**C++ tutor**

**Lukáš Churý a kol.**

[C++ - Úvod 6](#_Toc475650712)

[C++ oproti C 6](#_Toc475650713)

[Historie jazyka 6](#_Toc475650714)

[Filosofie programování v C 7](#_Toc475650715)

[Objektově orientované programování 7](#_Toc475650716)

[Generické programování 7](#_Toc475650717)

[C++ 8](#_Toc475650718)

[Přenositelnost a standardy 8](#_Toc475650719)

[Mechanismus vytváření programu 8](#_Toc475650720)

[Jak tedy postupovat 9](#_Toc475650721)

[Dev-C++ 9](#_Toc475650722)

[Visual C++ 6.0 9](#_Toc475650723)

[GCC 9](#_Toc475650724)

[Code::Blocks 10](#_Toc475650725)

[Pojmy 10](#_Toc475650726)

[Lukáš Churý 10](#_Toc475650727)

[C++ - 0. lekce 11](#_Toc475650728)

[Předmluva 11](#_Toc475650729)

[Doporučená literatura 11](#_Toc475650730)

[Nedoporučená literatura 14](#_Toc475650731)

[Ostatní literatura o jazycích C a C++ 15](#_Toc475650732)

[Literatura o standardní šablonové knihovně (STL) 21](#_Toc475650733)

[C++ - 1. lekce 23](#_Toc475650734)

[První program 23](#_Toc475650735)

[Zdrojový program 24](#_Toc475650736)

[Překlad, sestavení a spuštění 24](#_Toc475650737)

[Co jsme naprogramovali 24](#_Toc475650738)

[Funkce main() 25](#_Toc475650739)

[Vstup a výstup 25](#_Toc475650740)

[Hlavičkový soubor 27](#_Toc475650741)

[Zápisy programu 28](#_Toc475650742)

[Chyby a varování 28](#_Toc475650743)

[Direktiva #include 28](#_Toc475650744)

[Ještě o překladu programu 28](#_Toc475650745)

[Úkol č. 1 29](#_Toc475650746)

[C++ - 1. lekce - dodatek 29](#_Toc475650747)

[Některá pravidla při programování 29](#_Toc475650748)

[Jména proměnných 29](#_Toc475650749)

[Pravidla při tvorbě jména proměnné 29](#_Toc475650750)

[Klíčová slova v  C++ 31](#_Toc475650751)

[C++ - 2. lekce 32](#_Toc475650752)

[Celočíselné, reálné a jiné typy 32](#_Toc475650753)

[Celočíselné typy 32](#_Toc475650754)

[Reálné typy 34](#_Toc475650755)

[Proměnné 34](#_Toc475650756)

[Definice 34](#_Toc475650757)

[Inicializace 35](#_Toc475650758)

[Operátory 37](#_Toc475650759)

[Přetypování 39](#_Toc475650760)

[Zkracování 40](#_Toc475650761)

[Postranní efekty 40](#_Toc475650762)

[Vstupy 41](#_Toc475650763)

[Úkol č.2: 43](#_Toc475650764)

[C++ - 2. lekce - dodatek 44](#_Toc475650765)

[Priorita operátorů 44](#_Toc475650766)

[Nejčastější chyby v úkolech: 47](#_Toc475650767)

[C++ - 3. lekce 48](#_Toc475650768)

[Řetězce 48](#_Toc475650769)

[Cykly 50](#_Toc475650770)

[a) for 51](#_Toc475650771)

[b) WHILE 53](#_Toc475650772)

[c) do while 54](#_Toc475650773)

[d) if else 55](#_Toc475650774)

[Logické výrazy 56](#_Toc475650775)

[Logický operátor NEBO || 56](#_Toc475650776)

[Logický operátor A && 57](#_Toc475650777)

[Logický operátor NE ! 57](#_Toc475650778)

[Úkol 57](#_Toc475650779)

[C++ - 3. lekce - test 59](#_Toc475650780)

[Zkouška č. 1 59](#_Toc475650781)

[C++ - 4. lekce 60](#_Toc475650782)

[Nový typ bool 60](#_Toc475650783)

[Kvalifikátor const 61](#_Toc475650784)

[Úvod do polí 61](#_Toc475650785)

[Inicializace prvků 63](#_Toc475650786)

[Volání prvků 64](#_Toc475650787)

[Zajímavosti s řetězci 64](#_Toc475650788)

[Nějaké další vychytávky 65](#_Toc475650789)

[Řádkově orientovaný vstup: getline() a get() 66](#_Toc475650790)

[Proč tedy používáme cin.get(), když máme cin.getline()? 68](#_Toc475650791)

[Míchání numerického a řetězového vstupu 69](#_Toc475650792)

[Trošku lidštěji 70](#_Toc475650793)

[Úkol č. 4 70](#_Toc475650794)

[C++ - 5. lekce 71](#_Toc475650795)

[Struktury 71](#_Toc475650796)

[Úvod do struktur 71](#_Toc475650797)

[Deklarace struktury: 71](#_Toc475650798)

[Inicializace struktury 72](#_Toc475650799)

[Volání některé proměnné ze struktury 72](#_Toc475650800)

[Pole struktur 76](#_Toc475650801)

[Bitová pole 76](#_Toc475650802)

[Uniony 77](#_Toc475650803)

[Úkol: 77](#_Toc475650804)

[C++ - 6. lekce 78](#_Toc475650805)

[Funkce – programové moduly v C++ 78](#_Toc475650806)

[Definice funkce 78](#_Toc475650807)

[Vytváření prototypů 80](#_Toc475650808)

[Volání funkce 81](#_Toc475650809)

[Parametry funkcí a předávání hodnotou 83](#_Toc475650810)

[Násobné parametry 84](#_Toc475650811)

[Úkol: 85](#_Toc475650812)

[C++ - 7. lekce 86](#_Toc475650813)

[Funkce a pole – vytváření 86](#_Toc475650814)

[Naplnění pole 87](#_Toc475650815)

[Zobrazení pole a jeho ochrana pomocí const 88](#_Toc475650816)

[Modifikace pole 89](#_Toc475650817)

[Sestavení 89](#_Toc475650818)

[Funkce a struktury 92](#_Toc475650819)

[Předání a navrácení struktur 92](#_Toc475650820)

[Příklady 94](#_Toc475650821)

[Rekurze 98](#_Toc475650822)

[Úkol 100](#_Toc475650823)

[C++ - 8. lekce 101](#_Toc475650824)

[Dodatek k cyklu FOR 101](#_Toc475650825)

[Operátor „ , “ (čárka) 102](#_Toc475650826)

[Kopírování a spojování řetězců, jejich velikost a délka 102](#_Toc475650827)

[Zjištění délky řetězce 105](#_Toc475650828)

[Zjištění velikosti řetězce 106](#_Toc475650829)

[Alias jména 106](#_Toc475650830)

[Časová prodleva 107](#_Toc475650831)

[Dvourozměrné pole 110](#_Toc475650832)

[Vytváření 2D pole 110](#_Toc475650833)

[Inicializace 2D pole 110](#_Toc475650834)

[Volání prvku 111](#_Toc475650835)

[Operátor „?“ 111](#_Toc475650836)

[Příkaz SWITCH 112](#_Toc475650837)

[Příkazy break a continue 116](#_Toc475650838)

[Příkaz goto 117](#_Toc475650839)

[Knihovna cctype 117](#_Toc475650840)

[Symbolické konstanty climits 120](#_Toc475650841)

[Chyby, které možná děláte 123](#_Toc475650842)

[Vícesouborové projekty 124](#_Toc475650843)

[Úkol 125](#_Toc475650844)

[C++ - 9. lekce 126](#_Toc475650845)

[Ukazatele volná paměť 126](#_Toc475650846)

[Ukazatele a filozofie C++ 128](#_Toc475650847)

[Deklarování a inicializace ukazatelů 130](#_Toc475650848)

[Ukazatele a čísla 132](#_Toc475650849)

[Úkol 132](#_Toc475650850)

[C++ - 10. lekce 134](#_Toc475650851)

[Alokace paměti pomocí operátoru new 134](#_Toc475650852)

[Uvolnění paměti pomocí operátoru delete 137](#_Toc475650853)

[Úkol č.10 138](#_Toc475650854)

[C++ - 11. lekce 139](#_Toc475650855)

[Automatická, statická a volná paměť 139](#_Toc475650856)

[Automatické proměnné 139](#_Toc475650857)

[Statická paměť 139](#_Toc475650858)

[Volná paměť 140](#_Toc475650859)

[Ukazatele na funkce 140](#_Toc475650860)

[Získání adresy funkce 140](#_Toc475650861)

[Deklarování ukazatele na funkci 141](#_Toc475650862)

[Použití ukazatelů na vyvolání funkce 143](#_Toc475650863)

[Úkol č. 11 145](#_Toc475650864)

[C++ - 12. lekce 146](#_Toc475650865)

[Standardní paramatery 146](#_Toc475650866)

[Šablony funkcí 149](#_Toc475650867)

[Explicitní specializace 151](#_Toc475650868)

[Úkol č.12 154](#_Toc475650869)

[C++ – řetězce 154](#_Toc475650870)

[Reťazce 155](#_Toc475650871)

[Spájanie reťazcov 157](#_Toc475650872)

[Použitie reťazcov v polia 157](#_Toc475650873)

[Problémy so vstupom 159](#_Toc475650874)

[Riadkovo orientovaný vstup: getline() a get() 161](#_Toc475650875)

[Prázdne riadky a ďalšie problémy 164](#_Toc475650876)

[Zmiešanie reťazového a číselného vstupu 165](#_Toc475650877)

[Základní implementace MySQL v C/C++ 167](#_Toc475650878)

# [C++ - Úvod](http://programujte.com/clanek/2005080601-c-uvod/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**12. 4. 2005       86 732×**

**Začínáme s  C++, historie a filozofie tohoto jazyka, mechanizmus vytváření programu, jak a s čím pracovat, standardy a přenositelnost…**

<http://programujte.com/clanky/39-serial-c/>

**Jazyk C++, který je jakousi nástavbou jazyka C a s podporou OOP (objektově orientovaného programování) se stal důležitým jazykem 20. století a jeho důležitot pokračuje.**

## C++ oproti C

**Jazyk C přináší do jazyka C++ tradici efektivity, ucelený, rychlý a přenositelný jazyk. C++ spojuje tři oddělené programovací tradice: tradici procedurálního jazyka reprezentovanou C, tradici objektově orientovaného jazyka reprezentovanou rozšířením C o třídy C++ a programování pomocí předloh podporované šablonami jazyka C++. Pokud neznáte jazyk C - základní typy, operátory, řídící struktury, syntaktická pravidla, nevadí - naučíte se zde. Znáte-li C, jste připraveni se učit C++. Není to však naučení několika klíčových slov a konstrukcí. Přechod z C na C++ vyžaduje asi tolik úsilí, jako naučení samotného C. Pokud tedy přecházíte z C na C++, musíte se odnaučit některým programovým zvyklostem. Tento kurz tedy nepředpokládá jakoukoliv znalost jakéhokoliv programovacího jazyka.**

## Historie jazyka

**Mnozí si pamatují rychlý vzestup výpočetní techniky, kdy jsme přešli z děrných štítků na dnešní technologie. Takto se vyvíjely i jazyky, ne tak drasticky, ale změny byly důležité. V 70. letech jazyky C a Pascal pomohly zavést období strukturovaného programování, základní principy, které přinesly jistý řád a kázeň do oblasti, která silně potřebovala tyto vlastnosti. Kromě prostředků pro strukturované programování, jazyk C produkoval i kompaktní rychle běžící programy spolu se schopností adresovat hardwarové záležitosti, jako jsou řídící komunikační porty a ovladače disků. C se stal dominantním jazykem 80. let, poté nástup OOP zakotvením v C++ a Smalltalku. začátkem 70. let, Dennis Ritchie z Bell Laboratories pracoval na svém projektu rozvoje operačního systému UNIX. K němu potřeboval jazyk, který by byl stručný a výstižný. Tradičně se programátoři setkávají s těmito potřebami, když používají jazyk assembler, který je těsně svázán se strojovým kódem počítače. Avšak assembler je strojově orientovaný jazyk - je závislý na procesoru počítače. Avšak pokud byste chtěli přesunout program z assembleru na jiný počítač, možná byste ho museli přepsat pomocí jiného assembleru. UNIX byl ale určen pro práci na různých typech počítačů. Zvláštní programy – kompilátory – překládají vyšší programovací jazyk do strojového kódu určitého počítače. Ritchie chtěl jazyk, který by spojoval účinnost strojového jazyka na nízké úrovni s přístupem k hardwaru s vyšší úrovní všeobecnosti vyššího jazyka a přenositelnosti. Na základě starších programovacích jazyků vytvořil jazyk C.**

## Filosofie programování v C++

**Obecně se programovací jazyk zabývá dvěma pojmy - daty a algoritmy. Data představují informaci, kterou program používá a zpracovává. Algoritmy jsou metody, které program používá. Spojením dat a algoritmů vzniká samotný program. Dřívější procedurální jazyky, jako Fortran či Basic, měli většinou organizační problémy, když se program zvětšoval. Jejich programy často obsahovaly příkazy skoku, což vede k nepřehlednosti celého kódu a následná úprava byla katastrofa. A tak vyvinuli ukázanější styl programování – strukturované.**

## Objektově orientované programování

**Ačkoli principy strukturovaného programování zlepšily srozumitelnost, spolehlivost a usnadnily údržbu programů, velká část programování stále zůstávala výzvou. OOP přináší nový přístup – místo zdůrazňování algoritmů (jako tomu bylo u strukturovaného programování) OOP zdůrazňuje data. Programování OOP je mnohem více než vázání dat a metod do definice třídy, ale o tom později, nebudeme se tím teď zatěžovat.**

## Generické programování

**Generické programování je ještě další programovací postup podporovaný C++. S OOP sdílí cíl vytváření jednoduššího kódu pro znovupoužití. Programování pomocí předloh zahrnuje rozšíření jazyka, takže můžete napsat jedinou f-ci pro všeobecný typ a použít ji pro varianty různých typů. Šablony C++ poskytují mechanismus, který to provádí.**

## C++

**C++ začíná také v Bell Labs, kde počátkem 80. let vyvinul tento jazyk Bjarne Stroustrup. Jméno C++ pochází z přírůstkového operátoru C++, který přičítá k hodnotě proměnné jedničku. Připomíná tedy jasně rozšířenou verzi jazyka C. Stroustrup založil C++ na stručnosti C, přidal rysy OOP aniž by významně změnil složku C. C++ je tedy nadstavba znamenající, že každý program napsaný v C je také platným programem pro C++. Programy v C++ mohou využívat softwarové knihovny C. Knihovny jsou kolekce programových modulů, které můžete volat z programu.**

## Přenositelnost a standardy

**Pokud program napíšete na starším PC 286, nespustíte ho na pracovní stanici Sun - používá jiný procesor a jiný OS. Musíte překompilovat program za použití kompilátoru, který je navržen pro danou platformu. Pokud můžete přeložit program bez provedení změn v kódu, a ten běží bez nesnází, říkáme, že je program přenositelný. Překážek je více, například hardware. Program, který je hardwarově závislý, pravděpodobně nebude přenositelný. Postupně vznikly standardy ANSI a ISO, které spojili svoje úsilí vytvořit standardy a zdůrazňovaly tak C++ jako široce rozšířitelný jazyk.**

## Mechanismus vytváření programu

**Popíšeme si kroky tvorby programu:**

1. **Použijte nějaký editor na sepsání kódu a uložte ho do souboru. Toto je zdrojový kód vašeho programu (myšleno napsat pár řádků textu například do poznámkového bloku).**
2. **Přeložte zdrojový kód. To dělá zmíněný kompilátor – přeloží zdrojový kód do strojového jazyka. Nyní máte objektový kód vašeho programu.**
3. **Sestavte objektový kód spolu s dalšími kódy (knihovnami) – funkcemi. Tímto vytvoříte spustitelnou verzi programu – proveditelný kód.**

## Jak tedy postupovat

**My tyto kroky budeme dělat v jednom programu. Na výběr máme z více produktů – Microsoft Visual C++ 6.0, Dev-C++, Borland C++ či GNU C++ na Unixu nebo Linuxu. Přípony zdrojového kódu bývají většinou \*.cpp, \*.h, \*.c –jsou závislé na implementaci C++. My se budeme zabývat tvorbou programů pod Dev-C++, který je zdarma ke stažení. V sekci Download naleznete odkaz na odzkoušenou verzi, kterou používám já. Není nejnovější, ale spustíte na ní vše, co zde uvidíte, proto ji doporučuji. O nic ochuzeni nebudete. Dále ještě popíšu, jak na práci s Microsoft Visual C++ 6.0 –tam je to složitější –či jak na gcc.**

### Dev-C++

**Kde ho seženu: zdarma ke stažení v sekci Download
Čeština: ano
Velikost: cca 7,6 MB
Velmi kvalitní editor i kompilátor dohromady. Je zdarma a navíc česky. Spustíme program, Klikneme na Program → Nový → Zdrojový kód nebo použijeme zkratku v navigační liště či Ctrl + N. Do okna vepíšeme náš zdrojový kód a stiskneme F9. Budeme dotázáni, kam chceme uložit zdrojový soubor, objektový kód a nakonec i spustitelný program (.exe), poté se program sám zkompiluje a zároveň se nám i spustí.**

### Visual C++ 6.0

**Kde ho seženu: nutno zakoupit si sadu Visual Studio C++ 6.0
Čeština: ne Velikost: cca 3 CD komplet
Editor a kompilátor z dílny MS, kvalitní, obtížnější manipulace pro začátečníka, anglicky. Spustíme Visual C++ (najdeme ho ve Start nabídce pod záložkou Visual Studio 6.0), klikneme na File → New → záložka Files → vybereme C++ Source file a do okna vepíšeme zdrojový kód. Poté stiskneme F5 (zeptá se nás to, zda chceme zapnout pracovní plochu, dáme ano, poté kam uložit daný soubor a nakonec se program zkompiluje a spustí.**

### GCC

**Kde ho seženu: integrován v Linuxu Čeština: ne (starší verze částečně počeštěné)
Velikost: cca 8,5 MB Je v každé distribuci Linuxu. Vytvořil ho Richard Stallman - vůdčí osobnost Free software foundation a propagátor GNU. Používá ho integrované vývojové prostředí Kdevelop, které je zdarma. Založíme projekt, vepíšeme kód, zadáme automake, autoconfigure a compile a máme spustitelný kód. Navíc i skript configure, pomocí kterého si to kdokoli zkompiluje.**

### Code::Blocks

Kde ho seženu: zdarma ke stažení v sekci Download [http://www.codeblocks.org/](http://www.codeblocks.org/%20) Čeština: ne

Code::Blocks je jedno z nejlepších free vývojových nástrojů pro C/C++, jehož vývoj byl ukončen. Program podporuje nastavení několika kompilátorů (minGW, Digital Mars a jiné…verze ke stažení obsahuje kompilátor minGW). Dále lze v programu tvořit projekty s DirectX, OpenGL, GLUT, Irrlich, programovacím jazykem D, Ogre, Qt, SDL, wxWidgets a jinými.

## Pojmy

* **Compile – znamená přeložit kód, který jste právě napsali.**
* **Build – přeložit zdrojové soubory.**
* **Link – sestavení objektového kódu s dalšími kódy.**
* **Run nebo Execute – znamená spustit program, vykonat ho.**

****

## [Lukáš Churý](http://programujte.com/profil/1483-lukas-chury/)

**Lukáš je šéfredaktorem Programujte, vyvíjí**[**webové aplikace**](http://devtea.cz/)**, fascinuje ho umělá inteligence a je lektorem na FI MUNI, kde učí navrhovat studenty GUI. Poslední dobou se snaží posunout**[**Laser Game**](http://lasergameareny.cz/)**o stupeň výše a vyvíjí pro něj nové herní aplikace a elektroniku.**


# [C++ - 0. lekce](http://programujte.com/clanek/2005041401-c-0-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       83 246×**

* **Předmluva**
* **Doporučená literatura**
* **Nedoporučovaná literatura**
* **Ostatní literatura**
* **Literatura o standardní šabl. knihovně STL**

## Předmluva

**Učit se budeme dle standardu ISO/IEC 14882:1998.**

**Pokud něčemu nebudete rozumět, zeptejte ve**[**Fóru**](http://programujte.com/index.php?akce=diskuze&kam=diskuze&sekce=9-c-c++)**, kde vám někdo (i já) určitě odpoví, jak nejrychleji to bude možné. Pište i do komentářů k lekcím či do Diskuze, jak se vám lekce líbí, a také lekce hodnoťte – pomůže mi to při tvorbě nových lekcí.**

## Doporučená literatura

1. ***M. Virius:***[***Programování v C++***](http://tjn.fjfi.cvut.cz/~virius/publ-list.html#CPP) **Toto skriptum pokrývá programovací jazyky C a C++ v potřebném rozsahu.**
2. ***D. Louis, P. Mejzlík, M. Virius:***[***Jazyky C a C++ podle normy ANSI/ISO***](http://tjn.fjfi.cvut.cz/~virius/publ-list.html#ANSI)**Grada Publishing, Praha 1999, 644 stran.
V této knize najdete podrobný přehled standardních knihoven C a C++. V tomto směru dobře doplňuje předchozí skriptum.**
3. ***M. Virius:***[***Od C k C++***](http://tjn.fjfi.cvut.cz/~virius/publ-list.html#ODCKCPP)**Kopp, České Budějovice 2000. 230 stran.
Tato může posloužit posluchačům, kteří již znají jazyk C.**
4. ***B. Stroustrup: The C++ Programming Language. 3rd edition.* Addison-Wesley 1997. ISBN 0-201-88954-4. 910 stran. Anglicky.
Učebnice C++. Ne každému styl výkladu v této knize vyhovuje, mně ano a já ji považuji za velice dobrou. Najdete tu nejen výklad jazyka C++, ale i výklad a příklady na použití standardních knihoven.
Pozor: Tato kniha existuje ve 3 různých vydáních, které popisují tři různé verze jazyka C++.
*První vydání* hovoří o C++ z r. 1986 (Cfront 1), které nemá s dnešní podobou jazyka mnoho společného. Pokud na toto vydání někde narazíte, raději je tam ponechte, dnes už není v podstatě k ničemu (a k přednášce se naprosto nehodí).
*Druhé vydání* z r. 1991 hovoří o C++ z doby těsně před počátkem standardizace. To znamená, že se v některých drobnostech odlišuje od standardu a především vůbec nehovoří o standardních knihovnách. Toto vydání vyšlo v českém překladu v nakladatelství Ben r. 1997 pod názvem *C++ Programovací jazyk*. Pokud tuto knihu máte, nezahazujte ji, není špatná, i když je trochu zastaralá. (Na české poměry je ovšem dost drahá - stojí 699,- Kč.)
*Třetí vydání* vyšlo zhruba v době, kdy se objevil český překlad druhého vydání. V knize najdeme výklad o jazycích a knihovnách, příklady programových konstrukcí i použití knihoven. Já sám vidím jedinou nevýhodu této knihy - její cenu (cca 2500,- Kč).
Obsahově prakticky stejné je i vydání označené Special Edition, které vedle tvrdých desek a opravy několika tiskových chyb obsahuje navíc i dodatky D a E, věnované použití třídy locale a bezpečnosti při používání výjimek. Tyto dodatky jsou k dispozici na internetu na adrese**[**http://www.research.att.com/~bs/**](http://www.research.att.com/~bs/)**.**
5. ***Margaret A. Ellis, B. Stroustrup: The Annotated C++ Reference Manual.* Addison-Wesley 1994. ISBN 0-201-51459-1. 460 stran. Anglicky.
Není to učebnice, je to referenční příručka s podrobným výkladem, předpokládá znalost jazyka C++, ale najdete tu i výklad některých důležitých detailů, zdůvodnění apod. Velice dobré čtení pro pokročilejší programátory. Dnes je tato kniha bohužel již zastaralá, neboť vznikla na počátku 90. let a proto neobsahuje některé věci, které přinesl standard, nicméně i tak může pomoci v mnoha ohledech. (Ostatně stala se podkladem pro návrh standardu. Nehovoří ovšem o knihovnách.)**
6. ***M. Virius:***[***Pasti a propasti jazyka C++.***](http://tjn.fjfi.cvut.cz/~virius/publ-list.html#PASTI)**Grada Publishing, Praha 1997. ISBN 80-7169-607-2. 246 stran.
Tato kniha se zabývá běžnými programátorskými chybami, které vznikají z nepochopení významu některých konstrukcí jazyka C++, díky podobnosti s některými jinými jazyky apod. Dobré doplňkové čtení, pokud již znáte alespoň základy jazyka.
V současné době je v tisku druhé, doplňěné a rozšířené vydání, které se zabývá i šablonami, standardní šablonovou knihovnou a objektovým návrhem.**
7. ***Bruce Eckel: Myslíme v jazyku C++.* Grada Publishing, Praha 2000. ISBN 80-247-9009-2. 552 stran. (První díl)
Velice dobře napsaná učebnice C++. Výklad začíná od základních pojmů objektového programování a při seznamování s nimi používá jazyk UML. Pak následuje rychlokurs jazyka C pro účely programování v C++. Následně se věnuje jazyku C++ (používání objektů, funkcí, šablon, jmenných prostorů). Vše je vysvětlováno na velmi jednoduchých příkladech. Neobsahuje podrobnosti o šablonách, výjimky, RTTI a standardní šablonovou knihovnu - to má být spolu s tvorbou robustních aplikací ve druhém dílu (který ovšem dosud česky nevyšel).
Výraznou vadou této knihy je kvalita překladu. Například nadpisy "Převod konstruktoru", resp. "Převod operátorů" vlastně znamenají "Převod *pomocí* konstruktoru", resp. "Převod *pomocí*operátorů", což svědčí o tom, že to překládal někdo, kdo netušil, o čem mluví. Najdeme tu i slohové krásy typu "na jiných strojích může docházet k jiným výsledkům".
Poznámka: Jde o překlad druhého anglického vydání knihy *Thinking in C++*, Prentice Hall 2000, ISBN 0-13-979809-9. První anglické vydání předpokládalo znalost jazyka C (neobsahovalo kapitolu "Jazyk C v C++").**
8. ***Herb Sutter: Exceptional C++.*Addison-Wesley 2000. ISBN 0-201-61562-2. 208 stran. Anglicky.
Zaměřením se podobá knize výše zmíněné Pasti a propasti C++, je však o úroveň dál (a je také novější, takže obsahuje zkušenosti se standardem ISO 14882). Zabývá se problémy, které vznikají při používání výjimek, chybami při práci se standardní knihovnou, s alokací paměti, rozebírá vybrané návrhové vzory pro vytváření tříd a jejich hierarchií, ukazuje souvislost Koenigova vyhledávání s tzv. principem rozhraní (interface principle) a chyby, které z nedodržení tohoto principu mohou vzniknout, atd. Obsahuje 47 vybraných problémů, které ukazují chybný kód, nespolehlivý kód, neefektivní kód atd. Např. hned první problém začíná slovy: *Následující ukázka kódu obsahuje nejméně 4 chyby v práci s iterátory. Dokážete je najít?* Pak následuje jejich řešení a výklad.
Skvělé doplňkové čtení, pokud dobře znáte alespoň základy jazyka C++ včetně standardní knihovny a chcete v C++ programovat opravdu profesionálně.
Poznamenejme, že podkladem pro tuto a následující knihu se staly autorovy články z rubriky "Guru of the weak" z časopisu The C++ User Journal.**
9. ***Herb Sutter: More Exceptional C++.*Addison-Wesley 2002. ISBN 0-201-70434-X. 279 stran. Anglicky.
Navazuje na předchozí knihu; obsahuje dalších 40 problémů a jejich řešení. Zabývá se generickým programováním, standardní knihovnou, optimalizací, problémy s bezpečností při používání výjimek, objektovým návrhem, správou prostředků, ale také makry, volnými funkcemi atd. Velice dobře napsaná kniha, i když ve srovnání s předchozí upovídanější.
Stejně jako předchozí kniha představuje skvělé doplňkové čtení.**
10. ***Andrew Koenig: C Traps and Pitfalls.*Addison-Wesley 1989. ISBN 0-201-18928-8. 146 stran. Anglicky.
Čtivě napsaná kniha o častých i méně častých programátorských chybách v jazyce C (nikoli C++). Rozebírá situace, kdy se program sice přeloží, ale dělá něco jiného, než programátor očekává; vedle chyb, jako je záměna operátorů = a == nebo překročení mezí pole, rozebírá i problémy s přenositelností programu na jiné počítače nebo s funkcemi ze standardní knihovny. Obsahuje cvičení a odpovědi na ně.
Skvělé doplňkové čtení; i když by se podle data vydání mohlo zdát, že jde o počítačovou prehistorii, na její platnosti se nic nezměnilo.
Poznamenejme, že tato kniha mne inspirovala k napsání podobné knihy o C++ - zmíněných Pastí a propastí C++ - a to nejen pokud jde o zaměření, ale i pokud jde o název.**
11. ***John J. Barton, Lee R. Nackman: Scientific and Engineering C++*An Introduction with Advanced Techniques ans Examples. Addison-Wesley 1994. ISBN 0-201-53393-6. 665 stran. Anglicky.
Dobře napsaný úvod do jazyka, který předpokládá znalost buď Fortranu nebo C. Na příkladech, zaměřených především na věděcké a technické výpočty, ukazuje použití tříd, dědičnosti, polymorfizmu, šablon atd.
Nevýhodou je, že tato kniha vznikla v době, kdy ještě nebyla k dispozici standardní šablonová knihovna, a proto řeší problémy, které jsou v této knihovně vyřešeny. Přesto je to i dnes velice dobrá kniha. (Nedávno vyšla v novém vydání.)**

## Nedoporučená literatura

1. ***Martin Beran: Učebnice Borland C++.*Technická literatura BEN, Praha 1995. ISBN neuvedeno. 438 stran.
Kniha obsahuje řadu neuvěřitelně hrubých chyb, příklady, které nelze přeložit, příklady, které dělají něco jiného, než mají, atd. NEPOUŽÍVAT V ŽÁDNÉM PŘÍPADĚ!**
2. ***Jan Pokorný: Rukověť uživatele Borland C++ AF.* Plus, Praha 1992. ISBN 80-85297-39-6. 464 stran.
Zastaralé -- popisuje Borland C++ 3.1, takže výklad jazyka C++ končí u šablon, neobsahuje výjimky, prostory jmen atd. Řada věcí je vysvětlena tak, že to čtenář nepochopí (to se týká mj. i šablon), řada podstatných věcí chybí.**
3. ***Vladimír Rudolf: ABC programátora v C++.* Kopp, České Budějovice 1990, 1999. 112 stran.
Kniha obsahuje pouze základní informace o objektovém programování a o přetěžování operátorů, nic víc. Navíc ve výkladu o přetěžování operátorů je řada věcí špatně a řada informací chybí (mnohé velmi podstatné). Také ve výkladu objektů několik důležitých věcí chybí a jsou tam chyby. Šablony, výjimky, RTTI, prostory jmen a šablonová knihovna tu vůbec nejsou, výklad o proudech vychází ze zkušenosti s překladači, které se odchylovaly od tehdejšího neoficiálního standardu (Specifikace AT&T 2.0). Závěr: NEPOUŽÍVAT.**
4. ***Dirk Louis: C und C++ -- Programierung und Referenz.* Markt & Technik, Haar bei München. ISBN 3-8272-5066-8. Německy. 588 stran.
Kniha obsahuje referenční přehled C a C++ a především knihoven, ovšem bohužel s řadou chyb, občas dosti obludných. NEPOUŽÍVAT! (Překladem a podstatnou úpravou této knihy vznikla kniha**[**Jazyky C a C++ podle normy ANSI/ISO**](http://programujte.com/publ-list.html#ANSI)**, ze které se mi doufám podařilo chyby odstranit.)**
5. ***Dalibor Kačmář: Jazyk C -- učebnice pro střední školy.* Computer Press 2000. ISBN 80-7226-295-5. 185 stran.
Je to učebnice jazyka C, nikoli C++ -- to ovšem není důvod, proč ji nedoporučuji. Pokud jde o obsah, drží se osvědčeného schématu -- začne jednoduchým příkladem, který ukáže základní možnosti jazyka, a pok postupně probere všechny důležité nástroje. Problém je, že výklad je uspořádán poměrně velmi chaoticky, obsahuje řadu odkazů dopředu, řada věcí je vysvětlována v místech, kde to čtenář neočekává, některé konstrukce použité v příkladech nejsou vysvětleny a není tam ani odkaz na místo, kde budou probírány. Vůbec tu chybí např. informace o tom, že příkaz break lze použít k ukončení cyklu nebo že ukazatele lze porovnávat pomocí relačních operátorů. O parametrech funkce main se dočteme v kapitole o vstupech a výstupech, o inicializaci struktur a unií je stručná zmínka v kapitolce o polích. Vedle toho obsahuje tato kniha i věcné chyby; nejde věci opravdu závažné, snad až na tvrzení, že rozdílem dvou ukazatelů je počet bajtů ležících mezi adresami, na které ukazují, ale v učebnici -- navíc označené "pro střední školy" -- nemají chyby co dělat. Také počet gramatických chyb, především špatně umístěných čátek ve větách, je zarážející (2 -- 3 na stránce).
Nedoporučuji tuto knihu používat, neboť i když pomineme chyby, i zkušený čtenář z této knihy získá dojem, že jazyk C je nesmírně složitý -- a co teprve studenti, kteří s programováním začínají.**

## Ostatní literatura o jazycích C a C++

**Knihy v tomto oddílu jsou převážně kvalitní, nedoporučuji je ale jako literaturu k přednášce z jazyka C++. Důvodem může např. být, že jde o knihu věnovanou pouze jazyku C, nebo proto, že hovoří více o nějakém vývojovém prostředí než o C++. Mnohé z knih zde uvedených představují dobré doplňkové čtení.**

1. ***Kernighan, B. W. -- Ritchie, D. M.: The C Programming Language. Second Edition.* Prentice Hall 1988. ISBN 0-13-110362-8. Anglicky.
Klasická a přitom velice dobře napsaná učebnice jazyka C. Obsahuje výklad jazyka C podle standardu ANSI (a tedy také ISO). V posledních kapitolách je referenční přehled jazyka a přehled standardní knihovny jazyka C. Lze jen doporučit.
Ovšem pozor: Tato kniha existuje ve dvou vydáních. První vydání vyšlo v r. 1978 a popisovalo starší verzi jazyka, pro kterou se vžila označení "Jazyk C podle Kernighana a Ritchieho", "K&R" ap. Tato archaická verze jazyka C se sice občas ještě někde vidí, ale... od standardu se v několika bodech odlišuje, neobsahuje některé konstrukce běžně ve standardu používané ap. Takže starší vydání nedoporučuji používat, pokud opravdu nemusíte pracovat s překladačem, který tuto verzi implementuje. (Kniha sama je napsána dobře a pokud se starou verzí jazyka C pracujete, bude vám velice užitečná.)**
2. ***Kernighan, B. W. - Ritchie, D. M.: Programovací jazyk C.* SNTL - Alfa, Bratislava 1988. 248 stran. Slovensky.
Slovenský překlad prvního vydání (z r. 1978) slavné učebnice jazyka C. Platí o ní vše, co bylo řečeno o**[**prvním**](http://tjn.fjfi.cvut.cz/~virius/liter/litCpp.htm#KR)**anglickém vydání - tedy používat, pouze pokud pracujete s překladačem, který implementuje odpovídající verzi jazyka C.**
3. ***Stroustrup, B.: The Design and Evolution of C++.* AT&T 1994. ISBN 0-201-54330-3. Anglicky.
Kniha věnovaná historii jayzka C++, logice jeho návrhu ap. Najdeme to mimo jiné i zdůvodnění, proč tento jazyk obsahuje či neobsahuje určité rysy (např. proč chybí automatická správa paměti -- garbage collector), diskuzi o některých temných zákoutích jazyka ap. Velice zajímavé doplňkové čtení pro ty, které zajímají další souvislosti. Předpokládá ovšem solidní znalost C++.**
4. ***Herout, P.: Učebnice jazyka C, III upravené vydání.* Kopp, České Budějovice 1994. ISBN 80-85828-21-9. 270 stran. (*VI přepracované vydání.* Kopp, České Budějovice 2004.)
Dobře napsaná učebnice jazyka C podle ISO 9899-1990 v češtině. Vysvětlí všechny základní konstrukce a seznámí s běžnými knihovními funkcemi.
Čtvrté vydání obsahuje i kapitolu o novinkách, které přinesl standard ISO 9899-1999 a které se zvolna začínají prosazovat v nových verzích překladačů.**
5. ***Herout, P.: Učebnice jazyka C, 2. díl.* Kopp, České Budějovice 1995. ISBN 80-85828-50-2. 236 stran.
Vzdor názvu to není učebnice jazyka C. V první části najdete podrobný výklad o vstupních a výstupních operacích v C, popis některých užitečných funkcí a maker ze standardní knihovny, výklad o ladění v jazyce C. Druhá část obsahuje informace o Borland C++ a jeho využití pro programování v DOSu (programování přerušení, práce s porty, ovládání EMS a XMS a další speciální informace. Užitečné doplňkové čtení.**
6. ***Herout, P.: Borland C++, návod k použití.* Kopp, České Budějovice 1994. ISBN 80-85828-19-7. 185 stran.
Není to učebnice C++, je to kniha o prostředí Borland C++ 3.1 a o práci s ním.**
7. ***Mark Andews: Programujeme v jazyce Visual C++.* Computer Press, Praha 1997. ISBN 80-85896-91-5. 385 stran.
Na obálce je slovo "Visual" skryto uvnitř "C", takže snadno unikne pozornosti a láká čtenáře jako učebnice C++. Nicméně je to především kniha o vývojovém prostředí Microsoft Visual C++ 4.x; kromě toho v ní najdeme velice stručný rychlokurs základů C++ -- učebnici C++ ovšem nemůže nahradit -- a základy práce s knihovnou MFC. (Jde o překlad knihy *Learn Visual C++ Now!* vydané nakladatelstvím Microsoft Press r. 1996).**
8. ***Racek, S.: Objektově orientované programování v C++.* Kopp, České Budějovice 1994. ISBN 80-85828-20-0. 205 stran.
Ke čtení je třeba znát jazyk C. Autorovi jde především o základní principy OOP, jazyk C++ zůstává poněkud stranou (a je neustále srovnáván s jazykem Eiffel). Navíc výklad C++ v této knize vychází ze zkušeností s Borland C++ 3.1, takže pochopitelně neprobírá výjimky, prostory jmen, dynamickou identifikaci typů nebo standardní šablonovou knihovnu. Vedle základů teorie OOP tu lze najít zajímavý rozbor problematiky perzistence objektů.**
9. ***Racek, S. -- Kvoch, M.: Třídy a objekty v C++.* Kopp, České Budějovice 1998. ISBN 80-7232-017-3. 215 stran.
Podstatně přepracovaná a aktualizovaná verze knihy S. Racka *Objektově orientované programování v C++* (viz předchozí text). Ke čtení je opět třeba znát jazyk C. Autoři vysvětlí základní principy OOP a pak ukazují jejich aplikaci v C++ -- projdou objektové typy, přetěžování operátorů, šablony, výjimky, stručně hovoří i o objektových knihovnách. Z této knihy se lze základy C++ naučit.**
10. ***Nenadál, K. -- Václavíková, D.: Borland C++.* Grada Publishing, Praha 1992. 225 stran.
Nepříliš kvalitně přepsané manuály k Borland C++ 2.0. To znamená, že se zde mj. ještě nehovoří ani o šablonách, natož pak o výjimkách nebo prostorech jmen. Dnes prakticky nemá smysl používat.**
11. ***Šaloun, P.: Programovací jazyk C pro zelenáče.* Neokortex, Praha, 2000. ISBN 80-86330-02-X. 205 stran.
Kniha sice pokrývá většinu témat, která by měl programátor v jazyce C znát, ale nepatří mezi podařené. Výklad je poměrně málo přehledný (i díky jazyku), autor často bez vysvětlení předbíhá a používá neprobrané konstrukce. Kniha neobsahuje sice příliš mnoho chyb, ale najdou se, a to i poměrně vážné. Asi největší výhrada, kterou proti ní mám, se týká výkladu o znakových řetězcích a polích vůbec: Autor zaměňuje pojmy "řetězec" a "ukazatel na řetězec", a to občas i v jedné větě -- a to je v textu pro začátečníky vyloženě nešťastné, neboť z nepochopení vztahu polí a ukazatelů pocházejí ty nejhrubší chyby při práci s pamětí.
Celkově působí dojmem knihy spáchané v horečném chvatu. Nemohu ji v žádném případě doporučit.
Poznámka ke druhému vydání (léto 2003, ISBN 80-86330-08-7): Pro druhé vydání byla kniha přepracována, autor odstranil nejhrubší chyby. Celkový dojem neobratnosti výkladu ovšem trvá.**
12. ***Kruglinski, D. -- Shepherd, G. -- Wingo, S.: Programujeme v Microsoft Visual C++.* Computer Press, Praha, 2000. ISBN 80-7226-362-5.
Přepracované vydání úspěšné knihy D. Kruglinského *Mistrovství ve Visual C++* (*Mastering Visual C++*, Microsoft Press), upravené pro Visual C++ 6.0.
Ke čtení potřebujete znát C++. Naučíte se zde používat knihovnu MFC (základy, architekturu dokument-pohled), používat prvky ActiveX, pracovat s automatizací, vytvářet komponenty COM a (včetně použití knihovny ATL), pracovat s databázemi pomocí ODBC, DAO a OLE DB a programovat pro internet (speciálně pro Microsoft IIS). I přes značný rozsah nejde v žádném z témat do přílišné hloubky.
První dojem: užitečná kniha, i když poměrně drahá (doporučená cena 975 Kč, 1404 Sk); tomu ale odpovídá i rozsah (1000 stran).**
13. ***S. Prata: Mistrovství v C++.* Computer Press, Praha 2001. ISBN 80-7226-339-0
Vysvětluje podrobně jednotlivé konstrukce, ukazuje jejich význam a použití. Postupuje od základů k pokročlým konstrukcím, používá od počátku i standardní knihovnu. Na první pohled také vypadá dobře, i když je na můj vkus trochu rozvláčná (cca 1000 stran).**
14. ***Stanley B. Lippman: Essential C++.* Addison-Wesley 1999. ISBN 0-201-48518-4. 270 stran. Anglicky.
Dobrá učebnice základů jazyka C++, která hned od počátku -- od programu Hello, world -- používá standardní knihovnu. Zajímavé je uspočádání výkladu podle přístupů k programování: Po probrání základů se autor věnuje procedurálnímu programování, pak využití generických algoritmů a kontejnerů ze standardní knihovny (což označuje za generické programování), následuje programování s objekty, objektové programování, programování se šablonami a práce s výjimkami.**
15. ***Andrei Alexandrescu: Modern C++ Design.*Addison-Wesley 2001. ISBN 0-201-70431-5. 321 stran. Anglicky.
Šablony byly původně navrženy proto, aby usnadnily vytváření opakujícího se podobného kódu, který se liší jen datovými typy nebo hodnotou konstant známých už v době překladu. Ovšem šablony v dnešní podobě otevřely obrovské a netušené možnosti, které se dnes shrnují pod označení "generické metaprogramování". Lze je s trochou nadsázky označit za programování, při kterém vše podstatné proběhne v době překladu a za běhu už jen vystoupí výsledky.
Autor v této knize ukazje, jak takové konstrukce vytvářet. V úvodu např. najdeme analogii makra assert(), ovšem vyhodnocovaného v době překladu. Jako další ukázky najdeme seznamy typů, zobezněné funktory atd. V mnoha případech jde o "generické" implementace klasických návrhových vzorů.
Tato kniha nabízí vyšší školu programování pomocí šablon. Je to zajímavé čtení, i když dosti náročné.
V současné době je v tisku český překlad pod názvem "Moderní programování v C++".**
16. ***F. Kašpárek a kol.:***[***Co v manuálu nenajdete.***](http://tjn.fjfi.cvut.cz/~virius/publ-list.html#Nenajdete)**Unis Publishing, Brno 1993. ISBN neuvedeno. 762 stran.
Kniha o nedokumentovaných vlastnostech překladače Borland C++ 3.1 a starších a některých (v době vydání) méně známých vlastnostech C++. Popis jazyka C++ odpovídá variantě použité v tomto překladači, tzn. dost významně se odchyluje od standardu ISO 14882.
Má smysl používat pouze v případě, pokud pracujete s uvedeným překladačem.**
17. ***D. Matoušek: C++Builder -- vývojové prostředí.*Ben -- Technická literatura, Praha 2000. ISBN 80-86056-70-8. 630 stran.**
18. ***D. Matoušek: C++Builder -- vývojové prostředí. 2. díl.*Ben -- Technická literatura, Praha 2001. ISBN 80-7300-003-2. 520 stran.
Velmi podrobná kniha o C++Builderu verzí 4 a 5 s řadou příkladů. První díl probírá vývojové prostředí, běžně používané komponenty, práci s INI soubory, tvorbu nápovědy, tvorbu vlastních komponent, komponenty pro práci s DDE a OLE. Autor také vysvětluje použití výjimek.
Ve druhém dílu najdeme informace o pokročilejších rysech vývojového prostředí. Najdeme tu informace o rpáci s metasoubory, rolování, způsobu zpracování zopráv, práci s prostředky (resources) včetně textu atd. V této knize najdeme i informace o odpovídajících funkcích z Windows API. Poslední kapitoly seznamují se základy knihovny OpenGL, se sériovým portem ap.
Velmi užitečná kniha, i když nepokrývá některé témata (ale nikde není řečeno, že druhý díl je poslední).**
19. ***J. Bergin, M. Stehlik, J. Roberts, R. Pattis: Karel++.*J. Wiley and Sons, 1997. ISBN 0-471-13809-6. Anglicky.
Robot Karel, ovšem v objektovém provedení založeném na jazyce C++. (Robot Karel je dnes už klasická pomůcka pro výuku základů programování, zavedená R. Pattisem v knize Karel the Robot, 1981.)**
20. ***A. Koenig, Barbara E. Moo: Accelerated C++.*Addison-Wesley, 2000. ISBN 0-201-70353-X. Anglicky.
Kvalitní úvod do jazyka C++.**
	1. ***S. Meyers: Effective C++.*Addison-Wesley, 1993, 2001. Anglicky.**
	2. ***S. Meyers: More Effective C++.*Addison-Wesley, 1996. ISBN 0-201-63371-X. Anglicky.**
	3. ***S. Meyers: Effective STL.*Addison-Wesley 2001. ISBN 0-201-74962-9. Anglicky.**

**Všechny tři knihy obsahují řadu kráktých kapitol, které rozebírají některé důležité a často špatně chápané stránky jazyka C++. První z nich vznikla na počátku 90. let a v 50 kapitolkách se zabývá tématy jako jsou ukazatele, používání konstruktorů a destruktorů objektových typů, přetěžování operátorů atd. Druhá vznikla v polovině 90. let a v 35 kapitolkách se zabývá chybami objektového návrhu, problémy okolo používání výjimek, efektivitou atd. Poslední kniha vznikla už po přijetí mezinárodního standardu IS0 14882 a v 50 kapitolkách rozebírá problémy a nedorozumění kolem správného používání kontejnerových tříd a iterátorů ve standardní šablonové knihovně.
Velice užitečné knihy. I když první dvě vznikly ještě před přijetém standardu, neztratily nic na platnosti. Jediná nevýhoda: Čím novější kniha, tím upovídanější. Přesto je doporučuji.**

1. ***V. Kadlec: Učíme se programovat v jazyce C.*Computer Press, Praha 2002. ISBN 80-7226-715-9.
Docela dobře napsaný úvod do jazyka C. Jediné mé výhrady se týkají skutečnosti, že autor hovoří o normě ANSI jazyka C, která de facto neexistuje (ANSI distribuuje mezinárodní standard ISO 9899). Autor také vychází ze standardu z r. 1990 a možná i starších, takže např. nehovoří o typu long long a dalších věcech, které tato norma zavedla.**
2. ***R. Neruda, T. Holan: C++ Builder v příkladech.* Ben -- Technická literatura, Praha 2002. ISBN 80-7300-042-3.
Řada řešených příkladů v C++Builderu. Kniha pokrývá tvorbu GUI, počítačovou grafiku s pomocí C++Builderu a databázové aplikace. Každý příklad obsahuje vždy zadání, řešení a poznámky a komentáře. Předpokládá znalost C++ a ve výkladu o C++Builderu (tedy o knihovně VCL) nejde do hloubky, jen naznačuje, ale to obvykle snaprosto stačí. Skvělý doplněk jakéhokoli úvodního nebo i lehce pokročilého kurzu C++Builderu, velmi dobré i pro samostatné čtení.**
3. ***D. Vandervoorde, N. M. Josuttis: C++ Templates. The Complete Guide.*Addison-Wesley 2003. ISBN 0-201-73484-2. 530 stran, cena 55 $.
Kniha předpokládá znalost jazyka C++ (kromě šablon, neboť ty vysvětluje od základů). Zachází ovšem do hloubky, s níž se v běžných učebnicích nesetkáme -- pokrývá nejen základní použití šablon v C++, ale i základy šablonového metaprogramování a někteté další pokročilejší aplikace. Vynikající doplňkové čtení, i když má smysl spíše pro tvůrce knihoven než pro běžné programátory.**
4. ***R. Chalupa: 1001 tipů a triků pro Visual C++.*Computer Press 2003. ISBN 80-7226-842-2. 434 stran, cena 397 Kč, 595 Sk.
kniha obsahuje 1003 (nikoli 1001) krátkých návodů a ukázek, jak řešit různé problémy v jazyce C++ v prostředí Windows. Vychází z Visual C++ .NET, hovoří ovšem převážně o knihovně MFC a o Windows API. Najdme tu vedle sebe poučení o vlastnostech celočíselných typů, návod, jak napsat spořič obrazovky nebo třeba jak pracovat s bitovou mapou a mnoho dalšího. Užitečná publikace, i když mj. neobsahuje rejstřík (a to mi připadá jako docela závažná chyba).**

## Literatura o standardní šablonové knihovně (STL)

**Na českém trhu (pokud vím) v současné době chybí kniha, která by se systematicky zabývala standardní šablonovou knihovnou jazyka C++. Jediná publikace, která stojí za zmínku, je kniha**[**Jazyky C a C++ podle normy ANSI/ISO**](http://tjn.fjfi.cvut.cz/~virius/publ-list.html#ANSI)**, která obsahuje referenční přehled této knihovny. To ovšem nestačí pro pochopení způsobu práce s touto knihovnou. Proto zde uvádím pouze zahraniční publikace.**

1. ***P. J. Plauger, A. A. Stepanov, Meng Lee, D. R. Musser: The C++ Standard Template Library.*Prentince Hall 2001. ISBN 0-13-437633-1. Anglicky.
Popis hlavičkových souborů obsahujících standardní knihovnu jazyka C++, dále popis tříd a dalších konstrukcí v nich, základní informace o jejich účelu a použití. Může sloužit především jako reference, jako učebnice je to omezeně použitelné.**
2. ***D. R. Musser, G.J. Derge, A. Saini: STL Tutorial and Reference guide, Second Edition.*Addison-Wesley, 2001. ISBN 0-201-37923-6. Anglicky.
Ukazuje použití konstrukcí ze STL (nejen tříd, ale i algoritmů). Podstatně lépe čitelné než předchozí publikace. Ve skutečnosti asi jedna z nejlepších publikací o STL. Anglicky.**
3. ***P. J. Plauger: The Draft Standard C++ Library.*Prentince Hall 1995. ISBN 0-13-117003-1. Anglicky.
Popis hlavních hlavičkových souborů obsahujících standardní knihovnu jazyka C++ a tříd v nich, ovšem podle návrhu standardu z doby před rokem 1995. To znamená, že dnes jde o zastaralou záležitost. Například datové proudy zde nejsou ještě implementovány prostřednictvím šablon, liší se používání tříd rysů (traits), nepopisuje alokátory, lokální nastavení pouze zmiňuje atd. Liší se i jména hlavičkových souborů. Dnes již jen omezeně použitelné, pokud nepracujete s překladačem z oné doby. (Pozor, slovo "draft" je na obálce velice nenápadné).**
4. ***S. Meyers: Effective STL.*Addison-Wesley 2001. ISBN 0-201-74962-9. Anglicky.
Předpokládá povrchní znalost základních tříd knihovny STL. Vysvětluje použití kontejnerů (prostřednictvím iterátorů) a v 50 kapitolkách ukazuje problémy, na které můžeme narazit. Velice užitečná kniha.**
5. ***Nicolai S. Josuttis: The C++ Standard Library.*Addison-Wesley 1999. ISBN 0-201-37926-0. Anglicky. 799 stran, cena 57,99$.
Podrobná a dobře napsaná kniha. Na rozdíl od většiny autorů se nevěnuje jen kontejnerům, iterátorům a případně algoritmům a pomocným třídám, ale např. i třídě locale a problémům neanglických jazyků. (Autor žije v Německu. Zřejmě je nutné, aby šlo o knihu od autora, jehož rodnou řečí není angličtina, aby tuto problematiku považoval za hodnou zmínky. Poznamenejme ale, že v dodatku ke Stroustrupově knize je výklad o lokálním nastavení možná přehlednější.)**
6. ***Leen Ammeraal: STL for C++ Programmers.*John Wiley & Sons 1997. ISBN 0-471-97181-2. Anglicky.
Stručný, ale efektivně napsaný úvod do STL. Recenze citované na obálce říkají, že "hned po přečtení lze v STL programovat". Vychází ovšem ze starší verze návrhu standardu, a proto se mohou některé drobnosti odlišovat od konečné podoby standardu.**

**Seznam schválen Ing. Miroslavem Viriem, CSc. (Programování v C++)**

# [C++ - 1. lekce](http://programujte.com/clanek/2005041402-c-1-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       197 899×**

* **První program**
* **Zdrojový program**
* **Překlad, sestavení, spuštění**
* **Komentáře**
* **Funkce main()**
* **Vstup a výstup**
* **Zápisy programu**
* **Chyby a varování**
* **Direktiva #include**
* **Překlad programu**
* **Úkol č. 1**

**Ukážeme si, jak vypadá program v C++, setkáme se s objektovými typy a jejich používáním.**

## První program

**Nejdříve bych chtěl upozornit, že budete-li mít JAKÝKOLIV problém s čímkoliv, navštivte Diskuzní fórum, kde vám poradím. Navštěvuji ji často, a proto se odpovědi dočkáte ve velmi krátkém čase. Pokud se chcete opravdu učit C++, zaregistrujte se.**

**Jako všechny učebnice, i my začneme klasickým „Ahoj světe“.**

**#include <iostream> // 1**

**using namespace std; // 2**

**int main() // 3**

**{ // 4**

 **cout << "Ahoj svete!" << endl; // 5**

 **return 0; // 6**

**}**

**Výsledek:**

**Ahoj svete!**

## Zdrojový program

**Program v C++ se skládá z jednoho nebo více textových souborů. Pozor na velká a malá písmena, jazyk C++ je rozlišuje!**

## Překlad, sestavení a spuštění

**Zdrojový program musíte přeložit z C++ do strojového kódu. Jaké programy použít, jste se dověděli v úvodní lekci.**

**Překladač nám vytvoří relativní soubor (\*.obj nebo \*.o), ale ten nelze spustit; musíme k němu připojit knihovny a vytvořit z něj spustitelný soubor (\*.exe).**

## Co jsme naprogramovali

**// 1, // 2… Toto vidíte na každém konci řádku. Není to nic jiného než komentář. Absolutně nepovinný, použili jsme ho k očíslovaní řádků a slouží spíše k orientaci v programu, když se k němu např. po několika měsících vrátíme. Komentář začíná // a vše, co je na TOMTO řádku za těmito lomítky, se tedy vůbec nepřekládá, slouží jen jako poznámky. Můžeme je zapisovat ještě jedním způsobem. Začneme znaky /\* a skončíme znaky \*/ (mezi *\** a */* nesmí být mezera). Takto se zapisují komentáře delší (je jedno na kolik řádků, protože je komentář i zakončen):**

|  |  |
| --- | --- |
| **// muj komentar** | **správně** |
| **// muj komentarktery pokracuje** | **špatně - musí být na stejném řádku** |
| **/\* muj komentar** | **špatně - chybí zakončovací znak \*/** |
| **/\* muj komentar \*/** | **správně** |
| **/\* mujkomentar \*/** | **správně** |

## Funkce main()

**Programy se skládají z hlavního programu a z řad podprogramů. Roli hlavního programu zde právě hraje funkce *main* (malými písmeny). V každém programu musí jedna f-ce*main* být. Prázdné závorky říkají, že se jedná o funkci s parametry. Slovo *int* říká, že f-ce (funkce) vypočte a vrátí hodnotu typu *int*, tedy celé číslo.**

**Tělo funkce začínáme složenými závorkami { a končíme } (mají podobnou úlohu jako v Pascalu *begin* a *end*).**

**Každý příkaz musí končit středníky. *main()* je f-ce dělená na dvě části (*main()* a *return 0;* ) a tedy až druhá část se zakončuje středníky. Příkaz *return* znamená vždy ukončení f-ce, která ji zavolala (v našem případě ukončit *main()* a tím celý program, návrat do OS).**

**Ta nula za *return* znamená, že program skončil bez problémů. Nenulové číslo pak určuje kód chyby.**

## Vstup a výstup

**Náš program vypisuje znakový řetězec „Ahoj svete!“. O to se stará příkaz na řádku 5.**

**cout << "Ahoj svete!" << endl;**

**Teď vám vysvětlím, co znamená using namespace std; . Kdybyste toto nenapsali, museli byste pokaždé psát std::cout << "Ahoj svete!" << std::endl;.**

**using namespace std; nám zpřístupní definice. Nejlépe se s tím smiřte a trápit se s tím budeme později.**

**Vysvětlení (neodborné): Představte si několik balíků krabic. Chcete červenou krabici, ale ty jsou tam 3 (pojmenovány std, ahj, bnm). A protože bychom chtěli banán z krabice ahj, řekli bychom dotyčnému „Chci banán z krabice ahj“. To samé je v C++. Krabice *std* obsahuje prostory jmen, které se za ní skrývají. std::cout je tedy *cout* umístěný v krabici *std*. Proto na začátku programu definujeme, že jakýkoliv název funkce obsažený v balíku *std* (a možná, že i v ostatních balících) se použije PRÁVĚ *cout* z krabice *std*. Kdybychom (teoreticky) neuvedli using namespace std;, překladač by nevěděl, ze které krabice má *cout* vzít (pokud by byl ve všech), a v každé by měl jiný význam! Takže si vyberte, jestli budete psát  std::cout;  nebo napíšete na začátek  using namespace std;  a pak už jen názvy příkazů.**

**endl není nic jiného, než ukončení řádku (Enter), ale můžete to udělat i jinak, lépe:**

**cout << "Ahoj svete!\n";**

**\n – příkaz „jdi na další řádek“, nemusí mít mezeru mezi slovy (Z toho vyplývá, že pokud ho do textu kdykoliv napíšete, nevypíše se \n, ale provede se přeskok na další řádek.). Mezi *\* a *n* NESMÍ být mezera.**

**ZÁKLADEM UČENÍ JE ZKOUŠET! PROTO ZKOUŠEJTE!**

**cout << "Ahoj svete!" << endl; // správně**

**cout << "Ahoj svete!<< endl" // špatně,**

 **// endl nesmí být umístěn v textu**

**cout << "Ahoj svete!\n" // správně**

**cout << "Ahoj \nsvete!" // správně**

 **// (svete bude již na druhém řádku)**

**Část uzavřená do uvozovek je zpráva pro tisk (např monitor). Označení << signalizuje, že se posílá řetězec do *cout*, je to vkládací operátor.**

**Možná, že se vám výsledný exe soubor při spuštění hned zavře, aniž byste měli možnost stihnout přečíst text. V takovém případě musíte na konec souboru připsat  cin.get();.**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **cout << "Ahoj svete!" << endl;**

 **cin.get(); /\* někdy je třeba použít 2×. cin.get(),**

 **cin.get(); vyčká na zmáčknutí klávesy ENTER \*/**

 **return 0;**

**}**

**Ve Visual C++ dáme pro spuštění Ctrl + F5. V Borland C++ 3.1 stačí stisknout klávesu F9 (pro přeložení). Použijeme-li překladač MinGW32, který se spouští z příkazové řádky, zadáme příkaz *g++ PRVNI* (název souboru je PRVNI).**

## Hlavičkový soubor

**Podobně jako v jiných programovacích jazycích, i v C++ platí, že můžeme v programu pracovat s tím, co překladač už zná. Standardní hlavičkové soubory v C++ jsou bez přípony, v jazyce C mají příponu .h a někdy také .hpp.**

**Informace o základních vstupních a výstupních proudech jsou v hlavičkovém souboru *iostream* (ve starších překladačích iostream.h).**

**Aby je měl překladač k dispozici, musíme jej vložit do našeho programu, nejlépe hned na začátek.**

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

**using namespace std; // zpřístupní definice**

**int main() // hlavička f-ce**

**{ // začátek těla f-ce**

 **cout << "Ahoj svete!" << endl; // zpráva**

 **return 0; // ukončuje main()**

**} // konec těla f-ce**

## Zápisy programu

**Klíčová slova (return, int…) nemůžeme rozdělit mezerou nebo přechodem na nový řádek.**

## Chyby a varování

**Pokud něco napíšeme špatně, překladač oznámí chybu. Vedle chyb C++ hlásí také varování. Chyby musíme bezprostředně odstranit, neboť s nimi nelze program spustit. Varování vždy upozorňují na situaci, která je z hlediska syntaxe správná, ale může mít pro programátora jiný význam, než očekává. Proto je důležité sledovat, co hlásí (vlastní zkušenost je, že 70 % varování chybu znamená).**

## Direktiva #include

**Pokud překladač narazí na výskyt *#include*, nahradí ji obsahem určeného souboru.**

**#include <soubror> – pokud použijeme tento zápis a nespecifikujeme úplnou cestu, bude soubor hledán ve standardním adresáři pro ukládání hlavičkových souborů.**

**#include "soubor" – soubor bude hledán nejdříve v adresáři se zdrojovým souborem a pak teprve v adresáři s hlavičkovými soubory.**

## Ještě o překladu programu

**Překlad probíhá standardně ve 3 fázích. Zdrojové soubory se zpracují preprocesorem, který provede kontextovou úpravu (odstraní komentáře, vloží text souborů uvedených v direktivách #include apod.), pouze zařídí, že takto bude vidět soubory překladač, nic takového se ve skutečnosti nestane.**

**Ve druhé fázi se vytvoří relativní soubory a ve třetí je spojí linker s knihovnou a vytvoří spustitelný soubor.**

## Úkol č. 1

**Vytvořte program, který vypíše na 5 řádcích informace o vás (jakékoliv).**

# [C++ - 1. lekce - dodatek](http://programujte.com/clanek/2005041403-c-1-lekce-dodatek/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       51 624×**

* **Některá pravidla při programování**
* **Klíčová slova v C++**

## Některá pravidla při programování

### Jména proměnných

**C++ vám dovoluje použít všechna smysluplná jména pro proměnné, ale měli bychom pamatovat, že ne všechna jména můžeme použít.**

### Pravidla při tvorbě jména proměnné

* **Jediné znaky, které můžeme použít, jsou písmena, číslice a znak podtržení \_.**
* **První znak ve jménu nemůže být číslice.**
* **Velká písmena se považují za jiná, než malá písmena.**
* **Pro jména nemůžete použít klíčová slova C++.**
* **Nezačínejte jméno znakem podtržení \_.**
* **Nejste omezeni délkou jména proměnné.**
* **Všechny znaky ve jménu jsou významné.**

**ANSI C (Céčko) právě považovalo maximální délku proměnné (užitečnou délku) za 31 znaků. Tedy 2 jména, která se lišila třeba v 32. znaku se považovala za stejná (mohli jste mít libovolně dlouhá jména, ale jen prvních 31 bylo důležitých). V C++ jsou počítány všechny znaky.**

|  |  |
| --- | --- |
| **int prom;** | **správně** |
| **int Prom;** | **správně, ale různé od prom** |
| **int PROM;** | **správně, ale také různé od 2 předchozích** |
| **Int prom;** | **špatně – nemá být Int, ale int** |
| **int \_prom;** | **správně, ale nepožívejte to** |
| **int 4prom;** | **špatně – začíná číslicí** |
| **int double;** | **špatně – double je klíčovým slovem C++** |
| **int prom\_je\_nazev\_promenne** | **správně, ale zbytečně moc dlouhé** |
| **int prom-m;** | **špatně – pomlčka není povolena** |

## Klíčová slova v  C++

**Klíčová slova jsou identifikátory, které tvoří slovní zásobu programovacího jazyka. Pro jiné účely, jako například název proměnné, je použít nelze. Klíčová slova jazyka jsou zobrazena v následujícím seznamu; ne všechna jsou v současné době implementována.**

 ****

# [C++ - 2. lekce](http://programujte.com/clanek/2005041404-c-2-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       113 151×**

* **typy celočíselné, reálné**
* **proměnné - definice, inicializace a volání**
* **operátory, inkrementace, dekrementace**
* **přetypování**
* **zkracování zápisů**
* **postraní efekty**
* **vstupy**
* **úkol č. 2**

**Dnes si probereme, jak definovat proměnné, jaké můžou být typy a jednoduchou práci s nimi, dále také vstup (komunikace s uživatelem).**

## Celočíselné, reálné a jiné typy

### Celočíselné typy

**Hlavním celočíselným typem je *int*. Je dobré ho preferovat, protože je optimalizovaný pro co nejrychlejší práci. Dalším typem je *short int* - nebo-li *short*. Int má velikost někdy 2 někdy 4 byty, v závislosti na překladači. *Short* by měl vyhovovat požadavkům na 2 byty. V přehledné tabulce uvidíte, jaké jsou limity. Když určujeme proměnnou, musíme určit její typ. (Zda je celočíselný, jakou může mít maximální velikost, zda obsahuje desetinnou část...)**

**Nyní se ale zabývejme pouze typem proměnné. Pro celočíselné typy je to :**

| **typ:** | **dolní mez:** | **horní mez:** | **velikost (v bytech):** |
| --- | --- | --- | --- |
|  **char** | **0** | **255** | **1** |
| **signed char** | **- 128** | **127** | **1** |
| **unsigned short int** | **0** | **65 535** | **2** |
| **signed short int** | **- 32 768** | **32 767** | **2** |
| **unsigned int** | **0** | **65 535** | **2** |
| **signed int** | **- 32 768** | **32 767** | **2** |
| **unsigned long int** | **- 231** | **231 - 1** | **4** |
| **signed long int** | **0** | **232** | **4** |

**typ - typ proměnné
dolní mez - nejmenší číslo, které unese daný typ
horní mez - nejvyšší číslo, které unese daný typ
velikost - velikost výsledné proměnné v paměti [b]
<="" p="" style="margin: 0px; padding: 0px; border: 0px; list-style-type: none; text-size-adjust: none;">**

***signed* - znaménkovost typu, signed = vynucení znaménka (+ -)
*unsigned* - neznaménkovost typu, unsigned = výsledný typ bude nezaménkový
*char* - nejmenší typ, uchovává právě pouze 1 znak**

### Reálné typy

**Existují tři - jak ukazuje tabulka:**

| **typ:** | **maximum:** | **přesnost (počet des. míst):** | **velikost (v bytech):** |
| --- | --- | --- | --- |
| **Float** | **3.40282 . 1038** | **6** | **4** |
| **Double** | **1.79769313486232 . 10308** | **15** | **8** |
| **long double** | **1.18973149535723176 . 104932** | **19** | **10** |

**maximum - max. velikost čísla
přesnost - maximální počet čísel za desetinnou čárkou**

**Stejně jako u celočíselných typů preferujeme *int*, zde preferujeme *double*.**

**Poznámka: pokud neuvedeme *signed* či *unsigned*, budou typy *int*, *short* a *long* považovány za *signed* - budou mít znaménko.**

**U *char* je to jiné, zde záleží na překladači.**

**Nebojte se, hned pochopíte k čemu nám toto je.**

## Proměnné

### Definice

**Když už známe typy, pojďme si vytvořit proměnné. Je to velmi jednoduché. Nejprve si řekněme, k čemu by nám měla proměnná sloužit. Dejme tomu, že chceme získat z uživatele jeho věk; to je celočíselný typ - nepotřebuje desetinnou část. Máme na výběr z typů *int*, *short* a *long*. I když můžeme požít *short*, použijeme raději preferovaný *int.* Jak by tedy vypadala definice proměnné, kterou si nazveme vek?**

**int vek; // toť vše ;)**

**Nyní jsme vytvořili proměnnou s názvem vek a typem *int*. Pokud se na ni budete odkazovat, volat ji apod., stačí napsat jen název proměnné, překladač si již bude pamatovat, že proměnná vek je typu *int*. Ovšem proměnná je vytvořena, ale není inicializována (= nemá žádnou hodnotu).**

**V našem případě ji mít nemusí, protože ji tam zadá uživatel až při vykonávání programu. Ale o tom až dále, nyní se pojďme podívat na samotné inicializování. Uvažujme následující kód:**

### Inicializace

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **cout << "Ahoj.\n";**

 **int vek; // tady ji vytváříme**

 **vek = 18; // zde inicializujeme**

 **cin.get();**

 **return 0;**

**}**

**Ale jde to udělat i jednodušeji:**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **cout << "Ahoj.\n";**

 **int vek = 18; // tady ji vytváříme a rovnou inicializujeme**

 **cin.get();**

 **return 0;**

**}**

**Máme proměnnou. Ovšem když si program spustíme, napíše pouze**

**Ahoj.**

**Nikde se nám neukáže naše proměnná, proč? Protože jsme ji nezavolali.**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **cout << "Ahoj.\n";**

 **int vek = 18;**

 **cout << vek; // cout dosadí hodnotu proměnné *vek* (18)**

 **cin.get();**

 **return 0;**

**}**

**Výstup:**

**Ahoj.18**

**Když vyvoláváme proměnnou, nedáváme ji do uvozovek a nikdy nesmíme vyvolat proměnnou, která nebyla před voláním inicializována (nenabyla hodnotu před zavoláním.). Tedy:**

 **cout << "Ahoj.\n";**

 **int vek;**

 **cout << vek; // NENÍ hodnota! chyba (buď ji napíše kompilátor nebo ve výsledku se objeví nesmyslné číslo)**

 **cin.get();**

 **return 0;**

**Zkoušejte si vytvářet proměnné různých typů a naučte se s nimi pracovat!**

**Inicializovat se dá samozřejmě i způsobem, jaký známe z matematiky.**

 **cout << "Ahoj.\n";**

 **int vek = 18;**

 **cout << vek + 2;**

 **cin.get();**

 **return 0;**

**Výstup:**

**Ahoj.20**

## Operátory

**Kromě klasických operátorů, které jistě znáte (doufám ;)) z matematiky, se nabízejí i jiná řešení.**

| **operátor:** | **popis:** |
| --- | --- |
| **+, -, \*** | **klasické operátory** |
| **/** | **celočíselné, nebo reálné, dělení (To, jestli bude provedeno dělení celočíselné nebo reálné, záleží na typu operandů. Pokud oba operandy budou celočíselného typu, provede se celočíselné dělení. Pokud ovšem alespoň jeden z operandů bude reálného typu, provede se dělení reálné. V případě, že potřebujeme provést reálné dělení na operandech celočíselných typů, musíme jeden z nich přetypovat pomocí operátoru přetypování (viz. dále) na reálný typ.)** |
| **%** | **zbytek po celočíselném dělení.** |
| **=** | **Přiřazení** |
| **<, <=, >, >=** | **menší než, menší nebo rovno, atd...** |
| **!=** | **Nerovnost** |
| **==** | **Rovnost** |

**Myslím, že k tomu není co dodat.**

| **operátor:** | **popis:** |
| --- | --- |
| **++** | **inkrementace (zvýší hodnotu operandu o 1)** |
| **--** | **dekrementace (sníží hodnotu operandu o 1)** |

**Zde se jedná jen o zvětšení či zmenšení o 1. Používá se v podmíněných cyklech (tedy za nějaké podmínky udělej to, jinak to - o tom až později ;))**

**Příklad:**

 **int a, b=10, c; // inic. proměnné stejného typu (jako v Pascalu)**

 **++b; // nyní *b* = 11**

 **c = b; // přiřadili jsme hodnotu *b* do proměnné *c*b--; // *b* má nyní hodnotu 10**

 **a = b; // *a* má stejnou hodnotu jako *b* (10)**

**Jaký je rozdíl v použití *--x* a *x--*?**

**Oba operátory se dají použít jak před svým operandem, tak za ním. Význam je však pokaždé jiný. Pokud operátor uvedeme před operandem, jedná se o tzv. *preinkrementaci*. To znamená, že hodnota operandu se nejprve zvýší a tato zvýšená hodnota je vrácena jako výsledek operace. Naopak při *postinkrementaci* je jako hodnota celého výrazu vrácena původní hodnota operandu a pak je teprve operand samotný zvýšen o 1.**

 **a = b++;**

**// Postinkrementace. Výraz *b++* je nejdříve vyhodnocen (a jeho hodnota přiřazena proměnné *a*) a pak je teprve inkrementována proměnná *b*. Proměnná *a* teď tedy má hodnotu 1 a *b* je rovno 2.**

 **a = ++b;**

**// Preinkrementace. Nejdříve je inkrementováno *b* na hodnotu 3. Tato hodnota je i výsledkem vyhodnocení výrazu *++b* a tedy je přiřazena proměnné *a*. Hodnota *a* je 3.**

##

## Přetypování

**int a=15;**

**Máme proměnnou *a*, a její hodnota je 15. Nyní chceme, aby byla jako typ v desetinné čárce. Zápis vypadá takto:**

**jméno\_typu(název\_proměnné);**

**Příklad:**

**double(a); // konvertuje proměnnou a na typ double**

## Zkracování

**Někdy se můžete setkat s následujícím zápisem:**

**a\*=a;**

**Nebojte, není to nic jiného, než:**

**a = a \* a;**

**A těchto operátorů je celá řada: *+=*; *-=*; *\*=*; */=*; *%=***

**Tedy:**

**int a=10; // a=10**

**a+=a; // a=20**

**a = a + a; // a=20**

## Postranní efekty

**V souvislosti s operátory v C++ je třeba zmínit se také o jejich postranních efektech. Výsledkem použití operátoru na operandy získáme nějakou konkrétní hodnotu. Například použitím '+' na čísla 2 a 3 získáme hodnotu 5. Operátory ale mohou mít i postranní efekt, který je však u některých operátorů jejich hlavním účelem. Například vyhodnocením výrazu a=2 získáme hodnotu 2, protože výsledkem operace přiřazení je sama hodnota přiřazovaného výrazu. Postranním efektem je pak uložení hodnoty 2 do proměnné *a*. Problém nastává, když v jednom výrazu aplikujeme na jednu proměnnou více operací s postranním efektem. To proto, že jazyk C++ nezaručuje pořadí vyhodnocování operandů u většiny binárních operátorů.**

 **int i=2;**

**i + ++i;**

**Příklad demonstruje nebezpečnost podobných zápisů. Před vyhodnocením celého výrazu má proměnná i hodnotu 2. Bohužel ale nevíme, který z operandů operátoru *+* bude vyhodnocen dříve. Bude-li to levý operand, bude mít celý výraz hodnotu 5. Pokud však bude jako první vyhodnocen pravý operand, bude hodnotou výrazu číslo 6. Proto je dobré takovéto výrazy nepoužívat a radši je rozepsat do více kroků tak, aby na jednu proměnnou byl aplikován vždy maximálně jeden operátor s postranním efektem.**

## Vstupy

**Nyní se dostáváme konečně pod kůži tomu, aby uživatel při běhu programu zadával hodnoty.**

**K tomu je zapotřebí f-ce *cin*, která přikáže počkat, dokud uživatel něco nenapíše a jakmile zmáčkne ENTER, bude text uložen tam, kam mu přikážeme.**

**Není nad příklad! ;)**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **cout << "Zadej prosim tvuj vek: ";**

 **int vek; // vytváříme proměnnou, ale nemusíme ji inicializovat (je to zbytečné)**

 **cin >> vek; // nechá uživatele napsat věk a po zmáčknutí ENTERu uloží hodnotu do proměnné *vek***

 **cout << "\nVas vek je: " << vek;**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

**Všimněte si, že zatímco << přepošlou text zprava do f-ce *cout* (a tím se zobrazí na obrazovce), *>>* vezme hodnotu (nyní zapsanou někde v dočasné proměnné u f-ce *cin*) z *cin* a uloží ji do proměnné *vek*. #############**

**Poté jsme chtěli vypsat větu, kolik nám je (výsledek, co uživatel zapsal).**

**Berte to tak, jako byste chtěli napsat:**

**cout << "\nVas vek je :";cout << vek;**

**To je to samé! Jen to zkrátíme, abychom nemuseli psát znova *cout*. Tzn, že napíšeme vše jak normálně, ale na stejný řádek a bez *cout* a *;*:**

**cout << "\nVas vek je: " << vek; // nám říká, vezmi text *Vas vek je:*  a pošli ho na výstup, poté vezmi proměnnou *vek* a pošli ji na výstup.**

**A mohli bychom pokračovat:**

**cout << "\nVas vek je: " << vek << "\nDekujeme."; // to samé jako:**

**// cout << "\nVas vek je: ";**

**// cout << vek;**

**// cout << "\nDekujeme.";**

**// neboli:**

**// cout << "\nVas vek je: ";**

**// cout << vek;**

**// cout << "\nDekujeme.";**

**Není jednoduší i praktičtější zkrácený zápis? Je, proto ho prosím používejte.**

**Zapomněl jsem v první lekci něco říct. Je jedno, jak bude program vypadat. Tedy např.:**

**cout << "ahoj";**

**cout << "cau";**

**cout << "bye";**

**nebo**

**cout << "ahoj"; cout << "cau"; cout << "bye";**

**Pokud je tam vše potřebné, překladač to nakompiluje správně. Spíše jak vidíte, z estetického hlediska a pro pozdější úpravy je to velmi nepřehledné, nehledě v těžších věcech.**

**V příští lekci si budeme povídat o uložení znaků do proměnné (tedy ne jen čísel) pomocí typu *char* a asi začneme podmíněnými výrazy *if*, *while*, *do while*...**

## Úkol č.2:

**Vytvořte program, který na prvních třech řádcích (!) pozdraví uživatele, oznámí mu, kdo je tvůrcem programu a požádá ho o věk, výšku a váhu.**

**Výsledek by měl vypadat přibližně takto:**

**Vase vyska je *x* cm. Vas vek je *x* let. Vase vaha je *x* kg.**

**kde *x* je to, co uživatel zadá (jedná se tady o více proměnných a musíte tedy použít vícekrát *cin* a *cout*).**

**Poznámka: Je mi jedno, jak bude vypadat kód, ale musí fungovat. Proto si vyberte takový styl zápisu, které vyhovuje právě vám.¨**

# [C++ - 2. lekce - dodatek](http://programujte.com/clanek/2005041405-c-2-lekce-dodatek/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       46 158×**

* **priority operátorů**
* **nejčastější chyby**

## Priorita operátorů

**Můžete C++ věřit při provádění komplikovaných aritmetických výpočtů? Ano, ale musíte znát pravidla, která C++ používá. Použijete-li více operandů, můžete se začít ptát, který se aplikuje první.**

**Příklad**

**int celkem = 3**

**+ 4 \* 5; // 35 nebo 23?**

**Pokud použijete více operandů, C++ uplatní pravidla o prioritě. Aritmetické operátory dodržují obvyklé algebraické pořadí - násobení, dělení a zjištění zbytku po celočíselném dělení se provádějí před sčítáním a odčítáním. Proto 3 + 4 \* 5 znamená 3 + (4 \* 5). Výsledek: 23.**

**Samozřejmě na posílení vašich priorit můžete libovolně používat závorky.**

**Ale...**

**float oper = 120 / 4 \* 5; // 150 nebo 6?**

**Operátory *\** a */* mají stejnou prioritu, který bude první?**

**V takovém případě C++ postupuje dle takovýchto priorit:**

**Jinak posuzuje, zda má operand asociaci zleva doprava, čí zprava doleva.**

**Ale vysvětlení je poněkud těžkopádné, proto jsou jednoduší závorky - jste si jistí, že se stane to, co chcete.**

****

****

**Ve sloupci PRIORITA vidíte několik skupin, všechny operátory v dané skupině mají stejnou prioritu a stejnou asociativitu, která je buď zleva doprava (v tabulce označena L-P) nebo zprava doleva (P-L).**

**To znamená, že například 120 / 6 \* 5 má asociativitu dle tabulky L-P - zleva doprava, bude to tedy stejné, jako (120/6) \* 5, což je 100.**

## Nejčastější chyby v úkolech:

1. **Nevím, proč includujete zbytečné či neexistující soubory (*#include <vcl.h>* a *#include <stdlib.h>***
2. **Místo *int main()* píšete jen *int main* - tam musí být závorky!**
3. **Nepoužíváte na konci *cin.get();* (viz 1. lekce)**
4. **Nepoužívejte syntaxe z C, učíte se přeci C++, ne? Žádné *printf* apod..**
5. **Nemůžete napsat jen tak *endl;* - musíte to poslat do výstupu, tedy: *cout << endl;* nebo *cout << " ";***

# [C++ - 3. lekce](http://programujte.com/clanek/2005041406-c-3-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       135 723×**

* **řetězce - char**
* **cykly a podmínky - for, while, do while, podmínka if else**
* **logické výrazy**
* **úkol č. 3**

## Řetězce

**Takže, začneme se učit o ukládání textu (znaků) do proměnných.**

**Pamatujete, jak jsme se učili o typech a zazněl tam *char*?**

**Vytváří se stejně jako typ *int*, ale má jinou inicializaci.**

**char prom[10]; // vytvoří proměnnou prom jako typ**

 **// char o maximální délce 9 znaků**

**Obecně:**

**char jméno\_proměnné[max\_počet\_znaků];**

**Trošku se zmíním o maximální délce znaků, protože se jistě ptáte, proč, když jsme dali maximální počet znaků 10, je jich jen 9.. Později se budeme učit o řetězcích *string* ale nyní o *char* ;) :**

**Styl řetězců má jednu zvláštnost: Poslednim znakem každého řetězce je \0 - tzv. prázdný znak. Tento znak je ASCII kódem 0 a slouží pro označení konce řetězce (0 = nula).**

**Abyste tedy mohli vytvořit proměnnou typu *char*, nesmíte zapomenout započítat poslední znak \0.**

**Chceme vytvořit proměnnou *jmeno*, která bude mít maximální možnou délku 7 znaků.**

**Napíšeme tedy:**

**char jmeno[8];**

**Musíme totiž započítat onen prázdný znak (proto 8).**

**Teď vám ukážu dva možné způsoby inicializace, doporučuju používat ten druhý, uvidíte ;)**

**char jmeno[8] = { 'b', 'e', 'r', 'u', 's', 'k', 'a', '\0' };**

**a druhý způsob:**

**char jmeno[8]="Beruska"; // znak \0 se už přidá sám**

**Jinak když vyvoláte *jmeno[3]*, tak se vám vyvolá 4. písmeno (musíte počítat i tu nulu), tedy u.**

**Tak, jaký způsob zápisu si vyberte? Myslím, že to vím ;)**

**Ještě vám ukážu jeden způsob:**

**char jmeno[]="Beruska";**

 **// nechá kompilator počítat a sám**

 **// zakomponuje prázdný znak \0**

**Pokud nevyplníte max. počet znaků a necháte tedy kompilátor počítat, MUSÍTE již rovnou inicializovat.**

**Nemůžete tedy udělat toto:**

**char jmeno[]; // toto nemůžete napsat a až poté inicializovat**

**Takže nyní byste měli dokázat stvořit program, který se zeptá na jméno, vek apod.**

**A teď ještě ukážu, nač se dále používá *char*.**

**Na ASCII systému je 'S' jiným způsobem zápisu 83.**

**Proto příkaz...**

**char pismeno=64; // ...je @, tedy když vyvoláme**

 **// pismeno, dostaneme @**

**Díky tomu můžeme vložit češtinu do našeho programu.**

**char ch=159; // 159 == č**

**cout << ch << "tverec"; // výsledek : čtverec**

## Cykly

**Počítače provádějí více než jen ukládání dat. Analyzují, spojují, přeskupují a jiným způsobem manipulují s daty. Na provádění svých manipulačních zázraků potřebují nástroje na vykonávání opakovatelných činností a na učinění rozhodnutí. C++ je samozřejmě poskytuje.**

**Jsou to: *for*, *while*, *do while*, *if else* a *switch*.**

**Používá je i C, takže pokud je znáte, máte plus.**

### a) for

**Pokud chcete něco poslat na výstup vícekrát, než jednou, *for* vám pomůže.**

**Nejprve příklad:**

**int i;**

**for (i=0; i<5; i++) //1.řádek**

 **cout << "Zde je ten text.\n"; //2.řádek**

**cout << "konec"; //3.řádek**

**V prvním řádku píšeme samotnou podmínku, ve druhém příkaz, který se provede, pokud je výsledek pravdivý a ve třetím příkaz, který se provede, pokud je podmínka nepravdivá.**

**Výsledek by tedy byl:**

**Zde je ten text.**

**Zde je ten text.**

**Zde je ten text.**

**Zde je ten text.**

**Zde je ten text.**

**konec**

**Druhou část jistě chápete (2. řádek), proto si vysvětlíme první:**

**Čtěte *for* jako "pokud", ulehčí vám to práci.**

**cyklus začíná nastavením proměnné *i* (vytvořené před cyklem) na nulu.**

**i = 0; // pozor, musí být středník!**

**Toto byla inicializační část cyklu.**

**Dále se testuje podmínka:**

**i < 5;**

**zda-li je i menší, než 5. Je-li tomu tak, program provede následující příkaz:**

**cout << "Zde je ten text.\n";**

**Potom cyklus použije změnovou část cyklu, aby zvýšil *i* o 1:**

**i++ //vysvětlení v minulé lekci i+1**

**Jak tedy probíhá děj step-by-step? ;)**

**i=0, 0<5 a tak vytiskne text *Zde je ten text.***

**i=1, 1<5 a tak vytiskne text *Zde je ten text.***

**i=2, 2<5 a tak vytiskne text *Zde je ten text.***

**i=3, 3<5 a tak vytiskne text *Zde je ten text.***

**i=4, 4<5 a tak vytiskne text *Zde je ten text.***

**i=5, 5<5**

**neplatí, a tudíž se cyklus opouští a následuje další, na cyklu nezávislý příkaz: *konec***

***For* tedy probíhá jen tehdy, dokud je děj pravdivý.**

**Obecně:**

**for (inicializace; testovací\_výraz; změnový\_výraz)**

 **příkaz**

**další příkazy**

**Inicializace se provádí pouze jednou, jinak by neměl cyklus smysl. Při dalším úkonu tedy není *i* znovu 0, ale nabývá hodnoty dle *změnového výrazu*.**

**Vytvořit i inicializovat proměnnou můžete přímo v cyklu *for*, ale poté již není možné proměnnou vyvolat (Proměnná bude existovat jen v cyklu, mimo něj neexistuje!).**

**for (int i=0; i<5; i++)**

 **cout << "Zde je ten text.\n";**

**cout << "Konec, protoze hodnota i je " << i;**

 **// NELZE!**

**Proměnná *i* je viditelná jen v tučné části textu! (v cyklu)**

**\* \* \* Nyní prosím přejděte úplně nakonec této lekce a přečtěte si text pod čarou, poté se sem opět vraťte.**

### b) WHILE

**Cyklud while je cyklus for zbavený inicializační a změnové části, má pouze jednu testovací podmínku a příkaz.**

**while (testovací\_podmínka)**

 **příkaz**

**Zde také platí, že se cyklus stále opakuje, pokud je podmínka pravdivá. Jakmile je nepravdivá, opouští se cyklus.**

**int i=0;**

**while (i<5)**

**{**

 **cout << "ahoj\n";**

 **i++;**

**}**

**Díky while můžeme vytvořit časovou prodlevu :**

**long cekej=0;**

**while (cekej < 10000)**

 **cekej++; // počítá potichu**

**Problém nastává, že na každém PC je jiný procesor a tudíž jiná rychlost :(
Proto se používá jiný (lepší) způsob, ale o něm později ;)**

### c) do while

**Třetím cyklem je do while. Liší se od obou předchozích, ptž je cyklem se vstupní podmínkou. Tzn, že cyklus nejprve provede tělo cyklu, a potom teprve vyhodnotí výraz, aby věděl, zda by měl pokračovat dalším průběhem. Když se podmínka ukáže nepravdivým, cyklus končí.

V jako jediném se tedy příkazy provedou minimálně jednou.**

**do**

 **příkaz(y)/blok // nebo-li tělo**

**while (testovací\_podmínka)**

**Může být použitelný např. při hádání čísla :**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **int n;**

 **cout << "Zdej cislo od 1 do 10 pro nalezeni ";**

 **cout << "meho oblibeneho cisla\n";**

 **do**

 **{**

 **cin >> n; // provádí tělo**

 **} while (n != 7); // potom testuje**

 **cout << "Ano, 7 je me oblibene cislo.\n" ;**

 **return 0;**

**}**

**Cyklus skončí až tehdy, kdy uživatel uhodne číslo /v tomto případě 7/.**

### d) if else

**Čtěte if jako jestliže. A to přesně dělá, vyhodnotí podmínku a je-li pravdivá, provede ji, v opačném případě ji prostě přeskočí.**

**if (testovací\_podmínka)**

 **příkaz1**

**else**

 **příkaz2**

**není na tom nic složitého.
Nejprve se testuje podmínka. je-li pravdivá provede se příkaz1 (a příkaz2 již NE) , nebo, pokud podmínka není pravdivá, je proveden příkaz2 (příkaz jeden nebude proveden). else vlastně nám jen říká, že i v případě nepravdivé podmínky se má něco udělat.**

**I zde platí že můžeme použít bloky.
else a příkaz2 můžeme vynechat :**

**if (testovací\_podmínka)**

 **příkaz1**

**To způsobí, že není-li podmínka pravdivá, pokračuje program dále.. nic se neprovede.. např :**

**int heslo;**

**cout << "Zadejte heslo : ";**

**cin >> heslo;**

**if (heslo == 1892)**

 **cout << "Spravne, vase heslo je v poradku.";**

**else**

 **cout << "Vase heslo je spatne.";**

**Co možná děláte za chybu?
Nesmíte použít operátor přiřazení "=" ale podmínky rovnosti "==" //lekce 2
Podmínky lze samozřejmě stupňovat.**

**int n;**

**int Fave = 27;**

**cout << "Zadejte cislo v rozsahu 1-100 ke zjisteni ";**

**cout << "mého oblibeneho cisla: ";**

**cin >> n;**

**if (n < Fave)**

 **cout << "Prilis nizke -- hadejte znovu: ";**

 **else if (n > Fave)**

 **cout << "Prilis vysoke -- hadejte znovu: ";**

 **else**

 **cout << Fave << " je spravne!\n";**

## Logické výrazy

**Často musíte testovat jednu, či více podmínek.**

### Logický operátor NEBO ||

**-znamená, že je-li alespoň jeden výraz pravdivý, je podmínka pravdivá. 5 ==5 || 5==9 // pravdivá, protože 1. výraz je pravdivý 5 > 8 || 5 < 2 // nepravdivá, oba jsou nepravdivé tyto výrazy můžete použít např v cyklech**

**cin >> heslo;**

**if (heslo==1892 || heslo==2981)**

 **cout << "Spravne";**

**else**

 **cout << "Spatne";**

**Pravdivá jen tehdy, pokud uživatel zadal heslo 1892 nebo 2981.**

### Logický operátor A &&

**-znamená, že jsou li oba výrazy pravdivé, je podmínka pravdivá.**

**5 == 5 && 4 == 4 // pravdivá, oba jsou pravdivé**

**5 == 3 && 4 == 4 // nepravdivá, jeden je nepravdivý**

**5 > 6 && 8 < 7 //nepravdivá, oba jsou nepravdivé**

**tyto výrazy můžete použít např v cyklech:**

**int vek;**

**cin >> vek;**

**int heslo=1892;**

**if (heslo==1892 && vek>=18)**

 **cout << "Spravne";**

**else**

 **cout << "Spatne";**

**Pravdivá jen tehdy, pokud uživatel zadal dobré heslo a je mu více než 18 let.**

### Logický operátor NE !

**-obrací pravdivostní hodnotu výrazu
Pokud je tedy vyraz pravdivý, !vyraz je nepravdivý a naopak.**

**if ( ! ( x > 5 ) ) // if (x <= 5)**

## Úkol

**Na začátek Vás program 10x uvítá(10x se vypíše ta samá hláška pomocí některého cyklu.
Poté se zeptá na Váše jméno a uloží jej, a poté Vám vypíše první písmeno ve jméně.
Vše musí být přehledné,odřádkované apod.
Dále vytvoříte proměnnou (typ char ktera bude mit místo na 5 čísel ) a řeknete uživateli, ať zadá 1. jednociferné číslo, až ho zadá, uloží se jako 1.znak v proměnné typu char , zeptá se znova a uloží až do té doby, až uživatel zadá všech 5 čísel. (Řešte pomocí cyklu)
Nyní Vypíše program první 2 písmena vašeho jména a za ně s mezerami hodnoty, které jste zadali.
Nakonec poděkuje a seznámí uživatele se stránkami www.programujte.com o čem jsou a jak mohou pomoci. (na 3 řádky, ne obecně, ale vaše názory na ně)
Tak, doufám, že máte co dělat ;))

\* \* \*
Zápis příkazů:**

**Jak můžete například u for, while apod. zapisovat**

**for (i=0; i<5; i++)**

 **cout << "ahoj.";**

**tak můžete použít i více příkazů. A to tak, že je dáte do tzv. BLOKU.**

**for (i=0; i<5; i++)**

 **{**

 **cout << "ahoj.";**

 **cout << "ahoj.";**

 **}**

**Vše, co je v závorkách { } je tělo f-ce, a všechny se provedou je-li podmínka pravdivá.
Více v dodatku, který vyjde. (spíše jen o možnostech tohoto vyžití)**

# [C++ - 3. lekce - test](http://programujte.com/clanek/2005041407-c-3-lekce-test/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       47 382×**

* **test probraných lekcí**

## Zkouška č. 1

**Vítám Tě při 1. zkoušce.**

**Úkol NIKAM neposílej! je to pouze zkouška pro tebe, zda jsi zvládl probranou látku. Ovšem program, co zde ve 3. úkolu vytvoříš, se ti bude náramně hodit.**

**1) najdi chybu v příkladu :**

**1: include <iostream>**

**2: int main()**

**3: {**

**4: cout << "Ahoj svete**

**";**

**5: return 0;**

**6: }**

**2) Vypište typy komentářů, a určete rozdíly.**

**3) Vytvořte program pomocí cyklu,který vám na obrazovku vypíše všechny znaky, jako byste psali např. @ (Alt+64). Program se Vám bude hodit, ptž vypíše jak české znaky, tak určité znaky, o kterých možná ani nevíte. Pořád nevíte jak nato? je to prosté, využijte schopnosti jak dostat češtinu do programu. Detailisti mohou ještě ve výpisu ukázat, jaký kód ke kterému znaku patří.**

**např.: *64  -   @***

# [C++ - 4. lekce](http://programujte.com/clanek/2005041408-c-4-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 4. 2005       92 239×**

* **nový typ bool**
* **kvalifikátor const**
* **úvod do polí – inicializace a volání prvků**
* **zajímavosti s řetězci**
* **řádkově orientovaný vstup: getline() a get()**
* **míchání numerického a řetězového vstupu**
* **úkol č. 4**

**Tak nějak jsme se dostali ke 4. lekci. Co dnes probereme? Samé zajímavé věci! Naučíme se, jak zjistit velikost proměnné, počet znaků, vytvářet pole apod.**

## Nový typ bool

**Naučíme se používat jeden další typ, nový v C++. Bool má hodnotu buď pravda (true), nebo nepravda (false), tedy 1 a 0. True je tedy jakákoliv nenulová hodnota, false nulová hodnota. Literály true a false se mohou konvertovat do typu int. True se interpretuje jako 1, false jako 0.**

**bool odpoved = true;**

**int odpov = false;**

**bool start = -100; // start se přiřadí true**

**bool stop= 0; // stop se přiřadí false**

## Kvalifikátor const

**Slouží na vytváření symbolických konstant – použitím klíčového slova const. Předpokládejme, že chcete vytvořit konstantu na počet měsíců v roce:**

**const int MESICE = 12; // mesice je konstanta pro 12**

**Nyní můžete v programu místo 12 použít MESICE. Po inicializaci konstanty se nastaví hodnota a kompilátor vám následně nedovolí změnit hodnotu proměnné MESICE, protože proměnná je konstantní. Pokud byste chtěli do takovéto proměnné přiřadit poté hodnotu, zobrazí se chybové hlášení, že je překročena Lvalue. Je to to samé, jako byste chtěli přiřadit 5 do 6. (Lvalue je hodnota, jako například proměnná, která se vyskytuje na levé straně přiřazovacího operátoru.)**

**Jedna taková maličkost: názvy proměnných, které inicializujete jako konstantu, pište velkými písmeny, abyste si připomněli, že se jedná o konstantu. Obecný tvar pro vytvoření konstanty je:**

**const typ jméno = hodnota;**

**Při vytváření konstanty ji musíte i inicializovat!**

**Poznámka od**[**xHire**](http://programujte.com/autori.php?autor=45&kolikata=1)**: Přecházíte-li na C++ z C a chystáte se použít #define na definování konstanty, použijte místo toho const.**

**Deklarace znamená vytvoření proměnné (např. *int i;*), definice znamená deklarace a následná inicializace proměnné v jednom příkazu (např. int i = 1;), samotná inicializace znamená přiřazení hodnoty proměnné – bez deklarace v témže příkazu (např. i = 1;).**

## Úvod do polí

**Brzy dojdete k závěru, že potřebujete něco víc, než jsou jen jednoduché základní typy. C++ vám nabízí víc – odvozené typy. Nejdále dosažitelným typem je třída, lahůdka C++, ke které se dopracujeme. Ale C++ podporuje i několik skromnějších typů převzatých z C. Například pole – může obsahovat několik hodnot stejného typu. Určitý druh pole může obsahovat řetězec. Struktury mohou obsahovat několik proměnných různých typů a nakonec ukazatele, které říkají počítači, na jaké adrese (kde) jsou umístněna data. Postupně se ke všemu propracujeme.**

**Pole je datová forma, která může obsahovat několik hodnot stejného typu, například 60 hodnot typu int. Každá hodnota se ukládá do samostatného prvku pole a počítač je ukládá za sebou do paměti. Deklarace pole obsahuje tři věci:**

* **typ hodnoty, která má být uložena do každého prvku**
* **jméno pole**
* **počet prvků v poli**

**Pole se vytvářejí jako klasické proměnné, jen za ně přidáme hranaté závorky, které obsahují počet prvků (už jste se setkali s polem char):**

**typ jméno\_pole [velikost\_pole];**

**Např.:**

**short mesice[12]; //vytváří pole 12ti short**

**…vytváří pole, které se jmenuje mesice a má 12 prvků, z nichž každý může obsahovat hodnotu typu short. Každý prvek je vlastně samostatná proměnná. Velikost pole musíme hned nastavit. K prvkům pole můžeme přistupovat jednotlivě, číslování začíná od 0. Takže mesice[0] je prvním prvkem pole mesice a mesice[11] je posledním prvkem pole mesice.**

**Všimněte si, že index posledního prvku je o 1 menší než velikost pole. (Je to dáno právě tím, že číslování začíná od 0.)**

****

### Inicializace prvků

**Jsou 2 možné způsoby:**

1. **Vytvoříme si pole hromada a inicializujeme jednotlivé prvky:**

**int hromada[3];**

**hromada[0] = 5;**

**hromada[1] = 2;**

**hromada[2] = 8;**

1. **Vytvoříme si pole hromada a rovnou inicializujeme:**

**int hromada[3] = {5, 2, 8};**

### Volání prvků

**Prvky voláme klasicky:**

**cout << hromada[0]; // vypíše obsah prvního prvku,tedy 5 \*/**

**cout << hromada; // vypíše adresu pole - tedy tak ne**

**Při vytváření polí můžeme nechat kompilátor počítat:**

**int mesice[] = {5, 4, 3, 2}; // nechá kompilátor vypočítat velikost pole**

**Můžete také inicializovat pole částečně:**

**int mesice[5] = {5, 4}; // inicializujete pouze první 2 prvky**

**Zbylé prvky jsou inicializovány automaticky na nulu, není tedy těžké je všechny inicializovt na nulu:**

**int mesice[5] = {0};**

## Zajímavosti s řetězci

**Abychom zpřístupnili standardní knihovní f-ci strlen() na zjištění délky řetězce, musíme includovat cstring (string.h pro starší implementace), tedy:**

**#include // zpřístupní f-ci strlen()**

**Mezi závorky dejte proměnnou, u které chcete zjistit délku řetězce:**

**char jmeno[] = "Curo";**

**cout << "vase jmeno ma delku " << strlen(jmeno) << " znaky.";**

**Výpis:**

**vase jmeno ma delku 4 znaky**

**Dále se naučíme, jak zjistit velikost proměnné, použijeme k tomu f-ci sizeof. Například máme proměnnou vek:**

**int vek = 15;**

**cout << sizeof vek; // vypíše velikost proměnné vek**

### Nějaké další vychytávky

**Podobně jako \n používáme v textu na výstup, můžeme použít další sekvence:**

| **akce** | **sekvence** |
| --- | --- |
| **nový řádek** | **\n** |
| **horizontální tabulátor** | **\t** |
| **vertikální tabulátor** | **\v** |
| **návrat na předchozí znak** | **\b** |
| **návrat vozu** | **\r** |
| **zvuková výstraha** | **\a** |
| **zpětné lomítko** | **\\** |
| **otazník** | **?** |
| **jednoduchá uvozovka** | **\'** |
| **dvojitá uvozovka** | **\"** |

**Příklad:**

**cout << "Napis tvuj kod : \_\_\_\_\_\b\b\b\b\b";**

## Řádkově orientovaný vstup: getline() a get()

**Abyste byly schopni zavádět celé věty místo jednotlivých slov, potřebujete pro vstup řetězce jiný přístup. Například f-ce getline() čte celý řádek a přitom používá znak nového řádku (Enter) na označení konce vstupu.**

**Tuto metodu využíváme použitím cin.getline() jako funkční volání. Má dva parametry. Jeden je jméno pole a druhý je počet znaků, které se mají přečíst. Je-li nastavena velikost na 20 znaků, f-ce přečte 19 znaků a dvacátý použije na dodání znaku null na konec (\0). F-ce ukončuje vstup, jakmile hranice překročí nastavený počet znaků nebo se stiskne Enter, záleží na tom, co přijde první.**

**Například, chcete-li uložit jméno do pole o velikosti 20 znaků s pomocí f-ce getline():**

**char jmeno[20];**

**cin.getline(jmeno, 20); // přečte 19 znaků a uloží**

**Přečte celý řádek do pole jmeno za předpokladu, že řádek obsahuje 19 nebo méně znaků.**

**#include**

**using namespace std;**

**int main()**

**{**

 **char jmeno[20];**

 **cout << "Zadejte vase jmeno:\n";**

 **cin.getline(jmeno, 20); // čte celý řádek**

 **cout << "vase jmeno je " << jmeno << ".\n";**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

**Teď zabrousíme trochu do praktičnosti. Když chcete změnit např. velikost pole jmeno či maximální počet přečtených znaků, vytvořte si konstantu, které přidělíte tento počet, a vždy, zvláště ve složitějších příkladech, stačí změnit hodnotu konstanty a tím pádem se změní vše, pomocí jednoho zásahu:**

**#include**

**using namespace std;**

**int main()**

**{**

 **const int VEL = 20;**

 **char jmeno[VEL];**

 **cout << "Zadejte vase jmeno:\n";**

 **cin.getline(jmeno, VEL); // čte celý řádek**

 **cout << "vase jmeno je " << jmeno << ".\n";**

 **return 0;**

**}**

**Co vlastně dělá onen cin.getline();? Čte znak nového řádku, tedy čeká na stisknutí Enteru.**

**Další variantou je cin.get(); s parametrem. Funguje úplně stejně jako cin.getline();.**

**Pokud používáte příkaz cin.get(); vícekrát, pište raději za něho příkaz cin.get(); bez parametru:**

**cin.get(jmeno, 20); // čte první řádek**

**cin.get(); // čte znak nového řádku**

**cin.get(jmeno2, 20); // čte druhý řádek**

**nebo**

**cin.get(jmeno, 20).get(); // zřetězení členských f-cí**

**Nepočítám s tím, že to hned pochopíte.**

**Vysvětlení:**

**cin.get(jmeno, 20); // jmeno a 20 jsou paametry předávané f-ci cin.get();**

**Jediný rozdíl mezi cin.getline(); a cin.get(); je, že cin.get(); zanechává znak nového řádku ve vstupní frontě.**

**cin.get(jmeno, 20);**

**cin.get(jmeno2, 20); // problém**

**Protože první volání zanechává znak nového řádku ve vstupní frontě, tento znak je prvním znakem, který uvidí druhé volání. Tedy get(); dospívá k závěru, že dosáhla konce řádku aniž by nalezla něco k přečtení. Volání cin.get() (žádné parametry) čte samotný následující znak, i když je to znak nového řádku. Chce to vyzkoušet, abyste pochopili, jak to pracuje.**

### Proč tedy používáme cin.get(), když máme cin.getline()?

**Starší implementace možná cin.getline(); nemají a také vás to nutí být pečlivější. Někdy v budoucnu vám ukážu, v čem je někdy cin.get(); lepší (k ošetření chyb), ale cin.getline(); – nejlepší volba pro čtení celého řádku.**

**Teď si představte, že jste vytvořili pole o velikosti 20 znaků a uživatel zadá 28 znaků, tedy více, než bylo alokováno. Tyto znaky přijdou do vstupní fronty, což by se zapsalo do následující proměnné, a to přece vůbec nechceme! Proto to můžeme ošetřit tím, že za cin.get(); s parametrem (ne za cin.getline();) napíšeme cin.clear();**

**cin.get(jmeno, 20);**

**cin.clear(); // vymaže znaky ve vstupní frontě**

## Míchání numerického a řetězového vstupu

**Toto může zapříčinit problémy. Jak se jich vyvarovat?**

**#include**

**using namespace std;**

**int main()**

**{**

 **cout << "V kterem roce byl vas dum postaven?\n";**

 **int year;**

 **cin >> year;**

 **cout << "Jaka je jeho adresa?\n";**

 **char address[80];**

 **cin.getline(address, 80);**

 **cout << "Rok vystavby: " << year << "\n";**

 **cout << "Adresa: " << address << "\n";**

 **cout << "Konec!\n";**

 **cin.get();**

 **return 0;**

**}**

**Výpis:**

**V kterem roce byl vas dum postaven?**

**1966**

**Jaka je jeho adresa?**

**Rok vystavby: 1966**

**Adresa:**

**Konec!**

**Nikdy nedostanete příležitost zadat adresu. Problém spočívá v tom, že když cin přečte rok, tak zanechá znak nového řádku generovaný klávesou Enter ve vstupní frontě. Potom cin.getline() přečte nový řádek jako prázdný a přiřadí prázdný řetězec do pole address. Oprava spočívá v přečtení a odhození znaku nového řádku před čtením adresy, např. použitím cin.get(); bez parametru.**

## Trošku lidštěji

**cin zanechává vždy ve vstupní frontě Enter, který automaticky jde do dalšího vstupu. Pokud tedy je cin >> vek;, uloží hodnotu do proměnné vek a nechá Enter do dalšího vstupu. Hned následující cin >> vyska; nejprve dostane Enter z předchozího vstupu, ale (pokud jste si všimli, cin vás jen tak nepustí, pokud jen zmáčknete Enter, aniž byste zadali hodnotu) ten přeskočí a čeká na hodnotu. Zadáme 8 a stiskneme Enter, ten jde zas do vstupní fronty. cin.getline() si s tím už ovšem neporadí, protože řetězec může být prázdný a přijme tedy Enter a tím i ukončí vstup, což právě přeskočí vstup do proměnné address. Jak to tedy opravit?**

**To je úkol pro vás! Není to vůbec těžké, proto ještě přidáme:**

## Úkol č. 4

1. **Opravit kód, aby fungoval v plné míře.**
2. **Sestavte program, který se v tomto pořadí zeptá na následující položky:**
	1. **Na celé jméno (řetězec se uloží do jednoho pole), otázka bude vypadat přesně takhle (včetně češtiny!):
	„Vaše jméno prosím: “.**
	2. **Na váš věk.**
	3. **Na pět čísel, která se postupně uloží do dalšího pole; tato čísla sečte a ukáže výsledek.**
	4. **Program vypíše jméno pozpátku (help – cykly).**
	5. **Vše bude hezky provedeno, tedy odřádkováno, odmezerováno a s použitím znaků z tabulky, co jste dělali v testu, tedy nějaké ohraničení či čáry apod.**

**Věřím, že to budete mít hned :o).**

# [C++ - 5. lekce](http://programujte.com/clanek/2005041409-c-5-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Josef Vítů**](http://programujte.com/profil/20356-josef-vitu/)**13. 4. 2005       87 111×**

* **struktury – deklarace, inicializace, volání**
* **pole struktur**
* **bitová pole**
* **uniony**
* **úkol č. 5**

## Struktury

### Úvod do struktur

**Předpokládejme, že chcete (a nekecejte, že ne) uložit jméno, věk a BMI index několika osob. Pole, leč může obsahovat několik položek, to neudělá, protože musí být stejného typu. Odpovědí je struktura v C++. Chcete-li si udělat přehled o všech zaměstnancích, vytvoříte pole struktur. Struktura je jistým prostředkem vedoucím do lahůdky C++ v OOP, třídy. Díky strukturám se přiblížíte OOP (Objektově orientovanému programování), srdci C++.**

**Struktury tvoříme nejčastěji vně f-ce main. Nazývá se externí deklarací struktury. Proč? Externí deklarace se může použít u všech f-cí, které ji následují, kdežto interní deklarace (např. hned za složenou závorkou za int main) se může použít pouze ve f-ci, ve které se nachází.**

###  Deklarace struktury:

**struct nazev\_promenne**

**{**

 **char jmeno[10];**

 **int vek;**

 **double bmi;**

**};**

### Inicializace struktury

**(za nazev\_promenne jsme si zvolili lide):**

**lide ales =**

**{**

 **"Ales", //jmeno**

 **22, // vek**

 **11.49 // bmi index**

**};**

### Volání některé proměnné ze struktury

**Chceme-li vyvolat věk alese, napíšeme:**

**cout << ales.vek;**

**To vypíše 22, tedy věk Aleše.**

**. je operátor příslušnosti. (ales.vek – proměnná vek ve struktuře ales)**

**Vytvořit struktur si můžeme, kolik chceme, klidně celý počet zaměstnanců. Tedy např.:**

**lide petr =**

**{ ... }**

**…atp**

**Nebo si můžeme vytvořit prázdné struktury:**

**lide martin, honza, alena;**

**Vytvořili jsme tři struktury, každá má svoje proměnné stejně jako ales, ale nemají žádné hodnoty. K čemu by nám ale byly struktury, které mají pevně stanovené hodnoty. Pokud chceme například zapsat vek do struktury alena, napíšeme:**

**alena.vek = 17;**

**Je to snadné, stejně jako proměnné, jen tu je navíc tečka, která nám ukazuje, kam daná proměnná patří.**

**Vytvořit struktury můžeme už při její deklaraci:**

**struct lide**

**{**

 **int vek;**

**} milan;**

**Je to stejné jako:**

**struct lide**

**{**

 **int vek;**

**};**

**lide milan;**

**Kdo přesně stále nechápe struktury, malé vysvětlení:**

**struct osoba**

**{**

**char jmeno[20];**

**int vek[20];**

**char prace[20];**

**} zamestnanec;**

**Proč nefunguje např. *cout << osoba.jmeno;*, ale *cout << zamestnanec.jmeno;*? Co značí „osoba“ a co „zamestnanec“?**

**Protože:**

**struct osoba**

**{**

**char jmeno[20];**

**int vek[20];**

**char prace[20];**

**};**

**…je jen DEFINICE, nemá žádné hodnoty.**

**Vysvětlím to jinak:**

**Jsou to šuplíky, koupil sis nábytek se třemi šuplíky a teď do nich chceš dát mrkev, hrách a petržel a nazveš to ZELENINA. Takže pokud chceš hrách, dáš *zelenina.hrach***

**A zase: můžeš do nich dát i rýži, fazole a čočku a nazveš to LUSTENINY. Přístup bude: *lusteniny.fazole***

**Ale všechno můžeš mít v těch šuplících. Oni dávají jen místo, je na tobě, jak si to celé pojmenuješ a pak jim dáš hodnotu.**

**Příklad použití:**

**#include <iostream>**

**using namespace std;**

**struct rostliny**

**{**

**char jmeno[20];**

**float vaha;**

**double cena;**

**};**

**int main()**

**{**

**rostliny venkovni =**

**{**

**"slunecnice",**

**0.20,**

**12.49**

**};**

**rostliny vyber;**

**cout << "kytice: " << venkovni.jmeno << " za ";**

**cout << venkovni.cena << " Kc\n";**

**vyber = venkovni; // přiřadí jednu strukturu druhé**

**cout << "vyber: " << vyber.jmeno << " za ";**

**cout << vyber.cena << " Kc\n";**

**return 0;**

**}**

**Tady jsme si ukázali i kopírování jedné struktury do druhé.**

### Pole struktur

**Je možné vytvořit i pole, jehož prvky jsou struktury:**

**rostliny vnitrni[100]; // pole o 100 strukturách rostliny**

**vnitrni[0].cena; // pouzije promennou cena ze struktury vnitrni[0]**

### Bitová pole

**C++ (podobně jako C) vám umožňuje specifikovat strukturní členy, které zaujímají určitý počet bitů. To můžete použít např. na vytvoření datové struktury, která odpovídá registru nějakého hardwarového zařízení.**

**struct vypocet**

**{**

 **int key : 4; // 4 bity pro hodnotu key**

 **bool otazka : 1; // platný vstup (1 bit)**

**};**

**Pro přístup k bitovým polím použijte stejnou syntaxi jako u struktur.**

**vypocet.key = ...**

**Bitová pole se často používají na úrovni strojově orientovaného programování.**

### Uniony

**Union je datovým formátem, který může používat různé datové typy, ale pouze jeden typ v daném okamžiku. Zatímco struktura může používat typy int, long a float, union může buď int, nebo long, nebo float. Syntaxe je podobná jako u struktur, jen význam různý.**

**union zkouska**

**{**

 **int hodn1;**

 **long hodn2;**

 **double hodn3;**

**};**

**Proměnnou zkouska můžete využít na uchování int, long, double pouze tak dlouho, dokud potřebujete:**

**zkouska stetec;**

**stetec.hodn1 = 15; // uloží int**

**cout << stetec.hodn1;**

**stetec.hodn2 = 1.25; /\* uloží double, hodnota int
 se ztratí \*/**

**cout << stetec.hodn2;**

**Protože v jednom okamžiku proměnná obsahuje jen jednu hodnotu, musí mít dostatek prostoru na úschovu svého nevětšího členu. Z toho důvodu je velikost unionu velikostí největšího členu. Kde se používá? Úspora prostoru…**

## Úkol:

**Vytvořte fiktivní podnik, který bude mít x struktur na seznam zaměstnanců. X je počet zaměstnanců, které určí uživatel na začátku programu. Poté bude uživatel vyzván k zadání 1. zaměstnance a tak dále, dokud se nedostane k poslednímu zaměstnanci, nebo pokud uživatel již nechce přidat dalšího zaměstnance (mezi přidáním dalšího zaměstnance bude otázka, zda si přeje založit dalšího zaměstnance). Pokud odpoví, že již ne, program vypíše všechny údaje o všech zaměstnancích v TABULCE (použití \t, \n apod.) a název firmy. Každý zaměstnanec bude mít následující udaje: jmeno, vek a druh prace. Maximální počet zaměstnanců bude 32 000 (aby to bylo int) ;)Rada pro ty, co neví: struktury a cykly.**

# [C++ - 6. lekce](http://programujte.com/clanek/2005041410-c-6-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Josef Vítů**](http://programujte.com/profil/20356-josef-vitu/)**13. 4. 2005       55 965×**

* **funkce**
* **definice funkce**
* **vytváření prototypů**
* **volání funkce**
* **parametry funkcí a předávání hodnotou**
* **násobné parametry**
* **úkol č. 6**

## Funkce – programové moduly v C++

**C++ poskytuje velikou knihovnu užitečných f-cí, ale skutečné potěšení nastává při psaní vlastních f-cí. Abyste uměli používat f-ce, musíte udělat následující:**

1. **Definici funkce**
2. **Prototyp funkce**
3. **Vyvolat funkci**

**Jestliže používáte knihovní funkci, tak již byla definována a překompilována. Například knihovní f-ce cstring zahrnuje f-ci strlen() k nalezení délky řetězce. cstring obsahuje i další f-ce, ale takto vyvolaná knihovní f-ce vám poté dovolí kdykoliv vyvolat f-ci strlen() ve vašem programu.**

### Definice funkce

**Funkce můžeme rozdělit na dva druhy – funkce bez návratové hodnoty, funkce s návratovou hodnotou. Funkce bez návratové hodnoty se nazývají f-ce typu void a mají následující obecný tvar:**

 **void jméno\_funkce(seznam\_parametrů)**

 **{**

 **příkaz(y)**

 **return; // nepovinné**

 **}**

**Pochopte tuto f-ci tak, že když ji zavoláte, udělá to, co v ní napíšete. seznam\_parametrů zde určuje typ a počet parametrů předaných f-ci, tedy jaké hodnoty jí předáte v době volání. Nepovinný příkaz return zde ukazuje na konec f-ce. F-ce bez návratové hodnoty používáme, když chceme například provést nějaký cyklus (neboli f-ce bez návratové hodnoty nám slouží ke zpřehlednění samotného kódu v main). F-ci odtud pak již jen zavoláme a je jedno, zda bude součástí souboru, či si ji vyvoláme jako externí soubor. Například definice f-ce na vytištění deseti pozdravů:**

 **void pozdrav(int n) /\* funkce bez návratové hodnoty, předává se jí při volání hodnota \*/**

 **{**

 **for (int i=0; i<n; i++)**

 **cout << "Ahoj \n";**

 **cout << "konec";**

 **}**

**Parametr int n v seznamu znamená, že f-ce pozdrav() očekává při jejím volání hodnotu typu int (proměnná n zde není důležitá).**

**Funkce s návratovou hodnotou vyžadují použití příkazu return, který se stará o navrácení hodnoty volající f-ci:**

 **jmeno\_typu jmeno\_funkce(seznam\_parametru)**

 **{**

 **příkazy**

 **return hodnota; /\* hodnota má stejný typ,
 jako jméno\_typu\*/**

 **}**

**Hodnotou myslíme proměnnou, konstantu či obecněji výraz. Hodnota se vždy reprodukuje na jmeno\_typu nebo se na něj konvertuje (je-li definován typ double a navrácená hodnoty je typu int (např. 8), tak se konvertuje int na double (8.0)), návratová hodnota nemůže být pole. Vše ostatní je možné. F-ce se ukončí, jakmile narazí na příkaz return. F-ce ale může mít i více než jeden příkaz return – např. jako alternativy k různým volbám:**

 **int porovnani(int a, int b)**

 **{**

 **if (a > b)**

 **return a; // jel-li a>b, funkce se ukončí zde a navrátí hodnotu a**

 **else**

 **return b; // jinak se f-ce ukončí zde a navrátí se hodnota b**

 **}**

**Else zde není třeba, ale pomůže lépe pochopit závěr. Následující příklad nám vrací druhou mocninu čísla:**

 **double ctverec(double x)**

 **{**

 **return x \* x; // hodnota typu double**

 **}**

**Jestliže tedy při volání f-ce předáme hodnotu 2, navrátí nám f-ce hodnotu 4.**

### Vytváření prototypů

**Prototyp vytvoříme velmi lehce (nezabývejte se teď tím, kam jednotlivé části f-ce patří). Stačí zkopírovat první řádek definici f-ce a dát za něj středník. Pokud tedy použijeme v programu naše napsané f-ce porovnani a ctverec, jejich prototyp bude vypadat takto:**

 **int porovnani(int a, int b);**

 **double ctverec(double x);**

**Prototyp popisuje rozhraní f-ce ke kompilátoru, říká mu, jakého typu je návratová hodnota, počet a typ parametrů f-ce. Stále jste možná zvědaví, proč prototyp f-ce kompilátor potřebuje. Kompilátor se díky prototypům seznámí se všemi f-cemi použitými v programu, aniž by je musel „hledat“. Neboli čtení kódu probíhá shora dolů a museli byste bez použití prototypu f-ci definovat před jejím prvním použitím. To není efektivní, protože hlavní f-cí je main. Prototyp tedy oznamuje přítomnost dalších f-cí a odkáže kompilátor na její umístnění. Prototyp f-ce nevyžaduje jména proměnných – seznam typů postačuje.**

**void pozdrav(int); /\* vypustit jména proměnných
 je v prototypech v pořádku \*/**

**Automatická konverze ovšem nemusí vždy konvertovat hodnotu správně, a to v případě z většího na menší. Například předáváte-li hodnotu 8.54E26 f-ci, která čeká int, nemusí se konvertovat tak velké číslo správně. V tomto případě vás kompilátor varuje před možnou ztrátou dat.**

### Volání funkce

**Pokud chceme zavolat f-ci, není nic lehčího, než napsat její jméno a popřípadě jí předat parametry:**

 **jméno\_funkce(seznam\_parametrů);**

**…takže například:**

 **ctverec(5);**

**Tímto voláním zavoláme f-ci ctverec a předáme jí parametr, v tomto případě číslo 5. Když se podíváme nahoru (jak jsme definovali f-ci ctverec), tak zjistíme, že nám má vrátit hodnotu na druhou, tedy 25:**

**cout << ctverec(5);**

**…zobrazí hodnotu 25. A stejně můžeme zobrazit proměnné, které se inicializují až za běhu programu:**

 **int a;**

 **cin >> a; //dejme tomu, že jsme zadali hodnotu 6**

 **cout << ctverec(a); //zobrazí 36**

**Nyní si ukážeme příklad:**

**#include <iostream>**

**using namespace std;**

**void pozdrav(int); /\* prototyp: žádná návratová
 hodnota \*/**

**double kostka(double x); // prototyp: vrací hodnotu
 typu double \*/**

**int main()**

 **{**

 **pozdrav(5); /\* volání funkce => předá 5ku
 f-ci pozdrav \*/**

 **cout << "Zadejte cislo: ";**

 **double strana;**

 **cin >> strana;**

 **double prom = kostka(strana); /\* volání funkce a přiřazení hodnoty do proměnné prom \*/**

 **cout << "Krychle o velikosti strany " << strana <<" metru ma objem ";**

 **cout << prom << " metru krychlovych.\n";**

 **pozdrav(kostka(2)); // ukázka prototypové
 ochrany \*/**

 **return 0;**

 **}**

**void pozdrav(int n)**

 **{**

 **for (int i = 0; i < n; i++)**

 **cout << "Slava! ";**

 **cout << "\n";**

 **}**

**double kostka(double x)**

 **{**

 **return x \* x \* x;**

 **}**

**Co se týče prototypové ochrany, není to nic víc, než použití f-ce ve f-ci.**

### Parametry funkcí a předávání hodnotou

**C++ předává obvykle parametry hodnotou. To znamená, že se číselná hodnota parametru předá f-ci, která se přiřadí do nové proměnné.**

 **double prom = kostka(strana);**

**…a její hlavička funkce byla:**

 **double kostka(double x)**

**Když se tato f-ce vyvolá, vytváří novou proměnnou typu double, která se nazývá x a přiřadí jí hodnotu 5. Dále jako parametr můžete použít proměnnou stejného jména, jako máte již někde použité. Tedy pokud máte např.:**

 **..**

 **int main()**

 **{**

 **int lek=15;**

 **..**

 **}**

 **double druha (int lek)**

 **...**

**…tak tyto dvě proměnné jsou úplně rozdílnými proměnnými (lek), poněvadž druhá lek existuje jen uvnitř těla f-ce, tudíž se na ní nemůžete nějak dovolávat a nelze si ji tedy splést s první proměnnou lek. Je to podobné, jako se liší Albana v Kalifornii a Albana v New Yorku.**

### Násobné parametry

**Funkce mohou mít víc než jeden parametr. Ve volání f-ce pouze oddělte parametry čárkou:**

 **kostka('R', 25);**

**To předává f-ci, kterou si za chvíli definujeme, dva parametry. Když ji deklarujete, postupujte stejně:**

 **void kostka(char c, int n) // 2 parametry**

**POZOR, pokud definujeme dva parametry stejného typu, nemůžeme je vytvářet jako klasické proměnné:**

 **int a,b; //klasické proměnné**

 **void(int a, b) // nelze**

 **void(int a, int b) // takto je to správně**

**Stejně tak postupujeme i u vytváření prototypu:**

 **void(int a, int b);**

**Tak to by bylo všechno. Knihovnu cctype si necháme na později. Možná vyjde jako dodatek (jelikož jsem zjitil, že jsou dodatky důležité, tak jsou nyní již pojmenované jako normální lekce). V příští lekci (pokud to nebude ta knihovna) se budeme zabývat… funkcemi… nečekaně ;) Protože f-ce jsou velké téma! Ještě si musíme ukázat, jak pracují s řetězci a poli. A co nás čeká potom? Dvourozměrná pole, ukazatele, ukazatele na struktury, pole…**

### Úkol:

**Vytvořte program, který bude v int main() obsahovat:**

***int a;
cout << "zadejte číslo: ";
cin >> a;***

**…a nyní bude následovat menu (počet položek nechám na vás), kde vám ukáže výsledky operací s tímto číslem. Tedy uděláte několik f-cí a každá z nich bude počítat jednu operaci.**

**Příklad:**

***zadejte číslo: 5
číslo na druhou je: 25
číslo je: jednociferné***

**…a podobně. Prostě toto vám ukázalo dvě operace (dvě položky) a vy se snažte udělat menu s co nejvíce položkami, co vás napadne udělat s číslem. Každá položka bude ale počítaná ve vlastní f-ci, tedy ne přímo v int main! int main může obsahovat pouze kód uveden výše a jen volání f-cí.**

# [C++ - 7. lekce](http://programujte.com/clanek/2005041411-c-7-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Josef Vítů**](http://programujte.com/profil/20356-josef-vitu/)**13. 4. 2005       58 906×**

* **Funkce a pole – vytváření**
* **Naplnění pole**
* **Zobrazení pole a jeho ochrana pomocí const**
* **Modifikace pole**
* **Sestavení**
* **Funkce a struktury**
* **Předání a navrácení struktur**
* **Příklady**
* **Rekurze**
* **Úkol č. 7**

**Další lekce je tu, všichni se netrpělivě těšíte na ukazatele (pointery)… Ale o tom příště:) Dnes si povíme o f-cích podrobněji, mnohem podrobněji. Důsledně nastudovat!**

## Funkce a pole – vytváření

**Dosavadní f-ce byly jednoduché, používaly pouze základní typy parametrů a návratových hodnot. Ale f-ce mohou zacházet s mnohem komplikovanějšími typy – pole, struktury. Abychom vytvořili f-ci obecně, neomezíme pole velikostí, ale předáme jí jako parametr velikost pole:**

**int sum\_pole(int pole[], int n)**

**//pole = jméno pole, n = velikost**

**Tato f-ce nám může posloužit s použitím cyklu na součet prvků pole. Vypadá to jednoduše: hranaté závorky nám říkají, že se jedná o pole (libovolné velikosti). Nenechte se zmást, *pole* není polem, ale ukazatelem! Ale tím si nezatěžujte hlavu, protože zacházet s ním můžeme jako s polem. Ještě bych vám mohl ukázat kód f-ce pro zmíněný součet prvků:**

**int sum\_pole(int pole[], int n)**

**{**

 **int celkem = 0;**

 **for (int i = 0; i<n ; i++)**

 **celkem = celkem + pole[i];**

 **return celkem;**

**}**

**Pamatujte:**

**Indikaci druhu pole a počet prvků předávejte pomocí dvou samostatných parametrů:**

**void dobrepole(int pole[], int vel); // prototyp**

**void spatnepole(int pole [vel]); /\* NE - nesprávný prototyp \*/**

## Naplnění pole

**Nyní si napíšeme f-ci, která má dva parametry (jméno pole a specifikaci max. počtu položek, který se má přečíst) a navrací skutečný počet zadaných položek. Jestliže například použijete f-ci s polem o pěti prvcích, předáte 5 jako druhý parametr. Zadáte-li pouze dvě hodnoty, f-ce vrátí 2. K načtení po sobě jdoucích hodnot můžeme použít cyklus. Jak ale ukončíme vstup (když nechceme zadat již další hodnotu)? Uděláme to tak, že pokud uživatel zadá zápornou hodnotu, bude to znamenat signál pro ukončení.**

**int pocpole(double pole[], int limit)**

**{**

 **double temp;**

 **int i;**

 **for (i=0 ; i<limit ; i++)**

 **{**

 **cout << "zadejte hodnotu #" << i+1 << ":";**

 **cin >> temp;**

 **if (temp < 0) //signál pro ukončení**

 **break; // vysvětlení najdete níže u cyklu CASE**

 **pole[i] = temp;**

 **}**

 **return i;**

**}**

**Pokud uživatel zavede nezápornou hodnotu, přiřadí se do pole, jinak cyklus končí. Pokud přiřadí pouze platné hodnoty, cyklus končí po naplnění maximálního počtu prvků *i*. F-ce poté tuto hodnotu navrací.**

## Zobrazení pole a jeho ochrana pomocí const

**Pokud účelem f-ce není měnit data, která se jí předávají, měli byste to zabezpečit. C++ automaticky zabezpečuje obvyklé parametry, protože je předává hodnotou a f-ce pracuje s jeho kopií. F-ce pracující s polem pracují s originálem. Abychom zabránili změně parametrů obsahů pole, můžeme použít již známé *const*:**

 **void ukazpole(const double novepole[], int n);**

**Deklarace stanoví, že se odkazuje na konstantní data. Můžete použít *novepole[0]*, ale nemůžete je změnit. Pozor! To neznamená, že musí být původní pole konstantou, ale že nemůžete použít *novepole* ve f-ci ukazpole na změnu dat. F-ce tedy zachází s polem jako s daty pouze pro čtení. Nezapomeňte, nyní pracujeme se ZOBRAZENÍM pole:**

**void ukazpole(const double novepole[], int n)**

**{**

 **for (int i = 0; i<n ; i++)**

 **{**

 **cout << "Majetek #" << i+1 << ": Kc: ";**

 **cout << novepole[i] << "\n";**

 **}**

**}**

## Modifikace pole

**Další operace pro naše pole je násobení (např.) každého prvku pole určitým koeficientem. Prostě modifikace prvků pole. Potřebujeme k tomu tři parametry: koeficient, pole a počet prvků pole. Nejsou potřeba žádné návratové hodnoty.**

 **void zmenapole(double r, double pole[], int n)**

**{**

 **for (int i = 0; i<n ; i++)**

 **pole[i] \*= r; // neboli**

 **pole[i] = pole[i] \* r**

**}**

**Protože se od f-ce předpokládá, že bude měnit hodnoty, nepoužívejte *const*, když deklarujete pole.**

## Sestavení

**Nyní příklad, který to vše sumarizuje (všimněte si, že ve f-ci *main* je čím dál méně příkazů). Následující příklad uloží do pole o pěti prvcích pět hodnot a ty následně zvětší o zadaný koeficient:**

**#include <iostream>**

**using namespace std;**

**const int Max = 5;**

 **// prototypy funkcí**

**int fill\_array(double ar[], int limit);**

**void ukaz\_array(const double ar[], int n); /\* nemění data \*/**

**void revalue(double r, double ar[], int n);**

**int main()**

**{**

 **double properties[Max];**

 **int size = fill\_array(properties, Max);**

 **ukaz\_array(properties, size);**

 **cout << "Zadejte koeficient prehodnoceni: ";**

 **double factor;**

 **cin >> factor;**

 **revalue(factor, properties, size);**

 **ukaz\_array(properties, size);**

 **cout << "Hotovo.\n";**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

**int fill\_array(double ar[], int limit) // naplnění pole**

**{**

 **double temp;**

 **int i;**

 **for (i = 0; i < limit; i++)**

 **{**

 **cout << "Zadejte hodnotu c." << (i + 1) << ": ";**

 **cin >> temp;**

 **if (!cin) // neplatný vstup**

 **{**

 **cin.clear();**

 **while (cin.get() != '\n')**

 **continue;**

 **cout << "Neplatny vstup, zadavani ukonceno.\n";**

 **break;**

 **}**

 **else if (temp < 0) // signál ukončení zadávání**

 **break;**

 **ar[i] = temp;**

 **}**

 **return i;**

**}**

**// následující funkce může používat, ale ne měnit**

**// pole, jehož adresa je ar**

**void ukaz\_array(const double ar[], int n) // čtení pole**

**{**

 **for (int i = 0; i < n; i++)**

 **{**

 **cout << "Majetek c." << (i + 1) << ": ";**

 **cout << ar[i] << " Kc\n";**

 **}**

**}**

**// násobí každý prvek pole ar[] koeficientem r**

**void revalue(double r, double ar[], int n) // úprava všech prvků pole**

**{**

 **for (int i = 0; i < n; i++)**

 **ar[i] \*= r;**

**}**

**Výsledek:**

***Zadejte hodnotu c.2: 80000
Zadejte hodnotu c.3: 222000
Zadejte hodnotu c.4: 240000
Zadejte hodnotu c.5: 118000
Majetek c.1: 100000 Kc
Majetek c.2: 80000 Kc
Majetek c.3: 222000 Kc
Majetek c.4: 240000 Kc
Majetek c.5: 118000 Kc
Zadejte koeficient prehodnoceni: 1.10
Majetek c.1: 110000 Kc
Majetek c.2: 88000 Kc
Majetek c.3: 244200 Kc
Majetek c.4: 264000 Kc
Majetek c.5: 129800 Kc
Hotovo.***

**…nebo v případě špatného vstupu:**

***Zadejte hodnotu c.1: 200000
Zadejte hodnotu c.2: 84000
Zadejte hodnotu c.3: 160000
Zadejte hodnotu c.4: -2
Majetek c.1: 200000 Kc
Majetek c.2: 84000 Kc
Majetek c.3: 222000 Kc
Zadejte koeficient prehodnoceni: 1.20
Majetek c.1: 240000 Kc
Majetek c.2: 100800 Kc
Majetek c.3: 192000 Kc
Hotovo.***

## Funkce a struktury

**Napsat f-ci pro strukturu je jednodušší než pro pole. Struktury můžete předat hodnotou – funkce pracuje s kopií původní struktury. Funkce může také vrátit strukturu. Nejlepší je předávat je jako parametry a, pokud je to nezbytné, použít jako návratové hodnoty. Pokud zvolíme předávání struktury hodnotou, je zde jedna nevýhoda. Pokud je struktura příliš velká, úsilí spojené s vytvořením kopie zvýší požadavky na paměť a zpomalí systém. Proto mnoho programátorů v C dává přednost předání struktury adresou a potom pro přístup k jejímu obsahu použít ukazatel. C++ poskytuje třetí alternativu – předání odkazem, to probereme jindy.**

## Předání a navrácení struktur

**Předání struktur hodnotou má největší smysl. Předpokládejme, že chceme spočítat časy cestování. Definování struktury je jednoduché:**

**struct cestovani**

**{**

 **int hodiny;**

 **int minuty;**

**};**

**Dále vytvoříme prototyp f-ce *sum*, která navrací součet dvou takových struktur. Návratová hodnota by měla být typu *cestovani*, a tudíž by měla mít dva parametry:**

 **cestovani sum(cestovani t1, cestovani t2);**

**Abychom sečetli dva časy, nejprve sečteme minutové členy, díky celočíselnému dělení 60-ti získáme počet hodin a operátor modulo % poskytne počet zbývajících minut:**

**#include <iostream>**

**using namespace std;**

**struct travel\_time**

**{**

 **int hours;**

 **int mins;**

**};**

**const int Mins\_per\_hr = 60;**

**travel\_time sum(travel\_time t1, travel\_time t2);**

**void ukaz\_time(travel\_time t);**

**int main()**

**{**

 **travel\_time day1 = {5, 45}; // 5 hodin, 45 min**

 **travel\_time day2 = {4, 55}; // 4 hodiny, 55 min**

 **travel\_time trip = sum(day1, day2);**

 **cout << "Celkem za dva dny: ";**

 **ukaz\_time(trip);**

 **travel\_time day3 = {4, 32};**

 **cout << "Celkem za tri dny: ";**

 **ukaz\_time(sum(trip, day3));**

 **return 0;**

**}**

**travel\_time sum(travel\_time t1, travel\_time t2)**

**{**

 **travel\_time total;**

 **total.mins = (t1.mins + t2.mins) % Mins\_per\_hr;**

 **total.hours = t1.hours + t2.hours + (t1.mins + t2.mins) / Mins\_per\_hr;**

 **return total;**

**}**

**void ukaz\_time(travel\_time t)**

**{**

 **cout << t.hours << " hodin, " << t.mins << " minut\n";**

**}**

**Zde *travel\_time* funguje jako standardní jméno typu; můžete ho použít na deklaraci proměnných, funkčních návratových typů a typů parametru f-ce.**

***Celkem za dva dny: 10 hodin, 40 minut
Celkem za tri dny: 15 hodin, 12 minut***

## Příklady

**Další příklad se zabývá prostorem, místo času.**

****

**Předpokládejme, že chceme popsat pozici bodu na mapě vzhledem k nějakému počátku. Tradičně X pro horizontální posunutí a Y pro vertikální. Jeden způsob je stanovit posunutí jak horiz., tak vertik. bodu od počátku.**

**struct sourad**

**{**

 **double x;**

 **double y;**

**};**

**Druhý způsob popisu pozice bodu je stanovit, jak daleko je od počátku a ve kterém je směru (např. 40 stupňů severně od počátku):**

****

**struct uhel**

**{**

 **double pocat;**

 **double uhel;**

**};**

**F-ce zobrazující obsah struktury *bod2*. Matematické knihovny v C++ počítají s úhly v radiánech, ale pro zobrazovací údaje je konvertujeme na stupně (násobením 180/(pi), což je přibližně 57.29577951). Zde je:**

**void ukaz\_sour(uhel prvni) // f-ce se jmenuje *ukaz\_sour* a jako parametr předáváme strukturu *prvni***

**{**

 **const double rad\_na\_stupne = 57.29577951;**

 **cout << "vzdalenost = " << prvni.pocat;**

 **cout << ",uhel = " << prvni.uhel \* rad\_na\_stupne;**

 **cout << " stupnu\n";**

**}**

**Všimněte si, že formální proměnná je typu *uhel*. Když předáte f-ci strukturu *uhel*, obsah se kopíruje do struktury *prvni* a f-ce potom pracuje s touto kopií. Protože je *prvni* struktura, f-ce používá k identifikaci členů struktury operátor příslušnosti . (tečku). Podívejme se dál a napišme f-ci, která konvertuje pravoúhlé souřadnice na polární (v našich strukturách *sourad*na *uhel*). Pro volání *sqrt* potřebujeme knihovnu *cmath* (na starších systémech překvapivě math.h).**

**pocat = sqrt (x \* x + y \* y) /\*neboli pocat = odmocnina z (x nadruhou + y na druhou) \*/**

 **// pythagorova věta**

**F-ce *atan2()* z matematické knihovny počítá z hodnot *x* a *y* úhel:**

**uhel = atan2(y,x)**

**…(existuje f-ce *atan()*, ale ta nerozlišuje mezi úhly 180° odděleně).**

**Jsou dány vzorce, tady je f-ce, která konvertuje pravoúhlé souřadnice na polární:**

**uhel sourad\_na\_uhel(sourad xypos) // typ uhel**

**{**

 **uhel prenos;**

 **prenos.pocat = sqrt(xypos.x \* xypos.x + xypos.y + xypos.y);**

 **prenos.uhel = atan2(xypos.y, xypos.x);**

 **return prenos;**

**}**

**No a nyní zbytek programu:**

**#include <iostream>**

**#include <cmath>**

**using namespace std;**

**// šablony struktur struct uhel**

**{**

 **double pocat; // vzdálenost od počátku**

 **double uhel; // směr od počátku**

**};**

**struct sourad**

**{**

 **double x; // horizontální vzdálenost od počátku**

 **double y; // vertikální vzdálenost od počátku**

**};**

**// prototypy**

**uhel sourad\_to\_uhel(sourad xypos);**

**void ukaz\_uhel(uhel prvni);**

**int main()**

**{**

 **sourad rplace;**

 **uhel pplace;**

 **cout << "Zadejte hodnoty x a y: ";**

 **while (cin >> rplace.x >> rplace.y) // úhledné použití cin**

 **{**

 **pplace = sourad\_to\_uhel(rplace);**

 **ukaz\_uhel(pplace);**

 **cout << "Dalsi dve cisla (q na ukoceni): ";**

 **}**

 **cout << "Hotovo!\n";**

 **return 0;**

**}**

**// konvertuje pravoúhlé souřadnice na polární**

**uhel sourad\_to\_uhel(sourad xypos)**

**{**

 **uhel pristup;**

 **pristup.pocat = sqrt( xypos.x \* xypos.x + xypos.y \* xypos.y);**

 **pristup.uhel = atan2(xypos.y, xypos.x);**

 **return pristup; // vrací strukturu uhel**

**}**

**// zobrazení polárních souřadnic spolu s konverzí úhlu na stupně**

**void ukaz\_uhel (uhel prvni)**

**{**

 **const double Rad\_to\_deg = 57.29577951;**

 **cout << "vzdalenost = " << prvni.pocat;**

 **cout << ", uhel = " << prvni.uhel \* Rad\_to\_deg;**

 **cout << " stupnu\n";**

**}**

***Zadejte hodnoty x a y: 30 40
vzdalenost = 50, uhel = 53.1301 stupnu
Dalsi dve cisla (q na ukoceni): -100 100
vzdalenost = 141.421, uhel = 135 stupnu
Dalsi dve cisla (q na ukoceni): q
Hotovo!***

**Poznámky k programu:**

**(cin >> rplace.x >> rplace.y)**

**…je to samé jako:**

**cin >> rplace.x;**

**cin >> rplace.y;**

## Rekurze

**F-ce v C++ mají zajímavou vlastnost – můžou volat samy sebe (kromě f-ce *main()*). Rekurze je důležitá například pro programování umělé inteligence. Když funkce rekurzním voláním volá sama sebe, pak nově vyvolaná f-ce volá opět sama sebe a tak dále, dokud kód neobsahuje něco na ukončení tohoto řetězce volání. Obvykle se dělá rekurzivní volání příkazu *if*:**

**void nazev\_fce(seznamparametrů)**

**{**

 **příkazy1**

 **if (test)**

 **nazev\_fce(příkazy) // tady se volá f-ce sama sebe**

 **příkazy2**

**Díky štěstí či prozíravosti se test nakonec stane nepravdivý a řetěz volání se přeruší. Jestliže *nazev\_fce* podstoupí pět rekurzivních volání, nejprve se provede sekce *příkazy1* 5× v pořadí, ve kterém se volaly a potom se provede 5× sekce *příkazy2* v opačném pořadí, než byly f-ce volány. Po proběhnutí pěti úrovní rekurze potom program musí vycouvat přes pět stejných úrovní.**

**#include <iostream>**

**using namespace std;**

**void odpocet(int n);**

**int main()**

**{**

 **odpocet(4); // volá rekurzivní funkci**

 **return 0;**

**}**

**void odpocet(int n)**

**{**

 **cout << "Pocitani smerem dolu... " << n << "\n";**

 **if (n > 0)**

 **odpocet(n-1); // funkce volá sama sebe**

 **cout << n << ": Kaboom!\n";**

**}**

***Pocitani smerem dolu ... 4 " úroveň 1 -- začátek přidávání úrovní rekurze
Pocitani smerem dolu ... 3 " úroveň 2
Pocitani smerem dolu ... 2 " úroveň 3
Pocitani smerem dolu ... 1 " úroveň 4
Pocitani smerem dolu ... 0 " úroveň 5 -- poslední rekurzivní volání
0: Kaboom! " úroveň 5 -- začátek návratu přes posloupnost volání
1: Kaboom! " úroveň 4
2: Kaboom! " úroveň 3
3: Kaboom! " úroveň 2
4: Kaboom! " úroveň 1***

**Víc proniknout do f-cí nemůžeme, poněvadž zbytek f-cí a jejich možností se váže na ukazatele. Ty probereme zřejmě již v další lekci. Ale možná přijde taková vsuvka v podobě zpětného rozšíření probraného učiva, které je neméně potřebné.**

## Úkol

**Vytvořte katalog: Uživatel bude zadávat ceny položek (tedy program se zeptá na první položku, druhou, atd.) a až nebude již chtít uživatel zadat další cenu pro další položku, bude tam něco na ukončení zadávání (např. „q“), vymyslete nějaký zajímavý způsob. Kompletně bude program obsahovat češtinu a znaky ASCII (nějaké orámování, podtrhnutí…). Poté program vypíše všechny položky a jejich ceny a dá uživateli možnost editovat kteroukoliv položku (program se zeptá, zda chce uživatel upravit nějakou položku, pokud ano, zadá číslo položky a napíše novou cenu). Poté se program zeptá, zda chce ke všem položkám připočíst DPH (19 %) pokud ano, všechny ceny se upraví samy dle DPH, pokud ne, ceny zůstanou. Poté se provede znovu výpis položek a jejich cen. Následně program vypočítá celkovou cenu všech položek. Nakonec, poté co uživatel odsouhlasí konečnou cenu, se spustí odpočítávání ukončení programu (max. 5).**

[**C++ - 8. lekce**](http://programujte.com/clanek/2005041412-c-8-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Josef Vítů**](http://programujte.com/profil/20356-josef-vitu/)**13. 4. 2005       50 325×**

* **Dodatek k cyklu FOR**
* **Operátor „,“ (čárka)**
* **Kopírování a spojování řetězců, jejich velikost a délka**
* **Alias jména**
* **Časová prodleva**
* **Dvourozměrné pole**
* **Operátor „?“**
* **Příkaz SWITCH**
* **Příkazy break a continue**
* **Příkaz goto**
* **Knihovna cctype**
* **Symbolické konstanty climits**
* **Chyby, které možná děláte**
* **Vícesouborové projekty**
* **Úkol**

**Dnes si dáme odpočinek od f-cí a podobných témat. Tato lekce bude víc odpočinková, ale o to víc zábavnější! Poučíme se o věcech, na které jsem zapomněl upozornit, nebo je odložil. Rozhodně jsou to důležité příkazy, se kterými obohatíte každý program.**

## Dodatek k cyklu FOR

**Klasicky vypadá cyklus *for* takto:**

**for (i=0 ; i<5 ; i++)**

 **příkazy;**

**…ale pokud chceme vynechat některou z podmínek (například proto, že již před začátkem cyklu je hodnota někde stanovená), není nic snazšího, než danou podmínku vynechat (ale středník tam přesto musí být!):**

**for ( ; i<5 ; i++)**

 **příkazy;**

## Operátor „ , “ (čárka)

**Operátor čárka nám dovoluje provést více podmínek, změn, deklarací atp.:**

**for (j=5 , i=4 ; j<i ; i-- , j++)**

 **příkazy;**

**Myslím, že je to dostatečně pochopitelné. Operátor čárka:**

* **má nejnižší prioritu**
* **čte zleva doprava**
* **když má na výběr, použije druhý výraz**

**int kocka = 17, 145; // nastaví proměnnou kočka na 17 (čísla 145 si nevšímá)**

**int kocka = (17, 145); // nastaví proměnnou kočka na 145**

## Kopírování a spojování řetězců, jejich velikost a délka

**V celém bloku 8.12 budeme muset na začátek programu vložit tuto direktivu:**

**#include <cstring>**

**Kopírování jednoho řetězce do druhého:**

**strcpy(retezec1, retezec2); //retezec1 je příjemcem, z retezec2 se kopíruje**

**Z toho plyne, že obsah řetězce retezec2 nakopíruje svůj obsah do řetězce retezec1:**

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**int main()**

**{**

 **char dotaz[20];**

 **cin >> dotaz;**

 **char dotaz1[20];**

 **strcpy(dotaz1, dotaz);**

 **cout << dotaz << endl;**

 **cout << dotaz1;**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

**Pozor! U všech těchto funkcí (*strxxx*) musíme zabezpečit místo pro kopírovaný či spojovaný řetězec! Kompilátor nekontroluje, zda se obsah řetězce „vleze“ do cílového řetězce – zda je tedy místo pro něj.**

**Ovšem může nastat situace, že chceme zkopírovat pouze několik znaků. V tom případě použijeme podobnou f-ci:**

**strncpy(dotaz1, dotaz, 20);**

**…způsobí, že se do řetězce *dotaz1* zkopíruje obsah řetězce *dotaz*, ale nejvýše 20 znaků!**

**Příklad:**

**#include**

**<iostream>**

**#include <cstring>**

**using namespace std;**

**int main()**

**{**

 **cout << "zadejte text ke zkopirovani (min. 5 znaku)";**

 **char dotaz[20];**

 **cin >> dotaz;**

 **char dotaz1[20];**

 **strncpy(dotaz1, dotaz, 3);**

 **cout << dotaz << endl;**

 **cout << dotaz1;**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

**Vezme řetězec *dotaz* a nakopíruje jeho první 3 znaky do proměnné *dotaz1*.**

**Spojování řetězců**

**Spojování probíhá podobně, jako f-ce na kopírování:**

**strcat(retezec\_cilovy, retezec\_zdrojovy);**

**Řetězec *retezec\_cilovy* obsahuje nyní jak obsah sama sebe, tak i obsah řetězce *retezec\_zdrojovy*:**

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**int main()**

**{**

 **char dotaz[50];**

 **cin >> dotaz;**

 **char dotaz1[20];**

 **cin >> dotaz1;**

 **strcat(dotaz, dotaz1);**

 **cout << dotaz << endl;**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

Zjištění délky řetězce

 **Na zjištění délky řetězce nám poslouží příkaz:**

**strlen(jmeno\_promenne);**

**Ten nám zjistí počet znaků, které obsahuje daná proměnná.**

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**int main()**

**{**

 **char dotaz[50];**

 **cout << "zadej slovo :";**

 **cin >> dotaz;**

 **cout << "\nPromenna obsahuje " <<**

 **strlen(dotaz) << " znaku.";**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

### Zjištění velikosti řetězce

**Slouží nám ke zjištění velikosti proměnné v bajtech.**

**sizeof nazev\_promenne;**

**Příklad:**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **char dotaz[50];**

 **cout << "zadej slovo :";**

 **cin >> dotaz;**

 **cout << "\nPromenna ma velikost " << sizeof dotaz << " bajtu.";**

 **cin.get();**

 **cin.get();**

 **return 0;**

**}**

## Alias jména

**Vytváří stejný název proměnné, ale jiného typu. Pokud tedy chceme použít 2× stejný název, ale pokaždé jiný typ, použijeme:**

**typedef jméno\_typu alias\_jmeno;**

**…takže například pokud máme proměnnou typu *int* a chceme tutéž proměnnou typu *long*:**

**int slovo;**

**typedef long slovo;**

## Časová prodleva

**Pokud chceme pracovat s časem, slouží k tomu několik dalších příkazů. Ale k jejich zpřístupnění potřebujeme další direktivu:**

**#include <ctime>**

**Nejprve výpis:**

**clock( ); // tento příkaz označuje aktuální čas, když je volán, nastaví se na nynější čas**

**clock\_t // je to typ (stejně jako int, long...) proměnné orientované na uchování času**

**CLOCKS\_PER\_SEC // musí být psáno velkými písmeny; díky tomuto systém definuje konstantu = počet systémových jednotek za sekundu (neboli synchronizuje rychlost procesoru a přepočítává z ní sekundy)**

**Starší hry právě byly někdy nehratelné na nynějších strojích, poněvadž neobsahovaly tento přepočet, a byly tak závislé na času procesoru (počtu operací). Čím silnější PC, tím rychlejší hra (uspěchanější).**

**clock\_t doba = 5 \* CLOCKS\_PER\_SEC;**

**//clock\_t - typ**

**//doba - proměnná**

**//5 - počet sekund**

**//CLOCKS\_PER\_SEC - převádí rychlost procesoru na sekundy**

**clock\_t start = clock();**

**//clock\_t - typ**

**//start - proměnná**

**//clock() - nynější čas**

**Poté si již stačí určit podmínku:**

**while(clock() - start < doba)**

 **; // středník místo těla cyklu**

***clock() - start* je vlastně *čas teď - čas při vytvoření proměnné start***

**Jak plyne právě z této řádky, *start* se musí inicializovat těsně před cyklem, jinak se krátí čas a může se stát, že na básněnou časovou prodlevu ani nedojde.**

**Příklad:**

**#include <iostream>**

**#include <ctime>**

**using namespace std;**

**int main()**

**{**

 **clock\_t doba = 2\* CLOCKS\_PER\_SEC;**

 **clock\_t start = clock();**

 **for (int i=0; i< 20000; i++)**

 **cout << "ahoj";**

 **while (clock() - start < doba)**

 **;**

**return 0;**

**}**

**Na přesně dvousekundovou časovou prodlevu nikdy nedojde, protože provedení cyklu zabere určitý čas. Pokud bychom tedy chtěli onu dvousekundovou prodlevu, náprava je jednoduchá:**

**#include <iostream>**

**#include <ctime>**

**using namespace std;**

**int main()**

**{**

 **clock\_t doba = 2\* CLOCKS\_PER\_SEC;**

 **for (int i=0; i< 20000; i++)**

 **cout << "ahoj";**

 **clock\_t start = clock();**

 **while (clock() - start < doba)**

 **;**

 **return 0;**

**}**

## Dvourozměrné pole

**…neboli 2D pole. Je to to samé, jako u klasických polí, jen delší.**

**int pole[4] = { 1, 2, 3, 4 }; //jednorozměrné pole (klasické)**

**int pole[4][5]; //dvou rozměrné pole, má 4 pole, každé 5 proměnných**

### Vytváření 2D pole

**int pole[4][5];**

**Vytvoří 2D pole, které obsahuje 4 klasické pole a každé klasické pole má 4 prvky.**

### Inicializace 2D pole

**int pole[0] = { 1, 2, 6, 4, 5 };**

**int pole[1] = { 8, 2, 6, 5, 9 };**

**...**

**…nebo…**

**int pole[4][5] =**

 **{**

 **{ 1, 2, 6, 4, 5 }, // hodnoty pro pole[0]**

 **{ 8, 2, 6, 5, 9 }, // hodnoty pro pole[1]**

**...**

 **};**

### Volání prvku

**pole[1][3];**

* **pole - proměnná**
* **[1] - číslo pole**
* **[3] - číslo prvku**

## Operátor „?“

**Obecně: *výraz1 ? výraz2 : výraz3***

 **int c = a > b ? a : b;**

**Je-li *a>b*, pak přiřaď proměnné *c* hodnotu *a*, pokud ne (tedy *a<b*), pak přiřaď hodnotu *b*. Pokud je tedy podmínka pravdivá, proměnná *c* dostane hodnotu proměnné *a*, pokud podmínka pravdivá není, tak dostane hodnotu proměnné *b*.**

**Příklad :**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **int a, b;**

 **cout << "Zadejte dve cela cisla: ";**

 **cin >> a >> b;**

 **cout << "Vetsi z cisel " << a << " a " << b;**

 **int c = a > b ? a : b; // c = a jestliže a > b, jinak c = b**

 **cout << " je " << c << "\n";**

 **return 0;**

**}**

## Příkaz SWITCH

**Pokud chcete vytvořit například menu, můžete ho udělat pomocí sekvencí *if else if else*, ale příkaz switch pracuje mnohem snadněji.**

**Obecně:**

**switch (celočíselný\_výraz)**

**{**

 **case 1 : příkaz(y)**

 **case 2 : příkaz(y)**

 **...**

 **default : příkaz(y)**

**}**

**Příkaz *switch* v C++ funguje jako směrovací zařízení, které říká počítači, jaký řádek programového kódu má dále provést. Při čtení *switch* skáče program na řádek, který je označen hodnotou odpovídající hodnotě *celočíselný\_výraz*. Například má-li tento *celočíselný\_výraz* hodnotu 4, pak jde na řádek, který má číslo *4*. Hodnota *celočíselný\_výraz* musí být výrazem, který produkuje číselnou hodnotu. Také označení (1, 2…) musí být konstantním výrazem. Nejčastěji se používají jednoduché konstanty typu int nebo char (1,2,3…, e,f,g,h…) nebo emulátory. Jestliže *celočíselný\_výraz* neodpovídá žádnému označení (1,2,3…), přejde na řádek *default*, které ale není povinné. Vynecháte-li ho a nenajde-li žádnou shodu, jednoduše opustí příkaz switch a pokračuje dále. Pokud ale *default* použijete, automaticky za vás opraví možnosti, jako když uživatel zadá číslo, které není nabízeno. V tom případě se switch opakuje.**

***Switch* se velmi liší od podobných příkazů v jiných jazycích (např. Pascal). Jakmile přeskočí na určitý řádek ve switch, potom provádí všechny příkazy, které za ním následují, dokud ho nějak nepřesměrujete. Vykonávání se automaticky NEZASTAVÍ na dalším *case*. Abyste zastavili na konci určité skupiny příkazů, musíte použít příkaz *break*. To způsobí opuštění příkazu switch.**

**Ukážeme si příklad jednoduché implementace *break* do *switch*:**

**#include <iostream>**

**using namespace std;**

**void showmenu(); // prototypy funkcí**

**void report();**

**void comfort();**

**int main()**

**{**

 **showmenu();**

 **int choice;**

 **cin >> choice;**

 **while (choice != 5)**

 **{**

 **switch(choice)**

 **{**

 **case 1 : cout << "\a\n";**

 **break;**

 **case 2 : report();**

 **break;**

 **case 3 : cout << "Vedouci byl pritomen cely den.\n";**

 **break;**

 **case 4 : comfort();**

 **break;**

 **default : cout << "To neni volba.\n";**

 **}**

 **showmenu();**

 **cin >> choice;**

 **}**

 **cout << "Sbohem!\n";**

 **return 0;**

**}**

**void showmenu()**

**{**

 **cout << "Prosim, zadejte 1, 2, 3, 4 nebo 5:\n"**

 **"1) poplach 2) zprava\n"**

 **"3) vymluva 4) uklidneni\n"**

 **"5) ukonceni\n";**

**}**

**void report()**

**{**

 **cout << "To byl vytecny tyden pro obchod.\n"**

 **"Trzby vzrostly o 120%. Vydaje poklesly o 35%.\n";**

**}**

**void comfort()**

**{**

 **cout << "Vasi zamestnanci si mysli, ze jste nejlepsi vykonny reditel\n"**

 **"v prumyslu. Spravni rada si mysli,\n"**

 **"ze jste nejlepsi vykonny reditel v prumyslu.\n";**

**}**

**Výpis:**

***Prosim, zadejte 1, 2, 3, 4 nebo 5:
1) poplach 2) zprava
3) vymluva 4) uklidneni
5) ukonceni
4
Vasi zamestnanci si mysli, ze jste nejlepsi vykonny reditel
v prumyslu. Spravni rada si mysli,
ze jste nejlepsi vykonny reditel v prumyslu.
Prosim, zadejte 1, 2, 3, 4 nebo 5:
1) poplach 2) zprava
3) vymluva 4) uklidneni
5) ukonceni
2
To byl vytecny tyden pro obchod.
Trzby vzrostly o 120%. Vydaje poklesly o 35%.
Prosim, zadejte 1, 2, 3, 4 nebo 5:
1) poplach 2) zprava
3) vymluva 4) uklidneni
5) ukonceni
6
To neni volba.
Prosim, zadejte 1, 2, 3, 4 nebo 5:
1) poplach 2) zprava
3) vymluva 4) uklidneni
5) ukonceni
5
Sbohem!***

**Cyklus *while* se ukončí, jakmile uživatel zavede 5. Zavedení čísel 1-4 aktivuje odpovídající volbu ze seznamu *switch* a zavedení čísla 6 spustí standardní příkazy (následující za příkazem switch). Jak bylo poznamenáno výše, program potřebuje příkazy *break*, aby omezil vykonávání programu na určitou část příkazu switch. Abyste viděli, že to tak opravdu funguje, odstraňte z předchozího programu příkazy *break* a sledujte. Zjistíte, že při zadání například čísla 2 program provede všechny operace začleněné k 2, 3, 4 a standardnímu *case default*. Ptáte se, proč to tak dělá? Takový způsob vykonávání může být totiž užitečný. Ukážeme si příklad, kde:**

**char vyber;**

**cin >> vyber;**

**while (vyber != 'Q' && vyber != 'q' )**

**{**

 **switch(vyber)**

 **{**

 **case 'a' :**

 **case 'A': cout << "\a\n";**

 **break;**

 **case 'r' :**

 **case 'R': cout << "ok";**

**...**

 **}**

**}**

**Protože bezprostředně za *case 'a'* neexistuje žádný *break*, vykonávání programu přechází na následující řádek → příkaz *case 'A'*. Jak vidíte, můžete takto ošetřit používání velkých a malých písmen a další věci. Nebo také pro více voleb udělat jeden příkaz.**

***Switch* tedy požíváme na větší větvení (3 a více položek), kdežto u menších použijeme *if else*. Dále také není *switch* přizpůsoben na zacházení s rozsahy. Každé označení case příkazu *switch* musí být jediná hodnota. Pokud tedy chceme rozsah, použijeme *if else*:**

**if (vek > 17 && age < 35)**

 **index = 0;**

 **else if (age >= 35 && age < 50)**

 **index = 1;**

 **else if (age >= 50 && age < 65)**

 **index = 2;**

 **else**

 **index = 3;**

## Příkazy break a continue

**Příkazy *break* a *continue* umožňují programu přeskočit část programového kódu. *Break* můžeme použít v příkazu *switch* nebo v kterémkoliv cyklu. *Break* způsobí, že se okamžitě opouští příkaz *switch* nebo cyklus a pokračuje se na příkazech následujících za cyklem či *switch*. *Continue* se používá v cyklech a způsobuje, že se přeskočí zbytek těla cyklu a znova ho celý opakuje.**

**#include <iostream>**

**using namespace std;**

**const int ArSize = 80;**

**int main()**

**{**

 **char line[ArSize];**

 **int spaces = 0;**

 **cout << "Zadejte textovy radek:\n";**

 **cin.get(line, ArSize);**

 **for (int i = 0; line[i] != '\0'; i++)**

 **{**

 **cout << line[i]; // zobrazi znak**

 **if (line[i] == '.') // ukončení, když je tečka**

 **break;**

 **if (line[i] != ' ') // přeskočení zbytku příkazů v cyklu**

 **continue;**

 **spaces++;**

 **}**

 **cout << "\n" << spaces << " mezer\n";**

 **return 0;**

**}**

## Příkaz goto

**C++, podobně jako C, basic a mnoho dalších jazyků má příkaz *goto*.**

**goto paris**

**…znamená přeskočení na místo, které nese *paris*:**

**char(ch);**

**cin >> ch;**

**if (ch == 'P')**

 **goto paris;**

**cout << ...**

**...**

**paris: cout << "Prave jsi udelal skok.";**

**Neznamená to tedy nic jiného, než skákání na určitá místa v programu. Ve většině případů je ale použití příkazu *goto* nevhodné. Měli byste raději použít strukturované řídící příkazy jako *if else*, *switch*, *continue*…**

## Knihovna cctype

**C++ zdědil z C příhodný balík f-cí, které souvisí se znaky, jejichž proto prototypy jsou v hlavičkovém souboru *cctype*, které zjednodušují takové úkoly, jako je určení, zda je znak velkým písmenem, číslicí či interpunkčním znaménkem. Použití těchto f-cí je vhodnější než použití operátorů A a NEBO. Například tady vidíte, jak byste mohli použít A a NEBO na otestování, zda je znak *ch* alfabetickým znakem:**

**if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))**

**Porovnejte to s použitím isalpha():**

**if (isalpha(ch))**

**Nejenže je to kratší, ale taky obecnější.**

| **Jméno funkce** | **Návratová hodnota** |
| --- | --- |
| **isalnum()** | **True, je-li parametr alfanumerický, písmeno, nebo číslice** |
| **isalpha()** | **True, je-li parametr alfabetický** |
| **iscntrl()** | **True, je-li parametr řídící znak** |
| **isdigit()** | **True, je-li parametr desítková číslice (0-9)** |
| **isgraph()** | **True, je-li parametr libovolný tisknutelný znak, než mezera** |
| **islower()** | **True, je-li parametr malé písmeno** |
| **isprint()** | **True, je-li parametr libovolný tisknutelný znak včetně mezery** |
| **ispunct()** | **True, je-li parametr interpunkční znak** |
| **isspace()** | **True, je-li parametr standardním oddělovacím znakem, to jest mezera, posun formuláře, nový řádek, návrat vozu, horizontální a vertikální tabulátor** |
| **isupper()** | **True, je-li parametr velké písmeno** |
| **isxdigit()** | **True, je-li parametr hexadecimální číselný znak (0-9, a-f, A-F)** |
| **tolower()** | **Je-li parametrem velké písmeno, navrací malé písmeno onoho znaku jinak navrací nezměněný parametr** |
| **toupper()** | **Je-li parametrem malé písmeno, navrací velké písmeno onoho znaku jinak navrací nezměněný parametr** |

**Příklad:**

**#include <iostream>**

**#include <cctype> // prototypy znakových funkcí**

**using namespace std;**

**int main()**

**{**

 **cout << "Zadejte text pro analyzu a znakem @"**

 **" ukoncete vstup.\n";**

 **char ch;**

 **int whitespace = 0;**

 **int digits = 0;**

 **int chars = 0;**

 **int punct = 0;**

 **int others = 0;**

 **cin.get(ch); // získá první znak**

 **while(ch != '@') // test ukončení vstupu**

 **{**

 **if(isalpha(ch)) // je to alfabetický znak?**

 **chars++;**

 **else if(isspace(ch)) // je to bílý znak?**

 **whitespace++;**

 **else if(isdigit(ch)) // je to číslice?**

 **digits++;**

 **else if(ispunct(ch)) // je to znak interpunkce?**

 **punct++;**

 **else**

 **others++;**

 **cin.get(ch); // získá další znak**

 **}**

 **cout << chars << " pismen, "**

 **<< whitespace << " bilych znaku, "**

 **<< digits << " cislic, "**

 **<< punct << " znaku interpunkce, "**

 **<< others << " ostatnich.\n";**

 **return 0;**

**}**

## Symbolické konstanty climits

**Hlavičkový soubor *climits* definuje symbolické konstanty na zachycení typů mezí. Např. *INT\_MAX* představuje největší hodnotu, kterou může typ int obsahovat.**

|  |  |
| --- | --- |
| **CHAR\_BIT** | **počet bitů v char** |
| **CHAR\_MAX** | **maximální hodnota char** |
| **CHAR\_MIN** | **minimální hodnota char** |
| **SCHAR\_MAX** | **maximální hodnota signed char** |
| **SCHAR\_MIN** | **minimální hodnota signed char** |
| **UCHAR\_MAX** | **maximální hodnota unsigned char** |
| **SHRT\_MAX** | **maximální hodnota short** |
| **SHRT\_MIN** | **minimální hodnota short** |
| **USHRT\_MAX** | **maximální hodnota unsigned short** |
| **INT\_MAX** | **maximální hodnota int** |
| **INT\_MIN** | **minimální hodnota int** |
| **UINT\_MAX** | **maximální hodnota unsigned int** |
| **LONG\_MAX** | **maximální hodnota long** |
| **LONG\_MIN** | **minimální hodnota long** |
| **ULONG\_MAX** | **maximální hodnota unsigned long** |

**Inicializace spojuje přiřazení s deklarací. Např.:**

**#inlude <climits>**

**...**

**int n\_int = INT\_MAX;**

**…deklaruje proměnnou *n\_int* a nastaví jí největší možnou hodnotu typu *int*. Soubor *climits* obsahuje řádky, které jsou podobné následujícímu:**

**#define INT\_MAX 32767**

**Zde je *#define* (podobně jako *#include*) direktiva preprocesoru. Doslova říká: „Prohlédni si program, zda tam nejsou instance *INT\_MAX* a nahraď každý výskyt hodnotou *32767*.“ Takže direktiva *#define* pracuje podobně jako příkaz globálního vyhledávání a nahrazování v editoru nebo v textovém editoru. Až poté, co se tyto změny uskuteční, se program zkompiluje. Takto si můžete vytvářet i vlastní symbolické konstanty. Ovšem direktiva *#define* je přežitek z C. C++ má lepší řešení: *const*. Pokud ovšem chcete použít *#define*, není to nic těžkého.**

**Příklad:**

**#include <iostream>**

**using namespace std;**

**#define ZERO 0 // vytváří symbol ZERO pro hodnotu 0**

**#include <climits> /\* definuje INT\_MAX jako
 největší celočíselnou hodnotu \*/**

**int main()**

**{ short sam = SHRT\_MAX; /\* inicializuje proměnnou na maximální hodnotu \*/**

 **unsigned short sue = sam; /\* v pořádku, je-li proměnná sam již definována \*/**

 **cout << "Sam ma ulozeno " << sam << " dolaru a Sue " << sue;**

 **cout << " dolaru.\nNa kazdy ucet pridame 1 dolar.\nNyni ";**

 **sam = sam + 1;**

 **sue = sue + 1;**

 **cout << "ma Sam ulozeno " << sam << " dolaru a Sue " << sue;**

 **cout << " dolaru.\nChudak Sam!\n";**

 **sam = ZERO;**

 **sue = ZERO;**

 **cout << "Sam ma ulozeno " << sam << " dolaru a Sue " << sue;**

 **cout << " dolaru.\n";**

 **cout << "Z kazdeho uctu vezmeme 1 dolar.\nNyni ";**

 **sam = sam - 1;**

 **sue = sue - 1;**

 **cout << "ma Sam ulozeno " << sam << " dolaru a Sue " << sue;**

 **cout << " dolaru.\nStastna Sue!\n";**

 **return 0;**

**}**

## Chyby, které možná děláte

**Nepleťte si testovací operátor je rovno (==) s přiřazovacím operátorem (=).**

**musik == 4 // porovnání**

**…se zeptá: „Je musik rovno 4?“ Výraz má hodnotu true, nebo false.**

**musik = 4 // přiřazení**

**…přiřadí hodnotu 4 do musik.**

**Příklad:**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **int quizscores[10] = { 20, 20, 20, 20, 20, 19, 20, 18, 20, 20};**

 **cout << "Provedeno spravne:\n";**

 **int i;**

 **for (i = 0; quizscores[i] == 20; i++)
 //porovnani**

 **cout << "kviz " << i << " je 20\n";**

 **cout << "Provedeno nebezpecne spatne:\n";**

 **for (i = 0; quizscores[i] = 20; i++)**

 **// prirazeni**

 **cout << "kviz " << i << " je 20\n";**

 **return 0;**

**}**

## Vícesouborové projekty

**Pokud chceme vložit knihovnu do našeho programu, napíšeme:**

**#include <iostream>**

**Pokud chceme importovat náš vlastní soubor, postupujeme takto:**

**a) Vytvoříme např. soubor pro import češtiny (resp. kódy znaků) *cestina.h*. Náš soubor musí mít vždy příponu *.h*. Do něho napíšeme text (výpis znaků jako třeba *char cc=97* atd.) a uložíme.**

**b) Do našeho hlavního programu vepíšeme pod *#include <iostream>* řádku:**

**#include "cestina.h"**

**//soubor *cestina.h* se musí nacházet přímo v rootu programu, kde ho máme uložený, pokud tam není, kompilátor začne hledat ve standardním umístnění knihoven.**

**…nebo…**

**#include "C:/cpp/cestina.h"**

**//přímá adresa na náš soubor**

**…nebo…**

**#include <cestina.h>**

**//*cestina.h* se musí nacházet přímo v seznamu, kde se nacházejí zbylé (iostream, cctype, cmath…)**

## Úkol

**Pro dnešek bude jiný úkol. Poněvadž tato lekce byla spíše doplňující, nicméně myslím, že neméně důležitá a hlavně zajímavá, bude úkol následující:**

**Vytvořte co nejlepší program || hru, ale POUZE s využitím toho, co jsme se naučili pomocí těchto kurzů! Hodnotí se originalita, grafika, čeština, nápad a samozřejmě komentáře a správnost programování, dále bude velký důraz kladen na syntaxi. Žádné dlouhé nepřehledné kódy, ale přesné, přehledné a napsané co nejúspornějším a nejefektivnějším způsobem (tedy pokud mám rozsáhlé větvení, žádné *if else*, ale *switch*). Nejlepší práce vyhodnotím(e) a vystavím(e).**

# [C++ - 9. lekce](http://programujte.com/clanek/2005060502-c-9-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**2. 6. 2005       35 840×**

* **Ukazatele a volná paměť**
* **Ukazatele a filosofie**
* **Deklarování a inicializace ukazatelů**
* **Ukazatele a čísla**
* **Úkol**

## Ukazatele volná paměť

**Konečně se podíváme na to, na co všichni čekáte a budete nenávidět – ukazatele, nebo-li pointery. Nechci tuto lekci dělat příliš dlouhou proto, aby byla snáze pochopitelná. Snad se to podaří.Když počítačový program ukládá data, musí sledovat tři základní vlastnosti:**

* **kde je informace uložena**
* **jakou má hodnotu**
* **jaký druh informace je uložen**

**Doteď jste to používali: definovali jste jednoduchou proměnnou.**

**int a;**

**Poskytli jste typ (int) a symbolické jméno (a), které odkazuje na alokovanou paměť programem a toto místo sleduje. Jednoduše, proměnná musí být někde uložena, někde v paměti. Pojďme se ale nyní na to podívat z druhé stránky, která hraje důležitou roli v rozvoji tříd v  C++. Jsou to ukazatele.Ukazatel je proměnná, která, spíše než hodnotu, ukládá její adresu. Než však pojednáme o ukazatelích, pojďme se podívat, jak zjistíme adresu normálních proměnných. Je to velmi jednoduché. Na proměnnou pouze aplikujeme adresový operátor &, abyste zjistili její umístnění (adresu). Pokud máme proměnnou vek, tak její adresa je &vek.**

**int a=9;**

**cout << a << endl << &a;**

**Výsledek:**

**9**

**0x22ff7c**

**Adresa se samozřejmě bude lišit. Ukažme si to na dalším příkladu.**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **int a = 6;**

 **double cislo = 4.5;**

 **cout << "hodnota promenne a = " << a;**

 **cout << " a adresa promenne a = " << &a << " ";**

 **cout << "hodnota promenne cislo = " << cislo;**

 **cout << " a adresa promenne cislo = " << &cislo << " ";**

 **cin.get();**

 **return 0;**

**}**

**A výsledek:**

**hodnota promenne a = 6 a adresa promenne a = 0x22ff7c**

**hodnota promenne cislo = 4.5 a adresa promenne cislo = 0x22ff70**

**Když *cout* zobrazuje adresy, používá hexadecimálního označení – je to typické pro označení adresy. Podívejme se blíže:**

**→ 0x22ff7c − 0x22ff70 = 8**

**Je to logické, protože cislo je typu double a ten zaujímá 8 bajtů.**

**Na PC se reprezentace adres odrážejí metodou, která popisuje adresy pomocí hodnoty segmentu a ofsetu. Hodnota segmentu – 22 – u proměnné a identifikuje blok paměti, který se používá pro uložení dat. Ofsety – ff7c – reprezentují pozici paměti relativně k začátku segmentu.**

## Ukazatele a filozofie C++

**Objektově orientované programování se liší od tradičního procedurálního v důrazu OOP na rozhodování během běhu programu namísto v době kompilace. Běh programu znamená, když se program provádí, a doba kompilace znamená, když kompilátor sestavuje program dohromady. Rozhodnutí v době běhu poskytuje pružnost v přizpůsobení se aktuálním okolnostem. Uveďme si příklad: pole – abychom ho mohli deklarovat, musíme zadat určitou velikost pole, tedy velikost pole nastavujeme před kompilací, jeho velikost se nastaví v době kompilace. Možná si myslíte, že pole o 20 prvcích je dostatečná ve většině případů, ale jednou budete potřebovat výjimku, a tak nastavíte velikost na 200 prvků. To má ale za následek plýtvání pamětí po většinu času. OOP se pokouší vytvořit program mnohem pružněji tím, že ponechává taková rozhodnutí až do doby běhu. Tímto způsobem, poté co program běží, můžeme říci, že potřebujeme v jednom okamžiku pouze 20 prvků nebo v jiném 205 prvků. Zkrátka velikost pole je rozhodnutím v době běhu.**

**Ale vraťme se k ukazatelům, k jakožto zvláštnímu typu proměnné. Ukazatel tedy obsahuje adresu hodnoty a jméno ukazatele reprezentuje umístnění. Použitím operátoru \*, který se nazývá *nepřímou hodnotou* nebo *dereferenčním operátorem*, poskytuje hodnotu v dané lokaci. (Nehledejte v tom vědu, jestliže provincie je ukazatel, pak \*provincie je hodnota uložená na této adrese – ekvivalent k obyčejné proměnné.) Možná lépe pochopíte na příkladu:**

 **#include <iostream>**

 **using namespace std;**

 **int main()**

 **{**

 **int provincie = 6; // deklaruje proměnnou**

 **int \* farma; // deklaruje ukazatel na int**

 **farma = &provincie; // přiřazuje adresu int ukazateli**

 **// vyjadřuje hodnoty dvěma způsoby**

 **cout << "Hodnoty: provincie = " << provincie;**

 **cout << ", \*farma = " << \*farma << " ";**

 **// vyjadřuje adresy dvěma způsoby**

 **cout << "Adresy: &provincie = " << &provincie;**

 **cout << ", farma = " << farma << " ";**

 **// používá ukazatel na změnu hodnoty**

 **\*farma = \*farma + 1;**

 **cout << "Promenna provincie nyni = " << provincie << " ";**

 **cin.get();**

 **return 0;**

 **}**

**Výsledky:**

**Hodnoty: provincie = 6, \*farma = 6**

**Adresy: &provincie = 0x22ff7c, farma = 0x22ff7c**

**Promenna provincie nyni = 7**

**Jak můžete vidět, proměnná typu int provincie a ukazatelová proměnná farma jsou pouze dvě strany jedné mince. Detailně:**

**Proměnná provincie vypisuje primárně hodnotu, na získání její adresy použijeme operátor & → &provincie.**

**Proměnná farma vypisuje primárně adresu, na získání její hodnoty použijeme operátor \* → \*farma.**

**Poznámka: Je jedno, zda bude vypadat zápis int\* farma, int \* farma nebo int \*farma.**

**Použití mezer je volitelné, programátoři v C používali 3. způsob, v  C++ spíše 1. způsob.**

**Nechápete stále, k čemu nám ukazatele jsou?**

**Neodborně vysvětlím. Berte ukazatele jako další typ proměnných (jako jsou int, double, float…). Pokud napíšeme int \*farma, vytvoří se nám proměnná, stejně jako když deklarujeme int farma. Ovšem ukazatele mají tu zvláštnost, že jako hodnoty nejsou čísla a znaky, nýbrž adresy. Tudíž díky ukazateli můžeme přistupovat k samotné hodnotě proměnné jak klasicky, tak přes její adresu. Když to rozšířím, int \*farma má svou vlastní adresu, ale jako hodnotu si bere adresu cizí proměnné. Takže při změně hodnoty ukazatele int \*farma se změní i hodnota proměnné, na kterou ukazuje. Musíte si to vyzkoušet. Uvedu zde takový malý program, který slouží JEN k lepšímu pochopení.**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **int luke=9;**

 **int \*pavel=&luke;**

 **cout << luke <<endl << &luke <<endl << \*pavel <<endl << pavel <<endl << &pavel <<endl <<endl ;**

 **\*pavel = 10;**

 **cout << luke <<endl << &luke <<endl << \*pavel <<endl << pavel <<endl << &pavel;**

 **cin.get();**

 **return 0;**

**}**

## Deklarování a inicializace ukazatelů

#### Deklarování ukazatelů

**Počítač musí sledovat typ proměnné, na kterou se ukazatel odvolává. Proč? Adresa char nebo double vypadají stejně, ale každý používá jiný počet bajtů, proto musí deklarace ukazatele specifikovat, na jaký typ dat ukazatel ukazuje.**

**int \*farma;**

**To stanovuje, že \*farma je typu typu int. Protože se operátor \* používá ve vztahu k ukazateli, sama proměnná farma je ukazatelem. Říkáme, že farma ukazuje na *int*, respektive farma je ukazatel (adresa) a \*farma je *int* a nikoli ukazatel.**

**Ale pozor!**

**Buďte si vědomi věci, že deklarace**

**int \*p1, p2;**

**vytváří jeden ukazatel (p1) a jednu obyčejnou proměnnou (p2) typu int. Pro každé jméno ukazatelové proměnné potřebujete \*.**

**Na deklaraci ukazatelů ostatních typů použijeme stejnou syntaxi:**

**double \*meridlo;**

**char \*str;**

**Na ty, kteří nepozorně používají ukazatele, čeká nebezpečí. Jedním zvláště důležitým bodem při vytváření ukazatele je, že počítač alokuje paměť na úschovy adresy, ale nealokuje paměť na úschovu dat, na která adresa ukazuje. Vytvoření prostoru pro data je oddělený krok. Opomenutím tohoto kroku, jak je ukázáno v následující ukázce, je krokem k pohromě:**

**long \*melnik; // vytváří ukazatel na long**

**\*melnik = 123; //umístnění hodnoty do země nikoho**

**Jistě, melnik je ukazatel. Ale kam ukazuje? Programový kód selhal při přiřazení adresy do melnik. Nemůžeme ani říct, kam se uložila hodnota 123, protože melnik nebyla inicializována. Cokoli, co je hodnotou, program interpretuje jako adresu, na kterou se uloží 123. Když se přihodí, že melnik má hodnotu 1 200, potom se program pokusí umístit data na adresu 1 200, dokonce i když se stane, že je ta adresa uprostřed vašeho programového kódu. Tento druh chyby je jedním z nejzákeřnějších a těžce vystopovatelných opomenutí.**

**Vždy inicializujte ukazatel jednoznačnou a vhodnou adresou, než na něj použijete deferenční operátor (\*).**

## Ukazatele a čísla

**Ukazatele nejsou celočíselné typy, třebaže počítače spravují adresy jako celá čísla. Pojmově jsou ukazatele jinými typy než celá čísla. Celá čísla jsou čísla, která můžeme dělit, sčítat, odčítat apod., ale ukazatel popisuje lokaci a například nemá smysl násobit dvě lokace. Proto nemůžeme ukazateli jednoduše přiřadit celé číslo:**

**int \*pi;**

**pi = 0xB8000000; // nesoulad typů**

**Zde je levá strana ukazatel na int, takže jí můžete přiřadit adresu, ale pravá strana je pouze celým číslem. Možná víte, že *0xB8000000* je složenou adresou segmentu a ofsetu video paměti vašeho systému, ale nic v systému programu neříká, že toto číslo je adresou. C vám taková přiřazení dovoluje, avšak C++ podporuje mnohem přísněji soulad typů a kompilátor vám dá chybové hlášení – nesoulad typů. Chcete-li tedy použít numerickou hodnotu jako adresu, musíte vhodně přetypovat:**

**int \*pi;**

**pi = (int \*) 0xB8000000; // typy se nyní shodují**

**Nyní obě strany příkazu reprezentují adresy, takže je přiřazení platné.**

**Ukazatele mají některé další zajímavé vlastnosti, o kterých dále pojednáme, jakmile budou relevantní.**

**Příště se podíváme na to, jak se mohou ukazatele používat na alokaci paměťového prostoru v době běhu programu.**

## Úkol

**Vytvořte 5 proměnných a zrovna inicializujte na samé nuly. Necháte vypsat. Poté ke každé z nich vytvoříte jednoho ukazatele a pomocí nich změníte postupně hodnoty původních proměnných. Necháte vypsat. Nakonec necháte, aby uživatel zadal 1 číslo a o toto číslo se zvětší všechny hodnoty, avšak přes ukazatele! A úplně nakonec vypíšete postupně u každého čísla jeho adresu.**

**Př.:**

**1. cislo = 0**

**2. cislo = 0**

**3. cislo = 0**

**4. cislo = 0**

**5. cislo = 0**

**1. cislo = 5**

**2. cislo = 6**

**3. cislo = 8**

**4. cislo = 1**

**5. cislo = 5**

**zadej cislo : 5**

**1. cislo = 10**

**2. cislo = 11**

**3. cislo = 13**

**4. cislo = 6**

**5. cislo = 10**

# [C++ - 10. lekce](http://programujte.com/clanek/2005072302-c-10-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**16. 7. 2005       52 704×**

* **alokace paměti pomocí operátoru new**
* **uvolnění paměti pomocí operátoru delete**
* **upozornění na práci s pamětí**
* **úkol č.10**

**Nyní byste měli mít jistý pocit, jak ukazatelé pracují. Ovšem pojďme dále - alokování paměti za běhu programu.**

**Doposud jste inicializovali proměnné adresami proměnných. Co to jsou tedy proměnné? Pojďme si to zopakovat. Proměnné jsou *pojmenovanou* pamětí alokovanou v době kompilace. Ukazatele pouze poskytují druhé jméno pro paměť, ke které byste mohli přistupovat podle jména. K čemu tedy ukazatele? Pravá cena ukazatelů vyplouvá na povrch, když alokujete nepojmenovanou paměť na úschovu hodnot v době běhu programu. V tomto případě se k paměti dostanete pouze pomocí ukzatelů. Pokud jste se dříve učili C, mohli jste paměť alokovat pomocí f-ce malloc() - to stále můžete, ale C++ má lepší způsob - operátor new**

## Alokace paměti pomocí operátoru new

**Takže jak na to?
Zkusíme si vytvořit nepojmenovanou paměť za běhu programu pro hodnotu typu int a budeme k ní přistupovat pomocí ukzatele. Poskytneme operátoru *new* typ dat, pro kterou chceme paměť, *new* nalezne blok správné velikosti a navrátí jeho adresu, tu přiřadíme ukazateli.**

**int \* vek = new int**

**Tento příkaz říká programu, že chceme novou paměť vhodnou pro úschovu int. Jakmile ji nalezne, vrátí jeho adresu a ta se přiřadí do *vek* - což je typ ukazatele na int. Nyní je tedy *vek* adresou a *\* vek* hodnotou uložené na dané adrese. Ukazatele se vám možná jeví nešikovně, ale máte větší pohled nad tím, jak váš program spravuje paměť. Ukažme si tedy obecný tvar:**

**jmeno\_typu jmeno\_ukazatele = new jmeno\_typu**

**Poprvé určujete datový typ (int) požadované paměti a podruhé deklaraci vhodného ukazatele.**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

 **int \* promenna = new int; // alokuje prostor pro int**

 **\*promenna = 1001; // uloží tam hodnotu**

 **cout << "int hodnota = " << \*promenna << ": umisteni = "**

 **<< promenna << "**

**";**

 **double \* promenna2 = new double; // alokuje prostor pro double**

 **\*promenna2 = 10000001.0; // uloží tam double**

 **cout << "double hodnota = " << \*promenna2 << ": umisteni = "**

 **<< promenna2 << "**

**";**

 **cout << "velikost promenna = " << sizeof promenna;**

 **cout << ": velikost \*promenna = " << sizeof \*promenna << "**

**";**

 **cout << "velikost promenna2 = " << sizeof promenna2;**

 **cout << ": velikost \*promenna2 = " << sizeof \*promenna2 << "**

**";**

 **cin.get();**

 **return 0;**

**}**

**A výsledek :**

**int hodnota = 1001: umisteni = 00420750**

**double hodnota = 1e+007: umisteni = 00420710**

**velikost promenna = 4: velikost \*promenna = 4**

**velikost promenna2 = 4: velikost \*promenna2 = 8**

**Ukazatele *promenna* a *promenna2* ukazují na dva datové objekty - bez nich nemůžete k tmto paměťovým lokacím přistupovat. Zde také vidíme, proč musíme deklarovat typ, na který ukazatel ukazuje. Adresa odhalí jen začátek, kde je objekt uložen - jak vidíte, velikost ukazatele na int i double je stejná. Ale protože jsme deklarovali typy ukazatelů, program ví, že *promenna2* má 8 bajtů.**

**Mimo rozsah paměti?
Pokud počítač nemá dostatek volné paměti, vrací hodnotu 0\* => prázdný ukazatel. C++ zaručuje, že prázdný ukazatel nikdy neukazuje na platná data, takže se často používá na indikování selhání operátorů či f-cí, které jinak navracejí nepoužitelné ukazatele.**

**\* pan Mirek Virius doplňuje:
Podle standardu ISO 14882-2003 operátor new v základní podobě nevrací při neúspěchu 0, ale vyvolává vyjímku ntypu std::bad\_alloc. Chceme-li, aby vracel 0, musíme použít podobu new(nothrow) (a předtím uvést #include).**

**Možná stále nerozumíte - "proč bych měl používat něco, co je zbytečné, těžké... proč bych měl pracovat s nepojmenovanou pamětí, když můžu jednoduše pracovat s proměnnou?"
Odpověď je jednoduchá - klasické proměnné alokují paměť v době kompilace, kdežto alokovaná paměť pomocí operátoru *new* se alokuje ZA běhu programu - tedy pouze tehdy, pokud bude využita, potřebná... Klasická proměnná se vždy "přihlásí" o své místo na začátku běhu programu. Ale co když nepotřebujete tolik proměnných? Co když se ukáže, že některé proměnné nebudou za celý běh programu ani jednou využity? Myslíte si, že to není možné? A co podmínky? Jestliže uživatel zadá ano, provede se určená část programu. Takže možnost "ne" zůstane neupotřebena a veškerý následující kód po této podmínce nebude upotřeben - ani proměnné. Tedy zbytečně jste plýtvali místem. Kdežto při alokaci paměti během programu se vám toto nestane - paměť se alokuje pouze tehdy, je-li to nutné. Možná si říkáte, proměnné jsou jednodušší..., ale vy si nemůžete poté u složitějších programů plýtvat pamětí!! Určitě vás u nových her štve, že mají takové hardwarové nároky. Takže neučte se zavrhovat a odsuzovat alokoci paměti za běhu programu, naopak, naučte se to používat. No a proč tedy ukazatele? No přeci proto, abyste nějak mohli s danou pamětí pracovat. Tedy:
Pojmenovaná paměť - int a - můžeme se na ní odkázat jménem proměnné nebo ukazatelem na proměnnou.
Nepojmenovaná paměť - int vek = new int - můžeme se na ni odkázat pouze ukazatelem**

## Uvolnění paměti pomocí operátoru delete

**Mysleli jste si, že alokujete paměť za běhu programu a konec? Ne, tak jednoduché to není. Klasická proměnná je po ukončení programu zničena (uvolněna), ale paměť alokovaná za běhu programu zůstává i nadále po ukončení progrmu alokována, není uvolněna!**

**Pamatujte, takto alokovanou paměť (pomocí operátoru new) VŽDY uvolněte pomocí operátoru delete!**

**Takto uvolněná paměť může být poté znova použita jinou částí vašeho programu. Jedná se o nejefektivnější využití paměti.**

**int \* promenna = new int; //alokace paměti pomocí new**

**delete promenna; //uvolnění paměti pomocí delete**

**Toto odstraňuje paměť, na kterou ukazuje *promenna*, neodstraňuje to samotný ukazatel. Můžete ho znovu použít, aby ukazoval na jinou lokaci. Dále byste se neměli pokoušet uvolňovat blok paměti, který jste již uvolnili - výsledek není definován.**

**int \* promenna = new int; //ok**

**delete promenna; //ok**

**delete promenna; //chyba - již uvolněná paměť**

**int vek = 5; //ok**

**int \* cil = & vek; //ok**

**delete cil; //chyba - paměť nebyla alokována pomocí new**

**Upozornění:
Používejte *delete* pouze k uvolnění paměti alokované pomocí *new* a zapamatujte si, že je bezpečné aplikovat *delete* na nulový ukazatel.**

**Rozhodujícím testem na použití *delete* je to, že ho použijete na paměť alokovanou pomocí new. To znamená, že nepotřebujete použít stejný ukazatel k uvolnění paměti, který jste použili u new:**

**int \* promenna = new int; //alokuje paměť**

**int \* prom = promenna; /\*nastavuje druhý ukazatel na stejný blok \*/**

**delete prom; /\*uvolníte paměť pomocí druhého ukazatele \*/**

**Obyčejně dva ukazatele na stejný blok paměti nechcete vytvořit, protože tím vzrůstá pravděpodobnost, že se chybně pokusíte uvolnit stejný blok 2×. Ale jak brzy uvidíte, použití druhého ukazatele má smysl, když pracujete s f-cí, která navrací ukazatel.**

## Úkol č.10

**Bude lehký - alokujte 2× int, 1× double a 1× float(ne jako proměnné, ale pomocí operátoru new), nechte do nich uživatele zadat hodnoty a poté se ho zeptejte, zda bude potřebovat danou paměť, pokud ne, paměť se uvolní, pokud ano, nabídnete mu přepsání stávajících hodnot. A tak dokola, dokud uživatel neuzná data za zbytečná a dá uvolnit. Jakmile dá uvolnit, vypíšou se napsaná čísla. Budete tedy muset čísla nakopírovat do klasických proměnných.**

**V příští lekci se naučíte vytvářet dynamická pole pomocí operátoru new a jejich použití.**

# [C++ - 11. lekce](http://programujte.com/clanek/2005081901-c-11-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**9. 8. 2005       36 399×**

* **automatická, statická a volná paměť**
* **ukazatele na funkce**
* **získání adresy funkce**
* **deklarování ukazatele na funkci**
* **použití ukazatelů na vyvolání funkce**
* **úkol**

**Dnes se podíváme trošku na paměť, resp. její typy a poté zabrousíme zpět k funkcím – rozšíříme si obzory a na chvíli opustíme proklínané ukazatele.**

## Automatická, statická a volná paměť

**C++ má pro datové objekty tři způsoby řízení paměti, které jsou závislé na použitém způsobu alokace paměti: automatická, statická a volná paměť. Datové objekty alokované těmito třemi způsoby se jeden od druhého liší svou dobou existence.**

## Automatické proměnné

**To jsou proměnné definované uvnitř funkcí – rodí se automaticky, když se vyvolá daná funkce a jejich platnost (existence) končí, když funkce končí. Tento typ proměnné tedy existuje pouze tehdy, když program provádí příkazy uvnitř funkce.**

## Statická paměť

**Statická paměť existuje za běhu celého programu. Buď ji definujeme vně funkce, nebo pomocí klíčového slova static:**

**static int pocet\_psu = 88;**

**Později si o statické paměti řekneme více. Hlavním rozdílem tedy zůstává doba existence – buď za běhu celého programu (statická proměnná) nebo jen při vykonávání určité funkce (automatická proměnná).**

## Volná paměť

**Pomocí operátorů new a delete existují proměnné po takovou dobu, dokud je chcete. Tato volná paměť se ukládá odděleně od paměti používané pro statické a automatické proměnné. Paměť můžete alokovat v jedné funkci a uvolnit v jiné. Není tedy existenčně vázána na život funkce samotné, či dokonce programu. Dobu její existence řídíte vy.**

## Ukazatele na funkce

**Nyní se začneme orientovat zpět na funkce a ukazatele na chvíli opustíme. Ukazatele na funkce probereme jen ukázkově, toto téma si necháme na později. Funkce mají adresy, podobně jako datové objekty. Je možné napsat funkci, která přijímá adresu jiné funkce jako parametr. To první funkci umožňuje nalézt druhou a vykonat ji.**

**Vezměme si příklad, že chceme navrhnout funkci, která bude něco dělat, a dáme ji programátorům, aby ji použili. Každý programátor bude mít možnost použít svůj vlastní algoritmus, i když část programového kódu naší funkce zůstane. Mechanismus bude spočívat v předání adresy určitého (programátorova) algoritmu funkce, který chce programátor použít, naší funkci. Nic v tom nehledejte, pouze naší funkci předáme adresu programátorova algoritmu. Musíme pro to udělat následující:**

* **vzít adresu funkce**
* **deklarovat ukazatel na funkci**
* **použít ukazatel na funkci na vyvolání funkce**

## Získání adresy funkce

**Získání adresy funkce je doslova primitivní – pouze použijeme jméno funkce bez závorek – tedy samotné jméno. Jestliže je tedy nadruhou() funkce, pak nadruhou je její adresou. Uvažujme, že máme naši funkci, která se jmenuje nadruhou (resp. nadruhou()) a ta bude právě volat tu programátorovu funkci (algoritmus) nesoucí název prog (resp. prog()):**

**nadruhou(prog); // předává adresu prog() do nadruhou()**

**Ovšem pozor, neplést si s tímto:**

**natreti(prog()); //předává návratovou hodnotu prog()**

 **// do nadruhou()**

**Volání funkce nadruhou() umožňuje vyvolat funkci prog() z funkce nadruhou(). Volání funkce natreti() nejprve vyvolá funkci prog() a poté předá její návratovou hodnotu funkci natreti().**

## Deklarování ukazatele na funkci

**Podobně jako ukazatele na datové typy musí i ukazatel na funkci specifikovat, na jaký typ funkce ukazatel ukazuje. Tedy deklarace funkce by nám měla říct o funkci stejné věci jako její prototyp. Klasický prototyp je zde:**

**double nadruhou(int); // prototyp**

**A zde je deklarace ukazatele na funkci:**

**double (\*na\_druhou)(int);**

 **// na\_druhou ukazuje na funkci, která přijímá**

 **// jeden parametr typu int a navrací typ double**

**Obecně, chcete-li deklarovat ukazatel na funkci, vytvořte si nejprve klasický prototyp dané funkce a potom její jméno nahraďte výrazem ve tvaru (\*název\_ukazatele). To z název\_ukazatele vytvoří ukazatel na funkci tohoto typu.**

**Pozor na závorky!**

**Závorky mají vyšší prioritu než operátor \*, takže:**

**double (\*na\_druhou)(int); // na\_druhou ukazuje na**

 **// funkci, která navrací double**

**double \*na\_druhou(int); // na\_druhou() je funkce,**

 **// která navrací ukazatel-na-double**

**Když jste již nadeklarovali na\_druhou, můžete mu přiřadit adresu odpovídající funkce:**

**double nadruhou(int);**

**double (\*na\_druhou)(int);**

**na\_druhou = nadruhou;**

 **// na\_druhou nyní ukazuje na funkci nadruhou()**

**Zapamatujte si: nadruhou() musí odpovídat na\_druhou jak popisem, tak typem návratu. Neodpovídající řazení kompilátor odmítne!**

**double med(double);**

**int vcela(int)**

**double (\*vosa)(int);**

**vosa = med; // chybně - neodpovídající popis**

**vosa = vcela; // chybně - neodpovídající návratový typ**

**Tedy, když chceme, aby funkce nadruhou() použila funkci natreti(), předáme jí její adresu:**

**nadruhou(int pocet\_znaku, double (\*funkce)(int));**

**nadruhou(50, natreti); // funkční volání nadruhou(),**

 **//aby použila natreti()**

**Jak tedy vidíte, napsání prototypů je ta složitá část, na rozdíl od předání adres, což je velmi jednoduché.**

## Použití ukazatelů na vyvolání funkce

**Dostali jsme se ke konečné části postupu, který spočívá v použití ukazatele na vyvolání funkce, na kterou ukazuje. Nezapomínejte na deklaraci ukazatele! (\*nadruhou) hraje stejnou roli jako jméno funkce:**

**double jedna(int);**

**double (\*dva)(int);**

**dva = jedna; // dva nyní ukazuje na funkci jedna()**

**double x = jedna(4); // volání jedna() používá jméno funkce**

**double y = (\*dva)(5); // volání jedna() používá ukazatel dva**

**Dokonce vám C++ dovoluje použít pf, jako by to bylo jméno funkce:**

**double z = pf(7); // také volání jedna(), pomocí**

 **// ukazatele pf**

**I když je to pěkné, budeme používat tvar double y, i když vypadá drastičtěji. Poskytuje nám silnou vizuální připomínku, že programový kód využívá ukazatel na funkci. Jak vidíte, není to zase tak složité, jak se na první pohled zdá.**

**Možná přemýšlíte, jak mohou být pf a (\*pf) navzájem ekvivaletní? Je to z toho důvodu, že pf ukazuje na funkci a \*pf je funkce, a proto byste měli pro volání funkce použít (\*jedna)(). Druhý názor je, že jméno funkce je ukazatel na funkci, ukazatel by tedy měl fungovat jako její jméno – z toho důvodu byste měli pf() použít jako volání funkce. C++ zastává kompromisní stanovisko, že obě formy jsou správné – povoleny, i když jsou jedna s druhou logicky neslučitelné.**

**Nyní si ukážeme příklad, jak použít ukazatele v programu:**

**#include <iostream>**

**using namespace std;**

**double vypocet(int);**

**double nasob(int);**

**// druhý argument je ukazatel na funkci typu double,**

**// která přebírá argument typu int**

**void vypis(int lines, double (\*pf)(int));**

**int main()**

**{**

 **int code;**

 **cout << "Kolik radku programoveho kodu potrebujete? ";**

 **cin >> code;**

 **cout << "Tady je odhad od vypocet:**

**";**

 **vypis(code, vypocet);**

 **cout << "Tady je odhad od nasob:**

**";**

 **vypis(code, nasob);**

 **return 0;**

**}**

**double vypocet(int lns)**

**{**

 **return 0.05 \* lns;**

**}**

**double nasob(int lns)**

**{**

 **return 0.03 \* lns + 0.0004 \* lns \* lns;**

**}**

**void vypis(int lines, double (\*pf)(int))**

**{**

 **cout << lines << " radku zabere ";**

 **cout << (\*pf)(lines) << " hodin**

**";**

**}**

**A výsledek:**

**Kolik radku programoveho kodu potrebujete? 30**

**Tady je odhad od vypocet:**

**30 radku zabere 1.5 hodin**

**Tady je odhad od nasob:**

**30 radku zabere 1.26 hodin**

**Program volá funkci vypis() dvakrát, jednou jí předává adresu funkce vypocet(), jednou nasob(). V prvním případě funkce vypis() používá vypocet() k výpočtu nezbytného počtu hodin a ve druhém případě k tomuto výpočtu používá nasob(). Návrh tohoto programu podporuje budoucí rozvoj programu. Když někdo vyvine svůj vlastní algoritmus na odhad času, nemusí přepisovat vypis(). Místo toho pouze dodá svoji vlastní funkci moje() a zajistí, že má správnou signaturu a návratový typ. Tento postup se samozřejmě používá u složitějších programových kódů, kde by bylo přepsání funkce vypis() zdlouhavější a obtížnější. Metoda ukazatele na funkci také dovoluje dotyčnému modifikovat chování funkce vypis(), dokonce i když nemá přístup k jejímu zdrojovému kódu.**

## Úkol č. 11

**Tentokrát bude teorie. Odpovídejte stylem**

1. **vaše odpověď**
2. **vaše odpověď**
3. **Vytvořte prototypy funkcí, které vyhovují následujícím popisům:**
	1. **curo() nepřijímá žádné parametry, ani nevrací hodnotu**
	2. **prom() přijímá parametr typu int a vrací hodnotu typu float**
	3. **mpg() přijímá 2 parametry typu double a navrací double**
	4. **summa() přijímá jméno pole typu long a jeho velikost jako hodnotu a navrací hodnotu typu long**
	5. **doctor() přijímá řetězový parametr (řetězec se nemá modifikovat) a navrací hodnotu typu double**
	6. **meta() přijímá jako parametr strukturu typu boss a nic nevrací**
4. **Funkce program má návratovou hodnotu typu int. Jako parametr přijímá adresu funkce, která má jako parametr ukazatel na const char a která také navrací typ int. Napište prototyp funkce.**
5. **Napište jednoduchý program, kde bude obsažena jedna automatická proměnná, jedna statická a jedna volná paměť (okomentováno, kde je která). Program nemusí mít uživatelský vstup – jde o kód.**

**Důležitým faktorem je jednoduchost!**

# [C++ - 12. lekce](http://programujte.com/clanek/2005111104-c-12-lekce/)

[**Lukáš Churý**](http://programujte.com/profil/1483-lukas-chury/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**13. 11. 2005       46 459×**

* **standardní parametry**
* **šablony funkcí**
* **explicitní specializace**
* **úkol č. 12**

## Standardní paramatery

**Podíváme se blíže na samotné volání f-ce a jeho detaily. Pamatujete si na funkce? Pokud ne, menší připomenutí. Nejvíce nás nyní zajímá samotné volání f-ce.**

**...**

**double obsah(double x); //prototyp f-ce**

**...**

**int main();**

**{**

**obsah(5); //volání f-ce**

**...**

**return 0;**

**}**

**double obsah(double x) //definice f-ce**

**{**

 **return x\*x\*x;**

**{**

**A teď si představte, že hodnotu '5' používáte často. Proč byste ji tedy vždy měli zadávat, resp. proč ji pokaždé nutně zadávat? Existuje vychytávka - standardní parametr. Standardní parametr je hodnota, která bude použita, pokud vynecháte odpovídající skutečný parametr ve volání funkce. Než-li budu cokoliv vysvětlovat, ukážeme si přímo, jak ho použít:**

**...**

**double obsah(double x=2); //změněný prototyp f-ce**

**...**

**int main()**

**{**

**obsah(); //volání f-ce**

**...**

**return 0;**

**}**

**double obsah(double x) //definice f-ce**

**{**

 **return x\*x\*x;**

**}**

**Jak jste tedy viděli, jediné, co musíte pro použití standardního parametru udělat, je definovat tento parametr v prototypu f-ce. A nyní si to vysvětlíme. V našem případě bude použita defaultně hodnota '5'. Možná se ptáte, proč byste to měli dělat? Možná vám to přijde zbytečné, zvláště v tomto případě, ale později uznáte, že je to víceméně chytrá věcička, jak si ušetřit čas. Představte si f-ci, která se vás ptá, kolik znaků z nějakého řetězce má vypsat (řetězec má velikost 20 znaků, typ char(char retezec[20])). No pokud chceme vypsat celý řetězec, proč bychom měli zadávat '20'? Vynecháme, a automaticky se hodnota 20 doplní a tím se vypíše celý řetězec. Možná se teď ptáte, co když ale chcete vypsat jen 10 znaků? Tak místo vynechání napíšete '10'. Standardní parametry nám tedy šetří práci.
Viz program napsaný výše: pokud nezadáme při volání f-ce žádné číslo, bude dosazena '2', pokud zadáme, bude dosazena zadaná hodnota. Samozřejmě existují jistá pravidla:**

**Takto vypadá kód, kdy bude automaticky doplněna hodnota:**

**#include <iostream>**

**using namespace std;**

**double obsah(double x=2); //změněný prototyp f-ce**

**int main()**

**{**

**double s;**

**s=obsah(); //volání f-ce**

**cout << s;**

**cin.get();**

**cin.get();**

**return 0;**

**}**

**double obsah(double x) //definice f-ce**

**{**

**return x\*x\*x;**

**}**

**<\*/kod\*>**

**<p>A takto se zadaným parametrem : </p>**

**<\*kod\*>**

**#include <iostream>**

**using namespace std;**

**double obsah(double x=2); //změněný prototyp f-ce**

**int main()**

**{**

**double n,s;**

**cin >> n;**

**s=obsah(n); //volání f-ce**

**cout << s;**

**cin.get();**

**cin.get();**

**return 0;**

**}**

**double obsah(double x) //definice f-ce**

**{**

**return x\*x\*x;**

**}**

**Jestliže použijete f-ci s více parametry, musíte upřednostnit v seznamu obyčejné parametry před standardními, resp. nejdříve vypsat klasické paramtery, poté až standardní. Př.:**

**int rovnice(int x, int y=4, int z=7); //ok**

**int rovnice2(int x, int y=4, int z); /\*špatně,*int z* musí být před *int y=4 \*/***

**int rovnice3(int x=1, int y=3, int z=7); //ok**

**Volání *rovnice* tedy může vypadat takto :**

**vysledek = rovnice(5); //stejné jako rovnice(5,4,7)**

**vysledek = rovnice(2,8); //stejné jako rovnice(2,8,5)**

**vysledek = rovnice(1,6,3); //nebyly použity žádné standartní parametry**

**Přiřazování probíhá zleva doprava, nemůžete je přeskočit. Následující je tedy nepřípustné:**

**vysledek = rovnice(1, ,9); //chybně**

## Šablony funkcí

**Předpokládejme, že chcete udělat f-ci, která zaměňuje 2 hodnoty typu int. Jednoduché. Ale co když pak chcete zaměnit 2 hodnoty jiného typu, například double? Máte několik možností:
- vyměnit v programu typy a zkompilovat
- udělat další funkci
- ...
Nepřipadá vám to zbytečné plýtvaní časem? V C++ existují šablony pro f-ce, které nám dávají nové možnosti - například stačí jedna funkce pro záměnu 2 hodnot jakéhokoliv typu. Zajímavé, že? Tak se pojďme podívat, jak na to.**

**template <typename jmeno\_typu>**

**void nazev\_funkce(seznam\_parametru)**

**{**

**prikazy;**

**}**

**První řádek určuje, že jste vytvořili šablonu a jak jste nový typ pojmenovali. Klíčové slova *typename* a *template* jsou povinná. Místo *typename* můžete použít starší "verzi" - *class*. Sama šablona nevytváří žádné funkce - pouze říká kompilátoru, jak má funkci definovat. Když to řeknu laicky, kompilátor nahradí vámi pojmenovaný typ typem, který je právě potřeba. Nyní si ukážeme šablonu na záměnu oněch 2 hodnot:**

**template <typename Types>**

**void vymena(Types &a, Types &b)**

**{**

**Types temp;**

**temp=a;**

**a=b;**

**b=temp;**

**}**

**A celý program :**

**#include <iostream>**

**using namespace std;**

**// prototyp šablony funkce**

**template <class Types> // nebo typename Any**

**void Swap(Types &a, Types &b);**

**int main()**

**{**

**int i = 10;**

**int j = 20;**

**cout << "i, j = " << i << ", " << j << ".\n";**

**cout << "Pouziti prekladacem generovane funkce pro zamenu cisel typu int:\n";**

**Swap(i,j); // generuje void Swap(int &, int &)**

**cout << "Nyni i, j = " << i << ", " << j << ".\n";**

**double x = 24.5;**

**double y = 81.7;**

**cout << "x, y = " << x << ", " << y << ".\n";**

**cout << "Pouziti prekladacem generovane funkce pro zamenu cisel typu double:\n";**

**Swap(x,y); // generuje void Swap(double &, double &)**

**cout << "Nyni x, y = " << x << ", " << y << ".\n";**

**return 0;**

**}**

**// definice šablony funkce**

**template <class Types> // nebo typename Any**

**void Swap(Types &a, Types &b)**

**{**

**Types temp; // temp je proměnná typu Types**

**temp = a;**

**a = b;**

**b = temp;**

**}**

**i, j = 10, 20.**

**Pouziti prekladacem generovane funkce pro zamenu cisel typu int:**

**Nyni i, j = 20, 10.**

**x, y = 24.5, 81.7.**

**Pouziti prekladacem generovane funkce pro zamenu cisel typu double:**

**Nyni x, y = 81.7, 24.5.**

**Musíte si uvědomit, že šablony funkcí nedělají vaše programy stručnějšími, kratšími. Výsledný programový kód neobsahuje žádné šablony, obsahuje pouze skutečné funkce. Výhody šablon jsou tedy jednoduchost a spolehlivost.**

## Explicitní specializace

**Slouží ke specializaci při volání. Používají se, pokud definice přesně odpovídá funkčnímu volání.**

**1. normální funkce ->void show(int n)
2. explicitní specializace -> template<> void vymen(int a, int b)
3. šablonové funkce -> template
4. void swap(Neco a, Neco b)**

**Kdy se použije co?**

**...**

**double u,v;**

**vymen(u,v); //použije se 3.**

**...**

**int a,b;**

**vymen(a,b); //použije se 2. => specializace na int**

**#include <iostream>**

**using namespace std;**

**template <class Any>**

**void Swap(Any &a, Any &b);**

**struct job**

**{**

**char name[40];**

**double salary;**

**int floor;**

**};**

**// explicitní specializace**

**template <> void Swap<job>(job &j1, job &j2);**

**void Show(job &j);**

**int main()**

**{**

**cout.precision(2);**

**cout.setf(ios::fixed, ios::floatfield);**

**int i = 10, j = 20;**

**cout << "i, j = " << i << ", " << j << ".\n";**

**cout << "Pouziti prekladacem generovane funkce pro zamenu cisel typu int:\n";**

**Swap(i, j); // generuje void Swap(int &, int &)**

**cout << "Nyni i, j = " << i << ", " << j << ".\n";**

**job sue = {"Susan Yaffee", 63000.60, 7};**

**job sidney = {"Sidney Taffee", 66060.72, 9};**

**cout << "Pred zamenou obsahu struktur job:\n";**

**Show(sue);**

**Show(sidney);**

**Swap(sue, sidney); // používá void Swap(job &, job &)**

**cout << "Po zamene obsahu struktur job:\n";**

**Show(sue);**

**Show(sidney);**

**return 0;**

**}**

**template <class Any>**

**void Swap(Any &a, Any &b) // obecná verze**

**{**

**Any temp;**

**temp = a;**

**a = b;**

**b = temp;**

**}**

**// zaměňuje pouze pole salary a floor struktury job**

**template <> void Swap<job>(job &j1, job &j2) // specializace**

**{**

**double t1;**

**int t2;**

**t1 = j1.salary;**

**j1.salary = j2.salary;**

**j2.salary = t1;**

**t2 = j1.floor;**

**j1.floor = j2.floor;**

**j2.floor = t2;**

**}**

**void Show(job &j)**

**{**

**cout << j.name << ": " << j.salary**

**<< " Kc na poschodi " << j.floor << "\n";**

**}**

**i, j = 10, 20.**

**Pouziti prekladacem generovane funkce pro zamenu cisel typu int:**

**Nyni i, j = 20, 10.**

**Pred zamenou obsahu struktur job:**

**Susan Yaffee: 63000.60 Kc na poschodi 7**

**Sidney Taffee: 66060.72 Kc na poschodi 9**

**Po zamene obsahu struktur job:**

**Susan Yaffee: 66060.72 Kc na poschodi 9**

**Sidney Taffee: 63000.60 Kc na poschodi 7**

**Když použijete explicitní specializaci, kompilátor zjistí specializovanou definici, která přesně odpovídá funkčnímu volání, použije ji, aniž by se podíval na šablony.**

## Úkol č.12

**Napište program, který zamění 2 hodnoty (pomocí funkce a šablon) jakéhokoliv typu. Tedy, uživatel zadá 2 hodnoty a ty se poté ve výpisu prohodí.**

[C++ – řetězce](http://programujte.com/clanek/2007061002-c-retezce/)

[**Martin Bodocký**](http://programujte.com/profil/20356-martin-bodocky/)[**Martin Valent**](http://programujte.com/profil/7799-martin-valent/)**19. 8. 2007       27 347×**

**Dneska si zoberieme na pozorovanie, čo sú to reťazce a ako sa používajú.**

## Reťazce

**Reťazec je postupnosť znakov uložených za sebou v pamäti. Používame dva spôsoby práce s reťazcami. Prvým, je spôsob, ktorý bol prevzatý od jazyka C a naučíme sa ho teraz. Druhý, je využitie knižnice string, a to si ukážeme v neskorších pokračovaniach. Zatiaľ nám postupnosť znakov uložených za sebou v pamäti pripomína pole, a je to správna úvaha. Reťazec je pole prvkov typu char, kde má každý znak svoj prvok poľa. Reťazce ponúkajú vhodný spôsob ukladania textových informácií. Reťazce majú ešte jednu zvláštnu vlastnosť, posledný znak každého reťazca je nulový znak. Tento znak má zápis '\0' a má kódovú hodnotu ASCII 0, slúži na ukončenie reťazca. Pozrite sa na nasledujúce dve deklarácie:**

**char pes[5] = {'p', 'e', 't', 'k', 'o'}; /\* nie je
 retazec \*/**

**char macka[5] = {'c', 'i', 'c', 'a', '\0'}; //retazec**

**Obidve polia sú typu char, ale iba druhé je reťazcom. Jazyk C++ ponúka funkciu pre prácu s reťazcami, ktorá používa objekt cout. Všetky funkcie spracúvajú reťazce, až kým nenarazia na nulový znak. Ak použijeme cout, aby vypísal premennú mačka, tak vypíše cica, prečíta nulový znak a skončí. Ale ak mu dáme vypísať premennú pes a on vypíše pole znakov, ktoré nasleduje v pamäti až kým niekde v pamäti nenájde nulový znak. Aj keď sa tento znak v pamäti vyskytuje celkom často nemôžeme takéto pole brať za reťazec. Podľa príkladu poľa mačka vyzerá inicializácia dosť zložito. Ale existuje aj jednoduchší spôsob inicializácie reťazcov. Stačí jednoducho použiť reťazec v úvodzovkách, ktorý sa nazýva reťazcová konštanta. Napr.:**

**char meno[10] = "Pan Peter"; /\* '\0' sa doplni samo ale musite mu tam nechat miesto \*/**

**char jav[] = "Bublinky"; /\* Velkost si spocita prekladac a doplni si sam '\0' \*/**

**Reťazcom v úvodzovkách je vždy doplnený nulový znak, tak ho nemusíte vypisovať. Taktiež aj vstupy z klávesnice si samo dopĺňajú nulový znak. Ale musíme si byť istý, že pole je dostatočne veľké pre uloženie všetkých znakov, vrátane nulového znaku. Inicializácia znakového poľa reťazcovou konštantou je jeden z prípadov, kedy môže byť bezpečnejšie nechať určenie počtu prvkov na prekladači. Ak vytvoríte pole väčšie než je reťazec, okrem zaberaniu zbytočne pamäte neurobíte nič zlé. Je to preto, lebo funkcie pracujúce s reťazcami, neriadia veľkosť poľa ale hľadajú nulový znak.**

**Zapamätajte si: Pri určovaní minimálnej veľkosti poľa pre uloženie reťazca nezabudnite na koncový nulový znak.**

**Všimnite si, že reťazcová konštanta je nezameniteľná za znakovú konštantu. Znaková konštanta je „S“ predstavuje zápis kódu znaku. Na systéme so znakovou sadou ASCII je 'S' iba iným zápisom čísla 83. Preto príkaz:**

**char shirt\_size = 'S'; //toto je v poriadku**

**priradzujeme hodnotu 83 premennej shirt\_size. Ale reťazcová konštanta „S“ reprezentuje reťazec zložený z dvoch znakov S a '\0'. Horšie je, že v skutočnosti predstavuje adresu pamäte, na ktorej je reťazec uložený. Takže príkaz:**

**chart\_size = "S"; // nepovolene miesanie typov**

**sa pokúša premennej chart\_size priradiť pamäťovú adresu. Ale v C++ má adresa samostatný typ a preto to prekladač nedopustí.**

## Spájanie reťazcov

**C++ umožňuje spájanie reťazcových konštánt, to znamená spojenie dvoch reťazcov do jedného. Ľubovoľné dva reťazce oddelené iba bielymi znakmi ako medzera, tabulátor alebo znakom nového riadku. Všetky nasledujúce príkazy sú rovnaké:**

**cout << "Som programator ale musim sa este " "dost ucit aby som bol dobry.\n";**

**cout << "Som programator ale musim sa este dost ucit aby som bol dobry.\n";**

**cout << "Som programator ale musim sa este "**

**"dost ucit aby som bol dobry.\n";**

**Všimnite si, že spojenie nepridáva žiadne medzery do spojených reťazcov. Prvý znak druhého reťazca bezprostredne nasleduje za posledným znakom prvého reťazca, čiže znak „\\0“ sa nepoužíva.**

## Použitie reťazcov v polia

**Dva najbežnejšie spôsoby vkladania reťazcov do poľa, sú inicializácie poľa reťazovou konštantnou a načítanie vstupu do poľa z klávesnice alebo zo súboru. Teraz si to ukážeme za použitia objektu cin. Program ďalej používa knižnú fuknciu strlen() za zistenie dĺžky reťazca, kvôli tej to funkcii sme pridali do súboru knižnicu cstring, ktorá nám to dovolí využívať a obsahuje aj ďalšie funkcie na prácu z reťazcami.**

**// retazce1.c -- ukladanie retazcou do pola**

**#include**

**#include // pre funkciu strlen()**

**using namespace std;**

**int main(){**

 **const int velkost=20 ;**

 **char meno1[velkost]; //prazdne pole**

 **char meno2[velkost]= "ProgramerC++"; //inicializacia pola**

 **cout << "Dobry den! Ja som " << meno2 << "\n";**

 **cout << "Kto ste vy??\n";**

 **cin >> meno1;**

 **cout << "Dobre, "<< meno1 << " vase meno obsahuje ";**

 **cout << strlen(meno1) << " znakov a je ulozene\n";**

 **cout << "v poli o velkosti "<< sizeof(meno1) << " bajtov.\n";**

 **cout << "Prve pismeno vasho mena je "<< meno1[0] << ".\n";**

 **meno2[9]='\0'; // nulovy znak**

 **cout << "A tu je prvych 9 znkov mojho mena: "<< meno2;**

 **cout << "\n";**

 **return 0;**

**}**

**A môj výstup je:**

**Dobry den! Ja som ProgramerC++**

**Kto ste vy??**

**Martin**

**Dobre, Martin vase meno obsahuje 6 znakov a je ulozene**

**v poli o velkosti 20 bajtov.**

**Prve pismeno vasho mena je M.**

**A tu je prvych 9 znkov mojho mena: Programer**

**Najprv si všimnite, že operátor sizeof vracia veľkosť celého poľa 20 bajtov ale funkcia strlen() vracia iba dĺžku reťazca a nie celého poľa. Funkcia strlen() vracia veľkosť viditeľných znakov bez nulového. Preto vracia hodnotu 6 namiesto 7. Ale na jeho uloženie potrebujeme pole o veľkosti strlen(meno1)+1. Pretože meno1 a meno2 sú polia, dá sa k ním pristupovať pomocou indexov. To využíva náš program, keď prečíta iba prvý znak reťazca meno1[0]. Ďalej sme nastavili meno2[9]. A to nám spôsobili skrátenie reťazca na 9 znakov aj keď ďalšie znaky v poli ostávajú. Nakoniec si ešte všimnite, že na určenie veľkosti poľa sme použili symbolickú konštantnú, to nám zjednodušuje úpravu programu pri zmenách veľkosti programu. A ešte pozor, ak do programu zadáte viac ako 20 znakov (19+nulový), tak Vás operačný systém zastaví, že chcete využívať pamäť, ktorú nemáte blokovanú.**

## Problémy so vstupom

**Program má chybu, ktorú si teraz opravíme a vysvetlíme. Ale na začiatok ešte jeden príklad, ako by sa to nemalo robiť:**

**// retazce2.cc -- citanie viacerych retazcov**

**#include**

**using namespace std;**

**int main() {**

 **const int velkost=20;**

 **char meno[velkost];**

 **char dezert[velkost];**

 **cout << "Zadajte vase meno: \n";**

 **cin >> meno;**

 **cout << "Zadajte vas oblubeny dezert: \n";**

 **cin >> dezert;**

 **cout << "Mam bohovsky dezert "<< dezert;**

 **cout << ", ktory si zasluzi iba ten "<< meno << ".\n";**

 **return 0;**

**}**

**Zadanie je jednoduché chceme načítať do dvoch reťazcov vaše meno a názov dezertu. A náš výstup je:**

**Zadajte vase meno:**

**Martin Bodocky**

**Zadajte vas oblubeny dezert:**

**Mam bohovsky dezert Bodocky, ktory si zasluzi iba ten Martin.**

**No, a ako sme zistili, nebolo nám ani umožnené zadať názov dezertu. Program síce vypísal požiadavku na zadanie dezertu, ale hneď za ním vypísal poslednú správu. Problém spočíva v tom, že objekt cin rozoznáva koniec reťazca. A z klávesnice sa nedá zadať nulový znak, tak objekt cin potrebuje iný spôsob ukončenia reťazca. Objekt cin preto používa pre určenie nulového znaku medzery, tabulátory a znaky nového znaku. Preto objekt cin načítal iba prvý reťazec nášho vstupu a ukončí ho nulovým znakom. V praxi to znamená, že cin prečíta Martin ako prvý reťazec a umiestni ho do poľa meno. Reťazec ostane vo vstupnej fronte. Potom, keď hľadá cin odpoveď na druhý reťazec, nájde Bodocky a vloží ho do poľa dezert. Ďalší problém, ktorý sme si ešte nedemonštrovali, a to je to, že nám program nezabráni vložiť dlhší reťazec ako 20 znakov. Napr.:**

**Dobry den! Ja som ProgramerC++**

**Kto ste vy??**

**MartinBodockyProgramator**

**Dobre, MartinBodockyProgramator vase meno obsahuje 24 znakov a je ulozene**

**v poli o velkosti 20 bajtov.**

**Prve pismeno vasho mena je M.**

**A tu je prvych 9 znkov mojho mena: Programer**

**\*\*\* stack smashing detected \*\*\*: ./retazce1 terminated**

**Zrušené (core dumped)**

**Ukázal som to na prvom príklade retazce1.cc a tento problém nebudeme teraz riešiť je na to potrebné viacej znalostí.**

## Riadkovo orientovaný vstup: ****getline()**** a ****get()****

**Aby sme mohli, ako reťazec, ukladať celé vety a nemuseli by sme čítať vstup po slovách potrebujeme iný prístup. Treba použiť riadkovú orientáciu. Našťastie máme knižnicu iostream, ktorá obsahuje objekt cin a ten ma ešte funkcie cin.getline() a cin.get(). Cin.getline() prečíta celý riadok a pre označenie konca používa znak nového riadku respektíve ENTER. Preberá dva argumenty, a to meno poľa preuloženie vstupného riadku, druhý argument obmedzuje počet vstupných znakov. Ak je to napríklad 20, tak prečíta 19 znakov a ponechá miesto na nulový znak. čiže funkcia cin.getline končí ak prečíta požadovaný počet znakov, alebo narazí na znak konca riadku. Volanie funkcie vyzerá tak to:**

**cin.getline(meno,20);**

**Tento príkaz prečíta celý riadok do poľa meno o maximálnej dĺžke 19 znakov puls nulový znak. Tak si ukážeme opravenú verziu nášho programu:**

**// retazce3.cc -- citanie viac ako jedneho slova pomocou getline**

**#include**

**using namespace std;**

**int main() {**

 **const int velkost=20;**

 **char meno[velkost];**

 **char dezert[velkost];**

 **cout << "Zadajte vase meno: \n";**

 **cin.getline(meno,velkost); //precita cely riadok**

 **cout << "Zadajte vas oblubeny dezert: \n";**

 **cin.getline(dezert,velkost);**

 **cout << "Mam vyborny "<< dezert;**

 **cout << ", ktory si zasluzi iba " << meno << ".\n";**

 **return 0;**

**}**

**Môj výstup:**

**// retazce3.cc -- citanie viac ako jedneho slova pomocou getline**

**#include**

**using namespace std;**

**int main() {**

 **const int velkost=20;**

 **char meno[velkost];**

 **char dezert[velkost];**

 **cout << "Zadajte vase meno: \n";**

 **cin.getline(meno,velkost); //precita cely riadok**

 **cout << "Zadajte vas oblubeny dezert: \n";**

 **cin.getline(dezert,velkost);**

 **cout << "Mam vyborny "<< dezert;**

 **cout << ", ktory si zasluzi iba " << meno << ".\n";**

 **return 0;**

**}**

**Program teraz prečíta meno aj dezert a vypíše ho podľa očakávania. Prečíta vstup až do znaku nového riadku, ktorý označuje jeho koniec. Ale teraz skúsime iný prístup, lebo poznáme ešte druhú funkciu na orientovaný vstup a to cin.get(), ktorá má niekoľko variant. Jedna z nich pracuje pomaly, tak isto ako cin.getline(). Preberá rovnaké argumenty a číta tiež do konca riadku. Ale namiesto prečítania znaku konca riadku ho nechá vo vstupnej fronte. Predpokladajme použitie dvoch funkcii za sebou:**

**cin.get(meno,velkost);**

**cin.get(dezert,velkost); // tu vznika problem**

**Pretože prvé volanie nechá znak konca riadku vo vstupnej fronte a tento znak je prvý, ktorý uvidí druhá funkcia cin.get(). Preto si táto funkcia myslí, že už dosiahla konca aj bez toho, že by niečo prečítala. Bez nejakej pomoci sa funkcia cin.get() cez tento znak nedostane. Ale môžeme si pomôcť volaním funkcie cin.get() bez parametrov, ktorá prečíta jeden znak aj keď je to znak konca riadku. To môžeme použiť pri odstránení znaku konca riadku a prípravu na ďalší vstup. napr.:**

**cin.get(meno,velkost); //precita prvy riadok**

**cin.get(); //precita znak konca riadku**

**cin.get(dezert,velkost); // precita druhy riadok**

**Ďalšou možnosťou použitia funkcie cin.get() je spojenie dvoch členov knižnice:**

**cin.get(meno,velkost).get();**

**Tento zápis umožňuje skutočnosť, že funkcia cin.get(meno,velkost) vracia objekt, ktorého použitie môže vyzerať aj takto:**

**cin.getline(meno1,velkost).getline(meno2,velkost);**

**ktorý prečíta dva za sebou idúce riadky do poli meno1 a meno2, presne ako dve za sebou idúce funkcie cin.getline(). Teraz budeme demonštrovať použitie funkcie cin.get():**

**//retazce3.cc -- precitanie jedneho a viacej slov pomocou cin.get()**

**#include**

**using namespace std;**

**int main() {**

 **const int velkost=20;**

 **char meno[velkost];**

 **char dezert[velkost];**

 **cout << "Zadajte vase meno: \n";**

 **cin.get(meno,velkost).get(); //precita riadok a potom aj znak konca riadku**

 **cout << "Zadajte vas oblubeny dezert: \n";**

 **cin.get(dezert,velkost).get();**

 **cout << "Mam vyborny "<< dezert;**

 **cout << ", ktory si zasluzi iba " << meno << ".\n";**

 **return 0;**

**}**

**Náš očakávaný výstup:**

**Zadajte vase meno:**

**Martin Bodocky**

**Zadajte vas oblubeny dezert:**

**Pernik domaci**

**Mam vyborny Pernik domaci, ktory si zasluzi iba Martin Bodocky.**

**Všimnite si, že C++ povoľuje viacnásobné volanie funkcií, ak keď majú rôzny zoznam argumentov. Prečo by sme mali používať cin.get() namiesto cin.getline? Funkcia cin.get() má viacej možností použitia. Predpokladajme, že čítame riadok do poľa. Ako zistíme či ho prečíta funkcia celý alebo skončí pri naplnenom poli? Pozrieme sa na znak ďalšieho prvku, a ak to je znak konca riadku, tak ho funkcia prečítala celý. Ak iné, tak nám chýbajú ešte ďalšie dáta. Ale túto problematiku budeme preberať v neskorších častiach kurzu. Stručne povedané funkcia cin.getline() je jednoduchšia na použitie ale cin.get() zjednodušuje ošetrenie chýb. Na načítanie riadku môžete použiť obidve z týchto funkcií, ale musíte myslieť na ich rozdielne chovanie.**

## Prázdne riadky a ďalšie problémy

**Čo sa stane, keď funkcia cin.getline() alebo cin.get() prečíta prázdny riadok? Pôvodný postup bol taký, že nasledujúci príkaz vstupu začal tam, kde posledná funkcia vstupu skončila. Avšak v súčasnosti je po prečítaní prázdneho riadku funkciou cin.get() nastavený takzvaný chybový bit (failbit). Potom je ďalší vstup zablokovaný, a musíte ho odblokovať funkciou:**

**cin.clear();**

**Ďalšie problémy nastávajú, keď je reťazec dlhší ako blokovaný priestor. Ak je vstupný riadok dlhší, ako je stanovený počet obidve funkcie cin.getline() aj cin.get() nechávajú ostatné znaky vo fronte. A funkcia cin.getline() navyše nastaví chybový bit a vypne ďalší vstup.**

## Zmiešanie reťazového a číselného vstupu

**Zmiešanie číselného a reťazového vstupu môže spôsobiť problémy. Urobíme si na to názorný program:**

**//retazce5.cc -- zmiesanie retazoveho a ciselneho vstupu**

**#include**

**using namespace std;**

**int main() {**

 **const int velkost=80;**

 **cout << "V akom roku si sa nastahoval?\n";**

 **;**

 **cin >> rok;**

 **cout << "Aku mas adresu?\n";**

 **char adresa[velkost];**

 **cin.getline(adresa,velkost);**

 **cout << "Rok nastahovania: "<< rok<< "\n";**

 **cout << "Adresa:"<< adresa << "\n";**

 **cout << "finito! ;-)\n";**

 **return 0;**

**}**

**Výstup:**

**V akom roku si sa nastahoval?**

**1987**

**Aku mas adresu?**

**Rok nastahovania: 1987**

**Adresa:**

**finito! ;-)**

**Nikdy nedostaneme príležitosť zadať adresu. Tento problém je spôsobený tým, že keď cin prečíta rok zanechá vo fronte generovanú adresu ENTER čiže znak konca riadku. Potom cin.getline() prečíta znak konca riadku a priradí prázdny reťazec do poľa adresa. Oprava spočíva v prečítaní a zahodení znaku konca riadku pred čítaním adresy. Môžeme to urobiť niekoľkými spôsobmi, aj volaním cin.get() bez argumentu. Volanie môžeme uskutočniť oddelene:**

**cin >> rok;**

**cin.get();**

**Alebo spoločne:**

**(cin >> rok).get();**

**Náš výstup po úprave:**

**V akom roku si sa nastahoval?**

**1987**

**Aku mas adresu?**

**kolejni 2, praha 5**

**Rok nastahovania: 1987**

**Adresa:kolejni 2, praha 5**

**finito! ;-)**

**Programy v C++ často používajú pre prácu z reťazcami ukazatele namiesto polí. Túto vlastnosť aj z ukazateľmi si preberieme už skoro.**

**Záver**

**Dúfam, že Vám bol tento výklad zrozumiteľný. Úloha je pochopiť teóriu.**

# [Základní implementace MySQL v C/C++](http://programujte.com/clanek/2009052300-zakladni-implementace-mysql-v-c-c/)

[**Matej Tomčík**](http://programujte.com/profil/13267-matej-tomcik/)[**Zdeněk Lehocký**](http://programujte.com/profil/118-zdenek-lehocky/)**15. 9. 2009       31 706×**

**Na začiatok je potrebné podotknúť, že v dnešnej dobe, v ktorej je komplikovaný ako samotný návrh, tak aj tvorba aplikácií, je využívanie uý navrhnutých a stabilných systémov veľkou výhodou. MySQL pozná skoro každý programátor, ktorý pracuje či už na vývoji alebo správe webovej stránky. V mnohých prípadoch je potrebné takéto technológie používať aj vo vývoji desktopových aplikácií. Rád by som teraz osvetlil základné kroky k implementácii MySQL do vášho projektu.**

**V prvom rade potrebujete vedieť, kde máte hľadať hlavičkové súbory MySQL, LIB súbory a samotnú libmysql.dll - srdce MySQL. Najlepším spôsobom je pri inštalácii serveru MySQL zvoliť možnosť inštalovania dodatočných developerských súborov (pri inštalácii zvolíte „Custom“ inštaláciu a vyberiete produkt „C Include Files / Lib Files“). Do priečinka sa nainštalujú taktiež podpriečinky „include“ a „lib“ („lib“ obsahuje „debug“ a „opt“). Následne je vhodné si tieto priečinky vložiť do vášho programovacieho prostredia. Ak používate Visual Studio, priečinok pre „include“ pridáte týmto spôsobom: *Project -> Properties -> Configuration Properties -> C/C++ -> General -> Additional Include Directories.* Následne si do svojho projektu pridáte hlavičkový súbor:**

**#include**

**; Pre vloženie statickej knižnice použijete obdobný spôsob: *Project -> Properties -> Configuration Properties -> Linker -> General -> Additional Library Directories* Ako som už naznačil, v priečinku „lib“ sa nachádzajú dva ďalšie, a to „debug“ a „opt“. Buď si vyberiete jeden z týchto priečinkov a potom použijete len „libmysql.lib“ alebo si v projekte necháte možnosť využiť či debug knižnicu alebo release (teda použijete „debug/libmysql.lib“) alebo „opt/libmysql.lib“. Knižnicu do svojho projektu môžete pridať týmto preprocesorovým príkazom:**

**#pragma comment(lib,"libmysql.lib")**

**Alebo nastaviť priamo v možnostiach projektu: *Project -> Properties -> Configuration Properties -> Linker -> Input -> Additional Dependencies* Samotnú implementáciu budem komentovať postupne a začnem základným kameňom a to je typ MYSQL, resp. st\_mysql. S týmto typom sa budete stretávať pri používaní funkcií na inicializáciu, spojenie, nastavenie komunikácie, odosielanie dotazov atď. Deklarujeme si teda objekt tohto typu:**

**MYSQL mySQL;**

**Ďalším krokom je inicializácia tohto objektu. Urobíme to jednoduchou funkciou mysql\_init(MYSQL \*), ktorá ako argument preberá odkaz na štruktúru MYSQL. V popise tejto funkcie je jedna síce vhodná, no ale možno zbytočná vlastnosť. Ak ako argument použijete NULL, resp. 0, funkcia mysql\_init() vytvorí tento objekt automaticky a vráti jeho adresu. Platí teda, ak sa inicializácia podarí, funkcia mysql\_init() vracia ukazovateľ na objekt MySQL (v prípade, že argument MYSQL \* použijete, funkcia vráti ukazovateľ použitý ako argument), v opačnom prípade je návratová hodnota 0. My teda objekt inicializujeme a výsledok otestujeme takto:**

**if (mysql\_init(&mySQL))**

**{…}**

**V druhom kroku potrebujeme nadviazať spojenie s MySQL serverom. Toto docielime použitím funkcie MYSQL \* mysql\_real\_connect(MYSQL \*, const char \* host, const char \* pouzivatel, const char \* heslo, const char \* databaza, unsigned int port, const char \* unix\_socket, unsigned long client\_flag). Prvým argumentom je ukazovateľ na náš objekt MYSQL, ktorý sme si inicializovali v prvom kroku. Ako druhý parameter použijeme názov hostu, či už ip adresu alebo názvové stvárnenie. Ďalší parameter je meno používateľa, ktorý bude na MySQL serveri prihlásený. K tomu potrebujeme ešte aj heslo, ktoré je štvrtým argumentom. Piatym je názov databázy a šiestym číslo portu. U MySQL je to 3306, ak použijete hodnotu 0, port bude nastavený na automatickú hodnotu. Posledné dva argumenty nás v tomto prípade zaujímať nemusia, ide o rozšírené nastavenia. Funkcia mysql\_real\_connect() vracia ukazovateľ na objekt spojenia. V prípade, že sa spojenie nadviazať nepodarí, je táto hodnota 0. V opačnom prípade je nastavená na hodnotu prvého argumentu, ktorý je tiež týmto typom. Teda v našom programe pokračujeme takto:**

**if (mysql\_real\_connect(&mySQL,"127.0.0.1","root","","databaza",3306,0,0))**

**{…}**

**Ako host používam adresu 127.0.0.1, mám pri tom spustený MySQL server. Meno používateľa je root, heslo žiadne. Názov databázy: databaza. Vo vašom prípade budú tieto argumenty s najväčšou pravdepodobnosťou odlišné. V programe ďalej pokračujeme spustením SQL dotazu. Najskôr si teda určíme samotný dotaz:**

**string dotaz = "SELECT \* FROM skusobna\_tabulka";**

**Použil som typ „string“, pretože jeho využiteľnosť je vyššia, ak by ste si chceli tento malý program upraviť. Teraz potrebujeme náš dotaz vykonať. Na toto slúži funkcia int mysql\_real\_query(MYSQL \*mysql, const char \* dotaz, unsigned long dlzka\_textu\_dotazu). Táto funkcia nám vykoná zvolený dotaz, resp. pokúsi sa vykonať a informuje nás o svojom výsledku pomocou návratovej hodnoty takto: ak vykonanie dotazu prebehlo, hodnota je 0. V opačnom prípade je hodnota nenulová.**

**if (mysql\_real\_query(&mySQL,dotaz.c\_str(),dotaz.length()) == 0)**

**{…}**

**Predpokladáme, že náš dotaz bol vykonaný, nech je akéhokoľvek typu. Budeme chcieť obdržať výsledok tohto dotazu, resp. koľko údajov bolo zmenených. Dostávame teda dve možnosti. Buď náš dotaz údaje iba upravoval, mazal a pod. alebo z DB vyberal dáta. Tieto dve situácie rozlíšime nasledovne:**

**MYSQL\_RES \* vysledok;**

**vysledok = mysql\_store\_result(&mySQL);**

**if (vysledok)**

**{**

 **// tento krok si popíšeme za chvíľu**

 **mysql\_free\_result(vysledok);**

**}**

**else**

**{**

 **if (mysql\_field\_count(&mySQL) == 0)**

 **cout << "Pocet pozmenenych riadkov: " << mysql\_affected\_rows(&mySQL) << endl;**

 **else**

 **cout << mysql\_error(&mySQL) << endl;**

**}**

**Výsledok MySQL je uložený ako typ MYSQL\_RES, resp. st\_mysql\_res. Tento obsahuje údaje ako stĺpce, dáta atď. Získame ho pomocou funkcie MYSQL\_RES \* mysql\_store\_result(MYSQL \*) a výsledok uvolníme pomocou mysql\_free\_result(MYSQL\_RES \*). Z horeuvedeného kódu vyplýva, že táto funkcia vracia ukazovateľ na objekt výsledku dotazu. Ak nie je možné výsledok obdržať, vrátená hodnota je 0. V tomto prípade sú možné dva scenáre. Dotaz spôsobil len zmenu údajov, nie ich výber z DB. Toto otestujeme použitím funkcie unsigned int mysql\_field\_count(MYSQL \*), ktorá vracia informáciu o o počte stĺpcov v údaji o výsledku. Ak je teda počet stĺpcov nulový, jedná sa o údaj, ktorý upravoval určité dáta, preto by mal výsledok dotazu obsahovať počet zmenených riadkov. Funkcia my\_ulonglong mysql\_affected\_rows(MYSQL \*mysql) nám tento údaj poskytuje. V prípade, že je počet stĺpcov väčší ako nula, t.j. výsledok by mal obsahovať dáta, je príčinou prázdneho výsledku nejaká chyba. Funkcia const char \*mysql\_error(MYSQL \*) vracia textový údaj o tejto chybe. Ďalej budem popisovať, ako obdržať dáta z výsledku. Všetky údaje sú uložené ako textové reťazce štandardne ukončené nulovým znakom. K týmto dátam je možné pristúpiť najlepšie cez funkciu MYSQL\_ROW mysql\_fetch\_row(MYSQL\_RES \* vysledok), ktorej návratová hodnota je typu MYSQL\_ROW, v prevedení je to char \* \* (teda pole ukazovateľov na ukazovatele textových reťazcov štýlu C (zakončenie nulovým znakom)). Riadkami môžeme prechádzať v cykle práve pomocou popísanej funkcie mysql\_fetch\_row().**

**unsigned int pocetStlpcov = mysql\_num\_fields(vysledok);**

**if (pocetStlpcov > 0)**

**{**

 **MYSQL\_ROW riadok = 0;**

 **while ((riadok = mysql\_fetch\_row(vysledok)))**

 **{**

 **// „riadok“ je teraz pole textových hodnôt**

 **}**

**}**

**else**

 **cout << "Ziadne stlpce" << endl;**

**V prvom rade si funkciou unsigned int mysql\_num\_fields(MYSQL\_RES \* vysledok) získame počet stĺpcov z konkrétneho výsledku. Ak je tento počet väčší ako 0, v cykle while (), kde využívame mysql\_fetch\_row(), prejdeme celý obsah výsledku. Môžem upozorniť na prípady, kedy budete potrebovať vykonať viacero dotazov naraz. Váš kód bude musieť byť pozmenený hneď v niekoľkých častiach. Prvou zmenou je nastavenie možnosti používania viacerých MySQL dotazov naraz. Docielite to funkciou int mysql\_set\_server\_option(MYSQL \* mysql, enum enum\_mysql\_set\_option option), kde ako druhý argument použijete hodnotu MYSQL\_OPTION\_MULTI\_STATEMENTS\_ON. Túto funkciu je potrebné umiestniť za nadviazaním spojenia. Následne vykonáte dotaz, vytvoríte objekt MYSQL\_RES ako v uvedenom príklade, a nastavíte ho funkciou mysql\_store\_result(). Ďalej však už pokračujeme iným spôsobom, a to tak, že v cykle budeme testovať, či existuje nasledovný výsledok, ak áno, použijeme ho a proces sa opakuje. Viď príklad:**

**MYSQL\_RES \* vysledok;**

**vysledok = mysql\_store\_result(&mySQL);**

**while (vysledok)**

**{**

 **// s výsledkom môžete pracovať ako v predchádzajúcom príklade**

 **mysql\_free\_result(vysledok);**

 **if (mysql\_next\_result(&mySQL) == 0)**

 **vysledok = mysql\_store\_result(&mySQL);**

 **else**

 **break;**

**}**

**Nikdy však nesmiete zabudnúť na uvoľnenie výsledku pomocou mysql\_free\_result() pred tým, než budete chcieť obdržať ďalší výsledok. Prikladám zdrojový kód, kde som popísané postupy (okrem viac-výsledkového) spracoval do jednoduchého programu, ktorý sa pripojí na server 127.0.0.1:3306 s menom „root“ bez hesla, vykoná dotaz (buď zadaný priamo v kóde alebo zadaný ako príkazový argument) a zobrazí výsledok dotazu. Ak výsledok obsahuje dáta, tie sú zobrazené v zozname, kde je uvedený index zvoleného stĺpca, jeho názov a hodnota. Navyše, ak je údajovým typom dát textová hodnota, je zobrazená v apostrofoch (aby ste si všimli použitie atribútov stĺpcov).**

**Referencie:**[**Dokumentácia MySQL 5.1**](http://dev.mysql.com/doc/refman/5.1/en/index.html)

[http://dev.mysql.com/doc/refman/5.1/en/index.html](http://dev.mysql.com/doc/refman/5.1/en/index.html%20)

**Zdrojový kód programu, ktorý sa pripojí na MySQL server 127.0.0.1:3306 a načíta zoznam databáz. Program je však možné spustiť aj s parametrom, ktorý bude reprezentovať SQL dotaz. Pre prácu v určitej databáze je potrebné upraviť pripojenie (mysql\_real\_connect), aby sa program pripojil na existujúcu databázu.**

[**Matej Tomčík**](http://programujte.com/profil/13267-matej-tomcik/)

**Autor sa venuje programovaniu v jazykoch C#, C/C++, Delphi a v poslednej dobe sa taktiež zaujíma o vývoj hardvéru.**