. Příkaz by se pak četl „mujVek plus-

rovná-se dvě“. Kdyby byla hodnota proměnné `mujVek` rovna 4, pak bychom po provedení této operace dostali výsledek 6.
Existují také obdoby této operace pro odečítání (-=), dělení (/=), násobení (*=) a dělení se zbytkem (%=).

Inkrementace a dekrementace

Nejběžnější hodnotou, která se přičítá nebo odečítá a pak znovu přiřazuje proměnným, je číslo 1. V C++ se zvýšení hodnoty o 1 nazývá inkrementace a snížení pak dekrementace. Tyto operace se mohou provádět s pomocí zvláštních operátorů.

Operátor inkrementace (++) zvýší hodnotu proměnné o 1 a operátor dekrementace (--) sníží hodnotu proměnné o 1. Proto máte-li proměnnou C a chcete její hodnotu zvýšit o 1, mohli byste to provést s pomocí následujícího příkazu:

```
C++;                          // vezme proměnnou C a zvýší její hodnotu o 1
```

Tento příkaz je ekvivalentní s výmluvnějším příkazem

```
C = C + 1;
```

nebo také s obměnou tohoto příkazu:

```
C += 1;
```

POZNÁMKA Jak už jste asi sami uhodli, jméno jazyka C++ je odvozeno od operátoru inkrementace a od názvu jeho předchůdce, jazyka C. Vychází se z toho, že možnosti jazyka C++ jsou zlepšením oproti C.

Prefix a postfix

Operátory inkrementace (++) a dekrementace (--) mají dvě formy: prefixovou a postfixovou. Varianta prefixová se píše před názvem proměnné (`++mujVek`), zatímco varianta postfixová se píše za název proměnné (`mujVek++`).

U jednoduchého příkazu nebude příliš záležet na tom, kterou z nich použijete. V případě složitého příkazu, kdy budete zvyšovat (nebo snižovat) hodnotu proměnné o 1 a pak budete chtít výsledek přiřadit do další proměnné, na tom záleží velmi. Prefixový operátor se vyhodnocuje před přiřazením, zatímco postfixový operátor se vyhodnocuje až po něm.

Prefixová sémantika je následující: Zvýší se hodnota o 1 a pak se vrátí. Postfixová sémantika je jiná: Vrátí se nejprve hodnota a teprve pak se hodnota proměnné zvýší o 1.

Na první pohled se to může zdát trochu matoucí, ale je-li `x` celočíselná proměnná, jejíž hodnota je 5, a napíšete

```
int a = ++x;
```

tak kompilátoru řeknete, aby hodnotu proměnné `x` zvýšil o 1 (na 6) a pak tuto hodnotu vzal a přiřadil ji proměnné `a`. Proto `a` je teď 6 a `x` je teď také 6.

Jestliže pak napíšete

```
int b = x++;
```

řeknete kompilátoru, aby vzal hodnotu v proměnné x (6), přiřadil ji proměnné b, pak se vrátil zpět a zvýšil hodnotu x o jedničku. Proto proměnná b je teď 6, ale x je teď 7. Výpis 4.3 ilustruje, jak se oba typy používají a jaké jsou důsledky obou operací.

Výpis 4.3

Ukázka používání operátorů inkrementace a dekrementace

```
 0:  // Výpis 4.3 - demonstruje používání
 1:  // operátoru inkrementace a dekrementace
 2:  // (zvýšení a snížení hodnoty o 1)
 3:  // s prefixovým a postfixovým zápisem
 4:  #include <iostream>
 5:  int main()
 6:  {
 7:      using std::cout;
 8:
 9:      int mujVek = 39;       // inicializace dvou celočíselných proměnných
10:      int tvujVek = 39;
11:      cout << "Mam " << mujVek << " let.\n";
12:      cout << "Mas " << tvujVek << " let.\n";
13:      mujVek++;          // postfixová inkrementace
14:      ++tvujVek;         // prefixová inkrementace
15:      cout << "Uplynul jeden rok...\n";
16:      cout << "Mam " << mujVek << " let.\n";
17:      cout << "Mas " << tvujVek << " let.\n";
18:      cout << "Uplynul dalsi rok...\n";
19:      cout << "Mam " << mujVek++ << " let.\n";
20:      cout << "Mas " << ++tvujVek << " let.\n";
21:      cout << "Vytiskneme to znovu.\n";
22:      cout << "Mam " << mujVek << " let.\n";
23:      cout << "Mas " << tvujVek << " let.\n";
24:      return 0;
25:  }
```

Výstup

```
Mam 39 let.
Mas 39 let.
Uplynul jeden rok...
Mam 40 let.
Mas 40 let.
Uplynul dalsi rok...
Mam 40 let.
Mas 41 let.
Vytiskneme to znovu.
Mam 41 let.
Mas 41 let.
```

Analýza Na řádcích 9 a 10 jsou deklarovány dvě celočíselné proměnné a každá z nich je inicializována hodnotou 39. Jejich hodnoty se vytisknou na řádcích 11 a 12.

Na řádku 13 se hodnota proměnné mujVek zvýší o 1 prostřednictvím operátoru postfixové inkrementace a na řádku 14 se zvýší hodnota proměnné tvujVek o 1 prostřednictvím operátoru prefixové inkrementace. Na řádcích 16 a 17 se vytisknou výsledky a budou oba identické (40).

Na řádku 19 bude součástí příkazu k tisku zvýšení hodnoty proměnné `mujVek` o jedničku prostřednictvím operátoru postfixové inkrementace. Protože se jedná o postfixovou inkrementaci, dojde k ní až po vytištění proměnné, a proto se znovu vytiskne hodnota 40. Naproti tomu na řádku 20 se zvýší o jedničku hodnota proměnné `tvujVek`, a to prostřednictvím prefixové inkrementace. Proto se hodnota proměnné nejdříve zvýší a pak se teprve vytiskne. Na obrazovce ji uvidíme jako 41.

Na řádcích 22 a 23 se hodnoty vytisknou znovu. Protože se příkaz inkrementace dokončil, je hodnota proměnné `mujVek` nyní 41, stejně jako hodnota proměnné `tvujVek`.

Priorita operátorů

U složitého příkazu, jakým je například tento:

```
x = 5 + 3 * 8;
```

může vyvstat otázka, v jakém pořadí se budou jednotlivé operace provádět. Jestliže se provede nejdříve operace sčítání, bude odpověď 8 * 8, čili 64. Jestliže se dá přednost operaci násobení, odpověď bude 5 + 24, tedy 29.

Každý operátor má nějakou váhu, s jakou se mu dává před jinými operátory přednost. Úplný seznam najdete v dodatku C.

Násobení má přednost před sčítáním, proto bude mít výraz hodnotu 29.

Když mají dva matematické operátory stejnou váhu, provedou se v pořadí zleva doprava. Proto se u výrazu

```
x = 5 + 3 + 8 * 9 + 6 *4;
```

nejdříve vyhodnotí násobení zleva doprava. Tedy 8*9 = 72 a 6*4 = 24. Výraz vypadá nyní takto:

```
x = 5 + 3 + 72 + 24;
```

Nyní pro sčítání zleva doprava platí 5 + 3 = 8; 8 + 72 = 80; 80 + 24 = 104.

Buďte však opatrní, protože u některých operátorů, například u přiřazení, se vyhodnocení provádí zprava doleva!

Co se však stane v případě, že pořadí operací nesplňuje vaše potřeby? Vezměme v úvahu tento výraz:

```
CelkemSekund = PocetMinutMysleni + PocetMinutPsani * 60;
```

U tohoto výrazu nechceme, aby se proměnná `PocetMinutPsani` vynásobila číslem 60 a pak přičetla k proměnné `PocetMinutMysleni`. Chceme, aby se tyto dvě proměnné nejdříve sečetly, čímž dostaneme celkový počet minut, a ten pak chceme vynásobit číslem 60, abychom dostali celkový počet sekund.

V tomto případě ke změně pořadí v provádění operací použijeme závorky. Položky v závorkách mají při vyhodnocování vyšší prioritu než všechny ostatní matematické operátory. Proto s použitím závorek přepíšeme výraz takto:

```
CelkemSekund = (PocetMinutMysleni + PocetMinutPsani) * 60;
```

a dostaneme požadovaný výsledek.

Vnořené závorky

U složitých výrazů budete možná potřebovat vnořit jedny závorky do druhých. Například při stanovení počtu sekund se nejdříve určí celkový počet lidí, kteří se akce účastní, a ten se pak vynásobí počtem sekund:

```
CelkemClovekoSekund = ( ( (PocetMinutMysleni + PocetMinutPsani)  *  60) * (Li-
diVKancelari + LidiNaDovolene))
```

Tento složitý výraz se čte zevnitř ven. Nejdříve se sečtou proměnné `PocetMinutMysleni` a `PocetMinutPsani`, protože výraz je v nejvnitřnějších závorkách. Výsledek se pak vynásobí 60. Pak se sečtou proměnné `LidiVKancelari` a `LidiNaDovolene`. Nakonec se celkový počet lidí vynásobí celkovým počtem sekund.

Tento příklad ilustruje velmi důležitý fakt. Počítač tento výraz přečte velmi snadno, ale lidé ho budou číst, rozumět mu či měnit ho velmi obtížně. Stejný výraz můžeme s pomocí pomocných celočíselných proměnných přepsat následujícím způsobem:

```
CelkemMinut = PocetMinutMysleni + PocetMinutPsani;
CelkemSekund = CelkemMinut * 60;
CelkemLidi = LidiVkancelari + LidiNaDovolene;
CelkemClovekoSekundy = CelkemLidi * CelkemSekund;
```

Tento výraz budeme o něco déle psát a použijeme při tom více dočasných proměnných než v předcházejícím příkladě, ale odměnou nám bude jeho lepší srozumitelnost. Když na začátek kódu přidáme komentář popisující, co se v něm děje, a změníme 60 na symbolickou konstantu, dostaneme kód, který se bude snadno udržovat a každému bude srozumitelný.

ANO	NE
ANO, pamatujte si, že výrazy mají hodnotu. **ANO**, používejte prefixový operátor (`++promenna`) v případě, že chcete hodnotu této proměnné zvýšit či snížit o 1 před tím, než se má proměnná použít ve výrazu. **ANO**, používejte postfixový operátor (`promenna++`) v případě, že chcete hodnotu proměnné zvýšit nebo snížit poté, co se použije ve výrazu. **ANO**, používejte závorky, když chcete změnit pořadí provádění operací.	**NE**, nesnažte se výrazy vnořovat příliš hluboko, protože se v nich ztrácí snadná orientace a špatně se udržují. **NE**, nezaměňujte postfixový operátor s prefixovým operátorem.

Povaha pravdivosti

V předcházejících verzích C++ byla pravdivost a nepravdivost reprezentována celočíselnými proměnnými, ale standard ANSI zavedl typ `bool`. Tento typ může nabývat pouze dvou hodnot: `false` (nepravda) a `true` (pravda).

Každý výraz je možné vyhodnotit z hlediska jeho pravdivosti či nepravdivosti. Výrazy, které se matematicky vyhodnotí na nulu, budou vracet hodnotu `false`. Všechny ostatní budou vracet hodnotu `true`.

POZNÁMKA Dříve mnoho kompilátorů poskytovalo typ `bool`, který byl vnitřně reprezentován jako `long int`, a proto měl velikost čtyři bajty. Kompilátory splňující standard ANSI nyní často nabízí jednobajtový typ `bool`.

Relační operátory

Relační operátory se používají k určení, zda jsou si dvě čísla rovna nebo zda je jedno větší než druhé. Každý příkaz s relačním operátorem se vyhodnotí jako pravdivý (`true`) nebo nepravdivý (`false`). Relační operátory jsou uvedeny později v tabulce 4.1.

POZNÁMKA Všechny relační operátory vrací hodnotu typu `bool`: buď pravda (`true`), nebo nepravda (`false`). V předcházejících verzích C++ vracely tyto operátory pro nepravda nulu a pro pravda nenulovou hodnotu, obvykle však 1.

Jestliže má celočíselná proměnná `mujVek` hodnotu 45 a celočíselná proměnná `tvujVek` 50, můžete s pomocí relačního operátoru „rovná-se“ určit, zda se hodnoty těchto proměnných rovnají:

```
mujVek == tvujVek;   // je hodnota v proměnné mujVek stejná jako v proměnné tvujVek?
```

Výraz se vyhodnotí jako nepravdivý, protože proměnné se nerovnají. Výraz

```
mujVek < tvujVek;    //je mujVek menší než tvujVek?
```

se vyhodnotí jako pravdivý.

UPOZORNĚNÍ Mnozí začínající programátoři v C++ zaměňují operátor přiřazení (=) za operátor rovná se (==). V programu může díky tomu dojít k nepříjemným chybám.

Relačních operátorů je šest a jsou to: rovná se (=), menší než (<), větší než (>), menší nebo rovno (<=), větší nebo rovno (>=) a nerovná se (!=). V tabulce 4.1 jsou všechny relační operátory uvedeny s krátkým příkladem.

Tabulka 4.1 Relační operátory

Název	Operátor	Příklad	Vyhodnotí se jako
Rovná se	==	100 == 50; 50 == 50;	nepravda pravda
Nerovná se	!=	100 != 50; 50 != 50;	pravda nepravda
Větší než	>	100 > 50; 50 > 50;	pravda nepravda
Větší nebo rovno	>=	100 >= 50; 50 >= 50;	pravda pravda
Menší než	<	100 < 50; 50 < 50;	nepravda nepravda
Menší nebo rovno	<=	100 <= 50; 50 <= 50;	nepravda pravda

ANO	NE
ANO, pamatujte si, že relační operátory vrací hodnotu pravda (`true`) nebo nepravda (`false`).	**NE**, nezaměňujte operátor přiřazení (=) a relační operátor rovná se (==). Jedná se o jednu z nejčastějších programátorských chyb; mějte se před ní proto na pozoru.

Příkaz `if`

Normálně se program provádí řádek po řádku, a to v pořadí, v jakém se řádky vyskytují ve zdrojovém kódu. Příkaz `if` umožňuje testování podmínky (například zda se dvě proměnné rovnají) a rozvětvení kódu na různé části podle jejího výsledku.

Jedna z nejjednodušších podob příkazu `if` vypadá takto:

```
if (výraz)
      příkaz;
```

Výraz v závorce může být libovolný, ale obvykle se jedná o výraz, který obsahuje relační operátor. Má-li výraz hodnotu nepravda, příkaz se přeskočí. Jestliže se hodnota výrazu vyhodnotí jako pravda, příkaz se provede. Vezměme následující příklad:

```
if (velkeCislo > maleCislo)
      velkeCislo = maleCislo;
```

Tento kód porovná proměnné `maleCislo` a `velkeCislo`. Je-li proměnná `velkeCislo` větší, nastaví se na druhém řádku hodnota této proměnné na hodnotu proměnné `maleCislo`.

Protože blok příkazů, který je ohraničený složenými závorkami, je ekvivalentní s jediným příkazem, může být větvení poměrně rozsáhlé a mocné:

```
if (výraz)
{
            příkaz1;
            příkaz2;
            příkaz3;
}
```

Ukážeme tento model větvení na konkrétním příkladě:

```
if (velkeCislo > maleCislo)
{
        velkeCislo = maleCislo;
        std::cout << "velkeCislo: " << velkeCislo << "\n";
        std::cout << "maleCislo: " << maleCislo << "\n";
}
```

Jestliže je v tomto případě `velkeCislo` větší než `maleCislo`, pak nejenom že se nastaví na hodnotu proměnné `maleCislo`, ale vytiskne se i zpráva, která nás o tom bude informovat. Ve výpisu 4.4 najdete propracovanější příklad větvení, které je založeno na relačních operátorech.

Výpis 4.4

Ukázka větvení programu s relačními operátory v podmínce

```
0: // Výpis 4.4 - demonstruje příkaz if
1: // spolu s relačními operátory
2: #include <iostream>
3: int main()
4: {
5:     using std::cout;
6:     using std::cin;
7:
```

```
 8:     int skoreSparta, skoreSlavia;
 9:     cout << "Vlozte skore Sparty: ";
10:     cin >> skoreSparta;
11:
12:     cout << "\nVlozte skore Slavie: ";
13:     cin >> skoreSlavia;
14:
15:     cout << "\n";
16:
17:     if (skoreSparta > skoreSlavia)
18:             cout << "Sparta do toho!\n";
19:
20:     if (skoreSparta < skoreSlavia)
21:     {
22:         cout << "Slavia do toho!\n";
23:     }
24:
25:     if (skoreSparta == skoreSlavia)
26:     {
27:         cout << "Vyrovnane skore? To neni mozne!\n";
28:         cout << "Zadejte skutecne skore Slavie: ";
29:         cin >> skoreSlavia;
30:
31:         if (skoreSparta > skoreSlavia)
32:             cout << "Vedel jsem to! Sparta do toho!";
33:
34:         if (skoreSlavia > skoreSparta)
35:             cout << "Vedel jsem to! Slavia do toho!";
36:
37:         if (skoreSparta == skoreSlavia)
38:             cout << "Aha, je to skutecne vyrovnane!";
39:     }
40:
41:     cout << "\nDiky za informace.\n";
42:     return 0;
43: }
```

Výstup

```
Vlozte skore Sparty: 3

Vlozte skore Slavie: 3

Vyrovnane skore? To neni mozne!
Zadejte skutecne skore Slavie: 2
Vedel jsem to! Sparta do toho!
Diky za informace.
```

Analýza Program uživatele požádá, aby zadal skóre dvou fotbalových družstev. Skóre se uloží do celočíselných proměnných a ty se na řádcích 17, 20 a 25 porovnávají v příkazech `if`.

Bude-li jedno skóre vyšší než druhé, vytiskne se zpráva, která o tom bude informovat. Budou-li skóre stejná, přejde se k bloku kódu, který začíná na řádku 25 a končí na řádku 39. Uživatel bude znovu požádán o druhé skóre a obě skóre se znovu porovnají.

Všimněte si, že kdyby bylo počáteční skóre Slavie vyšší než Sparty, vyhodnotil by se příkaz `if` na řádku 17 jako nepravda a nedošlo by k vyvolání řádku 18. Test na řádku 20 by se vyhodnotil jako pravda a vyvolal by se příkaz na řádku 22. Pak by se na řádku 25 testoval příkaz `if` a ten by se vyhodnotil jako nepravda (kdyby se řádek 22 vyhodnotil jako pravda). Takto by program přeskočil celý blok a dostal by se až na řádek 39.

Na tomto příkladě jsme ilustrovali, že pravdivý výsledek jednoho příkazu `if` nezabraní testování ostatních příkazů `if`.

Všimněte si, že u prvních dvou příkazů `if` je akcí, která se má provést, jeden řádek (tisk „Sparta do toho!“ nebo „Slavia do toho!“). V prvním případě (řádek 18) není tento řádek v závorkách, protože se to u bloku s jedním řádkem nevyžaduje. Složené závorky jsou zde však povolené a používají se u řádků 21 až 23.

POZOR! Mnozí začínající programátoři v C++ nechtěně dávají za příkaz `if` středník:

```
if (NejakaHodnota < 10);
     NejakaHodnota = 10;
```

V tomto příkladě se mělo otestovat, zda je hodnota proměnné NejakaHodnota menší než 10, a v případě že ano, měla by se nastavit na 10, tedy na svou minimální hodnotu. Když se tento kousek kódu spustí, nastaví se hodnota proměnné NejakaHodnota vždy na číslo 10. Proč? Příkaz `if` je ukončen středníkem (operátor, který znamená, že se nemá dělat nic).

Nezapomínejte, že odsazení nemá pro kompilátor žádný význam. Výše uvedený úsek kódu je možné napsat také takto:

```
if (NejakaHodnota < 10)    // test
;   // nedělej nic
NejakaHodnota = 10;        // přiřazení
```

Když se středník odstraní, stane se poslední řádek příkazem `if` a kód bude fungovat tak, je byl původně zamýšlen.

Styly odsazování

Ve výpisu 4.4 si můžete všimnout jednoho způsobu odsazování příkazu `if`. Nic tak nevzbudí vášně, jako když se skupiny programátorů zeptáte, jaký je nejlepší styl zarovnání složených závorek. Ačkoli existují tucty možností, mezi tři nejoblíbenější patří tyto:

- Levou složenou závorku dát za podmínku a pravou složenou závorku uzavírající blok příkazů zarovnat s příkazem `if`:

```
if (výraz) {
     příkazy
}
```

- Zarovnat závorky s příkazem `if` a příkazy odsadit:

```
if (výraz)
{
      příkazy
}
```

- Odsadit závorky a příkazy:

```
if (výraz)
    {
    příkazy
    }
```

V této knize upřednostňujeme druhý způsob, protože je zřejmé, kde bloky příkazů začínají a kde končí, když jsou závorky zarovnané jedna s druhou i s testovanou podmínkou. Znovu však podotýkáme, že nezáleží příliš na tom, který styl si vyberete, pokud jej budete ve svém programu používat konzistentně.

Klauzule `else`

Často je potřeba program větvit ve dvou směrech. Jedním způsobem, když je podmínka pravdivá, a jinak, když je nepravdivá. V programu ve výpisu 4.4 se napsala jedna zpráva (`Sparta do toho!`), jestliže hodnota prvního testu (`skoreSparta > skoreSlavia`) byla pravda, zatímco druhá zpráva (`Slavia do toho!`) se napsala tehdy, když byla tato hodnota nepravda.

Tato metoda testování nejdříve jedné podmínky a pak druhé je správná, ale trochu těžkopádná. Klíčové slovo `else` může takové testování značně zjednodušit:

```
if (výraz)
     příkaz;
else
     příkaz;
```

Ve výpisu 4.5 je program, který ilustruje používání klíčového slova `else`.

Výpis 4.5

Ukázka použití klíčového slova `else`

```
 0: // Výpis 4.5 - demonstruje prikaz if
 1: // s klauzuli else
 2: #include <iostream>
 3: int main()
 4: {
 5:     using std::cout;
 6:     using std::cin;
 7:
 8:     int prvniCislo, druheCislo;
 9:     cout << "Vlozte prosim velke cislo: ";
10:     cin >> prvniCislo;
11:     cout << "\nVlozte prosim mensi cislo: ";
12:     cin >> druheCislo;
13:     if (prvniCislo > druheCislo)
14:         cout << "\nDiky!\n";
15:     else
16:         cout << "\nOuha. Druhe cislo je vetsi!";
17:
18:     return 0;
19: }
```

Výstup

```
Vlozte prosim velke cislo: 10

Vlozte prosim mensi cislo: 12

Ouha. Druhe cislo je vetsi!
```

Analýza Příkaz na řádku 13 se vyhodnotí. Jestliže bude podmínka pravdivá, provede se příkaz na řádku 14. V případě, že bude nepravdivá, provede se příkaz na řádku 16. Kdyby se odstranila klauzule `else` na řádku 15, příkaz na řádku 16 by se provedl, ať už by se podmínka příkazu `if` vyhodnotila jako pravdivá nebo nepravdivá. Uvědomte si, že příkaz `if` končí za řádkem 14. Kdyby tam klauzule `else` nebyla, byl by řádek 16 jen následujícím řádkem v programu.

Zapamatujte si, že jeden nebo oba příkazy by se mohly nahradit blokem kódu ve složených závorkách.

Příkaz `if`

Syntaxe příkazu `if` je následující:

Podoba 1:

```
if (výraz)
příkaz;
další příkaz;
```

Je-li výraz vyhodnocen jako pravdivý, provede se příkaz a program pokračuje dalším příkazem. Není-li výraz pravdivý, příkaz se ignoruje a program pokračuje dalším příkazem.

Zapamatujte si, že příkazem může být jeden příkaz zakončený středníkem nebo blok, který je uzavřen ve složených závorkách.

Podoba 2:

```
if  (výraz)
      příkaz 1;
else
      příkaz 2;
další příkaz;
```

Vyhodnotí-li se výraz jako pravdivý, provede se příkaz 1, jinak se provede příkaz 2. Pak bude program pokračovat dalším příkazem.

Příklad 1

```
Příklad
if (NejakaHodnota < 10)
   cout << "NejakaHodnota je mensi nez 10";
else
   cout << "NejakaHodnota neni mensi nez 10";
cout << "Hotovo." << endl;
```

Pokročilé příkazy `if`

Stojí za pozornost, že v klauzuli `if` nebo `else` je možné použít jakýkoli příkaz, dokonce i další příkaz `if` nebo `else`. Lze tedy vytvořit vnořené příkazy `if`, jakým je například tento:

```
if (výraz1)
{
    if (výraz2)
        příkaz1;
    else
    {
         if (výraz3)
              příkaz2;
         else
              příkaz3;
     }
}
else
     příkaz4;
```

V tomto těžkopádném příkazu `if` se říká: „Je-li výraz1 a výraz2 pravdivý, proveď příkaz1. Jestliže je výraz1 pravdivý, ale výraz2 je nepravdivý, pak je-li výraz3 pravdivý, proveď příkaz2. Je-li výraz1 pravdivý, ale výraz2 a výraz3 jsou nepravdivé, proveď příkaz3. Nakonec je-li výraz1 nepravdivý, proveď příkaz4.“ Jak můžeme vidět, vnořené příkazy `if` mohou být hodně komplikované.

Výpis 4.6 je příkladem jednoho takového složitého příkazu `if`.

Výpis 4.6

Složitý, vnořený příkaz `if`

```
 0: // Výpis 4.6 - složitý vnořený
 1: // příkaz if
 2: #include <iostream>
 3: int main()
 4: {
 5:     // Požádá o zadání dvou čísel
 6:     // Přiřadí čísla do proměnných prvniCislo a druheCislo
 7:     // Pokud je prvniCislo větší než druheCislo,
 8:     // zjistí, zda jsou dělitelná beze zbytku
 9:     // Pokud jsou, zjistí, zda se jedná o stejné číslo
10:
11:     using namespace std;
12:
13:     int prvniCislo, druheCislo;
14:     cout << "Zadejte dve cisla.\nPrvni: ";
15:     cin >> prvniCislo;
16:     cout << "\nDruhe: ";
17:     cin >> druheCislo;
18:     cout << "\n\n";
19:
20:     if (prvniCislo >= druheCislo)
21:     {
22:         if ( (prvniCislo % druheCislo) == 0) // dělitelná beze zbytku?
23:         {
24:             if (prvniCislo == druheCislo)
25:                 cout << "Jsou stejna!\n";
26:             else
```

```
27:                    cout << "Jsou delitelna beze zbytku!\n";
28:          }
29:          else
30:               cout << "Nejsou delitelna beze zbytku!\n";
31:      }
32:      else
33:          cout << "Druhe cislo je vetsi!\n";
34:      return 0;
35: }
```

Výstup

```
Zadejte dve cisla.
Prvni: 10

Druhe: 2


Jsou delitelna beze zbytku!
```

Analýza Nejdříve je uživatel vyzván, aby zadal dvě čísla. Na řádku 20 testuje první příkaz `if`, zda je první číslo větší nebo rovno druhému. Není-li tomu tak, provede se klauzule `else` na řádku 32.

Je-li první příkaz `if` pravdivý, provede se blok kódu, který začíná na řádku 21, a na řádku 22 bude proveden druhý test v příkazu `if`. Má určit, zda zbytek po dělení druhého čísla prvním číslem dává zbytek 0. Je-li tomu tak, jsou čísla buď beze zbytku dělitelná, nebo stejná. Příkaz `if` na řádku 24 zjistí, zda jsou si rovna, a zobrazí příslušnou odpověď.

Nebude-li příkaz `if` na řádku 22 pravdivý, provede se příkaz `else` na řádku 29.

Používání závorek ve vnořených příkazech `if`

I když je dovoleno vynechávat závorky u příkazů `if`, které jsou tvořeny pouze jedním příkazem, a i když je při psaní vnořených příkazů `if` povolen tento zápis:

```
if (x > y)                    // je-li x větší než y
    if (x < z)                // a je-li x menší než z
        x = y;                // pak nastav x na hodnotu y
```

může jeho používání vést k mnohým nesrovnalostem. Pamatujte si, že prázdné znaky a zarovnání jsou při psaní programu velmi praktickým nástrojem a pro kompilátor nepředstavují žádný rozdíl. Velmi snadno může dojít k neúmyslnému přiřazení příkazu `else` nesprávnému příkazu `if`. Program ve výpisu 4.7 tento problém ilustruje.

Výpis 4.7

Ilustrace toho, jak složené závorky pomáhají stanovit, který příkaz `else` patří ke kterému příkazu `if`

```
0: // Výpis 4.7 - demonstruje, proč jsou složené závorky
1: // tak podstatné ve vnořených příkazech if
2: #include <iostream>
```

```
 3: int main()
 4: {
 5:     int x;
 6:     std::cout << "Vlozte cislo mensi nez 10 nebo vetsi nez 100: ";
 7:     std::cin >> x;
 8:     std::cout << "\n";
 9:
10:     if (x >= 10)
11:         if (x > 100)
12:             std::cout << "Vetsi nez 100, diky!\n";
13:     else                    // tato klauzule else není správně umístěna!
14:         std::cout << "Mensi nez 10, diky!\n";
15:
16:     return 0;
17: }
```

Výstup

```
Vlozte cislo mensi nez 10 nebo vetsi nez 100: 20

Mensi nez 10, diky!
```

Analýza Záměrem programátora bylo požádat o číslo menší než 10 nebo větší než 100, ověřit správnost zadané hodnoty a vytisknout poznámku s poděkováním.

Jestliže se příkaz `if` na řádku 10 vyhodnotí jako pravdivý, provede se následující příkaz na řádku 11. V tomto případě se řádek 11 provede, když zadané číslo bude větší než 10. Řádek 11 také obsahuje příkaz `if`. Tento příkaz se vyhodnotí jako pravdivý, jestliže bude zadané číslo větší než 100. Je-li číslo skutečně větší než 100, provede se příkaz na řádku 12.

Bude-li zadané číslo menší než 10, vyhodnotí se příkaz na řádku 10 jako nepravdivý. Řízení programu se posune na další řádek za příkazem `if`, což v tomto případě bude řádek 17. Bude-li zadané číslo menší než 10, dostaneme následující výstup:

```
Vlozte cislo mensi nez 10 nebo vetsi nez 100: 9
```

Klauzule `else` na řádku 13 se měla připojit k příkazu `if` na řádku 10. Ve skutečnosti je bohužel tato klauzule `else` připojena k příkazu `if` na řádku 11, a proto je v programu nenápadná chyba.

Nenápadná je tato chyba proto, že kompilátor si na ni nebude stěžovat. Jedná se o správný program C++, který nedělá to, co se jím zamýšlelo. Navíc při většině testů se bude zdát, že program funguje správně. Pokud se budou zadávat čísla větší než 100, program bude provádět svou činnost bez problémů.

V programu ve výpisu 4.8 je chyba opravena s pomocí složených závorek.

Výpis 4.8

Ukázka správného používání složených závorek u příkazu `if`

```
 0: // Výpis 4.8 - demonstruje správné používání složených závorek
 1: // ve vnořených příkazech if
 2: #include <iostream>
 3: int main()
 4: {
 5:     int x;
```

```
 6:      std::cout << "Vlozte cislo mensi nez 10 nebo vetsi nez 100: ";
 7:      std::cin >> x;
 8:      std::cout << "\n";
 9:
10:      if (x >= 10)
11:      {
12:          if (x > 100)
13:              std::cout << "Vetsi nez 100, diky!\n";
14:      }
15:      else                              // opraveno!
16:          std::cout << "Mensi nez 10, diky!\n";
17:      return 0;
18: }
```

Výstup

```
Vlozte cislo mensi nez 10 nebo vetsi nez 100: 9

Mensi nez 10, diky!
```

Analýza Složené závorky na řádcích 11 a 14 uzavřou vše, co je mezi nimi, do jednoho příkazu a klauzule `else` na řádku 15 už bude patřit k příkazu `if` na řádku 10, což bylo původním záměrem.

Jestliže uživatel zadá 20, příkaz `if` na řádku 10 bude pravdivý, příkaz `if` na řádku 12 však bude nepravdivý, takže se nic nevytiskne. Bylo by asi lepší, kdyby programátor dal za řádek 13 ještě jednu klauzuli `else` a tak by došlo k zachycení chyby a vytištění zprávy.

Tip Většinu potíží vyplývajících z příkazů `if...else` lze odstranit uzavřením příkazů v klauzulích if a else do složených závorek, a to i v případě, kdy za podmínkou následuje jediný příkaz.

```
if (nejakaHondota < 10)
{
        nejakaHondota = 10;
}
else
{
  nejakaHondota = 25
};
```

POZNÁMKA Všechny programy v této knize určené k demonstraci konkrétních témat jsou záměrně zjednodušené. Nesnaží se být neprůstřelné a odolné proti chybám uživatele. V ideálním kódu na profesionální úrovni by se mělo s každou možnou chybou uživatele počítat a měla by se elegantně vyřešit.

Logické operátory

Často můžeme chtít v podmínce položit více než jednu otázku: „Je pravda, že x je větší než y, a je také pravda, že y je větší než z?“ Program se může rozhodnout, že jsou-li obě tyto podmínky pravdivé nebo když je pravdivá nějaká jiná podmínka, má se provést nějaká akce.

Představte si propracovaný poplašný systém, který pracuje takto: „Jestliže poplašný systém dveří zní A ZÁROVEŇ je šest hodin odpoledne A ZÁROVEŇ NEjsou prázdniny NEBO je víkend, zavolej policii.“ U takového typu podmínky se v jazyce C++ používají tři logické operátory. Najdete je uvedené v tabulce 4.2.

Tabulka 4.2 Logické operátory

Operátor	Symbol	Příklad
AND	&&	výraz1 && výraz2
OR	||	výraz1 || výraz2
NOT	!	!výraz

Logické `AND` (a zároveň)

Logický operátor `AND` vyhodnocuje dva výrazy, a jsou-li oba pravdivé, bude pravdivý i celý logický výraz `AND`. Jestliže je pravda, že máte hlad, a zároveň je pravda, že máte peníze, pak je pravda, že si můžete koupit oběd. Tedy výraz

```
if ((x == 5) && (y == 5))
```

se vyhodnotí jako pravdivý, jestliže bude hodnota obou proměnných `x` i `y` rovna 5, a vyhodnotí se jako nepravdivý, jestliže se hodnota jedné z těchto proměnných nebude rovnat 5. Všimněte si, že má-li být pravdivý celý výraz, musí být pravdivé obě jeho strany.

Logický operátor `AND` je symbolizován dvěma znaky `&&`. Zapamatujte si, že samotný jeden znak `&` se používá jako symbol pro jiný operátor, jehož význam probereme v den 21. – „Co dál“.

Logické `OR` (nebo)

Logický výraz `OR` vyhodnocuje dva výrazy. Je-li jeden z těchto výrazů pravdivý, bude celý výraz pravdivý. Máte-li peníze nebo kreditní kartu, můžete zaplatit účet. K tomu, abyste účet zaplatili, nepotřebujete i peníze i kreditní kartu. Stačí vám jedno nebo druhé, a i když máte obojí, je to také v pořádku. Výraz

```
if ((x == 5) || (y == 5))
```

se vyhodnotí jako pravdivý, jestliže buď proměnná `x` nebo `y` bude mít hodnotu 5 nebo jestliže budou mít tuto hodnotu obě proměnné.

Všimněte si, že symbolem logického OR jsou dva znaky `||`. Jeden znak `|` je symbolem pro jiný operátor, který probereme v den 21.

Logické `NOT` (ne)

Logický výraz `NOT` se vyhodnotí jako pravdivý, jestliže je testovaný výraz nepravdivý. Opakujeme, je-li testovaný výraz nepravdivý, hodnota testu bude pravdivá. Výraz

```
if (!(x == 5))
```

je pravdivý, jestliže se proměnná x nerovná číslu 5. Je to stejné, jako kdybychom napsali

```
if (x != 5)
```

Zkrácené vyhodnocování

Při vyhodnocování příkazu AND, jakým je například

```
if ((x == 5) && (y == 5))
```

kompilátor vyhodnotí pravdivost prvního výrazu (x == 5), a když zjistí, že je nepravdivý, nebude pokračovat vyhodnocováním druhého výrazu (y == 5), protože příkaz AND vyžaduje, aby byly oba příkazy pravdivé.

Podobně, když při vyhodnocování příkazu OR, jakým je například

```
if ((x == 5) || (y == 5))
```

kompilátor zjistí, že první příkaz (x == 5) je pravdivý, nebude nikdy pokračovat vyhodnocováním druhého příkazu (y == 5), protože pravdivost *libovolného* z nich je pro příkaz OR dostatečná.

Byť se to nemusí zdát důležité, zvažte následující příklad:

```
if ( (x == 5) || (++y == 3) )
```

Není-li x rovno pěti, pak se výraz (++y == 3) nevyhodnotí. Budete-li se spoléhat na vždy probíhající navýšení proměnné y, nemusí k němu dojít.

Priorita relačních a logických operátorů

Relační i logické operátory jsou výrazy v C++ a jako takové vrací hodnotu pravda nebo nepravda. Podobně jako pro všechny výrazy, existuje pro ně pořadí, které stanoví, jaký podvýraz se bude vyhodnocovat jako první (viz dodatek C). Tento fakt je důležitý při stanovení hodnoty následujícího výrazu:

```
if ( x > 5 && y > 5 || z > 5)
```

Programátor může například chtít, aby se tento výraz vyhodnotil jako pravdivý, jestliže budou hodnoty obou proměnných x a y větší než 5 nebo jestliže bude hodnota proměnné z větší než 5. Programátor by však také mohl chtít, aby se tento výraz vyhodnotil jako pravdivý pouze tehdy, jestliže bude hodnota proměnné x větší než 5 a jestliže bude také pravda, že buď proměnná y, nebo proměnná z je větší než 5.

Jestliže je hodnota proměnné x rovna 3 a hodnota proměnných y a z je rovna 10, bude v prvním případě tento výraz pravdivý (z je větší než 5, proto se zbytek ignoruje), ale ve druhém případě bude hodnota tohoto výrazu nepravda (není pravda, že x je větší než 5, a proto nezáleží na tom, co je na pravé straně symbolu &&, protože obě jeho strany musí být pravdivé).

Ačkoliv pravidla pro přednost operátorů určují, který podvýraz se bude vyhodnocovat jako první, závorky mohou posloužit ke stejnému účelu a zároveň přispět k přehlednosti výrazu:

```
if ((x > 5 && (y > 5 || z > 5 ))
```

Pro výše uvedené hodnoty je tento výraz nepravdivý. Není totiž pravda, že proměnná `x` je větší než 5, tedy levá strana výrazu `AND` je nepravdivá, a proto je celý výraz nepravdivý. Nezapomínejte, že výraz `AND` vyžaduje, aby byly obě jeho strany pravdivé – ani koláč nemůže zároveň chutnat dobře a zároveň chutnat špatně.

TIP Je často vhodné použít závorky navíc, aby bylo zřejmé, co chcete sdružit. Nezapomínejte, že cílem je psát programy, které fungují a ve kterých se dobře orientuje. Použitím závorek vyjasňujete své záměry a vyhýbáte se chybám vyplývajícím z nesprávného pochopení předností operátorů.

Více o pravdivosti a nepravdivosti

V jazyce C++ se nula vyhodnocuje jako nepravda a všechny ostatní hodnoty se vyhodnocují jako pravda. Protože výraz má vždy hodnotu, využívají toho mnozí programátoři C++ v příkazech `if`. Příkaz jako je

```
if (x)                  // jestliže x je pravda (nenulová hodnota)
    x = 0;
```

se dá přečíst takto: „Jestliže má proměnná x nenulovou hodnotu, nastav hodnotu této proměnné na 0.“ Jedná se zde o malý podvod. Vše by bylo jasnější, kdyby zápis vypadal takto:

```
if  (x != 0)            // jestliže má x nenulovou hodnotu
      x = 0;
```

Oba příkazy jsou povolené, ale ten druhý je srozumitelnější. Je dobrým zvykem vyhradit si první způsob pro logické testy pravdivosti a nepoužívat jej při testování nenulových hodnot.

Následující dva příkazy jsou ekvivalentní:

```
if (!x)                     // jestliže x je nepravda (nula)
if (x == 0)                 // jestliže x je nula
```

Druhý výraz je poněkud srozumitelnější, a jestliže se testuje spíše matematická hodnota proměnné `x` než logický stav, bude toto vyjádření i explicitnější.

ANO	NE
ANO, umisťujte výrazy logického testování do závorek. Výraz bude srozumitelnější a bude mít explicitně vyjádřenou prioritu. **ANO**, používejte u vnořených příkazů `if` složené závorky. Bude pak zřejmé, kam patří příkaz `else`, a vyhnete se chybám.	**NE**, nepoužívejte výraz `if (x)` jako synonymum pro `if (x != 0)`. Druhý výraz je srozumitelnější. **NE**, nepoužívejte `if (!x)` jako synonymum pro `if (x == 0)`. Druhý výraz je srozumitelnější.

Podmínkový operátor

Operátor podmínky (`?:`) je jediným ternárním operátorem v jazyce C++. To znamená, že se jedná o jediný operátor, který očekává tři operandy.

Operátor podmínky akceptuje tři výrazy a vrací hodnotu:

```
(výraz1) ? (výraz2) : (výraz3)
```

Tento řádek by se mohl číst takto: „Jestliže je výraz1 pravdivý, vrať hodnotu výrazu výraz2, jinak vrať hodnotu výrazu výraz3.“ Typické je přiřadit výsledek podmínkového operátoru proměnné.

Program ve výpisu 4.9 ukazuje, jak se příkaz `if` přepíše s pomocí operátoru podmínky.

Výpis 4.9

Ukázka použití operátoru podmínky

```
 0: // Výpis 4.9 - demonstruje operátor podmínky
 1: //
 2: #include <iostream>
 3: int main()
 4: {
 5:     using namespace std;
 6:
 7:     int x, y, z;
 8:     cout << "Vlozte dve cisla.\n";
 9:     cout << "Prvni: ";
10:     cin >> x;
11:     cout << "\nDruhe: ";
12:     cin >> y;
13:     cout << "\n";
14:
15:     if (x > y)
16:         z = x;
17:     else
18:         z = y;
19:
20:     cout << "z: " << z;
21:     cout << "\n";
22:
23:     z =  (x > y) ? x : y;
24:
25:     cout << "z: " << z;
26:     cout << "\n";
27:     return 0;
28: }
```

Výstup

```
Vlozte dve cisla.
Prvni: 5

Druhe: 8

z: 8
z: 8
```

Analýza Vytvoří se tři celočíselné proměnné: `x`, `y` a `z`. Prvním dvěma uživatel přiřadí hodnotu. Příkaz `if` na řádku 15 otestuje, která je větší, a tu přiřadí proměnné `z`. Na řádku 20 se tato hodnota vytiskne.

Operátor podmínky na řádku 23 provede stejný test a přiřadí proměnné `z` větší hodnotu. Přečte se takto: „Jestliže je hodnota proměnné `x` větší než hodnota proměnné `y`, vrať hodnotu proměnné `x`, jinak vrať hodnotu proměnné `y`.“ Vrácená hodnota se přiřadí proměnné `z`. Ta se pak vytiskne na řádku 25. Jak sami vidíte, příkaz podmínky je kratší ekvivalent příkazu `if ... else`.

Souhrn

V této kapitole jsme probrali celou řadu témat. Dozvěděli jste se, co jsou to výrazy a příkazy jazyka C++, k čemu slouží operátory C++ a jak funguje příkaz C++ `if`.

Seznámili jste se s blokem příkazů, jenž je uzavřen párem složených závorek a který lze použít všude tam, kde se dá použít jeden příkaz.

Dozvěděli jste se, že každý výraz se vyhodnotí na hodnotu a tuto hodnotu je pak možné testovat příkazem `if` nebo s pomocí operátoru podmínky. Seznámili jste se také se způsobem vyhodnocování více příkazů prostřednictvím logických operátorů. Dále už víte, jak se s pomocí relačních operátorů dají porovnávat hodnoty a jak se s pomocí operátoru přiřazení dá hodnota přiřadit.

Seznámili jste se s prioritou operátorů a také víte, jak ke změně priority použít závorky. Ty také přispívají k tomu, že vyjádření v kódu je explicitnější a že kód se lépe udržuje.

Otázky a odpovědi

Otázka: **Proč používat závorky, které nejsou nezbytné, když je priorita operátorů jasně určena?**

Odpověď: Přestože je to pravda, kompilátor ví, kterým výrazům má dát přednost, a programátor si to může zjistit, je takový kód mnohem přehlednější a lépe se udržuje.

Otázka: **Jestliže relační operátory vždy vrací hodnotu pravda nebo nepravda, proč se každá nenulová hodnota považuje za pravdivou?**

Odpověď: Tato konvence pochází z jazyka C, který se často používal k vytváření nízkoúrovňového softwaru, jako jsou operační systémy nebo programy řízení v reálném čase. Je pravděpodobné, že se zmíněné použití vyvinulo jako zkratkové testování toho, zda jsou všechny bity v nějaké masce nebo proměnné rovny nule. Relační operátory vrací hodnotu pravda nebo nepravda, ale každý výraz vrací hodnotu a tyto hodnoty lze také vyhodnocovat v příkazu `if`. Zde je příklad:

```
if (( x = a + b) == 35)
```

Jedná se o naprosto správný příkaz C++, který se vyhodnotí na hodnotu, i když se nebude součet `a` a `b` rovnat 35. Také si všimněte, že proměnné `x` bude v každém případě přiřazena hodnota, která je součtem proměnných `a` a `b`.

Otázka: **Jaký vliv mají na program tabulátory, mezery a nové řádky?**

Odpověď: Tabulátory, mezery a nové řádky nemají na program vůbec žádný vliv, i když jejich rozumné používání může přispět k jeho lepší čitelnosti.

Otázka: **Jakou logickou hodnotu mají záporná čísla?**

Odpověď: Všechna nenulová čísla, kladná i záporná, mají hodnotu pravda.

Úkoly pro vás

Testovací otázky v této části vám pomohou upevnit znalosti, které jste v této kapitole získali. Cvičení slouží k tomu, aby vám nabídlo praktickou zkušenost s tím, co jste se naučili. Pokuste se na otázky v testu a cvičení odpovědět dříve, než se podíváte do klíče v dodatku D. Než přejdete k další kapitole, ujistěte se, že každé odpovědi rozumíte.

Test

1. Co rozumíme výrazem?
2. Je x = 5 + 7 výraz? Jaká je jeho hodnota?
3. Jaká je hodnota 201/4?
4. Jaká je hodnota 201%4?
5. Jsou-li `mujVek`, `a` a `b` celočíselné proměnné, jaké budou jejich hodnoty poté, co se provedou následující příkazy:

```
mujVek = 39;
a = mujVek++;
b = ++mujVek;
```

6. Jaká je hodnota výrazu 8+2*3?
7. Jaký je rozdíl mezi `if(x = 3)` a `if(x == 3)`?
8. Vyhodnotí se následující výrazy na logickou hodnotu pravda nebo nepravda?

```
a/    0
b/    1
c/    -1
d/    x = 0
e/    x == 0 // přepokládáme, že x má hodnotu 0
```

Cvičení

1. Napište jeden příkaz `if`, který porovná dvě celočíselné proměnné a s pomocí pouze jedné klauzule `else` změní tu větší na menší.
2. Projděte si následující program. Představte si, že zadáte tři čísla. Napište výstup, který očekáváte.

```
 1:        #include <iostream>
 2:        using namespace std;
 3:        int main()
 4:        {
 5:              int a, b, c;
 6:              cout << "Prosim, zadejte tri cisla \n";
 7:              cout << "a: ";
 8:              cin >> a;
 9:              cout << "\nb: ";
10:              cin >> b;
```

```
11:              cout << "\bc: ";
12:              cin >> c;
13:
14:              if  (c = (a - b))
15:                     cout << "a: " << a << "minus b: " << b << "
                                                        _se rovna c: " << c;
16:              else
17:                     cout << "a - b se nerovna c: ";
18:              return 0;
19:         }
```

3. Zadejte program z příkladu 2, zkompilujte jej, sestavte a spusťte. Zadejte čísla 20, 10 a 50. Dostali jste očekávaný výstup? Proč ne?
4. Prostudujte si tento program a řekněte, co se stane.

```
1:        #include <iostream>
2:        using namespace std;
3:        int main()
4:        {
5:              int a = 2, b = 2, c;
6:              if  (c = (a-b))
7:                  cout << "Hodnota c je: " << c;
8:              return 0;
9:         }
```

5. Program ze cvičení 4 zadejte, zkompilujte a spusťte. Jaký jste dostali výstup? Proč?

den 5

Funkce

Přestože se pozornost v objektově orientovaném programování přesunuje od funkcí směrem k objektům, zůstávají funkce i nadále ústřední součástí každého programu. Globální funkce existují vně objektů a členské funkce (které se nazývají též členské metody) existují uvnitř objektů, ve kterých vykonávají svou činnost.

Dnes se naučíte a dozvíte:

- Co je to funkce a jaké má části.
- Jak se funkce deklaruje a definuje.
- Jak se funkci předají parametry.
- Jak se z funkce vrátí hodnota.

Začneme globálními funkcemi a v příští kapitole se budeme věnovat funkcím, které jsou součástí objektů.

Co je to funkce?

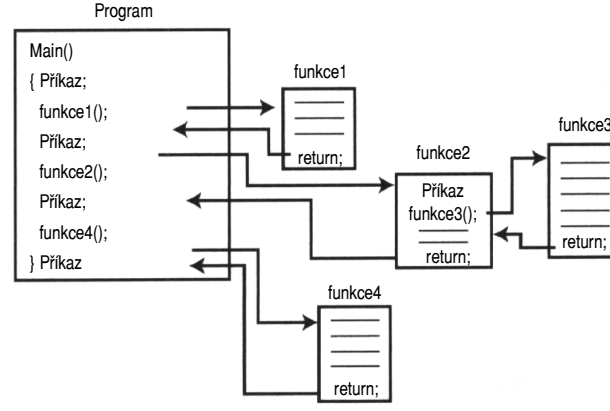
Funkce je prakticky podprogram, který umí zpracovat data a vrátit hodnotu. Každý program C++ má alespoň jednu funkci, kterou je funkce `main()`. Při spuštění programu se automaticky zavolá právě funkce `main()`. Funkce `main()` může volat ostatní funkce, z nichž některé mohou volat ještě další funkce.

Protože takové funkce nejsou součástí žádného objektu, nazývají se globální. To znamená, že přístup k nim je možný z libovolného místa v programu. Dnes, když budeme mluvit o funkcích, budeme mít vždy na mysli globální funkce, pokud nebude výslovně uvedeno jinak.

Každá funkce má svůj název, a když se tento název vyskytne v průběhu provádění programu, předá se řízení do těla funkce s tímto názvem. Tento proces se nazývá *volání* funkce. Když funkce vrátí řízení, obnoví se provádění programu na řádku, který následuje za voláním funkce. Na obrázku 5.1 je tento postup názorně ukázán.

Obrázek 5.1

Když program volá funkci, dojde k přesměrování provádění programu dovnitř této funkce a k pozdějšímu obnovení řízení na místě, které následuje ihned za voláním funkce.



Dobře navržené funkce provádí specifické úkoly, které jsou snadno srozumitelné. Složité úkoly by měly být rozděleny na více funkcí, které se volají postupně.

Můžeme se setkat se dvěma variantami funkcí: zabudované (vestavěné) a uživatelem definované. Zabudované funkce jsou součástí balíku kompilátoru; výrobce je dodává pro potřebu uživatele. Uživatelem definované funkce jsou takové, které si uživatel napíše sám.

Vrácené hodnoty, parametry a argumenty

Jak jste se dozvěděli v den 2., funkce mohou hodnoty nejen přebírat, ale také nějakou hodnotu *vracet*. Když se zavolá funkce, může provést úkol a jako výsledek své práce vrátit zpět hodnotu. Jedná se o *vrácenou hodnotu* a typ vrácené hodnoty je třeba deklarovat. Napíšeme-li tedy:

```
int mojeFunkce();
```

deklarujeme tím, že `mojeFunkce` bude vracet celočíselnou hodnotu.

Hodnoty lze také poslat *dovnitř* funkce. Tyto hodnoty fungují jako proměnné, se kterými je možné uvnitř funkce manipulovat.

Popis hodnot, které se funkci posílají, se nazývá seznam parametrů.

```
int mojeFunkce (int nejakaHodnota, float nejakyZlomek);
```

Touto deklarací se říká nejen to, že funkce `mojeFunkce` bude vracet celočíselnou hodnotu, ale také to, že si bude jako parametry brát jednu celočíselnou hodnotu a jednu hodnotu typu `float`.

Když posíláte hodnoty *do* funkce, pak fungují jako proměnné, s nimiž můžete ve funkci manipulovat. Popis posílaných hodnot se označuje jako seznam parametrů. Podle předchozího příkladu zahrnuje tento seznam parametrů proměnnou `nejakaHodnota` celočíselného typu a proměnnou `nejakyZlomek` typu `float`.

Parametr popisuje typ hodnoty, který se bude funkci předávat, když se funkce zavolá. Skutečné hodnoty, které se funkci předávají, se nazývají *argumenty*.

```
int VracenaHodnota = mojeFunkce(5,6.7);
```

V tomto případě se celočíselná proměnná `VracenaHodnota` inicializovala hodnotou vrácenou funkcí `mojeFunkce` a hodnoty `5` a `6.7` se jí předaly jako argumenty. Typy argumentů se musí shodovat s typy deklarovaných parametrů.

Deklarace a definice funkce

Při používání funkcí programem je nezbytné funkce nejdříve deklarovat a pak definovat. Deklarací se kompilátoru oznámí název funkce, návratový typ a parametry funkce. Definicí funkce se kompilátoru oznámí, jak přesně funkce pracuje. Žádnou funkci není možné volat z nějaké jiné funkce, aniž by tato funkce byla nejdříve deklarována. Deklarace funkce se nazývá její *prototyp*.

Deklarace funkce

Funkci je možné deklarovat třemi způsoby:

- Prototyp funkce se napíše do souboru a pak se tento soubor s pomocí direktivy `#include` zahrne do programu.
- Prototyp funkce se napíše do souboru, ve kterém se funkce používá.
- Funkce se definuje před tím, než ji jiná funkce bude volat. V tomto případě definice funguje jako svoje vlastní deklarace.

Ačkoli je možné funkci definovat před tím, než ji budete používat, a vyhnout se tak nutnosti vytvořit prototyp funkce, nejedná se o příliš dobrou praktiku, a to ze tří důvodů.

Zaprvé, není dobré, když se funkce v souboru musí objevit v daném pořadí. Když vyvstane potřeba změn, program se jen velmi obtížně spravuje.

Zadruhé, může se stát, že funkce `A()` musí být schopna volat funkci `B()`, ale funkce `B()` musí být také za určitých okolností schopna volat funkci `A()`. Není proto možné definovat funkci `A()` před tím, než se definuje funkce `B()`, a zároveň definovat funkci `B()` před tím, než se definuje funkce `A()`. Proto je v každém případě nutné alespoň jednu z nich deklarovat.

Zatřetí, prototypy funkcí jsou silným nástrojem při ladění programu. Jestliže se v prototypu deklaruje, že si funkce bere konkrétní množinu parametrů a že vrací konkrétní typ hodnoty, a pak se přihodí, že funkce s tímto prototypem nesouhlasí, kompilátor tuto chybu odhalí a není nutné čekat, až se sama objeví při spuštění programu. Podobá se to podvojnému

účetnictví. Prototyp se s definicí navzájem kontrolují, čímž se snižuje pravděpodobnost, že malý překlep povede k chybě v programu.

I přes vše výše uvedené volí velká většina programátorů třetí možnost. Snižuje se tím totiž počet řádků kódu, zjednodušuje se údržba (změny hlavičky funkce vyžadují také změnu prototypu) a pořadí funkcí v souboru je obvykle velmi stabilní. Ovšem v některých případech jsou prototypy nezbytné.

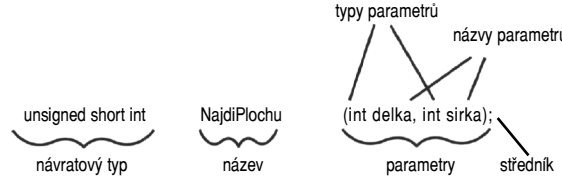
Prototypy funkcí

Většina zabudovaných funkcí má své prototypy již vytvořené. Vyskytují se v souborech, které se přidávají do programu s pomocí direktivy `#include`. U funkcí, které si uživatel píše sám, je třeba prototypy zapisovat.

Prototyp funkce je příkaz, takže musí být ukončen středníkem. Prototyp se skládá z návratového typu funkce a signatury. Signaturou funkce rozumíme její název se seznamem parametrů.

Seznam parametrů je seznamem všech parametrů a jejich typů, které jsou odděleny čárkami. Na obrázku 5.2 si můžete prohlédnout součásti prototypu funkce.

Obrázek 5.2
Součásti prototypu funkce



Prototyp funkce se musí přesně shodovat s definicí funkce v návratovém typu a signatuře. Jestliže se shodovat nebudou, obdržíte chybu v době kompilace. Všimněte si však, že prototyp funkce nemusí obsahovat názvy parametrů, pouze jejich typy. Prototyp, jako je ten následující, je v naprostém pořádku:

```
long Plocha(int, int);
```

V tomto prototypu se deklaruje funkce s názvem `Plocha()`, která vrací hodnotu typu `long` a která má dva parametry, oba typu `int`. Ačkoli se jedná o povolený způsob zápisu, není příliš vhodný. Když se k prototypu přidají i názvy parametrů, bude jeho smysl zřejmější. Stejná funkce s názvy parametrů by mohla vypadat takto:

```
long Plocha(int delka, int sirka);
```

Nyní je už zřejmé, co tato funkce provádí a jaké jsou její parametry.

Nezapomínejte, že všechny funkce mají návratový typ. Jestliže nebude explicitně žádný uveden, bude standardním návratovým typem `int`. Program však bude srozumitelnější, když budete návratový typ každé funkce explicitně uvádět, včetně funkce `main()`.

Pokud vaše funkce nevrací žádnou hodnotu, deklarujete její návratový typ jako `void`, jak je uvedeno zde:

```
void tisknoutCislo( int mojeCislo )
```

Jedná se o deklaraci funkce nazvané `tisknoutCislo()` s jedním celočíselným parametrem. Protože se jako návratový typ používá `void` (prázdnota), nevrací se nic.

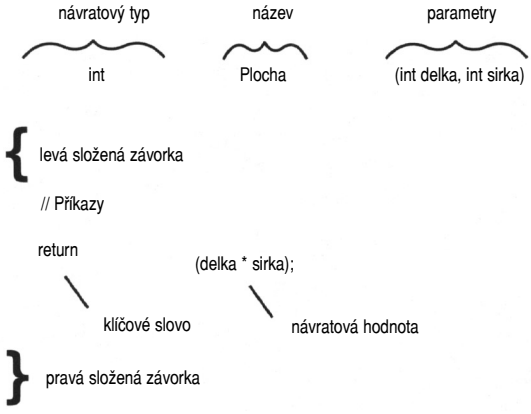
Definice funkce

Definice funkce se skládá z hlavičky funkce a jejího těla. Hlavička se podobá prototypu funkce, ale musí být opatřena názvy parametrů a na konci se neuvádí středník.

Tělem funkce rozumíme množinu příkazů, které jsou uzavřeny ve složených závorkách. Na obrázku 5.2 si můžete prohlédnout příklad hlavičky a těla funkce.

Výpis 5.1 představuje program, který obsahuje prototyp funkce `Plocha()`.

Obrázek 5.3
Hlavička a tělo funkce



Výpis 5.1

Deklarace funkce, její definice a použití

```
 0: // Výpis 5.1 - demonstruje používání prototypu funkcí
 1:
 2: #include <iostream>
 3: int Plocha(int delka, int sirka); //prototyp funkce
 4:
 5: int main()
 6: {
 7:     using std::cout;
 8:     using std::cin;
 9:
10:     int delkaDvorku;
11:     int sirkaDvorku;
12:     int plochaDvorku;
13:
14:     cout << "\nJak siroky je Vas dvorek? ";
15:     cin >> sirkaDvorku;
16:     cout << "\nJak dlouhy je Vas dvorek? ";
17:     cin >> delkaDvorku;
18:
19:     plochaDvorku= Plocha(delkaDvorku,sirkaDvorku);
20:
21:     cout << "\nVas dvorek ma plochu ";
22:     cout << plochaDvorku;
23:     cout << " ctverecnich metru.\n\n";
```

```
24:     return 0;
25: }
26:
27: int Plocha(int d, int s)
28: {
29:     return d * s;
30: }
```

Výstup

```
Jak siroky je Vas dvorek? 20

Jak dlouhy je Vas dvorek? 30

Vas dvorek ma plochu 600 ctverecnich metru.
```

Analýza Na řádku 3 je prototyp funkce `Plocha()`. Srovnejte tento prototyp s definicí funkce na řádku 27. Všimněte si, že název, návratový typ a typy parametrů jsou stejné. Kdyby se lišily, vygeneroval by kompilátor chybu. Ve skutečnosti jediný rozdíl mezi prototypem a funkcí je, že prototyp končí středníkem a nemá žádné tělo.

Také si všimněte, že názvy parametrů v prototypu jsou `delka` a `sirka`, ale názvy parametrů v definici jsou `d` a `s`. Jak jsme už uvedli, názvy v prototypu se nepoužívají, mají pouze informativní charakter. Je dobrým zvykem sjednotit názvy parametrů v prototypu s názvy parametrů v implementaci, ale není to nezbytně nutné.

Argumenty jsou funkci předávány v pořadí, ve kterém jsou deklarovány a definovány, ale neprobíhá žádné porovnávání názvů. Kdybyste funkci předali argumenty `sirkaDvorku` a `delkaDvorku` (v tomto pořadí), funkce `Plocha` by použila hodnotu `sirkaDvorku` pro délku a hodnotu `delkaDvorku` pro šířku.

Poznámka **Tělo funkce je vždy uzavřeno ve složených závorkách, dokonce i tehdy, když obsahuje jeden příkaz, jako je tomu i v uvedeném případě.**

Provádění funkcí

Když je funkce zavolána, její provádění začne prvním příkazem za levou (uvozovací) složenou závorkou ({). Větvení je možné provést s pomocí příkazu `if`. (Příkaz `if` a jiné podobné příkazy budou předmětem sedmého dne – „Více o toku programu“). Funkce mohou také volat jiné funkce, a dokonce mohou volat samy sebe (viz oddíl „Rekurze“ dále v této kapitole). Když vykonávání funkce skončí, vrátí se řízení volající funkci. Jakmile skončí funkce `main()`, řízení se vrátí operačnímu systému.

Obor platnosti proměnných

Každá proměnná má nějaký obor platnosti určující, jak dlouho je programu k dispozici a kde ji lze použít. Proměnné deklarované v bloku mají obor platnosti rovný danému bloku; pouze v rámci složených závorek tohoto bloku k nim lze přistupovat a za jeho koncem „přestanou existovat“. Globální proměnné mají globální obor platnosti a jsou k dispozici v každém místě vašeho programu.

Lokální proměnné

Kromě proměnných předaných funkci ve formě parametrů je možné v těle funkce proměnné také deklarovat. Proměnné deklarované uvnitř těla funkce se nazývají „lokální“ (nebo též „místní“), protože existují pouze lokálně uvnitř funkce samotné. Když funkce vrátí řízení, přestanou být lokální proměnné dostupné. Jsou kompilátorem označeny a určeny ke zničení.

Lokální proměnné se definují jako všechny ostatní proměnné. Za lokální proměnné se považují i parametry předávané funkci a lze je použít stejně, jako kdyby byly definovány uvnitř těla funkce. Ve výpisu 5.2 uvádíme příklad použití parametrů a proměnných lokálně definovaných uvnitř funkce.

Výpis 5.2

Používání lokálních proměnných a parametrů

```
 0: #include <iostream>
 1:
 2: float Prevod(float);
 3: int main()
 4: {
 5:     using namespace std;
 6:
 7:     float TeplotaFer;
 8:     float TeplotaCel;
 9:
10:     cout << "Vlozte prosim teplotu v jednotkach Fahrenheita: ";
11:     cin >> TeplotaFer;
12:     TeplotaCel = Prevod(TeplotaFer);
13:     cout << "\nA zde je teplota v jednotkach Celsia: ";
14:     cout << TeplotaCel << endl;
15:     return 0;
16: }
17:
18: float Prevod(float TeplotaFer)
19: {
20:     float TeplotaCel;
21:     TeplotaCel = ((TeplotaFer - 32) * 5) / 9;
22:     return TeplotaCel;
23: }
```

Výstup

```
Vlozte prosim teplotu v jednotkach Fahrenheita: 212

A zde je teplota v jednotkach Celsia: 100

Vlozte prosim teplotu v jednotkach Fahrenheita: 32

A zde je teplota v jednotkach Celsia: 0

Vlozte prosim teplotu v jednotkach Fahrenheita: 85

A zde je teplota v jednotkach Celsia: 29.4444
```

Analýza Na řádku 7 a 8 se deklarují dvě proměnné typu `float`. Jedna proměnná má obsahovat teplotu v jednotkách Fahrenheita a druhá má obsahovat teplotu ve stupních Celsia. Na řádku 10 je uživatel vyzván, aby zadal teplotu v jednotkách Fahrenheita, a tato hodnota se předá funkci `Prevod()`.

Provádění programu se přesune na první řádek funkce `Prevod()`. Na řádku 20, kde funkce začíná, se také deklaruje lokální proměnná `TeplotaCel`. Stojí za povšimnutí, že tato lokální proměnná není totožná s proměnnou `TeplotaCel` z řádku 8. Lokální proměnná existuje pouze uvnitř funkce `Prevod()`. Hodnota předávaná jako parametr `TeplotaFer` je také pouze lokální kopií proměnné, kterou předává funkce `main()`.

Tato funkce by mohla stejně dobře mít parametr pojmenovaný `FerTeplota` a lokální proměnnou `CelTeplota` a program by fungoval stejně dobře. Můžete tyto názvy změnit a program znovu zkompilovat, abyste se o tom přesvědčili.

Do lokální proměnné `TeplotaCel` funkce `Prevod()` se přiřadí hodnota, kterou dostaneme po odečtení 32 od parametru `TeplotaFer`, následném vynásobení výsledku číslem 5 a vydělení číslem 9. Tato hodnota se pak vrátí jako návratová hodnota funkce a na řádku 12 se ve funkci `main()`přiřadí do proměnné `TeplotaCel`. Na řádku 14 se tato hodnota vytiskne.

Program jsme spustili třikrát. Poprvé jsme zadali hodnotu `212`, abychom si ověřili, že bod varu vody ve stupních Fahrenheita (`212`) se převede na odpovídající hodnotu ve stupních Celsia (`100`). Ve druhém testu jsme správnost programu vyzkoušeli na bodu mrazu vody a ve třetím případě jsme zvolili náhodné číslo, které dá výsledek typu `float`.

Lokální proměnné v blocích

Proměnnou je možné definovat kdekoli uvnitř funkce, nejen na jejím začátku. Obor platnosti takové proměnné pak odpovídá bloku, ve kterém byla definována. Jestliže je proměnná definována uvnitř dvojice složených závorek v rámci funkce, bude tato proměnná dostupná pouze uvnitř tohoto bloku. Program ve výpisu 5.3 nám poslouží jako vhodná ukázka této vlastnosti lokálních proměnných.

Výpis 5.3

Proměnné s oborem platnosti bloku

```
 0: // Výpis 5.3 - demonstruje proměnné
 1: // s oborem platnosti bloku
 2:
 3: #include <iostream>
 4:
 5: void mojeFunkce();
 6:
 7: int main()
 8: {
 9:     int x = 5;
10:     std::cout << "\nV main je hodnota x: " << x;
11:
12:     mojeFunkce();
13:
14:     std::cout << "\nZpatky v main, hodnota x je: " << x;
```

```
15:     return 0;
16: }
17:
18: void mojeFunkce()
19: {
20:     int x = 8;
21:     std::cout << "\nUvnitr mojeFunkce, lokalni promenna x: " << x << std::en-
dl;
22:
23:     {
24:         std::cout << "\nV bloku funkce mojeFunkce, hodnota x je: " << x;
25:
26:         int x = 9;
27:
28:         std::cout << "\nZcela lokalni promenna x: " << x;
29:     }
30:
31:     std::cout << "\nMimo blok, uvnitr mojeFunkce, x: " << x << std::endl;
32: }
```

Výstup

```
V main je hodnota x: 5
Uvnitr mojeFunkce, lokalni promenna x: 8

V bloku funkce mojeFunkce, hodnota x je: 8
Zcela lokalni promenna x: 9
Mimo blok, uvnitr mojeFunkce, x: 8
Zpatky v main, hodnota x je: 5
```

Analýza Na řádku 9 začíná program inicializací proměnné `x` ve funkci `main()`. Tisk na řádku 10 potvrzuje, že proměnná `x` se inicializovala na hodnotu `5`.

Pak dochází k zavolání funkce `mojeFunkce()` a na hodnotu `8` se na řádku 20 inicializuje proměnná, která je také pojmenovaná `x`. Její hodnota se vytiskne na řádku 21.

Na řádku 23 začíná blok a na řádku 24 se znovu vytiskne proměnná `x` této funkce. Na řádku 26 se vytvoří nová proměnná také pojmenovaná `x`, která je v tomto bloku lokální. Hodnota proměnné se inicializuje na `9`.

Na řádku 28 se vytiskne hodnota nejnovější proměnné `x`. Na řádku 29 končí lokální blok a proměnné vytvořené na řádku 26 „vyprší“ platnost a už nebude nadále k dispozici.

Když se na řádku 31 vytiskne proměnná `x`, jedná se o tu proměnnou `x`, která byla deklarována na řádku 20. Tato proměnná `x` zůstala nedotčena proměnnou `x` definovanou na řádku 26 a její hodnota je stále `8`.

Na řádku 32 skončí platnost funkce `mojeFunkce` a její lokální proměnná přestane být dostupná. Provádění programu se vrátí na řádek 14 a vytiskne se hodnota lokální proměnné `x` vytvořené na řádku 9. Tato proměnná zůstala oběma proměnnými definovanými ve funkci `mojeFunkce()` nedotčena.

Asi není třeba uvádět, že tento program by byl mnohem přehlednější, kdyby všechny tři proměnné měly různé názvy.

Parametry jako lokální proměnné

Argumenty předané funkci jsou v této funkci lokální. Provedené změny hodnot argumentů nebudou mít vliv na hodnoty ve volající funkci. Jedná se o předávání hodnotou, což znamená, že se ve funkci vytvoří lokální kopie každého argumentu. S těmito lokálními kopiemi se pak zachází jako s ostatními lokálními proměnnými. Názorně to ilustruje program ve výpisu 5.4.

Výpis 5.4

Ukázka předání hodnotou

```
// Výpis 5.4 - demonstruje předání hodnotou

#include <iostream>

void zamena(int x, int y);

int main()
{
     int x = 5, y = 10;

     std::cout << "Funkce main. Pred zamenou, x: " << x << " y: " << y << "\n";
     zamena(x,y);
     std::cout << "Funkce main. Po zamene, x: " << x << " y: " << y << "\n";
     return 0;
}

void zamena (int x, int y)
{
     int pom;

     std::cout << "Funkce zamena. Pred zamenou, x: " << x << " y: " << y << "\n";

     pom = x;
     x = y;
     y = pom;

     std::cout << "Funkce zamena. Po zamene, x: " << x << " y: " << y << "\n";
}
```

Výstup

```
Funkce main. Pred zamenou, x: 5 y: 10
Funkce zamena. Pred zamenou, x: 5 y: 10
Funkce zamena. Po zamene, x: 10 y: 5
Funkce main. Po zamene, x: 5 y: 10
```

Analýza Tento program inicializuje ve funkci `main()` dvě proměnné a pak je předá funkci `zamena()`, jejímž úkolem je zaměnit je. Když se pak ve funkci `main()` zkontrolují jejich hodnoty, jsou nezměněny.

Na řádku 8 se inicializují proměnné a jejich hodnoty se zobrazí na řádku 10. Zavolá se funkce `zamena()` a proměnné se jí předají.

Provádění programu se přesune dovnitř funkce `zamena()`, kde se na řádku 20 hodnoty znovu vytisknou. Jak se očekávalo, jsou stejné, jaké byly ve funkci `main()`. Na řádcích 22 až 24 se hodnoty zamění a tuto akci potvrdí i tisk na řádku 26. Skutečně, během funkce `zamena()` se obě hodnoty vymění.

Provádění programu se vrátí na řádek 12 zpět do funkce `main()`, kde už hodnoty vyměněny nejsou.

Asi jste už přišli sami na to, že argumenty se předávají funkci `zamena()` hodnotou, což znamená, že se vytvoří kopie hodnot, které jsou ve funkci `zamena()` lokální. Tyto lokální proměnné se na řádcích 22 až 24 zamění, proměnných ve funkci `main()` se to ale nijak nedotkne.

V dni 8. – „Ukazatele“ – a dni 10. – „Pokročilé funkce“ – se seznámíte s jinými možnostmi než předávání hodnotou, které umožňují změnit hodnoty i ve funkci `main()`.

Globální proměnné

Proměnné definované vně jakékoli funkce mají globální obor platnosti, a jsou proto dostupné z každé funkce v programu, včetně funkce `main()`.

Lokální proměnné, které mají stejný název jako globální proměnné, nemají na globální proměnné vliv. Lokální proměnná se stejným názvem jako existující globální proměnná tuto globální proměnnou *skryje* neboli potlačí. Jestliže má funkce proměnnou se stejným názvem jako globální proměnná, vztahuje se tento název, pokud se používá uvnitř funkce, k lokální proměnné, nikoli ke globální. Ve výpisu 5.5 je toto tvrzení ilustrováno na příkladě.

Výpis 5.5

Ukázka globálních a lokálních proměnných

```
 0: #include <iostream>
 1: void mojeFunkce();            // prototyp
 2:
 3: int x = 5, y = 7;             // globální proměnné
 4: int main()
 5: {
 6:     using std::cout;
 7:
 8:     cout << "x ve funkci main: " << x << "\n";
 9:     cout << "y ve funkci main: " << y << "\n\n";
10:     mojeFunkce();
11:     cout << "Navrat z mojeFunkce!\n\n";
12:     cout << "x ve funkci main: " << x << "\n";
13:     cout << "y ve funkci main: " << y << "\n";
14:     return 0;
15: }
16:
17: void mojeFunkce()
18: {
19:     using std::cout;
```

```
20:
21:     int y = 10;
22:
23:     cout << "x ve funkci mojeFunkce: " << x << "\n";
24:     cout << "y ve funkci mojeFunkce: " << y << "\n\n";
25: }
```

Výstup

```
x ve funkci main: 5
y ve funkci main: 7

x ve funkci mojeFunkce: 5
y ve funkci mojeFunkce: 10

Navrat z mojeFunkce!

x ve funkci main: 5
y ve funkci main: 7
```

Analýza Tento jednoduchý příklad ilustruje několik klíčových a potenciálně matoucích okolností týkajících se lokálních a globálních proměnných. Na řádku 3 se deklarují dvě globální proměnné `x` a `y`. Globální proměnná `x` je inicializována hodnotou `5` a globální proměnná `y` je inicializována hodnotou `7`.

Na řádcích 8 a 9 se tyto hodnoty ve funkci `main()` vytisknou na obrazovku. Všimněte si, že funkce `main()` nedefinuje žádnou proměnnou. Vzhledem k tomu, že se jedná o globální proměnné, jsou dostupné i ve funkci `main()`.

Když se na řádku 10 zavolá funkce `mojeFunkce()`, přesune se provádění programu na řádek 17 a na řádku 21 se definuje lokální proměnná `y`, která se zde inicializuje na hodnotu 10. Na řádku 23 vytiskne funkce `mojeFunkce()` hodnotu proměnné `x` a v tomto případě se používá globální proměnná `x` přesně tak, jako tomu bylo ve funkci `main()`. Když se však na řádku 24 používá název proměnné `y`, jedná se o lokální proměnnou, která potlačí globální proměnnou se stejným názvem.

Volání funkce skončí a řízení programu se vrátí funkci `main()`, která opět vytiskne hodnoty v globálních proměnných. Všimněte si, že globální proměnná `y` zůstala zcela nedotčená hodnotou, která byla ve funkci `mojeFunkce()` přiřazena lokální proměnné `y`.

Globální proměnné: na co si dávat pozor

V C++ jsou globální proměnné povolené, ale téměř nikdy se v tomto jazyce nepoužívají. C++ vychází z jazyka C a v něm jsou globální proměnné nezbytným, ale i nebezpečným nástrojem. Pro programátora je někdy tento typ proměnné nepostradatelný, protože může potřebovat data, která jsou dostupná mnoha funkcím a jež zároveň nechce předávat jako parametr z funkce do funkce.

Nebezpečí globálních proměnných spočívá ve společném sdílení dat. Jedna funkce může totiž změnit globální proměnnou takovým způsobem, že to zůstane jiné funkci skryté. To pak může mít za následek chyby, které se jen velmi těžko odhalují.

V den 15. – „Speciální třídy a funkce“ se dozvíte, že existuje velmi silný nástroj, který může sloužit jako rovnocenná alternativa globálních proměnných. Jedná se o statické členské proměnné.

Příkazy ve funkcích

Neexistují prakticky žádné meze pro počet nebo typy příkazů, které je možné umístit do těla funkce. Ačkoli není možné uvnitř jedné funkce definovat jinou funkci, je možné jinou funkci zavolat. Například funkce `main()` tak činí téměř v každém programu C++. Funkce mohou volat dokonce i samy sebe, což probereme v části této kapitoly věnované rekurzi.

Přestože se na velikost funkce v C++ meze nekladou, dobře navržené funkce bývají krátké. Někdy se doporučuje přizpůsobit velikost funkce velikosti obrazovky, aby bylo možné celou funkci přehlédnout najednou. Jedná se o praktickou zásadu, kterou často velmi dobří programátoři porušují. Stále však platí, že méně rozsáhlé funkce se lépe spravují a lépe se v nich orientuje.

Každá funkce by měla provádět jeden úkol, který je dobře srozumitelný. Jestliže se funkce začnou příliš rozrůstat, je dobré najít místa, kde se dají rozdělit na menší podúkoly.

Více o argumentech funkcí

Každý platný výraz C++ může být argumentem funkce, včetně konstant, matematických a logických výrazů a ostatních funkcí, které vrací hodnotu. Důležité je, aby výsledek výrazu odpovídal typu argumentu, jak jej daná funkce očekává.

Ačkoli je dovolené, aby jedna funkce dostala jako argument hodnotu vrácenou jinou funkcí, může dojít k tomu, že kód se stane obtížně čitelný a bude se špatně ladit.

Předpokládejme, že máme funkce `mujDvojnasobek()`, `trojnasobek()`, `naDruhou()` a `naTreti()`. Každá z nich vrací hodnotu. Dalo by se tedy napsat:

```
Odpoved = (mujDvojnasobek(trojnasobek(naDruhou(naTreti(mojeHodnota)))));
```

Uvedený příkaz lze chápat dvěma způsoby. Jednak je vidět, že funkce `mujDvojnasobek()` přebírá jako argument funkci `trojnásobek()`. Ta zase přebírá výstup funkce `naDruhou()`, jejímž argumentem je funkce `naTreti()`. Až funkce `naTreti()` přebírá jako svůj argument proměnnou `mojeHodnota`.

Z opačného pohledu uvidíme, že tento výraz si vezme proměnnou `mojeHodnota` a předá ji jako argument funkci `naTreti()`, jejíž vrácená hodnota se předá jako argument funkci `naDruhou()`, jejíž hodnota se pak předá funkci `trojnasobek()` a její vrácená hodnota se zase předá funkci `mujDvojnasobek()`. Vrácená hodnota čísla, které se nejprve umocnilo na třetí, pak na druhou, vynásobilo třemi a pak dvěma, se přiřadí do proměnné `Odpoved`.

U tohoto kódu si budeme jen velmi málo jisti v tom, co se přesně děje (vynásobila se hodnota před nebo až poté, co byla umocněna na druhou?), a jestliže bude odpověď špatně, jen obtížně rozhodneme, která funkce nefunguje.

Alternativou je přiřadit v každém kroku hodnotu do zvláštní průběžné proměnné:

```
unsigned long mojeHodnota = 2;
unsigned long tretiMocnina = naTreti(mojeHodnota);      // tretiMocnina = 8
unsigned long druhaMocnina = naDruhou(tretiMocnina);    // druhaMocnina = 64
unsigned long trikrat = trojnasobek(druhaMocnina);      // trikrat = 192
unsigned long Odpoved = mujDvojnasobek(trikrat);        // Odpoved = 384
```

Každou průběžnou proměnnou je možné zkontrolovat a pořadí provádění je dáno explicitně.