Jeff Kent

C++
bez předchozích znalostí

Computer Press, a. s.
Brno
2009

C++
bez předchozích znalostí

Jeff Kent

Computer Press, a.s., 2009. Vydání první.

Překlad: Tomáš Znamenáček
Odborná korektura: Matěj Dusík
Jazyková korektura: Petra Láníčková
Sazba: Vladimír Ludva
Rejstřík: Daniel Štreit
Obálka: Lance Lekander, Martin Sodomka

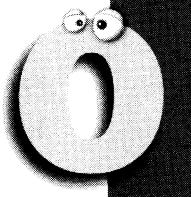
Komentář na zadní straně obálky: Martin Domes
Technická spolupráce: Jiří Matoušek, Zuzana Šindlerová, Dagmar Hajdajová
Odpovědný redaktor: Martin Domes
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Computer Press, a.s.,
Holandská 8, 639 00 Brno

Objednávky knih:
http://knihy.cpress.cz
distribuce@cpress.cz
tel.: 800 555 513

ISBN 978-80-251-2411-6
Prodejní kód: K1370
Vydalo nakladatelství Computer Press, a.s., jako svou 3256. publikaci.

Obsah



O autorovi

Jeff Kent vyučuje informatiku na Los Angeles Valley College v kalifornském Valley Glen. Přednáší více programovacích jazyků, například Visual Basic, C++, Javu nebo – když má zrovna masochistickou chvilku – assembler, ale většinou učí C++. Kromě toho spravuje síť jedné losangeleské právnické firmy, jejíž zaměstnanci slouží jako pokusní králíci pro jeho aplikace, a jako advokát udílí rady mladším advokátům (ať se jim to líbí, nebo ne). Je autorem několika knih o programování, jedním z jeho posledních titulů je *Visual Basic.NET: A Beginner's Guide* pro nakladatelství McGraw-Hill/Osborne.

Jeffova profesní dráha je pestrá – přesněji řečeno, jeho profesní dráhy jsou pestré. Promoval na UCLA jako bakalář ekonomie, pak vystudoval právo na losangeleské Loyola School of Law a pustil se do právnické praxe. Během této doby (kdy se o počítačích tak nanejvýš zdálo panu Gatesovi) se Jeff živil také jako šachista; získal třetí místo na mistrovství Spojených států do 21 let a později i mezinárodní titul.

I tak si najde čas pro svou ženu Devvie, což není zas tak těžké, protože i ona přednáší informatiku na Valley College. Zároveň Jeff plní úlohu osobního řidiče své mladší dcery Emily (ta starší, Elise, už má svůj vlastní řidičský průkaz) a ve zbývajícím volném čase si užívá přenosy mezinárodních šachových turnajů na Internetu. Jeho životním cílem je začít zase běhat maratony, protože jinak jeho příští kniha – vzhledem k jeho neúspěšné bitvě s nadváhou – pravděpodobně ponese titul *Sumo bez záhad.*

Tuto knihu bych chtěl věnovat své ženě Devvie Schneider Kentové. V osobním i profesním životě mi dala tolik, že by to vydalo na samostatnou knihu (jejíž malou část najdete v Poděkováních). Kromě jiného je i má učitelka programování – bez ní bych tuto ani žádnou jinou knihu o programování nikdy nenapsal.

—Jeff Kent

Poděkování

Poděkování autora vydavateli vypadá jako povinnost (zvlášť pokud chce autor pro vydavatele ještě někdy napsat nějakou knihu), ale já ho myslím upřímně. Toto je má čtvrtá kniha pro nakladatelství McGraw-Hill/Osborne a já doufám, že ještě přijde mnoho dalších. Opravdu mne těší spolupracovat s profesionály, kteří jsou zároveň příjemní lidé a zároveň velice dobří v tom, co dělají – i kdyby to zrovna mělo být pečlivé sledování všech termínů, které se mi podařilo nedodržet.

Jako první bych chtěl poděkovat Wendy Rinaldiové, která mě do vydavatelství McGraw-Hill/Osborne kdysi v roce 1998 uvedla (vážně už je to tak dávno?). I tato kniha ostatně začala telefonickým hovorem s Wendy. Zrovna nám se ženou končila dovolená a Devvie, která byla na doslech od telefonu, se mě nevěřícím tónem vyhrazeným potenciálním šílencům zeptala, jestli *skutečně hodlám psát další knihu.*

Také musím poděkovat své akviziční koordinátorce Atheně Honoreové a projektové redaktorce Lise Wolters-Broderové. Obě byly nekonečně trpělivé, vždy ochotné mi pomoci a zároveň dbaly, abych se neodchýlil od stanovených termínů. (Které se v oboru hodně drží. „Je nám doopravdy líto, že jste si zlomil obě ruce a nohy; ale do pátku nám prosím odevzdejte další kapitolu, ano?")

Redaktorské práce provedl Mike McGee společně s Lisou. Oba projevili velkou shovívavost k výpadkům, které zjevně musely doprovázet má školní léta. Text vylepšili, aniž by se odchýlili od jeho původního významu, takže jakékoliv chyby padají na mou hlavu. Mike navíc naznačil svou slabost pro některé z mých otřepaných vtipů, čímž si mě navěky získal.

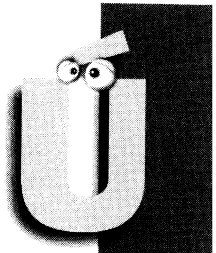
Technickým redaktorem byl Jim Keogh. Mezi mnou a Jimem panoval vyvážený vztah vzájemného ohrožení – zatímco on byl technickým redaktorem mé knihy, já jsem byl technickým redaktorem dvou jeho knih *Data Structures Demystified* a *OOP Demystified*. Ale vážně: Jimovy poznámky pomohly mně i knize.

Při vydání knihy asistovala řada dalších talentovaných lidí, ale jak se říká během přebírání Oskarů, všechny je tu vyjmenovat nemůžu. To neznamená, že bych si nevážil jejich práce – vážím.

Upřímně děkuji své ženě, která je i můj nejlepší (ne-li jediný) kamarád a parťák. (Jako nejlepší milenku ji uvést nemůžu, protože programátoři se přeci o takové věci nezajímají.) Kromě jiného byla i můj technický redaktor. Má pro takovou práci kvalifikaci, protože už patnáct let učí informatiku, a navíc je pedant na jazyky (ano, já vím, slovo *optimální* se nestupňuje). Moc této knize pomohla.

A konečně si můj dík zaslouží mé dcery Elise a Emily a moje maminka Bea Kentová za svou shovívavost, když jsem se omlouval z rodinných setkání a něco si mumlal o nepřiměřených termínech a ukrutných redaktorkách (pardon, Atheno a Liso). Celé své rodině bych chtěl předem poděkovat za to, že mě nezbaví svéprávnosti, až začnu uvažovat o další knize.

Úvod



C++ byl můj první programovací jazyk. I když jsem se od té doby naučil další, C++ jsem vždycky považoval za „nejlepší"; nejspíš kvůli tomu, jakou moc programátorům dává. Tato moc je samozřejmě dvojsečná zbraň, kterou si při troše neopatrnosti můžete uříznout vlastní větev. Přesto mám C++ ze všech jazyků nejraději.

Nejsem sám, kdo má C++ rád. Kromě komerční sféry, které C++ nabízí svou velkou sílu, je oblíbené i v akademickém světě. Navíc z něj vychází mnoho dalších jazyků, včetně Javy a C#. (Java byla ostatně v C++ napsána.) Znalost C++ tím pádem usnadňuje studium dalších jazyků.

Proč jsem knihu napsal

Nikoliv kvůli bohatství, slávě nebo krásným ženám. (Možná jsem trochu sešel z cesty, ale trocha zdravého rozumu mi ještě zbyla.) Není pochyb o tom, že úvodů do C++ existuje hodně. Přesto jsem tuto knihu napsal, protože si myslím, že mohu nabídnout jinou, a doufám že užitečnou perspektivu.

Jak možná víte z mého životopisu, učím informatiku na Los Angeles Valley College – vyšší odborné škole ze San Fernando Valley v Los Angeles, kde jsem vyrostl a prožil většinu svého života. Dělám i programátora, ale výuka programování mě o programování naučila věci, které bych se v praxi jinak nedozvěděl. Nejde jen o zodpovídání studentských dotazů během lekcí. Každý týden strávím hodiny v naší počítačové učebně a pomáhám studentům s jejich programy; každý týden strávím hodiny opravováním a známkováním jejich úkolů. Postupem času se ukázalo, jaký přístup doopravdy funguje, v jakém pořadí se mají probírat jednotlivá témata, na jaké úrovni složitosti se k nim poprvé dostat a podobně. Z legrace svým studentům říkám, že jsou beta testeři mých věčných pokusů o lepší výuku, ale v tom vtipu je velký kus pravdy.

Moji bet – studenti si navíc věčně stěžují na učebnici, ať už vyberu jakoukoliv. Hodně z nich se ptá, proč nějakou nenapíšu sám. Možná se mi jen snaží lichotit (neříkám, že to nefunguje), možná by mě kromě mizerné výuky rádi popotahovali i za mizernou učebnici. Protože jsem už ale několik knih napsal, jejich otázky mi do hlavy nasadily nápad na knihu, která by kromě široké veřejnosti posloužila i jako doplněk k učebnici.

Komu je kniha určena

Komukoliv, kdo zaplatí. Dělám si legraci. (Ale žádnému kupci neřeknu ne!)

Vydavatelé i autoři chtějí pro každou knihu získat co nejširší publikum, to není žádná novinka. Tato část knihy proto většinou vysvětluje, že kniha je přesně pro vás, ať už jste kdokoliv a děláte cokoliv. Žádná programátorská kniha ale není pro každého. Pokud například programujete výhradně hry v Javě, tato kniha vám nejspíš moc nepomůže. (I když jako učitel bych mohl být váš další zákazník, neměli byste zájem o titul *Učitelé versus vetřelci z vesmíru*?)

Takže i když tato kniha samozřejmě není pro každého, pro vás by docela dobře být mohla. C++ se chce nebo musí naučit hodně lidí, ať už v rámci univerzitního studia, pracovního školení nebo

třeba i z vlastního zájmu. C++ není úplně nejjednodušší téma. Řada autorů ho navíc moc neusnadňuje, protože před vás položí komplikovaný telefonní seznam plný cizích výrazů. Tato kniha se vám naproti tomu snaží vysvětlit C++ bez záhad, jak už napovídá její titul. Jde rovnou k základním pojmům a vysvětluje je pěkně po pořádku, hezky česky.

Co v knize najdete

Jsem pevným zastáncem myšlenky, že programovat se člověk naučí nejlépe programováním. Proto jsou myšlenky jednotlivých kapitol ilustrované jasně a podrobně vysvětleným kódem. Můžete si ho sami spustit a můžete nad ním postavit programy, na kterých si popisované myšlenky vyzkoušíte podrobněji.

První kapitola vás dostane do tempa – například vám vysvětlí, co je počítačový program a co programovací jazyk. Pak popisuje anatomii základního programu v C++, a to včetně „dění za oponou", tedy jak preprocesor společně s překladačem a linkerem přeloží váš kód do podoby srozumitelné počítači. Nakonec vám první kapitola ukáže, jak vytvořit a spustit projekt v integrovaném vývojovém prostředí (IDE).

Schopnost napsat a spustit program, který na obrazovku vypíše `Hello World`, je dobrý začátek. Většina programů ale potřebuje pracovat s daty, například čísly a textem. Druhá kapitola proto popisuje různé typy počítačové paměti, včetně paměti s náhodným přístupem neboli RAM. Následuje diskuse o adresách, které popisují umístění dat v RAM, a bajtech neboli jednotkách potřebných pro uložení informací. A protože informace mívají různou podobu, kapitola se dále věnuje různým datovým typům pro celá čísla, desetinná čísla a text.

Hlavní hvězdou třetí kapitoly je proměnná, která nejenže rezervuje místo v paměti pro uložení informací, ale navíc vám poskytne jméno, pod kterým můžete s těmito informacemi pracovat. Úkolem proměnných je ukládat informace, takže proměnná bez přiřazené hodnoty je užitečná zhruba stejně jako banka bez peněz. Proto kapitola dále vysvětluje, jak se proměnným přiřazují hodnoty – ať už během překladu pomocí operátoru přiřazení, nebo za běhu pomocí objektu `cin` a operátoru čtení z proudů (>> a <<).

Jako bývalého profesionálního šachistu mě vždycky ohromoval fakt, že šachový počítač může sehrát vyrovnanou partii se světovým šampiónem. Počítače těží ze své schopnosti provádět výpočty mnohem rychleji a přesněji než člověk. Aritmetické operátory, díky kterým můžeme jejich výpočetní kapacitu využít i my, popisuje čtvrtá kapitola.

Aby mohl program zvládat i složitější úkoly, musí mít možnost měnit průběh výpočtu podle pravdivosti určité podmínky. Kdybyste například využili aritmetické operátory ze čtvrté kapitoly a naprogramovali kalkulačku, konkrétní aritmetická operace počítaná vaším programem by se lišila podle toho, jestli uživatel vybral sčítání, odčítání, násobení, nebo dělení. V páté a šesté kapitole se proto podíváme na relační a logické operátory, které se hodí při rozhodování, a příkazy `if` a `switch`, kterými se dá na základě tohoto rozhodování měnit průběh výpočtu.

Malé děti své rodiče někdy trápí tím, že něco opakují neustále dokola. Občas je potřeba opakovat i kus kódu. Když například uživatel aplikaci zadá chybná data, můžete ho dokola prosit o nové zadání, dokud vám data nezadá správně nebo dokud program neukončí. Hlavním předmětem sedmé a osmé kapitoly jsou takzvané smyčky, které slouží k opakovanému provádění kódu po dobu platnosti nějaké podmínky. Sedmá kapitola se zabývá smyčkou `for` a zároveň vám ukáže operátory zvýšení a snížení hodnoty proměnné, které se ve smyčkách často používají. Osmá kapitola doplní diskusi smyčkami `while` a `do while`.

Devátá kapitola je o funkcích. Funkce je blok jednoho nebo více příkazů. Většina kódu, který v C++ napíšete, bude součástí nějaké funkce. Tato kapitola vám vysvětlí, proč se kód dělí do funkcí a jak se funkce dělají. Ukáže vám, jak se píše prototyp funkce, jak se funkce definuje a jak se volá. Také se dozvíte, jak pomocí parametrů předat funkci informace a jak informace z volané funkce vrátit. Vysvětlený bude i rozdíl v předávání parametrů hodnotou a odkazem. Nakonec si vysvětlíme rozsah platnosti proměnných, životnost proměnných a rozdíl mezi lokálními, statickými a globálními proměnnými.

Desátá kapitola je věnovaná polím. Od proměnných popisovaných v předchozí části knihy se pole liší v tom, že mohou najednou obsahovat větší počet hodnot. Často se používají ve spojení se smyčkami, o kterých se mluví v kapitolách sedm a osm. Mimo jiné se podíváme na rozdíly mezi polem znaků a polem jiných datových typů. Nakonec probereme konstanty, které se v mnohém podobají proměnným, ale jejichž hodnota se po dobu běhu programu nemění.

Jedenáctá kapitola je o ukazatelích. Při zaslechnutí slova ukazatel se mnohým adeptům C++ ježí chlupy na zádech, ale zbytečně. Jak už jsme si říkali v souvislosti s druhou a třetí kapitolou, informace se v paměti ukládají na nějakou adresu. Ukazatele jsou jednoduše efektivní nástroj pro práci s těmito adresami. V jedenácté kapitole se také dozvíte o operátoru hvězdička (`*`), dereferencování a ukazatelové aritmetice.

Hodně informací bývá uložených v podobě znaků, céčkových řetězců nebo řetězcových tříd C++. Kapitola dvanáct se proto věnuje užitečným funkcím pro práci s těmito datovými typy, například členským funkcím třídy `cin`.

Aby informace zůstaly k dispozici i po skončení programu, často se ukládají do souborů. Kapitola třináct vám vysvětlí práci se souborovými proudy, tedy třídami `fstream`, `ifstream` a `ofstream` a jejich členskými funkcemi `open`, `read`, `write` a `close`.

A konečně abyste měli solidní základy i pro další studium po absolvování této úvodní knihy, uvede vás čtrnáctá kapitola do objektově orientovaného programování (OOP) a dvou konceptů, které se v něm často používají: struktur a tříd.

Za každou kapitolou najdete test, kterým si můžete ověřit, jestli jste základní pojmy kapitoly bezpečně zvládli. (Na rozdíl od školních testů budete mít v druhém dodatku i odpovědi.) V prvním dodatku na vás čeká velká závěrečná prověrka; i její řešení najdete v druhém dodatku.

Jak knihu číst

Psal jsem knihu tak, aby se dala číst od začátku do konce. Na první pohled se asi zdá, že to jinak ani nejde. Od svých studentů ale často slýchám oprávněné stížnosti na to, že se učitel nebo kniha během vysvětlování myšlenky opírají o pojmy, které jsou vysvětlené až o několik kapitol dál (nebo v horším případě vůbec ne). Proto jsem se důsledně snažil postupovat lineárně, logicky. Jednak se vyhnete frustracím z textu, který je třeba číst napřeskáčku, a jednak můžete v každé kapitole stavět na výsledcích té předchozí.

Speciality

Každou kapitolu doprovází poznámky, tipy, upozornění na problematická místa a výpisy kódu. Abyste měli lepší zpětnou vazbu, najdete za každou kapitolou malý test a za celou knihou pak v prvním dodatku závěrečnou zkoušku. Odpovědi k oběma jsou v druhém dodatku. Hlavním cílem knihy je, abyste se rychle dostali do tempa, bez zbytečné suché teorie a nadbytečných podrobností. Tak pojďme na to. Programování v C++ není těžké a je to zábava.

Kontakt na autora

K čemu? (No dobře.) Nadšené ovace a lichotky nikdy neublíží, ale uvítám i komentáře, rady a dokonce snad i kritiky. Nejlepší bude, když mi napíšete e-mail na adresu *jkent@genghiskhent.com* (doména je nazvaná podle přezdívky, kterou jsem dostal od svých studentů). Případně se můžete podívat na mé webové stránky *www.genghiskhent.com*. Nenechte se zmást titulní stránkou – web slouží především jako zdroj informací pro mé přednášky, ale je na něm i odkaz týkající se této knihy.

Doufám, že si knihu užijete stejně, jako jsem si já užil její psaní.

Poznámka redakce českého vydání

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 PC 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/k1370*. Prostřednictvím uvedené adresy můžete též naší redakci zaslat komentář nebo dotaz týkající se knihy. Na vaše reakce se srdečně těšíme.

Jak funguje program napsaný v C++

S počítačovými programy se během průměrného dne potkáte mnohokrát. Když dorazíte do práce a zjistíte, že vám nefunguje počítač, zavoláte technickou podporu. Na druhém konci telefonní linky je počítačový program, který vás donutí prokličkovat bludištěm hlasového systému a během nekonečného čekání vás mučí neupřímnými vzdechy o tom, jak si vašeho hovoru váží a jak už vás dozajista brzy přepojí.

Když pak skoncujete s technickou podporou, rozhodnete se přihlásit k opravenému počítači a rozdat si to s gigantickými hmyzáky z planety Megazoid. Správce sítě vás bohužel nachytá na švestkách, a to díky dalšímu počítačovému programu pro sledování počítačů. Vaši výplatní pásku, pokud kdy ještě nějakou uvidíte, sestaví opět program.

Na cestě domů si uvědomíte, že potřebujete nějakou hotovost. Zastavíte se tedy u bankomatu, kde vám počítačový program – doufejme – potvrdí, že máte dost peněz na účtu, a nakáže bankomatu vyplatit hotovost, načež – bohužel – tutéž částku strhne z vašeho účtu.

Většina lidí se během tohoto každodenního setkávání s počítačovými programy vůbec nemusí zajímat, jak vlastně počítačový program vypadá nebo funguje. Počítačový programátor by to ale vědět měl – stejně jako řadu dalších souvisejících věcí, například co je to programovací jazyk nebo jak vlastně funguje program napsaný v C++. Až dočtete tuto kapitolu, budete odpovědi na podobné otázky znát a budete umět napsat a spustit svůj vlastní počítačový program.

Co je počítačový program

Počítače jsou v naší společnosti tolik rozšířené, protože mají oproti nám lidem tři výhody. Za prvé zvládají uložit ohromné množství informací, za druhé si tyto informace umí rychle a přesně vybavit, a za třetí umí rychlostí blesku provádět přesné výpočty.

Tyto výhody se přenáší i do intelektuálních sportů, například do šachu. V roce 1997 porazil počítač Deep Blue světového šachového šampióna Garry Kasparova. V roce 2003 se chtěl Kasparov revanšovat jinému počítači nazvanému Deep Junior, ale pouze remízoval. Přestože je Kasparov pravděpodobně vůbec nejlepší šachista všech dob, je pouze člověk, a tím pádem se nemůže měřit s rychlostí a kapacitou počítačů.

My ale máme oproti počítačům jednu zásadní výhodu, alespoň zatím: umíme samostatně přemýšlet. Počítač není chytrý, je jen rychlý. Cokoliv provádí, dělá jen na základě podrobného návodu,

počítačového programu. Autorem tohoto programu je samozřejmě člověk, programátor. Počítačové programy nám umožňují využít obrovskou výpočetní kapacitu počítače.

Co je programovací jazyk

Když vejdete do tmavé místnosti a chcete se rozhlédnout kolem, zapnete světlo. Když odcházíte, světlo vypnete.

První počítače se podobaly právě takovému vypínači. Obsahovaly řadu přepínačů a drátů a elektrický proud jimi procházel po různých cestách podle toho, které vypínače byly právě zapnuté a vypnuté. Sám jsem si takový jednoduchý počítač postavil, když jsem byl malý (což v představách mých dětí bývalo ještě v dobách, kdy se po Zemi prohánělí dinosauři).

Stav každého z vypínačů se dá vyjádřit číslem: jednička je vypínač zapnutý, nula vypínač vypnutý. Pokyny pro první počítače, tedy stav jejich jednotlivých vypínačů, byly proto v podstatě řady nul a jedniček. Dnešní počítače jsou samozřejmě mnohem výkonnější a složitější. Takzvaný strojový jazyk (tedy jazyk, kterému počítače rozumí) se ale nezměnil – pořád jde o nuly a jedničky.

Zatímco počítače přemýšlí v nulách a jedničkách, lidé obyčejně ne. Složitý program se navíc může skládat z milionů strojových příkazů, takže by jeho psaní zabralo neúnosně dlouhou dobu. To je důležitý faktor, protože díky tlaku konkurence se programy musí psát rychleji a rychleji.

My naštěstí programy ve strojových příkazech psát nemusíme. Můžeme místo nich použít příkazy *programovacího* jazyka, které jsou nám mnohem srozumitelnější, protože jsou bližší lidskému vyjadřování než nulám a jedničkám. V programovacím jazyce se navíc programuje mnohem rychleji, protože má vyšší vyjadřovací schopnost – jeden příkaz programovacího jazyka vydá za řadu příkazů jazyka strojového.

C++ je jedním z mnoha programovacích jazyků. Mezi další oblíbené programovací jazyky patří Java, C# nebo VisualBasic. Existuje řada dalších a neustále vznikají nové. Všechny programovací jazyky ale mají v podstatě tentýž účel – umožnit lidem vysvětlit počítači, co přesně má dělat.

Proč byste měli dát C++ přednost před jinými jazyky? Za prvé je hojně rozšířené, mezi firmami i na školách. Za druhé z něj vychází hodně dalších jazyků (například Java nebo C#); Java v něm byla přímo napsána. Když budete umět C++, bude pro vás snazší zvládnout další programovací jazyky.

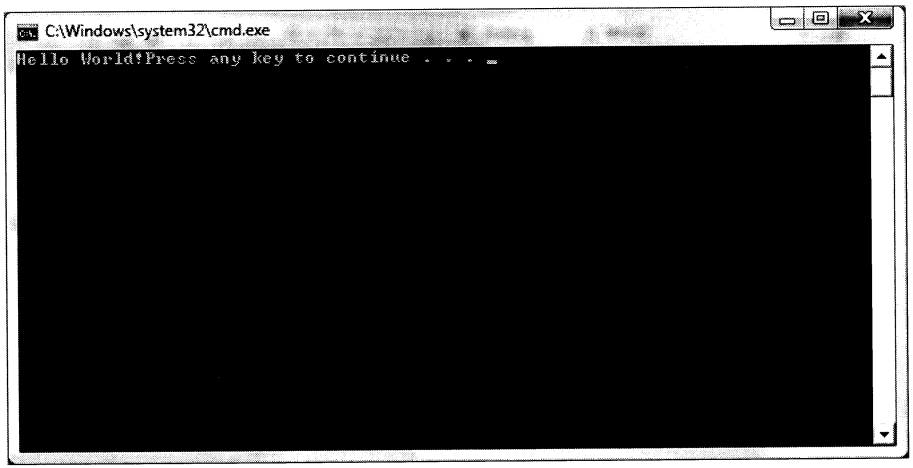
Anatomie programu v C++

Knihy o programování v C++ většinou dodržují nepsanou tradici a jako první ukázku kódu svým čtenářům předloží program, který na obrazovku vypíše zprávu „Hello, world!" (viz obrázek 1.1).

Poznámka: Text `Press any key to continue` vypsaný těsně za `Hello, world!` není součástí programu. Jen vám napovídá, jak zavřít příkazový řádek.

Za tímto prvním příkladem většinou bohužel rychle následují další, aniž by se kniha nebo učitel na chvilku zastavili a vysvětlili, jak vlastně *Hello world* funguje. Výsledkem je dezorientovaný student nebo čtenář, který je připravený se s krutým světem C++ zase hodně rychle rozloučit.

Hello world sice vypadá jednoduše, ale za oponou se toho děje hodně. Proto si teď program projdeme řádku po řádce, byť napřeskáčku.



Obrázek 1.1: Program v C++ vypisuje na obrazovku zprávu „Hello, world!"

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << "Hello, world!";
    return 0;
}
```

Poznámka: Kód, který píše programátor, se většinou označuje jako *zdrojový kód*. Ukládá se do souboru, který má v případě C++ příponu .cpp (cé plus plus).

Funkce main

Na začátku kapitoly jsme si řekli, že v C++ i všech ostatních programovacích jazycích zadává programátor příkazy počítači. Úloha, kterou má počítač provést, se většinou nevejde do jednoho jediného příkazu; bývá potřeba několik souvisejících příkazů. *Funkce* je skupina právě takových souvisejících příkazů, které společně slouží k provedení nějakého úkolu.

Každá funkce má jméno, kterým se na její příkazy můžete odvolávat. Výraz `main` v našem zdrojovém kódu *Hello world* je název funkce. Programy mívají funkcí víc (psaním funkcí se bude zabývat devátá kapitola), ale každý program v C++ má jen jednu a právě jednu hlavní funkci jménem `main`. Tato funkce je vstupním bodem programu – kdyby ji program neměl, počítač by nevěděl, kde program začít provádět. A kdyby naopak program obsahoval více než jednu funkci `main`, počítač by nevěděl, kterou z nich si má vybrat.

Poznámka: Významem slova `int` před názvem funkce `main` a závorkami se slovem `void` se budeme zabývat v deváté kapitole.

Tělo funkce

Všechny příkazy funkce `main` jsou uvedené v takzvaném *těle* funkce, které začíná levou složenou závorkou `{` a končí pravou složenou závorkou `}`.

Většina příkazů končí středníkem. V naší funkci `main` máme příkazy dva:

```
cout << "Hello, world!";
return 0;
```

Příkazy se provádí popořadě, odshora dolů. Prvním z nich je:

```
cout << "Hello, world!";
```

`cout` je objekt, který slouží pro výstup dat (proto *out* jako *ven*). Patří mezi takzvané *datové proudy*, které se dělí na dvě skupiny. První skupinu tvoří výstupní datové proudy, které slouží k výstupu dat z programu, například na monitor, na tiskárnu nebo do souboru. Druhou skupinu tvoří vstupní datové proudy, které řeší načítání dat do programu, například ze souboru nebo klávesnice.

`cout` je výstupní datový proud, který posílá data na standardní výstupní zařízení. V běžných případech je standardním výstupním zařízením monitor, ale můžete ho přesměrovat i na tiskárnu nebo soubor na pevném disku.

Konstrukce `<<` za `cout` je operátor. S nějakými operátory už jste se asi setkali – například aritmetickými operátory `+`, `-`, `*` a `/`, které slouží ke sčítání, odčítání, násobení a dělení. Operátor `<<` slouží ke vkládání dat do proudu. Vezme informace ze své pravé strany (v tomto případě text `Hello, world!`) a vloží je do proudu na své levé straně, což je v tomto případě standardní výstupní zařízení a tedy nejčastěji monitor.

Poznámka: Ve třetí kapitole se dozvíte o protějšku k objektu `cout`. Jmenuje `cin` a obsluhuje standardní vstup; používá se ve spojení s operátorem `>>`.

Druhý a poslední příkaz vrací operačnímu systému (ať už Windows, Unixu nebo čemukoliv jinému) chybový kód nula. Podle něj operační systém pozná, že program skončil úspěšně. Kdyby program skončil neúspěšně, například kdyby vyčerpal veškerou volnou paměť, vrátil by nenulový chybový kód a operační systém by věděl, že při běhu programu došlo k chybě.

Direktiva include

Váš program „ví", že má začít funkcí `main`, protože toto chování předepisuje jazyk C++. Program funkcí `main` začne, aniž byste kvůli tomu museli psát jediný řádek kódu. Podobně program ví, že objekt `cout` ve spojení s operátorem `<<` vypisuje informace na monitor. Rozhodně jsme kvůli tomu nemuseli psát žádný kód navíc. Objekt `cout` ale na rozdíl od funkce `main` není součástí samotného jazyka C++, patří do takzvané *standardní knihovny*. Standardní knihovna C++ obsahuje řadu souborů, z nichž každý definuje některé běžně používané objekty. Výpis na obrazovku rozhodně mezi běžné činnosti patří – mohli byste si ho sice napsat sami, ale jeho implementace ze standardní knihovny vám šetří starosti s „objevováním Ameriky".

Objekt `cout` už tedy máte k dispozici ve standardní knihovně, ale abyste ho mohli použít, musíte do svého programu vložit příslušný soubor ze standardní knihovny. K tomu slouží direktiva `#include`, za kterou následuje název knihovního souboru. Pokud soubor patří do standardní knihovny (tedy pokud jste si ho nepsali sami, což také můžete), jeho název se uzavírá do špičatých závorek `<` a `>`.

Objekt `cout` je definovaný ve standardním knihovním souboru `iostream`. Písmena „io" v názvu souboru jsou zkratka od slov *input* a *output*, vstup a výstup, a *stream* znamená proud. Pokud chceme používat objekt `cout` ve svém programu, musíme sáhnout do standardní knihovny a vložit z ní do našeho programu soubor `iostream`. K tomu slouží příkaz:

```
#include <iostream>
```

Direktiva `#include` je instrukcí pro *preprocesor* – jeden z programů, které budou zdrojový kód zpracovávat. Po preprocesoru se vašemu kódu bude věnovat ještě překladač a linker, o všech třech programech se podrobně pobavíme v následující části kapitoly. Direktivy preprocesoru se na rozdíl od běžných příkazů neukončují středníkem.

Jmenné prostory

Poslední příkaz, který nám zbývá, je:

```
using namespace std;
```

Každý program obsahuje řadu jmen. Například jméno `main` označuje hlavní funkci, jméno `cout` označuje objekt pro práci se standardním výstupem a podobně. Asi není potřeba zdůrazňovat, že jména musí být do velké míry jedinečná: Kdybyste měli dva objekty jménem `cout`, počítač by při zmínce o objektu `cout` nevěděl, o kterém z nich mluvíte.

Problém je v tom, že větší programy se obvykle skládají z mnoha různých kusů napsaných různými programátory. Když tyto části skládáte dohromady, snadno se může stát, že jméno použité v některé z nich už je zabrané jinou. K oddělení jmen z různých částí programu slouží právě *namespaces* neboli *jmenné prostory*. Ty fungují podobně jako naše příjmení, jen se místo za jméno dávají před něj a oddělují se dvěma dvojtečkami (`::`). Například objekt `cout` patří do jmenného prostoru standardní knihovny, která se jmenuje `std`, a tak se `cout` plným jménem jmenuje `std::cout`.

Když je v běžné řeči zjevné, koho myslíme, obejdeme se i bez příjmení. Stejně tak ve zdrojovém kódu je praktičtější uvádět plná jména jen tehdy, když hrozí nějaká záměna. Proto se můžete s počítačem dohodnout, že kdykoliv najde jméno bez jmenného prostoru, doplní si automaticky nějaký výchozí jmenný prostor. Příkaz `using namespace std` říká, že před každé jméno bez jmenného prostoru si má počítač doplnit `std::`.

Překlad zdrojového kódu

Vy už teď zdrojovému kódu *Hello world* rozumíte, ale počítač ještě ne. Počítač sám o sobě nerozumí C++ ani žádnému jinému programovacímu jazyku. Rozumí pouze svému strojovému jazyku. O překlad zdrojového kódu do spustitelné podoby neboli *binárky*, která už je srozumitelná počítači, se starají tři programy. V pořadí, ve kterém se dostanou k vašemu zdrojovému kódu, jsou to:

1. Preprocesor
2. Překladač
3. Linker

Preprocesor

Preprocesor je program, který se stará o předzpracování (*pre-processing*) zdrojového kódu. Když například narazí na direktivu `#include`, nahradí ji obsahem odkazovaného souboru. V našem případě se odkazujeme na soubor `iostream` ze standardní knihovny, a tak preprocesor na místo původní direktivy vloží obsah tohoto souboru (který mimo jiné obsahuje definici objektu `cout`).

Překladač

Překladač je další program, který vezme předzpracovaný zdrojový kód (tedy kód, ve kterém už preprocesor provedl všechny své náhrady) a přeloží ho do odpovídajících příkazů strojového jazyka. Výsledek uloží do samostatného *objektového souboru* s příponou `.obj` nebo `.o`. Každý jazyk má svůj vlastní překladač, ale základní úkol všech překladačů je v podstatě tentýž: Přeložit příkazy z programovacího jazyka do strojového.

Překladač váš zdrojový kód pochopí a přeloží pouze v případě, že se budete držet syntaxe daného programovacího jazyka. C++ má pevná pravidla pro zápis jednotlivých slov a jejich skládání do větších celků – v tom se nijak neliší od všech ostatních, nejen programovacích jazyků. Když budete mít v zápisu syntaktickou chybu, překladač váš program do strojového jazyka nepřeloží a bude si na chybu stěžovat. Z tohoto pohledu tedy překladač zároveň funguje jako kontrola překlepů a gramatiky.

Linker

V objektovém souboru už jsou sice příkazy strojového jazyka, ale spustit se ještě nedá. Musí se k němu ještě přidat kód z knihoven, které jste použili při psaní programu, například kód pro načítání vstupu z klávesnice nebo výpis na monitor. To za vás udělá linker. Jeho výstupem je spustitelný soubor, který v našem případě vypíše zprávu `Hello, world!`.

Proč tak složitě?

Možná vám teď překlad programu připadne zbytečně složitý. K čemu ukládat objektové soubory, když je stejně používá jen linker? Nebylo by jednodušší mít jeden program, který by ze zdrojového kódu bez větších okolků rovnou vytvořil binárku? Odpověď zní, že by to jednodušší bylo, ale jen pro menší programy.

Zdrojový kód každého alespoň trochu většího programu je hodně dlouhý, od stovek řádků až po desítky milionů. Kdybychom takový zdrojový kód nechali v jednom souboru, měli bychom s ním potíže my i počítač – my bychom se v něm nevyznali a počítači by jeho překlad do strojové podoby trval neúnosně dlouho.

Potřebujeme tedy zdrojový kód rozdělit na víc souborů. Je jasné, že nemá smysl zdrojový kód dělit řekněme po dvou stech řádcích na soubor, to by nám v orientaci moc nepomohlo. Zdrojový kód se dělí podle logických celků – co zdrojový soubor, to nějaká poměrně samostatná součástka programu. Takových zdrojových souborů čili součástek bývají u větších projektů desítky až stovky.

Jak bude probíhat překlad takto rozděleného kódu? Samozřejmě by šlo jednotlivé soubory pospojovat do jednoho velkého a ten pak přeložit. Tím bychom si ale moc nepomohli: překlad by trval stejně dlouho jako prve. Klíčový postřeh je v tom, že kdykoliv měníte velký program, měníte jen jeho malou část. Změníte jednu součástku, jeden soubor, a program přeložíte. Nabízí se nám tak možnost překládat jednotlivé zdrojové soubory samostatně a překládat vždy jen ty, které se od posledního překladu změnily.

Překladač tedy dostane ke zpracování seznam souborů, které se změnily, a každý z nich přeloží do objektového kódu. Pak přijde na řadu linker, který vezme všechny objektové soubory vašeho programu a vytvoří z nich spustitelný soubor. Když se změní jen jeden zdrojový soubor, překladač změní jen jeden objektový soubor a ušetří si většinu práce. Linker má vždycky práce stejně, ale to nevadí, protože linker je nesrovnatelně rychlejší než překladač.

A k čemu potřebujeme preprocesor? Musíte si uvědomit, že překladač nemůže s jednotlivými soubory pracovat úplně samostatně. Když chcete používat například objekt `cout` definovaný v jiném souboru, překladač musí mít přístup k definici tohoto objektu. Preprocesor se stará, aby měl překladač k dispozici definice všech objektů potřebných pro překlad aktuálního souboru.

Přijde vám to složité? Nebojte, ve skutečnosti je to ještě složitější. Ale tím se teď nemusíte trápit, přesné principy překladu nejsou prozatím podstatné. Jednak se k nim můžete vrátit později a jednak existují integrovaná vývojová prostředí, která vás od překladu do velké míry odstíní.

Práce v integrovaném vývojovém prostředí

Zdrojový kód můžete psát v libovolném textovém editoru, klidně i v Poznámkovém bloku. Také překladač si můžete stáhnout zdarma, většinou s ním dostanete i preprocesor a linker. Program si pak přeložíte ručně, například v Příkazovém řádku Windows.

Na textovém editoru a překladu z příkazové řádky sice není vůbec nic špatného, ale hodně programátorů – včetně mě – dává přednost integrovaným vývojovým prostředím neboli IDE. Slovo *integrovaný* znamená, že dostanete textový editor, preprocesor, překladač a linker pod jednou softwarovou střechou. IDE se o překlad kódu postará za vás, takže se nemusíte jednotlivými programy zabývat ručně. Většina IDE má navíc grafické rozhraní, které je pro hodně lidí schůdnější než příkazová řádka; a konečně většina IDE nabízí další funkce, které vám pomůžou s hledáním a opravou chyb.

Hlavní nevýhoda je v tom, že za většinu IDE musíte platit (i když existují i některá zdarma). IDE také zabere víc místa na disku a v paměti. Já bych vám IDE doporučil, protože se díky němu můžete soustředit výhradně na programování v C++ a nemusíte řešit, který přepínač máte použít na příkazové řádce.

Na trhu je mnoho dobrých IDE pro různé operační systémy a platformy. Dobrá zpráva je, že hodně výrobců nabízí limitované verze svých produktů pro nekomerční účely zadarmo. Microsoft například nabízí své komplexní řešení pro vývoj zvané Microsoft Visual Studio 2008 Express Edition ke stažení zdarma. Nás bude zajímat především nástroj Microsoft Visual C++ 2008. Borland nabízí svůj C++ Builder pro vývoj pro OS Windows. Pokud používáte operační systém Linux (nebo Unix) bude zřejmě nejlepší volbou používat GNU C/C++ překladač. Jedná se většinou o integrální součást každé linuxové distribuce, takže ho není třeba nějak instalovat. Používá se z příkazové řádky, ale existuje mnoho grafických nástaveb, které práci s ním udělají příjemnější. Například součástí populární linuxové distribuce Fedora je i vývojové prostředí KDevelop C/C++.

V této knize budu používat Microsoft Visual C++ 2008 Express Edition, protože je to prostředí, které mám nainstalované a často ho používám. Nicméně většina IDE pracuje na stejném principu a náš kód bude kompilovatelný a spustitelný bez ohledu na to, které IDE používáte, pokud nebudete používat knihovní soubory specifické pro dané IDE. Standardní knihovní soubory, které budeme používat, jako např. iostream a string, jsou stejné pro všechna IDE pro C++.

Instalace Microsoft Visual C++ 2008 Express Edition

Předtím než se pustíme do vytváření prvního projektu, je nutné mít nainstalované nějaké IDE. Popíšu zde, kde stáhnout a jak nainstalovat Microsoft Visual C++ 2008 Express Edition, protože se

na něj budu odkazovat v příkladech této knihy. Nic vám ale nebrání nainstalovat si vaše oblíbené IDE a pracovat na příkladech v něm, vše bude fungovat.

Nejprve si ukážeme odkud stáhnout naše IDE. Nejlépe přímo ze stránek Microsoftu:

http://www.microsoft.com/Express/

Klepněte na odkaz download a na stránce s produkty ke stažení zvolte Microsoft Visual C++2008 Express Edition (anglická jazyková verze). Na váš počítač se stáhne malý instalační program. Po jeho spuštění si instalátor zjistí softwarovou konfiguraci vašeho PC a stáhne si všechny potřebné komponenty z Internetu. Je zde také volba pro stažení ISO obrazu celé množiny produktů Microsoft Visual Studio, jejž lze následně vypálit na DVD.

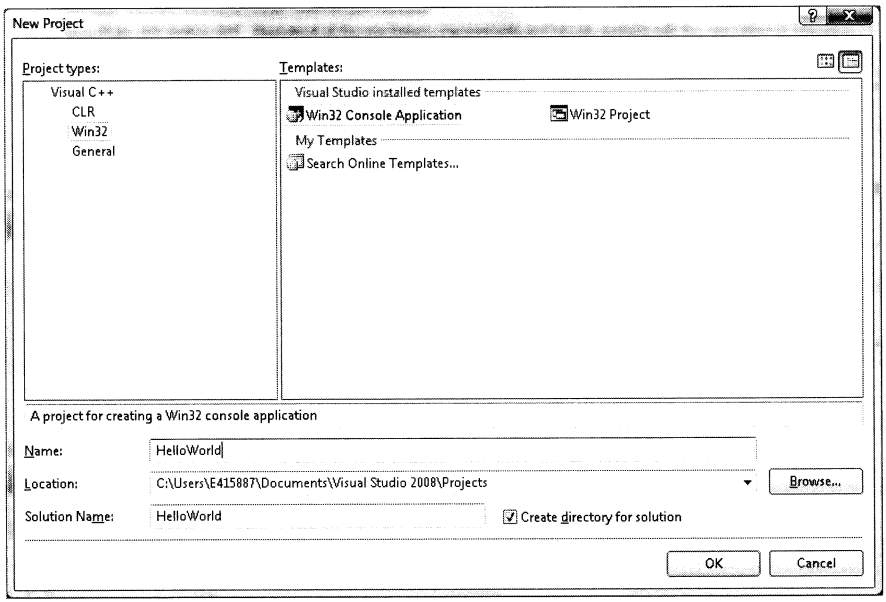
Zvolte ještě možnost nainstalovat si i dokumentaci k Microsoft Visual Studiu s názvem MSDN Express Library (Microsoft Developer Netowork). Může se hodit pro vaše další pokusy s C++.

Po úspěšné instalaci by se ve vaší nabídce Start měla objevit složka „Microsoft Visual C++ 2008 Express Edition" a v ní stejnojmenná ikona pro spuštění IDE.

Vytváříme projekt „Hello World!"

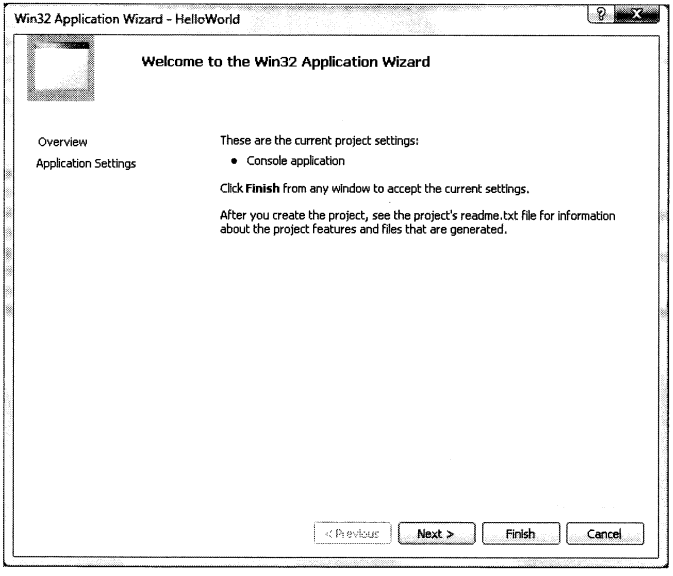
Pokud již na vašem počítači sídlí Microsoft Visual C++ 2008, pak jste připravení pro start vašeho prvního projektu, jehož cílem je vytvořit a spustit aplikaci „Hello World!".

1. Spusťte Microsoft Visual C++ 2008.
2. V menu vyberte příkaz File → New → Project. Tento příkaz aktivuje dialog pro vytvoření nového projektu (New Project dialog), viz obrázek 1.2 (hodnoty položek „Name", „Solution Name" a „Location" budou nastaveny v krocích 5 a 6).
3. V levém panelu dialogu pro vytvoření projektu vyberte položku Win32, jak je ukázáno na obrázku 1.2.



Obrázek 1.2: Vytváření nového projektu

4. Vyberte „Win32 Console Application" napravo v kontextovém panelu dialogu pro vytvoření nového projektu. Slovo konzola (console) značí, že vytváříme aplikaci, která bude běžet k okně konzoly. Označení Win32 značí, že aplikace bude určena pro 32bitový operační systém Windows, jako např. Windows 2000, XP, Vista atd.
5. V poli „Location" za použití tlačítka „Browse" vyberte existující adresář, ve kterém bude vytvořen projekt.
6. V poli „Name" zadejte jméno, které jste vybrali pro projekt. Jméno projektu se automaticky vyplní i jako jméno řešení (Solution) a vytvoří se adresář s tímto jménem, do něhož se uloží všechny projektové soubory.
7. Klepněte na tlačítko OK. Zobrazí se průvodce pro vytvoření Win32 konzolové aplikace (Win32 Application Wizard), viz obrázek 1.3.

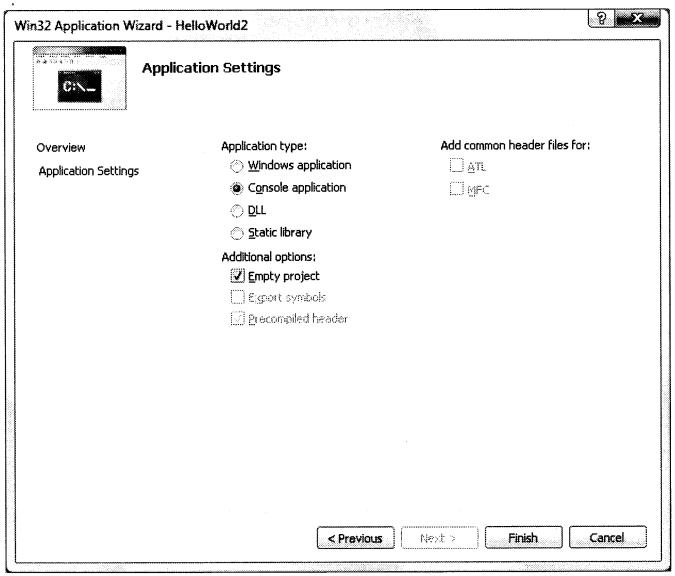


Obrázek 1.3: Průvodce pro vytvoření Win32 konzolové aplikace

8. Klepněte na tlačítko „Next" vpravo dole. Průvodce se přepne na nastavení parametrů aplikace, jak je vidět na obrázku 1.4.
9. Pokud to bude nutné, zvolte konzolovou aplikaci (Console application) jako aplikační typ (Application Type). Zaškrtněte checkbox v „Empty project" (prázdný projekt) v sekci Additional Options (Dodatečná nastavení). Po zaškrtnutí volby „Empty project" se zablokují ostatní volby ze sekce „Additional options".

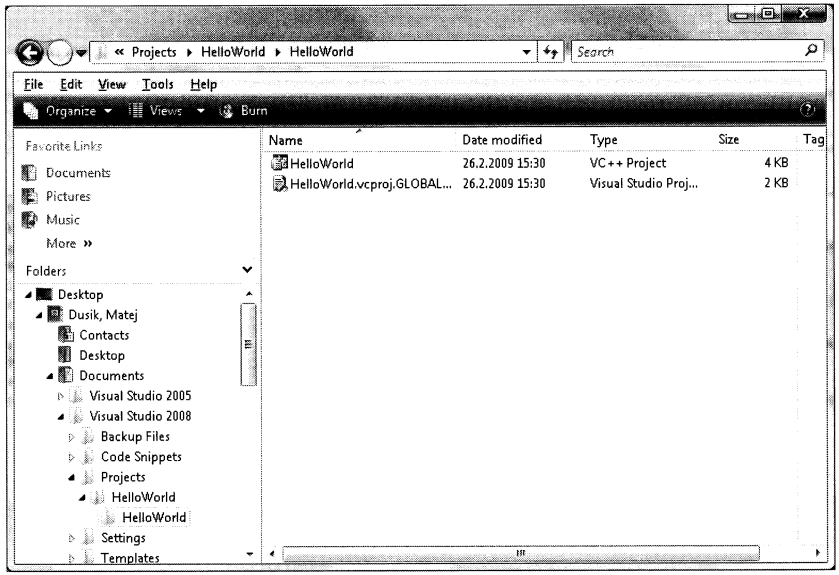
Upozornění: Věnujte tomuto kroku velkou pozornost, obzvláště nezapomeňte zaškrtnout volbu „Empty project". Nesprávná konfigurace aplikačních nastavení je obvyklou chybou a může zapříčinit to, že budete muset začít znovu.

10. Klepněte na tlačítko „Finish". Obrázek 1.5 zobrazuje nově vytvořené adresáře. Jména adresářů odpovídají zadaným jménům pro řešení (Solution) a projekt, tedy HelloWorld. Tato jména byla zadána v krocích 5 a 6.



Obrázek 1.4: Průvodce vytvořením Win32 konzolové aplikace – aplikační nastavení

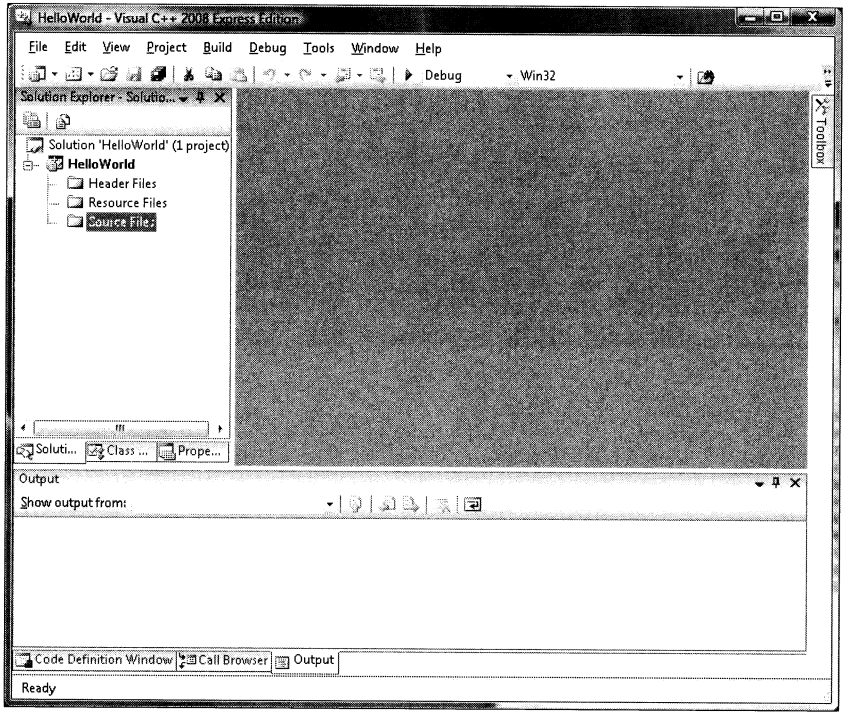
Nyní jste vytvořili projekt pro vaši aplikaci. Projekt je skořápkou pro vaši aplikaci, obsahuje soubory pro podporu vytváření a běhu aplikace. Nicméně teď je projekt prázdný, neobsahuje žádný kód, takže nedělá nic. Dalším krokem je začít se psaním kódu.



Obrázek 1.5: Průzkumník ukazující nově vytvořené projektové soubory a adresáře

Psaní zdrojového kódu

Visual C++ obsahuje panel podobný tomu z Průzkumníka. Jmenuje se „Solution Explorer" (Průzkumník řešení) a je zobrazen na obrázku 1.6. Pokud není „Solution Explorer" vidět, lze ho aktivovat v menu View → Solution Explorer.



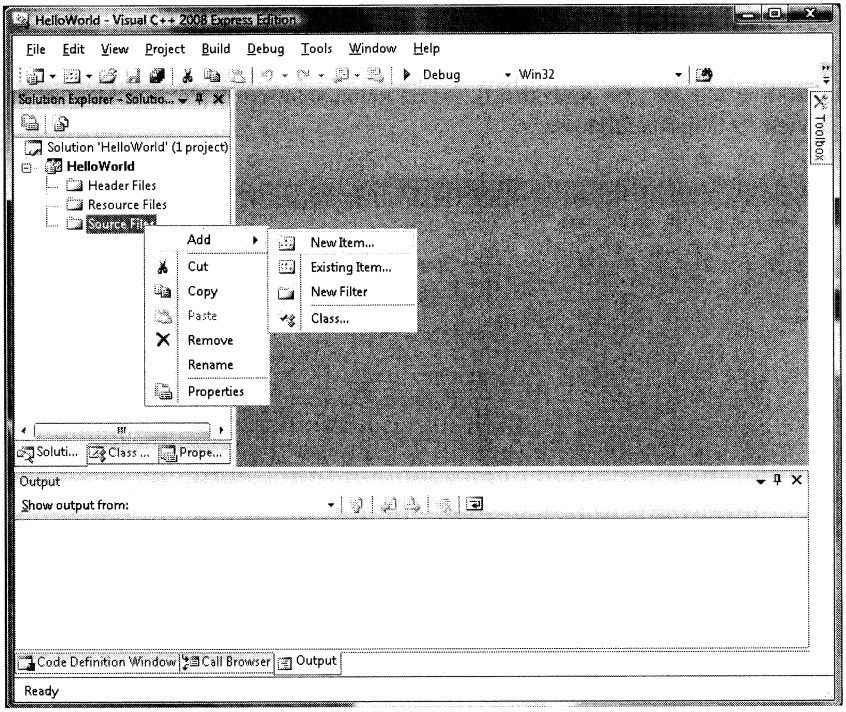
Obrázek 1.6: Průzkumník ukazující nově vytvořené projektové soubory a adresáře

Průzkumník řešení má složku pro zdrojové i hlavičkové soubory. Soubor, který se chystáme vytvořit a do kterého chceme umístit kód aplikace „Hello World!", je soubor zdrojový. Zdrojové soubory mají příponu .cpp pro programu napsané v C++. Oproti tomu soubor iostream, který je vložený direktivou include, je hlavičkový soubor. Hlavičkové soubory mají příponu .h (h jako header – hlavička).

My použijeme průzkumníka řešení (Solution explorer) k přidání nového zdrojového souboru do projektu, poté začneme s psaním kódu do tohoto souboru.

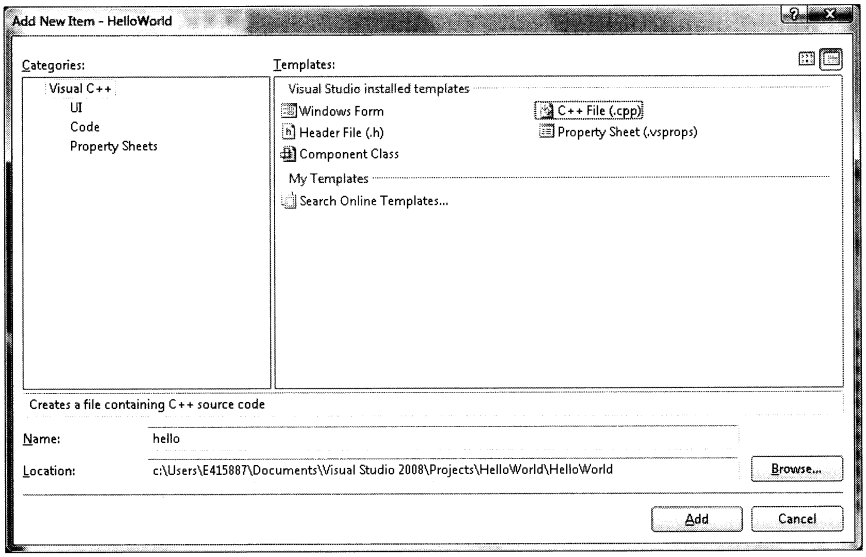
Následující kroky slouží k přidání nového zdrojového souboru do projektu:

1. Klepněte pravým tlačítkem myši na složku „Source Files". Zobrazí se kontextové menu zobrazené na obrázku 1.7.
2. Z menu vyberte položku Add → Add New, zobrazí se dialog pro přidání nové položky, viz obrázek 1.8.



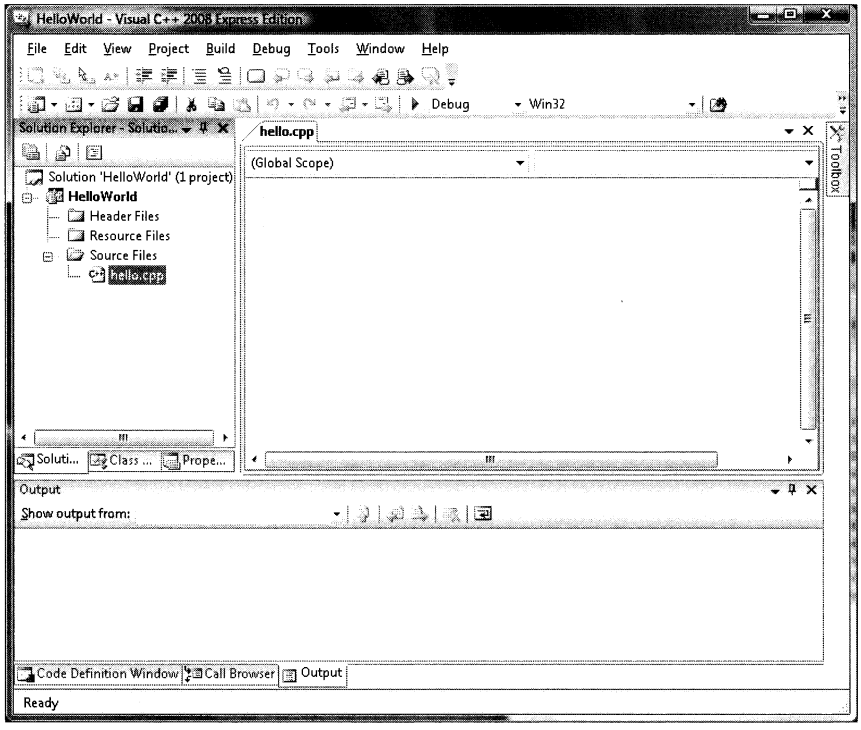
Obrázek 1.7: Kontextové menu složky „Source Files"

Poznámka: Pokud chcete vložit již existující zdrojový soubor, můžete to udělat pomocí položky Add → Add Existing Item z kontextového menu složky „Source Files".



Obrázek 1.8: Přidávání nového souboru do projektu

3. Obvykle není nutné měnit umístění nově vytvářeného souboru v poli „Location". Přednastavená složka je složkou, kde jsou umístěny projektové soubory projektu. Zadejte název nového zdrojového souboru do pole „Name". Není třeba zadávat příponu .cpp; bude přidána automaticky, protože přece vytváříme zdrojový soubor. Zadáme-li jako název „hello", viz obrázek 1.8, pak se nový soubor bude jmenovat hello.cpp.
4. Po zadání názvu souboru klepněte na tlačítko „Open". Na obrázku 1.9 si můžete prohlédnout nově vytvořený soubor v průzkumníku řešení (Solution Explorer).

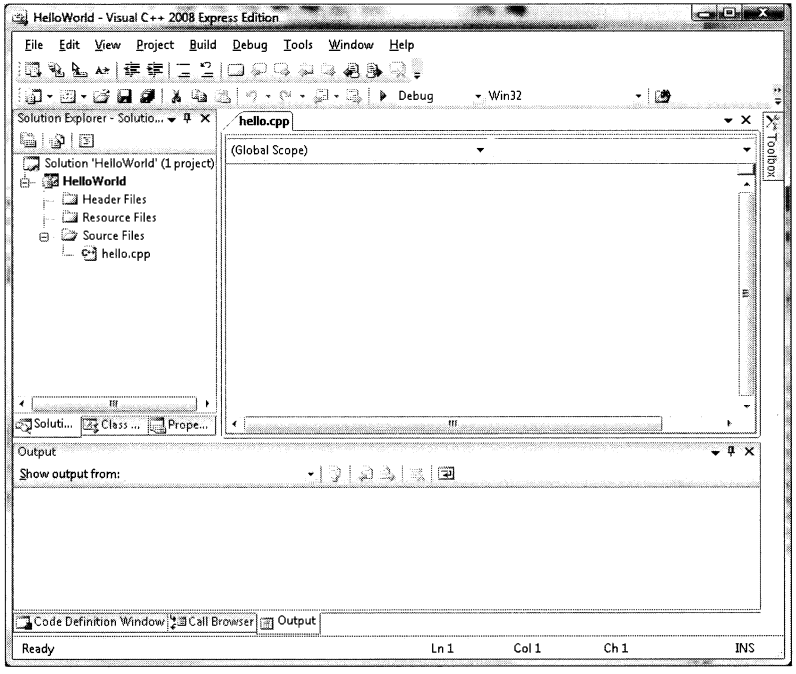


Obrázek 1.9: Průzkumník řešení (Solution Explorer) zobrazující nový .cpp soubor

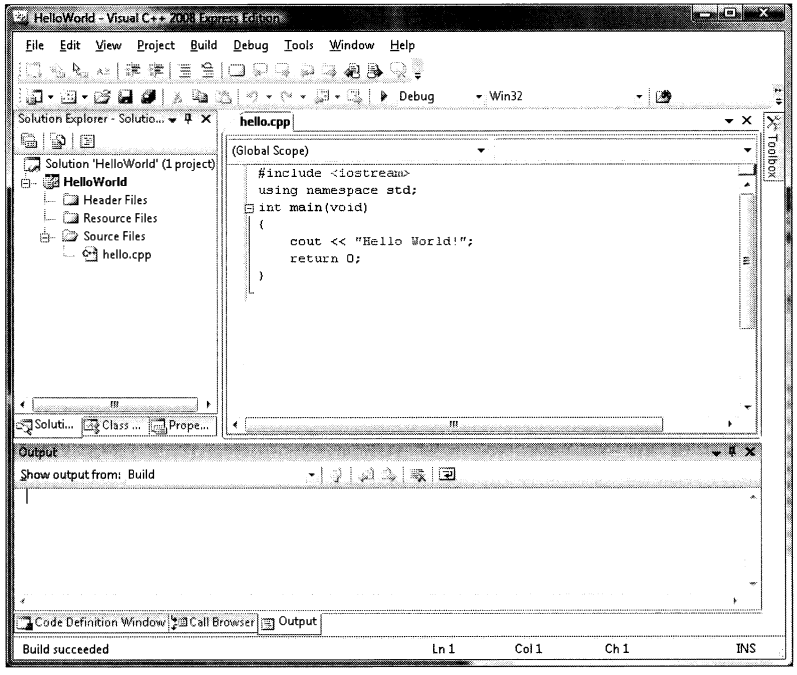
Psaní kódu není nic složitého. Stačí poklepání na soubor hello.cpp v průzkumníku řešení (Solution Explorer) a otevře se editor zdrojového souboru, který je zatím ještě prázdný (viz obrázek 1.10). Nyní jednoduše napište kód programu Hello World! Po dokončení by měl program vypadat tak jako na obrázku 1.11.

Upozornění: K psaní kódu můžete klidně použít Notepad nebo kterýkoliv jiný textový editor. Nicméně nepoužívejte Microsoft Word či jakýkoliv jiný textový procesor. Textové procesory umožnují pokročilé formátování textu, kterého je dosaženo použitím spousty skrytých formátovacích znaků, kterým překladač nerozumí a vyhodnotí je jako syntaktické chyby.

Uložte výsledky své práce klepnutím na tlačítko „Save" na nástrojové liště. Nyní jsme připraveni ke kompilaci programu.



Obrázek 1.10: Zdrojový soubor před psaním kódu



Obrázek 1.11: Zdrojový soubor po zadání kódu

Sestavení (build) projektu

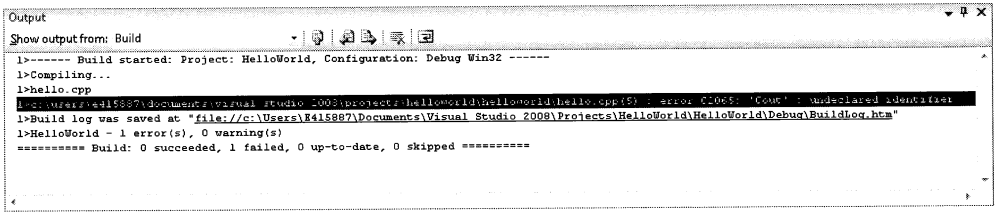
Kompilaci kódu provedeme příkazy z menu „Build". Ke zkompilování vedou všechny z následujících voleb:

- Build → Solution
- Rebuild → Solution
- Build → HelloWorld
- Rebuild → HelloWorld

HelloWorld je název našeho projektu. Řešení (neboli Solution) může obsahovat více než jeden projekt. Naše řešení ovšem obsahuje pouze jeden projekt, takže není žádný praktický rozdíl mezi projektem a řešením.

Sestavením (build) se myslí kompilace změn od poslední kompilace (pokud nějaká proběhla). Opětovné sestavení (rebuild) znamená kompilaci veškerých zdrojů od začátku, proto je sestavení většinou rychlejší než opětovné sestavení. Opětovné sestavení je dobré použít, pokud došlo v kódu od poslední kompilace k velkým změnám. Z praktického hlediska je mezi nimi mizivý rozdíl.

Před kompilací ještě uděláme jednu změnu ve kódu. Zaneseme do něj chybu, a to tak, že změníme písmeno malé c na velké C ve slovu cout (cout → Cout). Pak vybereme jednu ze čtyř kompilačních možností. V okně „Output" – výstup by se měla objevit chybová hláška překladače, viz obrázek 1.12. Chybová hláška „c2060" nám říká, že identifikátor Cout není deklarovaný.



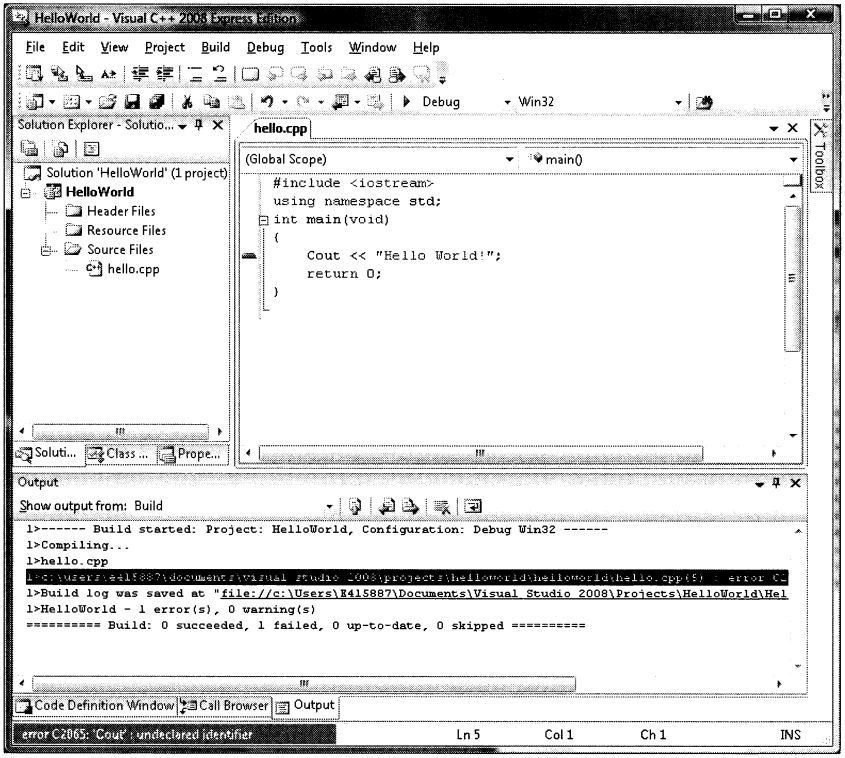
Obrázek 1.12: Okno „Output" s chybovou hláškou kompilátoru

Tip: Pokud v okně „Output" není chyba zobrazena celá, můžete si okno roztáhnout či uvolnit a rozbalit si ho na celou obrazovku.

Jak bylo vysvětleno v předchozích kapitolkách, kompilátor porozumí vašemu kódu a přeloží ho do strojového jazyka, pouze pokud váš kód odpovídá syntaktickým pravidlům daného programovacího jazyka. Jak již bylo vysvětleno, C++ má řadu pravidel pro zápis slov a pro gramatiku příkazů a konstrukcí. Pokud tato pravidla porušíte, jedná se o syntaktickou chybu. Následkem toho kompilátor nemůže přeložit váš kód do instrukcí strojového jazyka a odmění vás chybovou hláškou o syntaktické chybě.

Kód v C++ rozlišuje malá a velká písmena. To znamená, že stejné slovo napsané velkými a malými písmeny není považováno překladačem za jedno a totéž. Správný zápis je cout, Cout je špatně. Protože C++ neví, co Cout znamená, překladač vypíše chybovou hlášku, že se jedná o nedefinovaný identifikátor.

V našem projektu je pár řádek kódu, ale jakmile se kód rozroste, je nesnadné nalézt místo, kde se chyba nachází. Pokud dvakrát klepnete na chybovou hlášku v okně „Output", kurzor v editoru kódu se přesune na místo chyby a řádek, na kterém se nachází, je označen ikonou (viz obrázek 1.13).



Obrázek 1.13: Chyba označena v editoru kódu

Nyní opravte Cout za cout a znovu zkompilujte váš kód. Tentokrát by měla být kompilace úspěšná. Pokud se nyní pomocí průzkumníku z Windows podíváte do adresáře projektu HelloWorld, naleznete tam soubory hello.obj a hello.exe. Jedná se o objektový a spustitelný soubor, o kterém jsme se bavili v kapitole „Překlad zdrojového kódu". Jak bylo řečeno, při sestavení byl použit preprocesor, kompilátor a linker.

Spuštění kódu

Finálním krokem je spuštění kódu. K tomuto úkolu slouží menu „Debug" – ladit. Můžeme zvolit položku Debug → Start of Debug nebo Debug → Start Without Debugging. Rozdíl mezi nimi je použití debuggeru (funkce či program IDE pro usnadnění hledání chyb v programu), tato problematika bude probrána v následujících kapitolách. Protože tentokrát debugger nepoužijeme, vyberte Debut → Start Without Debugging, jenž je trochu rychlejší. Výsledkem je konzolové okno zobrazující nápis „Hello World!" (ukázáno na obrázku 1.1).

Test

1. Co je počítačový program?
2. Vyjmenujte několik věcí, které počítač při zpracování informací zvládá lépe než člověk.

3. Co je programovací jazyk?
4. Proč se vyplatí naučit se C++?
5. Co je funkce?
6. Kolik funkcí `main` najdete v běžném programu napsaném v C++?
7. Co je to standardní knihovna?
8. K čemu slouží direktiva `#include`?
9. Co dělá preprocesor?
10. Co dělá překladač?
11. Co dělá linker?

Paměť a datové typy

Když jsem dopsal svou první knihu, každý den jsem netrpělivě kontroloval poštovní schránku, jestli už začaly chodit prosby o autogram. Výsledek se záhy dostavil, schránka přetékala. Člověk by měl být opatrnější v tom, co si přeje – mezi hlavní zájemce o můj podpis patřila hypoteční banka, účet za kreditní karty, pojištění, telefon, elektřinu a další fanoušci, které si jistě dokážete představit.

Všechny firmy, které s takovou láskou posílají účty, by své zákazníky jistě nezvládly evidovat pomocí tužky a papíru. Místo nich používají počítačové programy, těží z jejich schopnosti uložit obrovské množství informací a bleskem si je vybavit.

My lidé si informace ukládáme do paměti. Totéž dělají počítače, ale jejich paměť se od té naší výrazně liší. Fungování počítačové paměti se bude věnovat tato kapitola.

Informace neboli data mají různou podobu. Některá data jsou číselná, například můj účet za plyn. Jiná data jsou textová, například jméno na mém účtu za plyn. Typ dat, například číslo nebo text nebo cokoliv dalšího, se celkem logicky označuje jako *datový typ*. Datový typ informací se promítá do způsobu jejich uložení i do množství paměti, kterou zaberou. Různými datovými typy se bude zabývat následující text.

Paměť

Počítačové programy se skládají z příkazů a dat. Jak bylo řečeno v předchozí kapitole, příkazy jsou napsané v nějakém programovacím jazyce (například C++) a pomocí překladače se převádí do strojového jazyka. Krok za krokem počítači říkají, co má dělat. Naproti tomu data jsou to, s čím program pracuje. Pokud například uživatel vašeho programu potřebuje seznam všech studentů se studijním průměrem pod tři, jeho data by mohl tvořit seznam studentů a jejich studijních průměrů. Program na základě příkazů seznam projde a vypíše všechny studenty s průměrem pod tři.

Příkazy programu i jeho data musí být v paměti počítače, jinak by program nefungoval. Následující text rozebere jednotlivé typy počítačové paměti a ukáže vám, kam se v paměti ukládají příkazy a kam data.

Druhy paměti

Váš počítač obsahuje tři základní paměťová centra:

- Procesor neboli CPU
- Operační paměť neboli RAM
- Trvalou paměť

Vyrovnávací paměť procesoru

Procesor je mozek počítače. Možná jste nad ním přemýšleli, když jste naposledy kupovali počítač, protože rychlost procesoru hraje při koupi počítače výraznou roli. Čím rychlejší procesor, tím rychlejší počítač.

Poznámka: Rychlost procesorů se udává v hertzech. Hertz, pojmenovaný po Heinrichovi Hertzovi, který poprvé dokázal existenci elektromagnetických vln, představuje jednu změnu stavu za sekundu. Rychlost procesorů se většinou udává v megahertzech (MHz), tedy milionech hodinových cyklů za sekundu, nebo gigahertzech (GHz), tedy miliardách hodinových cyklů za sekundu. Procesor s rychlostí 800 MHz provede za jednu vteřinu 800 milionů hodinových cyklů. Každá instrukce strojového jazyka zabere určitý počet cyklů, takže rychlost procesoru určuje, kolik příkazů zvládne procesor provést za jednu sekundu.

Kromě toho, že procesor řídí provoz počítače, má přímo u sebe i malé množství paměti. Této paměti se říká *vyrovnávací paměť* neboli *cache*. Používá se pro uložení nejčastěji používaných příkazů a dat. Jelikož je tak blízko procesoru, přístup do ní je velice rychlý. Na druhou stranu se toho do ní moc nevejde, stačí jen na nejčastěji používané příkazy a data. Mimo *cache* se v procesoru jestě nachází tzv. paměťové registry. Ty slouží k uložení dat, se kterými procesor přímo provádí operace. Zbytek příkazů a dat se musí ukládat jinam.

Operační paměť

„Jinam" znamená ve většině případů do operační paměti neboli RAM. I nad tou jste nejspíš při posledním nákupu počítače uvažovali, protože čím víc operační paměti počítač má, tím je rychlejší a tím víc programů na něm může najednou běžet.

Procesor se do operační paměti dostane skoro stejně rychle jako do cache, a navíc má operační paměť ve srovnání s vyrovnávací pamětí nesrovnatelně vyšší kapacitu. Obě paměti jsou ale dočasné, jejich obsah se ztratí v okamžiku vypnutí počítače. Na tento nepříjemný fakt už jste nejspíš narazili, když se vám počítač vypnul kvůli výpadku elektřiny nebo se musel restartovat a vy jste přišli o neuložená data.

My bychom byli rádi, kdyby data v pořádku přečkala vypnutí programu i vypnutí počítače. Na počítači navíc určitě máte další programy (například e-mailového klienta, webový prohlížeč nebo textový procesor), které momentálně neběží, ale které na počítači potřebujete. Podobně jistě máte řadu souborů, například ročníkových prací, dopisů, daňových tabulek, e-mailů a podobně, které zrovna nepoužíváte, ale které budete potřebovat časem. Proto potřebujeme ještě jeden typ paměti, který by byl na rozdíl od cache a RAM trvalý, tedy přežil vypnutí počítače.

Trvalá paměť

Trvalou pamětí je nejčastěji pevný disk, CD, DVD nebo flash disk. Všechny trvalé paměti mají společné to, že na nich příkazy i data zůstanou i po vypnutí počítače. Počítač můžete mít vypnutý třeba měsíce, ale když ho zase zapnete, všechny soubory jsou stále na svém místě.

Kromě toho, že je trvalá paměť nezávislá na zapnutém počítači, má i mnohem větší kapacitu než paměť operační – řekněme stokrát až tisíckrát. Jelikož je trvalá a má vysokou kapacitu, ukládají se do ní v počítači programy i data. Když například na počítač nainstalujete program Microsoft Word, jeho soubory se uloží na pevný disk. Stejně tak se na pevný disk ukládají všechny dokumenty, které ve Wordu napíšete.

Trvalá paměť sice vydrží a má velkou kapacitu, ale počítač neumí provádět příkazy na ní uložené. Pokud se mají nějaké příkazy provést, musí se načíst do operační paměti. S daty je to podobné – pokud má počítač změnit nějaká data, musí je načíst z trvalé paměti do operační.

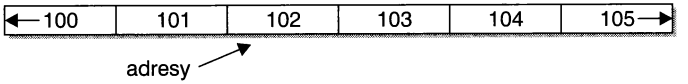
Poznámka: Zvláštním případem trvalé paměti je ***paměť virtuální*** neboli ***swap space***. To je trvalá paměť, která slouží jako dočasné odkladiště pro data z operační paměti. Její podrobnější popis je bohužel nad rámec této kapitoly.

Obecně vzato ukládají programy většinu svých příkazů a dat v operační paměti, takže ta nás bude zajímat nejvíc. Velká část naší diskuze bude platit i pro paměť trvalou. Vyrovnávací paměť procesoru a registry jsou úplně jiné téma a většinou přichází na pořad pouze u programovacích jazyků, které jsou blíže strojovému jazyku (například u assembleru).

Adresy

Když se vás někdo zeptá, kde bydlíte, odpovíte třeba že v Mockingbird Lane 1313. To je vaše adresa, podle které vás kdokoliv může najít. Adresy většinou dodržují určitý logický pořádek. Čísla v rámci jednoho bloku mohou jít od 1300 po 1399, v dalším bloku budou čísla 1400–1499 a tak dále.

Podobně jsou adresovaná i místa v paměti počítače. Vypadají úplně jinak než naše čísla ulic, na která jsme zvyklí, protože většinou jde o šestnáctková čísla – například 0x8fc1. Ať už je ale samotné číslo zapsané jak chce, adresy dodržují tentýž logický pořádek a jejich čísla jdou hezky za sebou (viz obrázek 2.1).



Obrázek 2.1: Posloupnost paměťových adres

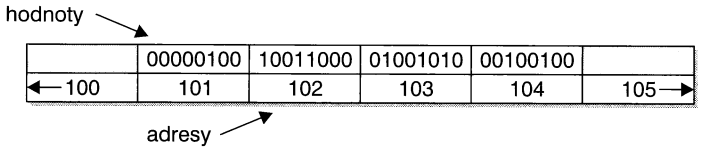
Poznámka: Co se šestnáctkových čísel týká – my jsme zvyklí na čísla v desítkové soustavě, ve které se používají číslice od nuly do devíti. Naproti tomu adresy v paměti se většinou udávají v soustavě šestnáctkové. Pro zápis číslic v šestnáctkové soustavě bychom potřebovali šestnáct různých číslic, ale jelikož máme číslic jen deset (0–9), místo zbývajících pěti „číslic" se používají písmena od A po F. Například desítkové číslo 16 se v šestnáctkové soustavě zapíše jako 10.

Šestnáctková soustava má výhodu v tom, že je úspornější. Paměťové adresy bývají poměrně veliká čísla, která jsou v šestnáctkovém zápisu kratší než v desítkovém – například 1 000 000 desítkově je f4240 šestnáctkově. Převody mezi desítkovým a šestnáctkovým zápisem si vysvětlíme za chvilku.

Bity a bajty

Zatímco na číslech ulic žijí lidé, na paměťových adresách jsou uložené *bajty*. V předchozí kapitole jsme mluvili o tom, že první počítače byly v podstatě hromada přepínačů, které mohly být buď zapnuté (1), nebo vypnuté (0). Máme tu tedy určitou nejmenší možnou užitečnou jednotku informace, která může mít hodnotu 1 nebo 0. Této jednotce se v počítačové terminologii říká *bit*. Počítač přemýšlí v bitech, ale zpracování dat po jednotlivých bitech by nebylo praktické. Proto se bity sdružují do skupin po osmi, a skupina osmi bitů se označuje jako *bajt*.

Na každé adrese v paměti je uložený právě jeden bajt informací, tedy osm jedniček nebo nul. A stejně jako můžeme podle čísla ulice najít člověka, podle paměťové adresy najdeme bajt informací. Posloupnost adres a hodnot na nich uložených ilustruje obrázek 2.2.



Obrázek 2.2: Posloupnost bajtů a jejich adresy v paměti

Dvojková číselná soustava

Informace jsou v paměti uložené jako posloupnost jedniček a nul, které toho většině z nás moc neřeknou. Zato počítač se ve šrůtcích jedniček a nul vyzná výborně.

Například já jsem se z pohledu svého počítače narodil v roce 11110100000. Ještě než začnete protestovat, klidně vám prozradím, že jsem se narodil v roce 1952. Jak jsem se mohl narodil v roce 11110100000 a zároveň v roce 1952?

My jsme zvyklí s čísly pracovat v desítkové soustavě, ve které je každé číslo zapsané posloupností číslic od nuly do devítky. Když jsem mluvil o roce 1952, uvedl jsem letopočet v desítkové soustavě.

Každá posloupnost jedniček a nul je také číslo, i když možná trochu jiné, než na jaké jste zvyklí. Když jsem mluvil o roce 11110100000, uvedl jsem letopočet v dvojkové neboli binární soustavě. Ve dvojkové soustavě se každé číslo zapisuje posloupností nul a jedniček.

Zatímco lidé většinou uvažují v desítkové soustavě, počítač „přemýšlí" v soustavě dvojkové. Proto se při programování setkáte s čísly v obou těchto soustavách.

Převody mezi číselnými soustavami

Bez schopnosti převádět mezi dvojkovým a desítkovým zápisem čísel se při programování docela dobře obejdete. Převod ale není těžký a občas vám pomůže pochopit dění za oponou. Pokud máte zájem, čtěte dál.

Převod z dvojkové do desítkové soustavy je prostý. Směrem zprava doleva se první dvojková číslice vynásobí číslem 2^0 (tedy jedničkou), druhá číslice se vynásobí číslem 2^1 (dvojkou), třetí číslice se vynásobí číslem 2^2 (čtyřkou) a tak dále. Výsledek všech násobení se sečte a dostanete desítkový zápis původního dvojkového čísla. Příklad výpočtu pro čísla od nuly do pěti najdete v tabulce 2.1.

Tabulka 2.1: Převod čísel od nuly do pěti z dvojkové do desítkové soustavy

Dvojkový zápis	Výpočet	Desítkový zápis
0	$0\times2^0 = 0\times1$	0
1	$1\times2^0 = 1\times1$	1
10	$1\times2^1+0\times2^0 = 2+0$	2
11	$1\times2^1+1\times2^0 = 2+1$	3
100	$1\times2^2+0\times2^1+0\times2^0 = 4+0+0$	4
101	$1\times2^2+0\times2^1+1\times2^0 = 4+0+1$	5

A co převod opačným směrem, z desítkové soustavy do dvojkové? Důležité je si uvědomit, co přesně zápis čísla v dané soustavě znamená. Například číslo 145 v desítkové soustavě znamená $1\times100 + 4\times10 + 5\times1$. Tento rozklad běžně děláme v řeči: „Jedno sto čtyřicet pět." Není na něm

nic objevného – zajímavější bude, když si ho přepíšeme do tvaru $1\times10^2 + 4\times10^1 + 5\times10^0$. Tady už je vidět, že každá číslice ze zápisu čísla v desítkové soustavě říká, kolikrát číslo obsahuje určitou mocninu desítky.

Tento postřeh se dá zobecnit na libovolnou číselnou soustavu: Každá číslice ze zápisu čísla o základu *n* říká, kolikrát číslo obsahuje určitou mocninu *n*. Praktickým zvykem je psát nejnižší mocniny vpravo, takže například číslo 110 ve dvojkové soustavě obsahuje nulakrát dvojku na nultou, jedenkrát dvojku na první a jedenkrát dvojku na druhou. Jde tedy o šestku, která se v desítkové soustavě zapisuje jako šestkrát deset na nultou: 6. Stejně tak bychom ji mohli převést do šestnáctkové soustavy, kde dopadne jako šestkrát šestnáct na nultou: 6.

Když tedy chcete převést číslo z desítkového zápisu na dvojkový, stačí se podívat, kolikrát jsou v něm zastoupené jednotlivé mocniny dvojky. K tomu nám dobře poslouží zbytek po dělení, zkusme to například pro číslo 6:

- 6/2 = 3, zbytek 0
- 3/2 = 1, zbytek 1
- 1/2 = 0, zbytek 1

Všimněte si, že zbytky tvoří přesně dvojkový zápis čísla šest, 110. Když tedy chcete převést číslo z desítkové soustavy do dvojkové, dělte jej dvojkou, zbytky po dělení si zapisujte odzadu a výsledek dělení dělte dál, dokud není nulový.

Stejný postup se dá použít pro převod mezi desítkovou a šestnáctkovou soustavou. Když převádíte ze šestnáctkové soustavy do desítkové, násobíte každou číslici odpovídající mocninou šestnáctky (přičemž šestnáctková „číslice" A odpovídá desítce, B jedenáctce a tak dále). Například číslo šestnáctkové číslo 5c je $5\times16^1 + 12\times16^0 = 80+12 = 92$.

A naopak když převádíte z šestnáctkové do desítkové soustavy, vydělíte 92 šestnácti, dostanete pětku a zbytek 12. Pětku opět vydělíte šestnácti, dostanete nulu a zbytek 5. Teď už byl výsledek dělení nulový, takže hledané šestnáctkové číslice jsou zbytky po dělení: pět a dvanáct. Číslice dvanáct se šestnáctkově zapisuje jako C, takže hledaný šestnáctkový zápis je 5c.

Datové typy

Jedničky a nuly uložené v paměti mohou reprezentovat text (například mé jméno, Jeff Kent), celá čísla (například mou výšku v palcích, 72) nebo desetinná čísla (například můj průměrný studijní prospěch, na který už si nevzpomenu). Stejně dobře ale mohou reprezentovat logickou hodnotu pravda/nepravda (true/false), například jestli jsem občan Spojených států.

Data mají zkrátka celou řadu podob, které se dají obecně rozdělit na textové a číselné. Čísla se dají rozdělit ještě na celá (například 6, 0 nebo –7) a desetinná (například ⅔, 7,3 nebo –6,1). Typ informací se kromě jiného projeví i na velikosti paměti, kterou informace zaberou.

Celočíselné datové typy

S celými čísly pracujeme neustále. Stačí se zamyslet nad tím, kolik aut stojí na parkovišti, na kolik přednášek chodíte nebo kolik máte sourozenců. Každá z odpovědí je nějaké číslo, u kterého není potřeba uvažovat žádná desetinná místa. (Pokud tedy nemáte 3,71 bratrů.)

Některá celá čísla bývají menší, například který nešťastník by chodil na 754 361 přednášek. Jiná celá čísla musí být větší – například studenti astronomie budou vědět, že Měsíc je od Země

asi 240 tisíc mil daleko. A některá celá čísla bývají hodně, hodně veliká; kupříkladu vzdálenost ze Země na Pluto je v nejlepším případě 2,7 miliardy mil.

Často se stává, že číslo nemůže být záporné. I kdybyste na zkoušce úplně vyhořeli, záporný počet bodů nedostanete. Jiná celá čísla bývají záporná běžně, například teplota na severním pólu.

S ohledem na tyto různorodé požadavky existuje v počítači několik různých datových typů pro celá čísla, viz tabulku 2.2. Uvedené velikosti a rozsahy hodnot jsou jen orientační, liší se podle použitého překladače a operačního systému. Hodnoty platné pro váš překladač a systém se naučíte zjistit později v této kapitole, až se dostaneme k operátoru `sizeof`.

Tabulka 2.2: Rozsah a velikost běžných celočíselných datových typů

Datový typ	Velikost v bajtech	Rozsah
`short`	2	–32 768 až 32 767
`unsigned short`	2	0–65 535
`int`	4	–2 147 483 648 až 2 147 483 647
`unsigned int`	4	0–4 294 967 295
`long`	4	–2 147 483 648 až 2 147 483 647
`unsigned long`	4	0–4 294 967 295

Poznámka: Asi vás napadne, k čemu je datový typ `long`, když je podle tabulky 2.2 úplně stejný jako `int`. Jak už bylo řečeno před tabulkou, skutečná velikost a tedy i rozsah datových typů závisí na operačním systému a překladači. U některých kombinací operačních systémů a překladačů může například `short` vycházet na jeden bajt, `int` na dva a `long` na čtyři.

Začínající programátoři propadají při pohledu na tabulku 2.2 skepsi, jelikož se bojí, že se čísla nezvládnou naučit nazpaměť. Dobrá zpráva je, že vás do toho nikdo nenutí. Dá rozum, že nějaké to memorování je potřeba téměř u všeho. Ale protože programování vyžaduje obrovské množství informací, programátoři v podstatě neustále hledí do nápovědy nebo nějaké referenční příručky. Věřte mi, nejsem výjimka.

Mnohem důležitější než mechanické učení je schopnost pochopit, proč a jak program funguje. Proto se v následujícím textu podrobně podíváme, jak fungují datové typy. Úplně bez výpočtů se samozřejmě neobejdeme, ale výpočty nebudou nijak složité, a když je pochopíte, ulehčíte si programování v dalších kapitolách.

Se znaménkem, nebo bez?

Tabulka 2.2 uvádí tři datové typy: `short`, `int` a `long`. Každý z těchto typů před sebou ještě může mít slovo `unsigned`, například `unsigned short`. Výraz *unsigned* znamená „bez znaménka". Datové typy bez znaménka mají pouze nezáporné hodnoty, tedy nulu a kladná čísla. Datové typy se znaménkem mohou mít hodnoty záporné, nulové i kladné. Když neuvedete modifikátor `unsigned`, datový typ se automaticky bere jako `signed`. (Například `short` a `signed short` je tím pádem totéž.)

Datové typy bez znaménka nemohou obsahovat záporné hodnoty, a tak je například u datového typu `unsigned short` v tabulce 2.2 jako nejmenší možná hodnota uvedena nula. Naproti tomu nejmenší možná hodnota typu `short` je –32 768, protože datový typ se znaménkem může obsahovat i záporná čísla.

Velikost datového typu

Každý z celočíselných datových typů z tabulky 2.2 má určitou velikost, ostatně stejně jako všechny datové typy C++. Velikost datových typů se ale na rozdíl od lidí neudává v palcích nebo librách (které jsou pro mě citlivé téma), nýbrž v bajtech.

Všechny datové typy zabírají alespoň jeden bajt, protože s menšími datovými typy by se pracovalo velice neefektivně. Většina datových typů zabírá bajtů hned několik, viz všechny typy z tabulky 2.2. Velikost datového typu je vždy celé číslo – neexistuje datový typ o velikosti dejme tomu 3,5 bajtu, protože se čtyřmi bity neboli polovinou bajtu by se špatně pracovalo.

Jelikož počítač pracuje ve dvojkové soustavě, velikost datových typů bývá mocnina dvojky. Mezi běžné velikosti datových typů tak patří jeden bajt (2^0), dva bajty (2^1), čtyři bajty (2^2) a osm bajtů (2^3). Z velikosti datového typu plynou dvě podstatné věci: rozsah datového typu a velikost paměti, která je potřeba pro jeho uložení.

Rozsah datového typu

Rozsahem datového typu se myslí jeho nejvyšší a nejnižší hodnota. Například rozsah typu `unsigned short` je 0–65 535. Rozsah není náhodně vybraný, plyne z velikosti datového typu a způsobu jeho uložení.

Datový typ o velikosti n bitů může celkem uložit 2^n různých hodnot. `Short` je dlouhý dva bajty neboli 16 bitů. Z toho plyne, že celkem zvládne uložit $2^{16} = 65\,536$ různých celých čísel.

Nejvyšší hodnota, kterou typ `unsigned short` zvládne uložit, ovšem není 65 536. Začíná totiž nulou, a proto zvládne uložit nanejvýš číslo 65 535. Nejvyšší možná hodnota bezznaménkového datového typu o velikosti n bitů je tedy 2^n-1.

U datových typů se znaménkem je situace o něco složitější. Jelikož se v nich mohou objevit i záporná čísla, musíme někam uložit informaci o tom, že je číslo záporné. Nám lidem se stačí podívat, jestli je před číslem minus. Ale bit v počítači je zkrátka jednička nebo nula, žádné minus pro záporná čísla nenabízí.

Informatika nabízí několik řešení tohoto problému, například samostatný znaménkový bit, jedničkový doplněk nebo dvojkový doplněk, ale my se do složitějšího vysvětlování pouštět nemusíme.

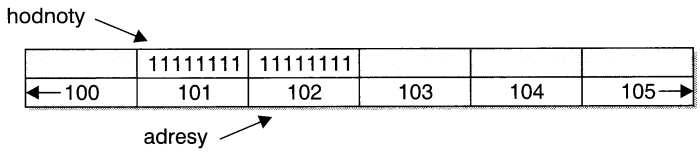
Například typ `signed short` opět uloží nanejvýš 2^0 neboli 65 536 různých kombinací. Jelikož ale musí pamatovat na záporná čísla, rozdělí tyto kombinace rovnoměrně mezi kladná a záporná čísla. To souhlasí s tabulkou 2.2, kde je rozsah typu `signed short` uvedený od –32 768 do 32 767.

K horní a spodní hranici rozsahu datového typu se znaménkem se dá dojít ještě jinak. Jelikož musí jeden bit nutně posloužit pro uložení znaménka, zůstane jen 15 bitů pro uložení čísel. Nejvyšší možná hodnota, která se dá uložit na patnácti bitech, je $2^{15}-1$ (nesmíme zapomenout odečíst jedničku, protože začínáme od nuly). Nejnižší možná hodnota je $-(2^{15})$ neboli –32 768. (A tentokrát žádnou jedničku neodečítáme, protože začínáme od –1, nikoliv od nuly.)

Způsob uložení v paměti

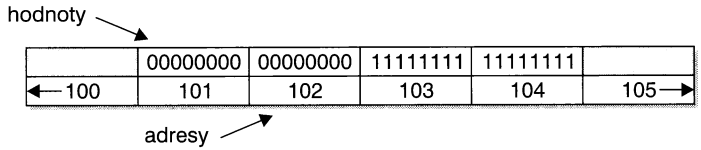
Ve dvojkové soustavě a formátu `unsigned short` má číslo 65 535 v paměti podobu šestnácti jedniček: 1 111 111 111 111 111. Šestnáct jedniček se nám na jednu adresu nevejde – na jednu adresu se vejde jen jeden bajt, osm bitů. Jak tedy hodnotu v paměti uložit?

Odpověď zní, že číslo 65 535 se musí uložit na dvě adresy. Dvě adresy znamenají dva bajty, a ty už nám pro uložení takového čísla stačí. Proto datový typ `short` zabírá dva bajty. Na obrázku je vidět, jak číslo 65 535 ve formátu `unsigned short` vypadá v paměti.



Obrázek 2.3: Číslo 65 535 jako unsigned short v paměti

Datový typ `int` zabírá čtyři bajty. Na obrázku 2.4 je vidět, jak by číslo 65 535 v paměti vypadalo, kdyby bylo uložené jako `int`.



Obrázek 2.4: Číslo 65 535 v paměti, tentokrát jako int

Oprávněně by vás mohlo zajímat, proč zabírá totéž číslo ve formátu `unsigned int` o dva bajty víc než jako `unsigned short`. Jinými slovy, proč se v případě typu `int` vyhradí čtyři bajty, když by stačily dva. Odpověď zní, že v okamžiku vyhrazení paměti ještě není jasné, jaká hodnota bude v paměti uložena. Navíc se hodnota může časem změnit. Proto se vždy vyhradí tolik bajtů, kolik je potřeba pro uložení největší možné hodnoty daného typu.

K čemu jsou menší datové typy

Také by vás mohlo zajímat, proč vlastně vůbec používat typ `short`, když nám `int` nabízí mnohem větší rozsah. Odpověď zní, že za větší rozsah platí `int` svou cenu: zabere dvojnásobek paměti, čtyři bajty namísto dvou.

Mohli byste namítnout, že dnešní počítače mají stovky megabajtů operační paměti a každý z megabajtů má 1 048 576 bajtů. Pár bajtů sem, pár bajtů tam – komu na tom záleží? Kdyby šlo skutečně jen o dva bajty, nezajímaly by nikoho. Kdybyste ale psali program pro pojišťovnu, která má milion klientů, byly by ze dvou bajtů rázem dva megabajty. Proto byste neměli po větších datových typech sahat automaticky.

Přesto platí, že `int` budete ze zmíněných datových typů používat nejčastěji. Jen je dobré o ostatních typech vědět, možnost volby se neztratí.

Datové typy pro desetinná čísla

Byl jsem odmalička krátkozraký, dokud jsem si nenechal oči opravit laserem. Při tomto zákroku zadá chirurg do programu informace, podle kterých laser odpaří tenoučkou vrstvu některých částí rohovky, zatímco jiné části – vzdálené na pouhé setiny milimetru – nechá netknuté.

Dovedete si představit můj výraz, kdyby mi chirurg řekl, že není žádný troškař a pracuje výhradně v celočíselné aritmetice? Že nějaká desetinná čísla jsou pod jeho úroveň? V ordinaci by po mně zůstala jen díra ve stěně a v dálce na obzoru moje silueta. (Jelikož ale ke svému očaři – který si mimochodem na vysokou školu vydělával jako programátor – ještě pořád chodím a nerad bych si to u něj rozházel, musím rychle dodat, že byl velice přesný a zákrok byl úspěšný.)

Celá čísla se hodí v mnoha případech, kdy zlomky vůbec nedávají smysl. Kdo by například tvrdil, že má 2¾ dítěte? V jiných případech desetinná místa přichází v úvahu, ale jsou zbytečná. Například u vzdáleností často stačí mluvit o devadesáti osmi mílích, nikoliv o 98,177 mílích.

Jindy na desetinných místech záleží hodně. Laserová operace očí je extrémní příklad, ale existuje i řada obyčejnějších. Kdybyste měli průměrný prospěch 3,9, asi by vás překvapilo, kdyby škola devět desetin zahodila a nechala průměr na třech. Podobně by dopadla banka, která by sledovala pohyb peněz jen v celých dolarech. Při milionech transakcí denně by záhy měla jen velice nepřesnou představu o tom, kolik peněz má k dispozici a kolik peněz má kdo na účtu.

Pro případy, ve kterých záleží na číslech napravo od desetinné čárky, jsou určené datové typy s plovoucí desetinnou čárkou. Výraz *plovoucí* znamená, že desetinná čárka tentokrát není na rozdíl od celých čísel umístěná pomyslně na konci čísla, ale může „plavat" kdekoliv uvnitř jeho zápisu. Čísla s plovoucí desetinnou čárkou se někdy označují jako *reálná*. (Ovšem reálným číslům v matematickém slova smyslu neodpovídají.)

Datové typy s plovoucí desetinnou čárkou shrnuje tabulka 2.3. Stejně jako u celočíselných datových typů platí, že uvedené velikosti a rozsahy se mohou v závislosti na operačním systému a překladači lišit.

Tabulka 2.3: Datové typy s plovoucí desetinnou čárkou

Datový typ	Velikost v bajtech	Rozsah
float	4	±3,4E-38 až ±3,4E38
double	8	±1,7E-308 až ±1,7E308
long double	10	±3,4E-4932 ±3,4E4932 (Poznámka českého vydavatele: Rozsah reálných čísel je trochu složitější. Záleží na tom, zda je číslo normalizované, či denormalizované. Nicméně rozsah je od +- absolutně nejmenších čísel až po +- absolutně největší čísla.)

Poznámka: Na mnoha kombinacích překladačů a operačních systémů má `long double` velikost 8 bajtů, nikoliv 10.

Vědecký zápis čísel

Zápis rozsahů v tabulce 2.3 vám možná připadne cizí. Čísla v něm totiž nejsou zapsaná tradičním způsobem, ale takzvanou *vědeckou notací*.

Datové typy s plovoucí desetinnou čárkou zvládnou uložit obrovská čísla, například jedničku se sedmatřiceti nulami, což by mohla být například vzdálenost přes celý vesmír. Stejně tak dobře uloží i čísla nevýslovně malá – například číslo, ve kterém je za desetinnou čárkou sedmatřicet nul a teprve pak jednička, což by mohl být například průměr subatomární částice. Aby se taková čísla nemusela roztahovat přes celou stránku, zapisují se zvláštním způsobem, viz tabulku 2.4.

Tabulka 2.4: Vědecký zápis čísel

Běžný zápis	Vědecký zápis	Zjednodušený vědecký zápis
123,45	$1{,}2345 \times 10^{2}$	1,2345E2
0,0051	$5{,}1 \times 10^{-3}$	5,1E–3
1 200 000 000	$1{,}2 \times 10^{9}$	1,2E9

Ve vědecké notaci se číslo zapisuje jako součin dvou čísel. První z nich, takzvaná *mantisa*, má vždy právě jednu číslici před desetinnou čárkou a libovolný počet číslic za ní. Druhé z čísel je mocnina desítky, která je u velkých čísel kladná a u čísel blízkých nule záporná. Vynásobením těchto dvou čísel dostaneme původní číslo. Zkrácená vědecká notace je totéž co běžná vědecká notace, jen znaménko součinu a desítka se nahradí písmenem E (za kterým zůstane původní exponent).

Způsob uložení v paměti

Jelikož v paměti mohou být jen nuly a jedničky, pro uložení čísel s plovoucí desetinnou čárkou se musí používat složité kódování, jehož popis už je za hranicí této knihy. I přes složité kódování ale počítač mnoho desetinných čísel dokáže uložit jen přibližně. Proto si programátoři u některých programů musí dávat pozor, aby se drobné nepřesnosti nesčítaly a nevedly k vážnějším chybám ve výpočtech.

Poznámka: Výpočty v plovoucí desetinné čárce jsou náročné na výpočetní výkon, a tak se často řeší specializovaným obvodem, takzvanou FPU (*floating point unit*). Ta dříve mívala podobu samostatného procesoru, takzvaného ***matematického koprocesoru***; dnes už bývá integrovaná do hlavního procesoru.

Textové datové typy

Textové datové typy jsou dva. První se jmenuje *char* neboli znak. Většinou zabírá jeden bajt a můžete do něj uložit libovolný jeden znak – písmeno, číslici, interpunkci, mezeru a podobně. Druhým textovým datovým typem je řetězec neboli *string*. Do toho se vejde větší počet znaků, například tato věta, odstavec nebo celá stránka. Bajtů zabere podle toho, kolik je v něm uložených znaků.

Poznámka: Na rozdíl od datového typu char a dalších typů, které jsme probírali, není `string` přímou součástí C++. Je součástí standardní knihovny a definovaný je v souboru `string`. Pokud ho chcete používat, musíte tento soubor příkazem `#include <string>` vložit. Podrobné informace o direktivě `include` najdete v první kapitole u příkladu Hello world.

Způsob uložení v paměti

Typ `char` nezabírá jeden bajt náhodou. Ještě v dobách dálnopisu vznikl kdysi v Americe standard ASCII, který popisuje kódování znaků pomocí čísel (například A=65, Z=90 a podobně). Obsahuje celkem 128 znaků: 94 tisknutelných, jednu mezeru a 33 řídicích znaků, které mění způsob zpracování textu. K uložení libovolného ze 128 znaků je potřeba celkem 7 bitů, 2^7=128.

Když vznikaly první počítače, sáhly po tomto sedmibitovém kódování, které se dobře vešlo do jednoho bajtu. ASCII kódy některých běžně používaných znaků najdete v tabulce 2.5. Například písmeno J má ASCII kód 74, binárně 1001010. Když na nějaké adrese v paměti najdete hodnotu 1001010, mohlo by to být písmeno J.

Poznámka: Binární hodnota 1001010 by ale stejně dobře mohla znamenat číslo 74. Záleží na tom, jaký datový typ je s daným kusem paměti spojený. V následující kapitole se budeme věnovat proměnným, které slouží právě ke spojení datového typu s určitou adresou v paměti.

Bajt má osm bitů, a tak zůstane po uložení ASCII kódu nějakého znaku jeden bit nevyužitý. Proto hodně lidí napadlo, že by se dala tabulka ASCII rozšířit na 256 znaků, aby se využila plná kapacita jednoho bajtu. A že bylo proč rozšiřovat – původní tabulka ASCII totiž neobsahuje žádné znaky s diakritikou, počítá jen se základními anglickými písmeny. Problém je v tom, že rozšíření ASCII

vznikala v různých částech světa živelně a do velké míry nezávisle, takže nejsou vzájemně kompatibilní. Zatímco v jednom kódování je číslo 130 vyhrazené písmenu é, jinde tento kód znamená hebrejský znak gimel (ג). Nejčastěji používané kódování znaků pro jazyky střední Evropy jsou ISO-8859-2 a Windows-1250.

Problémy s kódováním národních znaků nějakou dobu přetrvávaly, až je nakonec vyřešil nový standard Unicode. Ten počítá se všemi znaky světa, a dokonce i některými znaky cizích světů. Tolik znaků už ovšem neuloží do jednoho bajtu, a tak jsou v C++ pro Unicode nové dva typy: `wchar_t` pro uložení Unicode znaku a `wstring` pro Unicode řetězce. My se v knize budeme pro jednoduchost držet obyčejných typů `char` a `string`, ale až budete psát opravdový program, měli byste se na typy `wchar_t` a `wstring` podívat.

Tabulka 2.5: ASCII kódy běžně používaných znaků

Znaky	ASCII kódy
0–9	48–57
A–Z	65–90
a–z	97–122

Datový typ bool

Kromě popsaných základních datových typů existuje ještě typ `bool`. Ten má jen dvě možné hodnoty, pravdu (`true`) a nepravdu (`false`), a v paměti většinou zabírá jeden bajt. Jeho název je zkratka od příjmení britského matematika George Boolea a souvisí s takzvanou booleovskou algebrou.

Zmiňujeme ho samostatně, protože do škatulky číselných ani textových datových typů příliš nezapadá. Za číselný datový typ by se dal považovat, kdybychom nulu brali jako nepravdu a jedničku (nebo libovolné další číslo) jako pravdu. Možná vám nepřijde intuitivní, že by zrovna nula měla být nepravda a jednička pravda – stačí si uvědomit, že počítače informace v podstatě ukládají pomocí přepínačů, které jsou buď zapnuté (1), nebo vypnuté (0).

Projekt: Jak zjistit velikost datových typů

Jak bylo řečeno v předchozím textu, velikost datových typů závisí na vašem překladači a operačním systému. V rámci tohoto projektu se naučíte zjistit velikost datových typů na vašem počítači pomocí operátoru `sizeof`.

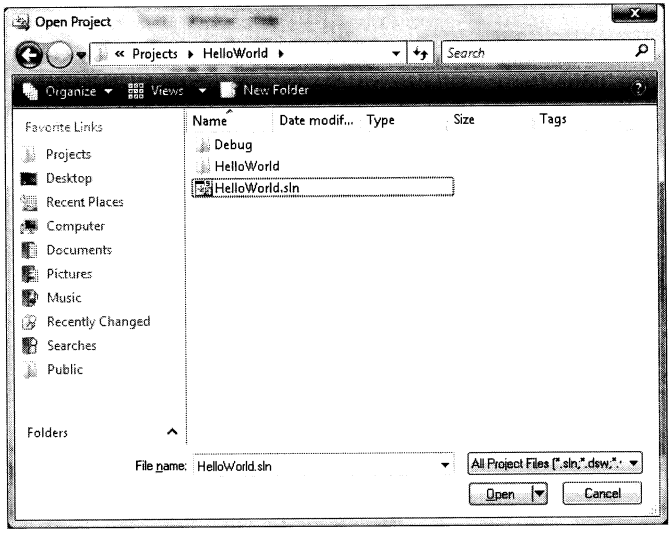
Operátor sizeof

Operátor `sizeof` vrací velikost datového typu v bajtech, stačí napsat `sizeof` a název datového typu v závorkách. Například na mém počítači má výraz `sizeof(int)` hodnotu 4. Znamená to, že při překladu tímto překladačem má typ `int` na mém počítači velikost čtyři bajty.

Změna zdrojového souboru našeho projektu

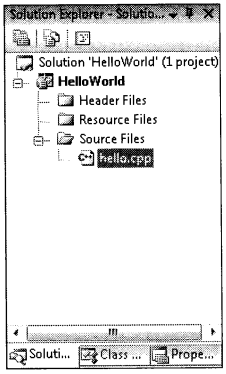
Zkusme si vytvořit a spustit následující program za pomocí postupu z první kapitoly, kde jsme vytvářeli program „Hello World!“. Mohli bychom vytvořit nový projekt, ale zkusíme znovu použít projekt z kapitoly 1. Je dobré znát obě varianty; jak vytvořit nový projekt a jak znovupoužít stávající projekt.

1. Spusťte Visual C++ 2008 Express Edition
2. Použijte příkaz z menu File → Open → Project/Solution k zobrazení dialogu pro otevření řešení (viz obrázek 2.5).



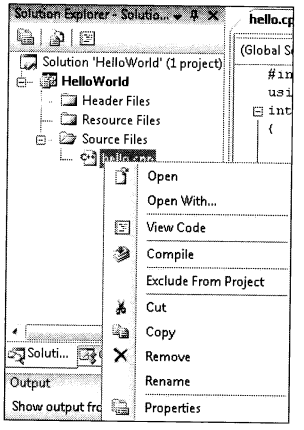
Obrázek 2.5: Otevření existujícího řešení

3. Přepněte se do adresáře, kde jste uložili svůj projekt, a najděte soubor řešení (solution file). Má příponu .sln, jako „solution". Soubor řešení je helloworld.sln, tak jak je ukázáno na obrázku 2.5.
4. Otevřete soubor s řešením. Mělo by dojít k otevření vašeho projektu.
5. Pokud je průzkumník řešení (Solution Explorer) skrytý, zobrazte ho pomocí příkazu v menu View → Solution Explorer. Rozbalte složku „Source Files" (zdrojové soubory) tak, abyste viděli soubor hello.cpp (viz obrázek 2.6).



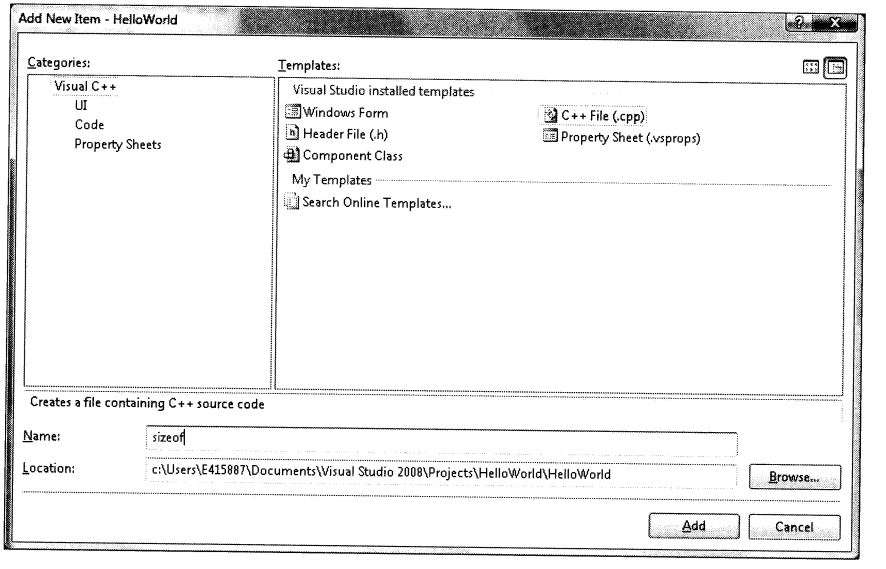
Obrázek 2.6: Existující zdrojový soubor v průzkumníku řešení

6. Klepněte pravým tlačítkem na soubor hello.cpp a z kontextového menu vyberte položku „Remove" – odstranit (viz obrázek 2.7). Nebojte se, nedojde ke smazání souboru z disku. IDE se vás zeptá, zda chcete soubor pouze odstranit z projektu či vymazat. Vyberte volbu „Remove" – odstranit. Vybraný soubor se pouze odstraní z projektu. Pokud si to budete přát, stále jej budete moct kdykoliv později použít.



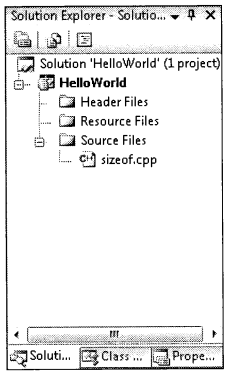
Obrázek 2.7: Položka „Remove" k kontextovém menu

7. Klepněte pravým tlačítkem myši na složku „Source Files" – zdrojové soubory a z kontextového menu vyberte příkaz „Add New Item" – přídání nové položky. Objeví se námi známý dialog, jenž je zobrazen na obrázku 2.8.



Obrázek 2.8: Přidání nového zdrojového souboru do projektu

8. Neměňte hodnotu pole „Location" – umístění. Nový soubor bude umístěn do složky projektu. Zadejte jméno nového zdrojového souboru do pole „Name", nazvěte ho sizeof.cpp.
9. Jakmile budete hotovi, klepněte na tlačítko „Open". Obrázek 2.9 ukazuje nový soubor sizeof.cpp v průzkumníku řešení.



Obrázek 2.9: Průzkumník řešení a nový soubor sizeoff.cpp

Poklepejte na soubor sizeoff.cpp v průzkumníku řešení (Solution Explorer). Soubor se otevře v editoru kódu. Nyní je sizeoff.cpp prázdný. V následujících sekcích do něj přidáme kód.

Zdrojový kód a výstup

Do zdrojového souboru, který jste vytvořili, zadejte následující kód. Jak funguje, si vysvětlíme v následujícím textu.

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << "Velikost typu short: " << sizeof(short) << "\n";
    cout << "Velikost typu int: " << sizeof(int) << "\n";
    cout << "Velikost typu long: " << sizeof(long) << "\n";
    cout << "Velikost typu float: " << sizeof(float) << "\n";
    cout << "Velikost typu double: " << sizeof(double) << "\n";
    cout << "Velikost typu long double: " << sizeof(long double) << "\n";
    cout << "Velikost typu char: " << sizeof(char) << "\n";
    cout << "Velikost typu bool: " << sizeof(bool) << "\n";
    return 0;
}
```

Teď projekt přeložte a spusťte, stejně jako jsme to udělali s *Hello world* v první kapitole. Na mém počítači je výstup následující:

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

Poznámka: Čísla zobrazená na vašem počítači se mohou lišit, protože velikost datového typu závisí na operačním systému a překladači. Je docela pravděpodobné, že se alespoň v jednom z nich lišíme.

Výrazy

Řádek zdrojového kódu:

```
cout << "Velikost typu int: " << sizeof(int) << "\n";
```

se na výstupu programu projeví takto:

```
Velikost typu int: 4
```

V principu se tedy kód `sizeof(int)` na výstupu nahradí čtyřkou. Kódu typu `sizeof(int)` se říká *výraz*. Výraz je kus kódu, který má nějakou hodnotu – obvykle hodnotu, která se musí zjistit za běhu programu. Jednoduchým výrazem je například kód `4+4`, který se za běhu vyhodnotí na osmičku. V našem případě se výraz `sizeof(int)` za běhu programu vyhodnotí na čtyřku, která se pak vypíše na obrazovku.

Naproti tomu část příkazu uzavřená v uvozovkách (`"Velikost typu int:"`) se na výstup dostane beze změn. Není co vyhodnocovat. Této části se říká *řetězcový literál* – výraz *řetězcový* označuje datový typ a *literál* znamená, že se má brát doslova, bez dalšího vyhodnocování. Příkladem řetězcového literálu byl i řetězec `"Hello, world!"` v první kapitole.

Jak vypsat výraz

Výraz `sizeof(int)` je od literálu `"Velikost typu int:"` oddělený operátorem vložení do proudu (`<<`). Co kdyby kód vypadal takto?

```
cout << "Velikost typu int: sizeof(int)\n";
```

Výstup by dopadl úplně jinak:

```
Velikost typu int: sizeof(int)
```

Protože jakmile výraz `sizeof(int)` uzavřete do uvozovek, přestane se brát jako výraz. Bere se jako obyčejný literál, a tudíž se vypíše bez vyhodnocování. Jelikož je `"Velikost typu int:"` literál a `sizeof(int)` výraz, musí se před vložením do výstupního proudu oddělit operátorem `<<`.

Poznámka: V řetězci `"Velikost typu int: "` je za dvojtečkou ještě mezera. Kdybyste ji vypustili, na výstupu byste dostali `Velikost typu int:4`. Na mezery musíte dávat pozor vy jako programátoři, C++ to za vás neudělá.

Jak vypsat speciální znaky

Řetězec `"\n"` za výrazem `sizeof(int)` je také literál, takže je od výrazu oddělený operátorem `<<` a nebude se nijak vyhodnocovat. Přeci jen to ale není úplně obyčejný řetězec – obsahuje totiž zpětné lomítko (`\`), které má zvláštní význam. Když překladač v řetězci narazí na zpětné lomítko, pozná, že další znak řetězce nemá brát doslova. Když například za zpětným lomítkem najde malé písmeno `n`, vloží do řetězce znak nového řádku. Několik nejpoužívanějších zvláštních znaků najdete v tabulce 2.6.

Tabulka 2.6: Běžně používané speciální znaky

Znaky	Popis
`\a`	při výpisu pípne
`\n`	konec řádky
`\t`	tabulátor
`\\`	zpětné lomítko
`\'`	apostrof
`\"`	uvozovky

Shrnutí

Aby program fungoval, jeho příkazy a data musí být v operační paměti. Kromě operační paměti neboli RAM má počítač ještě další dvě zásadní centra paměti: vyrovnávací paměť u procesoru (+ registry procesoru) a trvalou paměť, například pevný disk. Z našeho pohledu je nejdůležitější operační paměť, ve které se uchovávají příkazy a data běžících programů.

Příkazy a data jsou v paměti uložené v podobě jedniček a nul na číselných adresách. Nejmenší jednotce informace, která může být jen jednička nebo nula, se říká *bit*. Tak malá jednotka je ale nepraktická na práci, a tak se většinou jako nejmenší jednotka používá *bajt* neboli osmice bitů. Na každé adrese v paměti je uložený jeden bajt.

Některé informace mají podobu číselnou, jiné textovou. Podrobněji by se datové typy daly rozdělit na celá čísla, desetinná čísla (neboli čísla s plovoucí desetinnou čárkou) a text. Všechny datové typy mají nějakou velikost, zabírají v paměti určitý počet bajtů. Velikost číselných typů určuje i jejich *rozsah*, tedy nejmenší a největší hodnotu, kterou typ zvládne uložit.

Velikost datových typů závisí na překladači a operačním systému. Velikost typů na vašem systému můžete zjistit pomocí operátoru `sizeof`.

Test

1. Do jaké z těchto pamětí má procesor nejblíže – vyrovnávací, operační, nebo trvalé?
2. Která z pamětí není dočasná – cache, RAM, nebo pevný disk?
3. Kolik informací se dá uložit na jedné paměťové adrese?
4. Je velikost datových typů C++ nezávislá na počítači?
5. Co je rozsah datového typu?
6. Jaký je rozdíl mezi datovými typy se znaménkem a bez?
7. Jaké číslo se skrývá pod zápisem 5,1E–3?
8. Co je to ASCII kód?
9. Co dělá operátor `sizeof`?
10. Co je to řetězcový literál?
11. Co je to výraz?

Proměnné

Když na mě nedávno kdosi v plné místnosti zavolal *hej*, otočilo se kromě mě i hezkých pár dalších lidí. Nikdo z nás totiž netušil, na koho ten pán mluví. Kdyby takhle rovnou zakřičel *hej, Jeffe*, hned bych věděl, že mluví na mě. (Ledaže by samozřejmě v místnosti bylo Jeffů víc.)

My lidé říkáme jeden druhému jménem. S programováním je to podobné – když se potřebujete odkázat na jeden konkrétní údaj třeba i z tisíců dalších, můžete ho pojmenovat.

K pojmenování informací slouží proměnné. Proměnná vám jednak umožní odkazovat se na nějakou informaci jménem, a jednak pro ni vyhradí potřebné místo v paměti. V této kapitole si ukážeme, jak se proměnné vytváří, jak se do nich ukládají informace a jak se z nich informace zase získávají nazpět.

Deklarování proměnných

V předchozí kapitole jste se dozvěděli, že pokud má program pracovat s nějakými informacemi, musí je mít uložené v operační paměti. A než můžete do paměti něco uložit, musíte si vyhradit potřebné místo. Právě k tomu slouží takzvané *deklarování proměnných*.

Deklarováním proměnné si vyhradíte místo v paměti a získáte pohodlný způsob, jak se na vyhrazenou paměť v programu odkazovat. V předchozí kapitole jsme se bavili o tom, že adresy v paměti jsou čísla, nadto zapisovaná většinou v šestnáctkové soustavě (například 0012fed4). Takové hodnoty se dost špatně pamatují. Pokud se daná informace týká řekněme výsledku testu, mnohem lépe by se nám pamatovala například pod názvem `pocetBodu`. Po deklarování proměnné se můžete na paměť odkazovat jménem proměnné, což je mnohem jednodušší, než si pamatovat šestnáctkovou adresu.

Deklarovat proměnnou není nijak těžké, stačí řádek kódu. Za oponou se toho ale děje poměrně hodně. Program na konci této části vám ukáže, jak se dá zjistit adresa a velikost paměti vyhrazené deklarací proměnné.

Jak deklarovat proměnnou

Proměnnou musíte deklarovat ještě před tím, než ji začnete používat. Deklarace proměnné vypadá následovně:

```
[datový typ] [jméno proměnné];
```

Jako datový typ můžete uvést například libovolný z datových typů popisovaných v předchozí kapitole, tedy `int`, `float`, `bool`, `char` nebo `string`. Podle datového typu počítač pozná, kolik paměti má na proměnnou vyhradit. V předchozí kapitole jsme se bavili o tom, že různé proměnné mají různé paměťové nároky. Když deklarujete proměnnou, která má při překladu vaším překladačem na vašem operačním systému velikost čtyři bajty, počítač pro ni vyhradí čtyři bajty paměti.

Jméno proměnné je na vás; pojmenování proměnných se budeme věnovat později v této kapitole. Jméno proměnné bude ve zdrojovém kódu sloužit jako zástupce pro vyhrazenou paměť. Když tedy deklarujete proměnnou `pocetBodu`, do které budete chtít ukládat výsledky testu, můžete se v kódu na vyhrazenou paměť odkazovat pod jménem `pocetBodu` a nemusíte se zajímat o žádné šestnáctkové adresy.

Deklarace proměnné končí středníkem, podle kterého překladač pozná, že příkaz skončil. Proměnnou můžete deklarovat uvnitř funkce (například uvnitř funkce `main`) nebo též nad všemi funkcemi (hierarchicky myšleno), těsně pod případnými direktivami `#include`. Naše programy mají prozatím jen jednu funkci (`main`), a tak budeme všechny proměnné deklarovat v ní. Až se nám počet funkcí rozroste, k deklarování proměnných se vrátíme.

Následující program deklaruje ve funkci `main` celočíselnou proměnnou jménem `pocetBodu`:

```
int main(void)
{
    int pocetBodu;
    return 0;
}
```

Poznámka: Všimněte si, že na rozdíl od prvních dvou kapitol nepotřebujeme žádnou direktivu `#include`, protože nepoužíváme `cout` ani žádné další objekty ze standardní knihovny.

Když se pokusíte o proměnné zmínit ještě před její deklarací, překladač ohlásí chybu. V následujícím kódu si první zmínka o proměnné `pocetBodu` vyslouží chybové hlášení o nedeklarovaném identifikátoru:

```
int main(void)
{
    pocetBodu;
    int pocetBodu;
    return 0;
}
```

Překladač ohlásí chybu, přestože proměnnou deklarujete hned na následujícím řádku. Zdrojový kód totiž překladač čte odshora dolů, takže když dorazí na první zmínku o identifikátoru `pocetBodu`, deklaraci proměnné ještě neviděl.

Chyba s nedeklarovanou proměnnou se podobá chybě, kterou jsme dostali u *Hello world* z první kapitoly po záměrném přejmenování `cout` na `Cout`. Jméno proměnné `pocetBodu` nepatří na rozdíl od `main`, `int` a podobných jmen přímo do jazyka C++, a tak ho překladač sám od sebe nepozná. Teprve když proměnnou deklarujete, překladač pozná všechny další výskyty jejího jména jako odkaz na tuto proměnnou.

Deklarování více proměnných najednou

Když potřebujete několik proměnných téhož typu, můžete je deklarovat jednotlivě:

```
int pocetBodu;
int mojeVyska;
int mojeVaha;
```

Pokud jsou ale stejného typu, nemusíte mít pro každou samostatný příkaz. Můžete je všechny deklarovat jedním příkazem, v rámci kterého je oddělíte čárkou:

```
int pocetBodu, mojeVyska, mojeVaha;
```

V tomto případě se datový typ uvádí jen jednou, i když deklarujete tři proměnné. Jelikož jsou všechny tři proměnné deklarované jedním příkazem, datový typ bude platit pro všechny společně. Takto společně se dají deklarovat pouze proměnné stejného typu. Dvě proměnné typu `int` a `float` už jedním příkazem deklarovat nejde, ty musíte deklarovat zvlášť:

```
int pocetBodu;
float prumer;
```

Pojmenování proměnných

Proměnné mají stejně jako lidé jména, kterými se na ně můžete odkazovat v kódu. Pro pojmenování proměnných platí několik pravidel:

- Název proměnné nesmí začínat jinak než písmenem abecedy (A–Z, a–z) nebo podtržítkem (_). Tajný agent se 007 jmenovat může, proměnná nikoliv. Počínaje druhým znakem už může jméno obsahovat i číslice.
- Název proměnné nesmí obsahovat mezery (`moje promenna`) ani interpunkci; jedinou výjimkou je podtržítko (_).
- Názvy proměnných se musí vyhýbat vyhrazeným slovům jazyka samotného, například `main` nebo `int`.
- Proměnná se nesmí jmenovat stejně jako jiná proměnná ve stejném rozsahu platnosti. (Rozsahu platnosti se budeme podrobně věnovat v osmé kapitole.) Stručně řečeno to prozatím znamená, že ve funkci `main` nemůžete deklarovat dvě shodně pojmenované proměnné.

S ohledem na tato pravidla už můžete proměnné pojmenovat téměř libovolně. Celkem dobrý nápad je vybírat jména, která vám něco řeknou. Když své proměnné budete označovat `prom1`, `prom2`, `prom3` a tak dál až do `prom17`, asi budete mít časem problém vzpomenout si na rozdíl mezi `prom8` a `prom9`. (A když s tím budete mít problém už vy sami, co potom asi bude říkat váš kolega, který kód vůbec nepsal.)

Pokud si chcete zachovat příčetnost (a případně i zdraví, pokud máte nervóznější kolegy), nejlepší pro vás bude volit názvy proměnných podle jejich určení. Například `pocetBodu` je slušný popisný název proměnné, která reprezentuje výsledek testu.

Název `pocetBodu` vznikl spojením dvou slov, *počet* a *bodů*. Mezera se nesmí objevit v názvu proměnné, takže `pocet bodu` se proměnná jmenovat nemůže. (Poznámka českého vydavatele: Ani diakritika se v názvech proměnných nepoužívá. Teoreticky tomu nic nebrání, v praxi byste narazili na problémy.) Proto se slova musí spojit dohromady a odliší se velkým písmenem na začátku druhého slova. Názvy proměnných bývá zvykem psát s malým písmenem na začátku.

Jmenné konvence

V praxi se programátoři při pojmenování proměnných drží nějakého jednotného systému, takzvané jmenné konvence. Jednu jmennou konvenci jsme právě popsali. Další možnost je přidat před každý název proměnné třípísmennou předponu, která označuje typ proměnné:

- `intBody`, celočíselný počet bodů, například z písemky.
- `strJmeno`, řetězec se jménem, například jméno osoby.
- `blnMistni`, booleovská proměnná, podle které se pozná, jestli je člověk místní.

Je celkem jedno, kterou jmennou konvenci se rozhodnete používat. Důležité je nějakou vybrat a držet se jí.

Operátor adresy

Při deklaraci proměnné dojde k vyhrazení paměti. Adresu této rezervované paměti dostanete takzvaným *operátorem adresy* (`&`). Jeho zápis vypadá následovně:

```
&[název proměnné]
```

Například následující kód vypíše na mém počítači hodnotu `0012fed4`. U vás bude vypsaná adresa proměnné `pocetBodu` nejspíš jiná – dokonce i na mém počítači se může při pozdějším spuštění programu lišit.

```
#include <iostream>
using namespace std;
int main(void)
{
    int pocetBodu;
    cout << &pocetBodu;
    return 0;
}
```

Adresa `0012fed4` je v šestnáctkovém zápisu, jak už bývá u adres zvykem (viz předchozí kapitola). Adresu proměnné vybírá operační systém, nikoliv programátor. Vybraná adresa závisí na typu proměnné, aktuální velikosti obsazené paměti a dalších vlivech.

Vy se o adresy starat nemusíte, protože se budete na proměnnou odkazovat jménem, nikoliv adresou. Jak ale zjistíte v jedenácté kapitole, operátor adresy je přesto velice užitečný.

Použití operátorů & a sizeof na proměnné

Velikost paměti, která se na proměnnou vyhradí, závisí na typu proměnné. Už v předchozí kapitole jsme si říkali, že různé typy mají různou velikost. Zároveň jsme si zkoušeli operátorem `sizeof` zjistit velikost různých datových typů na vašem operačním systému a překladači. Stejným způsobem se pomocí operátoru `sizeof` dá zjistit velikost různých proměnných (která opět závisí na operačním systému a překladači).

Velikost proměnné se operátorem `sizeof` určuje skoro stejně jako velikost datového typu, jen do závorek za `sizeof` napíšete místo názvu datového typu název proměnné. Následující kód vypíše adresu a velikost dvou proměnných:

```
#include <iostream>
using namespace std;
int main(void)
{
    short pocetBodu;
    float prumer;
    cout "Adresa promenne pocetBodu: " << &pocetBodu << "\n";
    cout "Velikost promenne pocetBodu: " << sizeof(pocetBodu) << "\n";
    cout "Adresa promenne prumer: " << &prumer << "\n";
    cout "Velikost promenne prumer: " << sizeof(prumer) << "\n";
    return 0;
}
```

Výstup programu se bude počítač od počítače lišit, u mě vypadá takto:

```
Adresa promenne pocetBodu: 0012FED4
Velikost promenne pocetBodu: 2
Adresa promenne prumer: 0012FEC2
Velikost promenne prumer: 4
```

Rozložení paměti pro tyto dvě proměnné je vidět na obrázku 3.1. Obě proměnné jsou jinak veliké – `pocetBodu` je typu `short` a zabírá dva bajty, `prumer` je typu `float` a bajty zabírá čtyři.

float prumer					short pocetBodu	
0012FEC8	0012FEC9	0012FECA	0012FEFB	....	0012FED4	0012FED5

Obrázek 3.1: Paměť vyhrazená pro dvě proměnné

Na obrázku je vidět, že jsou obě proměnné v paměti blízko u sebe. O takové umístění se operační systém snaží záměrně, ale ne vždy se mu, podaří. Záleží na velikosti proměnných, obsazení paměti a dalších věcech. Nikdo vám nemůže zaručit, že proměnné v paměti neskončí úplně jinde.

Hodnota paměti na obou adresách z obrázku 3.1 je neznámá. Ještě jsme totiž neřekli, jaká hodnota by se na tyto adresy měla uložit – jak se to dělá, se dozvíte v následující části.

Přiřazování hodnot

Úkolem proměnné je ukládat informace. Když tedy vytvoříte nějakou proměnnou, dalším krokem by logicky mělo být určení hodnoty, kterou má proměnná obsahovat. Tomu se říká *přiřadit* proměnné hodnotu.

Proměnné může hodnotu přiřadit programátor přímo ve zdrojovém kódu. Stejně tak může hodnotu proměnné zprostředkovaně nastavit uživatel za běhu programu, například zadáním z klávesnice.

Hodnota se do proměnné ukládá operátorem přiřazení, kterému se budeme věnovat v následujícím textu. O něco dál pak přijde na řadu objekt `cin`, jehož prostřednictvím se dá proměnné přiřadit vstup od uživatele (například z klávesnice).

Operátor přiřazení

K přiřazení hodnoty proměnné slouží operátor přiřazení:

```
[název proměnné] = [hodnota];
```

Jelikož rovnítko slouží jako operátor přiřazení, pro test rovnosti se musí použít něco jiného. Jak se ukáže v páté kapitole, rovnost se v jazyce C++ zkouší operátorem ==, takzvaným operátorem rovnosti.

Ještě než proměnné přiřadíte nějakou hodnotu, musíte ji deklarovat. První řádek následujícího příkladu proměnnou deklaruje, druhý jí přiřadí hodnotu:

```
int pocetBodu;
pocetBodu = 95;
```

Deklaraci proměnné a přiřazení hodnoty můžete provést v rámci jednoho příkazu. Takovému přiřazení se říká *inicializace* proměnné:

```
int pocetBodu = 95;
```

Proměnnou ovšem nemůžete deklarovat až po přiřazení. U první řádky následujícího příkladu by překladač skončil s chybou, stěžoval by si na nedeklarovaný identifikátor:

```
pocetBodu = 95;
int pocetBodu;
```

Jak už jsme říkali výše v textu o deklaraci proměnných, překlad skončí chybou, i když je proměnná deklarována hned za přiřazením. Překladač totiž zdrojový kód čte odshora dolů, takže když dojde na řádek s přiřazením, o deklaraci proměnné ještě neví.

Přiřazovaná hodnota nemusí být jen literál, například `95`. Následující kód přiřadí jedné celočíselné proměnné hodnotu jiné celočíselné proměnné:

```
int a, b;
a = 44;
b = a;
```

Nejprve se do proměnné `a` uloží hodnota `44`. Na dalším řádku se pak hodnota proměnné `a`, tedy `44`, přiřadí proměnné `b`. Přiřazovat můžete i několika proměnným najednou, následující kód přiřadí hodnotu `0` třem celočíselným proměnným:

```
int a, b, c;
a = b = c = 0;
```

Zde přiřazení probíhá ve třech krocích:

1. Proměnné `c` se přiřadí hodnota `0`.
2. Proměnné `b` se přiřadí aktuální hodnota proměnné `c`, tedy `0`.
3. Proměnné `a` se přiřadí aktuální hodnota proměnné `b`, tedy `0`.

A konečně můžete hodnotu přiřadit i proměnné, která už nějakou hodnotu má. Od toho se proměnným říká „proměnné" – jejich hodnota se může měnit. Změnu hodnoty proměnné ukazuje následující kód:

```
#include <iostream>
using namespace std;
int main(void)
{
    int pocetBodu;
    pocetBodu = 95;
    cout << "Vysledek testu: " << pocetBodu << " bodu.\n";
    pocetBodu = 75;
    cout << "A ted: " << pocetBodu << " bodu.\n";
    return 0;
}
```

Výstup programu vypadá takto:

```
Vysledek testu: 95 bodu.
A ted: 75 bodu.
```

Kompatibilita datových typů I

Hodnota, kterou proměnné přiřazujete, musí být kompatibilní s datovým typem proměnné. Zjednodušeně řečeno to například znamená, že proměnným číselného typu můžete přiřazovat opět jen čísla. Nekompatibilní přiřazení vypadá například takto:

```
int pocetBodu;
pocetBodu = "Jeff";
```

Kdybyste se ho pokusili přeložit, překladač by skončil s chybou, že nemůže převést `const char[5]` na `int`. Tím vám říká, že se pokoušíte přiřadit řetězec do celočíselné proměnné. To samozřejmě nemůže fungovat, protože `"Jeff"` není žádný zápis čísla.

Přiřazovaná hodnota nemusí typu proměnné odpovídat úplně přesně. V následujícím kódu přiřazujeme celočíselné proměnné `pocetBodu` desetinnou hodnotu `77.83`. Všimněte si, že zatímco čeština používá desetinnou čárku ($\pi \approx 3{,}141$), C++ se drží anglické desetinné tečky. Z toho plyne, že ve zdrojovém kódu musíte desetinná čísla zapisovat s tečkou. Výjimkou jsou řetězcové literály, ve kterých si samozřejmě můžete dělat, co chcete: cout << „$\pi \approx 3{,}141$“, ale cout << „$\pi \approx$ „ << 3.141., výstupem programu je číslo `77`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int pocetBodu;
    pocetBodu = 77.83;
    cout << pocetBodu;
    return 0;
}
```

Desetinná část čísla se do celočíselné proměnné `pocetBodu` uložit nedá, a tak se po přiřazení ztratí.

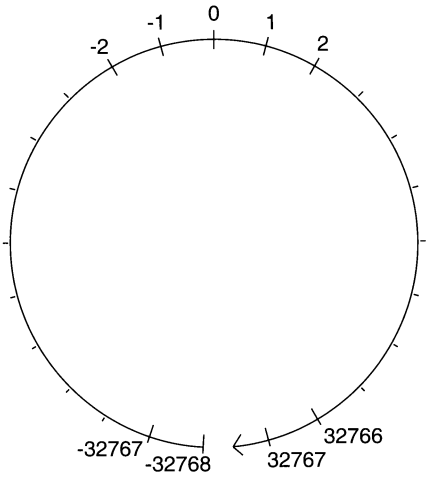
Přetečení a podtečení

Z předchozí kapitoly si možná pamatujete, že rozsah datového typu `short` je od –32 768 po 32 767. Zkuste si přeložit a spustit následující program a podívejte se, co se stane, když do proměnné uložíte číslo sice kompatibilní, ale za hranicí jejího rozsahu:

```
#include <iostream>
using namespace std;
int main(void)
{
    short pocetBodu;
    pocetBodu = 32768;
    cout << pocetBodu;
    return 0;
}
```

Výstupem programu je číslo –32 768. Nikoliv 32 768, ale *minus* 32 768. Tomuto jevu se říká *přetečení*. K přetečení dochází, když proměnné přiřadíte hodnotu, která je na její rozsah příliš velká. V našem případě je číslo 32 768 o jedničku větší než maximální možná hodnota datového typu `short`, takže přeteče a přetočí se na nejnižší možnou hodnotu typu: –32 768.

Podobně by dopadl pokus o přiřazení čísla 32 769, které je o dvojku větší než maximální možná hodnota a uloží se jako –32 767; čísla 32 770, které je o trojku větší než maximální možná hodnota a uloží se jako –32 766, a tak dále. Způsob, kterým k přetečení dochází, je vidět na obrázku 3.2.

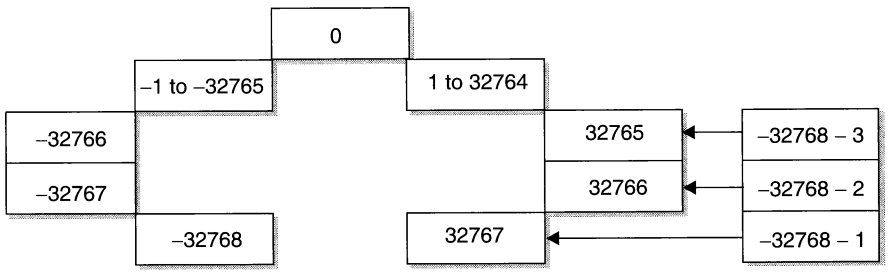


Obrázek 3.2: Přetečení

Opakem přetečení je *podtečení*, ke kterému dochází, když proměnné přiřadíme příliš malou hodnotu. V následujícím příkladu máme opět proměnnou `pocetBodu` typu `short`, ale tentokrát jí přiřadíme hodnotu –32 769. Ta je na datový typ `short` příliš malá, a tak dojde k podtečení a hodnota proměnné se přetočí na nejvyšší možnou hodnotu jejího typu: 32 767.

```
#include <iostream>
using namespace std;
int main(void)
{
    short pocetBodu;
    pocetBodu = -32769;
    cout << pocetBodu;
    return 0;
}
```

Podobně hodnota –32 770 překračuje nejmenší možnou hodnotu datového typu o dva, takže povede k číslu 32 766, hodnota –32 771 překračuje minimum o tři a povede k číslu 32 765 a tak dále. Podtečení je vidět na obrázku 3.3.



Obrázek 3.3: Podtečení

Poznámka: Podtečení a přetečení se týká i datových typů s plovoucí desetinnou čárkou, například typů `float` a `double`. Výsledek v tomto případě závisí na použitém překladači a může způsobit i ukončení programu s chybou.

Objekt cin

Až doposud určoval hodnoty proměnných programátor. Ale programy jsou většinou interaktivní: potřebné informace si od uživatele samy vyžádají, obvykle prostřednictvím klávesnice.

V první kapitole jsme pomocí objektu `cout` vypisovali informace na standardní výstup, tedy obvykle monitor. Teď se podíváme na podobný objekt `cin`, pomocí kterého se informace načítají ze standardního vstupu (většinou z klávesnice). Objekt `cin` je stejně jako `cout` definovaný ve standardním knihovním souboru `iostream`, který tím pádem musíte direktivou `#include` vložit do svého kódu. Objekt `cin` se používá takto:

```
cin >> [jméno proměnné];
```

Za objektem `cin` je operátor `>>`, který slouží ke čtení z proudu. Převezme hodnotu ze vstupu (obvykle z klávesnice) a přiřadí ji proměnné uložené na své pravé straně.

Tip: Operátory `<<` a `>>` se někdy pletou. Je dobré si pamatovat, že ukazují směrem, kterým přenáší data. Například ve výrazu `cin >> prom` se data přesouvají ze standardního vstupu do proměnné, zatímco ve výrazu `cout << prom` se naopak hodnota proměnné posílá na standardní výstup.

Když váš program během provádění narazí na čtení z objektu `cin`, zastaví se a počká, dokud uživatel nezadá nějakou hodnotu a nestiskne klávesu Enter. Zkuste si přeložit a spustit následující ukázku. Uvidíte blikající kurzor, který čeká na zadání čísla. Jakmile číslo zadáte a stisknete Enter, program vypíše `Vysledek testu:` a číslo, které jste zadali. Když zadáte například 100, program vypíše `Vysledek testu: 100`.

```
#include <iostream>
using namespace std;
int main(void)
{
    int pocetBodu;
    cin >> pocetBodu;
    cout << "Vysledek testu: " << pocetBodu << "\n";
    return 0;
}
```

Program ovšem není příliš uživatelsky přívětivý. Pokud ho uživatel nezná předem, nebude vědět, jaké informace od něj program chce. Proto bývá zvykem vypsat dopředu pomocí `cout` pokyny pro uživatele, takzvanou *výzvu*:

```
#include <iostream>
using namespace std;
int main(void)
{
    int pocetBodu;
    cout << "Zadejte pocet bodu: ";
    cin >> pocetBodu;
    cout << "Vysledek testu: " << pocetBodu << "\n";
    return 0;
}
```

Dialog s programem pak vypadá takto:

```
Zadejte pocet bodu: 78
Vysledek testu: 78
```

Kompatibilita datových typů II

Podobně jako u operátoru přiřazení nemusí být hodnota načtená z objektu `cin` stejného datového typu jako proměnná, do které hodnotu ukládáte. Když se vás předchozí ukázkový program zeptá na počet bodů a vy zadáte desetinné číslo 77,83, program vypíše `Vysledek testu: 77`. Desetinná část zadaného čísla se ztratí, protože ji nelze uložit do celočíselné proměnné `pocetBodu`.

Hodnota načtená z objektu `cin` ale musí být s datovým typem cílové proměnné kompatibilní. Kdybyste po dotazu na počet bodů zadali místo čísla řetězec `Jeff`, dostali byste například výstup `Výsledek testu: -858993460`.

Dá rozum, že -858 993 460 bodů by ze zkoušky nechtěl dostat nikdo. Kde se ale takové číslo vzalo? Problém je v tom, že řetězec `Jeff` se do celočíselné proměnné `pocetBodu` uložit nedá. Operátor `>>` proto hodnotu načtenou z `cin` do proměnné `pocetBodu` neuloží a na dalším řádku vypisujeme hodnotu proměnné, které nikdo žádnou hodnotu nepřiřadil.

Když jsme proměnnou `pocetBodu` deklarovali, na její adrese zbyla nějaká hodnota po předchozích programech. V našem případě zbyla v paměti hodnota -858 993 460, a právě tu vypíše `cout` na obrazovku.

Poznámka: Když jsme se na začátku této kapitoly pokusili do proměnné pocetBodu uložit řetězec (pocetBodu = „Jeff"), překladač to označil za chybu. Když jsme teď načetli tutéž hodnotu z objektu cin, dostali jsme místo chyby překladu chybnou hodnotu proměnné. Rozdíl je v tom, že hodnotu zadávanou uživatelem dopředu neznáme – dozvíme se ji až za běhu programu. Překladač si tedy nemá na co stěžovat, protože v době překladu k žádnému pokusu o chybné přiřazení nedošlo.

Načítání více hodnot najednou

Když potřebujete načíst víc proměnných, můžete hodnoty načíst jednu po druhé:

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    int vaha, vyska;
    string jmeno;
    cout << "Zadejte sve jmeno: ";
    cin >> jmeno;
    cout << "Zadejte svou vahu v kilogramech: ";
    cin >> vaha;
    cout << "Zadejte svou vysku v centimetrech: ";
    cin >> vyska;
    cout << "Jmenujete se " << jmeno << "\n";
    cout << "Vazite " << vaha << " kg\n";
    cout << "Merite " << vyska << " cm\n";
    return 0;
}
```

Dialog s programem vypadá například takto:

```
Zadejte sve jmeno: Jeff
Zadejte svou vahu v kilogramech: 90
Zadejte svou vysku v centimetrech: 182
Jmenujete se Jeff
Vazite 90 kg
Merite 182 cm
```

Samostatné výzvy a načítání každé proměnné zvlášť ale můžete nahradit načtením všech hodnot v jednom řádku:

```
cin >> [první proměnná] >> [druhá proměnná] >> třetí proměnná;
```

Tímto způsobem můžete načíst libovolný počet proměnných, stačí jednotlivé proměnné oddělit operátorem `>>`.

Když z `cin` načítáte větší počet proměnných najednou, uživatel musí zadávané hodnoty oddělit alespoň jednou mezerou. Podle mezery `cin` pozná, kde končí hodnota jedné proměnné a začíná hodnota té další. Stejně jako v předchozím případě se vstup ukončí klávesou Enter.

Načítání více proměnných v rámci jednoho příkazu ukazuje následující program:

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    int vaha, vyska;
    string jmeno;
    cout << "Zadejte sve jmeno, vahu v kilogramech a vysku v centimetrech.\n";
    cout << "Jednotlive udaje oddelte mezerou.\n";
    cin >> jmeno >> vaha >> vyska;
    cout << "Jmenujete se " << jmeno << "\n";
    cout << "Vazite " << vaha << " kg\n";
    cout << "Merite " << vyska << " cm\n";
    return 0;
}
```

Dialog mezi uživatelem a programem funguje následovně:

1. Uživatel napíše `Jeff` a mezeru.
2. Podle mezery `cin` pozná, že skončil vstup první hodnoty. Proměnné `jmeno`, která je první na řadě, přiřadí hodnotu `Jeff`.
3. Uživatel zadá číslo `90` a mezeru.
4. Podle mezery `cin` pozná, že skončila další hodnota. Vezme tedy další proměnnou na řadě (`vaha`) a přiřadí jí hodnotu `90`.
5. Uživatel zadá `182` a stiskne Enter.
6. Podle klávesy Enter `cin` pozná, že vstup skončil. Přiřadí tedy zbývající proměnné `vyska` hodnotu `182` a příkaz je hotov.

Zvenčí vypadá dialog takto:

```
Zadejte sve jmeno, vahu v kilogramech a vysku v centimetrech.
Jednotlive udaje oddelte mezerou.
```

```
Jeff 90 182
Jmenujete se Jeff
Vazite 90 kg
Merite 182 cm
```

Kompatibilita datových typů III

V rámci jednoho příkazu můžete z `cin` načíst proměnné různého datového typu. V našem případě je první proměnná řetězec, zatímco druhé dvě jsou celá čísla.

Podstatné je, aby pořadí hodnot na vstupu odpovídalo pořadí načítaných proměnných. Hodnoty `Jeff`, `90` a `182` se proměnným přiřadí v pořadí, ve kterém jsou proměnné uvedené v příkazu: `Jeff` přijde do proměnné `jmeno`, `90` do proměnné `vaha` a `182` do proměnné `vyska`.

Jak důležité je, aby pořadí hodnot odpovídalo pořadí proměnných, si můžete sami vyzkoušet: Změňte hodnoty z `Jeff`, `90` a `182` na `90`, `Jeff` a `182`. Výstup programu bude vypadat takhle:

```
Zadejte sve jmeno, vahu v kilogramech a vysku v centimetrech.
Jednotlive udaje oddelte mezerou.
90 Jeff 182
Jmenujete se 90
Vazite -858993460 kg
Merite 4096 cm
```

Já bych sice rád shodil, ale –858 993 460 kg už je vážně trochu extrém. Je docela pochopitelné, proč se řetězec `Jeff` nepodařilo přiřadit jako hodnotu proměnné `vaha`. Jak je ale možné, že není správně nastavená ani výška?

Jediná správně nastavená hodnota je jméno. Řetězec může být cokoliv, včetně číslic. Devadesátka je sice zvláštní jméno, ale jako řetězec je pro `cin` naprosto vyhovující, a tak se řetězec `90` uloží do proměnné `jmeno`.

Proč je v proměnné `vaha` uložena hodnota –858 993 460, už jsme probírali před chvilkou, když jsme se do celočíselné proměnné s počtem bodů pokoušeli dostat hodnotu `Jeff`. Ale hodnota `182` by určená do proměnné `vyska` by měla být naprosto v pořádku, proč se nevypisuje na výstupu?

Problém je v tom, že na další přiřazení nečeká hodnota `182`, ale `Jeff`. Když totiž `cin` nemohl přiřadit hodnotu `Jeff` do proměnné `vaha`, nechal si ji v zásobě na další přiřazení, tentokrát proměnné `vyska`. Ani tentokrát se ale přiřazení nepovede, a tak proměnné `vyska` i `vaha` zůstanou nastavené na výchozí hodnotu.

Jak načíst víc slov do jednoho řetězce

Dále je dobré si uvědomit, že `cin` načte z každého řetězce jen první slovo. Když následujícímu programu zadáte vstup `Jeff Kent`, vypíše `Jmenujete se Jeff`, nikoliv `Jmenujete se Jeff Kent`:

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    string jmeno;
    cout << "Zadejte sve jmeno: ";
    cin >> jmeno;
```

```
    cout << "Jmenujete se " << jmeno << "\n";
    return 0;
}
```

Příjmení program nevypíše, protože `cin` považuje mezeru za křestním jménem za konec hodnoty proměnné `jmeno`. Řešení nabízí metody `get` a `getline` objektu `cin`, k oběma se dostaneme v desáté kapitole.

Shrnutí

Proměnné mají dvojí účel: Za prvé nám dávají možnost odkazovat se na určité informace jménem a za druhé pro tyto informace vyhradí místo v paměti.

Ještě než proměnnou začnete používat, musíte ji vytvořit, *deklarovat*. Jedním příkazem se dá deklarovat i několik proměnných najednou. Adresa proměnné se dá zjistit operátorem adresy `&`, velikost pomocí `sizeof`.

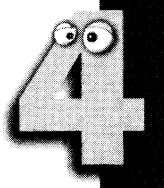
Úkolem proměnné je ukládat informace. Po vytvoření proměnné tedy logicky následuje nastavení její hodnoty, *přiřazení*. Hodnota může být proměnné přiřazena už ve zdrojovém kódu programu, ale může se také zjistit až za běhu programu – například ji uživatel může zadat z klávesnice. Ve zdrojovém kódu se hodnoty proměnných nastavují pomocí operátoru přiřazení `=`. Hodnotu od uživatele můžete za běhu programu převzít prostřednictvím objektu `cin`.

V následující kapitole se naučíte s proměnnými provádět výpočty.

Test

1. Co se stane při deklaraci proměnné?
2. Můžete s proměnnou pracovat, ještě než ji deklarujete? (Za předpokladu, že ji deklarujete později.)
3. Dá se v rámci jednoho příkazu deklarovat víc proměnných najednou?
4. Co v souvislosti s proměnnými znamená výraz *jmenná konvence*?
5. Jaký je rozdíl mezi operátorem adresy `&` a operátorem `sizeof`?
6. Co je inicializace?
7. Co je přetečení?
8. Co se stane, když proměnné celočíselného typu zkusíte operátorem `=` přiřadit řetězec?
9. Probíhá přiřazení hodnot čtením z objektu `cin` během překladu, nebo za běhu programu?
10. Můžete z objektu `cin` v rámci jednoho příkazu načíst hodnotu několika proměnných různého datového typu?

Aritmetické operátory



Když jsem chodil na základní školu (což bylo podle mých dětí někdy v dobách, kdy Zemi ještě ovládali dinosauři), musel jsem všechny výpočty dělat v duchu nebo na papíře. Žádné kalkulačky ještě nebyly, jen logaritmická pravítka. (Pozor! Jakmile byť jen připustíte, že vám výraz logaritmické pravítko něco říká, už budete za starého páprdu.)

Když začaly do školy chodit mé děti a nechal jsem je nad domácím úkolem něco spočítat, sáhly po kalkulačce. A když jsem jim řekl, aby to zkusily v duchu nebo na papíře, dostalo se mi pohledu někde na půli mezi úžasem a soucitem: „Takhle už to dneska nikdo nedělá!"

Možná mají pravdu. Když píšu program, ani já už to dneska takhle nedělám. Spolehnu se na tu nejrychlejší a nejpřesnější kalkulačku, kterou mám: na svůj počítač. Aritmetické výpočty se v programech objevují zcela běžně. Kromě toho, že počítače zvládnou uložit ohromné množství informací, umí také počítat mnohem rychleji a přesněji než my. Aritmetickým operátorům, které vám obrovský výpočetní výkon počítač zpřístupní, se budeme věnovat právě v této kapitole.

Aritmetické operátory

Operátor je symbol, který zastupuje nějakou akci. My už jsme se s operátory setkali v prvních kapitolách, v té předchozí to například byl operátor přiřazení (=). C++ obsahuje řadu aritmetických operátorů, konkrétně operátory pro sčítání, odčítání, násobení a dělení (viz tabulku 4.1).

Tabulka 4.1: Aritmetické operátory

Operátor	Význam	Příklad	Výsledek
+	sčítání	5+2	7
-	odčítání	5-2	3
*	násobení	5*2	10
/	dělení	5/2	2
%	zbytek po dělení	5%2	1

Operátor %, takzvané *modulo*, pro vás možná bude novinka. Počítá zbytek po dělení a podrobněji se mu budeme věnovat později v této kapitole.

Aritmetické operátory jsou většinou *binární*, což znamená, že vyžadují dva parametry neboli *operandy*. Ve výrazu 5+2 jsou pětka a dvojka operandy a znak + je operátor.

Poznámka: Ne všechny aritmetické operátory jsou binární. Například minus ve výrazu -3 je unární operátor, protože vyžaduje jen jeden parametr (v našem případě trojku). Existují i operátory ternární, které vyžadují operandy tři.

Aritmetické operátory pracují s kladnými i zápornými čísly a (s výjimkou modula) s čísly celými i desetinnými. Operátor sčítání funguje kromě čísel i na řetězcích. Následující text vám každý z operátorů ukáže na skutečném programu pro evidenci studentů. Program vychází z mých praktických zkušeností získaných během výuky informatiky na Los Angeles Valley College.

Operátor sčítání

Na škole, kde učím informatiku, se většina studentů do přednášek registruje ještě před začátkem semestru. Někteří si ale přednášku přidají až v průběhu semestru. Následující program obsahuje dvě celočíselné proměnné, `registrovano` a `pridano`. Do proměnné `registrovano` uloží hodnotu, kterou uživatel zadá jako počet předregistrovaných studentů. Do proměnné `pridano` uloží počet studentů, kteří si přednášku zapsali v průběhu semestru, a pak obě proměnné pomocí operátoru + sečte. Součet uloží do proměnné `registrovano`, která bude na konci programu obsahovat celkový počet studentů zapsaných na danou přednášku – těch, co se registrovali předem, i těch přidaných dodatečně. Úplně na závěr program součet vypíše.

```
#include <iostream>
using namespace std;
int main(void)
{
    int registrovano, pridano;
    cout << "Zadejte pocet predregistrovanych studentu: ";
    cin >> registrovano;
    cout << "Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: ";
    cin >> pridano;
    registrovano = registrovano + pridano;
    cout << "Celkovy pocet studentu: " << registrovano << "\n";
    return 0;
}
```

Vstup a výstup programu vypadají například následovně:

```
Zadejte pocet predregistrovanych studentu: 30
Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: 3
Celkovy pocet studentu: 33
```

Spojení s operátorem přiřazení

Začínajícím programátorům někdy dělají potíže výrazy typu `registrovano = registrovano + pridano`, protože matematicky vzato se proměnná jen zřídka rovná součtu sebe sama s jiným číslem. My jsme ale v C++, nikoliv v matematice. Operátor = zde neznamená rovnost, ale přiřazení.

Výraz `registrovano = registrovano + pridano` se dá v C++ vyjádřit stručněji:

```
registrovano += pridano;
```

Z pohledu překladače jsou obě varianty stejné, většina programátorů dává přednost zápisu `registrovano += pridano`. Pro někoho je elegantnější, pro někoho čitelnější a někdo je rád, že si ušetří psaní.

Tato zkrácená varianta není jen doménou operátoru sčítání. Jak je patrné z tabulky 4.2, dá se použít i s ostatními aritmetickými operátory. (Deklaraci celočíselné proměnné `a` si domyslete.)

Tabulka 4.2: Spojení aritmetických operátorů s přiřazením

Příkaz	Zkrácená podoba
`a = a+2;`	`a += 2;`
`a = a-2;`	`a -= 2;`
`a = a*2;`	`a *= 2;`
`a = a/2;`	`a /= 2;`
`a = a%2;`	`a %= 2;`

Priorita aritmetických operátorů a přiřazení

Výraz `registrovano = registrovano + pridano` obsahuje dva operátory, přiřazení a sčítání. Sčítání má jakožto aritmetický operátor *vyšší prioritu* než přiřazení – dostane při vyhodnocování výrazu přednost. Už intuitivně vzato to dává větší smysl než opačná možnost. Jak ale vysvětlím později v této kapitole, priorita hraje roli i mezi různými aritmetickými operátory. Tam už správné pořadí tak zjevné není.

Přetečení a podtečení

Při sčítání může dojít k přetečení a podtečení. Datový typ `int` má na mém operačním systému a překladači rozsah od –2 147 483 648 po 2 147 483 647. Takhle by dopadlo, kdyby má přednáška začala registrací 2 147 483 647 studentů a následně se v semestru přidal jeden navíc:

```
Zadejte pocet predregistrovanych studentu: 2147483647
Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: 1
Celkovy pocet studentu: -2147483648
```

Záporný vstup u tohoto programu nedává smysl, protože záporný počet studentů se na přednášku nepřihlásí. Jiné programy ale záporná čísla používají, například pro teplotu pod nulou. Takhle dopadne, když pomocí záporných čísel způsobíme podtečení:

```
Zadejte pocet predregistrovanych studentu: -2147483648
Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: -1
Celkovy pocet studentu: 2147483647
```

Sčítání řetězců

Při zmínce o sčítání se nám sice automaticky vybaví čísla, ale operátor sčítání se dá použít i na řetězce. Následující program vypíše `Jmenujete se JeffKent`:

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    string jmeno = "Jeff";
    string prijmeni = "Kent";
    cout << "Jmenujete se " << jmeno + prijmeni << "\n";
    return 0;
}
```

Sečíst dva řetězce tedy znamená připojit druhý na konec prvního. Sčítat můžete jen čísla s čísly a řetězce s řetězci. Pokud se pokusíte sečíst číslo s řetězcem, překladač ohlásí chybu. Operátor sčítání umí sčítat čísla s čísly a řetězce s řetězci; číslo k řetězci přičíst nedokáže.

Operátor odčítání

Na naší škole studenti přednášek nejen přibývají, ale i ubývají. Některé studenty vyškrtnu, protože na přednášky vůbec nezačnou chodit; jiní si přednášku vyzkouší a pak ji zruší.

Následující program vychází z předchozí ukázky sčítání. Přidává celočíselnou proměnnou `zruseno` pro počet studentů, kteří si přednášku zrušili nebo vůbec nezačali chodit. Hodnotu proměnné `zruseno` zadá uživatel, program ji odečte od proměnné `registrovano`.

```
#include <iostream>
using namespace std;
int main(void)
{
    int registrovano, pridano, zruseno;
    cout << "Zadejte pocet predregistrovanych studentu: ";
    cin >> registrovano;
    cout << "Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: ";
    cin >> pridano;
    registrovano += pridano;
    cout << "Zadejte pocet studentu, kteri si prednasku zrusili: ";
    cin >> zruseno;
    registrovano -= zruseno;
    cout << "Celkovy pocet studentu: " << registrovano << "\n";
    return 0;
}
```

Vstup a výstup programu může vypadat například následovně:

```
Zadejte pocet predregistrovanych studentu: 30
Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: 3
Zadejte pocet studentu, kteri si prednasku zrusili: 5
Celkovy pocet studentu: 28
```

Ve výrazu `registrovano -= zruseno` jsme použili spojení aritmetického operátoru `-` s operátorem přiřazení, abychom nemuseli psát delší `registrovano = registrovano - zruseno`. Jak už bylo popsáno u operátoru sčítání, obě varianty dělají v praxi totéž.

Při odčítání může stejně jako u sčítání dojít k podtečení a přetečení. Na rozdíl od sčítání se ale odčítání nedá použít na řetězce.

Operátor násobení

Když se vrátím ke svému školnímu příkladu, každý student musí za přednášku zaplatit školné 72 dolarů. Školné platí i studenti, kteří na registrované přednášky nechodí nebo si přednášku zruší. Následující program opět rozšiřuje naši předchozí verzi, do které přidává nový příkaz:

```
cout << "Skolne celkem: " << (registrovano + zruseno) * 72 << " dolaru.\n";
```

Celý program vypadá takto:

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

```
int main(void)
{
    int registrovano, pridano, zruseno;
    cout << "Zadejte pocet predregistrovanych studentu: ";
    cin >> registrovano;
    cout << "Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: ";
    cin >> pridano;
    registrovano += pridano;
    cout << "Zadejte pocet studentu, kteri si prednasku zrusili: ";
    cin >> zruseno;
    registrovano -= zruseno;
    cout << "Celkovy pocet studentu: " << registrovano << "\n";
    cout << "Skolne celkem: " << (registrovano + zruseno) * 72 << " dolaru.\n";
    return 0;
}
```

Vstup a výstup programu by mohl vypadat takto:

```
Zadejte pocet predregistrovanych studentu: 30
Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: 3
Zadejte pocet studentu, kteri si prednasku zrusili: 5
Celkovy pocet studentu: 28
Skolne celkem: 2376 dolaru.
```

Proměnné `registrovano` a `zruseno` se sčítají, protože školné musí zaplatit i studenti, kteří si přednášku nakonec zrušili.

Přetečení a podtečení fungují u násobení stejně jako u sčítání a odčítání. Na rozdíl od sčítání ale násobení nepracuje s řetězci (podobně jako odčítání).

Priorita aritmetických operátorů

Příkaz, který jsme do programu právě přidali, obsahuje dva aritmetické operátory: sčítání a násobení. Hodně záleží na pořadí, ve kterém se tyto operátory provedou. Pokud se první provede sčítání (28+5=33) a teprve výsledek tohoto výpočtu se vynásobí 72, dostaneme číslo 2376. Kdybychom ale první provedli 5×72=360 a výsledek přičetli k číslu 28, dostali bychom číslo 388.

Pořadí výpočtů se řídí takzvanou *prioritou operátorů*. Už jsme na ni v této kapitole jednou narazili – zajímalo nás, jestli se jako první provede operátor přiřazení, nebo operátor sčítání. Tentokrát nás bude zajímat vzájemná priorita jednotlivých aritmetických operátorů, viz tabulku 4.3.

Tabulka 4.3: Priorita aritmetických operátorů

Priorita	Operátory
nejvyšší	- (unární minus)
střední	* / %
nízká	+ -

Operátory, které jsou v tabulce 4.3 na stejném řádku, mají prioritu stejnou. Například násobení má stejnou prioritu jako dělení a podobně sčítání má stejnou prioritu jako odčítání.

V tabulce 4.4 je vidět, jak se priorita projeví na několika aritmetických výrazech. Operátorem dělení jsme se ještě nezabývali, ale v tabulce 4.4 se chová přesně jako v matematice.

Tabulka 4.4: Priorita operátorů v akci

Výraz	Výsledek
2+3×4	14 (nikoliv 20)
8/2–1	3 (nikoliv 8)

Když mají operátory stejnou prioritu, pořadí se určí podle jejich *asociativity*, viz tabulku 4.5.

Tabulka 4.5: Asociativita aritmetických operátorů

Operátor	Asociativita
- (unární minus)	pravá
* / %	levá
+ -	levá

Vezměte si například výraz 8/2×4. Bez závorek by nemuselo být úplně jasné, jak by se měl vyhodnocovat – jako (8/2)×4, nebo jako 8/(2×4)? Priorita operátorů nám tu nepomůže, protože obě operace mají prioritu shodnou. Pomůže nám asociativita. Operátory násobení a dělení asociují *zleva*, takže prostřední operand (v našem případě dvojka) patří *levému* z nich a výraz bude vyhodnocen jako (8/2)×4=16.

Běžně se stává, že potřebujete operace vyhodnotit v jiném pořadí, než v jakém by se vyhodnocovaly podle své asociativity a priority. Například v našem ukázkovém programu by se podle standardních pravidel nejprve vynásobila proměnná `zruseno` číslem 72 a výsledek by se přičetl k proměnné `registrovano`.

Pořadí vyhodnocování výrazů se dá změnit pomocí závorek:

```
cout << "Skolne celkem: " << (registrovano + zruseno) * 72 << " dolaru.\n";
```

Výrazy v závorkách se vyhodnotí jako první. Díky tomu se nejprve sečtou proměnné `registrovano` a `zruseno`, a teprve poté se výsledek vynásobí číslem 72.

Operátory dělení

O každou z operací sčítání, odčítání a násobení se stará jeden operátor. Pro dělení existují operátory dva: `/` vrací podíl a `%` zbytek po dělení.

Podíl a zbytek po dělení jsou – podobně jako dělenec a dělitel – výrazy, které jsem se naučil kdysi na základní škole a následně řadu let vůbec nepoužíval. Pokud si touto terminologií nejste jisti (ani já jsem nebyl), pomůže vám příklad. Když dělíme 7/2, sedmička je dělenec, dvojka dělitel, podíl je 3 a zbytek po dělení 1.

Podíl

Operátor `/` vrací podíl dvou čísel. Výsledek závisí na tom, jestli je alespoň jeden z operandů desetinné číslo. Například 10/4 je 2,5, ale v C++ by se odpovídající výraz vyhodnotil jako 2. Pokud jsou totiž oba operandy celá čísla, operátor `/` provede dělení celočíselné. Platí to i v případech, kdy výsledek ukládáme do proměnné typu `float`; výstupem následujícího programu je číslo 2:

```
#include <iostream>
using namespace std;
int main(void)
{
```

```
    int a=10, b=4;
    float vysledek = a/b;
    cout << a << "/" << b << "=" << vysledek << "\n";
    return 0;
}
```

Když ale výraz změníme na `10.0/4`, dostaneme výsledek 2,5. Kdykoliv je totiž alespoň jeden z operandů desetinné číslo (a výraz `10.0` by se určitě vyhodnotil jako desetinné číslo), operátor `/` provede klasické dělení. Když v předchozím programu změníme typ prvního operandu na `float`, výsledek dělení bude 2,5:

```
#include <iostream>
using namespace std;
int main(void)
{
    float a=10, vysledek;
    int b=4;
    cout << a << "/" << b << "=" << vysledek << "\n";
    return 0;
}
```

Vraťme se ale k předchozímu příkladu. Pokud chcete získat desetinný podíl dvou celočíselných proměnných, musíte jednu z těchto proměnných *přetypovat* na `float`. Přetypování nezmění typ proměnné, změní pouze datový typ hodnoty použité pro výpočet. Když chcete proměnnou přetypovat na nějaký datový typ, napíšete název tohoto datového typu před proměnnou a uzavřete název typu nebo proměnnou do závorek. Takto bychom v prvním příkladu mohli přetypováním dostat desetinný podíl obou čísel:

```
#include <iostream>
using namespace std;
int main(void)
{
    int a=10, b=4;
    float vysledek = (float) a/b;
    cout << a << "/" << b << "=" << vysledek << "\n";
    return 0;
}
```

Stejně dobře by fungoval i libovolný z následujících výrazů:

```
float vysledek = float(a) / b;
float vysledek = a / (float) b;
float vysledek = a / float(b);
```

V některých programech vám ale bude vyhovovat původní celočíselné dělení, viz například automat na drobné na konci této kapitoly.

Zkusme teď dělení využít v příkladu s registrací studentů. V rámci poslední změny jsme vypočetli školné za všechny studenty, včetně těch, kteří už na přednášky nechodí. Teď program upravíme tak, aby spočítal a vypsal průměrné školné na každého doposud zapsaného studenta. Zavedeme celočíselnou proměnnou `skolne`, do které se uloží celkové vybrané školné spočítané jako `(registrovano + zruseno) * 72`. Průměrné školné na zapsaného studenta se pak vypočítá následujícím příkazem:

```
cout << "Prumerne skolne na zapsaneho studenta: "
     << (float) skolne / registrovano << " dolaru.\n";
```

Celý kód programu teď vypadá takto:

```
#include <iostream>
using namespace std;
int main(void)
{
    int registrovano, pridano, zruseno;
    cout << "Zadejte pocet predregistrovanych studentu: ";
    cin >> registrovano;
    cout << "Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: ";
    cin >> pridano;
    registrovano += pridano;
    cout << "Zadejte pocet studentu, kteri si prednasku zrusili: ";
    cin >> zruseno;
    registrovano -= zruseno;
    cout << "Celkovy pocet studentu: " << registrovano << "\n";
    skolne = (registrovano + zruseno) * 72;
    cout << "Skolne celkem: " << skolne << " dolaru.\n";
    cout << "Prumerne skolne na zapsaneho studenta: "
         << (float) skolne / registrovano << " dolaru.\n";
    return 0;
}
```

Vstup a výstup programu by mohl vypadat například takto:

```
Zadejte pocet predregistrovanych studentu: 30
Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: 3
Zadejte pocet studentu, kteri si prednasku zrusili: 5
Celkovy pocet studentu: 28
Skolne celkem: 2376 dolaru.
Prumerne skolne na zapsaneho studenta: 84.8571 dolaru.
```

Přetypování ve výpočtu průměru je nutné, bez něj by průměrné školné vyšlo na 84 dolarů.

Modulo

Operátor `%` neboli modulo vrací zbytek po celočíselném dělení. Například `7%2` není podíl 3, ale zbytek 1. Operátor `%` funguje pouze na celých číslech, na desetinných číslech je jeho výsledek nedefinovaný. Většinou překladač nahlásí chybu, ale záleží na překladači. Praktický příklad použití `%` si ukážeme za okamžik.

Upozornění: Ani jeden z operátorů `/` a `%` se nedá použít pro dělení nulou. Když se o něj pokusíte, dojde k chybě.

Mocniny

Na rozdíl od mnoha jiných jazyků nemá C++ operátor pro umocňování. Místo něj obsahuje vestavěnou funkci `pow` definovanou v souboru `cmath`. Funkce `pow` má dva parametry. Prvním z nich je základ, druhým exponent. Například výraz `pow(2, 3)` umocní dvě na třetí, výsledkem je osm. Hodnota výrazu `pow(2, 3)` je sice celé číslo, ale typu `double`. Umocňovat můžete i desetinná čísla, výsledkem je opět desetinné číslo.

Funkce `pow` se hodí při řešení matematických problémů. Obsah kruhu se dá vypočítat podle vzorce πr^2, kde r je poloměr kruhu. Povrch kruhu o daném poloměru typu `double` tedy můžeme spočítat takto:

```
double povrch = 3.14159 * pow(polomer, 2);
```

Následující program spočítá povrch kruhu o poloměru zadaném uživatelem:

```
#include <iostream>
# include <cmath>
using namespace std;
int main(void)
{
    double polomer, povrch;
    cout << "Zadejte polomer kruhu: ";
    cin >> polomer;
    povrch = 3.14159 * pow(polomer, 2);
    cout << "Povrch kruhu je " << povrch << "\n";
    return 0;
}
```

Vstup a výstup programu vypadá takto:

```
Zadejte polomer kruhu: 6
Povrch kruhu je 113.097
```

Projekt: Automat na drobné

Když moje matka potřebovala zabavit otrávená nebo rozparáděná vnoučata, neštítila se ani takových triků, jakým byl automat na drobné. Děti nakrmily automat stovkami centů a užasle sledovaly, jak je automat třídí na větší částky, které už se dají odnést do banky a vyměnit za čtvrťáky, dolary a větší kořist. Děti se naštěstí dají zabavit snadno. I když v tomhle případě to nebyla úplně férová hra – asi byste uhádli, komu nakonec zůstaly čtvrťáky.

Popis programu

Náš program se uživatele zeptá na počet centů (řekněme, že uživatel vždy zadá kladné celé číslo). Pak vypíše počet dolarů, čtvrťáků, desetníků, pětníků a centů, na které se zadaný počet centů dá směnit:

```
Zadejte pocet centu: 387
Dolaru: 3
Ctvrtaku: 3
Desetniku: 1
Petniku: 0
Centu: 2
```

Nejdřív se podíváme na zdrojový kód, pak si ho vysvětlíme. Zkuste si zdrojový kód napsat předem sami. Pokud to zvládnete, dobře pro vás. Pokud ne, nevadí – i tak vám následující zdrojový kód a jeho popis řeknou víc, když program zkusíte napsat sami předem.

Napovím vám první tři řádky funkce `main` (kdo nechcete, nedívejte se):

```
int celkem, dolary, ctvrtaky, desetniky, petniky, zbytek;
cout << "Zadejte pocet centu: ";
cin >> celkem;
```

Do proměnné `celkem` se uloží celkový počet centů zadaný uživatelem. Do proměnné `dolary` přijde počet celých dolarů v zadaném počtu centů – například do 387 centů z předchozího příkladu se dolary vejdou tři. Do proměnných `ctvrtaky`, `desetniky` a `petniky` přijde počet celých

čtvrťáků, desetníků a pětníků; podle našeho předchozího příkladu tedy hodnoty tři, jedna a nula. Do proměnné `zbytek` se nakonec uloží počet centů, které zbudou (v našem předchozím příkladu dvojka). V průběhu výpočtu proměnná `zbytek` poslouží ještě k uložení mezivýsledků. Program by se samozřejmě dal napsat i s větším nebo menším počtem proměnných.

Zdrojový kód

Program se dá napsat více způsoby, takhle jsem ho napsal já:

```
#include <iostream>
using namespace std;
int main(void)
{
    int celkem, dolary, ctvrtaky, desetniky, petniky, zbytek;
    cout << "Zadejte pocet centu: ";
    cin >> celkem;
    dolary = celkem / 100;
    zbytek = celkem % 100;
    ctvrtaky = zbytek / 25;
    zbytek %= 25;
    desetniky = zbytek / 10;
    zbytek %= 10;
    petniky = zbytek / 5;
    zbytek %= 5;
    cout << "Dolaru: "    << dolary    << "\n";
    cout << "Ctvrtaku: "  << ctvrtaky  << "\n";
    cout << "Desetniku: " << desetniky << "\n";
    cout << "Petniku: "   << petniky   << "\n";
    cout << "Centu: "     << zbytek    << "\n";
    return 0;
}
```

Algoritmy

V první kapitole jsme si řekli, že počítačový program je seznam příkazů, které programátor dává počítači. Tyto příkazy jsou v nějakém programovacím jazyce, například C++. Ještě než ale něco přikážete počítači, musíte si příkazy rozmyslet v přirozeném jazyce.

Algoritmus je přesný návod, podle kterého lze krok za krokem vyřešit nějaký problém. Programátor je ten, kdo vymýšlí a implementuje algoritmy. Zápisu algoritmů v programovacím jazyce se říká programování. Tvorba algoritmů se dá naučit. Dobrým základem jsou pro ni veškeré obory, které vyžadují analytické myšlení, například matematika.

Řekněme, že byste dostali nějaký počet centů, dejme tomu 387, a měli je v hlavě směnit na větší mince – pětníky, desetníky, čtvrťáky a dolary. Jak byste to udělali?

Celkem logické je začít od dolarů. Jeden dolar je sto centů. Když vydělíme 387 stem, dostaneme počet dolarů v zadaných centech: tři. Jak to ale dopadne v C++, nevyhodnotí se výraz `387/100` jako 3,87? V předchozí části této kapitoly věnované operátoru dělení jsme si říkali, že pokud dělíme dva celočíselné operandy, dostaneme celočíselný výsledek. Kdyby se nám to nehodilo, mohli bychom jeden z operandů přetypovat na `float`. Ale nám se celočíselný výsledek hodí, a tak nic přetypovávat nebudeme. Celkový počet centů (387) je uložený v celočíselné proměnné `celkem`, takže když tuto proměnnou vydělíme celým číslem `100`, dostaneme celočíselný podíl: 3.

```
dolary = celkem / 100;
```

Když od 387 centů odečtete tři dolary, zůstane vám 87 centů neboli zbytek po dělení `celkem / 100`. Tento zbytek se dá získat operátorem modulo, uložíme ho do celočíselné proměnné `zbytek`:

```
zbytek = celkem % 100;
```

Stejným postupem zjistíte počet čtvrťáků ve zbývajících 87 centech. Jediný rozdíl je v tom, že namísto stovkou dělíte číslem 25 a místo proměnné `celkem` dělíte proměnnou `zbytek`:

```
ctvrtaky = zbytek / 25;
zbytek %= 25;
```

Analogicky se zjistí i počet desetníků a pětníků:

```
desetniky = zbytek / 10;
zbytek %= 10;
petniky = zbytek / 5;
zbytek %= 5;
```

Počet centů, které zůstanou po závěrečném dělení pětkou, už se na žádnou vyšší minci směnit nedá. Z toho plyne, že nemáme co dál dělit a program je hotový. Všimněte si, že nepotřebujeme samostatnou proměnnou pro počet zbývajících centů, protože ten už máme v proměnné `zbytek`. Zbývá jen vypsat počet dolarů, čtvrťáků, desetníků a centů.

Postup, který jsme si právě popsali, je algoritmus – první z mnoha, kterým se v knize budeme věnovat.

Shrnutí

Aritmetické výpočty jsou běžnou součástí počítačových programů. Kromě toho, že počítače dokážou uložit ohromné množství dat, umí ve srovnání s lidmi i mnohem rychleji a přesněji počítat. Tuto výpočetní sílu můžete využít prostřednictvím aritmetických operátorů.

C++ nabízí aritmetické operátory pro sčítání, odčítání, násobení a dělení. Pro dělení existují operátory hned dva: podíl `/` a zbytek po dělení neboli modulo, `%`.

Všechny aritmetické operátory umí pracovat s celými čísly a všechny kromě modula umí pracovat i s čísly desetinnými. Sčítání jako jediné pracuje i s řetězci, sečíst dva řetězce znamená spojit je dohromady.

C++ nemá na rozdíl od jiných jazyků operátor umocňování. Jeho funkci zastupuje vestavěná funkce `pow` definovaná v souboru `cmath`.

V následující kapitole se dozvíte o relačních a logických operátorech a řídicích strukturách, které umožňují měnit průběh výpočtu podle nějaké podmínky.

Test

1. Pro kterou ze čtyř základních aritmetických operací nabízí C++ více než jeden operátor?
2. Který z aritmetických operátorů funguje kromě čísel i na řetězcích?
3. Který z aritmetických operátorů nepracuje s desetinnými čísly?
4. Který z aritmetických operátorů nemůže mít jako druhý operand nulu?
5. Jak jinak byste mohli zapsat výraz `celkem = celkem + 2`?

6. Jaká je hodnota výrazu `2+3*4`?
7. Jaká je hodnota výrazu `8/2*4`?
8. Jaká je hodnota výrazu `10/4`?
9. Jakým operátorem nebo funkcí se v C++ umocňuje?
10. Co je to algoritmus?

Rozhodování: příkazy if a switch

Slavná báseň *Nezvolená cesta* od Roberta Frosta začíná následujícími verši: „Dvě cesty dělily se v žlutém háji / a já bych tak rád dozvěděl se / co každá z nich přede mnou tají." Je vidět, že nutnosti volby se v životě nevyhneme.

Podobně si musí vybírat i programy. Až doposud se všechny programy v rámci jednoduchosti držely od začátku až do konce jedné cesty, ale každý složitější program se musí rozhodovat podle vstupu od uživatele. Například při nakupování na Internetu si mohu do košíku přidat další zboží, přepočítat celkovou částku nebo přejít k pokladně. Když si budu chtít přidat zboží do košíku, program musí udělat něco jiného, než když budu chtít přejít k pokladně.

Program mou volbu srovnává s různými možnostmi. Toto srovnání se provádí pomocí relačních operátorů – existuje například relační operátor pro rovnost, nerovnost, porovnání velikosti a podobně. Zdrojový kód potom musí být strukturovaný tak, aby se v závislosti na uživatelově volbě provedl správný kus kódu. K tomu slouží příkazy `if` a `switch`, kterým se v této kapitole budeme věnovat.

Probereme také vývojové diagramy, kterými se dá tok programu zakreslit graficky. U jednoduchých programů jsou vývojové diagramy celkem zbytečné, ale až přejdeme ke složitějším programům s větším počtem větví, budou se nám diagramy hodit.

Relační operátory

My lidé v jednom kuse něco srovnáváme. Programy jsou na tom stejně – běžně potřebují zjistit, jestli jsou dvě hodnoty stejné, jestli je první z nich větší a podobně. Pokud například program počítá cenu lístků do kina a děti do 12 let mají vstup zdarma, program musí zjistit, jestli je zákazníkův věk větší než 12.

Ke srovnávání hodnot slouží relační operátory. Relační operátory podporované v C++ najdete v tabulce 5.1.

Tabulka 5.1: Relační operátory

Operátor	Význam
`>`	větší než
`<`	menší než
`>=`	větší nebo rovno
`<=`	menší nebo rovno
`==`	rovno
`!=`	nerovno

Relační výrazy

Podobně jako aritmetické operátory z předchozí kapitoly jsou i relační operátory binární, srovnávají dva operandy. Výrazům se dvěma operandy, mezi kterými je relační operátor, se říká *relační výrazy*. Výsledkem relačního výrazu je booleovská hodnota, tedy pravda (`true`) nebo nepravda (`false`). Několik relačních výrazů s různými relačními operátory ukazuje tabulka 5.2.

Tabulka 5.2: Relační výrazy a jejich hodnoty

Relační výraz	Hodnota
`4 == 4`	`true`
`4 < 4`	`false`
`4 <= 4`	`true`
`4 > 4`	`false`
`4 != 4`	`false`
`4 == 5`	`false`
`4 < 5`	`true`
`4 <= 5`	`true`
`4 >= 5`	`false`
`4 != 5`	`true`

V tabulce 5.2 jsou na místech operandů samé literály, tedy neměnné hodnoty. Srovnávat ale samozřejmě můžete i proměnné. Následující program vypíše výsledky srovnání několika proměnných:

```
#include <iostream>
using namespace std;
int main(void)
{
    int a = 4, b = 5;
    cout << a << " >  " << b << ": " << (a >  b) << "\n";
    cout << a << " >= " << b << ": " << (a >= b) << "\n";
    cout << a << " == " << b << ": " << (a == b) << "\n";
    cout << a << " <= " << b << ": " << (a <= b) << "\n";
    cout << a << " <  " << b << ": " << (a >  b) << "\n";
    return 0;
}
```

Výstup programu vypadá takto:

```
4 >  5: 0
4 >= 5: 0
4 == 5: 0
4 <= 5: 1
4 <  5: 0
```

Nula označuje nepravdu (`false`), jednička pravdu (`true`). Jak si asi pamatujete z první kapitoly, staré počítače bývaly složené z drátů a přepínačů. Elektrický proud chodil v drátech podle toho, jestli byl ten který přepínač zapnutý (jedna), nebo vypnutý (nula). Odtud pochází číselné hodnoty pro booleovské pravda a nepravda.

Upozornění: Obvyklá číselná hodnota pravdivé booleovské proměnné je sice jednička, ale v principu by to mohlo být libovolné nenulové číslo. Proto je jistější používat v booleovských výrazech místo jedničky hodnotu true.

Datové typy obou operandů nemusí být úplně stejné. Typ proměnné `b` z předchozího příkladu můžete klidně změnit z `int` na `float`; program se bez problémů přeloží a jeho výstup zůstane stejný. Operandy ale musí být navzájem kompatibilní, což ve zkratce znamená, že pokud je jeden z operandů číselný, musí být číselný i ten druhý (už jsme o tom mluvili ve třetí kapitole). Kdybyste z proměnné `b` udělali `string`, program by se ani nepřeložil.

Priorita relačních operátorů

Relační operátory mají vyšší prioritu než operátor přiřazení a nižší prioritu než operátory aritmetické. Vzájemnou prioritu jednotlivých relačních operátorů shrnuje tabulka 5.3.

Tabulka 5.3: Vzájemná priorita relačních operátorů

Priorita	Operátor
vysoká	>, >=, <, <=
nízká	==, !=

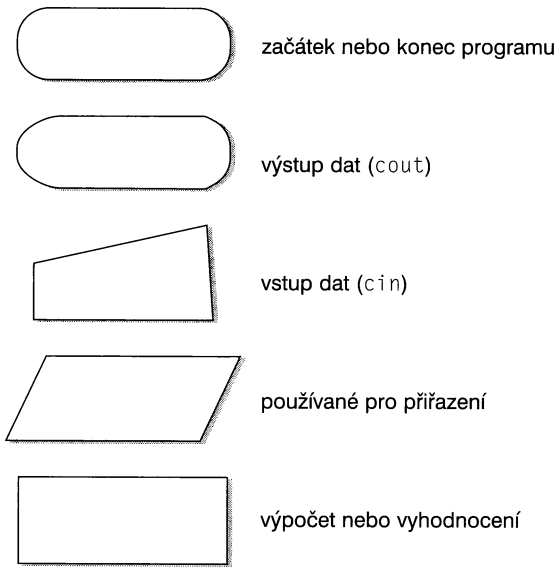
Operátory uvedené na stejném řádku mají prioritu shodnou. Všechny relační operátory asociují zleva, takže například výraz 4 < 5 < 2 se interpretuje jako (4 < 5) < 2. Rozmyslete si, co přesně tento výraz znamená, jestli je pravdivý a proč.

Vývojové diagramy

Program teče jako řeka od začátku do konce. Programátorům pomáhá, když si jeho tok mohou graficky znázornit – je celkem jedno, jestli se snaží lépe vyznat ve svém vlastním programu, nebo v cizím. Jak se říká: jeden obrázek vydá za tisíc slov. Grafické znázornění programu bude ještě užitečnější, až se od jednoduchých programů dostaneme ke složitějším, ve kterých se výpočet v závislosti na relačních výrazech několikanásobně větví.

Pro grafické znázornění toku programu se mezi programátory používají vývojové diagramy. Vývojový diagram je zjednodušená grafická podoba programu, ve které jsou jednotlivé části programu zastoupené dohodnutými grafickými symboly. Podoba symbolů je stanovená americkou standardizační organizací ANSI, která má na svědomí i některé z dalších standardů, na které v knize narazíme. Symboly popisují například začátek a konec programu, uživatelský vstup, zobrazení dat na monitoru a tak dále. Jednotlivé symboly jsou spojené šipkami, které označují možné přechody mezi stavebními bloky programu. Několik běžných symbolů používaných na vývojových diagramech najdete na obrázku 5.1. Další přibudou časem, až je budeme potřebovat.

Můžeme si zkusit vytvořit vývojový diagram ukázkového programu z předchozí kapitoly. Jak si asi vybavujete, program nejdříve načte počet studentů registrovaných na danou přednášku, a to do celočíselné proměnné `registrovano`. Pak od uživatele převezme počet dodatečně registrovaných studentů a uloží ho do celočíselné proměnné `pridano`. Obě proměnné sečte a výsledek uloží do proměnné `registrovano`, která tím pádem obsahuje počet všech studentů registrovaných na danou přednášku. Tento výsledek program vypíše.



Obrázek 5.1: Symboly běžně používané na vývojových diagramech

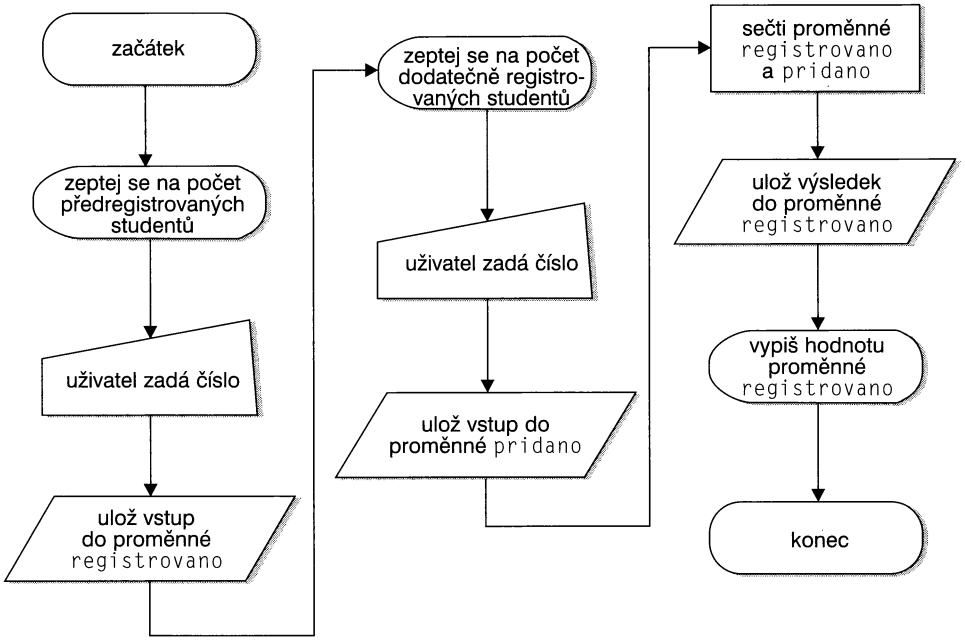
```
#include <iostream>
using namespace std;
int main(void)
{
    int registrovano, pridano, zruseno;
    cout << "Zadejte pocet predregistrovaných studentu: ";
    cin >> registrovano;
    cout << "Zadejte pocet studentu, kteri si prednasku registrovali dodatecne: ";
    cin >> pridano;
    registrovano += pridano;
    cout << "Celkovy pocet studentu: " << registrovano << "\n";
    return 0;
}
```

Vývojový diagram programu najdete na obrázku 5.2. Zatímco tento program je dokonale lineární, většina následujících programů se bude větvit v závislosti na uživatelském vstupu. Na vývojových diagramech se pak dobře vysvětluje, jaký kus kódu se při daném uživatelském vstupu provede.

Příkaz if

Příkaz `if` slouží k tomu, aby se nějaký kus provedl jen tehdy, když platí zadaný relační výraz. Jeho syntaxe vypadá následovně:

```
if (podmínka)
    příkaz;
```



Obrázek 5.2: Vývojový diagram programu pro sčítání předregistrovaných a dodatečně registrovaných studentů

Celý příkaz začíná klíčovým slovem `if`. Za ním následuje podmínka obklopená závorkami, například relační výraz, a na druhé řádce je takzvaný podmíněný příkaz. Podmíněný se mu říká, protože se provede jen při platnosti výše uvedené podmínky. Když podmínka splněná není, podmíněný příkaz se přeskočí. Použití podmíněného příkazu ukazuje následující program. Zkouší, jestli je uživatelem zadané číslo sudé:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo;
    cout << "Zadejte cele cislo: ";
    cin >> cislo;
    if (cislo % 2 == 0)
        cout << "Zadane cislo je sude.\n";
    return 0;
}
```

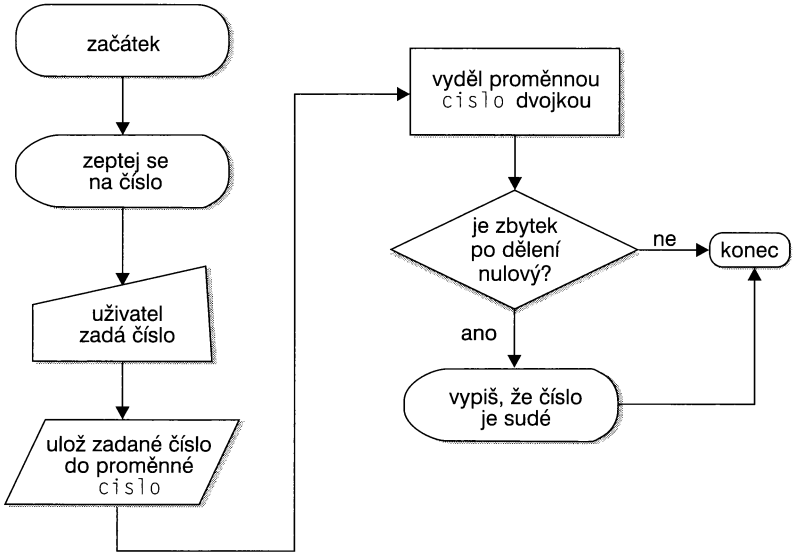
Když uživatel zadá sudé číslo, program vypíše, že zadal sudé číslo:

```
Zadejte cele cislo: 16
Zadane cislo je sude.
```

Pokud je číslo liché, program nevypíše nic:

```
Zadejte cele cislo: 17
```

Na obrázku 5.3 je vývojový diagram tohoto programu. Je na něm k vidění nový symbol, kosočtverec. Ten se používá pro kreslení podmíněných příkazů.



Obrázek 5.3: Vývojový diagram programu pro test lichosti

Pojďme se podívat, jak program funguje. Zároveň s textem můžete sledovat vývojový diagram, abyste se lépe orientovali.

Nejdřív program uživatele požádá, aby zadal číslo. Zadané číslo uloží do celočíselné proměnné `cislo` a v rámci podmíněného příkazu pak vyhodnotí relační výraz `cislo % 2 == 0`. Výraz se skládá ze dvou operátorů, aritmetického operátoru modulo (`%`) a relačního operátoru rovnost (`==`). Aritmetické operátory mají vyšší prioritu než operátory relační, takže jako první se vyhodnotí výraz `cislo % 2` a výsledek této operace se porovná s nulou.

Když sudé číslo vydělíte dvěma, dostanete zbytek rovný nule. V předchozí kapitole jsme si řekli, že zbytek po dělení vrací operátor modulo (`%`). Výraz `cislo % 2` tedy vrací zbytek po dělení dvojkou, který se pak operátorem `==` porovná s nulou. Pokud celý relační výraz platí, tedy pokud je uživatelem zadané číslo sudé, provede se podmíněný příkaz a vypíše zprávu `Zadane cislo je sude.` Pokud relační výraz neplatí, tedy pokud je uživatelem zadané číslo liché, podmíněný příkaz se neprovede.

Odsazování

Bývá dobrým zvykem podmíněný příkaz odsadit:

```
if (cislo % 2 == 0)
    cout << "Zadane cislo je sude.\n";
```

Překladači na odsazování nezáleží, ale člověk si díky němu lépe všimne, že druhý řádek patří k prvnímu.

Běžné chyby

Úvod do programování v C++ učím už řadu let. Za tu dobu jsem si všiml několika běžných chyb, které se při psaní podmíněných příkazů objevují. Na některé z nich přijdete snadno, protože skončí chybou překladu. Jiné jsou mnohem horší – během překladu ani za běhu programu žádnou chybu nezpůsobí, ale vedou k chybným výsledkům.

Středník za relační výraz nepatří

První běžnou chybou je středník za relačním výrazem:

```
if (cislo % 2 == 0); // za uzavírací závorkou nesmí být středník!
    cout << "Zadane cislo je sude.\n";
```

Jelikož překladač většinu mezer a podobných znaků ignoruje, tentýž příkaz se dá zapsat v následující podobě, na které je chyba lépe vidět:

```
if (cislo % 2 == 0)
    ; // za uzavírací závorkou nesmí být středník!
    cout << "Zadane cislo je sude.\n";
```

Překlad proběhne bez chyby. Středník za prázdnou řádkou si překladač vyloží jako prázdný příkaz, který nedělá nic. Prázdné příkazy jsou v C++ povolené a někdy se dokonce hodí, ale v tomhle případě rozhodně ne.

Prvním důsledkem chyby je, že se při splnění podmínky provede prázdný příkaz. To nám nevadí, protože prázdný příkaz nic nedělá. Chyba má ale ještě jeden horší důsledek – vadnou logiku programu. Text `Zadane cislo je sude` se vypíše nezávisle na pravdivosti podmínky, tedy i když bude číslo liché:

```
Zadejte cele cislo: 17
Zadane cislo je sude.
```

Výpis textu nebere ohled na platnost podmínky, protože příkaz pro výpis textu už není součástí příkazu `if`. Jako podmíněný příkaz se bere pouze první příkaz za klíčovým slovem `if` a podmínkou. Tímto prvním příkazem je v našem případě prázdný příkaz, a to kvůli středníku za podmínkou.

Více podmíněných příkazů jen se složenými závorkami

Pokud nevyužijete složené závorky (ke kterým se dostaneme za okamžik), jako podmíněný se bere pouze příkaz bezprostředně za klíčovým slovem `if` a podmínkou. Například v následujícím programu je podmíněný pouze první z výpisů. Druhý už ne, a tak se vypíše nezávisle na pravdivosti podmínky:

```
if (cislo % 2 == 0)
    cout << "Zadane cislo je sude.\n";
    cout << "A zadane cislo neni liche.\n";
```

Poznámka: Podle odsazení pozná podmíněné příkazy programátor, nikoliv překladač. Překladače C++ odsazení ignorují.

Když uživatel zadá liché číslo, například 17, první výpis se neprovede, protože podmínka neplatí. Druhý příkaz se ale provede a program vypíše zprávu `A zadane cislo neni liche`, protože příkaz už nepatří k `if`.

```
Zadejte cele cislo: 17
A zadane cislo neni liche.
```

Pokud chcete, aby podmínka platila pro víc příkazů, musíte tyto příkazy uzavřít do složených závorek:

```
if (cislo % 2 == 0)
{
    cout << "Zadane cislo je sude.\n";
    cout << "A zadane cislo neni liche.\n";
}
```

Teď už se druhý výpis provede pouze tehdy, když podmínka platí. Zapomínání složených závorek u většího počtu podmíněných příkazů je další běžná syntaktická chyba.

Přiřazení místo porovnání

Třetím běžným prohřeškem je záměna operátoru rovnosti (==) s operátorem přiřazení (=):

```
int cislo = 5;
if (cislo = 4) // špatný operátor!
    cout << "Cislo je rovno ctyrem.\n"
```

Výraz v závorce není porovnání proměnné `cislo` s hodnotou čtyři, ale přiřazení čtyřky do proměnné `cislo`. Jak už jsme si říkali, každé přiřazení zároveň vrací hodnotu, aby se daly psát příkazy typu `a = b = c = 5`. (Výraz `c = 5` uloží pětku do proměnné `c` a vrátí hodnotu pět, takže dostaneme příkaz `a = b = 5`, který se dále zjednoduší na `a = 5`.) Podmínka `if (cislo = 4)` tedy ve skutečnosti znamená `if ((cislo = 4) != 0)`, a jelikož přiřazení `cislo = 4` se vyhodnotí jako čtyřka, dostaneme podmínku `4 != 0`, která zjevně platí. Pokud jste ovšem původně chtěli napsat `cislo == 4` a jen jste zapomněli na jedno rovnítko (což se stává celkem běžně), budete se hodně divit.

Poznámka: Některé překladače vás na tuto chybu upozorní. Vyplatí se věnovat pozornost nejenom chybám překladu, ale i varováním vypisovaným překladačem.

Příkaz if/else

Náš program pro testování sudosti čísel má jednu drobnou chybu: Pokud je číslo liché, program nevypíše nic. Máme pouze podmínku, která kontroluje, jestli relační výraz platí. Chybí nám podmínka, která by zkontrolovala opak a případně vypsala, že je číslo liché. Jednodušší řešení nabízí `else` větev podmíněného příkazu, která vypadá takto:

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

Podmíněný příkaz uvedený pod klíčovým slovem `else` („v opačném případě“) se provede pouze tehdy, když podmínka není splněná. Náš program vypadá po rozšíření o `else` větev následovně:

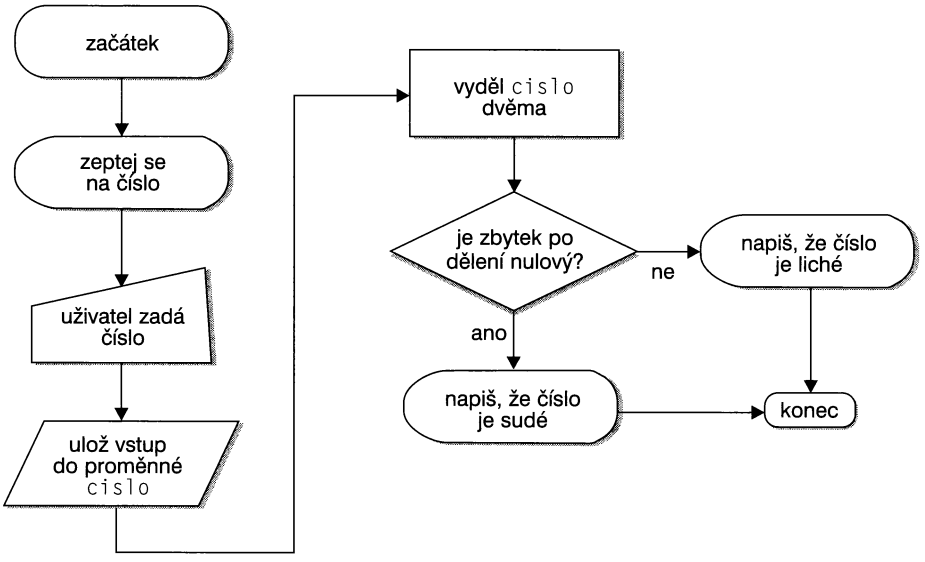
```
#include <iostream>
using namespace std;
int main(void)
{
```

```
    int cislo;
    cout << "Zadejte cele cislo: ";
    cin >> cislo;
    if (cislo % 2 == 0)
        cout << "Zadane cislo je sude.\n";
    else
        cout << "Zadane cislo je liche.\n";
    return 0;
}
```

Přeložte si ho a spusťte. Pokud je zadané číslo sudé, výstupem je opět zpráva `Zadane cislo je sude`. Pokud je číslo liché, program místo mlčení vypíše `Zadane cislo je liche`.

```
Zadejte cele cislo: 17
Zadane cislo je liche.
```

Na obrázku 5.4 je program popsaný vývojovým diagramem.



Obrázek 5.4: Vývojový diagram programu pro testování sudosti čísel

Ternární operátor (podmínkový operátor)

Hlavní podmínka našeho programu by se dala přepsat i pomocí *ternárního operátoru*:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo;
    cout << " Zadejte cele cislo: ";
    cin >> cislo;
    cout << " Zadane cislo je " <<
        (cislo % 2 == 0) ? "sude.\n" : "liche.\n"
    return 0;
}
```

Ternární operátor má následující syntaxi:

```
[relační výraz] ? [když platí] : [když neplatí]
```

V našem případě je relačním výrazem výraz `cislo % 2 == 0`. Pokud je pravdivý, ternární operátor vrátí hodnotu `sude`. V opačném případě vrátí `liche`.

Poznámka: Ternárnímu operátoru se říká ternární, protože vyžaduje tři operandy: Podmínku a dva výrazy. Ternární je samozřejmě jakýkoliv operátor, který potřebuje tři operandy, ale když se v rámci programování mluví o ternárním operátoru, prakticky vždy se myslí právě tento konkrétní.

Běžné chyby

Stejně jako u příkazu `if`, i u příkazu `if/else` se opakovaně vyskytuje několik běžných chyb.

Není else bez if

Příkaz `if` bez větve `else` se používá běžně, ale samotné `else` se bez `if` neobejde. `else` se může objevit výhradně jako součást `if`. Je to logické: jelikož `else` znamená „v opačném případě", bez `if` by nedávalo smysl.

Z toho mimo jiné plyne, že když za podmínku příkazu `if` s `else` větví napíšete středník, dostanete chybu překladu. Jak už jsme si říkali, středník za podmínkou si překladač vyloží jako prázdný příkaz. Další příkazy už tedy nebude brát jako součást `if`, a když narazí na `else`, bude si stěžovat, že chybí příslušné `if`.

```
if (cislo % 2 == 0); // sem středník nepatří
    cout << " Zadane cislo je sude.\n";
else
    cout << " Zadane cislo je liche.\n";
```

Za else už nepatří podmínka

Další běžnou chybou je napsat za klíčové slovo `else` druhou podmínku:

```
if (cislo % 2 == 0)
    cout << "Zadane cislo je sude.\n";
else (cislo % 2 == 1)
    cout << "Zadane cislo je liche.\n";
```

Takový kód se vůbec nepřeloží – překladač si nejspíš postěžuje, že mu před `cout` chybí středník. Chybové hlášení je ovšem poněkud zavádějící, protože řádka s `cout` je naprosto v pořádku. Problém je v nadbytečném relačním výrazu za klíčovým slovem `else`. Opět si stačí připomenout, že `else` znamená „v opačném případě", takže žádná další podmínka není potřeba.

Za else nepatří středník

Další běžnou chybou je psaní středníku za klíčové slovo `else`. Program s touto chybou se přeloží bez problémů, ale bude vracet špatné výsledky. Například následující kód vypíše, že je číslo liché, i když je ve skutečnosti sudé.

```
if (cislo % 2 == 0)
    cout << " Zadane cislo je sude.\n";
else; // sem středník nepatří!
    cout << " Zadane cislo je liche.\n";
```

Takhle to dopadne, když programu zadáme sudé číslo:

```
Zadejte cele cislo: 16
Zadane cislo je sude.
Zadane cislo je liche.
```

Zpráva `Zadane cislo je liche` se vypíše nezávisle na platnosti podmínky, protože příslušný příkaz už není součástí `if`. Problém je úplně stejný jako před chvílí u příkazu `if` – bez složených závorek je podmíněný pouze první příkaz za klíčovým slovem `else`. Tímto příkazem je v našem případě kvůli středníku prázdný příkaz, takže příkaz pro vypsání zprávy `Zadane cislo je liche` se provede vždy.

Více podmíněných příkazů jen se složenými závorkami

Stejně jako u samotného příkazu `if` platí, že pokud chcete v rámci větve `else` provést víc příkazů, musíte je uzavřít do složených závorek. Například příkaz pro výpis řetězce `Tento prikaz je soucasti else vetve` z následujícího kusu kódu se provede nezávisle na podmínce, protože k příkazu `if` už nepatří:

```
if (cislo % 2 == 0)
    cout << " Zadane cislo je sude.\n";
else
    cout << " Zadane cislo je liche.\n";
    cout << " Tento prikaz je soucasti else vetve.\n";
```

Vstup a výstup vypadají například takhle:

```
Zadejte cele cislo: 16
Zadane cislo je sude.
Tento prikaz je soucasti else vetve.
```

Problém se dá vyřešit uzavřením celé větve `else` do složených závorek:

```
if (cislo % 2 == 0)
    cout << " Zadane cislo je sude.\n";
else
{
    cout << " Zadane cislo je liche.\n";
    cout << "Tento prikaz je soucasti else vetve.\n";
}
```

Vícenásobné větvení

Náš modelový program používaný pro ukázku příkazu `if/else` má jen dvě větve. Obě větve se navíc navzájem vylučují, vždy se dá jít jen jednou z nich. Každé celé číslo dává po dělení dvojkou zbytek jedna, nebo nula. Obě varianty zároveň nastat nemohou a žádná třetí neexistuje. Podobných příkladů je hodně – člověk může buď živý, nebo mrtvý; buď muž, nebo žena; buď dospělý, nebo dítě.

Jsou ale i případy, kdy je navzájem výlučných možností víc. Například výsledek zkoušky může být jedno z písmen od A do F. Možností je šest a navzájem se vylučují, z jedné zkoušky nemůžete dostat B a zároveň C.

V úplném příkazu `if` ale máte jen jednu větev `if` a jednu větev `else`, takže další možnosti je potřeba ošetřit jiným výrazem. Tento výraz se jmenuje `else if`.

Příkaz `if / else if / else` se používá, když potřebujete rozlišit tři a více vzájemně výlučných možností. Příkaz má větev `if` a větev `else` – stejně jako příkazy, kterým jsme se doposud zabývali. Kromě nich má ale ještě jednu nebo více větví `else if`.

Poznámka: Bez větve `if` se příkaz neobejde, ale větev `else` povinná není.

Větev `else if` funguje podobně jako obyčejný `if`. Za klíčovými slovy `else if` následuje relační výraz. Pokud je pravdivý, provede se příkaz, který k příslušnému `else if` patří. Každý příkaz `if` může mít jen jednu větev `if` a nanejvýš jednu větev `else`, ale větví `else if` může být víc.

Následující program ukazuje, jak se pomocí vícenásobného větvení dá oznámkovat výsledek zkoušky:

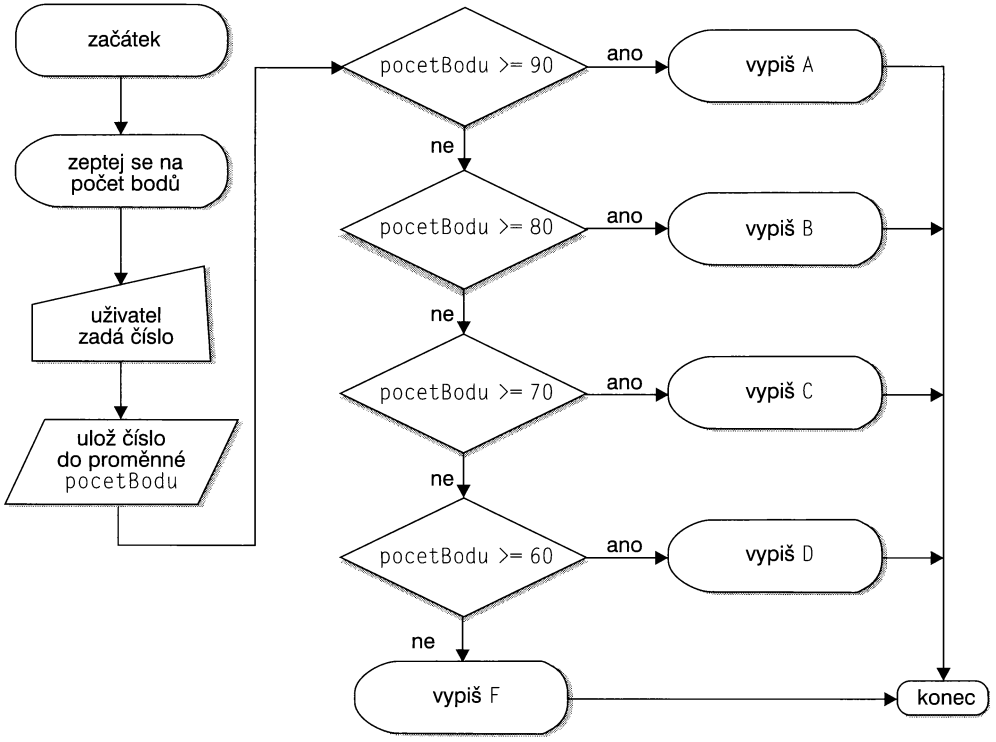
```
#include <iostream>
using namespace std;
int main(void)
{
    int pocetBodu;
    cout << "Zadejte pocet bodu ziskanych pri zkousce: ";
    cin >> pocetBodu;
    cout << "Vase znamka: ";
    if (pocetBodu >= 90)
        cout << "A";
    else if (pocetBodu >= 80)
        cout << "B";
    else if (pocetBodu >= 70)
        cout << "C";
    else if (pocetBodu >= 60)
        cout << "D";
    else
        cout << "F";
    cout << "\n";
    return 0;
}
```

Takhle vypadá několik ukázkových konverzací s programem; čára slouží jen k oddělení jednotlivých pokusů:

```
Zadejte pocet bodu ziskanych pri zkousce: 77
Vase znamka: C
----
Zadejte pocet bodu ziskanych pri zkousce: 91
Vase znamka: A
----
Zadejte pocet bodu ziskanych pri zkousce: 55
Vase znamka: F
```

Na obrázku 5.5 je program zapsaný vývojovým diagramem.

Pokud jste dosáhli 90 nebo více bodů, provede se podmíněný příkaz pod větví `if` a program vypíše, že jste dostali A. Pravdivé jsou i relační výrazy všech následujících větví: Pokud máte přes 90 bodů, máte i přes 80, 70 a 60 bodů. Jakmile se ale jedna z větví provede, další části celého příkazu už se přeskočí.



Obrázek 5.5: Vývojový diagram programu pro známkování zkoušky

Běžné chyby

Na větev `else if` se vztahují všechny chyby, o kterých jsme mluvili u větve `if`. Nepište středník za relační výrazy, a pokud v rámci jedné větve potřebujete provést víc příkazů, nezapomeňte je uzavřít do složených závorek.

Stejně jako nemůžete mít samotný příkaz `else` bez `if`, nemůžete mít ani `else if` bez `if`. Můžete mít ale `if` a několik větví `else if` bez záverečného `else`. Nevýhodou je, že bez větve `else` nebudete mít ošetřený případ, kdy žádný z relačních výrazů neplatí.

Příkaz switch

Příkaz `switch` je podobný příkazu `if` s několika větvemi. Porovná hodnotu celočíselného výrazu s několika možnostmi a podle toho se rozhodne, který kus kódu se má provést.

Následující program ukazuje `switch` v akci. Zadáte mu získanou známku a on podle ní vypíše, kolik jste museli mít bodů:

```
#include <iostream>
using namespace std;
int main(void)
{
```

```
    char znamka;
    cout << "Zadejte znamku: ";
    cin >> znamka;
    switch (znamka)
    {
        case 'A':
            cout << "Pocet bodu: 90-100\n";
            break;
        case 'B':
            cout << "Pocet bodu: 80-89\n";
            break;
        case 'C':
            cout << "Pocet bodu: 70-79\n";
            break;
        case 'D':
            cout << "Pocet bodu: 60-69\n";
            break;
        default:
            cout << "Pocet bodu: 59 a mene\n";
    }
    return 0;
}
```

Takhle vypadá několik ukázkových konverzací s programem; jednotlivé pokusy jsou opět oddělené čárou:

```
Zadejte znamku: C
Pocet bodu: 70-79
----
Zadejte znamku: A
Pocet bodu: 90-100
----
Zadejte znamku: F
Pocet bodu: 59 a mene
```

Pojďme si program rozebrat.

Příkaz `switch` vyhodnotí zadaný celočíselný výraz, v našem případě proměnnou `znamka`. Ta je sice typu `char`, tedy znak, ale každý znak má svůj celočíselný kód – ASCII kód. Kódování ASCII jsme se podrobněji věnovali ve druhé kapitole; ASCII kódy některých běžně používaných znaků ukazuje tabulka 5.4. Všimněte si, že mezi znaky jsou i desítkové číslice a že ASCII kódy velkých znaků jsou jiné než kódy malých znaků.

Tabulka 5.4: Vybrané ANSI/ASCII kódy

Znak	Kód
0	48
9	57
A	65
Z	90
a	97
z	122

Za každým klíčovým slovem `case` musí následovat konstantní celočíselný výraz a dvojtečka. *Konstantní* znamená, že se hodnota výrazu za běhu programu nesmí měnit, takže proměnná za

`case` být nemůže. V tomto případě jsou za `case` znakové literály `A`, `B` a podobně. Jejich celočíselnou hodnotou je jejich ASCII kód.

Upozornění: Častou chybou je záměna dvojtečky středníkem (který se běžně používá k ukončení příkazů). Když za klíčovým slovem `case` a celočíselným výrazem uvedete místo dvojtečky středník, dostanete chybu překladu.

Klíčové slovo `default` plní stejnou funkci jako `else` u podmínek, a tak už se za ním žádný výraz nepíše. Výraz uvedený za klíčovým slovem `switch` se postupně porovnává s jednotlivými výrazy za `case`, odshora dolů. Pokud se hodnoty obou výrazů rovnají, provedou se příkazy za příslušným `case`. V tomto ohledu se tedy `case` chová jako `if`. Na rozdíl od `if` ale není nutné víc příkazů v rámci jednoho bloku uzavírat do složených závorek.

Rozdíly mezi příkazy switch a if

Příkazy `if` a `switch` se do určité míry podobají, ale jsou mezi nimi i podstatné rozdíly. První rozdíl je v tom, že u příkazu `if` jsou podmínky jednotlivých větví naprosto nezávislé. Dobře to ukazuje následující, byť asi poněkud uhozený příklad:

```
if (jablka == hrusky)
    udělej něco;
else if (trzby >= 5000)
    něco jiného;
```

Naproti tomu u příkazu `switch` se hodnota výrazu uvedeného za `switch` porovnává s hodnotami výrazů za jednotlivými `case`, nic jiného. Další rozdíly mezi oběma příkazy popisuje následující kapitola o logických operátorech. Dva z nich ale můžeme zmínit hned: zpracování číselných rozsahů a *falling-through* neboli „propadání".

Propadání

Jednotlivé větve příkazu `if` jsou od sebe důsledně oddělené. Když se ale začne provádět některá z větví příkazu `switch`, program „propadne" přes všechny následující `case` (jejichž hodnotu už vůbec nekontroluje) až po nejbližší příkaz `break` nebo konec příkazu `switch`. Zkuste si z předchozího programu vymazat všechny příkazy `break`, dostanete následující chování:

```
Zadejte znamku: A
Pocet bodu: 90-100
Pocet bodu: 80-89
Pocet bodu: 70-79
Pocet bodu: 60-69
Pocet bodu: 59 a mene
```

Propadání není vždy na škodu. Podívejte se například na následující variaci předchozího programu. Díky propadání jsme snadno zařídili, aby program správně poznal i známky zadané malými písmeny.

```
#include <iostream>
using namespace std;
int main(void)
{
    char znamka;
    cout << "Zadejte znamku: ";
    cin >> znamka;
    switch (znamka)
    {
```

```
        case 'a':
        case 'A':
            cout << "Pocet bodu: 90-100\n";
            break;
        case 'b':
        case 'B':
            cout << "Pocet bodu: 80-89\n";
            break;
        case 'c':
        case 'C':
            cout << "Pocet bodu: 70-79\n";
            break;
        case 'd':
        case 'D':
            cout << "Pocet bodu: 60-69\n";
            break;
        default:
            cout << "Pocet bodu: 59 a mene\n";
    }
    return 0;
}
```

Další příklad nabízí následující program. Jelikož varianta D jako *de luxe* automaticky zahrnuje kožená sedadla z varianty K, `case D` schválně propadá do `case K`:

```
#include <iostream>
using namespace std;
int main(void)
{
    char volba;
    cout << "Vyberte si vybavu:\n";
    cout << "(S) Standard\n";
    cout << "(K) Kozena sedadla\n";
    cout << "(D) Kozena sedadla + chromovana kola\n";
    cin >> volba;
    cout << "Vybrana vybava:\n";
    switch (volba)
    {
        case 'D':
            cout << "Chromovana kola.\n";
        case 'K':
            cout << "Kozena sedadla.\n";
            break;
        default:
            cout << "(Nic jste nevybrali.)\n";
    }
    return 0;
}
```

Dialog s programem vypadá například takhle:

```
Vyberte si vybavu:
(S) Standard
(K) Kozena sedadla
(D) Kozena sedadla + chromovana kola
D
Vybrana vybava:
```

```
Chromovana kola.
Kozena sedadla.
```

Číselné rozsahy

Další rozdíl mezi příkazy `switch` a `if` se týká zpracování číselných rozsahů. Před chvilkou jsme například pomocí příkazu `if` hodnotili výsledek zkoušky podle počtu získaných bodů. Za 90–100 bylo A, za 80–89 bodů B a tak dále. Celý příkaz `if` vypadal následovně:

```
if (pocetBodu >= 90)
    cout << "A";
else if (pocetBodu >= 80)
    cout << "B";
else if (pocetBodu >= 70)
    cout << "C";
else if (pocetBodu >= 60)
    cout << "D";
else
    cout << "F";
```

Naproti tomu za `case` žádný výraz typu `pocetBodu >= 90` napsat nemůžete, protože zde musí být celočíselná konstanta. Místo této podmínky byste potřebovali `case` pro každý možný počet bodů. Následující kód ukazuje, jak by to vypadalo pro známky A a B, abychom si ušetřili zbytečně dlouhý příklad. Kód pro známky C a D by vypadal analogicky, o F by se postaralo klíčové slovo `default`.

```
switch (pocetBodu)
{
    case 100:
    case  99:
    case  98:
    case  97:
    case  96:
    case  95:
    case  94:
    case  93:
    case  92:
    case  91:
    case  90:
        cout << "Mate A.\n";
        break;
    case  89:
    case  88:
    case  87:
    case  86:
    case  85:
    case  84:
    case  83:
    case  82:
    case  81:
    case  80:
        cout << "Mate B.\n";
        break;
}
```

Tady je vidět, že na zpracování číselných rozsahů se příkaz `switch` na rozdíl od `if` nehodí.

Shrnutí

Počítačové programy se jen zřídka drží jedné jediné cesty, jejich výpočet většinou závisí na uživatelském vstupu. Pro porovnání uživatelského vstupu s různými alternativami slouží relační operátory; samotné větvení kódu se provádí pomocí příkazů `if` (s případnými větvemi `else if` a `else`) a `switch`. V této kapitole jste se seznámili s vývojovými diagramy, které tok programu znázorňují graficky a usnadňují jeho pochopení.

Všechny podmíněné příkazy v této kapitole měly jedno společné: V jednu chvíli jsme vždy zkoumali platnost pouze jedné podmínky. Někdy je ovšem potřeba zjistit, jestli některé podmínky platí současně. Například v Americe smí volit pouze ten, komu už bylo 18 let a má americké občanství. Pokud libovolná z těchto podmínek neplatí, volit nesmí. Podobně do kina dostanete lístek zdarma, pokud jste důchodce (starší 65 let) nebo dítě (do 12 let). Jinými slovy dostanete lístek zdarma, pokud platí alespoň jedna z těchto podmínek. Jak podmínky pomocí logických operátorů kombinovat, se dozvíte v následující kapitole.

Test

1. Kolik operandů má relační výraz?
2. K čemu slouží vývojový diagram?
3. Jaký bývá datový typ výrazu za klíčovým slovem `if`?
4. Kterou větev musí každý příkaz `if/else if/else` obsahovat právě jednou?
5. Kterou větev může příkaz `if/else if/else` obsahovat vícekrát?
6. Kterou větev můžete z příkazu `if/else if/else` vynechat?
7. Jaký je datový typ výrazu, který najdete za klíčovým slovem `switch`?
8. Může za klíčovým slovem `switch` následovat výraz typu `char`?
9. Může za klíčovým slovem `case` v příkazu `switch` následovat proměnná?
10. Jaké klíčové slovo hraje v příkazu `switch` podobnou roli jako `else` v příkazu `if`?

Vnořené podmínky a logické operátory

Předchozí kapitolu jsme otevřeli prvními slovy slavné básně *Nezvolená cesta* od Roberta Frosta: „Dvě cesty dělily se v žlutém háji / a já bych tak rád dozvěděl se / co každá z nich přede mnou tají."

Nechci si hrát na literárního kritika, ale cesty bývají často víc než dvě.

V předchozí kapitole jsme větev kódu vybírali vždy jen podle jednoho booleovského výrazu. Někdy je ale booleovských výrazů potřeba vyhodnotit víc – například volit může v Americe pouze ten, kdo dosáhl věku 18 let *a zároveň* má americké občanství. Obě podmínky musí platit zároveň, jinak volit nemůžete. Jindy stačí, když platí alespoň jedna ze dvou podmínek. Například do kina můžete jít zdarma, pokud jste důchodce (starší 65 let) *nebo* dítě do 12 let. Jinými slovy musí platit alespoň jedna z těchto podmínek.

Dvěma různými způsoby spojování logických podmínek se bude zabývat právě tato kapitola. Nejprve nás budou zajímat vnořené podmínky, pak se podíváme na logické operátory.

Vnořené podmínky

Uvnitř příkazu `if` se může objevit další `if`, takzvaná *vnořená podmínka*. Pomocí vnořených podmínek se dá zjistit, jestli nějaké dvě podmínky platí zároveň, případně jestli platí alespoň jedna z nich.

Současná platnost podmínek

Následující program ukazuje, jak pomocí vnořených podmínek otestovat současnou platnost dvou booleovských výrazů. Pokud uživatel zadá, že je starší 18 let a je americkým občanem, program vypíše, že může volit. Pokud libovolná z podmínek neplatí, program vypíše, že uživatel volit nesmí.

```
#include <iostream>
using namespace std;
int main(void)
{
    int vek;
    char volba;
    bool obcan;
```

```
    cout << "Kolik je vam let: ";
    cin >> vek;
    cout << "Jste americky obcan? (A/N): ";
    cin >> volba;
    obcan = (volba == 'A') ? true : false;
    if (vek >= 18)
        if (obcan == true)
            cout << "Muzete volit.\n";
        else
            cout << "Volit nemuzete.\n";
    else
        cout << "Volit nemuzete.\n";
    return 0;
}
```

Poznámka českého vydavatele: Na tomto místě autor používá tzv. ternárního operátoru. Má stejný význam jako příkazy `if-else`, jen má trochu jiný zápis: pokud je výraz před otazníkem pravdivý, výsledkem je hodnota před dvojtečkou (v tomto případě `true`), v opačném případě – `else` – je výsledkem hodnota za dvojtečkou (v tomto případě false). V tomto příkaze se navíc výsledná hodnota (`true`, nebo `false`) uloží do proměnné `obcan`.

Takhle vypadá několik ukázkových dialogů s programem:

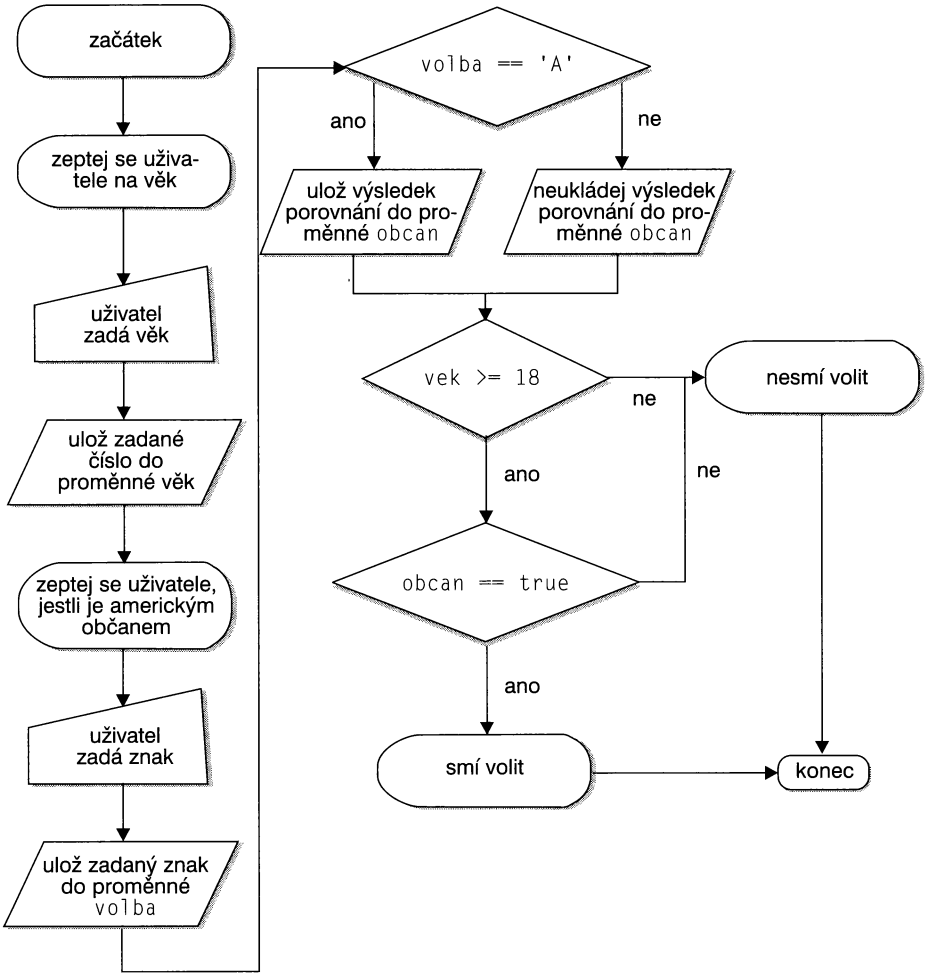
```
Kolik je vam let: 18
Jste americky obcan? (A/N): A
Muzete volit.
----
Kolik je vam let: 18
Jste americky obcan? (A/N): N
Volit nemuzete.
----
Kolik je vam let: 17
Jste americky obcan? (A/N): A
Volit nemuzete.
----
Kolik je vam let: 17
Jste americky obcan? (A/N): N
Volit nemuzete.
```

Na obrázku 6.1 je vývojový diagram programu.

Poznámka: Výraz (obcan == true) se dá zjednodušit na (obcan). Za příkazem if musí být v závorkách booleovský výraz, a jelikož už je proměnná obcan typu bool, je sama o sobě jednoduchým booleovským výrazem i bez jakéhokoliv porovnávání.

Vnořená podmínka vypadá takhle:

```
if (vek >= 18)
    if (obcan == true)
        cout << "Muzete volit.\n";
    else
        cout << "Volit nemuzete.\n";
else
    cout << "Volit nemuzete.\n";
```



Obrázek 6.1: Vývojový diagram programu, který kontroluje způsobilost volit

Příkaz `if`, který kontroluje občanství, je vnořený do podmínky kontrolující věk. Díky tomu se občanství kontroluje pouze v případech, kdy je uživateli alespoň 18 let. Je to logický postup, protože pokud uživateli není alespoň 18 let, občanství už nic nezmění.

Příkazu kontrolujícímu občanství se říká *vnitřní podmínka*, příkazu kontrolujícímu věk se říká *vnější podmínka*. V tomto případě máme vnitřní podmínku v `if` části vnější podmínky, ale vnořenou podmínku může obsahovat i `else` větev vnější podmínky.

Na uvedeném programu je dobře vidět, k čemu jsou vnořené podmínky dobré – pomocí obyčejných podmínek bez vnořování byste tento algoritmus zapisovali jen těžko. Za okamžik se ale dostaneme k logickým operátorům, které nabízí ještě lepší řešení problému.

Platnost alespoň jedné z podmínek

Následující program ukazuje, jak s pomocí vnořených podmínek ověřit, jestli platí alespoň jedna ze dvou daných podmínek. Když uživatel zadá věk nejvýš 12 *nebo* alespoň 65 let, program mu řekne, že má nárok na vstup zdarma. V opačném případě program uživateli řekne, že si lístek musí koupit.

```
#include <iostream>
using namespace std;
int main(void)
{
    int vek;
    cout << "Zadejte svuj vek: ";
    cin >> vek;
    if (vek > 12)
        if (vek >= 65)
            cout << "Mate vstup zdarma.\n";
        else
            cout << "Musite zaplatit.\n";
    else
        cout << "Mate vstup zdarma.\n";
    return 0;
}
```

Konverzace s programem vypadá takhle:

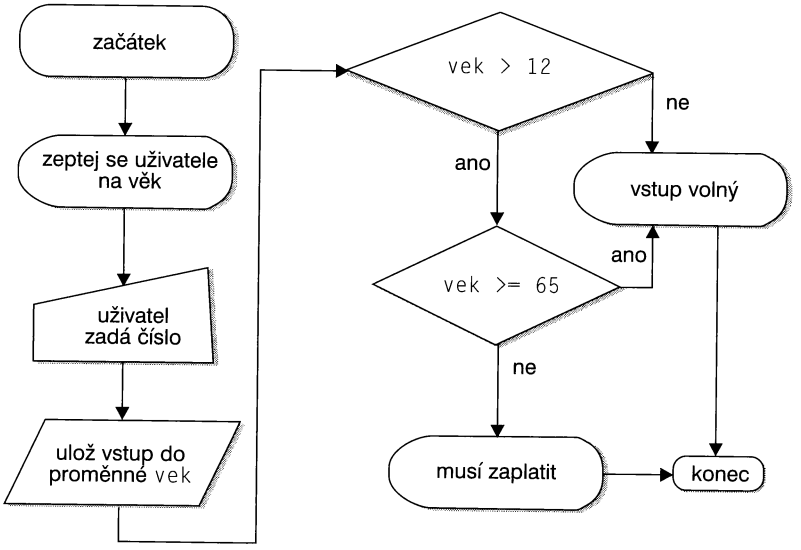
```
Zadejte svuj vek: 12
Mate vstup zdarma.
----
Zadejte svuj vek: 13
Musite zaplatit.
----
Zadejte svuj vek: 65
Mate vstup zdarma.
```

Na obrázku 6.2 je program zakreslený vývojovým diagramem.

Složená podmínka vypadá následovně:

```
if (vek > 12)
    if (vek >= 65)
        cout << "Mate vstup zdarma.\n";
    else
        cout << "Musite zaplatit.\n";
else
    cout << "Mate vstup zdarma.\n";
```

Vnitřní podmínka na věk alespoň 65 let je vnořená do vnější podmínky na věk přes 12 let. Díky tomu se provede jen v případě, že je uživatel starší dvanácti let. To je logický přístup, protože kdyby uživatel byl mladší dvanácti let, můžeme ho rovnou pustit zadarmo (a určitě nebude zároveň starší 65 let).



Obrázek 6.2: Vývojový diagram programu pro kontrolu vstupného do kina

Tentýž algoritmus bychom mohli kromě vnořených podmínek zapsat i následující větvenou podmínkou:

```
if (vek <= 12)
    cout << "Mate vstup zdarma.\n";
else if (age >= 65)
    cout << "Mate vstup zdarma.\n";
else
    cout << "Musite zaplatit.\n";
```

Každá z variant má své výhody a nevýhody. Vnořené podmínky nejsou tak přehledné, ale větvená podmínka zase musí zopakovat `cout` ve větvích `if` a `else if`. V tomto konkrétním programu jde jen o jednu řádku, u složitějších programů byste třeba museli opakovat větší kus kódu. Proto C++ nabízí třetí a nejlepší řešení, a to logické operátory.

Logické operátory

Logické operátory umožňují spojit několik podmínek do jedné. Logické operátory podporované v C++ najdete společně s jejich popisem v tabulce 6.1.

Tabulka 6.1: Logické operátory

Operátor	Název	Popis
&&	a zároveň	Slouží ke spojení dvou booleovských výrazů. Výsledný výraz je pravdivý, pokud jsou pravdivé oba spojované výrazy.
||	nebo	Slouží ke spojení dvou booleovských výrazů. Výsledný výraz je pravdivý, pokud je pravdivý alespoň jeden ze spojovaných výrazů.
!	negace	Obrátí pravdivostní hodnotu zadaného výrazu. Pokud mu zadáte pravdivý výraz, vrátí `false`, a když zadáte nepravdivý výraz, vrátí `true`.

Operátor &&

Operátoru && se říká *a zároveň, logický součin* nebo *konjunkce*. Je to binární operátor, vyžaduje dva booleovské výrazy jako operandy. Logický součin dvou operandů je pravdivý právě tehdy, když jsou pravdivé oba operandy. (Poznámka českého vydavatele: Všimněte si, jak chování operátoru hezky odpovídá názvu součin. Když si logické hodnoty true a false představíte jako jedničku a nulu, nenulový neboli pravdivý vyjde pouze součin dvou jedniček.) Pokud je libovolný z operandů nepravdivý, je nepravdivý i jejich součin. (Z toho samozřejmě plyne, že pokud jsou nepravdivé oba operandy, bude opět nepravdivý i jejich součin.) Všechny možnosti shrnuje tabulka 6.2.

Tabulka 6.2: Operátor &&, logický součin

První operand	Druhý operand	Součin
true	true	true
true	false	false
false	true	false
false	false	false

Praktické použití operátoru && ukazuje následující program. Jde opět o kontrolu, jestli uživatel může volit, tedy jestli je mu alespoň osmnáct let a má americké občanství:

```
#include <iostream>
using namespace std;
int main(void)
{
    int vek;
    char volba;
    bool obcan;
    cout << "Kolik je vam let: ";
    cin >> vek;
    cout << "Jste americky obcan? (A/N): ";
    cin >> volba;
    obcan = (volba == 'A') ? true : false;
    if (vek >= 18 && obcan)
        cout << "Muzete volit.\n";
    else
        cout << "Volit nemuzete.\n";
    return 0;
}
```

Dialog s programem vypadá následovně:

```
Kolik je vam let: 18
Jste americky obcan? (A/N): A
Muzete volit.
----
Kolik je vam let: 18
Jste americky obcan? (A/N): N
Volit nemuzete.
----
Kolik je vam let: 17
Jste americky obcan? (A/N): A
Volit nemuzete.
----
Kolik je vam let: 17
Jste americky obcan? (A/N): N
Volit nemuzete.
```

Logický součin je v programu použitý v následující podmínce:

```
if (vek >= 18 && obcan)
    cout << " Muzete volit.\n";
else
    cout << "Volit nemuzete.\n";
```

Pokud je uživateli alespoň 18 let, vyhodnotí se ještě druhé porovnání a program zkontroluje, jestli je uživatel americkým občanem. Pokud je uživatel mladší osmnácti let, pravá strana `&&` se vůbec nevyhodnocuje a rovnou se provede větev `else`. Pravá část `&&` už totiž v takovém případě nemůže ovlivnit výsledek – pokud je nepravdivý první operand `&&`, výsledný výraz bude nepravdivý nezávisle na pravdivosti druhého a počítač může ušetřit pár zbytečných instrukcí. (Poznámka českého vydavatele: Díky tomuto chování logických operátorů navíc můžete bezpečně psát výrazy typu ((delitel != 0) && (100 % delitel == 0)). Rozmyslete si, proč by důsledné vyhodnocování obou stran logických operátorů v takových případech vadilo.)

Jelikož se druhý operand vyhodnocuje pouze při pravdivosti prvního, vývojový diagram 6.1 nakreslený podle řešení s vnořenými podmínkami se vztahuje i na naši novou verzi.

Operátor ||

Operátoru `||` se říká *nebo*, *disjunkce a*nebo *logický součet*. Stejně jako `&&` je binární, vyžaduje dva operandy booleovského typu a je pravdivý, pokud je alespoň jeden z operandů pravdivý. Nepravdivý je pouze v případě, kdy jsou oba operandy nepravdivé. Z toho tedy plyne, že pokud jsou oba operandy pravdivé, je pravdivý i jejich logický součet (viz tabulku 6.3).

Tabulka 6.3: Logický součet

První operand	Druhý operand	Logický součet
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

Následující program pomocí logického součtu rozhoduje, jestli máte nárok na kino zdarma, tedy jestli je vám nanejvýš 12 nebo alespoň 65.

```
#include <iostream>
using namespace std;
int main(void)
{
    int vek;
    cout << "Zadejte svuj vek: ";
    cin >> vek;
    if (vek <= 12 || vek >= 65)
        cout << "Mate vstup zdarma.\n";
    else
        cout << "Musite zaplatit.\n";
    return 0;
}
```

Takhle vypadá program v akci:

```
Zadejte svuj vek: 12
Mate vstup zdarma.
```

```
----
Zadejte svuj vek: 18
Musite zaplatit.
----
Zadejte svuj vek: 65
Mate vstup zdarma.
```

Logický součet je použitý v následující podmínce:

```
if (vek <= 12 || vek >= 65)
    cout << " Mate vstup zdarma.\n";
else
    cout << " Musite zaplatit.\n";
```

Pokud je uživateli přes 12 let, program ještě provede porovnání na pravé straně, jestli uživateli není alespoň 65. Pokud má ale uživatel nanejvýš 12 let, pravá strana se vůbec nevyhodnocuje a rovnou se přejde na větev `else`. Důvod je stejný jako u operátoru `&&` – když je levá strana pravdivá, výsledek bude pravdivý nezávisle na pravé straně a můžeme si ušetřit pár instrukcí.

A jelikož se podmínka na 65 let vyhodnocuje pouze pro uživatele starší dvanácti let, naše nová verze opět přesně odpovídá vývojovému diagramu 6.2 kreslenému pro variantu s vnořenými podmínkami.

Operátor !

Operátoru `!` se říká negace. (Moje dcery mají s negováním velké zkušenosti z dob, kdy jim bylo kolem šestnácti nebo sedmnácti let, ale my se budeme držet C++.) Negace je unární operátor, vyžaduje jeden operand booleovského typu. Pokud je operand pravdivý, jeho negace je nepravdivá, a pokud je operand nepravdivý, jeho negace je pravdivá (tabulka 6.4).

Tabulka 6.4: Operátor negace

Operand	Negace
`true`	`false`
`false`	`true`

Následující program ukazuje negaci ve spojení s logickým součinem, opět hrajeme o lístky do kina zdarma.

```
#include <iostream>
using namespace std;
int main(void)
{
    int vek;
    cout << "Zadejte svuj vek: ";
    cin >> vek;
    if (! (vek > 12 && vek < 65))
        cout << "Mate vstup zdarma.\n";
    else
        cout << "Musite zaplatit.\n";
    return 0;
}
```

Program je téměř stejný jako předchozí verze s logickým součtem (||), jediný rozdíl je v následující řádce:

```
if (vek <= 12 || vek >= 65)
```

Tu jsme nahradili novým výrazem:

```
if (! (vek > 12 && vek < 65))
```

Poznámka: Tato změna je příkladem DeMorganova zákona pro negaci logického součtu a součinu. V této knize se DeMorganovými zákony zabývat nebudeme, ale kdybyste se k nim někdy v rámci jiné knihy nebo přednášky dostali podrobně, nezapomeňte, kde jste o nich slyšeli poprvé!

Pomocí operátoru negace můžete booleovský výraz přeformulovat do podoby, která vám připadne srozumitelnější. Například v tomto případě by pro vás mohla být srozumitelnější podmínka, že neplatící návštěvník kina „nesmí být mezi 13 a 64 lety", než že „mu musí být nanejvýš 12 nebo alespoň 65 let".

Priorita operátorů

Tabulka 6.5 shrnuje prioritu logických a relačních operátorů, operátory s vyšší prioritou jsou uvedené jako první.

Tabulka 6.5: Priorita logických a relačních operátorů

Operátor	Priorita
!	vysoká
>, >=, <, <=, ==, !=	střední
&&	nízká
||	nejnižší

Priorita negace

Jelikož má negace vyšší prioritu než relační operátory, museli jsme v předchozím programu použít závorky navíc:

```
if (! (vek > 12 && vek < 65))
```

Kdybychom závorky vynechali (viz následující řádek), program by za všech okolností tvrdil, že uživatel musí platit.

```
if (! vek > 12 && vek < 65)
```

Může za to vysoká priorita operátoru `!`. Negace má ještě vyšší prioritu než relační operátory, takže „ukradne" první parametr operátoru `>` a překladač celý výraz pochopí jako `((!vek) > 12 && vek < 65)`. Pokud bude `vek` různý od nuly (což se v booleovském kontextu chápe jako pravda), výraz `!vek` bude nulový (`false`), srovnání `!vek > 12` se vyhodnotí jako nepravda a celý výraz bude nepravdivý. Pokud bude `vek` nulový (`false`), výraz `!vek` bude mít hodnotu jedna (`true`), srovnání `!vek > 12` se opět vyhodnotí jako nepravda a celý výraz dopadne jako v předchozím případě. Proto se nezávisle na věku uživatele provede větev `else`.

Priorita logického součtu a součinu

Operátory logického součtu a součinu mají – na rozdíl od negace – nižší prioritu než relační operátory, proto je většinou nebývá nutné oddělovat pomocí závorek. Například následující dvě podmínky (které pochází z programu ilustrujícího logický součin) jsou ekvivalentní:

```
if (vek >= 18 && obcan == true)
if ((vek >= 18) && (obcan == true))
```

Závorky jsou ovšem potřeba v případech, kdy máte v jednom výrazu logický součet i součin a potřebujete součet (||) provést přes součinem (&&). Operátor && má vyšší prioritu než ||, takže bez závorek by se provedl jako první.

Představte si například, že by se změnila pravidla pro voliče a kromě občanů by mohli volit i lidé s trvalým bydlištěm v místě voleb. Omezení na věk 18 let bychom zachovali, takže celá podmínka by mohla vypadat takto:

```
if (mistni || obcan && vek >= 18)
```

Jenže operátor && má vyšší prioritu než operátor ||, takže překladač by podmínku pochopil takto:

```
if (mistni || (obcan && (vek >= 18)))
```

Tato podmínka by dovolila volit i všem místním mladším osmácti let. Protože pokud bude proměnná `mistni` pravdivá, výsledek operátoru || bude nezávisle na hodnotě výrazu `(obcan && (vek >= 18))` pravdivý. Takový výsledek se nám nelíbí. Abychom chybu opravili, přidáme do podmínky závorky navíc, díky kterým se logický součet || provede jako první.

```
if ((mistni || obcan) && vek >= 18)
```

Logické operátory a příkaz switch

Příkazu `switch` jsme věnovali slušnou část předchozí kapitoly, ale v této kapitole se o něm až doposud podezřele mlčelo. Ještě v minulé kapitole jsme si řekli, že se příkaz `switch` příliš nehodí pro číselné rozsahy – za klíčové slovo `case` může přijít pouze jediná celočíselná konstanta, nikoliv číselný rozsah. Logické operátory se ale v kombinaci s příkazem `switch` použít dají, protože mají jen dvě možné hodnoty: `true` a `false`. Tyto hodnoty jsou konstanty, `true` je vždy `true` a `false` je vždy `false`. Obě jsou hodnoty sice typu `bool`, ale mají odpovídající celočíselné hodnoty `1` a `0`. Proto je můžete uvést za klíčové slovo `case` – stejně jako jsme tam v předchozí kapitole mohli napsat konstanty typu `char`, které mají odpovídající celočíselné kódy z ASCII.

V této kapitole jsme pomocí logického součinu a příkazu `if` kontrolovali, jestli je uživatel starší 18 let a má americké občanství; tedy jestli může volit:

```
if (vek >= 18 && obcan)
    cout << "Muzete volit.\n";
else
    cout << "Volit nemuzete.\n";
```

Odpovídající řešení pomocí příkazu `switch` by vypadalo takto:

```
switch (vek >= 18 && obcan)
{
    case true:
```

```
        cout << " Muzete volit.\n";
        break;
    case false:
        cout << "Volit nemuzete.\n";
}
```

A podobně jsme pomocí podmíněného příkazu a logického součtu kontrolovali, jestli je mladší než 12 nebo starší než 65 let, aby mohl zadarmo do kina:

```
if (vek <= 12 || vek >= 65)
    cout << "Mate vstup zdarma.\n";
else
    cout << "Musite zaplatit.\n";
```

Odpovídající řešení příkazem `switch` vypadá takto:

```
switch (vek <= 12 || vek >= 65)
{
    case true:
        cout << " Mate vstup zdarma.\n";
        break;
    case false:
        cout << "Musite zaplatit.\n";
}
```

Na těchto příkladech je vidět, že se příkaz `switch` při vyhodnocování booleovských výrazů dá použít jako náhrada příkazu `if`. V praxi se ale tato možnost nevyužívá, protože vyhodnocování booleovských výrazů může vést jen ke dvěma možným výsledkům, zatímco `switch` je určený především pro zpracování většího počtu možností.

Shrnutí

V předchozí kapitole jsme se vždy rozhodovali jen na základě jednoduchého booleovského výrazu s jedním relačním operátorem. Řízení toku programu ale běžně vyžaduje vyhodnocení složitějších booleovských výrazů. Pokud například může v Americe volit jen ten, komu už bylo 18 let a zároveň má americké občanství, program musí vyhodnotit dvě podmínky a rozhoduje se podle toho, jestli obě platí *zároveň*. Jiný příklad nabízí volné vstupné do kina: Pokud je volné vstupné omezené na děti do dvanácti a důchodce nad 65 let, program potřebuje vědět, jestli je splněná *alespoň jedna* z těchto podmínek.

V této kapitole jsme se zabývali dvěma způsoby, kterými lze takové složené podmínky vyhodnotit. Prvním z nich jsou vnořené podmínky, tedy jeden příkaz `if` uvnitř druhého, a druhým jsou logické operátory. Operátor `&&` neboli logický součin je pravdivý pouze tehdy, když jsou pravdivé oba jeho operandy. Operátor `||` neboli logický součet je pravdivý, pokud je pravdivý alespoň jeden z jeho dvou operandů. A do třetice operátor `!` čili negace obrátí pravdivost svého operandu, z `true` na `false` a z `false` na `true`.

Na závěr kapitoly jsme si ukázali, jak se při vyhodnocování booleovských výrazů dá příkaz `if` nahradit příkazem `switch`.

Test

1. Dají se pomocí vnořených příkazů `if` nahradit logické operátory `&&` a `||`?
2. Může být příkaz `if` vnořený v `else` větvi jiného příkazu `if`? Nebo je vnořování povolené jen ve větvi `if`?
3. U kterého z logických operátorů musí být oba operandy bezpodmínečně pravdivé, aby byl i výsledek pravdivý?
4. U kterého z logických operátorů musí být oba operandy bezpodmínečně nepravdivé, aby byl i výsledek nepravdivý?
5. Který z logických operátorů převrací pravdivostní hodnotu svého operandu?
6. Pokud je `mistni` booleovská proměnná, znamená `if (mistni)` totéž co `if (mistni == true)`?
7. Který z logických operátorů je unární, nikoliv binární?
8. Který z logických operátorů má vyšší prioritu než relační operátory?
9. Který z operátorů `&&` a `||` má vyšší prioritu?
10. Dají se booleovské konstanty `true` a `false` použít za klíčovým slovem `case` v příkazu `switch`?

Cyklus for

Rodiče dětem často říkají, aby se neopakovaly. Vlastně jim to říkají pořád dokola, a tím názorně dokreslují jiný ze svých příkazů: „O mě se nezajímej a dělej to co ti říkám.“ Tím jsem se chtěl nenuceně dostat k myšlence, že i ve světě počítačů bývá potřeba nějaký kus kódu opakovat. Když například uživatel zadá chybná data, můžete se ho zeptat, jestli to chce zkusit ještě jednou, nebo jestli má program skončit. Pokud to zkusí ještě jednou a znovu zadá špatná data, můžete se ho zeptat, jestli to chce zkusit ještě jednou, nebo jestli má program skončit. A tak to jde dokola, dokud uživatel nezadá správná data nebo program neukončí.

Jedním z nástrojů, kterými lze zařídit opakování kódu, je cyklus `for`. Cyklus je obecně vzato struktura, která opakuje určitý kus kódu, dokud platí zadaná podmínka. U příkladu z předchozího odstavce jsme čekali, dokud uživatel nezadá správná data, a opakovali jsme vstup dat a jejich kontrolu.

V této kapitole se budeme zabývat jedním konkrétním typem cyklu, a to cyklem `for`. Ještě než se k němu ale dostaneme, ukážeme si operátory pro zvýšení a snížení hodnoty proměnné, které se nám budou v cyklech hodit. V následující kapitole se pak podíváme na další dva typy cyklů, cyklus `while` a cyklus `do while`.

Operátory ++ a --

C++ má dva samostatné operátory pro snížení a zvýšení hodnoty proměnné. V následujícím textu se dozvíte, jak se tyto operátory používají. Budou se nám hodit v cyklu `for`, ke kterému se dostaneme vzápětí.

Operátor ++

Následující program obsahuje proměnnou `cislo` inicializovanou na hodnotu 2. Hned za deklarací této proměnné je příkaz `cislo += 1`, které hodnotu proměnné zvýší o jedničku, takže výsledná vypsaná hodnota bude 3:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 2;
    cislo += 1;
    cout << cislo;
    return 0;
}
```

Totéž zvýšení proměnné o jedničku se dá zařídit operátorem ++. Operátor je unární, vyžaduje jeden operand. Většinou jde o celé číslo, například `int`. Takto by náš program vypadal, kdybychom operátor += nahradili operátorem ++:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 2;
    cislo++;
    cout << cislo;
    return 0;
}
```

Tentýž výstup bychom dostali, kdybychom výraz `cislo++` nahradili výrazem `++cislo`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 2;
    ++cislo;
    cout << cislo;
    return 0;
}
```

Když je operátor `++` nalevo od svého operandu, mluví se o *prefixovém* zvýšení hodnoty. Předpona *pre-* naznačuje, že je operátor před operandem. Analogicky se rozeznává varianta *postfixová*, u které je operátor za svým operandem. Obě varianty zvyšují hodnotu proměnné o jedničku, ale obě trochu jinak. U našeho jednoduchého programu vyjdou obě varianty úplně nastejno. Zastavme se ještě na okamžiku u operátoru `--`, pak se k rozdílu mezi prefixovou a postfixovou variantou vrátíme podrobněji.

Operátor --

Následující program opět obsahuje proměnnou `cislo` inicializovanou na hodnotu 2. Příkazem `cislo -= 1` hodnotu proměnné snížíme o jedničku, takže program vypíše hodnotu 1.

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 2;
    cislo -= 1;
    cout << cislo;
    return 0;
}
```

Snížení o jedničku se dá provést také operátorem `--`. Stejně jako v předchozím případě je to operátor unární, takže vyžaduje právě jeden operand (většinou jde o celé číslo, například `int`). Takhle by vypadalo, kdybychom operátorem `--` nahradili operátor `-=`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 2;
    cislo--;
    cout << cislo;
    return 0;
}
```

Podobně jako ++ má operátor -- svou prefixovou variantu, která se chová maličko odlišně, ale v našem jednoduchém programu funguje úplně stejně:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 2;
    --cislo;
    cout << cislo;
    return 0;
}
```

Rozdíl mezi prefixovou a postfixovou variantou

Už dvakrát jsem se zmínil o rozdílu mezi prefixovou a postfixovou variantou operátorů ++ a --, v čem přesně spočívá? Zkuste si přeložit a spustit následující jednoduchý program:

```
#include <iostream>
using namespace std;
int main(void)
{
    int a=1, b=1;
    cout << ++a << "\n";
    cout << b++ << "\n";
    return 0;
}
```

Rozdíl mezi výrazem ++cislo a cislo++ spočívá v tom, že první varianta nejprve hodnotu proměnné zvýší a pak až vrátí její novou hodnotu, zatímco druhá varianta nejprve vrátí aktuální hodnotu proměnné a teprve pak proměnnou zvýší. Nemělo by vás proto překvapit, že výše uvedený program vypíše následující:

```
2
1
```

Totéž v bledě modrém ilustruje následující program:

```
#include <iostream>
using namespace std;
int main(void)
{
    int a=1, b=1;
    cout << (a-- == 0) << "\n"; // vypíše nulu, neplatí
    cout << (--b == 0) << "\n"; // vypíše jedničku, platí
    // tady už jsou obě proměnné nulové
    return 0;
}
```

První porovnání se vyhodnotí jako nepravdivé, protože výraz a-- vrátí původní hodnotu proměnné a teprve pak proměnnou sníží. Druhé porovnání se vyhodnotí jako pravdivé, protože prefixová varianta --b nejprve sníží hodnotu proměnné b o jedničku a pak až vrátí její hodnotu.

Důležité je si uvědomit, že dopad obou variant na obsah proměnné je úplně stejný. Když do proměnné cislo uložíte pětku, po vyhodnocení libovolného z výrazů ++cislo a cislo++ bude proměnná obsahovat šestku. Rozdíl je pouze v hodnotách těchto výrazů: zatímco ++cislo už vrátí šestku, cislo++ vrátí pětku.

Bývá dobrým zvykem se vyhýbat výrazům, ve kterých se proměnná zvyšovaná nebo snižovaná pomocí operátorů ++ a -- vyskytuje více než jednou:

```
int a = 1;
int b = a++ + a++; // tohle není dobrý nápad
```

My se podobným případům vyhneme velkým obloukem, protože nás momentálně zajímá především jednoduché využití operátorů ++ a -- v cyklech.

Cyklus for

Pro výpis čísel od jedničky do desítky byste mohli napsat například následující program:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 1;
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    cout << cislo++ << "\n";
    return 0;
}
```

Mnohem stručnější způsob nabízí cyklus `for`:

```
#include <iostream>
using namespace std;
int main(void)
{
    for (int cislo = 1; cislo <= 10; cislo++)
        cout << cislo << "\n";
    return 0;
}
```

Rozdíl mezi oběma programy by byl ještě výraznější, kdybychom chtěli řekněme všechna čísla od jedničky do sta. Nebudu přepisovat první verzi programu, protože bychom tím zabrali hezkých pár stránek – stačí vědět, že bychom museli přidat 90 dalších výpisů proměnné `cislo`. Tentýž program napsaný s pomocí cyklu `for` by vypadal takto:

```
#include <iostream>
using namespace std;
int main(void)
{
    for (int cislo = 1; cislo <= 100; cislo++)
        cout << cislo << "\n";
    return 0;
}
```

Díky cyklu `for` by skoro stejný kód mohl vypsat čísla od jedné do tisíce nebo třeba deseti tisíc, stačilo by číslo 100 nahradit za 1 000 nebo 10 000.

Cyklus `for` je jen jedním ze tří typů cyklů neboli smyček; zbývajícím dvěma typům (smyčkám `while` a `do while`) se budeme věnovat v následující kapitole. Smyčka neboli cyklus je struktura, která opakuje zadaný kus kódu, dokud je splněná nějaká podmínka. Každému zopakování cyklu se říká *iterace*.

V příkladu s výpisem čísel od jedné do deseti se hodnota proměnné `cislo` vypisovala, dokud byla menší nebo rovno deseti. Iterací cyklu bylo celkem deset – desetkrát se aktuální hodnota proměnné `cislo` vypsala na standardní výstup.

Syntaxe cyklu for

Pojďme se podrobně podívat na syntaxi cyklu `for`. Za klíčovým slovem `for` následují závorky, v nich jsou tři části oddělené středníkem a na dalším řádku pak následuje jeden nebo více příkazů, které se mají v rámci smyčky provést.

První ze tří částí uzavřených v závorkách slouží většinou k inicializaci proměnné, takzvaného *čítače*. V našem případě je čítačem proměnná s názvem `cislo` a začínáme na hodnotě jedna. Inicializaci provede smyčka jako první a provede ji pouze jednou.

Druhá část je podmínka, která musí být splněná, aby cyklus pokračoval. V našem případě se zajímáme, jestli je hodnota proměnné `cislo` menší nebo rovná deseti.

Třetí část se většinou používá pro aktualizaci čítače, v našem případě čítač zvyšujeme o jedničku. Tato část se provádí na konci každé otáčky cyklu.

Poznámka: Ke zvýšení hodnoty čítače jsme použili postfixovou variantu operátoru ++. Stejně dobře bychom mohli sáhnout po variantě prefixové, nic by se nezměnilo. Nepsaným zvykem bývá postfixová varianta.

První otáčka cyklu `for` tedy vypadá takto:

1. Proměnná `cislo` se nastaví na jedničku.
2. Aktuální hodnota proměnné `cislo` se porovná s desítkou.
3. Podmínka platí (jednička je menší než desítka), a tak se provede tělo cyklu. Program vypíše aktuální hodnotu proměnné `cislo`, tedy jedničku.
4. Hodnota proměnné `cislo` se zvýší na dvojku.

Druhá otáčka cyklu proběhne takto:

1. Aktuální hodnota proměnné `cislo` (tedy dvojka) se porovná s desítkou.
2. Podmínka platí (2 <= 10), a tak program vypíše hodnotu proměnné `cislo`, dvojku.
3. Hodnota proměnné `cislo` se zvýší na trojku.

Všimněte si, že inicializace proběhla jen před první otáčkou cyklu. Přesně to jsme si před chvilkou říkali – inicializace proběhne vždy jen jednou, před první otáčkou.

Podobně jako druhá otáčka cyklu vypadá i otáčka třetí a všechny následující. Každá z nich zvýší hodnotu proměnné `cislo`, až se nakonec dostaneme k závěrečné desáté otáčce, která proběhne takto:

1. Aktuální hodnota proměnné `cislo`, tedy desítka, se porovná s desítkou.

2. Podmínka (10 <= 10) platí, takže program vypíše aktuální hodnotu proměnné `cislo`.
3. Hodnota proměnné `cislo` se zvýší na jedenáct.

Při následující otáčce se číslo 11 uložené v čítači `cislo` opět porovná s desítkou. Tentokrát už podmínka platit nebude (jedenáctka není menší nebo rovna deseti), a tak cyklus končí. Kód uvnitř cyklu už se neprovede, hodnota proměnné `cislo` už se nezvýší a program pokračuje dalším příkazem za smyčkou.

Poznámka: Místo operátoru ++ můžete v cyklu použít operátor `--`. Když obsah závorek za klíčovým slovem for změníte na (`int cislo=10; cislo >= 1; cislo--`), program vypíše čísla od desítky do jedné. Všimněte si změny relačního operátoru z <= na >=.

V našem příkladu patřil k cyklu pouze jeden příkaz. Kdybychom chtěli opakovaně provést větší počet příkazů, museli bychom použít složené závorky, podobně jako u příkazu `if`:

```
for (int cislo = 1; cislo <= 100; cislo++)
{
    cout << cislo;
    cout << "\n";
}
```

A podobně jako u příkazu `if` platí, že pokud přímo za klíčové slovo `for` a závorky omylem napíšeme středník, následující příkaz přestane být součástí cyklu. Například následující kus kódu by vypsal pouze číslo 11:

```
for (int cislo = 1; cislo <= 10; cislo++);
    cout << cislo << "\n";
```

Ke smyčce totiž kvůli středníku patří jen prázdný příkaz, takže smyčka desetkrát provede prázdný příkaz a skončí v okamžiku, kdy proměnná `cislo` dosáhne hodnoty 11. Výpis proměnné už není součástí cyklu, provede se tedy až po skončení cyklu a vypíše hodnotu proměnné `cislo` po skončení cyklu: 11.

Jednotlivé části závorek za klíčovým slovem `for` nejsou povinné. Následující program si inicializaci čítače `cislo` zařídí ještě před začátkem smyčky a stejně tak zvyšování čítače dělá v těle cyklu sám:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 1;
    for ( ; cislo <= 10; )
    {
        cout << cislo << "\n";
        cislo++;
    }
    return 0;
}
```

I když inicializace a příkaz pro zvýšení hodnoty čítače v závorkách za klíčovým slovem `for` chybí, středníky musí zůstat, aby se překladač ve zdrojovém kódu vyznal.

Pozor na nekonečnou smyčku

Kdybychom z předchozího programu vynechali příkaz `cislo++`, cyklus by nikdy neskončil:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 1;
    for ( ; cislo <= 10; )
        cout << cislo << "\n";
    return 0;
}
```

Proměnná `cislo` je totiž inicializovaná na jedničku, a protože ji nikde neměníme, podmínka `cislo <= 10` nikdy nepřestane platit. Smyčce, která sama od sebe nikdy neskončí, se říká *nekonečná smyčka*. Většinou ji poznáte podle toho, že vám na obrazovce létá jeden řádek za druhým a program ne a ne skončit.

Nekonečné smyčky většinou vznikají omylem. Chybička se vloudí – mně se vloudila už mockrát. Pokud narazíte na nekonečnou smyčku, zachovejte klid. Program se dá ukončit klávesovou kombinací Ctrl+C a nekonečnou smyčku pak můžete ve zdrojovém kódu najít a opravit.

Výpočet faktoriálu

Zatím byly všechny příklady použití cyklu `for` velice jednoduché, jen jsme vypisovali seznam čísel od desítky do jedné a nazpět. S cyklem `for` už se ale dají napsat mnohem užitečnější programy.

Následující program počítá faktoriál zadaného čísla. Faktoriál čísla x je součin všech celých čísel od jedné do x. Například faktoriál trojky je $3\times2\times1=6$, faktoriál pětky je $5\times4\times3\times2\times1$ neboli 120, faktoriálem nuly je jednička a faktoriálem jedničky je také jednička.

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo, celkem = 1;
    cout << "Zadejte cele cislo: ";
    cin >> cislo;
    cout << "Faktorial " << cislo << " je ";
    for (int citac = 1; citac <= cislo; citac++)
        celkem *= citac;
    cout << celkem << "\n";
    return 0;
}
```

Předčasné ukončení smyčky

Ve smyčce `for` se dá použít příkaz `break`, se kterým jsme se setkali už u příkazu `switch`. V cyklu `for` se `break` používá většinou v rámci nějaké podmínky. Když na něj program narazí, cyklus okamžitě skončí, i kdyby podmínka cyklu ještě platila.

Například následující program dává uživateli tři pokusy na uhádnutí čísla od jedné do deseti. (Konkrétně jde o trojku, ale nikomu to neříkejte.) Když uživatel uhádne na první nebo druhý

pokus, nemá smysl po něm chtít, aby hádal znovu. Proto v takovém případě využijeme příkaz `break`, který hádací smyčku ukončí.

```
#include <iostream>
using namespace std;
int main(void)
{
   int odhad, tajne = 3;
   cout << "Hadejte cislo od jedne do deseti.\n";
   cout << "Mate tri pokusy.\n";
   for (int i=1; i<=3; i++)
   {
        cout << "Vas tip: ";
        cin >> odhad;
        if (odhad == tajne)
        {
            cout << "Uhadli jste!\n";
            break;
        }
    }
    cout << "Konec programu.\n";
    return 0;
}
```

Následují dvě ukázkové hry. V první hádal uživatel třikrát neúspěšně. Ve druhé se mu číslo podařilo na druhý pokus uhádnout, takže program příkazem `break` ukončil smyčku a ke třetímu hádání už nedošlo.

```
Hadejte cislo od jedne do deseti.
Mate tri pokusy.
Vas tip: 2
Vas tip: 4
Vas tip: 6
Konec programu.
----
Hadejte cislo od jedne do deseti.
Mate tri pokusy.
Vas tip: 2
Vas tip: 3
Uhadli jste!
Konec programu.
```

Příkaz `break` je legitimní součástí jazyka C++, ale přesto byste ho měli používat jen střídmě. Běžná smyčka `for` může skončit jen jedním způsobem, a to nesplněním podmínky uvedené v jejím záhlaví. Jakmile do ní přidáte příkaz `break`, už může skončit dvěma různými způsoby, váš kód se znepřehlední a snadno můžete něco přehlédnout.

Následující program ilustruje alternativní použití operátoru logického součinu (&&) místo příkazu break.

```
#include <iostream>
using namespace std;
int main(void)
{
   int odhad, tajne = 3;
   cout << "Hadejte cislo od jedne do deseti\n";
   cout << " Mate tri pokusy \n";
   bool pokracovat = true;
```

```
    for (int pokusu = 1; pokusu <= 3 && pokracovat == true;
       pokusu ++)
    {
       cout << "Nyni zadejte cislo: ";
       cin >> odhad;
       if (odhad == tajne)
       {
          cout << "Uhadli jste tajne cislo!";
          pokracovat = false;
       }
    }
    cout << "Konec programu";
    return 0;
}
```

Běžně se příkaz `break` používá v kombinaci s nekonečnou smyčkou. Ne každá nekonečná smyčka totiž vniká omylem – někdy se nekonečná smyčka dá použít záměrně. Co kdybychom například chtěli upravit předchozí hru na hádání čísel a dali bychom uživateli nekonečně pokusů?

```
#include <iostream>
using namespace std;
int main(void)
{
   int odhad, pokusu = 1, tajne = 3;
   cout << "Hadejte cislo od jedne do deseti.\n";
   for (;;)
   {
        cout << "Vas tip: ";
        cin >> odhad;
        if (odhad == tajne)
        {
            cout << "Uhadli jste! Pocet pokusu: " << pokusu << ".\n";
            break;
        }
        pokusu++;
    }
    return 0;
}
```

Nový pro vás zřejmě bude jen řádek `for (;;)`, tedy samotná nekonečná smyčka. Jak už jsme si říkali, jednotlivé části závorek za klíčovým slovem `for` nejsou povinné. Krajním případem je cyklus `for (;;)`, u kterého tyto části chybí všechny. Cyklus přesto funguje – jen se sám od sebe nezastaví, takže ho musíte ukončit ručně příkazem `break`. Taková kombinace nekonečné smyčky a příkazu `break` se v praxi někdy používá.

Příkaz continue

V rámci cyklu `for` se dá použít ještě příkaz `continue`. Podobně jako `break` se používá v těle cyklu a většinou v rámci nějaké podmínky. Když na něj program narazí, přestane provádět aktuální iteraci cyklu a pustí se do další. Například následující program účtuje uživateli po třech dolarech za každou položku, ale každý třináctý kus dává zdarma:

```
#include <iostream>
using namespace std;
int main(void)
{
```

```
    int pocet, celkem = 0;
    cout << "Zadejte pocet kusu: ";
    cin >> pocet;
    for (int i=1; i<=pocet; i++)
    {
        if (i % 13 == 0)
            continue;
        celkem += 3;
    }
    cout << "Cena za " << pocet << " kusu je celkem "
        << celkem << " dolaru.\n";
    return 0;
}
```

Následují tři ukázky použití programu. Je vidět, že cena za 12 a 13 kusů je stejná, zatímco za 14 kusů už zákazník zaplatí tři dolary navíc. Za každý třináctý kus se nic neúčtuje, protože program narazí na příkaz `continue` a pustí se rovnou do další otáčky cyklu, aniž by přičítal tři dolary k celkové částce.

```
Zadejte pocet kusu: 12
Cena za 12 kusu je celkem 36 dolaru.
----
Zadejte pocet kusu: 13
Cena za 13 kusu je celkem 36 dolaru.
----
Zadejte pocet kusu: 14
Cena za 14 kusu je celkem 39 dolaru.
```

Příkaz `continue` je sice pevnou součástí jazyka C++, ale stejně jako příkaz `break` vám ho doporučuji používat s rozmyslem. Jakmile do cyklu `for` přidáte příkaz `continue`, jeho logika se zkomplikuje, váš zdrojový kód nebude tak čitelný a snadno můžete něco přehlédnout. Výše uvedený program by se dal bez příkazu `continue` přepsat obyčejnou podmínkou:

```
#include <iostream>
using namespace std;
int main(void)
{
    int pocet, celkem = 0;
    cout << "Zadejte pocet kusu: ";
    cin >> pocet;
    for (int i=1; i<=pocet; i++)
    {
        if (!(i % 13 == 0))
            celkem += 3;
    }
    cout << "Cena za " << pocet << " kusu je celkem "
        << celkem << " dolaru.\n";
    return 0;
}
```

Ještě lepší by bylo negaci `!` posunout dovnitř podmínky, aby se příkaz `if` zjednodušil na `if (i % 13 != 0)`.

Vnořené cykly

Cykly se dají stejně jako podmínky vnořovat, například následující program vypíše pět řádků po deseti znacích X:

```
#include <iostream>
using namespace std;
int main(void)
{
    for (int r=1; r<=5; r++)
    {
        for (int s=1; s<=10; s++)
            cout << "X";
        cout << "\n";
    }
    return 0;
}
```

První cyklus s čítačem `r` je takzvaný vnější cyklus, druhý cyklus s čítačem `s` se označuje jako vnitřní. S každou iterací vnějšího cyklu se provedou všechny iterace vnitřního cyklu. Kdybychom měli cykly přirovnat k hodinám, vnější cyklus by byl minutová ručička a vnitřní cyklus vteřinovka – s každým pohybem minutové ručičky obejde vteřinovka celý ciferník.

V našem programu s řádky a sloupci projde vnitřní smyčka s každou iterací vnější smyčky všech svých deset otáček, takže vypíše desetkrát X. Vnější smyčka pak doplní znak konce řádku a spustí další otáčku, takže vnitřní smyčka zase proběhne desetkrát, zase vypíše deset X, vnější smyčka uzavře řádek a tak dále. Společně obě smyčky vypíšou pět řádků po deseti X.

Vnořené smyčky nejsou dobré jen k výpisu řádků a sloupců tabulkových dat, mají i další využití. Následující program si od uživatele zjistí počet prodavačů a počet nákupů na jednoho prodavače, pak načte hodnotu jednotlivých nákupů a vypíše průměrnou hodnotu nákupu na každého prodavače. Vnější smyčka bude iterovat přes prodavače, vnitřní bude iterovat přes jednotlivé nákupy.

```
#include <iostream>
using namespace std;
int main(void)
{
    int prodavacu, nakupu;
    cout << "Zadejte pocet prodavacu: ";
    cin >> prodavacu;
    cout << "Zadejte pocet nakupu na jednoho prodavace: ";
    cin >> nakupu;
    for (int p=1; p<=prodavacu; p++)
    {
        int hodnota, celkem = 0;
        float prumer;
        for (int n=1; n<=nakupu; n++)
        {
            cout << n << ". nakup prodavace cislo " << p << ": ";
            cin >> hodnota;
            celkem += hodnota;
        }
        prumer = (float) celkem / nakupu;
        cout << "Prumerna hodnota nakupu prodavace cislo #"
             << p << ": " << prumer << "\n";
    }
    return 0;
}
```

Vstup a výstup programu by mohly vypadat například takhle:

```
Zadejte pocet prodavacu: 2
Zadejte pocet nakupu na jednoho prodavace: 3
```

```
1. nakup prodavace cislo 1: 4
2. nakup prodavace cislo 1: 5
3. nakup prodavace cislo 1: 7
Prumerna hodnota nakupu prodavace cislo #1: 5.33333
1. nakup prodavace cislo 2: 8
2. nakup prodavace cislo 2: 3
3. nakup prodavace cislo 2: 4
Prumerna hodnota nakupu prodavace cislo #2: 5
```

Poznámka: Pokud se ve vnitřním cyklu objeví příkaz `break` nebo `continue`, má vliv výhradně na vnitřní cyklus, nikoliv na vnější.

Shrnutí

Smyčky neboli cykly slouží k opakovanému provádění kódu po dobu platnosti nějaké podmínky. V této kapitole jste se dozvěděli, jak se používá smyčka `for`. Ještě než jsme se k ní ale dostali, ukázal jsem vám operátory `++` a `--`, které se používají i v jiných typech cyklů. Vysvětlili jsme si i rozdíl mezi prefixovou a postfixovou variantou obou operátorů.

Také jste se dozvěděli, jak se dá smyčka příkazem `break` předčasně ukončit, jak se příkazem `continue` předčasně ukončuje aktuální iterace cyklu a jak se smyčky dají vkládat do sebe.

Cyklus `for` se většinou používá v případech, kdy víte, kolikrát potřebujete kód zopakovat. Někdy ale nelze odhadnout, kolikrát se smyčka má otočit, například protože její ukončení závisí na vstupu od uživatele. V aplikacích pro zpracování uživatelských dat bývá běžně k vidění smyčka, která načte data od uživatele. Když data nejsou v pořádku, smyčka dá uživateli možnost vstup zopakovat nebo ukončit program. Tato smyčka cyklí, dokud uživatel nezadá správná data, popřípadě dokud program neukončí. Dopředu nevíte, kolikrát proběhne. Následující kapitola vám ukáže dva další typy smyček – cyklus `while` a cyklus `do while`, které se pro tyto situace hodí lépe než cyklus `for`.

Test

1. Co dělá operátor ++?
2. Co dělá operátor `--`?
3. Pokud máme v proměnné `cislo` hodnotu 5, co vypíše příkaz `cout << --cislo`?
4. Co je to iterace?
5. K čemu běžně slouží první část závorek za klíčovým slovem `for`?
6. K čemu běžně slouží druhá část závorek za klíčovým slovem `for`?
7. K čemu běžně slouží třetí část závorek za klíčovým slovem `for`?
8. Může být některá z částí závorek za klíčovým slovem `for` prázdná?
9. K čemu ve smyčce `for` slouží příkaz `break`?
10. K čemu ve smyčce `for` slouží příkaz `continue`?
11. Kdybychom chtěli pomocí dvou vnořených cyklů vytisknout tabulku, který z cyklů by měl na starosti sloupce – vnitřní, nebo vnější?

Cykly while a do while

Cyklus `for` se většinou používá tam, kde předem znáte celkový počet iterací. Někdy se ale počet iterací předem zjistit nedá, například když závisí na vstupu od uživatele. Představte si dejme tomu aplikaci pro zpracování uživatelských dat, která řeší vstup údajů smyčkou. Ve smyčce načte vstupní data a pokud tato data nejsou v pořádku, dá uživateli možnost vstup opakovat nebo ukončit program. Počet iterací se zde nedá zjistit předem – smyčka se bude opakovat, dokud uživatel nezadá správná data nebo program neukončí. Pro smyčky s neznámým počtem iterací se víc hodí cyklus `while`, který se naučíte používat v této kapitole.

Mezi smyčkami s neznámým počtem iterací je ještě jedna důležitá podskupina – smyčky, které se provedou alespoň jednou. Příkladem může být třeba hlavní smyčka kolem celého programu, která zobrazí menu a přebírá vstup od uživatele. Přestože v menu nejspíš bude možnost ukončit program, uživatel ji nemůže vybrat, dokud se menu alespoň jednou nezobrazí. Pro takové případy je určená smyčka `do while`, se kterou se v kapitole také setkáme.

Cyklus while

Cyklus `while` sdílí s cyklem `for` základní vlastnosti všech cyklů: opakuje zadaný kus kódu, dokud je splněna nějaká podmínka. Rozdíl je v obsahu závorek za klíčovými slovy `while` a `for`. Za klíčovým slovem `for` v závorkách následují tři části – inicializace, podmínka a aktualizace čítače. Naproti tomu v závorkách za klíčovým slovem `while` je pouze podmínka; aktualizaci a inicializaci si musíte vyřešit sami.

Rozdíl je vidět na následujícím programu, který vypisuje čísla od jedné do deseti. Jde o tentýž program, se kterým už jsme se setkali v předchozí kapitole:

```
#include <iostream>
using namespace std;
int main(void)
{
    for (int cislo = 1; cislo <= 10; cislo++)
        cout << cislo << "\n";
    return 0;
}
```

Kdybychom ho přepsali pomocí cyklu `while`, vypadal by následovně:

```
#include <iostream>
using namespace std;
int main(void)
{
```

```
    int cislo = 1;
    while (cislo <= 10)
    {
        cout << cislo << "\n";
        cislo++;
    }
}
```

Poznámka: Oba příkazy z těla cyklu `while` jsme mohli sloučit do jednoho, `cout << cislo++ << "\n"`. Varianta se dvěma příkazy je vybraná schválně, protože je pro vás momentálně přehlednější; nemusíte přemýšlet nad rozdílem mezi postfixovým a prefixovým ++.

Ve verzi se smyčkou `while` jsme museli proměnnou `cislo` deklarovat a inicializovat sami ještě před smyčkou, protože v závorkách za `while` žádná odpovídající sekce není. Navíc jsme museli čítač v těle cyklu ručně aktualizovat pomocí operátoru ++. (Aktualizace čítače se ale dá provést přímo v závorkách za `while`, za okamžik si to ukážeme.)

Aktualizace čítače je u smyčky `while` důležitá, protože kdybychom ji vynechali, vyrobili bychom si nekonečnou smyčku. Například následující kus kódu čítač neaktualizuje, takže nikdy neskončí:

```
int cislo = 1;
while (cislo <= 10)
    cout << cislo << "\n";
```

Na aktualizaci proměnné použité v podmínce cyklu `while` se zapomíná celkem často. U cyklu `for` k této chybě dochází zřídka, protože je zde pro aktualizaci čítače vyhrazena třetí část závorek za klíčovým slovem `for`.

Jinak se na cyklus `while` vztahují všechna syntaktická pravidla, která jsme v předchozí kapitole probírali u cyklu `for`. Například pokud k cyklu `while` patří víc než jeden příkaz, musíte příkazy uzavřít do složených závorek. Proto jsou složené závorky i v předchozím programu pro výpis čísel od jedničky do deseti:

```
while (cislo <= 10)
{
    cout << cislo << "\n";
    cislo++;
}
```

Proměnná `cislo` se zde aktualizuje v těle cyklu, ale stejně dobře bychom ji mohli provést přímo v podmínce:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo = 0;
    while (cislo++ < 10)
        cout << cislo << "\n";
}
```

Aktualizace čísla přímo v podmínce vyžaduje oproti předchozí verzi dvě změny. Za prvé, proměnná `cislo` se musí inicializovat na nulu, protože se o jedničku zvedne během vyhodnocování podmínky. A za druhé jsme ze stejného důvodu museli přepsat relační operátor `<=` na `<`, protože jinak by cyklus vypsal ještě jedenáctku.

Jelikož máme operátor ++ použitý ve výrazu, musíme si dát pozor na to, kterou jeho variantu použijeme. Když použijeme variantu postfixovou, `cislo++ < 10`, porovnání se provede ještě se starou hodnotou proměnné před jejím zvýšením. Kdybychom použili variantu prefixovou, porovnání už by se provedlo se zvýšenou hodnotou proměnné a podmínku bychom museli přepsat na `++cislo <= 10`.

Stejně jako u cyklu `for` platí, že pokud hned po závorkách za klíčovým slovem `while` napíšete středník, překladač si ho vyloží jako prázdný příkaz a následující příkaz přestane být součástí cyklu. Schválně, jaký bude výstup následujícího kódu?

```
int cislo = 1;
while (cislo <= 10);
    cout << cislo++ << "\n";
```

Výstup nebude žádný, protože program vůbec neskončí. K cyklu `while` patří jen prázdný příkaz, a jelikož ten nijak nemění hodnotu proměnné `cislo`, podmínka bude platit věčně a stejně dlouho poběží i cyklus.

Srovnání cyklů for a while

Doposud jsme se zabývali programem s pevným počtem iterací, na kterém není praktický rozdíl mezi cykly `for` a `while` nijak zvlášť patrný. Výhody cyklu `while` oproti cyklu `for` vyniknou teprve v okamžiku, kdy potřebujeme smyčku s předem neznámým počtem iterací – například smyčku, která se opakuje v závislosti na vstupu od uživatele.

Následující program chce od uživatele kladné číslo a smyčka se zadáním čísla se opakuje, dokud uživatel skutečně nezadá kladné číslo. Předem se nedá odhadnout, kolikrát se smyčka provede. Pokud uživatel zadá kladné číslo hned napoprvé, smyčka se neprovede vůbec, ale stejně dobře se může provést vícekrát.

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo;
    cout << "Zadejte prosim kladne cislo: ";
    cin >> cislo;
    while (cislo <= 0)
    {
        cout << "Cislo musí byt kladne, zkuste znovu: ";
        cin >> cislo;
    }
    cout << "Zadane cislo: " << cislo << "\n";
    return 0;
}
```

Vstup a výstup programu vypadá například takhle:

```
Zadejte prosim kladne cislo: 0
Cislo musí byt kladne, zkuste znovu: -1
Cislo musí byt kladne, zkuste znovu: 3
Zadane cislo: 3
```

Se smyčkou `for` by se program psal hůř. Šlo by to, ale pokud počet iterací neznáme předem, je lepší použít smyčku `while`.

Příkaz break

Cyklus `while` je pro náš program lepší volba než `for`, ale přesto má program ještě dvě chyby; jednu menší a jednu větší. Menší chyba je, že kód pro zadání čísla musíme psát dvakrát: jednou před smyčkou a podruhé uvnitř smyčky. Tomuto opakování bychom se mohli vyhnout použitím smyčky `do while`, kterou popisuje následující část kapitoly. (Smyčka `do while` ovšem vede k opakování kódu v jiných případech. Ve smyčkách zkrátka člověk musí dělat kompromisy, stejně jako v životě.)

Hlavní problém je, že se uživatel ze smyčky dostane jedině zadáním kladného čísla. Takhle dobrý návrh programu nevypadá. Vstup sice kontrolovat musíme, ale pokud uživatel chce, měl by mít možnost program ukončit bez zadávání kladného čísla. Následující úprava přidává možnost ukončit program bez zadávání dat; spoléháme na příkaz `break`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo;
    char vyber;
    cout << "Zadejte prosim kladne cislo: ";
    cin >> cislo;
    while (cislo <= 0)
    {
        cout << "Cislo musí byt kladne, zadate jine? (A/N) ";
        cin >> vyber;
        if (vyber != 'A')
            break;
        cout << "Zadejte cislo: ";
        cin >> cislo;
    }
    cout << "Zadane cislo: " << cislo << "\n";
    return 0;
}
```

Takhle vypadá, když uživatel časem zadá kladné číslo:

```
Zadejte prosim kladne cislo: 0
Cislo musí byt kladne, zadate jine? (A/N) A
Zadejte cislo: -1
Cislo musí byt kladne, zadate jine? (A/N) A
Zadejte cislo: 3
Zadane cislo: 3
```

A takhle to dopadne, když uživatel kladné číslo nezadá a rozhodne se program ukončit:

```
Zadejte prosim kladne cislo: -2
Cislo musí byt kladne, zadate jine? (A/N) N
Zadane cislo: -2
```

Příznaky

Poslední verze programu je pokrok, protože uživatel už není v pasti vstupní smyčky a může program ukončit. Druhý příklad vstupu a výstupu programu ale ukazuje další problém: Závěrečný `cout` číslo vypíše, i když není kladné.

V ideálním případě bychom chtěli, aby program vypisoval jen korektní data. A pokud vstup korektní není, aby na to upozornil. Současný kód nám ale neumožňuje rozlišit, jestli smyčka skončila zadáním kladného čísla, nebo jestli se uživatel rozhodl zadávání ukončit.

V předchozí kapitole jsem vám doporučoval, abyste příkaz `break` používali s rozmyslem, protože do smyčky přidá další výstupní bod, takže komplikuje čtení kódu a zvyšuje pravděpodobnost nějaké logické chyby. Tato rada platí i pro cyklus `while`. Místo příkazu `break` můžete využít logickou proměnnou:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo;
    char vyber;
    bool konec = false;
    cout << "Zadejte prosim kladne cislo: ";
    cin >> cislo;
    while (cislo <= 0 && !konec)
    {
        cout << "Cislo musí byt kladne, zadate jine? (A/N) ";
        cin >> vyber;
        if (vyber != 'A')
            konec = true;
        else
        {
            cout << "Zadejte cislo: ";
            cin >> cislo;
        }
    }
    if (konec)
        cout << "Nezadali jste kladne cislo.\n";
    else
        cout << "Zadane cislo: " << cislo << "\n";
    return 0;
}
```

Takhle vypadá, když uživatel časem zadá kladné číslo:

```
Zadejte prosim kladne cislo: -3
Cislo musí byt kladne, zadate jine? (A/N) A
Zadejte cislo: 3
Zadane cislo: 3
```

A takhle vypadá, když uživatel kladné číslo nezadá a rozhodne se program ukončit. Všimněte si, že tentokrát už program zadané číslo nevypíše. Místo toho oznámí, že nebylo zadáno kladné číslo.

```
Zadejte prosim kladne cislo: 0
Cislo musí byt kladne, zadate jine? (A/N) A
Zadejte cislo: -1
Cislo musí byt kladne, zadate jine? (A/N) N
Nezadali jste kladne cislo.
```

Novinkou oproti předchozí verzi je složená podmínka cyklu `while` a nová logická proměnná `konec`. Proměnná `konec` je použita jako *příznak* – slouží ke sledování nějaké podmínky. Naše nová varianta smyčky pokračuje, dokud uživatel zadává chybná data *a zároveň* chce zkusit zadávání znovu. Proto jsou za klíčovým slovem `while` dvě podmínky spojené operátorem `&&`.

Poznámka: Při programování cyklu `while` se složenou podmínkou se často stává, že člověk omylem zamění operátor `&&` za `||` nebo naopak. V tomto případě je použití operátoru `&&` nasnadě, jindy výběr správného operátoru tak zjevný nebývá. Pokud například chcete smyčku opakovat, dokud bude číslo mimo interval 1–10, použili byste podmínku `(i < 1 && i > 10)`, nebo podmínku `(i < 1 || i > 10)`? Správně je ta druhá. První se nesplní nikdy, protože žádné číslo nemůže být menší jedné a zároveň větší než deset. Kdybyste chtěli použít operátor `&&`, podmínka by musela znít `(!(i >= 1 && i <= 10))`.

První z podmínek je `cislo <= 0`. Pokud se vyhodnotí jako nepravdivá, data od uživatele jsou v pořádku a vůbec se nemusíme zabývat tím, jestli uživatel chce nebo nechce ukončit program. Jak jsme si říkali v předchozí kapitole, operátor `&&` svůj druhý operand v takovém případě vůbec nevyhodnocuje. Proto smyčka skončí, v proměnné `konec` zůstane hodnota `false` nastavená během inicializace a běh programu pokračuje `else` větví závěrečné podmínky.

Pokud podmínka `cislo <= 0` platí, data od uživatele jsou chybná a musí se vyhodnotit druhá podmínka, tedy jestli má proměnná `konec` hodnotu `false`. Proměnná `konec` může mít hodnotu `false` za dvou okolností. První možnost je, že jde o uživatelův první pokus o zadání dat a data jsou zadaná špatně. V takovém případě jsme se uživatele ještě neptali, jestli chce zadávání ukončit, a proměnná `konec` tudíž obsahuje hodnotu `false`, na kterou jsme ji inicializovali. (Kdybychom ji bývali inicializovali na hodnotu `true`, program by po prvním špatně zadaném čísle skončil.)

Druhá možnost je, že se uživatel o zadání pokoušel alespoň dvakrát, a opět špatně. V takovém případě už jsme se ho zeptali, jestli to chce zkusit znovu, a hodnota proměnné `konec` je nastavená podle jeho odpovědi.

Pokud má proměnná konec hodnotu `false`, smyčka pokračuje. Pokud ale uživatel řekl, že už chce skončit, podmínka `!konec` se vyhodnotí jako `false` (alias `!true`). V takovém případě smyčka skončí a program pokračuje `if` větví následující podmínky.

Když se dostaneme k podmínce za cyklem, rozhodneme se podle hodnoty proměnné `konec`. Pokud je nastavená na `false`, nutně to znamená, že uživatel data zadal správně. Naopak pokud je nastavená na `true`, uživatel zadávání vzdal. Uvnitř cyklu se tedy podle proměnné `konec` pozná, jestli už má cyklus skončit, a v podmínce za cyklem se podle ní pozná, jestli uživatel zadal správná data.

While true

V předchozí kapitole jsme se zabývali i nekonečným cyklem `for (;;)`, ze kterého se dá vyskočit výhradně příkazem `break`. Podobně se dá využít i cyklus `while`, u kterého nekonečné varianty dosáhnete například podmínkou `while (1)` nebo `while (true)`. Takto by vypadal náš program pro zadávání čísel, kdybychom ho přepsali do nekonečného cyklu `while`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo;
    char volba;
    bool ukonceno = false;
    while (true)
    {
        cout << "Zadejte kladne cislo: ";
        cin >> cislo;
        if (cislo > 0)
            break;
```

```
            cout << "Cislo musi byt kladne, zadate jine? (A/N) ";
            cin >> volba;
            if (volba != 'A')
            {
                ukonceno = true;
                break;
            }
        }
        if (!ukonceno)
            cout << "Zadane cislo: " << cislo << "\n";
        else
            cout << "Nezadali jste kladne cislo.\n";
        return 0;
    }
```

Výhodou této verze je, že nemusíte číslo od uživatele načítat na dvou různých místech. Naopak nevýhodou je nižší čitelnost programu, protože se podmínka pro ukončení cyklu přestěhovala ze záhlaví cyklu někam do jeho těla. Ideální řešení nabízí smyčka `do while`, ke které se dostaneme za chvíli.

Příkaz continue

Stejně jako ve smyčce `for` můžete i v cyklu `while` použít příkaz `continue`. Jak už jsme si říkali v předchozí kapitole, příkaz `continue` se používá v těle cyklu, obvykle v nějaké podmínce. Když na něj program narazí, aktuální iterace cyklu se přeruší a rovnou se začne další.

V předchozí kapitole jsme si využití příkazu `continue` ukazovali na programu, který účtoval každou třináctou položku zdarma. Takto by program vypadal se smyčkou `while`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int pocet, i = 0, celkem = 0;
    cout << "Zadejte počet kusu: ";
    cin >> pocet;
    while (i++ < pocet)
    {
        if (i % 13 == 0)
            continue;
        celkem += 3;
    }
    cout << "Cena za " << pocet << " kusu je celkem "
        << celkem << " dolaru.\n";
    return 0;
}
```

Poznámka: Operátor `%` (modulo) kontroluje, jestli po dělení čísla `i` třinácti dostaneme zbytek nula, tedy jestli je číslo `i` dělitelné třinácti.

I tentokrát platí, že byste příkaz `continue` měli používat jen střídmě. Stejně jako `break` totiž komplikuje chování cyklu, takže znepřehledňuje kód a zvyšuje pravděpodobnost, že někde uděláte chybu. Ve výše uvedeném příkladu se dá příkaz `continue` nahradit podmínkou:

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

```
int main(void)
{
    int pocet, i = 0, celkem = 0;
    cout << "Zadejte pocet kusu: ";
    cin >> pocet;
    while (i++ < pocet)
        if (i % 13 != 0)
            celkem += 3;
    cout << "Cena za " << pocet << " kusu je celkem "
        << celkem << " dolaru.\n";
    return 0;
}
```

Vnořené smyčky

V předchozí kapitole jsem vám ukázal, jak se dají cykly `for` skládat dohromady. Podobně můžete spojit i dva cykly `while`, popřípadě `for` a `while`. Vnořené cykly `for` jsme si ukazovali na programu, který vypíše pět řádků po deseti znacích X. Takto by program vypadal po přepsání na smyčky `while`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int x = 0;
    while (x++ < 5)
    {
        int y = 0;
        while (y++ < 5)
            cout << "X";
        cout << "\n";
    }
    return 0;
}
```

Proměnná `y`, která slouží jako čítač vnitřní smyčky, se musí před začátkem vnitřní smyčky vždy inicializovat na nulu. Deklarovat bychom ji mohli mimo obě smyčky (vedle proměnné `x`), ale pak bychom ji stejně museli před začátkem každého vnitřního cyklu znovu nastavovat na nulu, jinak by se vnitřní cyklus provedl jen poprvé.

V tomto případě mají obě smyčky předem daný počet iterací, takže by bylo lepší použít cyklus `for`. Obě možnosti ale fungují stejně dobře.

Cyklus do while

Cyklus `do while` se do velké míry podobá cyklu `while`. Hlavní rozdíl je v tom, že zatímco podmínka cyklu `while` se kontroluje hned na začátku, podmínka cyklu `do while` se kontroluje až na jeho konci. Z toho plyne, že cyklus `do while` se vždy provede alespoň jednou, zatímco cyklus `while` se nemusí provést vůbec, pokud jeho podmínka už na začátku není splněná.

Syntaxe

Cyklus do `while` vypadá takto:

```
do {
    příkazy;
} while (podmínka);
```

Klíčové slovo `do` otevírá smyčku. Za ním následují složené závorky a příkazy, které tvoří tělo cyklu. Za uzavírací složenou závorkou je klíčové slovo `while`, podmínka uzavřená do závorek a středník.

Příklad cyklu do while

Následuje upravená verze programu, kterým jsme se zabývali dříve v této kapitole. Program se uživatele ptá na kladné číslo a nepřestane, dokud uživatel skutečně nezadá kladné číslo nebo program neukončí. Pokud uživatel zadá kladné číslo, program ho vypíše. Nová verze programu nahrazuje smyčku `while` smyčkou `do while`:

```
#include <iostream>
using namespace std;
int main(void)
{
    int cislo;
    char volba;
    bool konec = false;

    do {
        cout << "Zadejte kladne cislo: ";
        cin >> cislo;
        if (cislo <= 0)
        {
            cout << "Cislo musi byt kladne, zadate jine? (A/N) ";
            cin >> volba;
            if (volba != 'A')
                konec = true;
        }
    } while (cislo <= 0 && !konec);

    if (!konec)
        cout << "Zadane cislo: " << cislo << "\n";
    else
        cout << "Nezadali jste kladne cislo.\n";
    return 0;
}
```

Následují ukázky práce s programem. Na první zadá uživatel hned napoprvé správné číslo:

```
Zadejte kladne cislo: 4
Zadane cislo: 4
```

V dalším případě zadá kladné číslo po dvou neúspěšných pokusech:

```
Zadejte kladne cislo: 0
Cislo musi byt kladne, zadate jine? (A/N) A
Zadejte kladne cislo: -1
Cislo musi byt kladne, zadate jine? (A/N) A
```

```
Zadejte kladne cislo: 4
Zadane cislo: 4
```

A konečně ve třetím případě uživatel po dvou neúspěšných pokusech ukončí program:

```
Zadejte kladne cislo: 0
Cislo musi byt kladne, zadate jine? (A/N) A
Zadejte kladne cislo: -1
Cislo musi byt kladne, zadate jine? (A/N) N
Nezadali jste kladne cislo.
```

Porovnání cyklů while a do while

Nová varianta programu se smyčkou `do while` už na rozdíl od původní varianty s `while` nemusí před smyčkou opakovat zadávání čísla. Na druhou stranu zase uvnitř smyčky opakuje podmínku `cislo <= 0`, což původní verze nemusela.

Obecně vzato se cyklus `do while` hodí v případech, kdy se smyčka musí provést alespoň jednou, aby vůbec mělo smysl kontrolovat její podmínku. Pokud si vzpomínáte, u předchozího řešení programu se smyčkou `while` měla proměnná `konec` hodnotu `false` ve dvou případech. Prvním z nich byla první otáčka smyčky, kdy uživatel ještě žádná data vůbec nezadával a proměnná konec obsahovala hodnotu `false` nastavenou během inicializace. Druhá možnost byla, že jde o druhý nebo pozdější pokus o zadání hodnoty a uživatel si vybral, že chce data zkusit zadat ještě jednou. Použitím smyčky `do while` se kód zjednoduší – první případ odpadne a proměnná `konec` vždy obsahuje výhradně volbu uživatele.

Náš příklad se zadáváním čísla (ať už kladného nebo jakéhokoliv jiného) je klasickou ukázkou situace, ve které musí smyčka proběhnout alespoň jednou a teprve pak má smysl kontrolovat podmínku. Dalším běžným případem je menu. Dejme tomu, že by program měl zobrazit následující menu:

```
Menu
====
1. Pridat zaznam
2. Upravit zaznam
3. Smazat zaznam
4. Konec
```

Jakmile si uživatel vybere některou z položek 1–3, program provede vybranou operaci a znovu zobrazí menu. Když si uživatel vybere položku 4, program skončí. Menu se vždy zobrazí alespoň jednou; uživatel nemůže program ukončit, aniž by prošel přes menu. Proto se menu hodí napsat pomocí smyčky `do while`.

Rozsah platnosti proměnné

Proměnné použité v podmínce cyklu `do while` nesmí být deklarované uvnitř cyklu. V předchozí ukázce jsme proměnné `cislo` a `konec` deklarovali před smyčkou:

```
int cislo;
char volba;
bool konec = false;
do {
    // příkazy
}  while (cislo <= 0 && !konec);
```

Kdybychom je deklarovali uvnitř cyklu (viz následující kus kódu), program by se nepřeložil. Překladač by zvýraznil podmínku cyklu a stěžoval by si, že identifikátory `cislo` a `konec` nejsou deklarované.

```
do {
    int cislo;
    bool konec = false;
    // příkazy
} while (cislo <= 0 && !konec);
```

Příčinou chyby je takzvaný *rozsah platnosti proměnných*. Už ve třetí kapitole jsme si říkali, že každou proměnnou musíte před prvním použitím deklarovat. Jakmile je proměnná deklarovaná, můžete se na ni odkazovat odkudkoliv, „kde je vidět".

Až doposud jsme většinu proměnných deklarovali ve funkci `main`, hned za otevírací složenou závorkou. Takto deklarované proměnné platí až k odpovídající uzavírací složené závorce, která uzavírá funkci `main`. A protože naše programy až doposud neměly jinou funkci než `main`, na deklarované proměnné jsme se mohli odkazovat kdekoliv ve zdrojovém kódu.

V posledním příkladu jsme ale proměnné `cislo` a `konec` deklarovali za otevírací složenou závorkou smyčky `do while`. To znamená, že platí pouze do odpovídající koncové složené závorky, která uzavírá cyklus. (Oblasti mezi dvěma složenými závorkami se říká *blok*.)

Klíčové slovo `while` a závorky za ním už jsou mimo tělo cyklu, tedy za složenou závorkou, která cyklus uzavírá. A protože jsme proměnné `cislo` a `konec` deklarovali v těle smyčky, za uzavírací složenou závorkou už neplatí. Proto si překladač v podmínce cyklu stěžuje na nedeklarované identifikátory.

U cyklu `do while` se tento problém objevuje mnohem častěji než u cyklu `while` nebo `for`. U cyklů `while` a `for` je totiž podmínka na začátku cyklu, takže jakékoliv proměnné v této podmínce použité musíte nutně deklarovat už před cyklem (nebo v závorkách za klíčovým slovem `for`). Naproti tomu u cyklu `do while` je podmínka až za tělem cyklu, a tak je celkem snadné udělat chybu a deklarovat proměnnou v těle cyklu.

Toto je naše první, ale zdaleka ne poslední zastavení u rozsahu platnosti proměnných. Rozsah platnosti se netýká jen cyklů. Běžně se s ním budete setkávat u funkcí, kterým se bude věnovat následující kapitola.

Shrnutí

V předchozí kapitole jsme uvedli první z několika typů smyček, a to cyklus `for`. Cyklus `for` se dobře hodí tam, kde předem znáte počet iterací. Často ale počet iterací neznáte, například když závisí na vstupu od uživatele. Příkladem je třeba program pro vstup dat, kterým jsme se v této kapitole zabývali. Ten uživatele poprosí o zadání čísla a vstup opakuje tolikrát, dokud nedostane správné číslo nebo dokud se uživatel nerozhodne program ukončit. Počet iterací zde předem neznáme, protože cyklus pokračuje, dokud uživatel nezadá správné číslo nebo program neukončí.

Když počet iterací neznáte předem, lepší než cyklus `for` je cyklus `while`, se kterým jsme se seznámili v této kapitole. Zatímco závorky za klíčovým slovem `for` obsahují tři části (inicializaci, podmínku a aktualizaci), v závorkách za klíčovým slovem `while` je pouze podmínka; případnou aktualizaci a inicializaci musíte vyřešit sami někde v kódu.

V některých situacích neznáme počet iterací předem, ale víme, že cyklus proběhne alespoň jednou. Jako příklad jsme v této kapitole zmínili smyčku s menu. I když menu obsahuje položku pro

ukončení programu, vždy se zobrazí alespoň jednou. (Uživatel nemůže vybrat ukončení programu, aniž by se nejprve zobrazilo menu.) V takových případech je nejlepší smyčka `do while`, kterou jste se v této kapitole také naučili používat a která nás poprvé dovedla k problému s rozsahem proměnných.

Až doposud měly naše programy jen jednu funkci, `main`. Kromě této funkce (bez které se žádný program neobejde) mohou mít programy více svých vlastních funkcí. U složitějších programů bývá užitečné nepsat všechen kód do funkce `main`, ale rozdělit ho do několika menších funkcí. Jak se funkce přidávají a používají, bude předmětem následující kapitoly.

Test

1. Která ze smyček `for`, `while` a `do while` se vždy provede alespoň jednou?
2. Která ze smyček `for`, `while` a `do while` se nejvíc hodí pro předem známý počet iterací?
3. Do závorek za klíčové slovo `while` patří inicializace, podmínka, nebo aktualizace?
4. Může být v závorkách za klíčovým slovem `while` hodnota `true`, `while (true)`?
5. Může být podmínka cyklu `while` složená z několika menších podmínek?
6. K čemu je v cyklu `while` příkaz `break`?
7. K čemu je v cyklu `while` příkaz `continue`?
8. Co nebo kdo je to příznak?
9. Kdybyste chtěli dvěma vnořenými smyčkami `while` vytisknout několik řádků a sloupců, která smyčka by se starala o řádky a která o sloupce?
10. Platí proměnná deklarovaná v těle cyklu `do while` i v závorkách za klíčovým slovem `while`?

Funkce

Funkce je skupina příkazů, které společně plní nějaký úkol. Naše programy měly až doposud jen jednu funkci (`main`) a občas jsme používali funkce ze standardní knihovny, například mocninu `pow` z knihovny `cmath`.

Žádnou další funkci kromě `main` programy v principu nepotřebují. Při psaní složitějších programů ale záhy zjistíte, že máte funkci `main` neúnosně dlouhou. Překladači a uživatelům je to úplně jedno, ale vám by to jedno být nemělo. Pokud vaše funkce `main` zabere několik stránek textu, těžko se v ní vyznáte a těžko se vám v ní budou opravovat případné chyby.

Kdybych měl sáhnout k přirovnání, tato kniha má několik set stránek. Kdyby každá kapitola nebyla rozdělená do několika částí, text by se vám hůř četl. A ještě horší by bylo, kdyby celou knihu tvořila jedna předlouhá kapitola. Rozdělení knihy na kapitoly a podkapitoly zvyšuje čitelnost textu.

Podobně se dá na menší části rozdělit kód. Jeho rozdělení do jednotlivých funkcí je čistě na vás, ale logicky se nabízí rozdělení podle toho, co která část kódu dělá. Například program pro aritmetické výpočty by mohl obsahovat funkci pro načtení vstupu od uživatele, funkci pro provedení výpočtu a třetí funkci pro výstup výsledků. Analogicky se kniha dělí na kapitoly a podkapitoly, z nichž každá se věnuje jinému tématu. Jedna kapitola je věnovaná proměnným, jiná (tato!) zase funkcím.

Rozdělení kódu do samostatných funkcí má kromě zpřehlednění programu i další výhody. Pokud má funkce na starosti konkrétní úkol, který je v programu potřeba provést několikrát (například výstup na tiskárnu), příslušný kód stačí napsat jednou a pak už jen voláte funkci. Kdybyste kód neměli zapsaný v podobě funkce, museli byste ho pokaždé opisovat znovu.

Teď už se mi vás doufám podařilo přesvědčit, že je rozdělení kódu do funkcí užitečné. Pojďme se podívat, jak se dělá.

Definování a volání funkce

Když chcete přidat nějakou funkci, musíte ji umět definovat a volat. Popis těchto dvou kroků vyžaduje novou terminologii, a tak následuje krátká terminologická odbočka.

Chvilka terminologie

Podívejme se na jednoduchý program s jedinou funkcí `main`:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, world!\n";
    return 0;
}
```

První řádek, `int main()`, se označuje jako *hlavička funkce*. Na rozdíl od příkazů se hlavička funkce nekončí středníkem. Hlavička začíná *návratovým typem* funkce, za kterým následuje *jméno funkce* a *seznam parametrů*. V našem případě má funkce návratový typ `int`, jmenuje se `main` a seznam parametrů je prázdný (což nemusí být vždy).

Za hlavičkou funkce a otevírací složenou závorkou začíná *tělo funkce*, které končí odpovídající koncovou složenou závorkou. V rámci těla funkce se samozřejmě mohou vyskytnout další složené závorky, například skupina podmíněných příkazů nebo skupina příkazů ve smyčce.

Tělo funkce může obsahovat libovolný počet příkazů. V našem případě jsou příkazy dva. Ten druhý, `return 0`, vrací *návratovou hodnotu* funkce. Návratovou hodnotu musí vracet všechny funkce, jedinou výjimkou jsou funkce s návratovým typem `void`.

Hlavička funkce a její tělo se společně označují jako *definice funkce*. Spustit neboli *zavolat* se dá jen funkce, která byla definována. Funkce se volají explicitně z kódu. Jedinou výjimkou je funkce `main`, která se automaticky provede po spuštění programu. Následující část kapitoly vám vysvětlí, jak můžete definovat a zavolat svou vlastní funkci.

Definování funkce

Zkusme si příklad s *Hello, world!* rozdělit na dvě funkce. Funkce `pozdrav` vypíše `Hello, world` a funkce `main` pouze zavolá funkci `pozdrav`. Komentáře (řádky začínající na `//`) označují začátek a konec definice funkce `pozdrav` a místo, ze kterého ji voláme.

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

// začíná definice funkce pozdrav
void pozdrav(void)
{
    cout << "Hello, world!\n";
}
// konec definice

int main()
{
    pozdrav(); // volání funkce
    return 0;
}
```

Funkci `pozdrav` musíme nejprve definovat. Klíčové slovo `void` před názvem funkce znamená, že funkce nevrací žádnou hodnotu, a `void` v závorkách za názvem funkce znamená, že funkce nemá žádné parametry. Kdybychom závorky nechali prázdné (viz hlavičku funkce `main`), znamenalo by to totéž co `(void)`. Kterou variantu si vyberete, je čistě otázkou vkusu, fungují úplně stejně.

Tělo funkce `pozdrav` obsahuje jediný příkaz, který vypíše řetězec `Hello, world!`. Jelikož funkce nevrací žádnou hodnotu, příkaz `return` uvádět nemusíme, překladač si ho domyslí. Pokud ale chcete, klidně můžete `return` uvést explicitně:

```
void pozdrav(void)
{
    cout << "Hello, world!\n";
    return;
}
```

Volání funkce

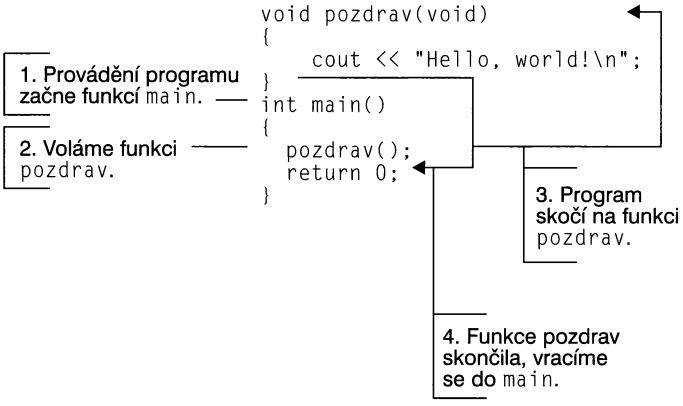
Dokud funkci `pozdrav` nezavoláte, chová se jako příslovečný padající strom v lese, který nikdo nevidí a neslyší. Sedí ve zdrojovém kódu a nedělá nic. Z funkce `main` ji voláme následujícím řádkem:

```
pozdrav();
```

Prázdné závorky znamenají, že funkci nepředáváme žádné parametry. (O předávání parametrů a návratových hodnotách se budeme bavit později.) Příkazy se provádí v následujícím pořadí:

1. Provádění programu začne vždy funkcí `main`.
2. Provede se první příkaz funkce `main`, volání funkce `pozdrav`.
3. Program začne provádět funkci `pozdrav` a vypíše řetězec „`Hello, world`".
4. Jakmile funkce `pozdrav` skončí, program pokračuje následujícím doposud neprovedeným příkazem funkce `main`. V našem případě jde o příkaz `return 0`, který ukončí program.

Obrázek 9.1 ukazuje pořadí provádění programu graficky.



Obrázek 9.1: Pořadí provádění programu Hello, world

Prototypy funkcí

Jelikož provádění programu vždy začíná funkcí `main`, zdálo by se logické uvádět tuto funkci na začátku programu, ještě před ostatními funkcemi:

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

int main()
{
    pozdrav();
    return 0;
}

void pozdrav(void)
{
    cout << "Hello, world!\n";
}
```

Tento kód se ale nepřeloží. Překladač zvýrazní volání funkce `pozdrav` v těle funkce `main` a bude si stěžovat na nedeklarovaný identifikátor. Když totiž překladač při své cestě odshora dolů kódem narazí na volání funkce, potřebuje vědět, že jde o funkci, a potřebuje znát její parametry a návratový typ.

Dokud jsme funkci `pozdrav` definovali nad funkcí `main`, všechno bylo v pořádku. Když jsme teď ale definici funkce `pozdrav` přesunuli za `main`, překladač narazí na její volání ještě před tím, než si přečte její definici.

Mohli bychom všechny funkce definovat před `main`, ale tím bychom program znepřehlednili. Provádění programu vždy začíná funkcí `main` – nezávisle na pořadí, ve kterém funkce definujete. V programech s velkým počtem funkcí slouží `main` jako „telefonní ústředna", která spojuje volání jednotlivých funkcí. Přečtením funkce `main` si tedy můžete udělat dobrý obrázek o tom, jak celý program funguje. Když `main` pohřbíte pod hromadu dalších funkcí, kdokoliv, kdo po vás bude kód číst, ji bude složitě hledat. Někdy se funkce navíc složitě odkazují jedna na druhou, a tak je těžké trefit pořadí volání, které nezpůsobí chybu překladu.

Řešením je *prototypování* všech funkcí (s výjimou `main`, která je v každém programu a kterou tedy překladač dobře zná). Prototyp funkce `pozdrav` je vidět v následujícím programu:

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

void pozdrav(void); // prototyp

int main()
{
    pozdrav();
    return 0;
}

void pozdrav(void)
{
    cout << "Hello, world!\n";
}
```

Prototyp uvádějte nad definice funkcí, aby na něj překladač narazil ještě před voláním libovolné funkce. Prototyp má podobnou úlohu jako hlavička funkce. Hlavní rozdíl je v tom, že končí středníkem, protože jde o příkaz. Hlavička funkce středníkem končit nesmí.

Poznámka: Mezi prototypem a hlavičkou funkce jsou i další rozdíly. Vrátíme se k nim později v této kapitole, až začneme používat parametry funkcí.

Životnost proměnných a rozsah platnosti

Zatím jsme všechny proměnné deklarovali na začátku funkce `main`. Pokud už program žádné další funkce nemá, funkce `main` tvoří celé jeho tělo, a tak jsou proměnné deklarované v `main` dostupné z celého programu. Jakmile ale kód začneme dělit na samostatné funkce, objeví se problémy s *rozsahem platnosti* a *životností* proměnných. Rozsahu platnosti už jsme se dotkli v předchozí kapitole v souvislosti se smyčkou `do while`, životnost proměnných je pro nás nová.

Místní proměnné

Funkce se dají volat vícekrát. Následující program obsahuje smyčku, která volá funkci `pocitej`. Smyčka běží, dokud se uživatel nerozhodne ji přerušit, a funkce `pocitej` při každém volání vypisuje, kolikrát už ji někdo zavolal. Jinými slovy bychom chtěli, aby při prvním volání vypsala `Uz me volali 1x`, při druhém volání `Uz me volali 2x`, a tak dále.

```
#include <iostream>
using namespace std;
void pocitej(void);

int main()
{
    int volani = 0;
    char volba;
    while (1)
    {
        cout << "K = konec, jine pismeno = pokracovat: ";
        cin >> volba;
        if (volba == 'K')
            break;
        pocitej();
    }
    cout << "Vstup ukoncen.\n";
    return 0;

void pocitej(void)
{
    volani++;
    cout << "Uz me volali " << volani << "x\n";
}
```

Poznámka českého vydavatele: V daném případě je použití break akceptovatelné. Obecně však není používání break příliš podporováno. Jedná se o další výstup z těla cyklu. Dělá to pak problém nástrojům na automatické white-box testování a certifikace kódu.

Tento kód se vůbec nepřeloží. Překladač označí odkaz na proměnnou `volani` ve funkci `pocitej` a bude si stěžovat na nedeklarovaný identifikátor.

Potíž je v tom, že proměnná `volani` platí jen v rámci funkce `main`, ve které jsme ji deklarovali. Už jsme na podobný problém narazili v předchozí kapitole v souvislosti se smyčkou `do while`: s proměnnou můžete pracovat jen v rámci jejího rozsahu platnosti. A rozsah platnosti proměnné udávají složené závorky, ve kterých jste ji deklarovali. Proměnná `volani` proto platí jen v rámci funkce `main`, jde o *místní proměnnou* funkce `main`. Když se místní proměnnou funkce pokusíte použít mimo tuto funkci, dostanete chybu.

Chybu překladu můžeme vyřešit přesunem proměnné `volani` do funkce `pocitej`:

```
#include <iostream>
using namespace std;
void pocitej(void);

int main()
{
    char volba;
    while (1)
    {
```

```
        cout << "K = konec, jine pismeno = pokracovat: ";
        cin >> volba;
        if (volba == 'K')
            break;
        pocitej();
    }
    cout << "Vstup ukoncen.\n";
    return 0;
}

void pocitej(void)
{
    int volani = 0;
    volani++;
    cout << "Uz me volali " << volani << "x\n";
}
```

Takhle vypadá ukázkový vstup a výstup programu:

```
K = konec, jine pismeno = pokracovat: X
Uz me volali 1x
K = konec, jine pismeno = pokracovat: Y
Uz me volali 1x
K = konec, jine pismeno = pokracovat: Z
Uz me volali 1x
K = konec, jine pismeno = pokracovat: K
Vstup ukoncen.
```

Program se sice přeloží, ale funguje jinak, než bychom chtěli. Proměnná `volani` z funkce `pocitej` si „nepamatuje" uloženou hodnotu.

Proměnné, stejně jako lidé, mají určitou životnost. Životní cyklus člověka začíná narozením, životní cyklus proměnné deklarací. Život člověka končí smrtí, život proměnné končí s rozsahem platnosti.

Proměnná `volani` je místní proměnná funkce `pocitej`, protože jsme ji v této funkci deklarovali. Jakožto místní proměnná vznikne s každým voláním funkce `pocitej` a zanikne při návratu z této funkce. Když funkci `pocitej` voláme podruhé, proměnná `volani` už není stejná jako proměnná `volani` z prvního volání funkce – vzniká a zaniká s každým voláním.

Existují dvě možnosti, jak uchovat hodnotu proměnné mezi dvěma voláními funkce: globální a statické proměnné.

Globální proměnné

Kromě místních proměnných existují též proměnné *globální*. Výraz *globální* znamená, že proměnná platí v celém programu. A protože globální proměnné platí v celém programu, jejich život končí až s koncem programu.

Globální proměnné se deklarují nad všemi definicemi funkcí, vedle prototypů. Následující verze programu se od té předchozí liší jen v jednom řádku – deklaraci proměnné `volani` jsme přesunuli z funkce `pocitej` nad funkci `main`, takže je z ní globální proměnná.

```
#include <iostream>
using namespace std;
void pocitej(void);
int volani;
```

```
int main()
{
    char volba;
    volani = 0;
    while (1)
    {
        cout << " K = konec, jine pismeno = pokracovat: ";
        cin >> volba;
        if (volba == 'K')
            break;
        pocitej();
    }
    cout << "Vstup ukoncen.\n";
    return 0;
}

void pocitej(void)
{
    volani++;
    cout << "Uz me volali " << volani << "x\n";
}
```

Vstup a výstup programu vypadají následovně:

```
K = konec, jine pismeno = pokracovat: X
Uz me volali 1x
K = konec, jine pismeno = pokracovat: Y
Uz me volali 2x
K = konec, jine pismeno = pokracovat:  Z
Uz me volali 3x
K = konec, jine pismeno = pokracovat: K
Vstup ukoncen.
```

Přesně tohle jsme chtěli!

Globální proměnné nabízí jednoduché řešení problému s životností proměnných a rozsahem jejich platnosti. Nejspíš proto deklarují začínající programátoři všechny proměnné jako globální, aby k nim měli přístup odkudkoli a kdykoliv. To není dobrý nápad.

Možnost změnit globální proměnnou z libovolné části programu je dvojsečná zbraň. Když můžete proměnnou změnit odkudkoliv, hůř se vám bude například zjišťovat, kde se v proměnné vzala chybná hodnota. Kdyby měla proměnná menší rozsah platnosti, měli byste jednodušší práci, protože byste nemuseli prohlížet tolik kódu.

Globální proměnné komplikují hledání chyb v programech. Proto někteří programátoři a učitelé zacházejí do krajnosti a tvrdí, že všechny globální proměnné jsou zlo. Já bych tak přísný nebyl, ale přesto bych vám globální proměnné doporučoval používat jen v nezbytných případech. Většinou se dá problém vyřešit lepším způsobem, například pomocí statických místních proměnných.

Statické místní proměnné

Až doposud závisela životnost všech proměnných na rozsahu jejich platnosti. Místní proměnné platí ve funkci, ve které jsou deklarované, a tak jejich život končí s návratem z této funkce. Globální proměnné platí v celém programu, a tak končí až s koncem programu.

Statické místní proměnné jsou jiné. Statická místní proměnná má rozsah platnosti běžné místní proměnné, ale životnost globální proměnné. To na první pohled nevypadá příliš intuitivně, a tak si statické proměnné ukážeme na našem příkladu s funkcí `pocitej`.

Statická místní proměnná se deklaruje úplně stejně jako jiné místní proměnné, tedy v rámci funkce – nikoliv nad všemi definicemi funkcí jako globální proměnné. Statické místní proměnné se ale na rozdíl od běžných, takzvaných *automatických* místních proměnných deklarují klíčovým slovem `static` a většinou se hned inicializují. V našem programu tedy deklaraci

```
int volani;
```

nahradíme novou deklarací:

```
static int volani = 0;
```

Takhle vypadá celý program:

```
#include <iostream>
using namespace std;
void pocitej(void);

int main()
{
    char volba;
    while (1)
    {
        cout << "K = konec, jine pismeno = pokracovat: ";
        cin >> volba;
        if (volba == 'K')
            break;
        pocitej();
    }
    cout << "Vstup ukoncen.\n";
    return 0;
}

void pocitej(void)
{
    static int volani = 0;
    volani++;
    cout << "Uz me volali " << volani << "x\n";
}
```

A takhle vypadá vstup a výstup programu:

```
K = konec, jine pismeno = pokracovat: X
Uz me volali 1x
K = konec, jine pismeno = pokracovat: Y
Uz me volali 2x
K = konec, jine pismeno = pokracovat: Z
Uz me volali 3x
K = konec, jine pismeno = pokracovat: K
Vstup ukoncen.
```

Výstup je opět správně. Pojďme se podívat, jak program funguje. Když se funkce `pocitej` zavolá poprvé, deklaruje se proměnná `volani` a inicializuje se na nulu:

```
static int volani = 0;
```

Pak její hodnotu zvýšíme o jedničku a vypíšeme ji:

```
Uz me volali 1x
```

Zatím program funguje úplně stejně, jako kdyby proměnná `volani` byla obyčejná automatická místní proměnná. Rozdíl je v tom, že když funkce `pocitej` skončí, statická proměnná `volani` zůstane naživu.

Když se funkce `pocitej` zavolá podruhé, deklarace a inicializace proměnné `volani` už se neprovádí, protože proměnná už existuje. Navíc ještě pořád obsahuje hodnotu nastavenou během předchozího volání `pocitej`, takže když ji zvýšíme o jedničku a vypíšeme, dostaneme následující výstup:

```
Uz me volali 2x
```

Tak jsme dokázali hodnotu proměnné udržet mezi dvěma voláními funkce, podobně jako s globální proměnnou. Rozdíl je v tom, že statická místní proměnná `volani` platí jen v rámci funkce `pocitej`, tedy nikoliv v celém programu jako globální proměnné. Díky menšímu rozsahu platnosti by se program lépe opravoval, kdyby se v proměnné `volani` měla objevit chybná hodnota.

Předávání parametrů

Funkce `pozdrav` v příkladu ze začátku kapitoly vypíše jednoduše řetězec `Hello, world`. Ke své práci nepotřebuje žádné další informace. Co kdybychom ale chtěli napsat její užitečnější variantu, která by na požádání vypsala libovolný řetězec? Funkce nemá křišťálovou kouli, takže sama neuhodne, jaký řetězec bychom chtěli vypsat. Musíme jí ho nějak předat.

Zkusme si napsat program, který ve funkci `main` načte řetězec od uživatele, uloží ho do proměnné `zprava` a funkce `vypis` se tuto zprávu pokusí vypsat. Jedna možnost je deklarovat proměnnou `zprava` jako globální, aby k ní byl přístup z obou funkcí:

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

void vypis(void);
string zprava;

int main()
{
    cout << "Zadejte retezec: ";
    cin >> zprava;
    vypis();
    return 0;
}

void vypis()
{
    cout << "Zadany text: " << zprava << "\n";
}
```

Poznámka: Pracujeme s řetězci, takže se nezapomeňte odkázat na hlavičkový soubor `string` ze standardní knihovny. Při čtení z `cin` navíc operátor `>>` načte jen řetězec po první mezeru, takže kdybyste zadali například `Jeff Kent`, program vypíše `Jeff`. V následující kapitole vás seznámím s funkcí `getline`, která umí načíst řetězce i s mezerami.

Dialog s programem vypadá takto:

```
Zadejte retezec: Jeff
Zadany text: Jeff
```

Program tedy sice funguje, ale jak jsme si říkali v předchozí části, globální proměnné komplikují hledání chyb v programech. Existuje lepší řešení, a to parametry funkcí.

Už jsme se v této kapitole zmínili o tom, že závorky v hlavičce funkce obsahují její parametry. Parametry jsou informace, které funkce dostává, aby mohla odvést svou práci. Před okamžikem jsme též mluvili o tom, že některé funkce se bez parametrů obejdou – viz například funkci `vypis`, která jen vypíše předem daný řetězec. Když ale chceme funkci upravit tak, aby místo pevně daného řetězce vypsala libovolný zadaný řetězec, musíme jí tento řetězec nějak předat. Právě k takovému předávání informací slouží parametry.

V této kapitole se budeme zabývat dvěma způsoby předávání parametrů, předáváním hodnotou a předáváním odkazem. Ke třetí možnosti – předávání adresou – se vrátíme až v jedenácté kapitole v souvislosti s ukazateli.

Předávání parametrů hodnotou

Následuje další verze programu s funkcí `vypis`, tentokrát s předáváním řetězce v parametru funkce:

```
#include <iostream>
#include <string>
using namespace std;
void vypis(string);

int main()
{
    string zprava;
    cout << "Zadejte retezec: ";
    cin >> zprava;
    vypis(zprava);
    return 0;
}

void vypis(string z)
{
    cout << "Zadany text: " << z << "\n";
}
```

Takhle vypadá vstup a výstup programu:

```
Zadejte retezec: Jeff
Zadany text: Jeff
```

Prototyp a hlavička funkce

V prototypu i hlavičce funkce nám přibyl jeden parametr typu `string`. Zatímco v prototypu se uvádí jen typ parametru (`string`), v hlavičce je kromě datového typu potřeba uvést i název parametru (`string z`). Název parametru můžete uvést i v prototypu:

```
void vypis(string param);
```

Jméno parametru zde ale nehraje žádnou funkční roli. Naproti tomu v hlavičce funkce je název i datový typ parametru povinný. K čemu slouží název parametru, si vysvětlíme vzápětí.

Jak se parametry používají

Funkci `vypis` zavolá následující příkaz:

```
vypis(zprava);
```

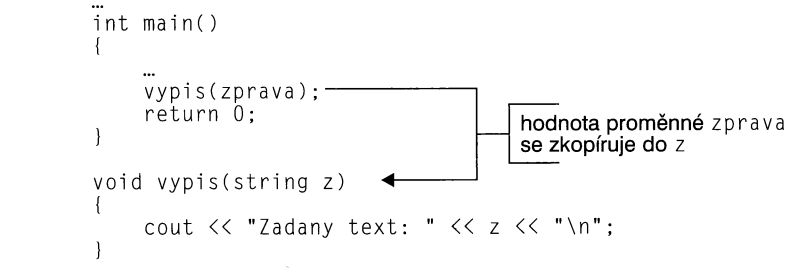
Proměnná `zprava`, do které jsme uložili řetězec od uživatele, se funkci `vypis` předává jako parametr. Její hodnota se uloží do řetězcové proměnné `z`, která je uvedená jako první parametr funkce `vypis`:

```
void vypis(string z)
```

Proměnnou `z` pak funkce `vypis` vypíše na standardní výstup:

```
cout << "Zadany text: " << z << "\n";
```

Na obrázku 9.2 je vidět, jak se hodnota parametru použitého při volání funkce předává do parametru uvedeného v hlavičce funkce a jak se následně zpracuje v těle funkce.



Obrázek 9.2: Předávání parametru

Hlavička funkce musí obsahovat datový typ i jméno parametru, aby se hodnota předávaná z volající funkce (v našem případě hodnota proměnné `zprava`) dala použít ve volané funkci. Kdybyste neuvedli název parametru, neměli byste s předaným parametrem ve funkci `vypis` jak pracovat.

Název parametru v hlavičce funkce může být stejný jako název proměnné předávané při volání funkce:

```
vypis(zprava);
void vypis(string zprava);
```

I v takovém případě je ale proměnná `zprava` z funkce `main` něco jiného než proměnná `zprava` ve funkci `vypis`. Já bych vám doporučoval obě proměnné nazvat jinak, kvůli přehlednosti.

Předávání více parametrů najednou

Náš program si zatím vystačil s jedním parametrem, ale funkce může mít parametrů víc. Následující verze funkce `vypis` má parametry dva, jeden pro křestní jméno a druhý pro příjmení:

```
#include <iostream>
#include <string>
using namespace std;
void vypis(string, string);
```

```
int main()
{
    string jmeno1, jmeno2;
    cout << "Zadejte krestni jmeno: ";
    cin >> jmeno1;
    cout << "Zadejte prijmeni: ";
    cin >> jmeno2;
    vypis(jmeno1, jmeno2);
    return 0;
}

void vypis(string krestni, string prijmeni)
{
    cout << "Jmenujete se " << krestni << " " << prijmeni << ".\n";
}
```

Takto vypadá vstup a výstup programu:

```
Zadejte krestni jmeno: Jeff
Zadejte prijmeni: Kent
Jmenujete se Jeff Kent.
```

Pořadí parametrů předávaných během volání funkce musí odpovídat pořadí parametrů v hlavičce této funkce. V našem případě vypadají volání a hlavička funkce takto:

```
vypis(jmeno1, jmeno2);
void vypis(string krestni, string prijmeni)
```

Při volání funkce se jako první parametr předává proměnná `jmeno1`. Hodnota této proměnné se tedy zkopíruje do proměnné `krestni`, která je uvedená jako první v hlavičce funkce `vypis`. Podobně druhá předávaná proměnná se jmenuje `jmeno2` a zkopíruje se do proměnné `prijmeni`, druhého parametru funkce `vypis`.

Kdybychom parametry při volání funkce přehodili:

```
vypis(jmeno2, jmeno1);
```

...dostali bychom následující výstup:

```
Zadejte krestni jmeno: Jeff
Zadejte prijmeni: Kent
Jmenujete se Kent Jeff.
```

Když jsme v tomto případě nedávali pozor, aby pořadí parametrů při volání funkce přesně odpovídalo pořadí parametrů v její hlavičce, mé jméno se vypsalo pozpátku. Mnohem horší je, když spletete pořadí parametrů různého typu. V následujícím programu je první parametr `jmeno` typu `string` a druhý parametr `vek` typu `int`:

```
#include <iostream>
#include <string>
using namespace std;
void vypis(string, int);

int main()
{
    string jmeno;
    int vek;
    cout << "Zadejte sve jmeno: ";
```

```
    cin >> jmeno;
    cout << "Zadejte svuj vek: ";
    cin >> vek;
    vypis(jmeno, vek);
    return 0;
}

void vypis(string j, int v)
{
    cout << "Jmenujete se " << j << " a je vam " << v << " let.\n";
}
```

Dialog s programem vypadá například takhle (všimněte si, že program nijak nemůže ověřit můj skutečný věk):

```
Zadejte sve jmeno: Jeff Kent
Zadejte svuj vek: 21
Jmenujete se Jeff Kent a je vam 21 let.
```

Volání funkce a její hlavička vypadají následovně:

```
vypis(jmeno, vek);
void vypis(string j, int v)
```

Jako první parametr funkce `vypis` je v hlavičce uvedený řetězec, takže první parametr při volání této funkce musí být bezpodmínečně typu `string`. Podobně druhý parametr je typu `int`, takže i při volání funkce musíte jako druhý předat `int`. Zkuste si parametry přehodit:

```
vypis(vek, jmeno);
```

Tentokrát už program nevypíše chybný výstup jako v předchozím případě. Místo toho vám překladač nahlásí, že nemůže převést první parametr z typu `int` na `string`. Podle prototypu funkce totiž čekal, že první parametr předávaný funkci `vypis` bude `string`, nikoliv `int`.

Předávání parametrů odkazem

Předávání parametrů hodnotou se hodí, když ve volané funkci nepotřebujete měnit jejich hodnoty. Například naše funkce `vypis` hodnoty svých parametrů nemění, jen je vypisuje. Někdy je ale potřeba, aby funkce předané parametry změnila. Podívejte se například na následující ukázku, ve které má funkce `zdvojnasob` vynásobit zadané číslo dvojkou.

```
#include <iostream>
using namespace std;
void zdvojnasob(int);

int main()
{
    int cislo;
    cout << "Zadejte cislo: ";
    cin >> cislo;
    zdvojnasob(cislo);
    cout << "Zpatky v main, cislo po nssobeni: " << cislo << "\n";
    return 0;
}

void zdvojnasob(int x)
{
```

```
    cout << "Nasobene cislo: " << x << "\n";
    x *= 2;
    cout << "Cislo po nasobeni: " << x << "\n";
}
```

Vstup a výstup programu vypadají například takhle:

```
Zadejte cislo: 3
Nasobene cislo: 3
Cislo po nasobeni: 6
Zpatky v main, cislo po nasobeni:: 3
```

Jak je vidět, proměnné `cislo` se zdvojnásobení odpovídajícího parametru `x` ve funkci `zdvojnasob` nedotklo. Jak by také mohlo, když se jako parametr funkce ve skutečnosti předává jen kopie této proměnné. Změna se provedla na kopii a původní proměnná `cislo` zůstala stejná. Je to podobné, jako kdybych vám dal kopii této stránky a vy jste ji roztrhali. Na mém originálu by to nic nezměnilo.

Aby se změna parametru projevila ve funkci `main`, musíte proměnnou *předat odkazem*. Proměnná předávaná odkazem se nekopíruje, příslušný parametr slouží jen jako přezdívka proměnné z funkce `main`. (Stejně jako například James Bond vystupuje pod krycím jménem 007. Je jedno, jestli mluvíte o Jamesovi Bondovi, nebo o agentovi 007. V obou případech mluvíte o stejném člověku.)

Za datový typ parametrů předávaných odkazem se v hlavičce i prototypu funkce píše ampersand (`&`). Ano, tentýž ampersand, který se používá jako adresovací operátor. Zde ho ovšem používáme v trochu jiném kontextu.

Následující program předává zdvojnásobovanou proměnnou odkazem:

```
#include <iostream>
using namespace std;
void zdvojnasob(int&);

int main()
{
    int cislo;
    cout << "Zadejte cislo: ";
    cin >> cislo;
    zdvojnasob(cislo);
    cout << "Zpatky v main, cislo po nasobeni: " << cislo << "\n";
    return 0;
}

void zdvojnasob(int& x)
{
    cout << "Nasobene cislo: " << x << "\n";
    x *= 2;
    cout << "Cislo po nasobeni: " << x << "\n";
}
```

Vstup a výstup vypadá následovně:

```
Zadejte cislo: 3
Nasobene cislo: 3
Cislo po nasobeni: 6
Zpatky v main, cislo po nasobeni: 6
```

Změny oproti předchozí verzi jsou pouze dvě, kvůli předávání parametru odkazem se změnil prototyp a hlavička funkce `zdvojnasob`. Původní varianta s parametrem předávaným hodnotou vypadala takto:

```
void zdvojnasob(int);
void zdvojnasob(int x)
```

Ve verzi s předáváním parametru odkazem přibyly ampersandy za datovými typy:

```
void zdvojnasob(int&);
void zdvojnasob(int& x)
```

Samotné volání funkce vypadá v obou případech stejně, zde žádný ampersand nepřibyl. Způsob předávání parametrů závisí pouze na prototypu funkce.

Pokud má funkce větší počet parametrů, můžete si libovolně vybrat, které z nich předáte hodnotou a které odkazem. Pokud parametr potřebujete měnit, předejte ho odkazem. Pokud ne, předejte ho hodnotou.

Poznámka: Mezi předáváním hodnotou a odkazem je ještě jeden rozdíl. Hodnotou se kromě proměnných dají předávat i konkrétní hodnoty a konstanty (ke kterým se dostaneme v následující kapitole). Odkazem se dají předávat pouze proměnné.

Například následující program obsahuje funkci `secti`, která má tři parametry. První dva jsou čísla k sečtení a předávají se hodnotou. Třetí parametr je proměnná pro výsledek sčítání, a protože ji chceme ve volané funkci změnit, musí se předávat odkazem.

```
#include <iostream>
using namespace std;
void secti(int, int, int&);

int main()
{
    int a, b, soucet;
    cout << "Zadejte prvni cislo: ";
    cin >> a;
    cout << "Zadejte druhe cislo: ";
    cin >> b;
    secti(a, b, soucet);
    cout << a << " + " << b << " = " << soucet << "\n";
    return 0;
}

void secti(int x, int y, int& z)
{
    z = x + y;
}
```

Vstup a výstup vypadají následovně:

```
Zadejte prvni cislo: 3
Zadejte druhe cislo: 6
3 + 6 = 9
```

Návratová hodnota funkce

K předávání hodnot volané funkci slouží parametry, k předávání informací z volané funkce zpět do volající slouží *návratová hodnota funkce*. Funkce `secti` z předchozího příkladu má tři parametry: dva zadávají čísla k sečtení, třetím se předává jejich součet. Následující verze programu nahrazuje třetí parametr návratovou hodnotou funkce:

```
#include <iostream>
using namespace std;
int secti(int, int);

int main()
{
    int a, b, soucet;
    cout << "Zadejte prvni cislo: ";
    cin >> a;
    cout << "Zadejte druhe cislo: ";
    cin >> b;
    soucet = secti(a, b);
    cout << a << " + " << b << " = " << soucet << "\n";
    return 0;
}

int secti(int x, int y)
{
    return x + y;
}
```

Vstup a výstup programu se neliší od předchozí verze:

```
Zadejte prvni cislo: 3
Zadejte druhe cislo: 6
3 + 6 = 9
```

Pokud funkce vrací nějakou hodnotu (v našem případě `int`), datový typ této hodnoty se píše před název funkce, v prototypu i hlavičce:

```
int secti(int, int);
int secti(int x, int y)
```

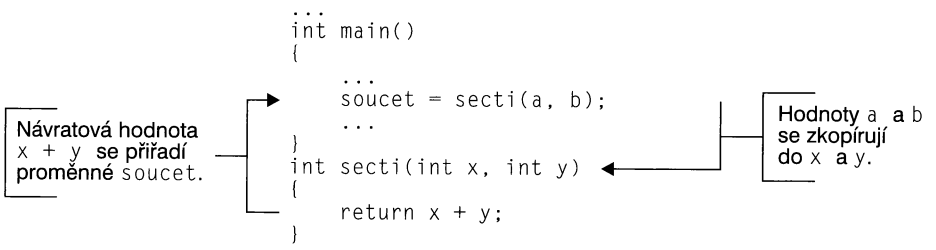
Ve volající funkci `main` pak návratovou hodnotu používáme na pravé straně operátoru přiřazení. Na levé straně tohoto přiřazení je proměnná, jejíž typ odpovídá typu návratové hodnoty funkce. Ve výsledku se návratová hodnota funkce do této proměnné uloží:

```
soucet = secti(a, b);
```

V těle volané funkce je příkaz `return` následovaný výrazem, který typem odpovídá typu návratové hodnoty funkce (`int`). Výraz za příkazem `return` se vyhodnotí a použije jako návratová hodnota funkce:

```
return x + y;
```

Tuto hodnotu pak ve funkci main přiřadíme proměnné `soucet`; celý proces je znázorněný na obrázku 9.3.



Obrázek 9.3: Funkce vrací hodnotu

Takové zpracování návratové hodnoty přiřazením do proměnné je běžné, ale není to jediná možnost, jak s návratovou hodnotou něco udělat. Například v našem programu jsme se mohli obejít bez proměnné `soucet`. Momentálně vypisujeme součet následujícími dvěma řádky:

```
soucet = secti(a, b);
cout << a << " + " << b << " = " << soucet << "\n";
```

Stejně dobře bychom ale mohli návratovou hodnotu funkce zobrazit rovnou:

```
cout << a << " + " << b << " = " << secti(a, b) << "\n";
```

Jediný rozdíl je v tom, že vypočtenou hodnotu pak nemůžeme použít v dalších příkazech, protože jsme si ji nikam neuložili. V našem programu to nevadí, protože už ji nepotřebujeme. Pokud ale budete návratovou hodnotu funkce používat vícekrát, bývá lepší ji uložit do proměnné.

Zatímco parametrů můžete funkci předat víc najednou, návratová hodnota je vždy jen jedna. Větší počet hodnot se dá snadno vrátit v poli, struktuře nebo třídě, ke kterým se dostaneme v následujících kapitolách.

Shrnutí

Funkce je skupina příkazů, které dohromady plní nějaký úkol. V principu sice každý program vystačí s funkcí `main`, ale čím složitější a důmyslnější programy budete psát, tím důležitější je rozdělit kód do samostatných funkcí, aby se vám program lépe psal, četl a opravoval.

Každou funkci (kromě `main`) musíte nejprve definovat a pak zavolat. Definice funkce se skládá z hlavičky a těla. V hlavičce je uvedený návratový typ funkce, jméno funkce a seznam parametrů. Za hlavičkou následuje tělo funkce uzavřené do složených závorek. Tělo obsahuje libovolný počet příkazů a většinou končí příkazem `return`. Pokud funkce není definovaná nad všemi funkcemi, které ji volají, musíte ji předem prototypovat.

Pokud program neobsahuje jinou funkci než `main`, všechny proměnné deklarované na začátku této funkce jsou dostupné z celého programu. Jakmile začneme kód dělit do funkcí, objeví se problémy s životností proměnných a jejich rozsahem platnosti. Rozsah platnosti proměnné říká, kde všude se v kódu můžete na proměnnou odkazovat. Životnost proměnné říká, kdy proměnná vznikne a zanikne. Místní proměnné mají rozsah platnosti i životnost omezené na funkci, ve které byly deklarovány. Naproti tomu globální proměnné platí v celém programu a zanikají až na jeho konci. Pak existují ještě statické místní proměnné, které jsou stejně jako místní proměnné omezené na funkci, ve které byly deklarovány, ale stejně jako globální proměnné zanikají až na konci programu.

Informace se funkcím předávají pomocí parametrů, a to odkazem nebo hodnotou. Předávání parametrů hodnotou se používá, pokud nepotřebujete, aby se změny parametru promítly do volající funkce. Pokud chcete, aby změna parametru způsobila změnu odpovídající proměnné ve volající funkci, použijte parametr předaný odkazem. Pořadí a datový typ parametrů uvedených v prototypu funkce musí odpovídat pořadí a typu parametrů v její hlavičce a musíte je dodržet i při samotném volání funkce.

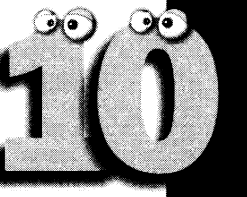
Zatímco parametry slouží k předávání informací volané funkci, k přenosu informací zpět funkci volající slouží návratová hodnota funkce. Parametrů může mít každá funkce víc, návratová hodnota je jen jedna.

Až doposud jsme pracovali s proměnnými, které vždy obsahují pouze jednu hodnotu. V následující kapitole se budeme věnovat proměnným, které mohou obsahovat větší počet hodnot najednou.

Test

1. Jaký je rozdíl mezi životností proměnné a jejím rozsahem platnosti?
2. Musí se každá funkce s výjimkou `main` prototypovat?
3. Musí mít každá funkce alespoň jeden parametr?
4. Může mít funkce víc než jeden parametr?
5. Změní se ve funkci `main` hodnota proměnné, když tuto proměnnou předáme hodnotou nějaké funkci a ta změní hodnotu odpovídajícího parametru?
6. Změní se ve funkci `main` hodnota proměnné, když tuto proměnnou předáme odkazem nějaké funkci a ta změní hodnotu odpovídajícího parametru?
7. Musí mít každá funkce nějakou návratovou hodnotu?
8. Může mít funkce více než jednu návratovou hodnotu?
9. Může být funkce bez parametrů i bez návratové hodnoty?
10. Může mít funkce zároveň návratovou hodnotu i parametry?

Pole



Do proměnných, se kterými jsme až doposud pracovali, se vejde jen jedna hodnota. Když deklarujete celočíselnou proměnnou `pocetBodu` určenou pro výsledky zkoušky, vejdou se do ní jen výsledky jedné zkoušky. To není problém, dokud potřebujete uložit výsledky jednoho studenta v jedné zkoušce. Co když ale bude studentů nebo zkoušek víc? Kdybyste další výsledky opět uložili do proměnné `pocetBodu`, přijdete o její předchozí hodnotu. Takže kdybyste chtěli uložit výsledky řekněme jednoho sta zkoušek, mohli byste začít nějak takhle:

```
int pocetBodu1;
int pocetBodu2;
int pocetBodu3;
int pocetBodu4;
int pocetBodu5;
int pocetBodu6;
int pocetBodu7;
int pocetBodu8;
int pocetBodu9;
...
int pocetBodu100;
```

Fuj! To je poměrně dost práce, že? Nebylo by jednodušší deklarovat jednu proměnnou, která by zvládla uložit všech sto hodnot najednou? Například takhle:

```
int pocetBodu[100];
```

Dobrá zpráva zní, že přesně takhle to můžete udělat; této proměnné se říká *pole*. Do jedné proměnné typu pole se dá uložit víc hodnot najednou. Jednotlivé hodnoty jsou číslované, první hodnota má číslo nula a každá další hodnota má číslo o jedničku větší.

Uložení sta hodnot do jednoho pole o velikosti 100 prvků má oproti řešení se stovkou proměnných řadu výhod. Nemusíte psát tolik kódu a v programu s jednou proměnnou se vyznáte mnohem lépe než v programu se stovkou proměnných. Mnohem důležitější ale je, že se jednotlivé prvky pole dají projít pomocí smyčky, což se stovkou samostatných proměnných udělat nemůžete.

Deklarování pole

Pole je obyčejná proměnná, takže se jako každá jiná proměnná musí před použitím deklarovat. Deklarace pole vypadá skoro stejně jako deklarace celočíselné, znakové nebo jiné proměnné. Celočíselnou proměnnou `pocetBodu` byste deklarovali takhle:

```
int pocetBodu;
```

A pole pro tři celočíselné výsledky zkoušky by vypadalo takto:

```
int pocetBodu[3];
```

Analogicky se dají deklarovat pole desetinných čísel, znaků nebo řetězců:

```
float prumery[5];
char znamky[7];
string jmena[6];
```

Pole tedy můžete udělat z libovolného datového typu, ale hodnoty v rámci pole musí mít datový typ stejný. Nemůžete mít pole, ve kterém bude jeden prvek typu `float`, jiný typu `string`, dalších pár typu `int` a podobně.

Deklarace jednoduché proměnné i pole začíná názvem datového typu, za kterým je název proměnné a středník. Jediný rozdíl mezi polem a obyčejnou proměnnou je v tom, že pole má za názvem proměnné ještě číslo v hranatých závorkách. Toto číslo určuje velikost pole.

Poznámka: Za chvíli se budeme věnovat inicializaci polí a při té příležitosti se dozvíte o jedné výjimce: Pokud pole zároveň s deklarací inicializujete, můžete jeho velikost z hranatých závorek vynechat.

Velikost pole se zadává, aby překladač věděl, kolik má pro pole vyhradit místa. Ve třetí kapitole jsme si říkali, že deklarace proměnné vyhradí v paměti místo potřebné pro její uložení, a že velikost proměnných závisí na operačním systému a překladači. Pokud například typ `int` ve vašem operačním systému a překladači zabere čtyři bajty, deklarace `int cislo` vyhradí v paměti čtyři bajty. Kdybyste místo jednoduché proměnné deklarovali pole tří čísel, `int cislo[3]`, v paměti by se vyhradilo 4×3=12 bajtů.

Tip: Velikost pole byste si měli důkladně rozmyslet předem, protože za běhu už se nedá měnit. Pokud program uprostřed svého běhu zjistí, že je pole moc malé, nebo naopak zbytečně velké, už s tím nic neudělá. Někdy je velikost pole zjevná, například pole pro názvy dní v týdnu bude mít sedm prvků. Jindy se velikost pole odhaduje špatně. V takových případech je lepší velikost pole přehnat, než ho deklarovat příliš malé – je lepší trochu plýtvat pamětí, než nemít místo na uložení nových dat.

Konstanty

Velikost všech polí jsme doposud zadávali *literálem*, tedy doslovnou hodnotou. Například `3` je literál, protože už se dál neinterpretuje. Hodnotou výrazu `3` nemůže být nic jiného než číslo tři, ať uděláte cokoliv. Proto se trojka dá použít pro deklaraci velikosti pole:

```
int main()
{
    int pocetBodu[3];
    return 0;
}
```

Proměnná se pro deklaraci velikosti pole použít nedá, viz následující program:

```
#include <iostream>
using namespace std;
int main()
{
    int pocet;
    cout << "Zadejte pocet testu: ";
    cin >> pocet;
    int vysledky[pocet];
    return 0;
}
```

Když se tento zdrojový kód pokusíte přeložit, překladač ohlásí chybu. Označí deklaraci pole `vysledky` a bude si stěžovat, že očekával konstantní výraz.

Poznámka: Velikost pole může být určená proměnnou, ale pak se pole musí alokovat dynamicky. K tomuto tématu se vrátíme v následující kapitole.

Výraz *konstanta* je pro nás nový. Konstanta je jméno, které v celém programu zastupuje jednu jedinou neměnnou hodnotu. Tuto hodnotu si můžete určit, ale jakmile ji jednou stanovíte, v průběhu programu už se nemění. Proto jsou konstanty určitým protipólem proměnných – proměnná je jméno, které může v průběhu programu zastupovat různé hodnoty.

Poznámka: Literály a konstanty mají společné to, že se jejich hodnota v průběhu programu nemění. Přesto nejsou totéž – konstanta je jméno, které zastupuje nějakou hodnotu, zatímco literál je přímo konkrétní hodnota sama.

Konstanta se dá použít v deklaraci pole místo literálu. Například následující program deklaruje pomocí konstanty tříprvkové pole:

```
int main()
{
    const int POCET = 3;
    int vysledky[POCET];
    return 0;
}
```

Zde jsme definovali konstantu `pocet`, která v celém programu zastupuje hodnotu `3`. Konstanty se deklarují podobně jako proměnné, musíte uvést datový typ (v našem případě `int`) a název konstanty (v našem případě `pocet`). Mezi deklarací proměnné a konstanty jsou ale dva rozdíly. Za prvé, deklarace konstanty musí začínat klíčovým slovem `const`. Podle něj překladač pozná, že deklarujete konstantu, a nikoliv proměnnou.

A za druhé končí deklarace konstanty přiřazením hodnoty. I proměnné můžete v rámci deklarace přiřadit hodnotu, o této takzvané inicializaci jsme se bavili ve třetí kapitole. U konstant je ale inicializace povinná. Když ji vypustíte, program se nepřeloží (překladač si bude stěžovat na to, že chybí inicializace konstanty). Je to logické – pokud se hodnota konstanty nedá měnit, musíte ji nastavit už v rámci deklarace.

Poznámka: Při deklaraci konstanty se v paměti vyhradí místo na její hodnotu, stejně jako při deklaraci proměnné. Rozdíl je v tom, že u konstanty se hodnota uložená v paměti nedá měnit.

Je zvykem psát název konstanty s velkými písmeny. Kód je pak přehlednější a čitelnější.

Že se hodnota jednou deklarované konstanty nedá měnit, ukazuje následující program:

```
#include <iostream>
using namespace std;
int main()
{
    const int POCET = 3;
    cout << "Zadejte počet testu: ";
    cin >> POCET;
    int vysledky[POCET];
    return 0;
}
```

Program se nepřeloží. Překladač označí příkaz `cin >> pocet` a bude si stěžovat, že operátor `>>` nemůže mít na pravé straně konstantu. Jinými slovy říká, že jednou deklarované konstantě už nemůžete nic přiřazovat. Následující program to zkouší jinak. Vstup od uživatele uloží do proměnné (zatím je všechno v pořádku) a pak se tuto proměnnou pokusí přiřadit konstantě (chyba):

```
#include <iostream>
using namespace std;
int main()
{
    const int POCET = 3;
    int cislo;
    cout << "Zadejte pocet testu: ";
    cin >> cislo;
    POCET = cislo;
    int vysledky[POCET];
    return 0;
}
```

Výsledkem je opět chyba v překladu. Překladač označí pokus o přiřazení hodnoty konstantě (`pocet = cislo`) a tentokrát si postěžuje, že na levé straně přiřazení je konstanta. Jinými slovy zase říká, že konstantě nemůžete přiřazovat.

K čemu nám je dobré, že můžeme literál v deklaraci pole nahradit konstantou? Výhody pojmenovaných konstant plynou především z toho, že se velikost deklarovaného pole v programu většinou objeví vícekrát, například při načítání hodnot pole nebo jejich výstupu. Zároveň se běžně stává, že musíte velikost pole kvůli potřebám programu změnit, obvykle zvýšit. Kdybych se například já jako učitel rozhodl, že budeme psát místo tří testů pět, musel bych změnit velikost pole `vysledky` ze tří na pět. Kdybych byl všude ve zdrojovém kódu používal literál `3`, musel bych teď hledat všechny trojky a přepisovat je na pětky. To jednak zabere dost času a jednak hrozí, že bych některou trojku přehlédl. Kdybych ale program původně napsal s konstantou, například `const int POCET = 3`, stačí přepsat tuto jedinou trojku a je hotovo.

Možná si říkáte: „Moment moment, vždyť jsme si zrovna říkali, že se velikost pole nedá změnit!" Ano, za běhu programu se velikost pole měnit nedá. Ve zdrojovém kódu ji ale můžete měnit podle libosti, pak stačí program znovu přeložit.

Konstanty se nepoužívají jen pro určení velikosti polí, mají řadu dalších využití. Průběžně se k nim budeme vracet v dalším textu.

Indexování

Celé pole má jen jedno jméno. My se ale potřebujeme odkazovat na jeho jednotlivé prvky. Pro tyto účely slouží pozice prvku v poli, takzvaný index. Indexování začíná od nuly, takže první prvek pole je bez výjimky na indexu nula a poslední prvek má (opět bez výjimky) index rovný počtu prvků minus jedna.

Indexování polí od nuly namísto od jedničky možná vypadá na první pohled nelogicky, ale rychle se vysvětlí, když se podíváme na způsob uložení pole v paměti. Jednotlivé prvky pole jsou v paměti uložené za sebou, počínaje určitou základní adresou (kterou si můžete snadno zjistit adresovacím operátorem `&`). Jelikož známe velikost každého prvku pole, můžeme si snadno odvodit adresu libovolného z prvků. Pokud chceme například adresu desátého prvku, stačí k základní adrese přičíst desetkrát velikost prvku pole. Dobře je to vidět na obrázku 10.1, který ukazuje pole se třemi prvky typu `int`.

int vysledky[3];

Index	0	1	2
Posun	0×4=0	1×4=4	2×4=8
Adresa	100	104	108

(Předpokládáme, že datový typ `int` zabírá čtyři bajty.)

Obrázek 10.1: Indexy v poli se třemi prvky typu int

První prvek pole leží přímo na základní adrese celého pole (`&vysledky`). Když chceme získat adresu kteréhokoliv prvku pole, stačí jeho index vynásobit velikostí datového typu prvku pole a výsledek přičíst k základní adrese. Například druhý prvek pole z obrázku 10.1 má adresu 100+2×4, kde 100 je základní adresa pole, 2 index prvku a 4 je `sizeof(int)`.

Kdybychom začínali indexovat od jedničky namísto od nuly, už bychom nemohli index prvku jednoduše násobit velikostí datového typu. Například adresa druhého prvku by se počítala jako (2–1)×4 plus základní adresa pole. Pro počítač je rychlejší a jednodušší první verze s indexováním od nuly, a tak se tato verze dostala i do většiny počítačových jazyků.

Poznámka: K adresování proměnných se ještě podrobně vrátíme v následující kapitole věnované ukazatelům.

Jelikož má první prvek pole index nula, poslední prvek musí mít vždy index o jedničku menší, než je počet prvků pole. (Když odpočítáte tři čísla od jedničky včetně, dojdete ke trojce. Když stejným způsobem odpočítáte tři čísla od nuly, dojdete ke dvojce.)

Upozornění: Začínající programátoři se často pletou a myslí si, že index posledního prvku pole je rovný celkovému počtu prvků v poli. Jak za chvilku uvidíte, takové přehmaty vedou ke špatným výsledkům nebo rovnou způsobí ukončení programu s chybou, záleží na překladači. Ani jedna varianta není příliš žádoucí.

V této chvíli jsme prvkům pole ještě nepřiřadili žádnou hodnotu. Proto budou nejspíš obsahovat nějaká podivná čísla, například –858 993 460. Jak už jsme si říkali ve třetí kapitole, tyto hodnoty zůstaly v paměti po předchozích programech a většinou moc nedávají smysl.

Poznámka: Pokud je pole deklarované globálně, všechny jeho prvky se inicializují na výchozí hodnotu, číselné proměnné na nulu a znakové proměnné na znak null. Už jsme si ale říkali, že globálním proměnným je lepší se vyhnout.

Inicializace

Jak už jsme si říkali ve třetí kapitole, inicializovat proměnnou znamená nastavit její hodnotu už v rámci deklarace. Nastavení hodnot pole se budeme podrobně věnovat za chvíli, teď nás bude na okamžik zajímat výhradně inicializace polí. Pole se dá inicializovat dvěma způsoby. Záleží na tom, jestli jeho velikost uvedete explicitně do hranatých závorek za jeho název, nebo jestli ji necháte implicitně zjistit překladačem.

Explicitní velikost pole

Inicializace pole s explicitním uvedením velikosti vypadá následovně:

```
int vysledky[3] = {  74, 87, 91 };
```

```
float spotreba[4] = {  7.8, 11.2, 5.6, 9.3 };
char znamky[5] = {  'A', 'B', 'C', 'D', 'E', 'F' };
string dny[7] = {  "pondeli", "utery", "streda", "ctvrtek",
                   "patek", "sobota", "nedele" };
```

Pravá strana inicializace nezávisí na tom, jestli je velikost pole daná implicitně, nebo explicitně. Za názvem pole (například `vysledky[3]`) je vždy operátor přiřazení a složené závorky s hodnotami jednotlivých prvků pole. Například následující příkaz uloží do prvního prvku pole (`vysledky[0]`) hodnotu 74, do druhého prvku hodnotu 87 a do třetího prvku číslo 91:

```
int vysledky[3] = {  74, 87, 91 };
```

Počet prvků zadaných na pravé straně přiřazení nesmí překročit velikost pole zadanou v hranatých závorkách. Následující příkaz se nepřeloží, překladač si bude stěžovat na „příliš mnoho inicializátorů“:

```
float spotreba[4] = {  7.8, 11.2, 5.6, 9.3, 4.8 }; // chyba
```

Naopak nemusíte inicializovat úplně všechny prvky pole, počet hodnot na pravé straně přiřazení může být menší než udávaná velikost pole:

```
float spotreba[4] = {  7.8, 11.2, 5.6 };
```

Neinicializované prvky pole budou mít výchozí hodnotu, která závisí na jejich typu. Celočíselné proměnné mají výchozí hodnotu 0, výchozí desetinná hodnota je 0,0 a výchozí znaková hodnota je znak null, `'\0'`.

Poznámka: Ke znaku null se podrobněji vrátíme za chvíli, až budeme mluvit o inicializaci znakových polí.

Když necháte neinicializovaný nějaký prvek pole, musíte bez inicializace nechat i všechny následující prvky pole. Inicializace se nedá dělat napřeskáčku, následující kód se vůbec nepřeloží:

```
float spotreba[4] = {  7.8, 11.2, , 5.6 }; // chyba
```

Implicitní velikost pole

Pokud do hranatých závorek za název pole neuvedete velikost pole, překladač si ji doplní automaticky podle počtu prvků na pravé straně přiřazení:

```
int vysledky[] = {  74, 87, 91 };
float spotreba[] = {  7.8, 11.2, 5.6, 9.3 };
char znamky[] = {  'A', 'B', 'C', 'D', 'E', 'F' };
string dny[] = {  "pondeli", "utery", "streda", "ctvrtek",
                  "patek", "sobota", "nedele" };
```

Na prvním řádku se tedy v paměti vyhradí místo pro tři celá čísla, druhý řádek v paměti vyhradí místo pro čtyři desetinná čísla, třetí řádek vytvoří místo na šest znaků a konečně čtvrtý řádek udělá v paměti místo pro sedm řetězců.

Velikost uvedená v hranatých závorkách má přednost, takže následující deklarace vyhradí v paměti místo pro pět celých čísel, ačkoliv na pravé straně inicializujeme jen tři:

```
int vysledky[5] = {  74, 87, 91 };
```

Jak už jsme si říkali před chvilkou, čtvrtý a pátý prvek pole budou inicializovány na výchozí hodnotu svého typu, v tomto případě na nulu. Velikost pole musí být určená alespoň jedním z obou způsobů, následující deklarace pole je neplatná:

```
int vysledky[];
```

Překladač vám řekne, že nezná velikost pole. To je samozřejmě problém, protože pro pole o neznámé velikosti nemůžete vyhradit paměť.

Inicializace znakových polí

Jak jste viděli v předcházejícím textu, znaková pole se dají inicializovat stejně jako pole jiných datových typů, například `int[]` nebo `float[]`. Znaková pole se ale od jiných polí v některých ohledech výrazně liší, a my se teď na první z těchto rozdílů podíváme.

Následující dva způsoby inicializace znakového pole na hodnotu mého křestního jména vypadají úplně jinak, ale ve výsledku dělají totéž:

```
char jmeno[] = {'J', 'e', 'f', 'f', '\0'};
char jmeno[] = "Jeff";
```

Programátoři dávají přednost druhé variantě, protože je zkrátka jednodušší. Konstrukce `'\0'` je znak s ASCII kódem nula, takzvaný *nulový znak* neboli *null*. (Všimněte si, že za zpětným lomítkem je nula, nikoliv velké písmeno ó.) My jsme se zvláštními znaky zabývali už ve druhé kapitole a běžně používáme znak `\n`, konec řádky. Konstrukce `\n` se v řetězci nebere doslova – `cout` podle zpětného lomítka pozná, že narazil na speciální znak.

Nulový znak označuje konec znakového pole. Například následující program vypíše – přesně podle všech očekávání – řetězec `Jeff`:

```
#include <iostream>
using namespace std;
int main()
{
    char jmeno[] = {'J', 'e', 'f', 'f', '\0'};
    cout << jmeno << "\n";
    return 0;
}
```

Výsledek je stejný, jako kdybychom pole inicializovali příkazem `char jmeno[] = "Jeff"`. Naproti tomu následující program vypíše `Jeff¦+*?:`

```
#include <iostream>
using namespace std;
int main()
{
    char jmeno[] = {'J', 'e', 'f', 'f'};
    cout << jmeno << "\n";
    return 0;
}
```

Podivným znakům za řetězcem `Jeff` (které mohou na vašem počítači vypadat jinak) se někdy říká „binární smetí". Co přesně se stalo? Když `cout` dostane k výpisu pole znaků, nezná jeho velikost, a tak vypisuje jeden znak za druhým, dokud nenarazí na nulový znak. Pokud nulový znak v poli není, `cout` neskončí posledním znakem pole, ale pokračuje ve výpisu paměti za tímto polem,

dokud časem nenarazí na nulový znak. Vypsané binární smetí jsou tedy znaky, které odpovídají hodnotám uloženým v paměti za vypisovaným polem.

To neznamená, že byste nutně veškerá znaková pole měli končit nulovým znakem. Pokud jsou jednotlivé prvky pole samostatné (například pokud jde o známky), asi nebudete nikdy vypisovat celé pole najednou a tím pádem není nutné pole končit nulovým znakem. Pokud ale prvky pole patří dohromady, například pokud jde o jednotlivé znaky něčího jména, pole by mělo být uzavřené nulovým znakem. Inicializace příkazem `char jmeno[] = "Jeff"` to udělá za vás, jako pátý prvek pole přidá `\0`.

Všimněte si, jak se syntaxe `char jmeno[] = "Jeff"` podobá inicializaci řetězce:

```
char jmeno[] = "Jeff";
string jmeno = "Jeff";
```

Není to náhoda – znaková pole zakončená nulovým znakem se totiž jako řetězce používají v jazyce C, se kterým má C++ hodně společného. C++ už má pro řetězce samostatný datový typ `string`, který žádné zakončení nulovým znakem nevyžaduje. Jelikož se ale řetězce z jazyka C (tedy pole znaků zakončená nulovým znakem) celkem běžně používají i v C++, chová se k nim C++ jinak než k ostatním polím. Na konkrétní příklady narazíme za chvíli.

Konstantní pole

Pole může být konstantní, například následující pole obsahuje počty dní v měsících nepřestupného roku:

```
const short dni[] = {31, 28, 31, 30, 31, 30,
                     31, 31, 30, 31, 30, 31};
```

V tomto případě je rozumné označit pole jako konstantní, protože počet dní v měsíci už se měnit nebude. Stejně jako obyčejnou konstantu musíte i konstantní pole inicializovat v rámci deklarace, protože později už se jeho hodnota měnit nedá.

Kdy pole inicializovat

Pole můžete inicializovat celé, částečně anebo vůbec. Nejčastěji se inicializují pole, u kterých znáte alespoň některé hodnoty předem. Inicializace ale není omezená jen na tento případ – běžně se inicializace používá k nastavení prvků pole na nějakou rozumnou výchozí hodnotu. Například prvky pole `vysledky` bychom mohli inicializovat na hodnotu `-1`, aby bylo jasné, že jsme do nich ještě žádný výsledek zkoušky neukládali. Číslo –1 je v tomto případě lepší volba než nula, protože nula by na rozdíl od minus jedné mohla klidně označovat výsledek testu.

U větších velikostí polí může být inicializace dost pracná. Někdy neznáte hodnoty pole předem (například u našeho příkladu se známkami) a rozhodnete se, že na výchozí hodnotu nemá pole smysl inicializovat. A i když pole inicializujete, většinou byste chtěli hodnoty některých prvků změnit. Proto potřebujete vědět, jak se polím přiřazuje a jak se jejich hodnota dá vypsat.

Nastavení a zobrazení polí

Následující program ukazuje, jak se prvky pole dají jeden po druhém nastavit a vypsat. Přiřazení začíná prvním indexem, tedy nulou, a končí posledním indexem neboli dvojkou, která je o jedničku menší než počet prvků pole.

```
#include <iostream>
using namespace std;
int main()
{
    int vysledky[3];
    cout << "Zadejte vysledek #1: ";
    cin >> vysledky[0];
    cout << "Zadejte vysledek #2: ";
    cin >> vysledky[1];
    cout << "Zadejte vysledek #3: ";
    cin >> vysledky[2];
    cout << "Vysledek #1: " << vysledky[0] << "\n";
    cout << "Vysledek #2: " << vysledky[1] << "\n";
    cout << "Vysledek #3: " << vysledky[2] << "\n";
    return 0;
}
```

Vstup a výstup programu vypadají například takhle:

```
Zadejte vysledek #1: 77
Zadejte vysledek #2: 91
Zadejte vysledek #3: 84
Vysledek #1: 77
Vysledek #2: 91
Vysledek #3: 84
```

Zpracování hodnot po jedné ale nenabízí příliš výhod oproti obyčejnému řešení se třemi samostatnými proměnnými:

```
#include <iostream>
using namespace std;
int main()
{
    int vysledek1, vysledek2, vysledek3;
    cout << "Zadejte vysledek #1: ";
    cin >> vysledek1;
    cout << "Zadejte vysledek #2: ";
    cin >> vysledek2;
    cout << "Zadejte vysledek #3: ";
    cin >> vysledek3;
    cout << "Vysledek #1: " << vysledek1 << "\n";
    cout << "Vysledek #2: " << vysledek2 << "\n";
    cout << "Vysledek #3: " << vysledek3 << "\n";
    return 0;
}
```

Hlavní výhoda polí se totiž projeví až při použití smyčky:

```
#include <iostream>
using namespace std;
int main()
{
    int vysledky[3];
    for (int i=0; i<3; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> vysledky[i];
    }
    for (int i=0; i<3; i++)
```

```
        cout << "Vysledek #" << i+1 << ": "
             << vysledky[i] << "\n";
    return 0;
}
```

A ještě o něco je lepší zadat velikost pole konstantou místo literálem:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX];
    for (int i=0; i<MAX; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> vysledky[i];
    }
    for (int i=0; i<MAX; i++)
        cout << " Vysledek #" << i+1 << ": "
             << vysledky[i] << "\n";
    return 0;
}
```

Na tomto příkladu jsou vidět výhody konstant oproti literálům. Řekněme, že bych tento program pro evidenci známek napsal v době, kdy jsem dával tři písemky za semestr. Co kdybych později začal psát pět písemek za semestr? S konstantou mi stačí udělat jedinou změnu, přepsat hodnotu `MAX` ze tří na pět. Kdybych byl velikost pole zadal literálem, musel bych příslušné číslo hledat v celém kódu – jednou je použité v deklaraci pole a dvakrát ve smyčkách. V tomto případě jde jen o tři změny, ale u složitějšího programu by počet změn mohl být mnohem vyšší. To jednak může zabrat hodně času a jednak je tu možnost, že bych některou trojku přehlédl.

Konstantu `MAX` máme deklarovanou globálně. V předchozí kapitole jsme si říkali, že globálním proměnným je dobré se vyhýbat, ale na konstanty se toto doporučení nevztahuje. Hlavní argument proti globálním proměnným totiž zní, že se jejich hodnota dá měnit odkudkoliv z programu, což komplikuje hledání chyb (například když se v proměnné ocitne divná hodnota). Hodnota konstant se měnit nedá, a tak globální konstanty ničemu nevadí; na rozdíl od globálních proměnných jsou celkem běžné.

Ať už ale máte velikost pole definovanou literálem nebo konstantou, musíte si dát pozor, abyste ji nepřekročili. Tuto běžnou programátorskou chybu ukazuje následující program:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX];
    for (int i=0; i<=MAX; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> vysledky[i];
    }
    for (int i=0; i<=MAX; i++)
        cout << "Vysledek #" << i+1 << ": "
             << vysledky[i] << "\n";
    return 0;
```

```
}
```

Tento program je úplně stejný jako ten předchozí, jen relační operátory v podmínkách cyklů se změnily z < na <=. Kvůli této změně se v poslední iteraci smyčky pokoušíme o přístup ke čtvrtému prvku pole (tj. prvku s indexem 3), `vysledky[3]`. Žádný takový ovšem samozřejmě není, poslední třetí prvek pole má index 2. Výsledek záleží na překladači a operačním systému, můžete čekat cokoliv od divného výstupu přes ukončení programu s chybou až po zaseknutí počítače.

Pole, cin a cout

Do pole znaků můžete hodnoty zadávat stejným způsobem, kterým jsme před chvilkou ukládali hodnoty do pole celých čísel:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    char znamky[MAX];
    for (int i = 0; i < MAX; i++)
    {
        cout << "Zadejte " << i+1 << ". znamku: ";
        cin >> znamky[i];
    }
    return 0;
}
```

Před chvilkou jsme si říkali, že pole znaků může posloužit i jako primitivní řetězec. Jelikož je tento způsob využití znakových polí velice běžný, obsahují objekty `cin` a `cout` pro znaková pole speciální podporu; umí je načítat i vypisovat přímo:

```
#include <iostream>
using namespace std;
int main()
{
    char jmeno[80];
    cout << "Zadejte sve jmeno: ";
    cin >> jmeno;
    cout << "Jmenujete se " << jmeno << "\n";
    return 0;
}
```

Vstup a výstup programu vypadají takto:

```
Zadejte sve jmeno: Jeff
Jmenujete se Jeff
```

Výhodou tohoto přístupu je, že se při načítání i výpisu pole obejdete bez smyčky; přiřazení hodnoty z `cin` i výpis do `cout` proběhnou v jednom kroku.

Objekt cout a číselná pole

Když zkusíte operátorem << poslat do `cout` nějaké číselné pole, program se přeloží, ale nejspíš se bude chovat jinak, než byste čekali. Následující program je variací na jednu z předchozích ukázek, jen o výpis hodnot pole se místo druhého cyklu stará operátor << a `cout`:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX];
    for (int i=0; i<MAX; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> vysledky[i];
    }
    cout << "Vysledky zkousek: " << vysledky << "\n";
    return 0;
}
```

Vstup a výstup programu vypadá například takto:

```
Zadejte vysledek #1: 76
Zadejte vysledek #2: 84
Zadejte vysledek #3: 91
Vysledky zkousek: 0xbffff450
```

Zatímco pro znaková pole má operátor << objektu cout speciální podporu, jiná pole už tuto výsadu nemají. Když je operátorem << pošlete do cout, vypíše se pouze hodnota jejich názvu, tedy adresa pole v paměti. (K adresám a polím se vrátíme v následující kapitole, kde se předchozí příklad ještě vyjasní.)

Objekt cin a číselná pole

Operátor << objektu cout tedy podporuje všechny typy polí, jen ke znakovým polím se chová jinak. Zato operátor >> objektu cin podporuje pouze znaková pole, s číselnými a dalšími poli si nerozumí vůbec:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX];
    cin >> vysledky;
    return 0;
}
```

Když se program pokusíte přeložit, dostanete chybu. Překladač ukáže na čtení z cin a bude si stěžovat, že operátor >> pro čtení celočíselných polí nemá. V tomto ohledu jsou pole znaků opět výjimečná, protože s nimi operátor >> objektu cin počítá.

Funkce getline objektu cin

V předchozím textu jsme si ukazovali následující program:

```
#include <iostream>
using namespace std;
int main()
{
    char jmeno[80];
    cout << "Zadejte sve jmeno: ";
```

```
    cin >> jmeno;
    cout << "Jmenujete se " << jmeno << "\n";
    return 0;
}
```

Tento program funguje dobře na vstupech, které neobsahují mezery, například na jménu `Jeff`. Podívejte se ale na následující ukázku:

```
Zadejte sve jmeno: Jeff Kent
Jmenujete se Jeff
```

Na stejný problém už jsme narazili ve třetí kapitole, kde jsme zadaný řetězec ukládali do proměnné typu `string`:

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    string jmeno;
    cout << "Zadejte sve jmeno: ";
    cin >> jmeno;
    cout << "Jmenujete se " << jmeno << "\n";
    return 0;
}
```

Tehdy jsme si také vysvětlovali, jak přesně problém vzniká: Když `cin` narazí na mezeru ve vstupu, myslí si, že došel na konec vstupu určeného pro první proměnnou. Pokud chceme řetězec načíst celý, musíme zavolat funkci `getline` objektu `cin`.

Poznámka: Objekt `cin` má kromě jiného ještě funkci `get`, která by nám též pomohla. Jediný rozdíl mezi oběma funkcemi je v tom, že funkce `get` načte uživatelský vstup až po konec řádku, zatímco funkce `getline` načte vstup včetně konce řádku. Díky tomu je `getline` při práci se znakovými poli o něco praktičtější, `get` se používá spíš pro jednotlivé znaky.

Funkce `getline` je *přetížená*. To neznamená, že by měla moc práce. Znamená to, že se dá volat s různými parametry různého typu. Následující program ji volá se dvěma parametry. Prvním z nich je pole, do kterého se má vstup uložit, a druhým je počet znaků, které se mají ze standardního vstupu načíst. (Řekli jsme si o 80 znaků, ale funkce `getline` jich přečte jen 79, aby ještě zbylo místo na závěrečný nulový znak pro ukončení řetězce.)

```
#include <iostream>
using namespace std;
int main()
{
    char jmeno[80];
    cout << "Zadejte sve jmeno: ";
    cin.getline(jmeno, 80);
    cout << "Jmenujete se " << jmeno << "\n";
    return 0;
}
```

Teď už můžete zadat i řetězec s mezerami:

```
Zadejte své jméno: Jeff Kent
Jmenujete se Jeff Kent
```

Další varianta přetížené funkce `getline` má tři parametry:

```
cin.getline(jmeno, 80, '\n');
```

Třetím parametrem je znak, na kterém se má čtení ukončit ještě před dosažením určeného počtu znaků. My jsme zadali znak konce řádku (`'\n'`), který se do řetězce vloží, když uživatel potvrdí vstup klávesou Enter. Po stisknutí klávesy Enter se vstup stejně ukončí sám od sebe, takže uvádět třetí parametr `'\n'` je zbytečné. Někdy se vám ale může hodit ukončit čtení na jiném znaku.

Proměnné typu `string` se funkcí `get` ani `getline` načítat nedají. K tomu slouží samostatná funkce `getline`. („Samostatná" znamená, že se před ni nepíše `cin` a tečka, jako jsme to dělali před chvílí.) Načtení řetězce do proměnné typu `string` ukazuje následující kód:

```
string jmeno;
getline(cin, jmeno);
```

Prvním parametrem funkce je objekt `cin`. Druhým parametrem je proměnná, do které chcete vstup uložit. Jelikož nemusíte uvádět velikost řetězce, počet načtených znaků není ničím omezený.

Předávání polí v parametrech funkcí

Před několika stránkami jsme si ukazovali, jak se dají prvky pole snadno nastavit a vypsat prostřednictvím cyklů:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX];
    for (int i=0; i<MAX; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> vysledky[i];
    }
    for (int i=0; i<MAX; i++)
        cout << "Vysledek #" << i+1 << ": "
             << vysledky[i] << "\n";
    return 0;
}
```

Teď program modularizujeme. Místo abychom všechno dělali v rámci funkce `main`, napíšeme samostatnou funkci pro naplnění pole a samostatnou funkci pro jeho výpis.

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

void napln(int[], int);
void vypis(int[], int);

const int MAX = 3;

int main()
{
    int vysledky[MAX];
    napln(vysledky, MAX);
```

```
    vypis(vysledky, MAX);
    return 0;
}

void napln(int body[], int velikost)
{
    for (int i=0; i<velikost; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> body[i];
    }
}

void vypis(int body[], int velikost)
{
    for (int i=0; i<velikost; i++)
        cout << "Vysledek #" << i+1 << ": "
             << body[i] << "\n";
}
```

Funkce `napln` se stará o naplnění pole hodnotami, funkce `vypis` zobrazí obsah pole. Obě funkce mají dva parametry, prvním z nich je samotné pole a druhým jeho velikost. Pole zadané prvním parametrem se projde smyčkou, počet iterací se řídí velikostí pole předanou v druhém parametru. Všimněte si hranatých závorek za jménem prvního parametru: Podle nich se pozná, že je parametr pole typu `int`, nikoliv obyčejný `int`.

Poznámka: Do složených závorek v hlavičce funkce ani jejím prototypu se žádné číslo nepíše.

Zbývá ještě jedna otázka. Funkce `napln` mění hodnoty pole, které jí předala funkce `main`. V předchozí kapitole jsme si říkali, že pokud se mají změny promítnout do volající funkce, musí být parametry předané odkazem. My ale pole odkazem nepředáváme. Jak to, že se hodnoty přesto úspěšně změní?

Pokud si vzpomínáte, v předchozí kapitole padla zmínka ještě o třetím způsobu předávání parametrů, a to předávání adresou. Předávání adresou funguje v jistém ohledu jako předávání odkazem – volaná funkce může změnit hodnotu předané proměnné. Hodnotou proměnné `vysledky` je adresa začátku pole v paměti, takže když předáváme pole `vysledky`, předáváme ho vlastně adresou. Tomuto problému se budeme podrobně věnovat v následující kapitole.

Shrnutí

Proměnné, se kterými jsme se setkali před touto kapitolou, měly vždy pouze jednu hodnotu. V této kapitole jste se seznámili s proměnnými typu pole, do kterých se dá uložit víc hodnot najednou. Jednotlivé hodnoty pole jsou v paměti uložené za sebou a pracuje se s nimi prostřednictvím čísel, takzvaných indexů. Indexy začínají od nuly, index každého prvku je o jedničku vyšší než index prvku předchozího.

Pole může obsahovat prvky libovolného datového typu, například `int`, `float` nebo `char`. Všechny prvky jednoho pole ale musí být stejného datového typu; jedno pole nemůže obsahovat celá čísla, desetinná čísla a znaky zároveň.

Pole se před použitím musí deklarovat. Deklarace pole vypadá skoro stejně jako deklarace proměnné typu `int`, `char` nebo `float`. Jediný rozdíl mezi deklarací jednoduché, takzvané *skalární* proměnné a deklarací pole je v tom, že pole má za názvem proměnné ještě hranaté závorky. V těchto závorkách může a nemusí být číslo, které určuje velikost pole.

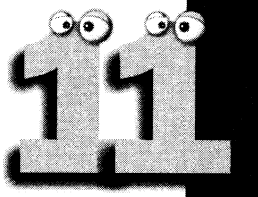
Velikost pole musí být zadaná literálem nebo konstantou. *Literál* je doslovná hodnota, která se nijak dál neinterpretuje. *Konstanta* je jméno, které v celém programu zastupuje jednu neměnnou hodnotu. V této kapitole jste se naučili konstanty deklarovat a používat.

Stejně jako každou jinou proměnnou můžete i pole inicializovat, tedy mu přiřadit nějakou hodnotu už v rámci deklarace. Pokud pole inicializujete, nemusíte do hranatých závorek uvádět jeho velikost; překladač ji odvodí sám podle počtu hodnot na pravé straně přiřazení.

Dále jste se naučili přiřazovat hodnoty pole v cyklu a načítat pole znaků pomocí funkcí `get` a `getline` z objektu `cin`. Nakonec jsme se podívali na předávání polí v parametrech funkcí a řekli jsme si, že v tomto případě jde o předávání parametrů adresou.

Test

1. Může jedno pole obsahovat zároveň celá čísla, desetinná čísla a znaky?
2. Jaký je index prvního prvku pole?
3. Jaký je index posledního prvku pole?
4. Jak se inicializace liší od běžného přiřazení?
5. Jakými dvěma způsoby se dá během inicializace určit velikost pole?
6. K čemu slouží nulový znak?
7. Jaká je hodnota proměnné typu pole?
8. Musí být posledním prvkem znakového pole vždy znak null?
9. Jaký je rozdíl mezi funkcemi `get` a `getline` objektu `cin`?
10. Když funkci předáváte pole, předáváte ho odkazem, adresou, nebo hodnotou?



Abyste nebloudili aneb Ukazatele

Jako malého mě rodiče napomínali, že je neslušné ukazovat. Přesto každý semestr učím studenty ukazovat. Podporuji špatné vychování? Ne. Učím studenty ukazatele, tedy proměnné, jejichž úkolem je „ukazovat" na jinou proměnnou nebo konstantu.

Vy sami jste si možná úlohu ukazatele vyzkoušeli na vlastní kůži. Ptal se vás někdy někdo na cestu? Vsadím se, že jste začali šermovat rukama a ukazovat. Podobně fungují i ukazatele.

Ukazatel je proměnná, která ukazuje na jinou proměnnou nebo konstantu. Dá rozum, že na rozdíl od vás neukazuje rukama ani prstem. Když ukazatel ukazuje na nějakou proměnnou, jeho hodnotou je adresa obsahu této proměnné v paměti. V tom se ostatně možná shodnete: Když se vás někdo zeptá na nějakou budovu a vy nejste dost blízko na ukazování prstem, řeknete *adresu* této budovy.

Ukazatele si mezi studenty vysloužily špatnou pověst, prý proto, že jsou těžké. Podle mě jsou takové obavy zbytečné. Když si v klidu promyslíte, jak ukazatele fungují, nic složitého na nich není. Naučit se s nimi pracovat musíte tak jako tak – některé věci se v C++ za pomoci ukazatelů dělají snáz a některé (například dynamická alokace paměti) se bez nich neobejdou vůbec. Pojďme se tedy do ukazatelů pustit.

Deklarace ukazatele

Stejně jako kterákoliv jiná proměnná se ukazatel musí nejdříve deklarovat. Deklarace ukazatele vypadá skoro stejně jako deklarace proměnné jiného typu, jen význam datového typu před názvem proměnné je jiný.

Jak vypadá deklarace ukazatele

Ukazatel se deklaruje skoro stejně jako proměnné, se kterými jsme pracovali v předchozích kapitolách. Následující příkaz deklaruje ukazatel na proměnnou typu `int`:

```
int* ukazatel;
```

Hvězdička za výrazem `int` je táž hvězdička, která se používá pro násobení. V tomto kontextu ovšem nemá s násobením nic společného, slouží pro deklaraci ukazatele. Později v této kapitole potkáme hvězdičku ještě v jednom dalším kontextu, jako operátor pro takzvané dereferencování ukazatele.

Poznámka: V C++ mívají symboly běžně několik různých významů podle toho, v jakém kontextu se objeví. Například zatímco ampersand (&) v seznamu parametrů funkce znamená, že je parametr předávaný odkazem, ampersand před jménem proměnné slouží jako adresovací operátor; vrací adresu této proměnné.

Ukazatel na `int` můžete stejně dobře deklarovat s hvězdičkou těsně před názvem proměnné:

```
int *ukazatel;
```

Obě deklarace jsou ekvivalentní, překladač většinu mezer a podobných znaků ignoruje. Stejně dobře by fungovala i následující deklarace:

```
int*ukazatel;
```

Já dávám přednost prvnímu příkladu, ve kterém je hvězdička těsně za datovým typem a za ní následuje mezerou oddělený název proměnné. Podle mě je z této varianty nejlépe vidět, že je proměnná ukazatel, ale správně jsou všechny tři možnosti. (Poznámka českého vydavatele: Když napíšete deklaraci int* a, b, deklarujete ukazatel a a celé číslo b. Proto hodně lidí dává přednost druhé variantě, ve které je tato deklarace čitelnější: int *a, b.) Jediným rozdílem mezi deklarací obyčejné proměnné typu `int` a deklarací ukazatele na `int` je ve všech třech případech hvězdička.

Co znamená deklarace ukazatele

Po syntaktické stránce je deklarace ukazatele skoro stejná jako deklarace běžné proměnné. Datový typ uvedený v deklaraci ukazatele má ale jiný význam než datový typ uvedený v deklaraci běžných proměnných. Až doposud určoval datový typ proměnné hodnoty, které se do této proměnné dají uložit. Do proměnné typu `int` můžeme ukládat celá čísla, proměnné typu `char` jsou určené pro znaky a tak dále.

Datový typ ukazatele znamená něco jiného. Popisuje datový typ proměnné, jejíž adresu může ukazatel obsahovat. Jinými slovy: Hodnotou ukazatele na `int` může být pouze adresa nějaké proměnné typu `int`, hodnotou ukazatele na `float` může být pouze adresa proměnné typu `float` a tak dále.

Samotná hodnota uložená v proměnné typu ukazatel je vždy stejného typu, ať už jde o ukazatel na `int`, `float`, `char` nebo cokoliv jiného. Hodnotou ukazatele je vždy nějaká adresa v paměti, obvykle zapisovaná dlouhým šestnáctkovým číslem. Jediný rozdíl mezi ukazateli různého typu je datový typ proměnné, na kterou ukazují. To je dobře vidět v následujícím programu, který za pomoci operátoru `sizeof` vypíše velikosti ukazatelů různého typu. (Datový typ `long` má v mém operačním systému a překladači velikost 4 bajty, velikost datových typů `int`, `float` a `char` se liší.)

```
#include <iostream>
using namespace std;
int main()
{
    int* na_int;
    float* na_float;
    char* na_char;
    cout << "Velikost ukazatele na int: " << sizeof(na_int) << "\n";
    cout << "Velikost ukazatele na float: " << sizeof(na_float) << "\n";
    cout << "Velikost ukazatele na char: " << sizeof(na_char) << "\n";
    return 0;
}
```

Program tedy vypíše následující:

```
Velikost ukazatele na int: 4
Velikost ukazatele na float: 4
Velikost ukazatele na char: 4
```

V ostatních ohledech je ukazatel obyčejná proměnná podobná těm, kterými už jsme se zabývali. Můžete deklarovat konstantní ukazatel, hodnota každého ukazatele je též uložená na nějaké adrese v paměti a tak dále. Jediný rozdíl je v tom, že hodnotou ukazatele je adresa jiné proměnné.

Přiřazování ukazatelům

Pokud je hodnotou ukazatele adresa jiné proměnné, co obsahuje ukazatel, kterému jsme zatím žádnou hodnotu nenastavili? Obsahem takového ukazatele je hodnota, která v příslušných pár bajtech paměti zbyla po předchozích programech – „náhodná" adresa někam do paměti:

```
#include <iostream>
using namespace std;
int main()
{
    int* ukazatel;
    cout << ukazatel << "\n";
    return 0;
}
```

Výstup programu záleží na vašem počítači, program může vypsat například něco jako `0x8fe0154b`.

Nulový ukazatel

Už dříve jsme si říkali, že globální proměnné se inicializují na výchozí hodnotu svého typu. Platí to i pro ukazatele, takže když si na zkoušku deklarujete globální ukazatel a zkusíte si vypsat jeho hodnotu, zjistíte výchozí hodnotu pro typ ukazatel:

```
#include <iostream>
using namespace std;
int *globalni_ukazatel;
int main()
{
    cout << "Výchozí hodnota ukazatele: " << globalni_ukazatel << "\n";
    return 0;
}
```

Program vypíše nulu, takže výchozí hodnotou ukazatelů je nula, stejně jako u celočíselných typů. (Je to celkem logické, protože ukazatele obsahují celočíselnou adresu v paměti.) Tato hodnota je pro ukazatele důležitá, a proto si vysloužila svůj vlastní název – říká se jí *null*. Řada knihoven definuje konstantu `NULL` s hodnotou nula, takže ukazatel můžete na nulu inicializovat takto:

```
int *ukazatel = NULL;
```

Ukazatel, který obsahuje hodnotu null, se označuje jako *nulový ukazatel*. Nulový ukazatel ukazuje na samotný začátek paměti, který je vyhrazený operačnímu systému a do kterého programy nemají přístup. Ukazatel nastavený na `NULL` má oproti neinicializovanému ukazateli jednu velkou výhodu: Když se omylem pokusíte přečíst nebo změnit hodnotu uloženou na adrese nula, program

skončí s chybou. Naproti tomu když se omylem pokusíte přečíst nebo změnit hodnotu na adrese uložené v neinicializovaném ukazateli, pracujete s náhodnou adresou někde v paměti. Program může spadnout a nemusí, a takové chyby se velice špatně hledají.

Nastavení ukazatele na adresu proměnné

Pojďme teď ukazatel konečně nastavit na nějakou „opravdovou" hodnotu, tedy adresu jiné proměnné. K tomu potřebujeme zjistit adresu této proměnné, kterou nám – jak jsme si říkali ve třetí kapitole – vrátí adresovací operátor `&`. Následující program zjistí pomocí operátoru `&` adresu proměnné a uloží ji do ukazatele. Získanou adresu i obsah ukazatele ještě pro pořádek vypíše, abyste viděli, že hodnotou ukazatele je adresa odkazované proměnné.

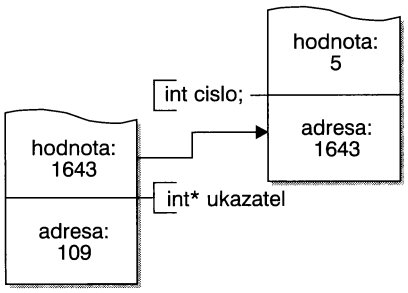
```
#include <iostream>
using namespace std;

int main()
{
    int cislo = 5;
    int* ukazatel = &cislo;
    cout << "Adresa promenne cislo podle operatoru &: " << &cislo << "\n";
    cout << "Hodnota ukazatele: " << ukazatel << "\n";
    return 0;
}
```

Vypsané adresy se budou počítač od počítače lišit, na mém počítači vypadá výstup program takto:

```
Adresa promenne cislo podle operatoru &: 0xbffff468
Hodnota ukazatele: 0xbffff468
```

Rozložení proměnných v paměti ukazuje obrázek 11.1.



Obrázek 11.1: Ukazatel na proměnnou typu int

Dereferencování ukazatelů

Ukazatele se používají především pro čtení a změny hodnot uložených v proměnných, na které ukazatel ukazuje. Následující program mění dvakrát hodnotu celočíselné proměnné `cislo`:

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

```
int main()
{
    int cislo = 5;
    int *ukazatel = &cislo;
    cout << "Pocatecni hodnota promenne: " << cislo << "\n";
    cislo = 10;
    cout << "Hodnota po beznem prirazeni: " << cislo << "\n";
    *ukazatel = 15;
    cout << "Hodnota po prirazeni pres ukazatel: " << cislo << "\n";
    return 0;
}
```

Výstup vypadá následovně:

```
Pocatecni hodnota promenne: 5
Hodnota po beznem prirazeni: 10
Hodnota po prirazeni pres ukazatel: 15
```

První změna proměnné (`cislo = 10`) by pro vás neměla být nic nového, jde o běžné přiřazení hodnoty. Ovšem druhá změna probíhá nově, pomocí takzvaného dereferenčního operátoru:

```
*ukazatel = 15;
```

Dereferenční operátor se zapisuje pomocí hvězdičky, tedy stejně jako násobení nebo deklarace ukazatele. V tomto kontextu ale nemá hvězdička s násobením ani deklarováním ukazatelů nic společného.

Poznámka: Jde o další ukázku jevu, na který jsme narazili už před chvilkou: v C++ se význam některých symbolů liší v závislosti na kontextu, ve kterém je použijete.

Dereferencovací operátor slouží – jak už jeho název napovídá – k dereferencování ukazatelů. Hodnotou dereferencovaného ukazatele je hodnota, na kterou ukazatel ukazuje. Například ve výše uvedeném programu ukazuje proměnná `ukazatel` na proměnnou `cislo`, takže jejím dereferencováním dostaneme hodnotu proměnné `cislo`. Oba následující příkazy proto dělají totéž, a sice mění hodnotu proměnné `cislo`:

```
cislo = 25;
*ukazatel = 25;
```

Dereferencovaný ukazatel se dá běžně používat místo proměnné, na kterou ukazuje. Následující dva příkazy dělají totéž:

```
cislo *= 2;
*ukazatel *= 2;
```

Ve výše uvedených případech je změna proměnné prostřednictvím dereferencovacího operátoru místo běžného přiřazení jen zbytečná komplikace navíc. Někdy jsou ale ukazatele užitečné, nebo dokonce nezbytné – viz například procházení pole pomocí ukazatele, dynamickou alokaci paměti a další příklady, ke kterým se později v této kapitole dostaneme.

Ukazatele jako proměnné a konstanty

Ukazatel může být proměnná i konstanta. Pojďme se na obě možnosti podívat.

Ukazatel jako proměnná

V předchozím příkladu jsme měli ukazatel na jednu celočíselnou proměnnou. Jelikož je ale ukazatel běžná proměnná, nemusí se omezovat na jednu hodnotu. Nic mu nebrání ukazovat postupně na několik různých proměnných, viz následující program:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5, y = 14;

    int *ukazatel = &x;
    cout << "x: " << x << "\n";
    *ukazatel *= 2;
    cout << "x *= 2: " << x << "\n";

    ukazatel = &y;
    cout << "y: " << y << "\n";
    *ukazatel /= 2;
    cout << "y /= 2: " << y << "\n";
    return 0;
}
```

Výstup programu vypadá takto:

```
x: 5
x *= 2: 10
y: 14
y /= 2: 7
```

Pole jako konstantní ukazatel

Ukazatel může být konstantní. Na jeden typ konstantních ukazatelů jsme ostatně narazili už v předchozí kapitole – konstantní ukazatel je každé pole. Jak si asi z předchozí kapitoly pamatujete, hodnotou proměnné typu pole je adresa začátku pole (neboli adresa jeho prvního prvku). Proto mají výrazy `vysledky` a `&vysledky[0]` v následujícím programu stejnou hodnotu.

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX] = {4, 7, 1};
    cout << "Adresa pole, vysledky: " << vysledky << "\n";
    cout << "Adresa pole, &vysledky[0]: " << &vysledky[0] << "\n";
    cout << "Hodnota prvniho prvku, *vysledky: " << *vysledky << "\n";
```

```
    cout << "Hodnota prvniho prvku, vysledky[0]: " << vysledky[0] << "\n";
    return 0;
}
```

Výstup programu:

```
Adresa pole, vysledky: 0xbffff464
Adresa pole, &vysledky[0]: 0xbffff464
Hodnota prvniho prvku, *vysledky: 4
Hodnota prvniho prvku, vysledky[0]: 4
```

Podobně jako `vysledky` a `&vysledky[0]` vrací stejné hodnoty i výrazy `*vysledky` a `vysledky[0]`, dereferencováním pole dostanete hodnotu jeho prvního prvku.

Hodnota samotného pole se ovšem měnit nedá, například příkaz `vysledky++` překladač označí za chybu. Bude si stěžovat na to, že výraz `vysledky` by se neměl objevit na levé straně přiřazení – jinými slovy, nemůžete zvýšit hodnotu konstanty.

Ukazatelová aritmetika

Adresa uložená v ukazateli je celé číslo, takže se s ukazateli dají provádět aritmetické operace jako s běžnými celými čísly.

Průchod pole běžným ukazatelem

Ukazatelová aritmetika se běžně dělává u polí. Jelikož hodnota samotného pole je konstantní ukazatel, který se nedá měnit, musíte si nejdříve deklarovat nový ukazatel a základní adresu do něj zkopírovat. Zkusme si ale nejprve vypsat adresy jednotlivých prvků pole:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX] = {4, 7, 1};
    for (int i = 0; i < MAX; i++)
        cout << i << ". prvek je na adrese " << &vysledky[i]
             << " a ma hodnotu " << vysledky[i] << "\n";
    return 0;
}
```

Výstup programu je následující:

```
0. prvek je na adrese 0xbffff460 a ma hodnotu 4
1. prvek je na adrese 0xbffff464 a ma hodnotu 7
2. prvek je na adrese 0xbffff468 a ma hodnotu 1
```

Tento program prošel pole tradičně pomocí proměnné `vysledky` a hranatých závorek. Říkali jsme si, že pole je jen konstantní ukazatel – zkusme ho tedy zkopírovat do běžného ukazatele a projít pole přes něj:

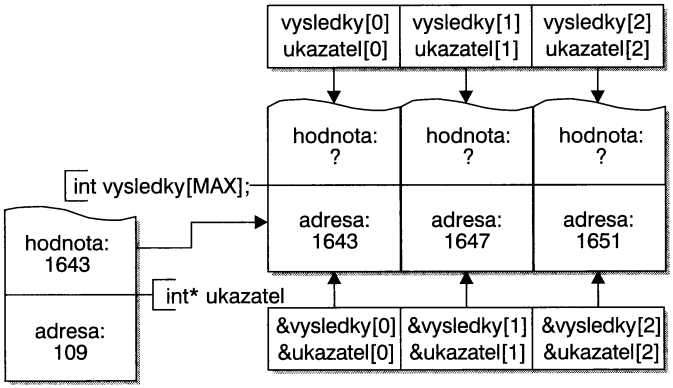
```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
```

```
    int vysledky[MAX] = {4, 7, 1};
    int *ukazatel = vysledky;
    for (int i = 0; i < MAX; i++)
        cout << i << ". prvek je na adrese " << &ukazatel[i]
             << " a ma hodnotu " << ukazatel[i] << "\n";
    return 0;
}
```

Zkopírování adresy pole do proměnné `ukazatel` provede následující příkaz:

```
int *ukazatel = vysledky;
```

Před názvem pole `vysledky` nemusí být adresovací operátor `&`, protože každá proměnná typu pole už je sama o sobě adresa, ukazuje na první prvek pole v paměti. Po přiřazení ukazují obě proměnné na začátek pole, a tak mají i výrazy `ukazatel[i]` a `vysledky[i]` stejnou hodnotu (viz obrázek 11.2).



Obrázek 11.2: Přístup k poli prostřednictvím ukazatelů

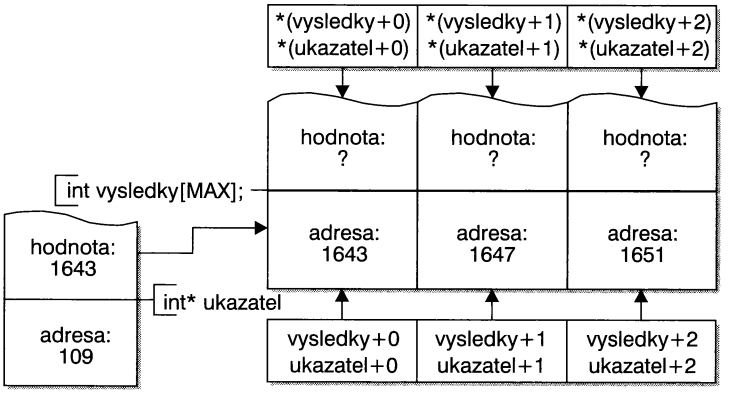
Zvýšení ukazatele operátorem ++

Pomocný ukazatel jsme si ve výše uvedeném příkladu deklarovali, protože na rozdíl od proměnné `vysledky` není konstantní a můžeme ho zvyšovat operátorem ++. Následující program ukazuje, jak se dá pomocí ukazatele a operátoru ++ projít celé pole:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX] = {4, 7, 1};
    int *ukazatel = vysledky;
    for (int i = 0; i < MAX; i++, ukazatel++)
        cout << i << ". prvek je na adrese " << ukazatel
             << " a ma hodnotu " << *ukazatel << "\n";
    return 0;
}
```

Obyčejné číselné typy zvyšuje operátor ++ o jedničku. Ukazatele zvyšuje o počet bajtů, které zabere jejich datový typ. Výše uvedený program je klasickým příkladem ukazatelové aritmetiky. Když ho spustíte, jako první dostanete adresu `0xbffff45c`, za ní bude následovat `0xbffff460` a nakonec `0xbffff464`. Adresy se zvyšují po čtyřech, protože na mém operačním systému a překladači má typ `int` velikost čtyři bajty.

Z tohoto důvodu neukazuje výraz `ukazatel+1` na základní adresu pole plus jedna, nýbrž na základní adresu pole plus čtyři (viz obrázek 11.3). Totéž platí pro `vysledky+1`. Ke druhému prvku pole se tedy v našem příkladu můžete dostat čtyřmi různými výrazy: `vysledky[1]`, `*(vysledky+1)`, `ukazatel[1]` a `*(ukazatel+1)`.



Obrázek 11.3: Zvýšení ukazatele o jedničku

Porovnávání adres

Stejně jako všechny celočíselné hodnoty se i adresy dají porovnávat. Následující program je variace na předchozí příklad – zvyšuje hodnotu ukazatele, dokud jeho adresa nedojde k adrese posledního prvku pole (`&vysledky[MAX-1]`).

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX] = {4, 7, 1};
    int *ukazatel = vysledky;
    int i = 0;
    while (ukazatel <= &vysledky[MAX-1])
    {
        cout << i << ". prvek je na adrese " << ukazatel
             << " a ma hodnotu " << *ukazatel << "\n";
        i++;
        ukazatel++;
    }
    return 0;
}
```

Jak je vidět z obrázků 11.2 a 11.3, výraz `&vysledky[MAX-1]` z podmínky cyklu by se dal nahradit výrazem `vysledky + MAX-1`.

Snížení ukazatele operátorem --

Analogické úvahy se vztahují na snížení hodnoty ukazatele operátorem `--`, který sníží hodnotu ukazatele o velikost jeho datového typu v bajtech. Operátor `--` se dá použít k průchodu pole pozpátku:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main()
{
    int vysledky[MAX] = {4, 7, 1};
    int *ukazatel = &vysledky[MAX-1];
    int i = MAX-1;
    while (ukazatel >= &vysledky[0])
    {
        cout << i << ". prvek je na adrese " << ukazatel
             << " a ma hodnotu " << *ukazatel << "\n";
        i--;
        ukazatel--;
    }
    return 0;
}
```

Výstup tím pádem vypadá následovně:

```
2. prvek je na adrese 0xbffff484 a ma hodnotu 1
1. prvek je na adrese 0xbffff480 a ma hodnotu 7
0. prvek je na adrese 0xbffff47c a ma hodnotu 4
```

Klíčové je přiřazení `*ukazatel = &vysledky[MAX-1]`, který nastaví ukazatel na poslední prvek pole. Adresa se pak ve smyčce snižuje na předchozí a předchozí prvek, dokud není menší než adresa začátku pole. Výraz `&vysledky[MAX-1]` z inicializace ukazatele bychom mohli nahradit výrazem `vysledky + MAX-1`, podobně jako v předchozím případě.

Ukazatele jako parametry funkce

Ukazatele se dají předávat funkcím jako parametry. V hlavičce a prototypu funkce se ukazatele označují hvězdičkou. Pokud jde o pole, díky příbuznosti polí a ukazatelů máte na výběr mezi zápisem s hvězdičkou a hranatými závorkami.

Předávání pole ukazatelem

V předchozí kapitole jsme měli následující program, který obsahoval jednu funkci pro naplnění pole a druhou pro jeho výpis, aby příslušný kód nepřekážel v `main`:

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

void napln(int[], int);
```

```
void vypis(int[], int);

const int MAX = 3;

int main()
{
    int vysledky[MAX];
    napln(vysledky, MAX);
    vypis(vysledky, MAX);
    return 0;
}

void napln(int body[], int velikost)
{
    for (int i=0; i<velikost; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> body[i];
    }
}

void vypis(int body[], int velikost)
{
    for (int i=0; i<velikost; i++)
        cout << "Vysledek #" << i+1 << ": "
             << body[i] << "\n";
}
```

Jak už bylo řečeno, obě funkce přebírají první parametr adresou. Když volaná funkce (například `napln`) změní hodnotu parametru předaného adresou, změní se i jeho hodnota ve volající funkci (zde `main`), podobně jako kdyby byl parametr předaný odkazem. Když funkce nacti načte hodnoty pole `body`, změní se i odpovídající pole `vysledky` ve funkci `main`.

V prototypech a hlavičkách funkcí `napln` a `vypis` jsou za prvním parametrem uvedené závorky `[]`, které označují parametr typu pole. Stejně dobře ale můžete využít souvislosti ukazatelů s poli a označit předávaný parametr jako ukazatel:

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

void napln(int*, int);
void vypis(int*, int);

const int MAX = 3;

int main()
{
    int vysledky[MAX];
    napln(vysledky, MAX);
    vypis(vysledky, MAX);
    return 0;
}

void napln(int* body, int velikost)
{
    for (int i=0; i<velikost; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
```

```
        cin >> body[i];
    }
}

void vypis(int* body, int velikost)
{
    for (int i=0; i<velikost; i++)
        cout << "Vysledek #" << i+1 << ": "
             << body[i] << "\n";
}
```

Jak ukazuje následující srovnání dvou prototypů funkce `napln`, jediný rozdíl spočívá v nahrazení hranatých závorek `[]` hvězdičkou *:

```
void napln(int[], int);
void napln(int*, int);
```

Podobně dopadne hlavička funkce; jen hranaté závorky se píšou až za název parametru, zatímco hvězdička za název typu:

```
void napln(int body[], int velikost)
void napln(int* body, int velikost)
```

Je na vás, jestli budete pole opravdu předávat jako pole, nebo jestli se rozhodnete ho předávat jako ukazatel. Z pohledu programu jsou obě možnosti rovnocenné.

Předávání jednoduché proměnné ukazatelem

Pole se adresou předává snadno, protože proměnná typu pole už sama o sobě obsahuje adresu. Běžně ale bývá potřeba adresou předat i jednotlivou proměnnou. Zápis hranatými závorkami v takovém případě nepřipadá v úvahu, protože nejde o pole. Když chcete adresou předat jednotlivou proměnnou, zbývá vám jen zápis s hvězdičkou.

Stejně jako změna parametru předaného odkazem se i změna parametru předaného adresou promítne do volající funkce, takže se oba způsoby předávání liší pouze zápisem. Začněme pro srovnání následujícím programem z deváté kapitoly, ve kterém funkce zdvojnásobí parametr předaný odkazem:

```
#include <iostream>
using namespace std;
void zdvojnasob(int&);

int main()
{
    int cislo;
    cout << "Zadejte cislo: ";
    cin >> cislo;
    zdvojnasob(cislo);
    cout << "Zpatky v main, cislo po nasobeni: " << cislo << "\n";
    return 0;
}

void zdvojnasob(int& x)
{
    cout << "Nasobene cislo: " << x << "\n";
    x *= 2;
    cout << "Cislo po nasobeni: " << x << "\n";
}
```

Vstup a výstup vypadají následovně:

```
Zadejte cislo: 3
Nasobene cislo: 3
Cislo po nasobeni: 6
Zpatky v main, cislo po nasobeni: 6
```

Teď předávání parametru odkazem přepíšeme na předávání adresou:

```
#include <iostream>
using namespace std;
void zdvojnasob(int*);

int main()
{
    int cislo;
    cout << "Zadejte cislo: ";
    cin >> cislo;
    zdvojnasob(&cislo);
    cout << "Zpatky v main, cislo po nasobeni: " << cislo << "\n";
    return 0;
}

void zdvojnasob(int* x)
{
    cout << "Nasobene cislo: " << *x << "\n";
    *x *= 2;
    cout << "Cislo po nasobeni: " << *x << "\n";
}
```

Po stránce syntaxe jsou mezi oběma programy čtyři rozdíly:

1. V prototypu funkce se parametry předávané odkazem označují ampersandem (&), zatímco parametry předávané adresou se označují hvězdičkou (*):

   ```
   void zdvojnasob(int&); // odkazem
   void zdvojnasob(int*); // adresou
   ```

2. K podobné výměně ampersandu za hvězdičku dojde i v hlavičce funkce:

   ```
   void zdvojnasob(int& x) // odkazem
   void zdvojnasob(int* x) // adresou
   ```

3. Při samotném volání funkce nemusíte před parametr předávaný odkazem psát nic, ale před parametr předávaný adresou musíte napsat adresovací operátor &, abyste skutečně předali adresu parametru:

   ```
   zdvojnasob(cislo);  // odkazem
   zdvojnasob(&cislo); // adresou
   ```

4. V těle volané funkce se parametr předaný adresou musí na rozdíl od parametru předaného odkazem dereferencovat, abyste dostali hodnotu ukazatele:

   ```
   // Parametr x je předaný odkazem, netřeba dělat nic.
   cout << "Nasobene cislo: " << x << "\n";
   x *= 2;
   cout << "Cislo po nasobeni: " << x << "\n";

   // Parametr x je předaný adresou, musíme ho dereferencovat.
   ```

```
    cout << "Nasobene cislo: " << *x << "\n";
    *x *= 2;
    cout << "Cislo po nasobeni: " << *x << "\n";
```

Oprávněně vás může napadnout otázka, proč předávat jednotlivé proměnné adresou, když je předávání odkazem jednodušší. Svým studentům naschvál odpovídám, že předávání adresou je výmysl některých sadistických učitelů informatiky (jen žádná jména!). Žerty stranou, některé knihovní funkce předávání adresou používají. Při dynamické alokaci paměti a vracení ukazatelů z funkcí navíc jinou možnost než předávání adresou nemáte.

Dynamická alokace paměti

V předchozí kapitole jsme si řekli, že velikost pole musí být pevně daná konstantou nebo literálem během deklarace, a že nemůže být určená proměnnou. Následující příklad z předchozí kapitoly se snaží velikost pole neúspěšně zadat proměnnou:

```
#include <iostream>
using namespace std;
int main()
{
    int pocet;
    cout << "Zadejte pocet testu: ";
    cin >> pocet;
    int vysledky[pocet];
    return 0;
}
```

Když se program pokusíte přeložit, dostanete chybu. (Poznámka českého vydavatele: Ne nutně na všech překladačích. Pokud překládáte pomocí g++, asi budete muset přidat parametr `-pedantic`.) Překladač označí deklaraci pole (`int vysledky[pocet]`) a bude si stěžovat, že očekával konstantní výraz. Paměť pro toto pole se totiž rezervuje už během překladu programu. Když velikost pole zadáte proměnnou, překladač nemá jak zjistit, kolik paměti má pro pole vyhradit – hodnota proměnné se může měnit, v našem případě se ji dokonce dozvíme až za běhu programu.

My bychom ale velikost pole chtěli nechat až na uživateli, aby pole nebylo ani moc velké, ani moc malé. K tomu je pole potřeba alokovat až za běhu programu, viz následující kód:

```
#include <iostream>
using namespace std;
int main()
{
    int pocet;

    cout << "Zadejte pocet testu: ";
    cin >> pocet;
    int* vysledky = new int[pocet];

    for (int i=0; i<pocet; i++)
    {
        cout << "Zadejte vysledek #" << i+1 << ": ";
        cin >> vysledky[i];
    }

    for (int i=0; i<pocet; i++)
        cout << "Vysledek #" << i+1 << ": "
```

```
            << vysledky[i] << "\n";

    delete [] vysledky;
    return 0;
}
```

Ukázkový vstup a výstup programu vypadá takto:

```
Zadejte pocet testu: 3
Zadejte vysledek #1: 66
Zadejte vysledek #2: 88
Zadejte vysledek #3: 77
Vysledek #1: 66
Vysledek #2: 88
Vysledek #3: 77
```

S dynamickou alokací pole už se program přeloží, protože se paměť pro pole bude alokovat až za běhu programu, a tak překladač nepotřebuje znát velikost pole předem. Proměnné alokované za běhu programu jsou v paměti na jiném místě než proměnné alokované během překladu. Zatímco během překladu se paměť rezervuje na takzvaném *zásobníku*, proměnné alokované za běhu programu končí na *haldě*.

Poznámka: Výrazy ***zásobník*** a ***halda*** se používají také pro označení konkrétních datových struktur. My je ale budeme používat výhradně pro označení dvou různých částí paměti, tedy ***zásobník*** jako paměť na proměnné alokované během překladu a ***halda*** jako paměť na proměnné alokované za běhu.

Dynamicky můžete alokovat libovolnou proměnnou, ale nejčastěji se dynamická alokace používá pro pole (viz náš příklad) a struktury nebo třídy, ke kterým se dostaneme ve čtrnácté kapitole.

Pro dynamickou alokaci paměti je potřeba ukazatel. Když alokujeme pole, ukazatel musí být stejného typu jako pole. Samotná alokace pak proběhne v rámci přiřazení, viz následující příkaz z naší ukázky:

```
int* vysledky = new int[pocet];
```

Na levé straně přiřazení je ukazatel. Pravá strana začíná operátorem `new`, který se stará o dynamickou alokaci paměti, a pak následuje datový typ, pro který chceme v paměti vyhradit místo. V našem případě jde o pole, jehož velikost je zadaná proměnnou (stejně dobře bychom samozřejmě mohli použít literál nebo konstantu).

Pro práci s alokovaným polem slouží ukazatel z levé strany přiřazení, například ve smyčce můžete obsah pole projít výrazem `vysledky[i]`. Důležitou vlastností dynamicky alokovaných proměnných je jejich životnost – podobně jako globální nebo statické místní proměnné zanikají dynamicky alokované proměnné až na konci programu. Pokud ovšem ještě před tím zanikne příslušný ukazatel, už se k alokované paměti nemáte jak dostat. Paměť je stále vyhrazená, ale nepřístupná. Takové situaci se říká *únik paměti* neboli *memory leak*.

Program, který dynamickou paměť jen alokuje, je jako knihovna, ve které čtenáři nevrací přečtené knihy. Knihy i paměť časem dojdou.

V našem konkrétním případě se o ztrátu přístupu k alokované paměti bát nemusíme, protože ukazatel na ni zanikne až na konci programu. Pokud je ale ukazatel na alokovanou paměť jen místní proměnná nějaké menší funkce (jak si hned ukážeme), po návratu z funkce zanikne a s ním zmizí i adresa alokované paměti, ke které se tím pádem už nemáme jak dostat.

Pro vrácení dynamicky alokované paměti zpět systému slouží operátor `delete`. Jako parametr zadáte ukazatel na paměť, kterou chcete uvolnit. Pokud místo samostatné proměnné uvolňujete pole, před název ukazatele přijdou ještě prázdné hranaté závorky:

```
delete [] ukazatel;
```

Se samotným ukazatelem nedělá operátor `delete` nic, jen uvolní paměť na odkazované adrese.

Poznámka: Operátoru `delete` byste měli dávat jen ukazatele na dynamicky alokovanou paměť. Pokus o uvolnění paměti alokované v zásobníku by nemusel dopadnout dobře.

Jelikož je ukazatel vaším jediným pojítkem s alokovanou pamětí, neměli byste do něj ukládat žádnou jinou adresu, dokud si tu původní neschováte do jiného ukazatele. V opačném případě už se k původní adrese nedostanete, takže jde opět o únik paměti.

Ukazatel jako návratová hodnota funkce

V předchozí kapitole jste se naučili několik variant inicializace znakového pole. Následující program ukazuje ještě jednu další:

```
#include <iostream>
using namespace std;
int main()
{
    char* jmeno = "Jeff Kent";
    cout << jmeno << "\n";
    return 0;
}
```

Program jednoduše vypíše řetězec `Jeff Kent`. Klíčový je příkaz `char* jmeno = "Jeff Kent"`, který se nápadně podobá příkazu `char jmeno[] = "Jeff Kent"`. V obou případech dostanete ukazatel na pole znaků, jehož velikost je zadaná inicializátorem. Rozdíl je v tom, že ukazatel vytvořený druhým příkazem (`char jmeno[] = ...`) je konstantní.

Ukazatel na místní proměnnou

Řekněme, že byste chtěli poslechnout mou radu z deváté kapitoly a modularizovat program napsáním samostatné funkce `precti_jmeno`, která se o načtení jména postará. Tato funkce vytvoří znakové pole, načte do něj uživatelský vstup funkcí `getline` objektu `cin` a nakonec vrátí ukazatel na toto pole:

```
#include <iostream>
using namespace std;
char *precti_jmeno();

int main()
{
    char* jmeno = precti_jmeno();
    cout << jmeno << "\n";
    return 0;
}
```

```
char* precti_jmeno()
{
    char jmeno[80];
    cout << "Zadejte jmeno: ";
    cin.getline(jmeno, 80);
    return jmeno;
}
```

Vstup a výstup programu vypadají například takto:

```
Zadejte jmeno: Jeff Kent
.................D ............ ........8 ........
```

Vypsané jméno je sice zajímavé, ale nepříliš praktické. A především vůbec neodpovídá tomu, co jsem zadával. Chyba je v tom, že funkce `precti_jmeno` vrací ukazatel na svou místní proměnnou, která po návratu do `main` přestane platit. My bychom potřebovali životnost znakového pole nějak prodloužit. Samozřejmě se nabízí možnost udělat z něj globální proměnnou, ale jak jsme si říkali v deváté kapitole, existují lepší řešení.

Ukazatel na statickou místní proměnnou

Jedno z lepších řešení je deklarovat pole `jmeno` jako statickou proměnnou:

```
#include <iostream>
using namespace std;
char *precti_jmeno();

int main()
{
    char* jmeno = precti_jmeno();
    cout << jmeno << "\n";
    return 0;
}

char* precti_jmeno()
{
    static char jmeno[80];
    cout << "Zadejte jmeno: ";
    cin.getline(jmeno, 80);
    return jmeno;
}
```

Výstup z takového programu už vypadá mnohem lépe:

```
Zadejte jmeno: Jeff Kent
Jeff Kent
```

Statická proměnná sice neplatí jinde než v rámci své funkce, ale zanikne až s koncem programu. Ukazatel vrácený funkcí `precti_jmeno` tedy i po návratu do `main` ukazuje na platná data a program funguje.

Ukazatel na dynamicky alokovanou proměnnou

Ještě lepší alternativa, která se nám nabízí po přečtení této kapitoly, je vrátit ukazatel na dynamicky alokovanou proměnnou:

```
#include <iostream>
```

```
using namespace std;
char *precti_jmeno();

int main()
{
    char* jmeno = precti_jmeno();
    cout << jmeno << "\n";
    delete [] jmeno;
    return 0;
}

char* precti_jmeno()
{
    char* jmeno = new char[80];
    cout << "Zadejte jmeno: ";
    cin.getline(jmeno, 80);
    return jmeno;
}
```

Tentokrát nepřestane pole `jmeno` po návratu do `main` platit díky tomu, že je alokované dynamicky. Jak už jsme si říkali před chvílí: Když ve funkci alokujete paměť a její adresu uložíte do místního ukazatele, po návratu z funkce tento ukazatel zanikne a paměť zůstane až do konce programu obsazená a nedostupná, protože už nebudeme mít její adresu. V tomto případě podobný problém nenastal, protože adresu alokované paměti vracíme jako návratovou hodnotu funkce a ve funkci `main` ji hned uložíme do proměnné `jmeno`. Na konci funkce `main` pak paměť dynamicky alokovanou ve funkci `precti_jmeno` uvolníme operátorem `delete`.

Tento program je příkladem programu, ve kterém se na jednu adresu odkazuje víc ukazatelů. Jakmile pomocí některého z těchto ukazatelů alokovanou paměť zase uvolníte operátorem `delete`, musíte si dát pozor, abyste se o totéž nepokusili u některého ze zbývajících ukazatelů. Pokud se kus paměti pokusíte uvolnit více než jednou, nedopadne to dobře.

Shrnutí

Ukazatel je proměnná nebo konstanta, která obsahuje adresu jiné proměnné nebo konstanty. Některé úkoly se díky ukazatelům řeší snáz, u jiných – například dynamické alokace proměnných – se bez ukazatelů neobejdete vůbec.

Ukazatel se stejně jako kterákoliv jiná proměnná nebo konstanta musí nejprve deklarovat. Jediný rozdíl mezi deklarací ukazatele a obyčejné proměnné je v tom, že ukazatel má mezi datovým typem a názvem proměnné hvězdičku. Datový typ zmíněný v deklaraci ukazatele ovšem na rozdíl od běžných proměnných neurčuje typ hodnoty, která se do ukazatele dá uložit. Hodnotou všech ukazatelů bez ohledu na jejich konkrétní typ je celočíselná adresa v paměti, která se většinou zapisuje jako šestnáctkové číslo. Datový typ ukazatele určuje, jaká hodnota je na této adrese uložená. Jinými slovy: na adrese uložené v ukazateli typu `int` je vždy uložená hodnota typu `int`, na adrese uložené v ukazateli typu `float` je vždy uložená hodnota typu `float` a tak podobně.

Pro čtení a změny hodnoty uložené na odkazované adrese slouží dereferenční operátor (`*`). Při dereferencování ukazatele je potřeba dávat pozor na to, jakou hodnotu ukazatel obsahuje. Pokud jste ho ještě neinicializovali, obsahuje pravděpodobně nějakou náhodnou adresu a jeho dereferencování by nedopadlo dobře (totéž platí pro ukazatele obsahující výchozí hodnotu `NULL`). Když chcete do ukazatele přiřadit adresu nějaké proměnné, použijte adresovací operátor `&`.

Ukazatel může být konstantní. Příkladem konstantního ukazatele je každé pole – hodnotou proměnné typu pole je konstantní ukazatel na první prvek pole. Pole a ukazatele jsou do velké míry zaměnitelné, a tak můžete proměnnou typu pole přiřadit do běžného ukazatele a s tím dál pracovat pomocí hranatých závorek `[]` jako s polem.

Adresa uložená v ukazateli je obyčejné celé číslo, takže se s ní dají provádět běžné aritmetické operace. Při použití ukazatele jako pole je běžné zvyšování adresy operátorem ++. Tento operátor nezvýší adresu nutně o jedničku, zvýší ji o velikost jednoho prvku pole v bajtech.

Parametry funkcí se v principu dají předávat jen dvěma způsoby, hodnotou a odkazem. Změny parametrů předaných odkazem se promítnou i do volající funkce, změny parametrů předaných hodnotou nikoliv. Když místo proměnné předáte ukazatel na tuto proměnnou, předáváte sice hodnotou, ale volaná funkce může dereferencováním ukazatele změnit hodnotu proměnné ve volající funkci. Předávání ukazatele na proměnnou je tak časté, že si vysloužilo vlastní název – předávání adresou.

Předávání adresou vypadá a funguje podobně jako předávání odkazem. Hlavička a prototyp funkce se téměř nezmění, jen ampersand (`&`) označující předávání odkazem se nahradí hvězdičkou (`*`). Rozdíl je vidět především na volání funkce a v jejím těle: pokud funkci předáváte něco jiného než ukazatel nebo pole, musíte před parametr vložit adresovací operátor `&` a v těle funkce pak předaný ukazatel dereferencovat.

Pokud chcete vytvořit pole o velikosti, kterou budete znát až za běhu programu, musíte paměť pro toto pole alokovat dynamicky. Dynamicky alokované proměnné jsou uložené v jiné části paměti než ostatní proměnné. Zatímco během překladu se paměť na proměnné vyhrazuje v takzvaném zásobníku, za běhu programu se proměnné alokují na haldě. Pro alokaci dynamické paměti slouží operátor `new`. Jako parametr mu zadáte datový typ, pro který chcete v paměti vyhradit místo, a operátor vám vrátí ukazatel na vyhrazený kus paměti.

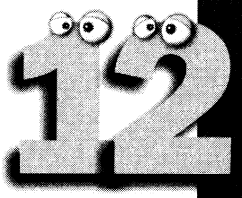
Dynamicky alokovaná paměť zůstává obsazená až do konce programu, nebo dokud ji operátorem `delete` neuvolníte. Pokud se ovšem stane, že ukazatel na tuto část paměti zanikne (například po návratu z funkce), přijdete o její adresu a paměť se uvolní až na konci programu. Dokud program neskončí, nebudete s ní moci pracovat, ani ji uvolnit. Této situaci se říká únik paměti, anglicky *memory leak*. Zmíněným problémům se můžete vyhnout tím, že paměť hned po použití uvolníte operátorem `delete`.

Ukazatel může být i návratovou hodnotou funkce. V takovém případě by měl ukazovat na statickou místní proměnnou nebo dynamicky alokovanou paměť, nikdy ne na obyčejnou místní proměnnou.

Test

1. Co je to ukazatel?
2. Která činnost se v C++ bez ukazatelů neobejde?
3. V čem se deklarace proměnné typu `int` liší od deklarace ukazatele na `int`?
4. Jaký má v deklaraci ukazatele význam datový typ?
5. Co je to `NULL`? K čemu slouží?
6. Když chceme do ukazatele uložit adresu nějaké proměnné, kterým operátorem ji získáme?

7. K čemu je dereferenční operátor?
8. Může jeden ukazatel v průběhu programu ukazovat na několik různých adres?
9. Může na jednu adresu ukazovat větší počet ukazatelů?
10. Co s ukazatelem provede operátor ++?
11. K čemu jsou operátory `new` a `delete`?



Znaky, céčkové řetězce a třída string

Slovo „znak“ se dá chápat mnoha způsoby, například charateristickým znakem datla je červená čepička. My se raději omezíme na významy související s programováním. Znaků existuje nepřeberné množství – máme písmena abecedy, číslice, interpunkci, různé typy mezer a podobně. Pro drtivou většinu prakticky používaných znaků existuje nějaký číselný kód, například z ASCII nebo Unicode. Proto se dá většina znaků bez problémů uložit do proměnné vhodně zvoleného znakového typu.

Jednotlivé znaky samozřejmě nestačí na všechno. Uživatelský vstup se běžně skládá z delších posloupností znaků, takzvaných řetězců. V desáté kapitole jsme si říkali, že jednoduchý způsob uložení řetězců nabízí *céčkové řetězce* neboli pole znaků zakončená nulovým znakem. Céčkové řetězce se jim říká, protože právě v této podobě se s řetězci běžně pracuje v jazyce C. Jazyk C++ z C vychází a má s ním hodně společného. Proto mu nejsou cizí ani céčkové řetězce, se kterými umí pracovat například operátor `>>` objektu `cin` nebo operátor `<<` objektu `cout`.

Řetězce se často používají při vkládání dat. Dejme tomu, že byste od uživatele chtěli načíst čtyřmístné číslo. Když uživatel místo čísla zadá řetězec `Jeff` a vy se tuto hodnotu pokusíte uložit přímo do číselné proměnné, program buď spadne, nebo se v proměnné objeví nějaká „nesmyslná“ hodnota typu –858 993 460. My ale můžeme vstup uložit i do řetězce, například do znakového pole o velikosti pět znaků (aby se nám vešly čtyři znaky vstupu i závěrečný nulový znak). Tak k žádné chybě nedojde, protože do znakového pole můžeme uložit cokoliv. Pak stačí ověřit, jestli uživatel skutečně zadal číslice. Pokud ano, můžete je standardními knihovními funkcemi převést na opravdové číslo ve formátu `int`, `long`, `float` a podobně.

Programovací jazyk C++ přišel s vlastním datovým typem pro řetězce, třídou `string`. Tato třída se v C++ používá místo starších céčkových řetězců. V následujícím textu se podíváme na různé funkce pro práci s oběma řetězcovými typy a navzájem je porovnáme.

Čtení znaků

Oprávněně by vás mohlo zajímat, proč je celá podkapitola věnovaná čtení znaků. Přečíst znak je přeci celkem jednoduché, stačí operátor `>>` objektu `cin`:

```
char znamka;
cout << "Zadejte znamku: ";
cin >> znamka;
```

V programování jsou ovšem věci jen zřídka tak jednoduché, jak na první pohled vypadají (stejně jako v životě). Čtení znaků není výjimka. Uživatelský vstup totiž končí stiskem klávesy Enter, což vede k několika podnětným a naštěstí i snadno řešitelným problémům.

Press any key to continue...

Předchozí kus kódu přebral od uživatele jeden znak a uložil ho do znakové proměnné. Ne vždy ovšem chceme načtený znak někam ukládat. Často chce program pouze počkat na stisk libovolné klávesy, aby pak mohl pokračovat. Anglická varianta příslušného hlášení („Press any key to continue...“) je všem uživatelům dobře známá. Vedla dokonce ke vzniku vtipů, ve kterých uživatelé volají na technickou podporu a stěžují si, že jim na klávesnici chybí klávesa Any. Dá rozum, že stačí stisknout libovolnou klávesu, včetně klávesy Enter. To je sice velká legrace, ale naprogramovat čekání na libovolnou klávesu je o něco těžší, než by se na první pohled mohlo zdát. Podívejme se na následující program:

```
#include <iostream>
using namespace std;
int main()
{
    char znak;
    while (1)
    {
        cout << "K = konec, libovolna jina klavesa = pokracovat: ";
        cin >> znak;
        if (znak == 'k' || znak == 'K')
            break;
        cout << "Tak jeste jednou.\n";
    }
    cout << "Konec.\n";
    return 0;
}
```

Pokud zadáte `K` nebo `k`, program poslušně skončí. Stejně tak fungují správně skoro všechny ostatní znaky, například písmena kromě k/K, čísla nebo interpunkce. Co ovšem nefunguje, je klávesa Enter. Když stisknete Enter, program poračovat nebude. Objekt `cin` bude dál čekat, až zadáte nějaký tisknutelný znak. Jeho operátor `>>` totiž všechny mezery, konce řádků a podobné znaky ignoruje.

Funkce cin.get

V desáté kapitole jsme si ukázali funkci `getline` objektu `cin`. Jde o takzvanou *členskou funkci*. Členské funkce se nedají volat samostatně – v tom se liší od obyčejných funkcí, jako je třeba `pow`, kterou jsme ve čtvrté kapitole umocňovali. Členská funkce je součástí objektu, například objekt `cin` má členskou funkci `getline`. Když voláte členskou funkci `getline` objektu `cin`, napíšete nejprve jméno objektu, za něj tečku a za tečku název funkce: `cin.getline(promenna, 80)`.

Nás teď bude zajímat další členská funkce objektu `cin`, a to funkce `get`. Zběžně jsme na ni narazili už v desáté kapitole. Tam jsme ji volali se dvěma nebo třemi parametry, z nichž první bylo vždy znakové pole. Členská funkce `get` se dá volat i s jedním parametrem, popřípadě úplně bez parametrů. Tyto varianty na rozdíl od těch předchozích nečtou céčkový řetězec, ale jen jeden znak. V následujícím textu nás bude zajímat varianta s jedním parametrem, k variantě bez parametrů se dostaneme později.

Varianta funkce `get` s jedním parametrem vyžaduje parametr typu `char`, který naplní znakem přečteným ze standardního vstupu. Znak nemusí být nutně tisknutelný, funkce klidně vrátí i znak `\n` zadaný klávesou Enter. (V tom se liší od operátoru >>.) Zkusme v předchozím programu nahradit operátor >> čtením pomocí funkce `get`:

```
#include <iostream>
using namespace std;
int main()
{
    char znak;
    while (1)
    {
        cout << "K = konec, libovolna jina klavesa = pokracovat: ";
        cin.get(znak);
        if (znak == 'k' || znak == 'K')
            break;
        cout << "Tak jeste jednou.\n";
    }
    cout << "Konec.\n";
    return 0;
}
```

Tentokrát už program pokračuje i po stisku klávesy Enter:

```
K = konec, libovolna jina klavesa = pokracovat:
Tak jeste jednou.
K = konec, libovolna jina klavesa = pokracovat: k
Konec.
```

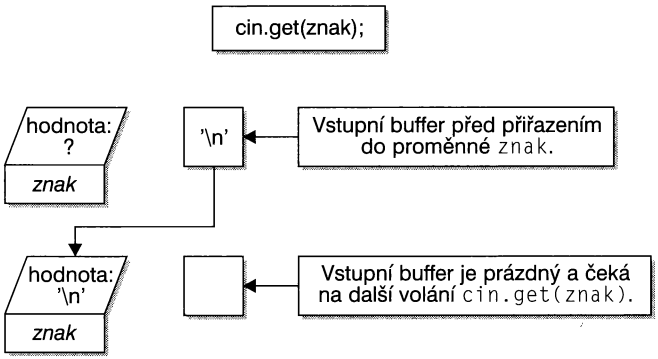
Jenže pokud zadáte libovolný znak kromě k/K, následující výzva programu se jaksi přeskočí:

```
K = konec, libovolna jina klavesa = pokracovat: x
Tak jeste jednou.
K = konec, libovolna jina klavesa = pokracovat: Tak jeste jednou.
K = konec, libovolna jina klavesa = pokracovat: k
Konec.
```

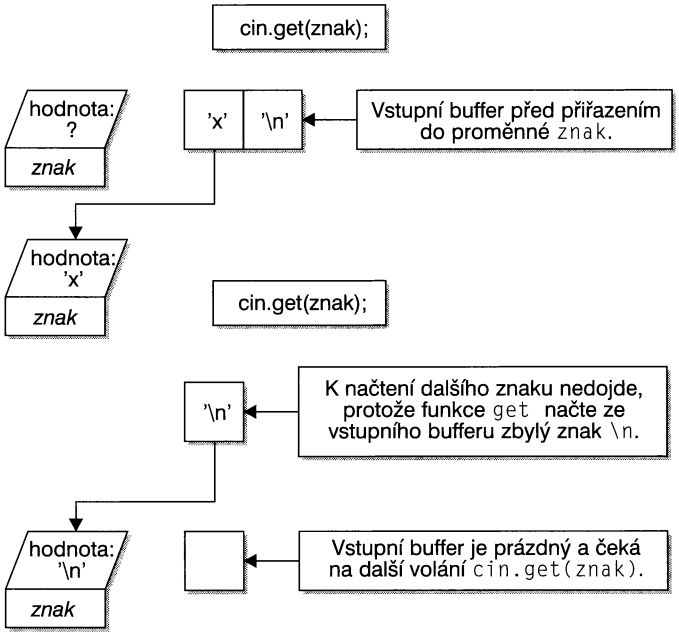
Je vidět, že vyřešením problémů s klávesou Enter jsme do kódu zanesli chybu, program špatně čte tisknutelné znaky. Vysvětlení problému vyžaduje stručnou zmínku o takzvaném *vstupním bufferu*. Vstupní buffer je kus paměti, do kterého se ukládají načtená data (například znaky z klávesnice), dokud je program nezpracuje, řekněme operátorem >> objektu `cin` nebo jeho členskými funkcemi `get` a `getline`.

Když smyčka začne, vstupní buffer je prázdný. Proto se smyčka zastaví na příkazu `cin.get(znak)`, který čeká na zadání znaku. Když stisknete Enter, do vstupního bufferu se uloží znak konce řádku. Odtud si ho převezme funkce `get`, která ho uloží do proměnné znak. (Viz obrázek 12.1.) Vstupní buffer je opět prázdný a v další iteraci se postup opakuje.

Naproti tomu když zadáte `x` a klávesou Enter ukončíte vstup, ve vstupním bufferu jsou místo jednoho znaku dva – `x` a `\n`, viz obrázek 12.2. Funkce `get` načte první z nich (`x`) a uloží ho do proměnné `znak`. Znak `\n` ve vstupním bufferu zůstane.



Obrázek 12.1: Vstupní buffer po stisku klávesy Enter



Obrázek 12.2: Vstupní buffer po zadání písmene x a potvrzení klávesou Enter

Při další otáčce cyklu přečte funkce `get` ze vstupního bufferu zbývající znak `\n`, takže na uživatelský vstup vůbec nedojde. Vstupní buffer se tím vyprázdní, takže další iterace už proběhne správně.

Funkce cin.ignore

Řešením je odstranit znak `\n` ze vstupního bufferu ještě před voláním funkce `get`. K tomu můžeme použít členskou funkci `ignore` objektu `cin`. Podobně jako `get` a `getline` je i tato funkce přetížená – dá se volat bez parametrů, s jedním parametrem nebo dvěma parametry. Když ji

zavoláte bez parametrů, načte jeden znak ze vstupního bufferu a zahodí ho. Přesně to potřebujeme. Přebytečný konec řádky nechceme nikam ukládat, jen se ho chceme zbavit.

Poznámka: Varianty funkce `ignore` s jedním a dvěma parametry slouží pro práci s více znaky najednou. Když funkci `ignore` voláte s jedním parametrem, zadáváte počet znaků, které se mají ze vstupního bufferu zahodit. Například příkaz `cin.ignore(80)` zahodí ze vstupního bufferu až osmdesát znaků. Varianta se dvěma parametry přidává možnost zadat znak, na kterém se má funkce zastavit. Například příkaz `cin.ignore(80, '\n')` zahodí až osmdesát znaků ze vstupního bufferu, ale pokud narazí na znak `\n`, skončí dříve.

Totéž co funkce `ignore` bez parametrů dělá i funkce `get` bez parametrů, příkazy `cin.ignore()` a `cin.get()` dělají totéž. My budeme používat funkci `ignore`, ale stejně dobře by stačilo volat `get`. Takto vypadá program rozšířený o volání `ignore`:

```
#include <iostream>
using namespace std;
int main()
{
    char znak;
    while (1)
    {
        cout << "K = konec, libovolna jina klavesa = pokracovat: ";
        cin.get(znak);
        cin.ignore();
        if (znak == 'k' || znak == 'K')
            break;
        cout << "Tak jeste jednou.\n";
    }
    cout << "Konec.\n";
    return 0;
}
```

Teď už program na tisknutelné znaky reaguje správně:

```
K = konec, libovolna jina klavesa = pokracovat: x
Tak jeste jednou.
K = konec, libovolna jina klavesa = pokracovat: k
Konec.
```

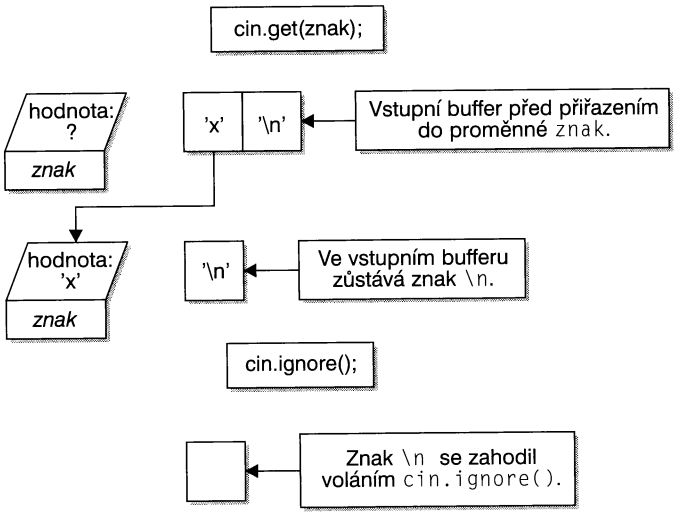
Nadbytečný znak `\n` jsme ze vstupního bufferu odstranili zavoláním členské funkce `ignore` bez jakýchkoliv parametrů, viz obrázek 12.3. Když ale chcete pokračovat stiskem klávesy Enter, musíte ji stisknout dvakrát. Napoprvé to podle všeho nefunguje:

```
K = konec, libovolna jina klavesa = pokracovat:

Tak jeste jednou.
K = konec, libovolna jina klavesa = pokracovat: k
Konec.
```

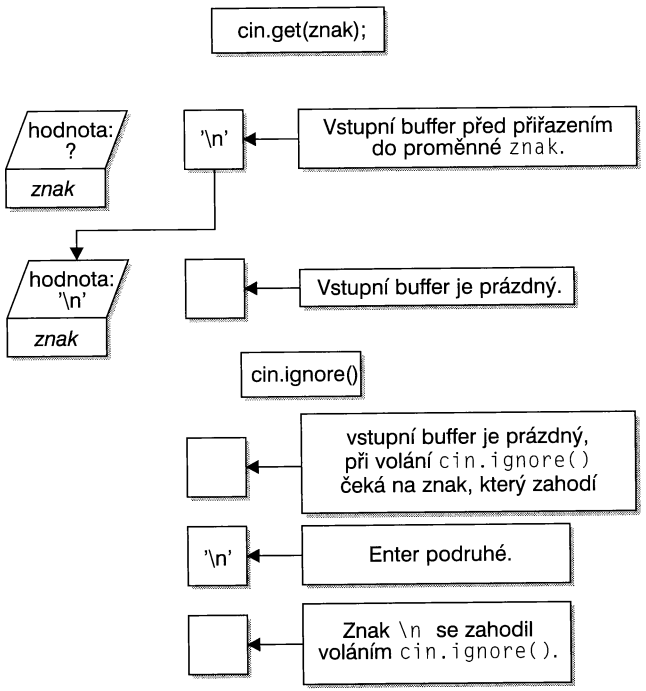
Tohle přestává být zábava. Buď nám funguje Enter, nebo nám fungují tisknutelné znaky, ale nikdy obojí najednou.

Pocity zmaru jsou u programování běžné. Důležité je se s nimi umět vypořádat, protože vytrvalost se v programování vyplácí – přeneseně i doslova. Nějaké řešení se vždycky najde, a náš případ není žádná výjimka. Nejprve je potřeba zjistit, kde přesně je problém.



Obrázek 12.3: Vstupní buffer po volání cin.ignore()

Chyba se objevila poté, co jsme přidali volání `cin.ignore()`. Tato funkce načte jeden znak ze standardního vstupu a zahodí ho. Problém je v tom, že když stisknete poprvé Enter, znak `\n` se načte voláním `cin.get(znak)`, a když přijde na řadu `cin.ignore()`, vstupní buffer už je zase prázdný, a tak se čeká na stisk další klávesy (viz obrázek 12.4).



Obrázek 12.4: Proč se Enter musí mačkat dvakrát

Jak je vidět z našich předchozích pokusů, funkce `ignore` se musí volat pouze v případech, kdy nám ve vstupním bufferu zbyl konec řádku navíc – jinými slovy pouze tehdy, když jsme funkcí `get` přečetli něco jiného než konec řádku. Pokud jsme přečetli něco jiného než konec řádku, ve vstupním bufferu zůstal konec řádku navíc a musíme ho odstranit. Pokud jsme přečetli konec řádku, vstupní buffer už je prázdný a `ignore` volat nebudeme. Jednoduše to lze vyjádřit následující podmínkou:

```
if (znak != '\n')
    cin.ignore();
```

Toto řešení ukazuje následující program. Oproti předchozí verzi volá funkci `ignore` pouze tehdy, když je načtený znak něco jiného než konec řádku:

```
#include <iostream>
using namespace std;
int main()
{
    char znak;
    while (1)
    {
        cout << "K = konec, libovolna jina klavesa = pokracovat: ";
        cin.get(znak);
        if (znak != '\n')
            cin.ignore();
        if (znak == 'k' || znak == 'K')
            break;
        cout << "Tak jeste jednou.\n";
    }
    cout << "Konec.\n";
    return 0;
}
```

Teď už program funguje nezávisle na tom, jestli uživatel stiskl Enter, nebo jestli zadal nějaký tisknutelný znak.

Problémy s koncem řádku

Problém s nadbytečným koncem řádku ve vstupním bufferu se neomezuje jen na programy, které chtějí zpracovat stisk libovolné klávesy. Narazíte na něj například i v programech, které používají čtení z `cin` operátorem `>>` a zároveň volají funkci `cin.get` nebo `cin.getline`:

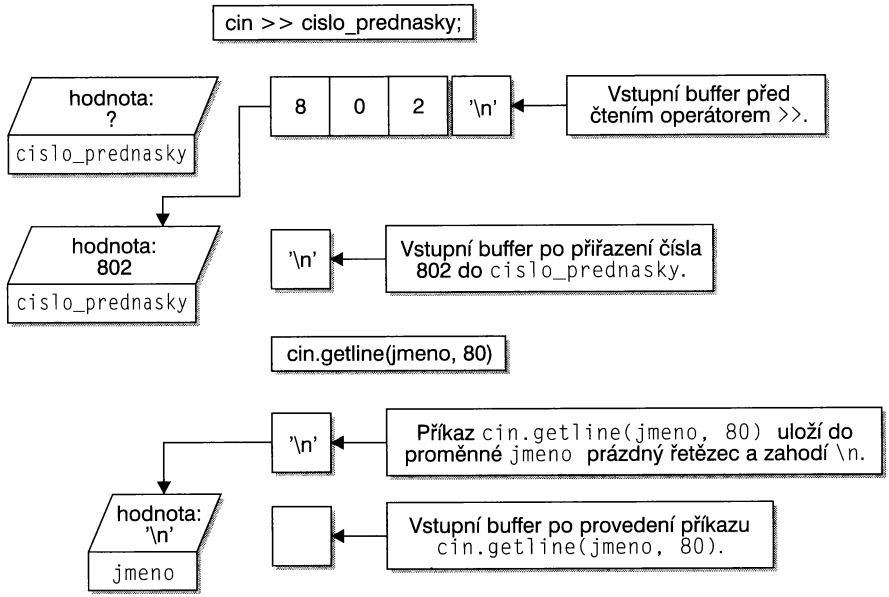
```
#include <iostream>
using namespace std;
int main()
{
    char jmeno[80];
    int cislo_prednasky;
    cout << "Zadejte cislo prednasky: ";
    cin >> cislo_prednasky;
    cout << "Zadejte sve jmeno: ";
    cin.getline(jmeno, 80);
    cout << "Cislo prednasky: " << cislo_prednasky << "\n";
    cout << "Vase jmeno: " << jmeno << "\n";
    return 0;
}
```

Vstup a výstup programu vypadá takto:

```
Zadejte cislo prednasky: 802
Zadejte sve jmeno: Cislo prednasky: 802
Vase jmeno:
```

Vůbec jsme nedostali možnost zadat jméno. Důvod je podobný jako u předchozího programu, kde po prvním stisku klávesy Enter nešlo zadat písmenko.

Jak je vidět na obrázku 12.5, po zadání čísla 802 a stisknutí klávesy Enter se ve vstupním bufferu objeví nejen znaky `802`, ale i konec řádku zadaný klávesou Enter. Operátor `>>` přečte ze vstupního bufferu všechno až po konec řádku, ale samotný konec řádku v bufferu zůstane. Právě to nám dělá problémy. Další totiž přijde na řadu čtení funkcí `cin.getline`, která čte všechny znaky až po oddělovač `\n`. Jelikož vstupní buffer není prázdný, funkce `getline` na nic nečeká a rovnou začne číst. Hned na začátku narazí na oddělovač `\n`, a tak do řetězce `jmeno` uloží pouze prázdný řetězec a oddělovač ze vstupního bufferu odstraní.



Obrázek 12.5: Čtení operátorem >> a funkcí cin.getline

Řešení je stejné jako v předchozí části kapitoly, stačí za čtení operátorem `>>` přidat volání `cin.ignore()` nebo `cin.get()`. Žádnou podmínku tentokrát nepotřebujeme, protože po čtení operátorem `>>` z `cin` zůstane znak `\n` ve vstupním bufferu vždy. Takto vypadá nová verze programu, která nadbytečný konec řádku uklízí pomocí `cin.ignore()`:

```
#include <iostream>
using namespace std;
int main()
{
    char jmeno[80];
    int cislo_prednasky;
    cout << "Zadejte cislo prednasky: ";
```

```
    cin >> cislo_prednasky;
    cin.ignore();
    cout << "Zadejte sve jmeno: ";
    cin.getline(jmeno, 80);
    cout << "Cislo prednasky: " << cislo_prednasky << "\n";
    cout << "Vase jmeno: " << jmeno << "\n";
    return 0;
}
```

Jak je vidět z následujícího výstupu, program už funguje správně:

```
Zadejte cislo prednasky: 802
Zadejte sve jmeno: Jeff Kent
Cislo prednasky: 802
Vase jmeno: Jeff Kent
```

Užitečná pravidla

Problémům s nadbytečným koncem řádku ve vstupním bufferu se můžete vyhnout dodržováním následujících tří jednoduchých pravidel. (Samozřejmě za předpokladu, že konec řádku ve vstupním bufferu nenecháváte záměrně.)

1. Po čtení operátorem `>>` z `cin` vždy zavolejte `cin.ignore()`.

   Po operátoru `>>` zůstane ve vstupním bufferu znak `\n`, voláním `cin.ignore()` ho odstraníte. Například takhle:

   ```
   char znak;
   cin >> znak;
   cin.ignore();
   ```

2. Za voláním `cin.getline` už `cin.ignore()` nevolejte.

   Funkce `cin.getline` odstraní závěrečný konec řádky ze vstupního bufferu, takže by funkce `ignore` neměla co zahazovat a čekala by na další vstup.

3. Když čtete jeden znak funkcí `cin.get`, například `cin.get(znak)`, zkontrolujte si, jestli ve vstupním bufferu ještě nezůstal konec řádku. Pokud ano, zahoďte ho funkcí `ignore`.

   Když funkcí `cin.get` načtete obyčejný znak, ve vstupním bufferu zůstane konec řádky vložený klávesou Enter. Když jen zmáčknete Enter, funkce `cin.get` vrátí znak `\n` a vstupní buffer bude prázdný. Proto si musíte zkontrolovat, jestli je vstupní buffer potřeba vymazat:

   ```
   char znak;
   cin.get(znak);
   if (znak != '\n')
       cin.ignore();
   ```

Užitečné funkce pro práci se znaky

Soubor `cctype` ze standardní knihovny C++ definuje řadu užitečných znakových funkcí. Dvě z nich se starají o změnu velikosti z malých písmen na velká a naopak, ostatní slouží ke klasifikaci znaků.

Změna velikosti písmen

Pro změnu velikosti znaku slouží funkce `toupper` a `tolower`. Obě funkce mají jeden parametr typu `char` a vrací hodnotu typu `char`. Pokud je zadaný znak písmeno abecedy, funkce `toupper` ho převede na velké a funkce `tolower` na malé. Pokud znak není písmeno, obě funkce ho vrátí beze změny.

Obě funkce jsou užitečné. Například následující program z předchozí části kapitoly potřebuje vědět, jestli jste zadali malé nebo velké písmeno K:

```
#include <iostream>
using namespace std;
int main()
{
    char znak;
    while (1)
    {
        cout << "K = konec, libovolna jina klavesa = pokracovat: ";
        cin.get(znak);
        if (znak != '\n')
            cin.ignore();
        if (znak == 'k' || znak == 'K')
            break;
        cout << "Tak jeste jednou.\n";
    }
    cout << "Konec.\n";
    return 0;
}
```

Funkcí `toupper` si můžeme ušetřit samostatnou kontrolu malého k:

```
#include <iostream>
#include <cctype>
using namespace std;
int main()
{
    char znak;
    while (1)
    {
        cout << "K = konec, libovolna jina klavesa = pokracovat: ";
        cin.get(znak);
        znak = toupper(znak);
        if (znak != '\n')
            cin.ignore();
        if (znak == 'K')
            break;
        cout << "Tak jeste jednou.\n";
    }
    cout << "Konec.\n";
    return 0;
}
```

Přestože podmínka pracuje jen s velkým K, program se správně ukončí i po stisku malého k:

```
K = konec, libovolna jina klavesa = pokracovat: k
Konec.
```

První změnou oproti původní verzi programu je odkaz na soubor `cctype` ze standardní knihovny, který přibyl pod odkazem na `iostream`. Funkce `toupper` a `tolower` totiž jsou definované v souboru `cctype`, nikoliv v `iostream`. Druhou změnou je samotné volání `toupper`:

```
znak = toupper(znak);
```

Jako parametr funkci předáme proměnnou `znak`, do které jsme načetli vstup od uživatele. Návratovou hodnotu funkce pak uložíme do téže proměnné. Na tento krok nesmíte zapomenout – funkce `toupper` nemění hodnotu svého parametru, výsledné písmeno vrací jako svou návratovou hodnotu.

Poznámka: Stejně dobře jsme mohli volat funkci `tolower` a v podmínce pak znak srovnávat s malým k.

Klasifikace znaků

Soubor `cctype` dále definuje několik funkcí, kterými se dá zjistit, jestli je zadaný znak písmeno, číslice, mezera a podobně. Každá z těchto funkcí bere jeden parametr typu `char` a vrací booleovskou hodnotu `true` nebo `false`, viz tabulku 12.1.

Tabulka 12.1: Funkce pro klasifikaci znaků

Funkce	Pro jaké hodnoty vrací hodnotu true
`isalpha`	písmena abecedy
`isalnum`	písmena abecedy a číslice
`isdigit`	číslice
`islower`	malá písmena abecedy
`isprint`	libovolné tisknutelné znaky včetně mezery
`ispunct`	všechny tisknutelné znaky s výjimkou mezery, číslic a písmen
`isupper`	velká písmena abecedy
`isspace`	mezera a podobné znaky, například tabulátor nebo konec řádku

Těmito funkcemi se dobře kontroluje uživatelský vstup. Například následující kus kódu ukazuje funkci pro kontrolu tříznakových hesel – vyhovují pouze hesla, která začínají velkým písmenem, uprostřed mají číslo a končí malým písmenem (například „Z3s“).

```
bool heslo_vyhovuje(char* heslo)
{
    if (!isupper(heslo[0]))
        return false;
    if (!isdigit(heslo[1]))
        return false;
    if (!islower(heslo[2]))
        return false;
    return true;
}
```

Užitečné funkce pro práci s řetězci

Soubor `cstring` ze standardní knihovny definuje řadu užitečných funkcí pro práci s céčkovými řetězci, a podobně soubor `string` definuje funkce pro práci s řetězci typu `string`. Popis několika z těchto funkcí najdete v následujícím textu.

Jak zjistit délku řetězce

Délka céčkového řetězce se dá zjistit funkcí `strlen`. Funkce má jeden parametr, kterým může být znakové pole (tedy ukazatel na znak) nebo řetězcový literál. Funkce vrací `int` s délkou zadaného řetězce, koncový znak `\0` nepočítaje. Její použití ukazuje následující kus kódu:

```
int delka;
delka = strlen("Jeff"); // 4
char* smrduty_pes = "Dante";
delka = strlen(smrduty_pes); // 5
char jmeno[80] = "Devvie";
delka = strlen(jmeno); // 6
char sileny_pes[80] = "Michaela";
delka = strlen(sileny_pes); // 7
```

Následující funkce ukazuje, jak se dá pomocí funkcí `strnlen` a `isdigit` zkontrolovat číslo sociálního pojištění, které musí mít přesně jedenáct znaků v sestavě tři čísla a pomlčka, dvě čísla a pomlčka a čtyři čísla:

```
bool zkontroluj_cislo_pojisteni(char *vstup)
{
    if (strlen(vstup) != 11)
        return false;
    for (int i=0; i<11; i++)
    {
        if (i == 3 || i == 6)
            if (vstup[i] != '-')
                return false;
        else if (!isdigit(vstup[i]))
                return false;
    }
    return true;
}
```

U řetězců typu `string` se místo funkce `strlen` používají členské funkce `length` a `size`. Ani jedna z nich nemá žádné parametry, obě vrací délku řetězce:

```
string s = "Jeff Kent";
cout << s.length(); // 9
cout << s.size();   // 9
```

Přiřazování řetězců

Céčkové řetězce se navzájem nedají přiřazovat jednoduše operátorem =, například takto:

```
char* cil = "Jeff Kent";
char zdroj[] = "Michaela";
cil = zdroj;
```

Přesněji řečeno se přiřazovat dají, ale výsledek bude asi jiný, než byste čekali. Hodnotou proměnné `zdroj` je adresa příslušného řetězce v paměti. Přiřazení `cil = zdroj` tedy nezkopíruje do proměnné `cil` hodnotu řetězce `zdroj`, ale pouze adresu tohoto řetězce. V čem přesně je to problém ukazuje následující příklad:

```
char* cil = "pif";
char zdroj[] = "paf";
cil = zdroj;
zdroj[1] = 'u';
cout << cil << ", " << zdroj; // puf, puf
```

Místo dvou samostatných řetězců se stejnou hodnotou jsme dostali dva ukazatele na jeden řetězec. Pokud chcete vytvořit dva nezávislé řetězce, musíte sáhnout po funkci `strcpy`. Jejím prvním parametrem je cílový řetězec, druhým parametrem je řetězec zdrojový. První parametr nesmí být literál, protože do literálu se nic zkopírovat nedá. Podívejte se, jak předchozí příklad dopadne po zkopírování řetězce funkcí `strcpy`:

```
char cil[] = "pif";
char zdroj[] = "paf";
strcpy(cil, zdroj);
zdroj[1] = 'u';
cout << cil << ", " << zdroj; // paf, puf
```

Tentokrát už v paměti máme dva nezávislé řetězce.

Při použití funkce `strcpy` si musíte dávat pozor, aby zdrojový řetězec nebyl delší než ten cílový. Jistější je sáhnout raději po funkci `strncpy`, která nabízí ještě třetí parametr pro maximální počet kopírovaných znaků:

```
char* cil = "Jeff Kent";
char zdroj[] = "Michaela";
strncpy(cil, zdroj, 9);
```

Naproti tomu řetězce typu `string` se dají přiřazovat běžně operátorem =, viz následující příklad:

```
string cil = "Jeff Kent";
string zdroj = "Michaela";
cil = zdroj;
zdroj = "Dante";
cout << cil; // Michaela
```

Spojování řetězců

Funkce `strcpy` přepíše obsah cílového řetězce zdrojovým. Někdy byste ale chtěli původní řetězec zachovat a nový řetězec připojit na jeho konec; k tomu slouží funkce `strcat`. Ta má stejně jako `strcpy` dva parametry, zdrojový a cílový řetězec. Cílový řetězec opět nesmí být literál, zdrojový řetězec může. Použití funkce `strcat` ukazuje následující kód:

```
char cil[80] = "Jeff";
char* zdroj = " Kent";
strcat(cil, zdroj);
cout << cil; // Jeff Kent
```

Řetězce typu `string` se dají spojovat jednoduše operátorem +, případně jeho spojením s operátorem přiřazení (+=). Viz následující příklad:

```
string cil = "Jeff";
string zdroj = " Kent";
cil += zdroj;
cout << cil; // Jeff Kent
```

Porovnání řetězců

Céčkové řetězce se nedají porovnávat klasickými relačními operátory. Přesněji řečeno toto porovnání dělá něco jiného, než byste si mysleli. Podívejte se na následující příklad:

```
char jmeno1[] = "Devvie Kent";
char jmeno2[] = "Devvie Kent";
cout << "Zadane retezce jsou ";
cout << ((jmeno1 == jmeno2) ? "stejne.\n" : "ruzne.\n");
```

Program bude za všech okolností tvrdit, že jsou řetězce různé. Neporovnává totiž řetězce, ale hodnoty proměnných `jmeno1` a `jmeno2`, tedy adresy. I když oba řetězce obsahují stejnou hodnotu, v paměti jsou uložené jinde, takže porovnání adres vyjde nepravdivé.

K porovnání hodnot dvou céčkových řetězců slouží funkce `strcmp`. Má dva parametry, pro dva srovnávané řetězce. Pokud jsou řetězce shodné, funkce vrátí nulu:

```
char jmeno1[] = "Devvie Kent";
char jmeno2[] = "Devvie Kent";
if (strcmp(jmeno1, jmeno2) == 0)
    cout << "Retezce jsou shodne.\n";
else
    cout << "Neco je spatne.\n";
```

Podmínku `if` bychom mohli zjednodušit:

```
if (!strcmp(jmeno1, jmeno2))
```

Nula se totiž bere jako `false`, takže negací (`!`) dostaneme `true`.

Pokud první řetězec patří podle abecedy výš než druhý, funkce `strcmp` vrátí zápornou hodnotu (řetězec je „menší"), a pokud naopak první řetězec patří podle abecedy až za druhý, funkce vrátí kladnou hodnotu. Porovnání se provádí srovnáním ASCII kódů jednotlivých znaků obou řetězců – začne se prvními znaky a postupuje se doprava, dokud znaky nejsou různé. Výsledky několika ukázkových porovnání ukazuje tabulka 12.2.

Tabulka 12.2: Výsledky porovnání řetězců

První řetězec	Druhý řetězec	strcmp	Vysvětlení
`Jeff`	`jeff`	–	Velká písmena jsou v ASCII před malými.
`aZZZ`	`Zaaa`	+	Malé `a` je v ASCII až po `Z`.
`salat`	`salam`	+	První čtyři znaky jsou stejné, ale `t` je v ASCII až za `m`.
`Jeff`	`Jeffrey`	–	První čtyři znaky jsou stejné, ale pátý znak druhého řetězce je v ASCII až za `\0`.

Jelikož strcmp vrací tři různé hodnoty, často se používá ve spojení s příkazem `if/else if/else`:

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

int main()
{
    char str1[80], str2[80];
    cout << "Zadejte prvni retezec: ";
    cin >> str1;
    cout << "Zadejte druhy retezec: ";
    cin >> str2;
    int vysledek = strcmp(str1, str2);
    if (vysledek == 0)
        cout << "Retezce jsou stejne.\n";
    else if (vysledek > 0)
        cout << "Prvni retezec je vetsi.\n";
    else
        cout << "Druhy retezec je vetsi.\n";
    return 0;
}
```

Následuje několik ukázek vstupu a výstupu:

```
Zadejte prvni retezec: Jeff
Zadejte druhy retezec: Jeff
Retezce jsou stejne.
----
Zadejte prvni retezec: Jeff
Zadejte druhy retezec: jeff
Druhy retezec je vetsi.
----
Zadejte prvni retezec: salat
Zadejte druhy retezec: salam
Prvni retezec je vetsi.
----
Zadejte prvni retezec: Jeff
Zadejte druhy retezec: Jeffrey
Druhy retezec je vetsi.
```

Naproti tomu řetězce typu `string` se dají srovnávat běžnými relačními operátory. Jestli je řetězec „větší" než jiný, se určuje podle stejných pravidel jako u céčkových řetězců:

```
string str1 = "Jeff";
string str2 = "jeff";
string str3 = "Jeffrey";
str1 < str2; // true
str3 > str1; // true
str2 > str3; // true
```

Převod řetězce na číslo

Ve standardním knihovním souboru `cstdlib` je definovaných několik funkcí pro převod céčkových řetězců na čísla a naopak. Jedna z nejdůležitějších je funkce `atoi`, jejíž název je zkratkou pro „ASCII to integer", tedy převod čísel z ASCII podoby na skutečné číslo. Funkce má jen jeden

parametr (céčkový řetězec) a vrací číselnou hodnotu tohoto řetězce jako `int`. Její použití v praxi ukazuje následující příkaz:

```
int cislo = atoi("7654");
```

Funkce `atoi` nekontroluje, jestli se zadaný řetězec dá převést. Jazyk C++ navíc ani nedefinuje chování funkce na řetězcích, které nejsou zápisem žádného čísla. Například hodnotou výrazu `atoi("12Jeff")` by mohla být dvanáctka – funkce by zkrátka převedla, co se převést dá. Stejně dobře ale takové řetězce nemusí převádět vůbec, může vrátit nulu jako chybový kód.

Běžně se funkce `atoi` používá tak, že uživatelský vstup načtete do znakového pole, ručně si zkontrolujete, jestli jde o číslo, a pak teprve zavoláte `atoi`:

```
#include <iostream>
#include <cstring>
using namespace std;
bool zkontroluj(char*);

int main()
{
    char vstup[80];
    cout << "Zadejte cele cislo: ";
    cin >> vstup;
    if (zkontroluj(vstup))
    {
        int cislo = atoi(vstup);
        cout << "Zadali jste " << cislo << ".\n";
    }
    else
        cout << "Nezadali jste cele cislo.\n";
    return 0;
}

bool zkontroluj(char* vstup)
{
    for (int i=0; i<strlen(vstup); i++)
    {
        if (i == 0 && vstup[i] == '-' && strlen(vstup) > 1)
            continue;
        if (!isdigit(vstup[i]))
            return false;
    }
    return true;
}
```

Načítání čísla do řetězce namísto číselného typu má výhodu v tom, že když uživatel zadá něco jiného než číslo, program nespadne ani nezačne vypisovat nesmysly. Hodnota řetězce se kontroluje předem, jestli obsahuje číslo. Jelikož číslo může být i záporné, řetězec může začínat spojovníkem (`-`). Teprve po úspěšné kontrole řetězce se vstup pomocí `atoi` převede na číslo a uloží do číselné proměnné.

Podobně jako `atoi` pracují funkce `atol` („ASCII to long“) a `atof` („ASCII to float“), které převádí řetězcový zápis čísel typu `long` a `float`; jejich návratové hodnoty jsou logicky typu `long` a `float`.

Třída `string` nemá žádnou členskou funkci, která by uměla totéž co `atoi`, `atol` nebo `atof`. Nabízí ale členskou funkci `c_str`, která vrátí odpovídající céčkový řetězec, takže pak už můžete snadno použít například `atoi`:

```
string vstup = "123";
cout << atoi(vstup.c_str()) + 1; // 124
```

Některé překladače podporují funkci `itoa` („integer to ASCII"), která je protipólem `atoi`. Funkce `itoa` má tři parametry. Prvním z nich je číslo, které chcete převést. Jako druhý parametr se uvádí céčkový řetězec, do kterého chcete výsledek uložit, a třetí parametr je základ číselné soustavy, ve které chcete číslo zapsat. Následující kus kódu ukazuje, jak do řetězce uložit zápis čísla 776 v běžné desítkové soustavě:

```
char vystup[10];
itoa(776, vystup, 10);
```

Shrnutí

V této kapitole jsme se věnovali třem tématům: znakům, céčkovým řetězcům a třídě `string`. Nejprve jsme se zabývali čtením znaků, u kterého může být problém se znakem konce řádku zapomenutým ve vstupním bufferu. Abychom se tomuto problému vyhnuli, přišli jsme s následujícími třemi pravidly:

1. Kdykoliv čtete z `cin` operátorem >>, zavolejte následně funkci `cin.ignore()`. Po čtení operátorem >> totiž ve vstupním bufferu zůstane konec řádku, který se voláním členské funkce `ignore` odstraní.
2. Po čtení pomocí funkce `cin.getline` už `cin.ignore` nevolejte. Funkce `cin.getline` po sobě konec řádku ve vstupním bufferu nenechává.
3. Pokud čtete jeden znak funkcí `cin.get`, podívejte se, jestli vám ve vstupním bufferu nezůstal konec řádku. Pokud ano, odstraňte ho funkcí `cin.ignore`. Po funkci `cin.get` totiž konec řádku ve vstupním bufferu zůstane jen někdy – když uživatel zadá tisknutelný znak a potvrdí ho klávesou Enter, konec řádku v bufferu zůstane. Pokud ale žádný znak nezadá a stiskne rovnou Enter, vstupní buffer zůstane prázdný.

Řadu užitečných funkcí pro práci se znaky definuje soubor `cctype` ze standardní knihovny C++. Funkce `toupper` a `tolower` mají každá jeden parametr typu `char`. Když jim zadáte nějaké písmeno abecedy, fukce `toupper` ho převede na velké písmeno a funkce `tolower` na malé. Pokud parametr není písmeno, funkce ho vrátí beze změn. Další funkce, například `isalpha` nebo `isdigit`, slouží ke klasifikaci znaků. Řeknou vám například, jestli je zadaný znak písmeno, číslice, mezera a podobně.

Soubor `cstrings` ze standardní knihovny definuje několik funkcí pro práci s céčkovými řetězci. K podobnému účelu slouží soubor `string`, který definuje funkce pro práci s řetězcovou třídou `string` jazyka C++.

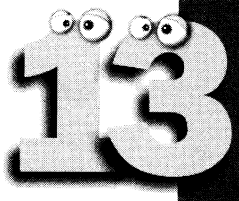
Délku céčkového řetězce vám prozradí funkce `strlen`, délku řetězce typu `string` vrací členská funkce `length` nebo `size`. Céčkové řetězce se kopírují funkcí `strcpy`, řetězc typu `string` se dají kopírovat jednoduše operátorem =. Ke spojení dvou céčkových řetězců slouží funkce `strcat`. Řetězce typu `string` to mají opět jednodušší, dají se spojovat operátorem +. Porovnání dvou céčkových řetězců řeší funkce `strcmp`; řetězce typu `string` se dají porovnávat operátorem ==.

Knihovna `cstdlib` definuje několik funkcí pro převod řetězců na čísla a nazpět, například funkci `atoi` pro převod řetězce na `int`, `atol` pro převod na `long`, `atof` pro převod na `float` a `itoa` pro převod celého čísla zpět na řetězec.

Test

1. Který z následujících způsobů čtení ze standardního vstupu ignoruje úvodní znak konce řádku: operátor >>, funkce `cin.get`, nebo funkce `cin.getline`?
2. Který z následujících způsobů čtení ze standardního vstupu nenechá ve vstupním bufferu znak konce řádku: operátor >>, funkce `cin.get`, nebo funkce `cin.getline`?
3. Po kterém z následujících zůsobů čtení byste měli vždy zavolat `cin.ignore()`: po operátoru >>, funkci `cin.get`, nebo funkci `cin.getline`?
4. Po kterém z následujících způsobů čtení byste `cin.ignore()` volat neměli: po operátoru >>, funkci `cin.get`, nebo funkci `cin.getline`?
5. Jakého typu je jediný parametr funkce `isdigit`? Jde o `char`, céčkový řetězec, nebo řetězec typu `string`?
6. Jakého typu je jediný parametr funkce `atoi`? Jde o `char`, céčkový řetězec, nebo řetězec typu `string`?
7. Dá se funkcí `atoi` převést řetězec typu `string` na číslo?
8. Funkce definované v souboru `cctype` slouží pro práci se znaky, céčkovými řetězci, nebo objekty typu `string`?
9. Funkce definované v souboru `cstdlib` slouží pro práci se znaky, céčkovými řetězci, nebo objekty typu `string`?
10. Dá se operátorem = přiřadit hodnota jednoho céčkového řetězce druhému?

Trvalé uložení dat aneb Soubory



Jako dítě jsem musel trpělivě poslouchat nekonečné rady svých rodičů ohledně toho, v čem všem bych se mohl zlepšit. Když jsem dospěl, ke svému úžasu jsem zjistil, že rodiče měli většinou pravdu. A když jsem se stal rodičem, ke své hrůze jsem jejich rady začal opakovat svým vlastním dětem, které si mé přednášky užívají zhruba stejně jako kdysi já.

Mí rodiče s oblibou hovořili o tom, jak důležité je vydržet. Tady se opět ukazuje, jak velkou měli pravdu, protože pokud někdo potřebuje výdrž, pak jsou to především dálkoví běžci a programátoři. Vydržet ovšem nepotřebují jen programátoři, vydržet musí také data. Nemůže se stát, že by data zmizela s ukončením programu. Dovedete si představit, že bych dopsal tuto kapitolu, ukončil Word a veškerý text by zmizel?

Přesně to by se ale stalo se všemi programy, které jsme doposud psali. Obsah proměnných nepřežije konec programu. Uchovává se totiž v operační paměti, kterou systém po skončení programu nabídne k použití ostatním aplikacím a jejíž obsah se dokonale ztratí po vypnutí počítače.

Naštěstí Word (a samozřejmě i většina ostatních programů) nabízí možnost uložit data na pevný disk nebo jiné médium, ze kterého se dají kdykoliv podle potřeby zase načíst. Zde už data zůstanou i po skončení programu a vypnutí počítače.

V této kapitole si ukážeme, jak se data dají trvale uchovat uložením do souboru. Dá rozum, že kdybychom je neuměli načíst nazpět, příliš bychom si nepomohli. Proto se v kapitole budeme věnovat i načítání dat ze souborů.

Přehled

Pokud pracujete na počítači, pracujete se soubory. Přesněji řečeno s desítkami, ne-li stovkami a tisíci souborů. Zastavili jste se ale někdy nad tím, co přesně soubor je?

Soubor je samostatný shluk dat umístěný v nějaké trvalé paměti, například na pevném disku nebo CD. (Typům paměti jsme se podrobně věnovali ve druhé kapitole.) Každý soubor má nějaké jméno, které popisuje jeho obsah – například dokument s touto kapitolou by se mohl jmenovat *kapitola13*.

Součástí názvu souboru bývá takzvaná přípona, několik písmen oddělených od názvu souboru tečkou. Například soubor s touto kapitolou se jmenuje *kapitola13.doc* a má příponu *.doc*. Přípona určuje typ souboru a většinou i program, který s příslušným typem souborů umí zacházet. Například *.doc* je přípona souborů otevíraných Wordem, *.xls* přípona souborů otevíraných Excelem a podobně. Při práci s touto knihou jste si zřejmě zvykli na příponu *.cpp*, kterou mívají soubory se zdrojovými kódy v jazyce C++.

Textové a binární soubory

Programů existuje nepřeberně, takže i typů souborů a různých přípon je moc. Všechny soubory se ale dají v principu rozdělit do dvou kategorií, na textové a binární. Textový soubor, jak už jeho jméno napovídá, obsahuje text. Do textových souborů ukládá data například Poznámkový blok a jiné čistě textové editory.

U binárních souborů je situace o něco složitější. Zkuste si dokument napsaný ve Wordu otevřít v Poznámkovém bloku nebo jiném čistě textovém editoru (v jakém jsem například já psal tuto kapitolu). Pokud vůbec uvidíte nějaký text, bude obklopený prapodivnými znaky, které do samotného textu rozhodně nepatří. To jsou formátovací informace, které Word uložil společně s textem – například informace o tabulkách, odrážkovaných a číslovaných seznamech a podobně.

Textové soubory mohou obsahovat pouze text. Naproti tomu do binárních souborů se dají uložit i další typy informací, například obrázky, databázové záznamy, spustitelné programy a podobně. Složitější programy typu Word, Excel nebo Access proto data ukládají do binárních souborů.

S textovými soubory se v mnoha ohledech pracuje lépe než s binárními, a tak se zpracování souborů většinou vysvětluje nejdříve na nich. Jelikož je tato kniha pouze úvod do C++, do binárních souborů se vůbec nebudeme pouštět. Pokud se ale text vztahuje na textové i binární soubory bez rozdílu, zmíním se o tom.

Souborové datové proudy

Téměř od samotného začátku knihy pracujeme s knihovním souborem `iostream`, který kromě jiného definuje objekty `cin` pro čtení ze standardního vstupu (většinou tedy klávesnice) a `cout` pro zápis na standardní výstup (obvykle monitor).

Čtení souborů a zápis do nich vyžaduje další knihovní soubor, `fstream`. Ze zdrojového kódu se na něj odkážete tradičně:

```
#include <fstream>
```

Oba knihovní soubory mají společné slovo *stream*, proud. Není to náhoda, oba definují objekty pro práci s proudy bajtů. Zatímco soubor `iostream` se zabývá proudy bajtů ze standardního vstupu a výstupu (*input/output*, odtud `io`), soubor `fstream` definuje objekty pro čtení proudů ze souborů a zápis do nich („soubor" se anglicky řekne *file*, proto `f`).

Soubor `fstream` definuje tři nové datové typy:

- `ofstream` pro výstup do souborů (*output*). Pomocí proměnné typu `ofstream` můžete vytvořit soubor a něco do něj zapsat, ale nikoliv z něj číst.
- `ifstream` pro vstup ze souborů (*input*). Pomocí proměnné typu `ifstream` můžete přečíst informace ze souboru, ale nic do něj nezapíšete.
- `fstream` pro čtení a zápis souborů, tedy kombinace `ofstream` a `ifstream`. Prostřednictvím proměnné typu `fstream` se soubory dají zapisovat i číst.

Životní cyklus práce se souborem

Práce se souborem – ať už čtení, zápis, nebo obojí – vždy probíhá v následujících třech krocích:

1. Soubor otevřete. Tím se vytvoří komunikační cesta mezi samotným souborem a objektem typu `ofstream`, `ifstream` nebo `fstream`, jehož prostřednictvím se souborem pracujete.

2. Čtete a zapisujete data. My se v textu budeme zabývat nejprve zápisem a teprve potom čtením, ale v programu může být pořadí operací samozřejmě libovolné. Klidně můžete napsat i program, který bude soubory používat jen jedním způsobem, tedy jen z nich číst nebo jen zapisovat.
3. Soubor zavřete. Udržování komunikační cesty mezi souborem a objektem vyžaduje určité systémové prostředky, které uzavřením souboru uvolníte. Kdybyste soubor nezavřeli, nemuseli byste se k němu sami v jiné části programu dostat.

Otevření souboru pro zápis

Pro zápis se soubor dá otevřít objektem typu `ofstream` nebo `fstream`. (Typ `ifstream` vám při zápisu nepomůže, je určený pouze pro čtení.) Oba typy nabízí dva způsoby otevření souboru – prvním je členská funkce pojmenovaná nepřekvapivě `open`, druhým je konstruktor.

Členská funkce open

Datové typy `ofstream` i `fstream` mají členskou funkci `open`, jejímž parametrem je cesta k otevíranému souboru. Pokud chcete, můžete uvést ještě druhý parametr, kterým se nastavuje režim práce se souborem.

Parametr první – cesta k souboru

Soubor otevíraný pro zápis nemusí existovat. Pokud neexistuje, otevřením ho na dané cestě vytvoříte. Cestu ovšem musíte uvést v každém případě; může být zadaná *relativně* nebo *absolutně*. Tyto výrazy jsou pro vás možná nové, a tak se u nich zastavíme.

Relativní cesta je cesta vzhledem k umístění vašeho programu. Například následující kód by pro zápis otevřel soubor `studenti.dat` z adresáře, ve kterém je váš program:

```
ofstream soubor;
soubor.open("studenti.dat");
```

Naproti tomu absolutní cesta začíná vždy od kořene souborového systému. Například kdybych chtěl pro zápis otevřít soubor `studenti.dat` uložený v podadresáři `Přednášky`, který se nachází v adresáři `Škola` na disku C, napsal bych následující příkazy:

```
ofstream soubor;
soubor.open("C:\\Skola\\Prednasky\\studenti.dat");
```

Poznámka: Samotné zpětné lomítko by se v řetězci bralo jako začátek speciálního znaku. Dvojité zpětné lomítko je speciální znak pro obyčejné zpětné lomítko.

Absolutní i relativní cestu samozřejmě nemusíte zadávat jen řetězcovým literálem. Můžete použít proměnnou, viz následující kód:

```
ofstream soubor;
char[80] jmeno;
cout << "Zadejte nazev souboru: ";
cin >> jmeno;
soubor.open(jmeno);
```

Poznámka: Obecně vzato je lepší dávat přednost relativním cestám, zvlášť pokud program poběží na různých počítačích. Zatímco umístění datových souborů vzhledem k programu bývá stejné, jejich absolutní cesta může být na každém počítači jiná.

Parametr druhý – režim práce se souborem

Druhý parametr členské funkce `open` určuje režim práce se souborem. Můžete si například zvolit, jestli budete ze souboru jen číst, jestli do něj budete i zapisovat, nebo jestli potřebujete obojí. Tím výběr nekončí, všechny možnosti ukazuje tabulka 13.1.

Tabulka 13.1: Nastavení režimu práce se souborem

Příznak	Popis
`ios::app`	Režim *append*, tedy připisování na konec souboru. Stávající obsah souboru zůstane beze změn, veškerý výstup se přidává na konec souboru.
`ios::ate`	Po otevření souboru se přejde na jeho konec.
`ios::binary`	Binární režim. Informace se do souboru zapisují v binární, nikoliv ve výchozí textové podobě.
`ios::in`	Otevřít pro čtení. Pokud soubor neexistuje, nový se nevytvoří.
`ios::out`	Otevřít pro zápis. Pokud už soubor existuje, jeho stávající obsah se zahodí.
`ios::trunc`	Pokud už soubor existuje, jeho stávající obsah se zahodí. Výchozí chování režimu `ios::out`.

Pokud soubor otevíráte prostřednictvím proměnné typu `ofstream`, už není nutné uvádět příznak `ios::out` – viz naše předchozí příklady, ve kterých jsme vždy uváděli jen první parametr. Proměnnou typu `ofstream` se totiž soubor dá otevřít pouze pro zápis, nikoliv pro čtení, a tak by příznak `ios::out` byl nadbytečný.

Ostatní příznaky ale nastavit můžete, například pokud chcete přepnout výchozí textový režim na binární nebo když chcete data k souboru přidávat, a nikoliv přepisovat jeho stávající obsah. Připisování na konec souboru se hodí například u protokolů o běhu programu. Když chcete do protokolu zapsat novou informaci, určitě nechcete smazat všechny ostatní, chcete nový záznam přidat na konec souboru.

Příznaky se dají navzájem kombinovat takzvaným *bitovým součtem* (`|`), například následující kód otevře soubor v binárním režimu a místo přepisu zapne přidávání na konec:

```
ofstream soubor;
soubor.open("studenti.dat", ios::binary | ios::app);
```

Upozornění: *Bitový součet* (`|`) je něco jiného než *logický součet* (`||`), ačkoliv se oba operátory podobně jmenují a oba obsahují znak `|`.

Zatímco u proměnných typu `ofstream` se bez nastavení režimu obejdete, u typu `fstream` byste ho nastavit měli. Typ `ofstream` slouží výhradně pro zápis do souboru, takže u něj je výchozí režim jasný. Typ `fstream` se ale dá použít i pro čtení, a tak musíte při volání funkce `open` určit, jestli budete ze souboru jen číst, jestli do něj budete jen zapisovat, nebo jestli potřebujete obojí. Následující kód otevře soubor prostřednictvím `fstream` jen pro zápis:

```
fstream soubor;
soubor.open("studenti.dat", ios::out);
```

Konstruktory typů fstream a ofstream

Soubor můžete pro zápis otevřít i pomocí konstruktorů datových typů `fstream` a `ofstream`. *Konstruktor* je funkce, která se automaticky volá při vytvoření nové *instance* datového typu. Instance je něco jako proměnná u primitivního datového typu, například typu `int`. Například následující příkaz by se dal brát jako vytvoření instance typu `int`:

```
int vek;
```

Podobně následující příkaz vytvoří instanci typu `fstream`:

```
fstream soubor;
```

Konstruktory se dají přetěžovat – objekt může mít konstruktor bez parametrů, s jedním parametrem, se dvěma parametry a tak dále. V předchozím příkladu se například automaticky zavolal konstruktor typu `fstream` bez parametrů.

Následující příklad ukazuje použití jednoparametrového konstruktoru datového typu `ofstream`, který vytvoří novou instanci typu `ofstream` a zároveň otevře soubor `studenti.dat` pro zápis:

```
ofstream soubor("studenti.dat");
```

A následující příklad otevře tentýž soubor pro zápis voláním dvouparametrového konstruktoru datového typu `fstream`:

```
fstream soubor("studenti.dat", ios::out);
```

Když deklarujete proměnnou typu `ofstream` nebo `fstream` a následně v samostatném příkazu zavoláte funkci `open`, je to jako kdybyste jedním příkazem deklarovali proměnnou primitivního typu a dalším příkazem ji inicializovali:

```
int vek;
vek = 39;
```

Naproti tomu když zavoláte konstruktor datového typu `ofstream` nebo `fstream` s nějakými parametry, je to podobné jako inicializace proměnné současně s její deklarací:

```
int vek = 39;
```

Obě varianty jsou víceméně rovnocenné. Většinou si mezi nimi vyberete na základě potřeb konkrétního programu.

Otevření souboru pro čtení

Předchozí text o práci se soubory se do velké míry týká i čtení ze souborů. Hlavní rozdíl je v tom, že při čtení můžete kromě typu `fstream` použít typ `ifstream` (nikoliv `ofstream`). Otevíraný soubor už navíc musí existovat – když se pokusíte otevřít neexistující soubor, na rozdíl od otevření pro zápis se vám automaticky nevytvoří. K tomuto problému se ještě vrátíme za chvilku.

Následující příklad otevře soubor pro čtení členskou funkcí `open` datového typu `ifstream`:

```
ifstream soubor;
soubor.open("studenti.dat");
```

Totéž byste mohli provést i prostřednictvím typu `fstream`. Stačí přidat parametr, který zapíná režim čtení:

```
fstream soubor;
soubor.open("studenti.dat", ios::in);
```

Následující příklad ukazuje, jak soubor pro čtení otevřete pomocí konstruktoru datového typu `ifstream`:

```
ifstream soubor("studenti.dat");
```

A totéž by se opět dalo zařídit konstruktorem datového typu `fstream`, kde druhým parametrem zapnete režim čtení:

```
fstream soubor("studenti.dat", ios::in);
```

Otevření souboru pro čtení i zápis

Pokud chcete do souboru zapisovat a zároveň z něj číst, musíte použít datový typ `fstream`. Typy `ifstream` ani `ofstream` se pro tyto účely použít nedají, jelikož `ofstream` neumožňuje čtení a `ifstream` zase zápis. Následující kus kódu otevře soubor pro čtení i zápis prostřednictvím proměnné typu `fstream`:

```
fstream soubor;
soubor.open("studenti.dat", ios::in | ios::out);
```

Analogicky můžete použít konstruktor datového typu `fstream` se dvěma parametry:

```
fstream soubor("studenti.dat", ios::in | ios::out);
```

Obě varianty používají operátor bitového součtu, který jsme si popsali dříve v této kapitole.

Poznámka: Kombinace příznaků `ios::in` a `ios::out` změní výchozí chování objektu. Pokud už soubor otevíraný výhradně pro zápis existuje, jeho obsah se vymaže. A když soubor otevíráte pouze pro čtení, už musí existovat. Naproti tomu soubor otevíraný pro čtení i zápis existovat nemusí, a pokud už existuje, jeho obsah se zachová.

Kontrola chyb

Po zavolání funkce `open` nebo konstruktoru byste si měli ověřit, jestli se soubor otevřel úspěšně. Existuje hned několik důvodů, proč by se soubor nemuselo povést otevřít, a pokud se pokusíte pracovat s neúspěšně otevřeným souborem, může dojít k chybám.

Hlavní rozdíl v otevírání souborů pro čtení a pro zápis je v tom, že pro zápis můžete otevřít i neexistující soubor – operační systém ho jednoduše vytvoří. Naproti tomu pro čtení se dá otevřít jen soubor, který už existuje. Pokud se tedy chystáte číst ze souboru otevřeného pro čtení, musíte předem zkontrolovat, jestli se otevření povedlo.

Tato kontrola se dá provést několika způsoby, například členskou funkcí `fail`. Funkce `fail` říká, jestli během poslední operace s proudem došlo k chybě. Pokud ji zavoláte po funkci `open` a vrátí vám `true`, znamená to, že se otevření souboru nepovedlo:

```
#include <fstream>
#include <iostream>
```

```
using namespace std;
int main()
{
    ifstream soubor;
    soubor.open("studenti.dat");
    cout << soubor.fail() << "\n";
    return 0;
}
```

Pokud se soubor `studenti.dat` nepodaří otevřít, program vypíše jedničku. Pokud otevření proběhne úspěšně, funkce `fail` vrátí hodnotu `false` a program vypíše nulu.

Když neexistující soubor zkusíte otevřít pro zápis proměnnou typu `ofstream`, operace se většinou povede a funkce `fail` vrátí `false`, protože operační systém neexistující soubor vytvoří. K chybě může dojít například tehdy, když už soubor existuje a je pouze pro čtení. Zkuste si schválně v adresáři s následujícím programem vytvořit soubor `studenti.dat` a nastavit, že je pouze pro čtení.

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    ofstream soubor;
    soubor.open("studenti.dat");
    cout << soubor.fail() << "\n";
    return 0;
}
```

Program vypíše jedničku (`true`), protože chráněný soubor se pro zápis otevřít nedá. Když se soubor nepodaří otevřít, nemá smysl, abyste do něj zkoušeli zapisovat nebo z něj četli. V takovém případě je lepší program přerušit:

```
ifstream soubor;
soubor.open("studenti.dat");
if (soubor.fail())
{
    cout << "Nemohu otevrit datovy soubor.\n";
    return 0;
}
// Následuje práce se souborem.
```

Poznámka: Z následujících příkladů v této kapitole jsme kontrolu chyb víceméně vypustili, abychom text zbytečně nenatahovali a neopakovali stále stejný kód dokola.

Uzavření souboru

Soubor jsme sotva otevřeli, a tak dá rozum, že ho hned nebudeme zavírat – nejdřív z něj něco přečteme nebo do něj něco zapíšeme. Zavření souboru je ale tak jednoduché, že se na něj podíváme mimo pořadí už teď, a teprve pak se pustíme do složitějšího čtení a zápisu.

Soubory byste měli zavírat v okamžiku, kdy už s nimi nepotřebujete pracovat. Operační systém sice zavře všechny otevřené soubory po skončení programu automaticky, ale program poběží lépe, když budete soubory zavírat při nejbližší možnosti sami – každý otevřený soubor totiž spotřebuje určité systémové prostředky. (A některé operační systémy mají omezení na maximální počet záro-

veň otevřených souborů.) Včasným zavřením si navíc ušetříte potíže se sdílením dat v rámci svého programu – kdybyste v jedné části programu otevřeli a nezavřeli soubor, mohli byste pak mít potíže s jeho otevřením v jiné části programu.

K uzavření souboru slouží členská funkce `close`, parametry nemá žádné. Následující příklad zavře soubor otevřený pro zápis:

```
ofstream soubor;
soubor.open("studenti.dat");
// (...) Práce se souborem.
soubor.close();
```

Zavření souboru otevřeného pro čtení se v ničem neliší:

```
ifstream soubor;
soubor.open("studenti.dat");
// (...) Práce se souborem.
soubor.close();
```

Zápis do souboru

Pro zápis informací do souboru slouží operátor <<, stejně jako pro výstup na obrazovku. Jediný rozdíl je v tom, že místo objektu `cout` použijete proměnnou typu `ofstream` nebo `fstream`. Následující program zapisuje uživatelem zadané informace do souboru `studenti.dat`. Pokud soubor ještě neexistuje, vytvoří se nový.

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    string vstup;
    ofstream soubor;
    soubor.open("studenti.dat");
    cout << "Zapis do souboru\n";
    cout << "================\n";
    cout << "Zadejte nazev prednasky: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    cout << "Zadejte pocet studentu: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    soubor.close();
    return 0;
}
```

Vstup a výstup programu vypadají například takhle:

```
Zapis do souboru
================
Zadejte nazev prednasky: Programovani
Zadejte pocet studentu: 32
```

Když si soubor `studenti.dat` otevřete nějakým obyčejným textovým editorem, například Poznámkovým blokem, měli byste v něm najít zadané informace:

```
Programovani
32
```

Za obsah proměnné jsme do souboru přidali ještě konec řádku, aby každá hodnota byla na samostatném řádku:

```
soubor << vstup << "\n";
```

Kdybychom konec řádku vynechali, obsah souboru by dopadl následovně:

```
Programovani32
```

Stejně byste pro zápis mohli použít proměnnou typu `fstream`, stačilo by změnit typ proměnné soubor z `ofstream` na `fstream` a přidat druhý parametr do volání funkce `open`:

```
fstream soubor;
soubor.open("studenti.dat", ios::out);
```

Oba příkazy by se daly spojit do jednoho použitím konstruktoru:

```
fstream soubor("studenti.dat", ios::out);
```

Kdybyste chtěli nové informace zapisovat na konec souboru, stačí do druhého parametru konstruktoru nebo funkce `open` přidat příznak `ios::app`. S prvním příznakem `ios::out` byste ho spojili operátorem |, `ios::app | ios::out`.

Čtení ze souboru

Ke čtení informací ze souboru slouží operátor >>, stejně jako ke čtení z klávesnice. Jediný rozdíl je v tom, že objekt `cin` nahradíte proměnnou typu `ifstream` nebo `fstream`. Následující program je rozšířením toho předchozího. Informace načte od uživatele a uloží do souboru, ze kterého je následně zase přečte a vypíše na obrazovku.

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    string vstup;
    fstream soubor;

    // Zápis.
    soubor.open("studenti.dat", ios::out);
    cout << "Zapis do souboru\n";
    cout << "================\n";
    cout << "Zadejte nazev prednasky: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    cout << "Zadejte pocet studentu: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    soubor.close();

    // Čtení.
    soubor.open("studenti.dat", ios::in);
    cout << "Cteni ze souboru\n";
    cout << "================\n";
```

```
    soubor >> vstup;
    cout << "Nazev prednasky: " << vstup << "\n";
    soubor >> vstup;
    cout << "Pocet studentu: " << vstup << "\n";
    soubor.close();

    return 0;
}
```

Ukázkový vstup a výstup:

```
Zapis do souboru
================
Zadejte nazev prednasky: Programovani
Zadejte pocet studentu: 32
Cteni ze souboru
================
Nazev prednasky: Programovani
Pocet studentu: 32
```

Čtení řetězců s mezerami

Co se stane, když našemu programu zadáme název přednášky s mezerou? Podívejte se na následující ukázku:

```
Zapis do souboru
================
Zadejte nazev prednasky: Programovani bez zahad
Zadejte pocet studentu: 32
Cteni ze souboru
================
Nazev prednasky: Programovani
Pocet studentu: bez
Obsah zapsaného souboru vypadá takto:
Programovani bez zahad
32
```

Při prvním volání operátoru `>>` jsme nepřečetli celý první řádek souboru (`Programovani bez zahad`), přečetli jsme jen první slovo (`Programovani`). Proto nám další volání operátoru `>>` místo počtu studentů vrátilo následující slovo, `bez`.

Operátor `>>` čte z proudu `ifstream` postupně, od prvního bajtu souboru. První čtení ze souboru vrátí všechno od prvního bajtu až po první mezeru, tabulátor, konec řádky nebo konec souboru. Každý další pokus o čtení začíná tam, kde předchozí skončil, a opět vrací všechno až po první prázdný znak nebo konec souboru.

První čtení tedy skončí na mezeře za slovem `Programovani`, druhé přečte slovo `bez` a zarazí se na konci řádku. K dalšímu čtení už nedojde, k číslu `32` se vůbec nedostaneme. Tohle by pro vás mělo být lehké déjà vu, protože na podobné problémy už jsme v souvislosti s operátorem `>>` jednou narazili v desáté kapitole. A stejně jako tehdy nám i tentokrát pomůže funkce `getline`.

Kdybyste pracovali s céčkovými řetězci, mohli byste celý řádek najednou načíst členskou funkcí `getline`. Jediný rozdíl oproti desáté kapitole je v tom, že zatímco tehdy šlo o členskou funkci objektu `cin`, tentokrát byste volali členskou funkci `getline` na proměnné typu `ifstream` nebo `fstream`. Kdyby tedy proměnná `vstup` nebyla typu `string`, ale z nějakého důvodu byste

chtěli mít céčkový řetězec (například `char[80]`), načítali byste ho příkazem `soubor.getline (vstup, 80)`.

My ale máme proměnnou `vstup` typu `string`, takže musíme – podobně jako v desáté kapitole – použít obyčejnou funkci `getline`. Jediný rozdíl je v tom, že v desáté kapitole jsme funkcí `getline` četli z objektu `cin`, zatímco teď budeme číst z proměnné `soubor`. Místo příkazu `soubor >> vstup` se tedy objeví příkaz `getline(soubor, vstup)`, viz následující přepsanou verzi programu:

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    string vstup;
    fstream soubor;

    // Zápis.
    soubor.open("studenti.dat", ios::out);
    cout << "Zapis do souboru\n";
    cout << "================\n";
    cout << "Zadejte nazev prednasky: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    cout << "Zadejte pocet studentu: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    soubor.close();

    // Čtení.
    soubor.open("studenti.dat", ios::in);
    cout << "Cteni ze souboru\n";
    cout << "================\n";
    getline(soubor, vstup);
    cout << "Nazev prednasky: " << vstup << "\n";
    getline(soubor, vstup);
    cout << "Pocet studentu: " << vstup << "\n";
    soubor.close();

    return 0;
}
```

Tentokrát už v obou případech načteme celý řádek:

```
Zapis do souboru
================
Zadejte nazev prednasky: Programovani bez zahad
Zadejte pocet studentu: 32
Cteni ze souboru
================
Nazev prednasky: Programovani bez zahad
Pocet studentu: 32
```

Průchod souboru ve smyčce

V předchozím programu jsme ze souboru četli právě dvakrát, protože jsme věděli, že soubor obsahuje přesně dvě řádky; ne více, ne méně. Častější je ale situace, kdy množství dat v souboru předem neznáme. Chtěli bychom číst, dokud nedojdeme na konec souboru.

Datový typ `ifstream` má pro tyto účely funkci `eof` neboli *end of file*, konec souboru. Tato funkce nemá žádné parametry a vrací `true`, pokud je příslušný soubor u konce. Pokud se ze souboru dá ještě něco přečíst, funkce vrátí `false`. Potíž je v tom, že funkce dobře funguje pouze u binárních souborů; u textových souborů tak spolehlivá není. Když totiž za posledním kusem dat v souboru následují nějaké mezery nebo podobné znaky, funkce tvrdí, že ještě nejste na konci souboru. U binárních souborů se tento problém neprojeví, protože tam se žádné zvláštní znaky nerozeznávají.

Proto je jistější sáhnout po funkci `fail`, se kterou jsme se setkali dříve v této kapitole. Následující kód ukazuje, jak se dá s pomocí funkce `fail` vypsat obsah celého souboru:

```
ifstream soubor;
string data;
soubor.open("studenti.dat");
getline(soubor, data);
while (!soubor.fail())
{
    cout << data << "\n";
    getline(soubor, data);
}
soubor.close();
```

Ze souboru zde čteme na dvou místech, před smyčkou a uvnitř smyčky. Funkce `fail` totiž vrací hodnotu `true` až poté, co při pokusu o čtení dojdete na konec souboru. Kdybyste první čtení před začátkem cyklu vynechali, výpis do `cout` by se provedl, i kdyby byl soubor prázdný.

Poznámka: Smyčku `while` byste mohli nahradit smyčkou `do while`. Ušetřili byste si čtení před začátkem smyčky, ale zase byste v těle smyčky museli pomocí `if` kontrolovat, jestli už poslední čtení nenarazilo na konec souboru. Takové kompromisy bývají u cyklů `while` a `do while` běžné.

Když tento kód přidáme k našemu ukázkovému programu, dostaneme následující:

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    string vstup;
    fstream soubor;

    // Zapis.
    soubor.open("studenti.dat", ios::out);
    cout << "Zapis do souboru\n";
    cout << "================\n";
    cout << "Zadejte nazev prednasky: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    cout << "Zadejte pocet studentu: ";
    getline(cin, vstup);
    soubor << vstup << "\n";
    soubor.close();

    // Ctení.
    soubor.open("studenti.dat", ios::in);
    getline(soubor, vstup);
    while (!soubor.fail())
```

```
    {
        cout << vstup << "\n";
        getline(soubor, vstup);
    }
    soubor.close();

    return 0;
}
```

Souborové proudy v parametrech funkcí

V deváté kapitole jsme si ukazovali, jak se dá kód modularizovat prostřednictvím funkcí. Zkusme si v tomto duchu přepsat předchozí program. Přibudou dvě nové funkce: funkce `zapis`, která otevře nový soubor pro zápis, a funkce `cteni`, která otevře nový soubor pro čtení. Obě funkce kontrolují, jestli se soubor otevřel úspěšně, a podle toho vrací hodnotu typu `bool`.

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

char* CESTA = "studenti.dat";

bool zapis(ofstream&, char*);
bool cteni(ifstream&, char*);

int main()
{
    string data;
    bool stav;
    ofstream vystup;

    stav = zapis(vystup, CESTA);
    if (!stav)
    {
        cout << "Chyba pri otevirani souboru " << CESTA << " pro zapis.\n";
        cout << "Program bude ukoncen.\n";
        return 1;
    }

    cout << "Zapis do souboru\n";
    cout << "================\n";
    cout << "Zadejte nazev prednasky: ";
    getline(cin, data);
    vystup << data << "\n";
    cout << "Zadejte pocet studentu: ";
    getline(cin, data);
    vystup << data << "\n";
    vystup.close();

    ifstream vstup;
    stav = cteni(vstup, CESTA);
    if (!stav)
    {
```

```
        cout << "Chyba pri otevirani souboru " << CESTA << " pro cteni.\n";
        cout << "Program bude ukoncen.\n";
        return 1;
    }

    getline(vstup, data);
    while (!vstup.fail())
    {
        cout << data << "\n";
        getline(vstup, data);
    }
    vstup.close();

    return 0;
}

bool zapis(ofstream& soubor, char* cesta)
{
    soubor.open(cesta);
    return (!soubor.fail());
}

bool cteni(ifstream& soubor, char* cesta)
{
    soubor.open(cesta);
    return (!soubor.fail());
}
```

Přestože ani jedna z nových funkcí nemění obsah souboru, souborový proud se předává odkazem, protože otevření souboru může změnit vnitřní stav proudu.

Shrnutí

Za trvale uložená se považují data, která přežijí ukončení programu i vypnutí počítače. Data uložená v proměnných se za trvalá považovat nedají, protože proměnné existují v operační paměti, která svůj obsah ztratí s vypnutím počítače. Pokud mají data vydržet déle, musí se uložit do souboru na pevný disk počítače nebo jiné médium. V této kapitole jste se dozvěděli, jak data do souborů ukládat. A jelikož to by vám samo o sobě příliš neposloužilo, naučili jste se i data ze souborů číst.

Řekli jsme si, že soubor je sbírka dat uložená na trvalém médiu, například pevném disku nebo CD. Soubory jsou dvojího druhu, textové a binární. Textové soubory obsahují data převedená na řetězce ASCII znaků. Naproti tomu binární soubory obsahují data ve stejné podobě, jakou mají v operační paměti, tedy v podstatě jako nuly a jedničky. V textovém formátu ukládají data například textové editory, viz Poznámkový blok. Binární soubory zvládnou libovolně složitá data, a tak je většinou používají větší programy, například textové procesory, tabulkové kalkulátory nebo databáze.

Standardní knihovna nabízí pro práci se soubory datové proudy `ifstream`, `ofstream` a `fstream` definované v souboru `fstream`. Typ `ofstream` je určený pro zápis do souborů, typ `ifstream` pro čtení ze souborů a typ `fstream` zvládá čtení i zápis zároveň.

Práce se souborem – ať už budete jen číst, jen zapisovat, nebo obojí – se skládá ze tří kroků. Za prvé musíte soubor otevřít, čímž vznikne komunikační kanál mezi souborem a datovým proudem typu `ifstream`, `ofstream` nebo `fstream` ve vašem programu. Druhým krokem je čtení a zápis informací, a závěrečným třetím krokem je zavření souboru. K uzavření souboru slouží členská funkce

close, která uvolní systémové prostředky potřebné k udržování zmíněného komunikačního kanálu. Kdybyste soubory včas nezavírali sami, mohli byste narazit na různé problémy. Například by se mohlo stát, že by některá část vašeho programu nemohla soubor otevřít, protože už by byl otevřený někde jinde.

Soubor se dá otevřít konstruktorem nebo členskou funkcí open. Konstruktor je funkce, která se automaticky volá při vzniku instance nějakého datového typu (v našem případě tedy typu ifstream, ofstream nebo fstream). Konstruktory zmíněných tří proudů i jejich členská funkce open mají po dvou parametrech. Prvním z nich je relativní nebo absolutní cesta k souboru. Druhým, nepovinným parametrem jsou bitové příznaky, které zadávají konkrétní způsob práce se souborem – například jestli chcete zapisovaná data přidávat až na konec, jestli chcete soubor otevřít pro čtení nebo pro zápis a podobně.

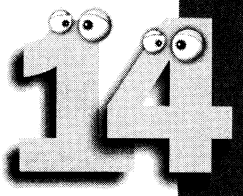
Po volání funkce open musíte zkontrolovat, jestli se soubor podařilo otevřít. K tomu se dá použít například funkce fail, která vrací true, pokud při poslední operaci s proudem došlo k chybě. Pokud tedy otevření souboru dopadlo dobře, funkce fail vrátí false.

Pro zápis do souboru slouží operátor <<, který jste zvyklí používat pro výstup na obrazovku. Jediný rozdíl je v tom, že místo cout zadáte proměnnou typu ofstream nebo fstream. Podobně o čtení ze souborového proudu se stará operátor >>, který už delší dobu používáme pro čtení z klávesnice. Rozdíl je pouze v tom, že místo cin uvedete proměnnou typu ifstream nebo fstream. Ke čtení celých řádek slouží členská funkce getline (pokud chcete řádku načíst do céčkového řetězce) a běžná funkce getline (pokud chcete řádku načíst do řetězce typu string). Při čtení celých souborů vám opět poslouží funkce fail, podle které poznáte, jestli už jste došli na konec souboru.

Souborové proudy se dají předávat jako parametry funkcím. Měli byste je předávat odkazem, nikoliv hodnotou, protože operace provedené v rámci funkce mohou změnit stav proudu.

Test

1. Co znamená „trvalé“ uložení dat?
2. Co je to soubor?
3. Jaké jsou dva hlavní typy souborů?
4. Jaký soubor ze standardní knihovny musíte přiložit ke svému zdrojovému kódu, abyste mohli pracovat se souborovými datovými proudy?
5. Který z datových typů ifstream, ofstream a fstream se dá použít pro čtení i zápis?
6. Kterými dvěma způsoby se dá soubor otevřít?
7. K čemu slouží otevření souboru?
8. K čemu slouží zavření souboru?
9. Co je to konstruktor?
10. Kdybyste potřebovali zkontrolovat, jestli už jste na konci textového souboru, sáhli byste po funkci eof, nebo funkci fail?
11. Mají se datové proudy funkcím předávat hodnotou, nebo odkazem?



Vyhlídky do budoucna: Struktury a třídy

„A co přijde teď?“ zeptal se kterýsi z mých studentů jednoho pozdního večera na závěr poslední přednášky o programování, když už bylo řečeno vše, co v posledních kapitolách řečeno bývá. „Teď půjdeme domů,“ odpověděl jsem.

Má odpověď byla sice technicky vzato správná, ale nepříliš užitečná. Student se proto pokusil navázat: „Nene, já jsem chtěl vědět, co můžeme čekat na příští přednášce.“ Odpověděl jsem mu jako pan Té, když se ho ve třetím dílu Rockyho ptají na očekávaný výsledek odvety s Rockym Balboou: „Bolest.“

I tato odpověď nemusela být daleko od pravdy (záleží na tom, jakého učitele si vybral pro další přednášky), ale opět nebyla moc užitečná, a tak jsem studenty ušetřil dalšího humoru a popsal mu zhruba to, co hodlám popsat i vám v této kapitole. Dá rozum, že má tehdejší odpověď byla mnohem stručnější – přeci jen bylo pozdě večer. Pro vás mám jako první nachystanou otázku: Proč jste si vybrali tuto knihu?

Proč jste si vybrali tuto knihu?

Svého studenta jsem znal z předchozích přednášek a věděl jsem, čemu by se chtěl v budoucnu věnovat. Proto jsem mu dokázal odpovědět. Vaše zájmy ale neznám, a proto bych od vás potřeboval slyšet, který z následujících bodů nejlépe vystihuje vaši motivaci pro čtení této knihy:

- Beru knihu jako doplněk přednášky, na kterou chodím v rámci vysokoškolského studia.
- Chtěl bych se čtením knihy zlepšit ve své současné práci nebo rekvalifikovat pro novou.
- Programování je můj koníček.
- Nemám nic lepšího na práci.

Pokud jste si vybrali poslední možnost, doporučoval bych vám víc chodit ven. V opačném případě vaše hlavní důvody samozřejmě ovlivní i vaše příští programátorské kroky, ať už jde o studium, práci nebo koníčka. Pokud vám například jde o rekvalifikaci na nové místo, vaše zaměření bude mnohem užší než při studiu informatiky.

Ať už je ale vaše zaměření jakékoliv, po tématech probraných v předchozích kapitolách většinou následuje *objektově orientované programování* neboli krátce OOP. Objektově orientované programování spoléhá na dva základní koncepty: struktury a třídy. Právě jim se proto bude věnovat závěrečná kapitola této knihy.

Objektově orientované programování

Výraz *objektově orientované programování*, často nahrazovaný zkratkou OOP, bývá k slyšení velice často. Méně časté je, aby někdo objektovému programování skutečně rozuměl. A to je škoda, protože objektové programování je doopravdy důležité. Základy potřebné pro jeho zvládnutí jste získali v předchozích kapitolách. Podrobný popis se do této kapitoly sice nevejde (objektovému programování jsou v knihovnách vyhrazené celé police!), ale ve stručnosti se na něj podívat můžeme.

Se zkušebními prográmky si člověk užije spoustu zábavy, ale skutečným úkolem programů bývá modelovat procesy našeho světa – tedy procesy, ve kterých vystupují lidé, místa, věci a koncepty. K popisu těchto lidí, míst, věcí a konceptů slouží v programech objekty.

Objekty nabízí *abstraktní*, zjednodušený popis skutečných věcí. Objekt můžete používat, aniž byste měli podrobnou představu o jeho vnitřním uspořádání – podobně jako se při řízení auta obejdete bez detailních znalostí spalovacího motoru. S objekty jsme už pracovali, naposledy v předchozí kapitole s objekty typu `ifstream`, `ofstream` a `fstream`. Mohli jsme otevírat soubory, číst z nich, zapisovat do nich a zase je zavírat, aniž bychom se museli zajímat, co se děje za scénou.

Objekty ovšem nejsou omezené jen na datové proudy a podobně abstraktní entity. Vy sami byste se dali popsat nějakým objektem, tato kniha by se dala popsat objektem, vaše auto by se dalo popsat objektem, herní postava bývá popsaná objektem. Počet objektů a jejich rozmanitost jsou omezené pouze představivostí a tvůrčími schopnostmi programátora.

Důležitou vlastností objektů jsou jejich vztahy s jinými objekty. Tyto vztahy se dají obecně rozdělit do dvou skupin, na vztahy typu *je* a vztahy typu *má*. Příkladem vztahu prvního typu je vztah mezi objekty učitel a člověk – každý učitel *je* člověk. (Tento příklad často zpochybňují moji studenti, ale většinou pak vyjednáme příměří příkladem student *je* člověk. I když tady bych zase občas protestoval já.)

Vztah tohoto typu je velice důležitý, protože když jednou napíšete kód pro třídu Člověk, při psaní kódu třídy Učitel už pak nemusíte opakovat kód pro vlastnosti, které mají učitelé s lidmi společné – například jméno, datum narození, výšku, váhu a podobně. Tyto vlastnosti třída všech učitelů „dědí" po třídě všech lidí, takže vám stačí napsat kód pro vlastnosti, které mají učitelé oproti lidem navíc (například titul nebo vyučované předměty). Schopnosti dědit vlastnosti jiné třídy se říká nepřekvapivě *dědičnost*. Díky dědičnosti se dá jednou napsaný kód snadno použít znovu, například když kód konkrétnější třídy Učitel využije část kódu obecnější třídy Osoba.

Příkladem vztahu typu *má* je auto a motor, každé auto *má* motor. Tento vztah se označuje výrazem *skládání (agregace)* a také podporuje recyklaci kódu, například při psaní třídy Auto můžete využít hotovou třídu Motor.

Recyklace hotového kódu zrychluje vývoj, protože nemusíte pokaždé znovu objevovat Ameriku. A kromě toho snižuje chybovost softwaru, protože jednotlivé „součástky" už bývají dobře otestované (...doufejme).

Struktury

Objekty bývají moc složité na to, aby se daly popsat jednou proměnnou. Pole sice dokáže uložit víc hodnot najednou, ale všechny musí být téhož typu. Takové omezení je na objekty moc přísné.

Například každý člověk má nějaké jméno a výšku. Hodnota obou se dá uložit do proměnné, ale obě proměnné musí být jiného datového typu. Výška by byla uložená jako `int` nebo jiný číselný typ, zatímco jméno by bylo uložené jako řetězec.

Obě proměnné spolu souvisí, popisují dvě různé vlastnosti jednoho konkrétního člověka. Pokud je ale deklarujete následujícím způsobem, půjde jednoduše o dvě samostatné, zcela nezávislé proměnné:

```
string jmeno;
int vyska;
```

C++ vám pro seskupení souvisejících proměnných nabízí takzvané *struktury*. Struktura je datový typ složený z více proměnných libovolného typu; pomocí struktur se v programech dají věrněji modelovat složitější objekty skutečného světa. Na rozdíl od datových typů, se kterými jsme se setkávali doposud (například `int` nebo `char[]`), nejsou konkrétní struktury součástí jazyka C++. Jsou to datové typy definované programátorem.

Deklarování struktury

Jelikož je struktura uživatelský datový typ, musíte ji deklarovat, aby o ní překladač věděl. Následující kus kódu deklaruje strukturu pro jméno a výšku nějaké osoby:

```
struct Osoba
{
    string jmeno;
    int vyska;
};
```

Deklarace začíná klíčovým slovem `struct`, podle kterého překladač pozná, že budete deklarovat strukturu. `Osoba` je název struktury, ten si můžete vybrat celkem libovolně. Stejně jako u proměnných ovšem platí, že je dobré vybrat nějaký dostatečně popisný název. Ve složených závorkách je tělo struktury. Po stránce syntaxe vypadá podobně jako tělo funkce, jen za koncovou složenou závorku se nepíše středník.

Poznámka: Zapomenutý středník za uzavírací složenou závorkou v deklaraci struktury se počítá mezi běžné začátečnické chyby. Hlášení překladače sice někdy bývá značně tajuplné, ale po chvilce hledání si na středník jistě vzpomenete.

Jednotlivé proměnné, které byste chtěli mít v jednom celku, se deklarují v těle struktury. Říká se jim *členské proměnné* a jedna struktura jich může obsahovat mnoho. Jak se navíc dozvíte později v této kapitole, členskou proměnnou struktury může být i další struktura.

Struktura se dá v kódu deklarovat téměř kdekoliv. Bývá ale zvykem struktury deklarovat mezi direktivami preprocesoru a funkcí `main`, viz následující příklad:

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

struct Osoba
{
    string jmeno;
    int vyska;
};
```

```
int main()
{
    // kód
    return 0;
}
```

Možná si teď říkáte: „Moment, moment, vždyť on deklaruje strukturu na místě, kde se deklarují globální proměnné. A globálním proměnným se máme pokud možno vyhýbat, tak jak je to?" Nebojte – tohle není případ kázání vody a pití vína. Deklarace struktury není deklarace proměnné. Struktura je datový typ. Když ji deklarujeme globálně, můžeme ji pak používat v celém programu. Argumenty proti globálním proměnným se na datové typy nevztahují. Struktuře `Osoba` nepřiřadíte hodnotu o nic lépe než například samotnému datovému typu `int`. Pokud chcete do struktury něco ukládat, musíte deklarovat proměnnou příslušného typu.

Instance struktur

Kterousi sobotu jsem jel pozdě v noci na párty vyzvednout svou dceru. Na párty bylo dost hlučno, musel jsem se prodírat snad stovkou mladých lidí. V programátorské hantýrce by se každý z nich dal označit za instanci struktury `Osoba` se svým vlastním jménem a výškou. (U některých by stálo za to zavést i proměnnou `nosni_piercing`, ale tu v zájmu stručnosti vynechám.) Mě zajímala jen jedna konkrétní struktura – když jsem našel paní domácí, neptal jsem se jí po struktuře `Osoba`, ptal jsem se po konkrétním člověku.

Datový typ `Osoba`, který jsme před chvilkou deklarovali, je obecný člověk, obecná osoba. Konkrétní lidé jsou instance tohoto typu a jako každá jiná instance se musí deklarovat. Instance struktur se deklarují stejně jako proměnné vestavěných datových typů, například instance typu `Osoba` se deklaruje takto:

```
Osoba o;
```

Stejně jako při deklaraci proměnné některého z vestavěných datových typů tedy deklarace začíná datovým typem, za kterým následuje jméno proměnné a uzavírací středník, aby překladač poznal konec příkazu. V rámci jednoho příkazu můžete definovat víc proměnných typu `Osoba` najednou:

```
Osoba o1, o2, o3;
```

A pro davy, kterými jsem se prodíral na párty, byste mohli deklarovat celé pole:

```
Osoba party[100];
```

Rozsah platnosti a životnost takto deklarovaných proměnných se nijak neliší od běžných proměnných typu `int`, `float` a podobně.

Přístup k členským proměnným

Deklarace proměnné typu `Osoba` nenastaví hodnoty jejích členských proměnných. Pro přístup k členským proměnným struktury slouží operátor tečka (`.`), například následující příkaz nastaví u proměnné `o1` typu `Osoba` členskou proměnnou `jmeno`:

```
o1.jmeno = "Emily Kentova";
```

Analogicky můžete hodnotu členské proměnné i vypsat:

```
count << "Jmeno o1 je " << o1.jmeno;
```

Je důležité si uvědomit, že před tečku patří název konkrétní proměnné, nikoliv název struktury. Začátečníci dělají běžně tuto chybu:

```
Osoba.jmeno = "Emily Kentova"; // nepřeloží se
```

Následující program deklaruje tříprvkové pole typu `Osoba`, načte jednotlivé členské proměnné od uživatele a zase je vypíše nazpět:

```
#include <iostream>
#include <string>

using namespace std;
const int MAX = 3;
struct Osoba
{
    string jmeno;
    int vyska;
};

int main()
{
    Osoba lide[MAX];
    for (int i=0; i<MAX; i++)
    {
        cout << "Zadejte jmeno: ";
        getline(cin, lide[i].jmeno);
        cout << "Zadejte vysku v centimetrech: ";
        cin >> lide[i].vyska;
        cin.ignore();
    }

    cout << "==========\n";
    cout << "Vystup dat\n";
    cout << "==========\n";

    for (int i=0; i<MAX; i++)
        cout << "Osoba cislo " << i+1
             << " se jmenuje " << lide[i].jmeno
             << " a meri " << lide[i].vyska << " cm.\n";
    return 0;
}
```

Vstup a výstup programu vypadá například takto:

```
Zadejte jmeno: Genghis Khent
Zadejte vysku v centimetrech: 198
Zadejte jmeno: Jeff Kent
Zadejte vysku v centimetrech: 182
Zadejte jmeno: Dante Kent
Zadejte vysku v centimetrech: 25
==========
Vystup dat
==========
Osoba cislo 1 se jmenuje Genghis Khent a meri 198 cm.
Osoba cislo 2 se jmenuje Jeff Kent a meri 182 cm.
Osoba cislo 3 se jmenuje Dante Kent a meri 25 cm.
```

Inicializace struktur

Struktura se dá inicializovat dvěma způsoby, pomocí inicializačního seznamu a pomocí konstruktoru. Inicializaci seznamem ukazuje následující kód:

```
Osoba o = {"Jeff Kent", 182};
```

To by vám mělo připadat povědomé, protože podobně se inicializují pole. Je tu ale jeden důležitý rozdíl – zatímco pole obsahuje hodnoty jednoho typu, u struktury se typ jednotlivých členských proměnných může lišit. Proto hodně záleží na pořadí hodnot v inicializačním seznamu. Například následující kód by si vysloužil chybové hlášení překladače, protože první členská proměnná struktury `Osoba` je `string`, a není tudíž kompatibilní s hodnotu typu `int`:

```
Osoba o = {182, "Jeff Kent"}; // nepřeloží se
```

Druhá možnost je inicializace struktury konstruktorem. Na konstruktory už jsme narazili v předchozí kapitole ve spojení se souborovými proudy: konstruktor je funkce, která se automaticky volá při vytvoření nového objektu.

Inicializace výchozím konstruktorem

Konstruktor není nutné psát. Všimněte si, že v předchozím příkladu s polem tří osob jsme se bez psaní konstruktoru docela dobře obešli. Když nenapíšete vlastní konstruktor, při vytvoření vašeho objektu se zavolá takzvaný *výchozí konstruktor* doplněný překladačem. Použití výchozího konstruktoru ukazuje následující příklad:

```
#include <iostream>
#include <string>
using namespace std;
struct Osoba
{
    string jmeno;
    int vyska;
};
int main()
{
    Osoba o;
    cout << "Osoba se jmenuje " << o.jmeno
         << " a meri " << o.vyska << " cm.\n";
    return 0;
}
```

Výstup programu vypadá takto:

```
Osoba se jmenuje a meri -1881141193 cm.
```

Výchozí konstruktor se volá při vytvoření instance, tedy v rámci následujícího příkazu:

```
Osoba o;
```

Výchozí konstruktor vytvořil novou instanci struktury `Osoba`, ale nenastavil žádnou výchozí hodnotu členských proměnných. Členská proměnná `jmeno` proto obsahuje prázdný řetězec a v proměnné `vyska` je nějaké „nesmyslné“ číslo.

Uživatelský konstruktor bez parametrů

Pokud chcete členské proměnné v rámci konstruktoru nastavit na nějakou rozumnou výchozí hodnotu, můžete si napsat svůj vlastní konstruktor:

```
struct Osoba
{
    string jmeno;
    int vyska;
    Osoba()
    {
        jmeno = "Anonym";
        vyska = -1;
    }
};
```

Samotný konstruktor vypadá takto:

```
Osoba()
{
    jmeno = "Anonym";
    vyska = -1;
}
```

Konstruktor se vždy jmenuje stejně jako struktura sama (bez výjimek) a stejně tak nikdy nemá návratovou hodnotu. Pokud se pokusíte před název konstruktoru dopsat `void`, překladač si pravděpodobně bude stěžovat. Celý program s přidaným konstruktorem vypadá následovně:

```
#include <iostream>
#include <string>
using namespace std;
struct Osoba
{
    string jmeno;
    int vyska;
    Osoba()
    {
        jmeno = "Anonym";
        vyska = -1;
    }
};
int main()
{
    Osoba o;
    cout << "Osoba se jmenuje " << o.jmeno
         << " a meri " << o.vyska << " cm.\n";
    return 0;
}
```

A na jeho výstupu už je vidět inicializace členských proměnných:

```
Osoba se jmenuje Anonym a meri -1 cm.
```

Upozornění: Stejně jako výchozí konstruktor se i tento konstruktor volá příkazem `Osoba o`. Přestože voláme konstruktor, za název proměnné nepatří prázdné závorky; příkaz `Osoba o()` by dělal něco jiného.

Parametrizované konstruktory

Deklarace uživatelského konstruktoru je ve srovnání s výchozím konstruktorem pokrok, protože teď už naše členské proměnné mají rozumné výchozí hodnoty. Ještě lepší by ale bylo, kdybychom mohli členské proměnné rovnou v rámci konstruktoru nastavit na hodnoty zadané uživatelem. To se dá snadno zařídit tím, že ke konstruktoru přidáme parametry. Původní konstruktor může zůstat, takže když přidáme parametrizovaný konstruktor, program bude vypadat následovně:

```
#include <iostream>
#include <string>
using namespace std;
struct Osoba
{
    string jmeno;
    int vyska;
    Osoba()
    {
        jmeno = "Anonym";
        vyska = -1;
    }
    Osoba(string j, int v)
    {
        jmeno = j;
        vyska = v;
    }
};
int main()
{
    string jmeno;
    int vyska;
    cout << "Zadejte jmeno: ";
    getline(cin, jmeno);
    cout << "Zadejte vysku: ";
    cin >> vyska;
    cin.ignore();

    Osoba o(jmeno, vyska);
    cout << "Osoba se jmenuje " << o.jmeno
         << " a meri " << o.vyska << " cm.\n";
    return 0;
}
```

Ukázkový vstup a výstup vypadá následovně:

```
Zadejte jmeno: Jeff Kent
Zadejte vysku: 182
Osoba se jmenuje Jeff Kent a meri 182 cm.
```

Dvouparametrový konstruktor se volá při následující deklaraci proměnné typu `Osoba`:

```
Osoba o(jmeno, vyska);
```

Tentokrát už závorky uvést musíme, protože konstruktoru předáváme parametry. Parametry navíc musí jít v pořadí, v jakém je konstruktor čeká. Kdyby konstruktor čekal jako první parametr `string` a jako druhý parametr `int` a vy jste pak při deklaraci proměnné typu `Osoba` zadali jako první parametr `int` a jako druhý `string`, překladač by si stěžoval.

Oddělení prototypu konstruktoru od jeho implementace

Následující varianta předchozího programu ukazuje, jak se dá prototyp konstruktoru oddělit od jeho implementace. Prototypy konstruktoru zůstávají v těle struktury, implementace jsou samostatné:

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

struct Osoba
{
    string jmeno;
    int vyska;
    Osoba();
    Osoba(string, int);
};

Osoba::Osoba()
{
    jmeno = "Anonym";
    vyska = -1;
}

Osoba::Osoba(string j, int v)
{
    jmeno = j;
    vyska = v;
}

int main()
{
    string jmeno;
    int vyska;
    cout << "Zadejte jmeno: ";
    getline(cin, jmeno);
    cout << "Zadejte vysku: ";
    cin >> vyska;
    cin.ignore();

    Osoba o(jmeno, vyska);
    cout << "Osoba se jmenuje " << o.jmeno
         << " a meri " << o.vyska << " cm.\n";
    return 0;
}
```

V hlavičkách konstruktorů je před jejich názvem ještě čtyřtečka (::) a název struktury, ke které konstruktor patří:

```
Osoba::Osoba()
Osoba::Osoba(string j, int v)
```

Důvod je prostý – těla konstruktorů nejsou součástí struktury, takže kdyby neměly zmíněnou speciální předponu, překladač by je nerozeznal od obyčejných funkcí (o kterých jsme se bavili v deváté kapitole).

Teď už víme, jak se dá prototyp konstruktoru oddělit od jeho implementace. Zbývá otázka, proč něco takového vůbec dělat. Důvod leží v principech objektového programování. Jedna ze základ-

ních zásad OOP zní oddělit rozhraní objektů od jejich implementace. Pokud programátoři dodrží rozhraní objektu (tedy prototypy funkcí), mohou jeho implementaci libovolně měnit, aniž by rozbili kód, který jejich objekty využívá.

Struktury jako parametry funkcí

Následující program předává proměnnou typu Osoba dvěma funkcím, `nacti` a `vypis`. První z nich nastaví hodnoty členských proměnných podle vstupu od uživatele, druhá tyto hodnoty vypíše.

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

struct Osoba
{
    string jmeno;
    int vyska;
};

void nacti(Osoba&);
void vypis(const Osoba&);

int main()
{
    Osoba nekdo;
    nacti(nekdo);
    cout << "==========\n";
    cout << "Vystup dat\n";
    cout << "==========\n";
    vypis(nekdo);
    return 0;
}

void nacti(Osoba& nekdo)
{
    cout << "Zadejte jmeno: ";
    getline(cin, nekdo.jmeno);
    cout << "Zadejte vysku v centimetrech: ";
    cin >> nekdo.vyska;
    cin.ignore();
}

void vypis(const Osoba& nekdo)
{
    cout << "Osoba se jmenuje " << nekdo.jmeno
         << " a meri " << nekdo.vyska << " cm.\n";
}
```

Takto vypadá vstup a výstup programu:

```
Zadejte jmeno: Genghis Khent
Zadejte vysku v centimetrech: 198
==========
Vystup dat
==========
Osoba se jmenuje Gengis Khent a meri 198 cm.
```

Hodnotou struktury není na rozdíl od pole adresa, takže pokud chcete strukturu ve funkci změnit, musíte ji předat odkazem nebo adresou. Proto se proměnná `nekdo` do funkce `nacti` předává odkazem. Odkazem ovšem parametr předáváme i funkci `vypis`, která strukturu nemění. Zde je důvodem vyšší efektivita – předávání odkazem nezabere tolik paměti jako předávání hodnotou, při kterém se potenciálně velký objekt musí kopírovat. Aby funkce parametr omylem nezměnila, je před ním ještě klíčové slovo `const`.

Vnořené struktury

V předchozích kapitolách jsme se už setkali s vnořováním podmínek a cyklů, takže vás asi nepřekvapí, že i struktury se dají vnořovat. „Že se něco udělat dá, ještě neznamená, že to udělat musíš," dostali jste tohle ponaučení od maminky? Já ano, ale vnořování struktur je dobrá věc.

Když vyjdeme z naší ukázkové struktury `Osoba`, každý člověk má například datum narození. A datum se dá popsat jako struktura obsahující den, měsíc a rok ve formátu `int`:

```
struct Datum
{
    int den;
    int mesic;
    int rok;
};
```

Kdybychom pak chtěli ve struktuře `Osoba` uložit i datum narození, jednoduše bychom přidali členskou proměnnou typu `Datum`:

```
struct Osoba
{
    string jmeno;
    int vyska;
    Datum narozeni;
};
```

Následující kód je rozšířením našeho předchozího programu. Přibyla deklarace struktury `Datum`, do struktury `Osoba` přibylo datum narození a podle toho se změnily i funkce `nacti` a `vypis`.

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

struct Datum
{
    int den;
    int mesic;
    int rok;
};

struct Osoba
{
    string jmeno;
    int vyska;
    Datum narozeni;
};

void nacti(Osoba&);
void vypis(const Osoba);
```

```
int main()
{
    Osoba nekdo;
    nacti(nekdo);
    cout << "==========\n";
    cout << "Vystup dat\n";
    cout << "==========\n";
    vypis(nekdo);
    return 0;
}

void nacti(Osoba& nekdo)
{
    cout << "Zadejte jmeno: ";
    getline(cin, nekdo.jmeno);
    cout << "Zadejte vysku v centimetrech: ";
    cin >> nekdo.vyska;
    cin.ignore();
    cout << "Zadejte den, mesic a rok narozeni oddelene mezerami: ";
    cin >> nekdo.narozeni.den >> nekdo.narozeni.mesic >> nekdo.narozeni.
rok;
    cin.ignore();
}

void vypis(const Osoba& nekdo)
{
    cout << "Osoba se jmenuje " << nekdo.jmeno
         << " a meri " << nekdo.vyska << " cm.\n";
    cout << "Narodila se " << nekdo.narozeni.den << ". "
         << nekdo.narozeni.mesic << ". " << nekdo.narozeni.rok << ".\n";
}
```

Takhle vypadá ukázkový vstup a výstup:

```
Zadejte jmeno: Genghis Khent
Zadejte vysku v centimetrech: 198
Zadejte den, mesic a rok narozeni oddelene mezerami: 4 3 1211
==========
Vystup dat
==========
Osoba se jmenuje Genghis Khent a meri 198 cm.
Narodila se 4. 3. 1211.
```

Struktura `Datum` musí být definovaná ještě před strukturou `Osoba`. V opačném případě by překladač nevěděl, co je identifikátor `Datum` ve struktuře `Osoba` zač.

Funkce `nacti` teď nově načítá i datum narození. Nemůže udělat jednoduše `cin >> nekdo.narozeni`, protože operátor `>>` objektu `cin` neumí se strukturami typu `Datum` pracovat. Musíme se dostat až na jednotlivé členské proměnné, které jsou typu `int` a se kterými už operátor pracovat umí:

```
cin >> nekdo.narozeni.den >> nekdo.narozeni.mesic >> nekdo.narozeni.rok;
```

Ze stejného důvodu nemůže funkce `vypis` jednoduše zavolat `cout >> nekdo.narozeni`. Také ona musí sejít až k jednotlivým členským proměnným:

```
cout << "Narodila se " << nekdo.narozeni.den << ". "
     << nekdo.narozeni.mesic << ". " << nekdo.narozeni.rok << ".\n";
```

Vztah mezi strukturami `Osoba` a `Datum` je příkladem skládání, o kterém jsme mluvili na začátku kapitoly: každá osoba *má* datum narození.

Třídy

Většina předchozího textu o strukturách platí i pro třídy. Z větší části můžete dokonce klíčové slovo `struct` jednoduše nahradit klíčovým slovem `class`. Mezi strukturami a třídami je ale důležitý rozdíl. Dobře patrný bude, když si zkusíte jeden z předchozích programů přepsat s klíčovým slovem `class` namísto `struct`:

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

class Osoba
{
    string jmeno;
    int vyska;
    Datum narozeni;
};

void nacti(Osoba&);
void vypis(const Osoba);

int main()
{
    Osoba nekdo;
    nacti(nekdo);
    cout << "==========\n";
    cout << "Vystup dat\n";
    cout << "==========\n";
    vypis(nekdo);
    return 0;
}

void nacti(Osoba& nekdo)
{
    cout << "Zadejte jmeno: ";
    getline(cin, nekdo.jmeno);
    cout << "Zadejte vysku v centimetrech: ";
    cin >> nekdo.vyska;
    cin.ignore();
}

void vypis(const Osoba& nekdo)
{
    cout << "Osoba se jmenuje " << nekdo.jmeno
         << " a meri " << nekdo.vyska << " cm.\n";
}
```

Při pokusu o překlad dostanete od překladače nepříliš lichotivý seznam chyb ve funkcích `nacti` a `vypis`. Překladač si bude stěžovat, že všechny členské proměnné třídy `Osoba` jsou soukromé. Co se stalo? Rozdíl je v tom, že zatímco členské proměnné struktur jsou standardně veřejné (`public`), členské proměnné tříd jsou standardně soukromé (`private`).

K veřejné členské proměnné může přistupovat kdokoliv. Ve struktuře `Osoba` byly veřejné všechny členské proměnné, a tak se překladač nijak nepozastavoval nad tím, že jsme ve funkcích `nacti` a `vypis` sahali na proměnné `nekdo.jmeno` a `nekdo.vyska`.

Zato k soukromým členským proměnným mají přístup pouze členské funkce třídy. Pokud mluvíme o třídě `Osoba`, její členskou funkcí je například její konstruktor, ale už nikoliv funkce `nacti` a `vypis`. Tyto funkce jsou deklarované mimo třídu, a tak ke členským proměnným `jmeno` a `vyska` přístup nemají. Pokud se s nimi přesto pokusíme něco dělat, překladač ohlásí chybu.

Soukromé členské proměnné se dají zpřístupnit veřejnými členskými funkcemi (alias metodami třídy) pro jejich čtení a zápis. Jelikož jde o funkce veřejné, může je použít kdokoliv i mimo třídu. Následující program ukazuje členské funkce `ukaz_jmeno` a `nastav_jmeno` pro přístup k soukromé členské proměnné `jmeno` a podobné funkce `ukaz_vysku` a `nastav_vysku` pro přístup k proměnné `vyska`.

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

class Osoba
{
private:
    string jmeno;
    int vyska;
public:
    string ukaz_jmeno() const;
    void nastav_jmeno(string);
    int ukaz_vysku() const;
    void nastav_vysku(int);
};

string Osoba::ukaz_jmeno() const
{
    return jmeno;
}

void Osoba::nastav_jmeno(string nove)
{
    jmeno = (nove.length() == 0) ?
        "Anonym" : nove;
}

int Osoba::ukaz_vysku() const
{
    return vyska;
}

void Osoba::nastav_vysku(int nova)
{
    vyska = (nova < 0) ? 0 : nova;
}

void nacti(Osoba&);
void vypis(const Osoba);

int main()
{
```

```
    Osoba nekdo;
    nacti(nekdo);
    cout << "==========\n";
    cout << "Vystup dat\n";
    cout << "==========\n";
    vypis(nekdo);
    return 0;
}

void nacti(Osoba& nekdo)
{
    string jmeno;
    int vyska;

    cout << "Zadejte jmeno: ";
    getline(cin, jmeno);
    nekdo.nastav_jmeno(jmeno);

    cout << "Zadejte vysku v centimetrech: ";
    cin >> vyska;
    cin.ignore();
    nekdo.nastav_vysku(vyska);
}

void vypis(const Osoba& nekdo)
{
    cout << "Osoba se jmenuje " << nekdo.ukaz_jmeno()
         << " a meri " << nekdo.ukaz_vysku() << " cm.\n";
}
```

Poznámka: Proč se proměnná `nekdo` ve funkci `vypis` předává odkazem a proč je před ní klíčové slovo `const` jsme si vysvětlovali v předchozí části kapitoly věnované předávání struktur v parametrech funkcí. Podobně klíčové slovo `const` za členskými funkcemi `ukaz_jmeno` a `ukaz_vysku` znamená, že funkce nemění hodnotu objektu.

Teď už se program přeloží, rozběhne a dává správné výsledky:

```
Zadejte jmeno: Jeff Kent
Zadejte vysku v centimetrech: 182
==========
Vystup dat
==========
Osoba se jmenuje Jeff Kent a meri 182 cm.
```

To je hezké, ale vás by oprávněně mohlo zajímat, proč jsme si dávali takovou práci s psaním veřejných členských funkcí, když by bývalo stačilo označit členské proměnné `jmeno` a `vyska` za veřejné.

Dokud byla `Osoba` obyčejná struktura, nikoliv třída, nemohli jsme nijak zabránit nastavení členských proměnných na chybnou hodnotu. Členské proměnné struktur jsou standardně veřejné, takže si s nimi může každý dělat, co chce. Zadané jméno může být klidně prázdné a výška záporná:

```
Zadejte jmeno:
Zadejte vysku v centimetrech: -5
==========
Vystup dat
==========
Osoba se jmenuje a meri -5 cm.
```

Jakmile ale strukturu přepíšeme na třídu, členské proměnné se dají označit za soukromé a zvenčí budou přístupné jen prostřednictvím členských funkcí, které budou zadané hodnoty kontrolovat. Například funkce `nastav_jmeno` kontroluje, jestli zadané jméno není prázdné, a pokud ano, uloží do členské proměnné výchozí hodnotu. Podobně funkce `nastav_vysku` kontroluje, jestli je zadaná výška kladná, a pokud ne, uloží do členské proměnné nulu:

```
Zadejte jmeno:
Zadejte vysku v centimetrech: -5
==========
Vystup dat
==========
Osoba se jmenuje Anonym a meri 0 cm.
```

Tomu se v objektovém programování říká *skrývání informací* nebo též *zapouzdření*. Jednu z jeho výhod jste viděli v předchozím příkladu – pokud jsou členské proměnné přístupné jen prostřednictvím členských funkcí, můžete celkem snadno zajistit, aby vždy obsahovaly jen přípustné hodnoty. Funkce určené ke čtení hodnot soukromých členských proměnných zase mohou kontrolovat, jestli je uživatel oprávněný ke čtení příslušných informací a podobně.

Shrnutí

Jak už samotné jméno napovídá, objektově orientované programování neboli OOP se zabývá objekty. Běžným úkolem programů bývá modelovat nějaké procesy reálného světa, ve kterých vystupují osoby, místa, věci nebo koncepty. Pro jejich modelování v programech slouží objekty.

Objekty bývají běžně spojené nějakým vztahem. Většina těchto vztahů se dá rozdělit do dvou skupin, na vztahy typu *je* a vztahy typu *má*. Ve vztahu prvního typu jsou například objekty člověk a učitel: každý učitel *je* člověk. Tomuto vztahu se říká *dědičnost*. Z programátorského hlediska je dědičnost důležitá, protože nám při psaní tříd (například třídy `Učitel`) umožňuje stavět na hotovém kódu pro obecnější třídy (v tomto případě třídu `Člověk`).

Vztah typu *má* je například mezi autem a motorem: každé auto *má* motor. Tomuto vztahu se říká *skládání (agregace)* a rovněž nám šetří práci, protože můžeme při psaní tříd (například `Auto`) použít hotový kód pro jiné třídy (`Motor`).

Použití hotového kódu urychluje vývoj, protože nemusíte pokaždé znovu objevovat Ameriku. Aplikace napsané s využitím hotových komponent navíc neobsahují tolik chyb, protože často používaný kód bývá dobře odladěný a bez chyb. Objektové programování tedy umožňuje modelovat složité objekty skutečného světa a recyklovat hotový, dobře otestovaný kód.

Objekty jsou většinou příliš složité na to, aby se daly popsat jednou proměnnou primitivního datového typu. Jednotlivé hodnoty popisující objekt navíc bývají různého typu, a tak se nedají uložit do pole. Pro popis složitějších objektů se proto používají takzvané *struktury*. Struktura je programátorem definovaný datový typ, který sdružuje několik proměnných libovolného datového typu. Tyto proměnné se označují jako *členské proměnné* struktury.

V této kapitole jsme si nejprve ukázali, jak se struktury deklarují. Deklarace začíná klíčovým slovem `struct`, za kterým následuje jméno struktury. Pak přijde tělo struktury, které je podobně jako tělo funkce uzavřené ve složených závorkách, ale na rozdíl od těla funkce za ním musí být středník. V těle struktury se uvádí jednotlivé členské proměnné. Pro přístup k členským proměnným – přesněji řečeno pro jejich čtení i zápis – slouží operátor tečka (`.`), například `clovek.jmeno`.

Deklarací struktury vytvoříte nový datový typ. Pokud má mít praktický smysl, musíte ještě deklarovat nějakou proměnnou tohoto typu, tedy vytvořit jeho instanci. Nic vám samozřejmě nebrání vytvořit víc proměnných daného typu, případně rovnou celé pole.

Struktury se dají inicializovat dvěma způsoby, inicializačním seznamem a konstruktorem. Konstruktor – jak už jsme si říkali v předchozí kapitole – je funkce, která se automaticky volá při vzniku nového objektu. Konstruktorů můžete definovat víc, například jeden bez parametrů a druhý s několika parametry.

Jedním ze základních principů objektového programování je oddělení rozhraní objektů od jejich implementace. Proto programátoři běžně oddělují prototyp a samotnou implementaci konstruktoru – prototyp zůstane v definici struktury, zatímco samotná implementace je někde jinde. Pokud konstruktor implementujete mimo definici struktury, musíte před jeho jméno připsat název příslušné struktury a čtyřtečku (`::`). Jinak by překladač nepoznal, že konstruktor ke struktuře patří.

Struktury se dají předávat v parametrech funkcí. Hodnotou struktury ovšem na rozdíl od polí není adresa, takže pokud chcete změnit nějakou členskou proměnnou struktury, musíte strukturu předat odkazem nebo adresou. Odkazem se ostatně běžně předávají i struktury, které měnit nechcete, protože předávání odkazem spotřebuje méně paměti než předávání hodnotou. V takových případech se před název struktury v seznamu parametrů ještě přidává klíčové slovo `const`, aby ji funkce omylem nezměnila.

Členskou proměnnou struktury může být i jiná struktura. Pokud se budeme držet příkladu se třídou `Osoba`, každý člověk má nějaké datum narození. Datum se dá popsat strukturou se třemi členskými proměnnými pro den, měsíc a rok. Když tedy chcete v rámci struktury `Osoba` uložit datum narození, stačí přidat členskou proměnnou typu `Datum`. To je klasický příklad skládání: každý člověk *má* datum narození.

Strukturám jsou velice blízké *třídy*. Deklarují se úplně stejně, jen klíčové slovo `struct` nahradíte klíčovým slovem `class`. Důležitý rozdíl je v tom, že členské proměnné struktur jsou standardně veřejné, zatímco členské proměnné tříd jsou standardně soukromé. K veřejným členským proměnným má přístup kdokoliv, na soukromé členské proměnné mohou sahat jen členské funkce třídy.

Členské funkce tříd se běžně používají pro přístup k členským proměnným. Bývají veřejné, aby se daly používat i mimo třídu. Když jsou členské proměnné soukromé a přístup k nim je možný pouze přes veřejné členské funkce, třída si může snadno pohlídat, aby žádná z jejích členských proměnných neobsahovala nepřípustnou hodnotu. Soukromé členské proměnné zpřístupněné pomocí veřejných funkcí jsou klasickým příkladem dalšího ze základních principů objektového programování, a to skrývání informací.

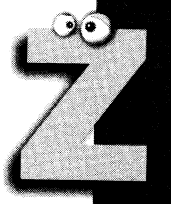
Doufám, že jste si čtení této knihy užili stejně, jako jsem si já užil její psaní. Přeji vám hodně úspěchů ve vašem dalším programátorském úsilí.

Test

1. Jaké jsou hlavní výhody objektového programování?
2. Dokázali byste uvést příklad skrývání informací neboli zapouzdření?
3. Dědičnost je vztah typu *má*, nebo vztah typu *je*?
4. Skládání (agregace) je vztah typu *má*, nebo vztah typu *je*?
5. Co je to struktura?

6. Když deklarujete strukturu, deklarujete datový typ, nebo jeho instanci?
7. Kterými dvěma způsoby se dá inicializovat struktura?
8. Může být členskou proměnnou struktury jiná struktura?
9. Proč je dobré předávat strukturu odkazem i v případech, kdy ji nehodláte měnit?
10. Jaký je rozdíl mezi strukturou a třídou?

Závěrečný test



1. Co je počítačový program?
2. Co je programovací jazyk?
3. Co je funkce?
4. Kolik funkcí `main` najdete v běžném programu napsaném v C++?
5. Co je to standardní knihovna?
6. K čemu slouží direktiva `#include`?
7. Co dělá preprocesor?
8. Co dělá překladač?
9. Co dělá linker?
10. Která z pamětí není dočasná – cache, RAM, nebo pevný disk?
11. Kolik informací se dá uložit na jedné paměťové adrese?
12. Je velikost datových typů C++ nezávislá na počítači?
13. Jaký je rozdíl mezi datovými typy se znaménkem a bez?
14. Co je to ASCII kód?
15. Co je to řetězcový literál?
16. Co je to výraz?
17. Co se stane při deklaraci proměnné?
18. Můžete s proměnnou pracovat, ještě než ji deklarujete? (Za předpokladu, že ji deklarujete později.)
19. Jaký je rozdíl mezi adresovacím operátorem a sizeof?
20. Jaký je rozdíl mezi inicializací a přiřazením?
21. Co je přetečení?
22. Probíhá přiřazení hodnot čtením z objektu `cin` během překladu, nebo za běhu programu?
23. Pro kterou ze čtyř základních aritmetických operací nabízí C++ více než jeden operátor?
24. Který z aritmetických operátorů nepracuje s desetinnými čísly?
25. Který z aritmetických operátorů nemůže mít jako druhý operand nulu?
26. Jak jinak byste mohli zapsat výraz `celkem = celkem + 2`?
27. Jaká je hodnota výrazu `2+3*4`?
28. Jaká je hodnota výrazu `8/2*4`?

29. Jaká je hodnota výrazu `10/4`?
30. Jakým operátorem nebo funkcí se v C++ umocňuje?
31. Co je to algoritmus?
32. Kolik operandů má relační výraz?
33. Jaký bývá datový typ výrazu za klíčovým slovem `if`?
34. Kterou větev musí každý příkaz `if/else if/else` obsahovat právě jednou?
35. Kterou větev může příkaz `if/else if/else` obsahovat vícekrát?
36. Kterou větev můžete z příkazu `if/else if/else` vynechat?
37. Jaký je datový typ výrazu, který najdete za klíčovým slovem `switch`?
38. Může za klíčovým slovem `case` v příkazu `switch` následovat proměnná?
39. Jaké klíčové slovo hraje v příkazu `switch` podobnou roli jako `else` v příkazu `if`?
40. Dají se pomocí vnořených příkazů `if` nahradit logické operátory `&&` a `||`?
41. U kterého z logických operátorů musí být oba operandy bezpodmínečně pravdivé, aby byl i výsledek pravdivý?
42. U kterého z logických operátorů musí být oba operandy bezpodmínečně nepravdivé, aby byl i výsledek nepravdivý?
43. Který z logických operátorů převrací pravdivostní hodnotu svého operandu?
44. Co dělá operátor ++?
45. Co dělá operátor `--`?
46. Pokud máme v proměnné `cislo` hodnotu 5, co vypíše příkaz `cout << --cislo`?
47. Co je to iterace?
48. K čemu běžně slouží první část závorek za klíčovým slovem `for`?
49. K čemu běžně slouží druhá část závorek za klíčovým slovem `for`?
50. K čemu běžně slouží třetí část závorek za klíčovým slovem `for`?
51. Může být některá z částí závorek za klíčovým slovem `for` prázdná?
52. K čemu ve smyčce `for` slouží příkaz `break`?
53. K čemu ve smyčce `for` slouží příkaz `continue`?
54. Kdybychom chtěli pomocí dvou vnořených cyklů vytisknout tabulku, který z cyklů by měl na starosti sloupce – vnitřní, nebo vnější?
55. Která ze smyček `for`, `while` a `do while` se vždy provede alespoň jednou?
56. Která ze smyček `for`, `while` a `do while` se nejvíc hodí pro předem známý počet iterací?
57. Do závorek za klíčové slovo `while` patří inicializace, podmínka, nebo aktualizace?
58. Co nebo kdo je to příznak?
59. Jaký je rozdíl mezi životností proměnné a jejím rozsahem platnosti?
60. Musí se každá funkce s výjimkou `main` prototypovat?
61. Musí mít každá funkce alespoň jeden parametr?

62. Může mít funkce víc než jeden parametr?
63. Změní se ve funkci `main` hodnota proměnné, když tuto proměnnou předáme hodnotou nějaké funkci a ta změní hodnotu odpovídajícího parametru?
64. Změní se ve funkci `main` hodnota proměnné, když tuto proměnnou předáme odkazem nějaké funkci a ta změní hodnotu odpovídajícího parametru?
65. Musí mít každá funkce nějakou návratovou hodnotu?
66. Může mít funkce více než jednu návratovou hodnotu?
67. Může být funkce bez parametrů i bez návratové hodnoty?
68. Může mít funkce zároveň návratovou hodnotu i parametry?
69. Může jedno pole obsahovat zároveň celá čísla, desetinná čísla a znaky?
70. Jaký je index prvního prvku pole?
71. Jaký je index posledního prvku pole?
72. Jakými dvěma způsoby se dá během inicializace určit velikost pole?
73. K čemu slouží nulový znak?
74. Jaká je hodnota proměnné typu pole?
75. Když funkci předáváte pole, předáváte ho odkazem, adresou, nebo hodnotou?
76. Co je to ukazatel?
77. V čem se deklarace proměnné typu `int` liší od deklarace ukazatele na `int`?
78. Jaký má v deklaraci ukazatele význam datový typ?
79. Co je to `NULL`? K čemu slouží?
80. Když chceme do ukazatele uložit adresu nějaké proměnné, kterým operátorem ji získáme?
81. K čemu je dereferenční operátor?
82. Co s ukazatelem provede operátor ++?
83. K čemu jsou operátory `new` a `delete`?
84. Dá se operátorem = přiřadit hodnota jednoho céčkového řetězce druhému?
85. Co znamená „trvalé“ uložení dat?
86. Co je to soubor?
87. Jaký soubor ze standardní knihovny musíte přiložit ke svému zdrojovému kódu, abyste mohli pracovat se souborovými datovými proudy?
88. Který z datových typů `ifstream`, `ofstream` a `fstream` se dá použít pro čtení i zápis?
89. Kterými dvěma způsoby se dá soubor otevřít?
90. K čemu slouží otevření souboru?
91. K čemu slouží zavření souboru?
92. Co je to konstruktor?
93. Mají se datové proudy funkcím předávat hodnotou, nebo odkazem?
94. Dědičnost je vztah typu *má*, nebo vztah typu *je*?

95. Skládání (agregace) je vztah typu *má*, nebo vztah typu *je*?
96. Co je to struktura?
97. Kterými dvěma způsoby se dá inicializovat struktura?
98. Může být členskou proměnnou struktury jiná struktura?
99. Proč je dobré předávat strukturu odkazem i v případech, kdy ji nehodláte měnit?
100. Jaký je rozdíl mezi strukturou a třídou?

Správné odpovědi

Kapitola 1

1. Počítačový program je návod, ve kterém programátor počítači krok za krokem vysvětluje, co má dělat.
2. Počítače zvládnou uložit větší objem informací, umí si tyto informace rychleji a přesněji vybavit a jsou i rychlejší a přesnější počtáři.
3. Programovací jazyk je jazyk, který nemá daleko od struktury a syntaxe přirozeného jazyka a slouží pro psaní počítačových programů.
4. C++ se hodí umět, protože je to široce rozšířený jazyk ve firmách i ve školách a protože se mu další jazyky podobají (viz například Javu nebo C#).
5. Funkce je skupina souvisejících příkazů, které společně plní nějaký úkol.
6. Každý program v C++ má právě jednu funkci `main`. Ani více, ani méně.
7. Standardní knihovna je řada funkcí a objektů, které implementují běžně používané úkoly, například výstup na obrazovku (`cout`).
8. Direktiva `#include` vloží do programu definice ze zadaného souboru.
9. Preprocesor je program, který projde zadaný zdrojový kód a všechny direktivy `#include` nahradí obsahem odkazovaných souborů.
10. Překladač je program, který vezme zdrojový kód zpracovaný preprocesorem a přeloží ho do jazyka procesoru. Výsledný kód se ukládá do takzvaných objektových souborů.
11. Linker je program, který spojí objektové soubory s potřebnými částmi použitých knihoven a vytvoří spustitelný soubor.

Kapitola 2

1. Procesor má nejblíže do vyrovnávací paměti neboli cache (pomineme-li paměťové registry přímo v procesoru), protože ta je přímo jeho součástí.
2. Pevný disk není dočasná paměť, data na něm vydrží i po vypnutí počítače.
3. Na jednu paměťovou adresu se dá uložit právě jeden bajt.
4. Není, velikost datových typů se liší v závislosti na operačním systému a překladači.
5. Rozsah datového typu udává nejnižší a nejvyšší hodnotu, kterou typ zvládne popsat.
6. Datové typy bez znaménka obsahují jen kladné nebo nulové hodnoty, datové typy se znaménkem mají i záporné hodnoty.
7. 5,1E–3 = 0,0051.

8. ASCII kód znaku je jeho číslo v tabulce ASCII.
9. Operátor `sizeof` vrací velikost datového typu v bajtech.
10. Řetězcový literál je řetězec (většinou uzavřený ve dvojitých uvozovkách), který se už nijak dál nevyhodnocuje.
11. Výraz je kus kódu, který vrací nějakou hodnotu.

Kapitola 3

1. Deklarace proměnné vyhradí v paměti místo na uložení informací a dá vám jméno, pod kterým se na tuto část paměti můžete odkazovat.
2. Ne. I kdybyste proměnnou deklarovali hned za řádkem, na kterém ji poprvé použijete, překladač si bude stěžovat na nedeklarovaný identifikátor. Zdrojový kód totiž čte odshora dolů, takže když narazí na první použití proměnné, deklaraci ještě nezná.
3. Ano, ale proměnné musí být stejného typu.
4. Jmenná konvence je jednotný způsob pojmenování proměnných.
5. Adresovací operátor `&` vrací adresu proměnné v paměti, operátor `sizeof` vrací velikost proměnné v paměti.
6. Inicializace je nastavení hodnoty proměnné současně s její deklarací.
7. Přetečení je situace, která nastane, když do proměnné uložíte příliš velkou nebo příliš malou hodnotu.
8. Dojde k chybě překladu.
9. Čtení z objektu `cin` probíhá za běhu programu.
10. Ano, z objektu `cin` se operátorem `>>` dá načíst několik proměnných různého datového typu najednou.

Kapitola 4

1. Dělení má dva operátory: podíl `/` a zbytek po dělení `%`.
2. Na řetězcích i číslech funguje sčítání, `+`.
3. S desetinnými čísly nepracuje zbytek po dělení.
4. Nulou se nedá dělit, takže nula nemůže být druhým operandem `/` a `%`.
5. `celkem += 2`
6. Výraz `2+3*4` má hodnotu 14, nikoliv 20, protože násobení má přednost před sčítáním.
7. Výraz `8/2*4` má hodnotu 16, nikoliv 1, protože dělení i násobení mají stejnou prioritu a levou asociativitu.
8. Výraz `10/4` má hodnotu 2, nikoliv 2,5. Když jsou oba operandy celočíselné, použije se celočíselné dělení.
9. Na rozdíl od mnoha jiných jazyků nemá C++ samostatný operátor pro umocňování. Umocňuje se funkcí `pow` definovanou v souboru `cmath`.
10. Algoritmus je logický postup, který krok za krokem říká, jak vyřešit nějaký problém.

Kapitola 5

1. Relační výrazy mají dva operandy.
2. Vývojový diagram je grafický model toku programu.
3. Za klíčovým slovem `if` musí následovat výraz booleovského typu.
4. Každý příkaz `if/else if/else` musí obsahovat právě jednu větev `if`.
5. Každý příkaz `if/else if/else` může obsahovat více větví typu `else if`.
6. Z příkazu `if/else if/else` můžete vynechat větev `else`.
7. Za klíčovým slovem `switch` musí následovat celočíselný výraz.
8. Ano, příslušný znak se automaticky převede na svůj celočíselný kód podle ASCII.
9. Ne, za klíčovým slovem `case` musí následovat konstanta.
10. Podobnou roli jako `else` v `if` hraje u příkazu `switch` klíčové slovo `default`.

Kapitola 6

1. Ano, operátory `&&` a `||` se dají nahradit vnořenými podmíněnými příkazy.
2. Ano, větev `if` i větev `else` mohou obsahovat další podmíněný příkaz.
3. Logický součin (`&&`) je pravdivý pouze tehdy, když jsou pravdivé oba jeho operandy.
4. Logický součet (`||`) je nepravdivý pouze v případech, kdy jsou nepravdivé oba jeho operandy.
5. Pravdivost svého operandu převrací operátor negace (`!`).
6. Ano, u logické proměnné je výraz `(promenna)` totéž co `(promenna == true)`.
7. Logická negace je unární.
8. Logická negace má vyšší prioritu než relační operátory.
9. Operátor `&&` má vyšší prioritu než `||`.
10. Ano, konstanty `true` a `false` se dají použít za klíčovým slovem case, protože obě mají odpovídající číslené hodnoty `1` a `0`.

Kapitola 7

1. Operátor `++` zvýší hodnotu proměnné o jedničku.
2. Operátor `--` zvýší hodnotu proměnné o jedničku.
3. Pokud máme v proměnné `cislo` hodnotu 5, příkaz `cout << --cislo` vypíše čtyřku, protože prefixová varianta operátoru `--` vrací už sníženou hodnotu proměnné.
4. Iterace je jedna otáčka cyklu.
5. V první části závorek za klíčovým slovem `for` se běžně inicializuje čítač.
6. V druhé části závorek za klíčovým slovem `for` je podmínka, která musí platit, aby cyklus pokračoval.
7. Ve třetí části závorek za klíčovým slovem `for` se většinou mění hodnota čítače.

8. Ano, libovolná část závorek za klíčovým slovem `for` může být prázdná.
9. Příkaz `break` okamžitě ukončí smyčku.
10. Příkaz `continue` ukončí aktuální iteraci smyčky a začne další.
11. Kdybyste chtěli pomocí dvou cyklů tisknout tabulku, sloupce by měl na starosti vnitřní cyklus.

Kapitola 8

1. Alespoň jednou se vždy provede smyčka `do while`.
2. Pokud počet iterací znáte předem, nejlepší je zvolit cyklus `for`.
3. Do závorek za klíčové slovo `while` patří podmínka.
4. Ano. Pokud je v závorkách za `while` hodnota `true`, cyklus poběží, dokud ho nepřerušíte příkazem `break`.
5. Ano. V podmínce cyklu může být libovolně složitý logický výraz poskládaný operátory `&&`, `||` a podobně.
6. Příkaz `break` funguje u cyklu `while` stejně jako u cyklu `for`, okamžitě ukončí zpracování cyklu.
7. Příkaz `continue` dělá u `while` totéž co u `for`, přeruší aktuální iteraci cyklu a pustí se do další.
8. Příznak je proměnná typu `bool`.
9. Kdybyste chtěli dvěma vnořenými smyčkami `while` vytisknout několik řádků a sloupců, řádky by měla na starost vnější smyčka a sloupce smyčka vnitřní.
10. Ne. Platnost proměnné deklarované v rámci cyklu `while` končí s uzavírací složenou závorkou těla cyklu.

Kapitola 9

1. Životnost proměnné říká, kdy proměnná zanikne. Rozsah platnosti říká, kde všude se na proměnnou můžete odkazovat.
2. Ne. Pokud funkci definujete, ještě než ji poprvé zavoláte, prototypovat ji nemusíte. Prototypování všech funkcí s výjimkou `main` je ale docela dobrý zvyk.
3. Ne, funkce nemusí mít žádné parametry. V takovém případě můžete do závorek za názvem funkce v její hlavičce uvést klíčové slovo `void`.
4. Ano, funkce může mít víc parametrů. V takovém případě se parametry oddělují čárkou.
5. Ne. Pokud nějakou proměnnou z funkce `main` předáte hodnotou jiné funkci a ta ji změní, hodnota proměnné v `main` zůstane nedotčena.
6. Ano. Pokud nějakou proměnnou z funkce `main` předáte odkazem jiné funkci a ta ji změní, změní se i hodnota proměnné v `main`.
7. Ne. Pokud funkce nevrací žádnou hodnotu, před její název se píše klíčové slovo `void`.
8. Ne, funkce má vždy nejvýš jednu návratovou hodnotu.

9. Ano, funkce nemusí mít žádné parametry ani nemusí nic vracet.
10. Ano, funkce může mít parametry a zároveň vracet nějakou hodnotu.

Kapitola 10

1. Ne, všechny prvky pole musí být stejného typu. Pole může mít typ `int`, `float` i `char`, ale nemůže obsahovat prvky různých typů zároveň.
2. První prvek pole má index nula.
3. Poslední prvek pole má index o jedničku menší, než je velikost pole.
4. Inicializace je přiřazení v rámci deklarace.
5. Velikost pole se dá určit explicitně, tedy v hranatých závorkách za jménem pole, anebo si ji překladač zjistí sám podle délky inicializačního seznamu.
6. Nulový znak označuje konec céčkového řetězce. Například `cout` podle něj pozná, kdy má přestat vypisovat.
7. Hodnotou proměnné typu pole je adresa prvního prvku pole v paměti.
8. Ne. Nulovým znakem se ukončují pouze céčkové řetězce. Pokud máte ve znakovém poli uložené samostatné hodnoty, které nebudete používat jako řetězec (například názvy známek), pole znakem null ukončovat nemusíte.
9. Funkce `get` načte vstup až po konec řádku, funkce `getline` načte vstup včetně konce řádku.
10. Když předáváte pole jako parametr nějaké funkci, předáváte ho adresou.

Kapitola 11

1. Ukazatel je proměnná, jejíž hodnotou je adresa jiné proměnné.
2. Bez ukazatelů se neobejdete při dynamické alokaci paměti.
3. Jediný rozdíl je v tom, že deklarace ukazatele na `int` obsahuje navíc hvězdičku. Ta se píše těsně před jméno proměnné nebo za jméno datového typu.
4. Datový typ ukazatele říká, jaký typ proměnné je uložený na odkazované adrese.
5. `NULL` je konstanta definovaná v několika souborech, například v souboru `iostream`. Jde o výchozí hodnotu proměnných typu ukazatel. Na `NULL` je dobré inicializovat ukazatele, které ještě nemají žádnou „opravdovou" hodnotu. Kdybyste se je pak později v kódu omylem pokusili dereferencovat, program spadne a na chybu snadno přijdete.
6. Adresu proměnné vrací adresovací operátor `&`.
7. Dereferenčním operátorem (`*`) se dá získat nebo změnit hodnota uložená na adrese, kterou ukazatel obsahuje.
8. Ano, ukazatel může v průběhu programu ukazovat na různé adresy, alespoň pokud není deklarovaný jako `const`.
9. Ano, na jednu adresu může ukazovat víc ukazatelů.
10. Operátor `++` zvýší adresu uloženou v ukazateli o velikost datového typu ukazatele.
11. Operátor `new` slouží k dynamické alokaci paměti, operátor `delete` tuto paměť zase uvolní.

Kapitola 12

1. Úvodní znak konce řádku ignoruje operátor >>.
2. Konec řádku ve vstupním bufferu nezůstane po čtení členskou funkcí `getline`.
3. Funkci `cin.ignore` je potřeba volat po čtení operátorem >>, protože po něm vždy ve vstupním bufferu zůstane konec řádku.
4. Funkci `cin.ignore` byste neměli volat po čtení členskou funkcí getline, protože ta konec řádku ve vstupním bufferu nenechává.
5. Parametrem funkce `isdigit` je znak, tedy `char`.
6. Parametrem funkce `atoi` je céčkový řetězec.
7. Funkce `atoi` neumí s řetězci typu `string` pracovat přímo, pracuje pouze s céčkovými řetězci. Datový typ `string` ale nabízí členskou funkci `c_str`, která vám vrátí céčkový řetězec. Ten pak můžete funkci `atoi` předložit bez problémů.
8. Funkce definované v knihovně `cctype` (například `toupper` nebo `isdigit`) slouží pro práci se znaky.
9. Funkce definované v knihovně `cstdlib` (například `atoi` nebo `itoa`) slouží pro práci s céčkovými řetězci.
10. Ne. Operátorem = byste přiřadili jen adresu řetězce, nikoliv jeho obsah.

Kapitola 13

1. „Trvale“ uložená jsou data, která přežijí ukončení programu.
2. Soubor je sbírka dat uložená v trvalé paměti, například na pevném disku nebo CD.
3. Soubory se dají rozdělit na textové a binární.
4. Pokud chcete pracovat se soubory, musíte vložit knihovnu `fstream`.
5. Pro čtení i zápis se dá použít datový typ `fstream`.
6. Soubor se dá otevřít členskou funkcí `open` a konstruktorem.
7. Otevřením souboru vznikne komunikační kanál mezi samotným souborem a datovým proudem ve vašem programu.
8. Uzavřením souboru uvolníte systémové prostředky potřebné pro udržování komunikačního kanálu mezi souborem a datovým proudem ve vašem programu.
9. Konstruktor je funkce, která se automaticky volá při vzniku nového objektu.
10. Pro detekci konce textového souboru je spolehlivější funkce `fail`.
11. Souborové proudy je lepší předávat odkazem, nikoliv hodnotou.

Kapitola 14

1. Objektové programování umožňuje modelovat složité objekty ze skutečného světa a usnadňuje opětovné použití hotového, dobře otestovaného kódu.

2. Příkladem skrývání informací je třeba označení členských proměnných za soukromé a jejich zpřístupnění pomocí veřejných členských funkcí.
3. Dědičnost je vztah typu *je*, student *je* člověk.
4. Skládání (agregace) je vztah typu *má*, auto *má* motor a člověk *má* datum narození.
5. Struktura je programátorem definovaný datový typ, kterým se dají seskupit logicky související proměnné libovolných datových typů.
6. Deklarace struktury je deklarace datového typu. Pokud chcete instanci, musíte deklarovat proměnnou tohoto typu.
7. Struktura se dá inicializovat seznamem a konstruktorem.
8. Ano, součástí struktury může být další struktura. Například struktura `Osoba` může mít členskou proměnnou `narozeni` typu `Datum`, kde datum je další struktura.
9. Struktury je dobré předávat odkazem za všech okolností. I když ji nehodláte měnit, předávání odkazem je paměťově efektivnější než předávání hodnotou, při kterém se potenciálně velká struktura musí kopírovat. Pokud předanou strukturu nechcete měnit, přidejte před parametr ještě klíčové slovo `const`, aby vás překladač na případný pokus o změnu upozornil.
10. Členské proměnné struktury jsou standardně veřejné, zatímco členské proměnné třídy jsou standardně soukromé.

Závěrečný test

1. Počítačový program je návod, ve kterém programátor počítači krok za krokem vysvětluje, co má dělat.
2. Programovací jazyk je jazyk, který nemá daleko od struktury a syntaxe přirozeného jazyka a slouží pro psaní počítačových programů.
3. Funkce je skupina souvisejících příkazů, které společně plní nějaký úkol.
4. Každý program v C++ má právě jednu funkci `main`. Ani více, ani méně.
5. Standardní knihovna je řada funkcí a objektů, které implementují běžně používané úkoly, například výstup na obrazovku (`cout`).
6. Direktiva `#include` vloží do programu definice ze zadaného souboru.
7. Preprocesor je program, který projde zadaný zdrojový kód a všechny direktivy `#include` nahradí obsahem odkazovaných souborů.
8. Překladač je program, který vezme zdrojový kód zpracovaný preprocesorem a přeloží ho do jazyka procesoru. Výsledný kód se ukládá do takzvaných objektových souborů.
9. Linker je program, který spojí objektové soubory s potřebnými částmi použitých knihoven a vytvoří spustitelný soubor.
10. Pevný disk není dočasná paměť, data na něm vydrží i po vypnutí počítače.
11. Na jednu paměťovou adresu se dá uložit právě jeden bajt.
12. Není, velikost datových typů se liší v závislosti na operačním systému a překladači.

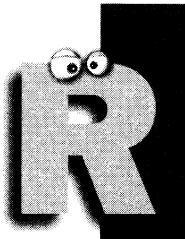
13. Datové typy bez znaménka obsahují jen kladné nebo nulové hodnoty, datové typy se znaménkem mají i záporné hodnoty.
14. ASCII kód znaku je jeho číslo v tabulce ASCII.
15. Řetězcový literál je řetězec (většinou uzavřený ve dvojitých uvozovkách), který se už nijak dál nevyhodnocuje a nemění.
16. Výraz je kus kódu, který vrací nějakou hodnotu.
17. Deklarace proměnné vyhradí v paměti místo na uložení informací a dá vám jméno, pod kterým se na tuto část paměti můžete odkazovat.
18. Ne. I kdybyste proměnnou deklarovali hned za řádkem, na kterém ji poprvé použijete, překladač si bude stěžovat na nedeklarovaný identifikátor. Zdrojový kód totiž čte odshora dolů, takže když narazí na první použití proměnné, deklaraci ještě nezná.
19. Adresovací operátor `&` vrací adresu proměnné v paměti, operátor `sizeof` vrací velikost proměnné v paměti.
20. Inicializace je nastavení hodnoty proměnné současně s její deklarací.
21. Přetečení je situace, která nastane, když do proměnné uložíte příliš velkou nebo příliš malou hodnotu.
22. Čtení z objektu `cin` probíhá za běhu programu.
23. Dělení má dva operátory: podíl `/` a zbytek po dělení `%`.
24. S desetinnými čísly nepracuje zbytek po dělení.
25. Nulou se nedá dělit, takže nula nemůže být druhým operandem `/` a `%`.
26. `celkem += 2`
27. Výraz `2+3*4` má hodnotu 14, nikoliv 20, protože násobení má přednost před sčítáním.
28. Výraz `8/2*4` má hodnotu 16, nikoliv 1, protože dělení i násobení mají stejnou prioritu a levou asociativitu.
29. Výraz `10/4` má hodnotu 2, nikoliv 2,5. Když jsou oba operandy celočíselné, použije se celočíselné dělení.
30. Na rozdíl od mnoha jiných jazyků nemá C++ samostatný operátor pro umocňování. Umocňuje se funkcí `pow` definovanou v souboru `cmath`.
31. Algoritmus je logický postup, který krok za krokem říká, jak vyřešit nějaký problém.
32. Relační výrazy mají dva operandy.
33. Za klíčovým slovem `if` musí následovat výraz booleovského typu.
34. Každý příkaz `if/else` `if/else` musí obsahovat právě jednu větev `if`.
35. Každý příkaz `if/else` `if/else` může obsahovat více větví typu `else if`.
36. Z příkazu `if/else` `if/else` můžete vynechat větev `else`.
37. Za klíčovým slovem `switch` musí následovat celočíselný výraz.
38. Ne, za klíčovým slovem `case` musí následovat konstanta.
39. Podobnou roli jako `else` v `if` hraje u příkazu `switch` klíčové slovo `default`.
40. Ano, operátory `&&` a `||` se dají nahradit vnořenými podmíněnými příkazy.

41. Logický součin (`&&`) je pravdivý pouze tehdy, když jsou pravdivé oba jeho operandy.
42. Logický součet (`||`) je nepravdivý pouze v případech, kdy jsou nepravdivé oba jeho operandy.
43. Pravdivost svého operandu převrací operátor negace (`!`).
44. Operátor `++` zvýší hodnotu proměnné o jedničku.
45. Operátor `--` zvýší hodnotu proměnné o jedničku.
46. Pokud máme v proměnné `cislo` hodnotu 5, příkaz `cout << --cislo` vypíše čtyřku, protože prefixová varianta operátoru `--` vrací už sníženou hodnotu proměnné.
47. Iterace je jedna otáčka cyklu.
48. V první části závorek za klíčovým slovem `for` se běžně inicializuje čítač.
49. V druhé části závorek za klíčovým slovem `for` je podmínka, která musí platit, aby cyklus pokračoval.
50. Ve třetí části závorek za klíčovým slovem `for` se většinou mění hodnota čítače.
51. Ano, libovolná část závorek za klíčovým slovem `for` může být prázdná.
52. Příkaz `break` okamžitě ukončí smyčku.
53. Příkaz `continue` ukončí aktuální iteraci smyčky a začne další.
54. Kdybyste chtěli pomocí dvou cyklů tisknout tabulku, sloupce by měl na starosti vnitřní cyklus.
55. Alespoň jednou se vždy provede smyčka `do while`.
56. Pokud počet iterací znáte předem, nejlepší je zvolit cyklus `for`.
57. Do závorek za klíčové slovo `while` patří podmínka.
58. Příznak je proměnná typu `bool`.
59. Životnost proměnné říká, kdy proměnná zanikne. Rozsah platnosti říká, kde všude se na proměnnou můžete odkazovat.
60. Ne. Pokud funkci definujete, ještě než ji poprvé zavoláte, prototypovat ji nemusíte. Prototypování všech funkcí s výjimkou `main` je ale docela dobrý zvyk.
61. Ne, funkce nemusí mít žádné parametry. V takovém případě můžete do závorek za názvem funkce v její hlavičce uvést klíčové slovo `void`.
62. Ano, funkce může mít víc parametrů. V takovém případě se parametry oddělují čárkou.
63. Ne. Pokud nějakou proměnnou z funkce `main` předáte hodnotou jiné funkci a ta ji změní, hodnota proměnné v `main` zůstane nedotčena.
64. Ano. Pokud nějakou proměnnou z funkce `main` předáte odkazem jiné funkci a ta ji změní, změní se i hodnota proměnné v `main`.
65. Ne. Pokud funkce nevrací žádnou hodnotu, před její název se píše klíčové slovo `void`.
66. Ne, funkce má vždy nejvýš jednu návratovou hodnotu.
67. Ano, funkce nemusí mít žádné parametry ani nemusí nic vracet.
68. Ano, funkce může mít parametry a zároveň vracet nějakou hodnotu.

69. Ne, všechny prvky pole musí být stejného typu. Pole může mít typ `int`, `float` i `char`, ale nemůže obsahovat prvky různých typů zároveň.
70. První prvek pole má index nula.
71. Poslední prvek pole má index o jedničku menší, než je velikost pole.
72. Velikost pole se dá určit explicitně, tedy v hranatých závorkách za jménem pole, anebo si ji překladač zjistí sám podle délky inicializačního seznamu.
73. Nulový znak označuje konec céčkového řetězce. Například `cout` podle něj pozná, kdy má přestat vypisovat.
74. Hodnotou proměnné typu pole je adresa prvního prvku pole v paměti.
75. Když předáváte pole jako parametr nějaké funkci, předáváte ho adresou.
76. Ukazatel je proměnná, jejíž hodnotou je adresa jiné proměnné.
77. Jediný rozdíl je v tom, že deklarace ukazatele na `int` obsahuje navíc hvězdičku. Ta se píše těsně před jméno proměnné nebo za jméno datového typu.
78. Datový typ ukazatele říká, jaký typ proměnné je uložený na odkazované adrese.
79. `NULL` je konstanta definovaná v několika souborech, například v souboru `iostream`. Jde o výchozí hodnotu proměnných typu ukazatel. Na `NULL` je dobré inicializovat ukazatele, které ještě nemají žádnou „opravdovou" hodnotu. Kdybyste se je pak později v kódu omylem pokusili dereferencovat, program spadne a na chybu snadno přijdete.
80. Adresu proměnné vrací adresovací operátor `&`.
81. Dereferenčním operátorem (`*`) se dá získat nebo změnit hodnota uložená na adrese, kterou ukazatel obsahuje.
82. Operátor `++` zvýší adresu uloženou v ukazateli o velikost datového typu ukazatele.
83. Operátor `new` slouží k dynamické alokaci paměti, operátor `delete` tuto paměť zase uvolní.
84. Ne. Operátorem `=` byste přiřadili jen adresu řetězce, nikoliv jeho obsah.
85. „Trvale" uložená jsou data, která přežijí ukončení programu.
86. Soubor je sbírka dat uložená v trvalé paměti, například na pevném disku nebo CD.
87. Pokud chcete pracovat se soubory, musíte vložit soubor `fstream`.
88. Pro čtení i zápis se dá použít datový typ `fstream`.
89. Soubor se dá otevřít členskou funkcí `open` a konstruktorem.
90. Otevřením souboru vznikne komunikační kanál mezi samotným souborem a datovým proudem ve vašem programu.
91. Uzavřením souboru uvolníte systémové prostředky potřebné pro udržování komunikačního kanálu mezi souborem a datovým proudem ve vašem programu.
92. Konstruktor je funkce, která se automaticky volá při vzniku nového objektu.
93. Datové proudy je lepší předávat odkazem.
94. Dědičnost je vztah typu *je*, student *je* člověk.
95. Skládání je vztah typu *má*, auto *má* motor a člověk *má* datum narození.

96. Struktura je programátorem definovaný datový typ, kterým se dají seskupit logicky související proměnné libovolných datových typů.
97. Struktura se dá inicializovat seznamem a konstruktorem.
98. Ano, součástí struktury může být další struktura. Například struktura `Osoba` může mít členskou proměnnou `narozeni` typu `Datum`, kde datum je další struktura.
99. Struktury je dobré předávat odkazem za všech okolností. I když ji nehodláte měnit, předávání odkazem je paměťově efektivnější než předávání hodnotou, při kterém se potenciálně velká struktura musí kopírovat. Pokud předanou strukturu nechcete měnit, přidejte před parametr ještě klíčové slovo `const`, aby vás překladač na případný pokus o změnu upozornil.
100. Členské proměnné struktury jsou standardně veřejné, zatímco členské proměnné třídy jsou standardně soukromé.

Rejstřík



E

F

G

H

CH

I

J

K

L

M

N

O

P

R

Ř

S

T

U

V

W

Z

Ž