Programovací jazyky

a

Stáhněte si sbalené stránky do jazyka C a C++ (jazyky.zip).

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

1. Vznik, vývoj a charakteristika

Časová náročnost: 5 minut

Programovací jazyk C vytvořil v Bellových laboratořích AT&T Denis Ritchie. Záměrem bylo napsat jazyk pro snadnou a přenositelnou implementaci Unixu. Na tomto přenosu (Unixu již existujícího pro PDP--7) se dále podíleli Brian Kernighan a Ken Thompson.

Jazyk C tedy původně vznikl jako systémový jazyk pro systémové programátory. Nicméně s postupem času se s C a Unixem seznamovali studenti na vysokých školách, odcházeli do praxe a začali používat C i v jiných než jen systémových oblastech. Uživatelé přirozeně požadovali, aby C bylo nejen mocným prostředkem (tím bylo od svého vzniku), ale i bezpečným a přehledným jazykem. Postupně se upouštělo od "fint". Vytvářel se základ normy.

Pojem **ISO norma jazyka C** definuje moderní vyšší programovací jazyk všeobecného použití. ISO je zkratkou slov International Standards Organisation. ISO norma jazyka C je bezpečnější, než byl původní jazyk K&R C. Současně je důležité, že většina zdrojových textů, napsaných před vydáním normy, je novými překladači přijímána.

Překladače C, nabízí řada nezávislých producentů programového vybavení. Dostupné jsou i verze volně (bezplatně) šířitelné. Podstatné však je, že C je k dispozici prakticky pro libovolné technické vybavení (hardware). Budeme-li ISO normu jazyka C dodržovat, máme velkou šanci, že budeme schopni přenést zdrojový text na jiný stroj (s jiným procesorem i OS), program přeložit, spojit s funkcemi z knihoven a konečně i spustit. Rozhodně by neměly vyvstat neřešitelné problémy.

S postupem doby se nezastavila ani teorie programování. Jestliže jazyk C umožňuje psát programy téměř modulárně, dnešní trend vyžaduje i podporu objektového přístupu. V roce 1986 publikoval Stroustrup knihu *The C++ Programming Language*, kde definuje nový programovací jazyk C++. Jedná se o objektové rozšíření jazyka C. Počátkem devadesátých let pak bylo i C++ normalizováno.

Ve druhé polovině devadesátých let je pouhá podpora objektového přístupu nepostačující, vzniká jazyk *Java*. Na rozdíl od C++ nepřejímá z C bezmyšlenkovitě vše a některé prvky dokonce záměrně nepodporuje, například ukazatele, nicméně opět staví na syntaxi jazyka C. Nicméně vývoj programování i vazby C++ a Javy na jazyk C jistě dokládají trvající užitečnost znalosti jazyka C. Použitelnost C je prokázána generacemi programátorů a desetiletími nasazení.

1.1 Krátké shrnutí

Časová náročnost: 3 minuty

Jazyk C

- je univerzální programovací jazyk nízké úrovně - což znamená, že pracuje "pouze" se standardními datovými typy, jako jsou znaky, celá a reální čísla, ...
- má velmi úsporné vyjadřování, je strukturovaný, má velký soubor operátorů a používá moderní datové struktury
- není specializován na jednu oblast používání
- pro mnoho úloh je efektivnější a rychlejší než jiné jazyky
- byl navržen a implementován pod operačním systémem UNIX a téměř celý UNIX je v C napsán
- C se na UNIX nijak neváže a neváže se ani na jiný konkrétní počítač či operační systém
- je vhodný pro rozsáhlé aplikace a přitom má blízko i ke strojové úrovni

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

 Obsah  

Obsah

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

Rejstřík

A

B

C

D

E

F

G

H

CH

I

J

K

L

M

N

S

Š

T

U

ukazatel	Čísla , Ukazatele , Ukazatele , Aritmetika ukazatelů , Ukazatele na funkce , Ukazatele na ukazatele a pole ukazatelů , Pole ukazatelů
#undef	Preprocesor
ungetc()	Proud a vstup a výstup znaků
union	Klíčová slova , Typ unie
Unix	Vznik, vývoj a charakteristika , Krátké shrnutí
unsigned	Klíčová slova , Čísla , Celočíselné konstanty

V

vertikální tabulátor	Odsazovače
void	Klíčová slova , Návratová hodnota funkce
volatile	Klíčová slova , Pamě•ové třídy
výraz	Výrazy
výraz aritmetický	Aritmetické výrazy
výraz logický	Logické výrazy

W

while	Klíčová slova , Cyklus while

Z

znak	Čísla , Znakové konstanty , Vstup a výstup

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

Glosář

A

ADRESA 1. Číslo označující umístění objektu v paměti počítače. 2. Adresa účastníka počítačové sítě.

ANSI Americká instituce, která vyvíjí americké průmyslové standardy ve shodě s mezinárodními standardy ISO.

APLIKACE Obecný pojem pro počítačový program či soubor programů, které plní daný účel. Pojem aplikace v mnohém splývá s pojmem program; obecně se program staví "níže", tedy do oblasti plnění menších, univerzálních a spíše pomocných funkcí, zatímco aplikace jsou větší programové balíky, plnící komplexnější úkoly. Aplikace se dělí podle oboru, druhu operačního systému, pod kterým pracují a někdy rovněž podle stupně komplexnosti.

AT&T American Telephone and Telegraph - Americká telefonní a telegrafní společnost - Založena v r. 1885, 1. 1. 1984 z rozhodnutí federálního soudu USA rozdělena na více nezávislých společností.

ASCII (Americký standardizovaný kód pro výměnu informací). Standardní sada znaků, definovaná a vytvořená v roce 1968. Definuje znaky, číslice a speciální písmena, použitelná v datové komunikaci. ASCII kód obsahuje 256 znaků, přičemž prvních 128 je zcela standardizováno prakticky na všech hardwarových i operačních platformách a dalších 128 je určeno pro další použití výrobci hardware či software. Velmi často druhých 128 znaků používá pro znaky lokálních abeced. Používá se dnes prakticky ve všech oborech výpočetní techniky.

B

BIT Základní jednotka informace vyjadřující dva stavy: ano - ne, pravda - nepravda, nízké napětí - vysoké napětí atd. Vyjadřuje se číslicemi 0 nebo 1. Pomocí nich lze tvořit čísla.

BYTE Jednotka informace složená z osmi bitů. Bajt je nejmenším adresovatelným prvkem v paměti počítače, přestože se lze odkazovat i na jeho jednotlivé bity.

BINÁRNÍ SOUSTAVA (binární soustava, dvojková číselná soustava) Polyadická poziční číselná soustava se základem 2. Stejně jako jsou ve známé desítkové soustavě čísla vyjadřována posloupnostmi číslic 0 až 9, jsou ve dvojkové soustavě užívány k vyjádření čísel pouze dvě číslice: 0 a 1.

BUFFER (vyrovnávací pamě•) Pamě•ový prostor pro přechodné uložení dat přesouvaných z rychlejšího pamě•ového média na pomalejší výstupní zařízení. Buffer plní úlohu mezičlánku zadržujícího přenášená data tak, aby je pomalé výstupní zařízení mělo okamžitě k dispozici a aby zároveň zdrojové médium nebylo zdržováno zdlouhavým přenosem. Je-li kapacita bufferu větší než objem přemís•ovaných dat, přesunou se všechna data do bufferu a zatížení zdrojového média je minimální. Vzhledem k tomu, že zdrojovým médiem je nejčastěji počítač (a cílovým tiskárna), je tato situace výhodná. Proto platí pravidlo: efektivita roste s velikostí bufferu, ne však nad určitou mez.

C

C

Programovací jazyk vyšší úrovně, vyvinutý Denisem Hallem v roce 1972 pro implementaci operačního systému UNIX. Kombinuje prostředky vyšších jazyků s možnostmi assembleru. Obsahuje základní malé jádro, které je závislé na použitém počítači. Další funkce jsou pak uloženy v knihovnách, dostupných pomocí jazyka C. Jazyk C má pečlivě propracovanou standardizaci, a je proto implementován na většině systémů. Díky svým možnostem a hutné syntaxi je oblíbený zejména mezi profesionálními programátory.

Č

ČÍSELNÁ SOUSTAVA

Druh zápisu dat pomocí hierarchické soustavy. Podstatný je základ číselné soustavy, což je číslo, které určuje maximální počet prvků jednoho řádu. Kromě běžné desítkové soustavy se vyskytuje soustava binární (základ 2), osmičková (základ 8) a šestnáctková (hexadecimální, základ 16).

ČÍSLO S PLOVOUCÍ ŘÁDOVOU ČÁRKOU

Číslo v počítači složené z celé a desetinné části. Je uloženo v pamě•ové buňce tak, že je umožněna změna jeho přesnosti - počtu desetinných míst v jistém rozsahu. Výhodou této interpretace desetinných čísel je vysoká přesnost při matematických operacích, nevýhodou jejich výpočtová složitost. K urychlení operací s plovoucí řádovou čárkou a pro práci s čísly s vysokým počtem desetinných míst slouží matematické koprocesory.

H

HARDWARE

Souhrn hmotných technických prostředků umožňujících nebo rozšiřujících provozování počítačového systému. Hardware je sám počítač, jeho komponenty (pamě•, (viz) základní deska s obvody, záznamová média, periférie, vstupně/výstupní zařízení, přídavné karty atd.), tiskárny, sítě, speciální zařízení. hardware je vše kromě programového vybavení (software). Na hranici mezi oběma skupinami leží firmware.

HEXADECIMÁLNÍ SOUSTAVA

Hexadecimální číselná soustava je číselná soustava se základem 16. Tomuto číslu odpovídá i počet cifer nutných k vyjádření čísla v hexadecimálním tvaru. Vzhledem k tomu, že běžně používaná desítková soustava má pouze deset cifer 0 až 9, musí hexadecimální soustava používat dalších šest cifer, za něž byly zvoleny písmena abecedy A až F. Skutečná hodnota jednotlivých hexadecimálních cifer je pro 0 až 9 totožná s desítkovou soustavou, nadstavené cifry A až F pak nabývají hodnot 10 až 15. Aby se odlišilo číslo zapsané v hexadecimálním tvaru neobsahující cifry A až F (a tedy na první pohled vypadající jako zápis decimálního čísla), používá se u hex. čísla speciální předpony nebo přípony (např. 10H, 10h, $0010 apod.). Výhodou hexadecimálního zápisu čísla v oblasti počítačového zpracování je skutečnost, že jím lze jednoduše vyjadřovat násobky hodnot bajtů. Např. čtyřbitové binární číslo 1110 (desítkově 14) lze vyjádřit jedinou hexadecimální cifrou E, tedy celý bajt (číslo s hodnotou v rozsahu 0 - 255) můžeme vyjádřit jako pouhé dvě hexadecimální cifry, dále pak slovo (číslo v rozsahu 0 - 65535 nebo podobně) lze vyjádřit čtyřmi hexadecimálními ciframi. Příklad: desítkové číslo 49534 se zapíše v hexadecimální soustavě jako C17E (tedy správněji C17Eh).

I

ISO

Mezinárodní autoritativní organizace pro zavádění celosvětových standardů. V této organizaci jsou zastoupeny všechny významné státy světa svými normativními ústavy; ISO definuje a přijímá standardy z celé průmyslové oblasti včetně výpočetní techniky a komunikací. Více informací na www.iso.ch.

J

JAVA Sí•ově orientovaný interpretovaný programovací jazyk vyvinutý firmou Sun Microsystems. Byl vyvinut pro používání v prostředí Internetu a umožňuje tvorbu malých programů zvaných applety, které je možné načíst z Internetu a okamžitě spouštět; obecně však lze v Javě vytvořit cokoli, od operačního systému po podnikový informační systém. Díky tomu, že je Java interpretovaný jazyk, může běžeet na jakékoli platformě (tj. může být na tuto platformu portována velice rychle). Pomalost daná touto technologií by měla být vyvažována tím, že mikrokód interpretující Javu by měl být "zadrátován" jednak ve speciálních čipech, jednak (snad) i v budoucích verzí procesoru Pentium, a tím se dosáhne běžně rychlého chodu aplikací v Javě. Více viz www.java.sun.com.

JEDNOUŽIVATELSKÝ SYSTÉM Systém zkonstruovaný pro používání jedním uživatelem. Opakem je víceuživatelský systém.

L

LEXIKÁLNÍ ANALÝZA První stupeň odlaďování při zpracování kódu v programovacím jazyce. Znaky zdrojového kódu jsou postupně čteny a seskupovány do funkčních celků (zvané lexemy či tokeny) - jsou to příkazy jazyka, identifikátory, citované řetězce atd. Tyto celky jsou pak předány parseru.

M

MAKRO Uživatelská definice posloupnosti více operací (např. stlačení kláves) nebo sekvence několika psaných příkazů. Makro se spouští jako jediný příkaz a je vlastně zkratkou zjednodušující uživateli provádění zdlouhavých posloupností kroků jejich nahrazením nějakou globální funkcí, která dokáže jednotlivé kroky provést automaticky. Makro může vystupovat např. v jednoduché formě náhrady některých delších řetězců nebo textů kratšími alternativami, které se při zpracování příslušného úseku textu rozvinou do původní podoby; makra se uplatňují také v aplikacích pro nahrazení posloupnosti opakovaných činností stlačením jediné kombinace kláves nebo např. v tabulkovém procesoru může makro nahradit definici složitějšího matematického výrazu.

MANTISA 1. Z matematického hlediska jde o část (viz) dekadického (viz) logaritmu čísla za desetinou tečkou. Např. dekadický logaritmus čísla 50 je přibližně 1.6989, odtud mantisa zaokrouhlená na čtyři desetinná místa odpovídá číslu 0.6989. 2. V počítačové terminologii je mantisa součástí popisu čísla s (viz) plovoucí desetinnou čárkou . Např. číslo 521 000 se v této notaci vyjádří jako 5.21 E+05, kde 5.21 je mantisa a E+05 je exponent (znamená 10^5).

O

OPERAČNÍ SYSTÉM Podstatné softwarové vybavení počítače, které provádí základní řízení veškerých zdrojů počítače a komunikaci s uživatelem - je nenahraditelným rozhraním mezi počítačem (hardware) a buď uživatelem přímo nebo dalšími programy. Bez operačního systému není možné počítač používat - veškeré příkazy uživatele jsou operačním systémem přijímány a zpracovávány, a rovněž veškeré programy využívají ke své činnosti služeb operačního systému. Operační systém je přítomen na pevném disku počítače a načítá se do paměti při startu počítače - tomuto procesu se říká bootování.

P

PROCEDURA Část programu, která je tvořena pojmenovanou posloupností příkazů. Díky své ucelenosti a pojmenovanosti může být procedura volána svým jménem jinou procedurou nebo hlavním jádrem programu. Procedura může rovněž přebírat parametry a vracet návratové hodnoty.

PROGRAM Ucelený souhrn instrukcí (příkazů), pomocí kterých provádí počítač určitou činnost. Program je tvořen souborem nebo více soubory, které jsou v úhrnu dostatečně schopné provádět předepsanou činnost. Příbuznými termíny, mezi kterými lze těžko vymezit ostrou hranici, jsou: - aplikace , čímž se označuje obvykle komplexnější souhrn často i několika programů, které plní úkoly dané oblasti - software , čímž se označuje jakékoli programové vybavení počítače, které je ucelené spíše svým vnějším zjevem.

PŘEKLADAČ Programový prostředek určený pro převod posloupnosti příkazů (zdrojového kódu programu) do přímo a samostatně spustitelné podoby. Programovací jazyky jako C++, Pascal, Modula a další jsou navrženy jako kompilátory. Na druhé straně existují jazyky, které lze používat i v interpretované formě (např. Basic).

S

SOFTWARE Obecně jakékoli programové vybavení. Oblast software zahrnuje programy od základních vstupně/výstupních systémů (BIOS), přes operační systémy a veškeré aplikace, od jednoduchých utilit až po komplexní programové systémy. Software je obecně série programových instrukcí, uložená v přirozených celcích (souborech) na záznamovém médiu či v paměti počítače. Software sám je vždy "nehmotný", ke svému šíření a používání vždy potřebuje hardware.

U

UNIX Víceuživatelský a multiprocesingový operační systém vyvinutý (zejména pány Thompsonem a Ritchiem) v laboratořích firmy AT&T Bell v roce 1969 a poprvé komerčně použitý v počítačích PDP firmy Digital. Většina dalšího vývoje pak byla provedena na univerzitě v Berkeley, čímž vznikla edice UNIXu nazvaná BSD (Berkeley SOFTWARE Distribution) UNIX. Další vývoj UNIXu přebírá firma UNIX System Laboratories (USL), kterou pohltila později firma Novell, která ji však později předala nezávislé organizaci X/Open. Existuje široká škála implementací (forem, podob) operačního systému UNIX. Všechny však mají přibližně shodnou vrstevnatou strukturu, založenou na silném jádře. Přínos operačního systému UNIX je mj. v tom, že programy jsou v rámci jednotlivých implementací přenositelné, nebo• jsou obvykle programovány v jazyce C. Slovo UNIX se vyvinulo z výrazu Uniplexed Information and Computing System, což byl jakýsi oponent neúspěšného všezahrnujícího systému Multics, Multiplexed Information and Computing System. Ze slova Unics se pak vyvinul název UNIX.

V

VÍCEUŽIVATELSKÝ SYSTÉM Počítačový systém schopný zpracovávat požadavky více uživatelů současně. Může a nemusí jít vždy o systém na bázi počítačové sítě. Počítač, na němž pracuje střídavě více uživatelů v průběhu delšího časového intervalu, bývá považován spíše za jednouživatelský systém.

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

Vysvětlivky

Na této stránce uvádím strukturu stránek, typografickou konvenci a seznam použitých značek se kterými se můžete setkat na následujících stránkách.

Struktura stránek

- Na začátku každé kapitoly je uváděna časová náročnost kapitoly zaokrouhlená na desítky minut:

  Časová náročnost kapitoly: 1 hodina 10 minut
- Poté následuje stručný popis látky, která bude v kapitole probírána.
- Dále časové náročnosti jednotlivých částí kapitoly:

  Časová náročnost: 15 minut
- V části

  ### Potřebné znalosti

  jsou uváděny základní znalosti nutné k bezproblémovému zvládnutí látky probírané v této kapitole.
- Následuje probíraná látka ukončená částí nazvanou:

  Cvièení

  ve které jsou uváděny příklady (a samozřejmě i jejich řešení) pro procvičení právě probrané látky.
- V místech, kde je vhodné zopakovat (připomenout) si některé aspekty probírané látky, následuje část nazvaná:

  ### Opakování

Typografické konvence

V textu této elektronické publikace můžeme nalézt velké množství **příkladů**. Každý příklad je uveden slovem:

Příklad:

Pokud je tento příklad určen ke stažení (není to tedy jenom fragment) je uveden také svým **pořadovým číslem**:

Příklad 4.1:

Aby došlo k jednoznačnému odlišení příkladu od ostatního textu - je příklad **barevně zvýrazněn**:

```
int i,j,k;
i = j = k = 2;
```

Pokud po zdrojovém kódu nějakého programu následuje jeho funkční výpis je zapsán **neproporcionálním písmem**:

```
1 << 1 =        2       0x2
1 << 7 =        128     0x80
```

Protože je nutné zdůraznit i chyby, které se často vyskytují, jsou **chybné příklady** uvozeny následujícím způsobem:

Chybný příklad:

Použité značky

Šipka ukazující vpravo slouží k přechodu na **další** stránku (kapitolu) v probírané látce.

Šipka ukazující vlevo slouží k přechodu na **předcházející** stránku (kapitolu) v probírané látce.

Šipka ukazující nahoru slouží k přechodu na **vrchol** (top) aktuální stránky.

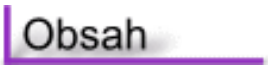
Šipka ukazující dolů slouží k přechodu na **patu** aktuální stránky.

Symbol kužele upozorňuje na ty části textu, které vyžadují zvýšenou pozornost. Zpravidla jsou takto označována místa s **důležitými informacemi**.

Symbolem diskety (floppy disku) jsou označeny výpisy zdrojových textů určených **ke stažení**. V případě Jazyka C půjde o stažení souborů s příponou *.c, v případě Jazyka C++ půjde o stažení souborů s příponou *.cpp.

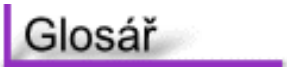
Symbolem tužky je označovaná část **cvičení**. V této části jsou uváděny příklady pro samostudium, na kterých si je vhodné vyzkoušet správné pochopení probírané látky.

Symbol písmene C nás informuje o tom, že získáváme znalosti o **Jazyku C**. Máme přece rádi Jazyk C, tak pro si to nepřipomenout pěknou ikonou? :-)

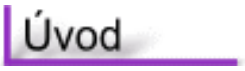


Značka **Obsah** nás přesouvá na obsahovou část této publikace. Můžeme si zde vybrat část která nás zajímá, případně se přesunout na Rejstřík, Glosář, Vysvětlivky či Úvodní stránku.

Značka **Rejstřík** nás přesouvá na část, která nám pomáhá se snadněji zorientovat v používaných příkazech a funkcích. Umožňuje nám nalézt konkrétní kapitoly (či jejich části), příklady ve kterých jsou příkazy či funkce použity.



Značka **Glosář** nám pomáhá se snadněji orientovat v používaných termínech, které se snaží jednoduchou formou vysvětlit.



Značka **Úvod** nás přesouvá na úvodní stránku této elektronické publikace. Zde si můžeme vybrat zda se budeme vzdělávat v Jazyku C či C++.

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

9. Ukazatele, pole a řetězce

Časová náročnost kapitoly: 1 hodina 50 minut

Nejprve trocha teorie. **Program** se skládá z instrukcí, také říkáme **kódu**, a **dat**. **Instrukce** musí procesor vykonávat, k **datům** pak bude přistupovat tehdy, když mu to instrukce nařídí. Teď musíme pouze určit, jak procesor pozná, kterou další instrukci má vykonat, kde najde potřebnou proměnnou, přesněji její hodnotu? Pozná to podle adresy. **Ukazatel** je název proměnné, která obsahuje adresu. Prostřednictvím této adresy **ukazuje** na data.

Adresa je klíčem pro procesor, kde má brát instrukce nebo data. Překladač vyššího programovacího jazyka, například jazyka C, za nás vykoná všechnu potřebnou práci, když se jedná o adresy instrukcí

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 8 - rozumět dříve probírané problematice jazyka C, chápat činnosti preprocesoru, provádění podmíněného předkladu a psaní maker.

9.1 Ukazatele

Časová náročnost: 8 minut

O ukazatelích jsme se už zmínili v kapitole 3 konkrétně v části **3.10**. Nyní se k problematice ukazatelů ještě vrátíme. Pro začátek trocha opakování.

Ukazatel představuje **adresu** objektu. Současně s sebou ukazatel nese i informaci o **datovém typu**, který se na dané adrese nachází. Ukazatel vytváříme stejně, jako proměnnou daného typu (nesmíme zapomínat, že ukazatel je spojen s datovým typem), jen před jeho jméno uvedeme symbol *** "hvězdička"**. Pak již stačí jen získat adresu proměnné (objektu), kterou chceme změnit nebo jen zjistit její hodnotu. K tomu použijeme **adresový operátor &**, uvedený opět před identifikátorem, tentokrát požadované proměnné.

A jak tedy změníme hodnotu ukazatele? K tomu vystačíme již se známou "hvězdičkou". Jí rozšířený ukazatel ovšem musí být **vlevo** od přiřazení.

Pár příkladů pro procvičení.

Příklad:

```
int x, y, *px, *p2x;    /* *px, *p2x jsou pointery (ukazatele) na int */

px = &x;                /* px nyní ukazuje na x */
*px = 5;                /* totéž co x = 5; */
y = *px + 1;            /* y = x + 1;  */
*px += 1;               /* x += 1; */
(*px)++;                /* x++;  závorky jsou nutné */
p2x = px;               /* p2x ukazuje na totéž, co px, tj. na x */
*p2x = *p2x + y;        /* x = x + y;  */
```

Při deklaraci ukazatele můžeme použít i modifikátor **const**. Pak se jedná o konstantní ukazatel, ukazatel na konstantu, nebo obojí, tedy konstantní ukazatel na konstantu. Tyto konstrukce se zejména používají při tvorbě nebo použití knihovních funkcí, kdy je například zdůrazněno (a překladačem kontrolováno), že argument funkce nebude v žádném případě modifikován.

Opět pár příkladů.

Příklad:

```
int i;
int *pi;                  /* pi je neinicializovaný ukazatel na typ int */
int * const cp = &i;      /* konstantní ukazatel na int */
const int ci = 7;         /* celoèíselná konstanta */
const int *pci;           /* neinicializovaný ukaz. na celoè. konst. */
const int * const cpc = &ci;   /* konst. uk. na konstantu  */
```

9.2 Pole

Časová náročnost: 20 minut

V jazyku C je jednorozměrné pole seznamem proměnných, které jsou všechny stejného typu a přistupuje se k ním přes společné jméno. Jednotlivá proměnná v poli se nazývá **prvek pole**. Pole umožňuje pohodlně zpracovávat skupinu souvisejících dat.

Pro deklaraci pole se používá obecný formát:

```
typ jméno-promìnné[velikost]
```

kde:

- **typ** je platný datový typ jazyka C
- **jméno-proměnné** je identifikátor pole
- **velikost** je kladný počet prvků pole, tj. celočíselný konstantní výraz, který musí být vyčíslitelný za překladu. Typicky se jedná o makro - konstantu, nebo přímo zapsaný počet prvků pole. Poslední možnost však nevidíme příliš rádi.

Klasický příklad deklarace celočíselného pole o 20 znaků.

Příklad:

```
#define ROZSAH 20

int pole[ROZSAH];
```

Prvek pole se získává indexováním pole pomocí čísla prvku. V jazyku C začínají všechna pole **nulovým** indexem. To znamená, že chceme-li pracovat s prvním prvkem pole, zadáme jako index nulu. Pro indexování pole zadáme index požadovaného prvku do hranatých závorek.

Příklad:

```
pole[0], pole[1], ... , pole[19]
```

Pamatujte, že pole začínají od nuly, takže index 1 označuje druhý prvek pole!!!

Pamě•ový prostor (velikost paměti) přidělený poli **pole** můžeme zjistit výpočtem, v němž počet prvků pole **ROZSAH** násobíme velikostí jednoho prvku, kterou zjistíme pomocí operátoru **sizeof**, jehož argumentem je požadovaný datový typ:

```
poèet obsazených byte = ROZSAH * sizeof(int)
```

Nebo naopak, **počet prvků pole**:

```
pole = sizeof(pole) / sizeof(int)
```

Jazyk C ukládá pole do souvislé oblasti paměti. První prvek pole má nejnižší adresu. Po provedení této části programu:

Příklad:

```
int i[5];
int j;

for(j=0; j<5; j++) i[j]=j;
```

Bude pole **i** vypadat takto:

```
i[0] hodnota 0
i[1] hodnota 1
i[2] hodnota 2
i[3] hodnota 3
i[4] hodnota 4
```

Jazyk C **netestuje** indexy polí, ale umožňuje současně s definicí pole provést i jeho **inicializaci**, tedy nastavení počátečních hodnot:

Příklad:

```

```

```
double cisla[5] = {1.22, 0.4, -1.2, 12.3, 4.0};
```

Princip je jednoduchý. Před středník, jímž by definice končila, přidáme "rovnítko" a do složených závorek vložíme seznam hodnot, jimiž chceme inicializovat prvky pole. Hodnoty pochopitelně musí být odpovídajícího typu. Navíc jich nemusí být stejný počet jako je dimenze pole. **Může jich být méně**. Přiřazení totiž probíhá následovně: první hodnota ze seznamu je umístěna do prvního prvku pole, druhá hodnota do druhého prvku pole, ... Pokud je seznam vyčerpán dříve, než je přiřazena hodnota poslednímu prvku pole, zůstávají odpovídající (zbývající) prvky pole neinicializovány. Výjimkou jsou **globální** a **statické** proměnné, které jsou inicializovány **nulou**.

Při inicializaci popsané výše jsme museli uvádět **dimenzi** pole. Tuto práci ovšem můžeme přenechat překladači. Ten poli vymezí tolik místa, kolik odpovídá inicializaci.

Příklad:

```
int pole[] = {21, 14, 55};
```

Překladač **nekontroluje** rozsah použitého indexu. Pokud se o tuto skutečnost nepostaráme sami, můžeme číst nesmyslné hodnoty při odkazu na neexistující prvky pole.

Po probrané teorii si práci s poli vyzkoušíme na několika příkladech.

Naším prvním úkolem bude naplnit pole **sqr** druhými mocninami čísel 1 až 10 a pak je vypsat.

Příklad 9.1:

```
#include <stdio.h>

#define VELIKOST 10

int main(void)
{
 int sqr[VELIKOST];
 int i;

 for(i=0; i<11; i++)
    sqr[i] = i*i;

 for(i=0; i<11; i++)
    printf("Druha mocnina cisla %d je %d\n",i,sqr[i]);

 return 0;
}
```

Naším dalším úkolem bude napsat program, který si bude v poli uchovávat zadávané teploty. Nakonec vypíše nejteplejší den (den s nejvyšší teplotou), nejstudenější den (den s nejnižší teplotou) a průměrnou měsíční teplotu.

Příklad 9.2:

```
#include <stdio.h>

#define NEJVICE 31

int main(void)
{
 int temp[NEJVICE];
 int i, dny, min, max, prum = 0;

 /* zjisteni poctu dnu */
 printf("Kolik dnu ma tento mesic? ");
 scanf("%d",&dny);

 /* nacteni teplot v jednotlivych dnech */
 for(i=0; i<dny; i++)
 {
  printf("Zadejte teplotu pro den %d: ",i+1);
  scanf("%d",&temp[i]);
 }

 /* zjisteni prumerne teploty */
 for(i=0; i<dny; i++)
    prum = prum + temp[i];
 printf("\nPrumerna teplota: %f",(float)prum/dny);

 /* nalezeni minimalni a maximalni teploty */
 min = max = temp[0];
 for(i=0; i<dny; i++)
 {
  if(min>temp[i]) min = temp[i];
  if(max<temp[i]) max = temp[i];
 }

 printf("\nMinimalni teplota: %d",min);
 printf("\nMaximalni teplota: %d",max);

 return 0;
}
```

9.3 Aritmetika ukazatelů

Časová náročnost: 17 minut

Jak jsme si již uváděli, ukazatel **ukazuje** na hodnotu nějakého typu. Nese s sebou informaci o tomto typu. Tato informace pochopitelně představuje počet bytů nutný pro uchování hodnoty typu. A při **aritmetice ukazatelů** smysluplně používáme obě části zmíněné informace, adresu i velikost položky.

Aritmetika ukazatelů je omezena na **tři základní operace**, **sčítání**, **odčítání** a **porovnání**. Nesmíme samozřejmě zapomenout na unární operace **inkrementace** a **dekrementace**, v daném kontextu a přehledněji řečeno jde o operace následovník a předchůdce. Podívejme se nejprve na první dvě operace.

Sčítání i **odčítání** jsou binární operace. Protože se zabýváme aritmetikou ukazatelů, bude jedním z operandů vždy ukazatel. Druhým operandem pak bude buď opět ukazatel (Týká se pouze operace odčítání dvou ukazatelů), nebo jím může být celé číslo. Pokud jsou oba operandy ukazatele, je výsledkem jejich rozdílu počet položek, které se mezi adresami, na něž ukazatele ukazují, nacházejí. Pokud k ukazateli přičítáme, respektive odčítáme celé číslo, je výsledkem ukazatel ukazující o příslušný počet prvků výše, respektive níže, představíme-li si prvky s vyšším indexem nad prvky s nižším indexem. Tak je ale pole v C definováno.

Příklad:

```
int i, *pi, a[N];
```

adresa prvku a[0] je &a[0], což je totéž jako a + 0. Tento výraz již představuje součet ukazatele s celočíselnou hodnotou. V našem případě je celočíselná hodnota rovna 0. Z toho vyplývá že platí i:

Příklad:

```
a + i <=> &a[i]
*(a+i) <=> a[i]
```

Pozn: znak <=> znamená ekvivalenci, což čteme jako "což je totéž".

Výrazy uvedené níže jsou po tomto přiřazení zaměnitelné (mají stejný význam, představují hodnotu prvku pole **a** s indexem **i**).

Příklad:

```
a[i] <=> *(a+i) <=> pi[i] <=> *(pi+i)
```

Chceme-li se například dostat na prostřední prvek u pole majícího 20 znaků, použijeme tento zápis s pomocí ukazatele **pi**:

Příklad:

```
pi = a + 9;
```

S tímto ukazatelem se pak můžeme posunout na následující prvek pole třeba pomocí inkrementace, nebo• i u ní překladač ví, o kolik bajtů má posunout (zvětšit) adresu, aby ukazoval na následující prvek: **++pi** nebo **pi++**, ale ne **++a** ani **a++**, nebo• **a** je konstantní ukazatel (je pevně spojen se začátkem pole).

A na závěr si opět ukážeme příklad. Funkce **printf()** umožňuje zobrazit adresu paměti obsaženou v ukazateli pomoci specifikátoru **%p**. Tuto schopnost můžeme použít pro

předvedení ukazatelové aritmetiky. Ukážeme si, jak ukazatelová aritmetika závisí na základním typu ukazatele.

Příklad 9.3:

```
#include <stdio.h>

int main(void)
{
 char *cp, c;
 int *ip, i;
 float *fp, f;
 double *dp, p;

 cp = &c;
 ip = &i;
 fp = &f;
 dp = &p;

 /* vytiskneme aktualni hodnoty */
 printf("%p %p %p %p\n", cp, ip, fp, dp);

 /* inkrementace */
 cp++;
 ip++;
 fp++;
 dp++;

 /* vytiskneme zmenene hodnoty */
 printf("%p %p %p %p\n", cp, ip, fp, dp);

 return 0;
}
```

Hodnoty obsažené v jednotlivých proměnných se mohou lišit v závislosti na překladači, ale přesto uvidíme, že adresa ukazující na **c** bude zvýšena o jeden byte. Ostatní budou inkrementovány o počet bytů jejich základních typů. V 16 bitovém prostředí to bude typicky 2 pro **int**, 4 pro **float** a 8 pro **double**.

9.4 Řetězce

Časová náročnost: 10 minut

Nejčastějším použitím jednorozměrného pole v jazyku C je **řetězec**. Na rozdíl od ostatních počítačových jazyků nemá jazyk C zabudován datový typ řetězec. Místo toho je řetězec definován jako **pole znaků ukončené nulovým znakem (null)**. V jazyku C má hodnotu nula. Skutečnost, že pole musí být ukončeno nulovým znakem znamená, že musíme nadefinovat pole, do něhož se vejde řetězec o jeden byte delší, než je požadovaný řetězec, tedy aby zbylo místo pro nulový znak. Řetězcová konstanta je ukončena nulovým znakem automaticky překladačem.

Příklad definice pole:

Příklad:

```
char retezec[SIZE];
```

Na identifikátor **retezec** můžeme nahlížet takto:

1. Jako na **proměnnou** typu řetězec (pole znaků doplněné o zarážku, kterou přidáváme někdy sami) délky SIZE, jehož jednotlivé znaky jsou přístupné pomocí indexů retezec[0] až retezec[SIZE-1].
2. Nebo jako na **konstantní ukazatel** na znak, tj. na první prvek pole retezec, retezec[0].

Řetězcové konstanty píšeme mezi dvojici uvozovek, uvozovky v řetězcové konstantě musíme uvodit speciálním znakem \, nazývaným "zpětné lomítko".

Příklad:

- **"abc"** - je konstanta typu řetězec délky 3+1 znak (zarážka), tedy tři znaky, čtyři bajty paměti (konstantní pole čtyř znaků)
- **"a"** - je řetězcovou konstantou, má délku 1+1 znak
- **'a'** - je znaková konstanta !!!!

Platí tedy, že následující zápisy jsou **ekvivalentí**:

Příklad:

```
char pozdrav[] = "ahoj";

char pozdrav[] = {'a','h','o','j',0};
```

Mějme nyní tuto definici:

Příklad:

```
char *ps = "retezec";
```

Jde o velice podobnou definici, ale přesto se podstatně liší. **ps** je ukazatel na znak, jemuž je přiřazena adresa řetězcové konstanty **retezec**. Tato konstanta je tedy umístěna v paměti, ale proměnná **ps** na ni ukazuje jen do okamžiku, než její hodnotu třeba změníme. Pak ovšem ztratíme adresu konstantního řetězce "retezec".

Následující příklad demonstruje práci s řetězci.

Příklad 9.4:

```
#include <stdio.h>

int main()
{
 char text[] = "world", *new1 = text, *new2 = text;
 printf("%s\t%s\t%s\n", text, new1, new2);
 /* menime hodnotu ukazatele, nedochazi ke kopirovani retezcu */
 new1 = "hello";
 printf("%s\t%s\t%s\n", text, new1, new2);
 printf("%s\n", "hello world" + 2);     /* posun retezce */

 return 0;
}
```

Výstup:

```
world   world   world
world   hello   world
llo world
```

Hodnota řetězcové konstanty je stejná jako hodnota jakéhokoliv ukazatele na řetězec (a nejen na řetězec) - **ukazatel na první položku**.

Při práci s řetězci nesmíme zapomínat na skutečnost, že jsou reprezentovány ukazateli. Pouhou změnou či přiřazením ukazatele se samotný řetězec **nezmění**.

9.5 Funkce pro práci s řetězci

Časová náročnost: 7 minut

Řetězce zřejmě budeme v našich programech používat velmi často. Naštěstí jsou již funkce pro práci s řetězci napsány a nemusíme si je psát sami. Jsou součástí standardní knihovny funkcí a jejich prototypy jsou obsaženy v hlavičkovém souboru **string.h**.

Syntaxe:

- *int strcmp(const char *s1, const char *s2);* - lexikograficky porovnává řetězce a vrací hodnotu:

```
<0 je-li s1 < s2
=0 je-li s1 = s2
>0 je-li s1 > s2
```

- *int strncmp(const char *s1, const char *s2, unsigned int n);* - totéž jako předchozí s tím, že porovnává nejvýše **n** znaků
- *unsigned int strlen(const char *s);* - vrátí počet významných znaků řetězce (bez zarážky)
- *char *strcpy(char *dest, const char *src);* - kopíruje **src** do **dest**
- *char *strncpy(char *dest, const char *src,unsigned int n);* - totéž jako předchozí, ale nejvýše **n** znaků (je-li jich právě **n**, nepřidá zarážku)
- *char *strcat(char *s1, const char *s2);* - **s2** přikopíruje za **s1**
- *char *strncat(char *s1, const char *s2, unsignedint n);* - totéž jako předchozí, ale nejvýše **n** znaků, **n** se týká délky **s2**, ne **s1**
- *char *strchr(const char *s, int c);* - vyhledá první výskyt (zleva) znaku **c** v řetězci **s**
- *char *strrchr(const char *s, int c);* - vyhledá první výskyt (zprava) znaku **c** v řetězci **s**
- *char *strstr(const char *str, const char *substr);* - vyhledá první výskyt (zleva) podřetězce **substr** v řetězci **str**

9.6 Vícerozměrná pole

Časová náročnost: 7 minut

Jazyk C umožňuje deklarovat pole pouze **jednorozměrné**. Jeho prvky ovšem mohou být libovolného typu. Mohou tedy být opět (například) jednorozměrnými poli. To však již dostáváme vektor vektorů, tedy **matici**. Budeme-li uvedeným způsobem postupovat dále, vytvoříme datovou strukturu prakticky libovolné dimenze. Pak již je třeba mít jen dostatek paměti. A operační systém který nám ji umí poskytnout.

Příklad definici matice:

Příklad:

```
type jmeno[4][5];
```

Kde:

- **typ** - určuje datový typ položek pole
- **jmeno** - představuje identifikátor pole
- **[4][5]** - určuje rozsah jednotlivých vektorů na čtyři řádky a pět sloupců

Grafické zobrazení takového dvourozměrného pole včetně indexů jednotlivých prvků je následující:

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4

S vícerozměrným polem můžeme pracovat stejně jako s jednorozměrným. To se přirozeně týká i možnosti inicializace. Jen musíme jednotlivé "prvky" vektory, uzavírat do složených závorek. Tento princip je ovšem stejný jako pro vektor.

Poslední index se mění nejrychleji (tak je vícerozměrné pole umístěno v paměti).

Nyní si na příkladu ukážeme jak vypsat dvou rozměrné pole 4x5, které bude vyplněno součiny indexů.

Příklad 9.5:

```
#include <stdio.h>

int main()
{
 int pole2[4][5];
 int i,j;

 for(i=0; i<4; i++)
 {
    for(j=0; j<5; j++)
        {
           pole2[i][j] = i*j;
         }
 }

 for(i=0; i<4; i++)
 {
    for(j=0; j<5; j++)
        {
           printf("%d ", pole2[i][j]);
        }
        printf("\n");
 }

 return 0;
}
```

9.7 Ukazatele na funkce

Časová náročnost: 10 minut

Jedná se často o ukazatele nebo o pole ukazatelů. Funkce ovšem nejen vracejí hodnotu jistého typu, ale mohou mít i různý počet argumentů různého typu.

Definujme si například ukazatel na funkci, která nemá argumenty a vrací hodnotu zvoleného datového typu:

Příklad:

```
typ (*jmeno)();
```

Závorky okolo identifikátoru **jmeno** a úvodní hvězdičky jsou nutné, nebo• zápis

```
typ *jmeno();
```

představuje funkci vracející ukazatel na typ.

Příkladem použití ukazatelů na funkce může být knihovní funkce **qsort()**.

Syntaxe:

```
void qsort(void *base, size_t nelem, size_t width,
int (*fcmp)(const void *, const void *));
```

Kde:

- **base** - je začátek pole, které chceme setřídit
- **nelem** - je počet prvků, které chceme setřídit (rozsah pole)
- **width** - je počet bajtů, který zabírá jeden prvek
- **fcmp** - je ukazatel na funkci, provádějící porovnání, ta má jako argumenty dva konstantní ukazatele na právě porovnávané prvky (nutno přetypovat).

Pozn.: Prototyp funkce **qsort()** je umístěn v hlavičkovém souboru **stdlib.h**.

Funkce **qsort()** je napsána obecně a tak nemá žádné konkrétní informace o typech hodnot, které třídí. Tuto část řeší naše uživatelská funkce, která správně porovnává hodnoty.

Příklad 9.6:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

#define POCET 10000
#define RND_START 1234

int float_sort (const float *a, const float *b)
{
 return (*a - *b);
}

void test (float *p, unsigned int pocet)
/* otestuje zda dane pole je setrideno vzestupne */
{
 int chyba = 0;
 for (; !chyba && --pocet > 0; p++)
   if (*p > *(p+1))
     chyba=1;
 puts ((chyba) ? "\npole neni setrideno\n" : "\npole je setrideno\n");
}

void vypln(float *p, int pocet)
/* vypni pole dane adresou a pocte prvku,
nahodnymi cisly pomoci generatoru nahodnych cisel */
{
 srand(RND_START);
 while (pocet-- > 0)
    *p++ = (float) rand();
}

int main (void)
{
 static float pole [POCET];

 vypln (pole, POCET);
 clock(); /* zacatek pocitani casu */
 qsort(pole, POCET, sizeof(*pole),
 (int (*) (const void *, const void *)) float_sort);
 /* uplynuly cas v ticich preveden na sekundy */
 printf ("Trideni qsort trvalo %.2fs\n",
 ((float) clock()) / CLOCKS_PER_SEC);
 test (pole, POCET);

 return 0;
}
```

Možný výstup:

```
Trideni qsort trvalo 0.26s

pole je setrideno
```

9.8 Ukazatele na ukazatele a pole ukazatelů

Časová náročnost: 12 minut

Vzhledem k tomu, že ukazatel je proměnná jako každá jiná. můžeme na ni ukazovat nějakým jiným (dalším) ukazatelem. Ve vícerozměrném poli provádíme v podstatě totéž, jen jednotlivé vektory nejsou pojmenované. Přistupujeme k nim pomocí **bázové adresy** základního pole a **indexu**. Z tohoto pohledu jsou opravdu nepojmenované. Lze říci, že pole ukazatelů je v jazyce C nejen velmi oblíbená, ale i velmi často používaná konstrukce.

Nejlépe si ukazatele procvičíme na příkladu. Tentokrát půjde o program, který lexikograficky setřídí jména měsíců v roce. Pro vlastní třídění jsme použili funkci **qsort**().

Příklad 9.7:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define N 13

int qsort_stringlist(const void *e1, const void *e2)
{
 return strcmp(*(char **)e1, *(char **)e2);
}

int main()
{
 char *name[N] =
  {
   "chyba", "leden", "unor", "brezen", "duben",
   "kveten", "cerven", "cervenec", "srpen",
   "zari", "rijen", "listopad", "prosinec"
  };
 int i;
 /* puts("jmena mesicu v roce (podle poradi):");
 for (i = 0; i < N; i++)
   printf("%2d. mesic je : %s\n", i, name[i]); */
 puts("\nsetrideno:");
 qsort(name, N, sizeof(char *), qsort_stringlist);

 for (i = 0; i < N; i++)
   printf("%2d. %s\n", i, name[i]);
 return 0;
}
```

Výstup:

```
setrideno:
 0. brezen
 1. cerven
 2. cervenec
 3. chyba
```

```
 4. duben
 5. kveten
 6. leden
 7. listopad
 8. prosinec
 9. rijen
10. srpen
11. unor
12. zari
```

Následující program používá dvojrozměrné pole ukazatelů pro vytvoření tabulky řetězců, která sdružuje odrůdy jablek s jejich barvami. Pro použití programu zadejte název odrůdy a program vypíše její barvu.

Příklad 9.8:

```
#include <stdio.h>
#include <string.h>

char *p[][2] = {
"Red Delicious", "cervene",
"Golden Delicious", "zlute",
"Winesap", "cervene",
"Gala", "oranzove",
"Lodi", "zelene",
"Mutsu", "zlute",
"Cortland", "cervene",
"Jonathan", "cervene",
"", "" /* ukonceni tabulky nulovymi retezci */
};

int main(void)
{
 int i;
 char jablko[80];

 printf("Zadejte jmeno odrudy jablka: ");
 gets(jablko);

 for(i=0; *p[i][0]; i++)
 {
  if(!strcmp(jablko, p[i][0]))
     printf("%s je %s\n", jablko, p[i][1]);
 }

 return 0;
}
```

Prohlédněte si pozorně podmínku řídící cyklus **for**. Výraz ***p[i][0]** nabývá hodnoty prvního bytu **i-tého** řetězce. Jelikož je seznam ukončen nulovým řetězcem, bude tato hodnota při dosazení konce tabulky nulová (nepravdivá). Ve všech ostatních případech bude nenulová a cyklus se bude opakovat.

9.9 Argumenty příkazového řádku

Časová náročnost: 10 minut

Mnoho programů umožňuje zadávat při jejich spuštění **argumenty na příkazovém řádku**. Argument příkazového řádku je informace, která následuje za jménem programu na příkazovém řádku operačního systému.

Samozřejmě i programy napsané v jazyce C mohou využívat argumenty příkazového řádku. Ty se předávají do programu pomocí dvou argumentů funkce **main**(). Tyto argumenty se nazývají **argc** a **argv** a jak jste již mohli odhadnout jsou tyto argumenty nepovinné.

Syntaxe:

```
int main(int argc, char *argv[])
```

Kde:

- **argc** - první argument funkce **main**(), tj. typicky **int argc**, udává **počet argumentů** příkazové řádky (jeden = jen jméno programu, dva = jméno programu + jeden argument, ...). Je celočíselného typu.
- **argv** - druhý argument funkce **main**(), tj. typicky **char *argv[]**, představuje **hodnoty argumentů** příkazového řádku. Jeho typ je pochopitelně pole ukazatelů na řetězce, nebo• jimi argumenty příkazového řádku skutečně jsou.

Pozn.: Je dobrým zvykem popsané argumenty funkce **main**() pojmenovat **argc** a **argv**, kde **arg** znamená argument, přípona **c** znamená **count** a přípona **v** znamená **value**.

Jazyk C nespecifikuje, co tvoří argument příkazového řádku, protože jednotlivé operační systémy se v tomto bodě od sebe značně liší. Nejčastější konvence zní: Každý argument příkazového řádku musí být oddělen mezerou nebo tabulátorem. Čárky, středníky a podobně nejsou považovány za oddělovač.

Příklad:

```
Toto je test
```

Je tvořen třemi řetězci, ale

Příklad:

```
Toto,je,test
```

je jeden řetězec.

Potřebujeme-li předat argument příkazového řádku který obsahuje mezery, musíme jej vložit do uvozovek, jak ukazuje následující příklad.

Příklad:

```
"Toto je test"
```

Nyní si napišme příklad, který vypíše své argumenty příkazového řádku.

Příklad 9.9:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 int i;
 for (i = 0; i < argc; i++)
   printf("%d\t%s\n", i, argv[i]);

 return 0;
}
```

Cvièení

1) Napište program s cyklem **for**, který počítá od 0 do 9 a vypisuje čísla na obrazovku. Čísla vypisujte pomocí ukazatele. Řešení

2) Ukažte jak deklarovat ukazatel na **double**. Řešení

3) Napište program, který přiřazuje hodnotu proměnné nepřímo, pomocí ukazatele na tuto proměnnou. Řešení

4) Co je špatně na tomto úseku programu?

```
#include <stdio.h>

int main(void)
{
 int i, count[10];

 for(i=0; i<100; i++)
 {
  printf("Zadejte cislo: ");
  scanf("%d",&count[i]);
 }
.
.
.
}
```

Řešení

5) Napište program, který načte deset čísel zadaných uživatelem a ohlásí zda jsou některá z nich shodná. Řešení

6) Co je špatně na tomto úseku programu?

```
int *p, i;
p = &i;
p = p * 8;
```

Řešení

7) Můžete k ukazateli přičíst neceločíselnou hodnotu? Řešení

8) Napište program, který přečte řetězec a pak jej vypíše na obrazovku pozpátku. Řešení

9) Co je špatně na tomto programu?

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 char str[5];

 strcpy(str, "tento text");
 printf(str);

 return 0;
}
```

Řešení

10) Napište program, který definuje trojrozměrné pole velikosti 3x3x3 a naplní jej čísly 1 až 27. Řešení

11) Napište program, který přebírá dva argumenty z příkazového řádku a zobrazte jejich součet. Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

8. Preprocesor

Časová náročnost kapitoly: 40 minut

Preprocesor zpracovává hlavičkové soubory, rozvíjí makra, nepropouští komentáře a umožňuje provádět podmíněný překlad zdrojového textu. Preprocesor předchází překladač. Preprocesor zpracovává vstupní text jako text, provádí v něm textové změny a jeho výstupem je opět text. Od preprocesoru tedy nemůžeme čekat kontrolu syntaxe, natož pak typovou kontrolu.

Preprocesor přijímá následující direktiva:

#define	#elif	#else	#endif
#error	#if	#ifdef	#ifndef
#include	#line	#pragma	#undef

Direktiva preprocesoru **není** příkaz jazyka C. **Neukončujeme** ji proto středníkem. Direktiva preprocesoru **musí** být vždy uvozena znakem **#**. **#** navíc **musí** být na řádku prvním jiným znakem než jsou odsazovače. Od direktivy samotné jej opět mohou oddělovat odsazovače.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 7 - pochopit doposud probíranou teorii. Umět psát programy umožňující vstup a výstup řádku a to jak formátovaných tak standardních. A umět také provádět vstupně výstupní operace v paměti.

8.1 Makra

Časová náročnost: 3 minuty

Makra se v programech využívají velice často a to proto, že zbavují programy tzv. "Magických čísel", tj. konstant, které se bez vysvětlení objevují v programu. Většinou jsou konstanty použity na začátku programu a jejich rozumné použití zvyšuje čitelnost zdrojového kódu.

Makra využíváme hlavně pro:

- definici konstant pro podmíněný překlad
- definici složitějších výrazů nebo příkazů, které se mají rozvinout v místě použití
- pro určení mezí polí, které musí být zapsány konstantním výrazem

Pokud se text makra nevejde na jeden řádek, můžeme jej rozdělit na více následujících řádků. Skutečnost, že makro pokračuje na následujícím řádku, se určí umístěním znaku \ jako posledního znaku na řádku.

8.2 Zabudovaná makra jazyka C

Časová náročnost: 6 minut

Pokud Váš překladač splňuje normu ISO C, bude mít nejméně pět předdefinovaných maker, která lze v programech použít. Jsou to:

__LINE__	definuje celočíselnou hodnotu, která je rovna číslu právě překládaného řádku zdrojového souboru
__FILE__	definuje řetězec, který je jménem právě překládaného souboru
__DATE__	definuje řetězec, který obsahuje systémové datum. Řetězec má obecný tvar *měsíc/den/rok*
__TIME__	definuje řetězec, který obsahuje čas zahájení překladu programu. Řetězec má obecný tvar *hodiny:minuty:sekundy*
__STDC__	je definováno jako hodnota 1, pokud překladač splňuje normu ISO C

Ukážeme si program, který předvádí výše popsaná makra.

Příklad 8.1:

```
#include <stdio.h>

int main(void)
{
 printf("Preklad: %s, radek: %d, dne: %s, v: %s", __FILE__, __LINE__,
  __DATE__, __TIME__);

 return 0;
}
```

Je důležité vědět, že hodnoty maker jsou pevně určeny v době překladu. Proto bude program vypisovat neustále stejné hodnoty bez ohledu na to, kdy bude spuštěn. Hlavní využití těchto maker je pro vytváření **časových a datumových razítek**, která ukazují, kdy byl daný program přeložen.

8.3 Podmíněný překlad

Časová náročnost: 10 minut

Preprocesor může během své činnosti vyhodnocovat, je-li nějaké makro definováno či nikoliv. Při použití klíčového slova preprocesoru **defined** pak může spojovat taková vyhodnocení do rozsáhlejších logických výrazů. Argument **defined** nemusí být uzavřen do závorek. Může se však vyskytnout jen za **#if** nebo **#elif**.

Příklad:

```
#if defined LIMIT && defined OSTRA && LIMIT==10
```

V závislosti na splnění či nesplnění podmínky můžeme určit, bude-li ohraničený úsek programu dále zpracován, nebo bude-li přeskočen a tak nebude přeložen. Této možnosti použití preprocesoru říkáme **podmíněný překlad**.

Direktivy které můžeme použít u podmíněného překladu:

- #if
- #else
- #elif
- #endif
- #ifdef
- #ifndef

Syntaktické definice popisující podmíněný překlad jsou:

```
#if text
[[#elif text] [#else text]]
#endif
```

Části zapsané v hranatých závorkách [] jsou **nepovinné**. Část označená jako **text** musí být vyhodnotitelná při činnosti preprocesoru

Vždy musí být jasno, kde podmíněná část zdrojového textu **začíná** a kde **končí**. Proto nesmíme zapomínat na **#endif** či **#elif**. Podmíněné části **musí** být ukončeny a omezeny v rámci jednoho zdrojového textu. Jinak oznámí preprocesor chybu. Dále je zde oproti C zavedena i podmínka **#elif**.

Napíšeme si program na kterém si demonstrujeme pravidla podmíněného překladu. Naším úkolem bude zobrazit samotnou ASCII sadu, nebo sadu rozšířenou na základě hodnoty obsažené v proměnné CHAR_SET.

Příklad 8.2:

```
#include <stdio.h>

/* Definujem CHAR_SET jako 256 nebo 128 */
#define CHAR_SET 256

int main(void)
{
 int i;
#if CHAR_SET == 256
 printf("Zobrazeni uplne sady ASCII znaku plus rozsireni.\n");
#else
 printf("Zobrazeni jen samotne sady ASCII znaku.\n");
#endif

 for(i=0;i<CHAR_SET;i++)
    printf("%c ",i);

 return 0;
}
```

8.4 Makro #include

Časová náročnost: 4 minuty

Makro **#include** je direktivou naprosto nepostradatelnou. Používáme ji pro včlenění zdrojového textu jiného souboru. Tento soubor může být určen více způsoby.

1. **#include <hlavičkový_soubor>** - hlavičkový_soubor je hledán ve standardním adresáři pro include. Takto se zpravidla začleňují standardní hlavičkové soubory. Není-li soubor nalezen, je ohlášena chyba.
2. **#include "hlavičkový_soubor"** - hlavičkový_soubor je hledán v aktivním (pracovním) adresáři. Není-li tam nalezen, postupuje se podle výše uvedené možnosti. Takto se zpravidla začleňují naše (uživatelské) hlavičkové soubory.
3. **#include jméno_makra** - jméno_makra je nahrazeno expandovaným makrem. Další činnost pokračuje podle některé z výše uvedených možností.

8.5 Makro #pragma

Časová náročnost: 1 minuta

Makro **#pragma** je speciální direktivou, která má uvozovat všechny implementačně závislé direktivy. Pokud jiný překladač speciální direktivu nezná, prostě ji bez chybového stavu ignoruje.

8.6 Opakování

Časová náročnost: 3 minuty

Preprocesor jazyka C

- Zpracovává zdrojový text programu **před** použitím překladače
- Nekontroluje syntaktickou správnost programu
- Provádí pouze zpracování maker - nahrazování konstant za odpovídající číselné hodnoty
- Vypouští ze zdrojového textu všechny komentáře
- Provádí podmíněný překlad

A dále:

- Všechny argumenty v definici makra by měly být uzavřeny do závorek
- Vyhýbejte se možnosti vzniku vedlejších efektů při vyhodnocení argumentů makra
- Každou použitou konstantu definujte jako symbolickou a to hned ze začátku programu - zvyšuje to jeho čitelnost
- Používejte podmíněnou kompilaci pro vynechání ladících částí programu
- Hlásí-li překladač nějakou chybu na kterou nemůžete přijít, je vhodné opět si prohlédnout soubor a zkontrolovat způsob, jakým preprocesor rozvíjí makra

Cvièení

1) Co je **__FILE__** a co představuje? Řešení

2) Pomocí **#ifdef** ukažte, jak podmíněně přeložit tento úsek programu v závislosti na tom, zda je nebo není definováno **DEBUG**.

```
if(!(j%2))
{
 printf("j = %d\n",j);
 j = 0;
}
```

Řešení

3) Je tento úsek správný? Pokud ne, ukažte jak jej opravit.

```
#define NUMBER

#ifdef !NUMBER
.
.
.
#endif
```

Řešení

4) Jaký je rozdíl mezi použitím uvozovek a úhlových závorek u direktivy **#include**?
Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

7. Vstup a výstup

Časová náročnost kapitoly: 1 hodina

V této části se budeme zabývat standardním textovým **vstupem** a **výstupem** a zaměříme se na možnosti jeho formátování. Každý program zpracovává nějaká vstupní data a sděluje nám výsledky touto činností získané. Pokud by tomu tak nebylo, neměli bychom zřejmě důvod takový program vůbec psát a spouštět. Vstup a výstup budeme obvykle zkráceně zapisovat **I/O** (Input / Output).

Některé základní pojmy:

- **Znak** je elementární textová informace. Znaku odpovídá jeho umístění v ASCII tabulce a číselný kód. Tisknutelné znaky mají kód v rozsahu 32 až 127. Mimo tento rozsah jsou umístěny buď znaky řídicí, viz například **escape sekvence**, nebo znaky národních abeced
- **Slovo** je posloupnost znaků. Slova jsou navzájem oddělena interpunkčními znaménky a odsazovači.
- **Řádek textu**je posloupnost slov ukončená symbolem (symboly) přechodu na nový řádek. Není zde žádná zmínka o délce řádku. Naše programy by se měly chovat "rozumně" pro vstupní řádky libovolné délky. Nicméně za "slušné" chování se obvykle pokládá přijetí řádku do délky nejvýše 512 znaků, jak činí mnohé unixové programy.
- **Filtry** jsou programy, které čtou ze (standardního) vstupu a zapisují do (standardního) výstupu. Jde tedy o přesměrování vstupu na výstup.

Každý program v jazyce C má při spuštění otevřen standardní vstup **stdin**, standardní výstup **stdout** a standardní chybový výstup **stderr**. Ty jsou obvykle směrovány na klávesnici a na terminál.

- Pro ukončení vstupu z klávesnice použijeme v prostředí Win32 kombinaci CTRL-Z, v Unixu CTRL-D.
- Standardní vstup a výstup používá vyrovnávací pamě• obsahující jeden textový řádek. Program tedy může vstup číst až v okamžiku, kdy ukončíme řádek klávesou ENTER.
- Při volání funkcí standardního vstupu či standardního výstupu musíme použít hlavičkový soubor **stdio.h**.

Vstup do programu provádíme pomocí vstupních funkcí. Tyto funkce nám umožňují zpracování vstupu buď opravdu po jednotlivých znacích, nebo po celých slovech či řádcích. Nejpokročilejší z možností vstupu je vstup formátovaný, kdy symbolicky určíme očekávaný formát vstupu a proměnnou, do níž má vstup proběhnout, a příslušná funkce takový vstup provede, nebo ohlásí chybu.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 6 - rozumět doposud probrané látce, umět vytvářet funkce, vědět jak funkcím předávat argumenty a jak z nich získávat hodnoty zpět. Umět vytvářet rekurzivní funkce a používat cizí funkce.

7.1 Standardní vstup a výstup znaků

Časová náročnost: 7 minut

Ikdyž jsou čísla důležitá, budou naše programy často potřebovat načítat z klávesnice také znaky. Pro vstup znaků používáme **getchar** a **putchar**, případně jim odpovídající volání funkcí pracujících se standardním vstupním a výstupním proudem - **getc(stdin)** a **putc(c, stdout)**.

Typ hodnoty, se kterou funkce pracují je nikoliv **char**, ale **int**. Důvodem je skutečnost, že ASCII tabulka obsahuje 256 znaků (přičemž přesně definuje dolních 128 z nich). A celkem 256 znaků je přesně tolik, kolik možností nabízí datový typ **char**. Vyčerpání všech kódových kombinací tohoto datového typu nedává odpovídajícím funkcím možnost pro zakódování řídicích informací. A proto je datový typ pro vstup i výstup znaků **int**.

Projděme si nyní syntaxi těchto příkazů:

```
int getchar(void);
```

Přečte ze standardního vstupu jeden znak, který vrátí jako svou návratovou hodnotu. V případě chyby vrátí hodnotu **EOF**.

```
int putchar(int c);
```

Zadaný znak zapíše na standardní výstup. Zapsaná hodnota je současně návratovou hodnotou, V případě chyby vrací **EOF**.

Nyní si použití právě probraných příkazů ukažme na příkladu. Naším úkolem bude načíst znak a vytisknout jeho ASCII hodnotu.

Příklad 7.1:

```
#include <stdio.h>

int main(void)
{
 char ch;

 ch = getchar();   /* nacitani znaku */
 printf("Zadali jste: %d",ch);

 return 0;
}
```

7.2 Standardní vstup a výstup řádků

Časová náročnost: 7 minut

Standardní vstup a výstup řádků je jednoduchou nadstavbou nad čtením znaků.

Syntaxe:

```
char *gets(char *str)
int puts(const char *str)
```

Tyto funkce používají hlavičkový soubor **stdio.h**. Funkce **gets()** čte zadávané znaky z klávesnice, dokud není načten znak "návrat na začátek řádku" (tj. dokud uživatel nestiskl klávesu **ENTER**). Přečtené znaky ukládá do pole **str**. Znak "návrat na začátek řádku" se k řetězci nepřidává. Místo toho je převeden na nulový ukončovací znak. Při úspěchu vrací **gets()** ukazatel na začátek **str**. Nastane-li chyba, vrací se nulový ukazatel.

Funkce **puts()** vypíše na obrazovku řetězec, na který ukazuje **str**. K řetězci se automaticky přidává sekvence "návrat na začátek řádku" a "nový řádek". Při úspěchu vrací **puts()** nezápornou hodnotu. Nastane-li chyba vrací se **EOF**.

Jednoduchost použití skrývá velké nebezpečí. Funkce **gets()** nemá informaci o délce oblasti vymezené pro čtený řetězec. Je-li oblast kratší, než vstupní řádek, dojde jeho načtením velmi pravděpodobně k přepsání pamě•ové oblasti související s vyhrazenou pamětí. A to se všemi důsledky z toho vyplývajícími. Situaci řeší funkce **fgets()**, která při volání vyžaduje informaci o velikosti cílového prostoru.

Správné pochopení funkcí **gets()** a **puts()** si prověříme na příkladu. Ukážeme si jak použít návratovou hodnotu funkce **gets()** pro přístup k řetězci obsahující zadané vstupní informace. Zároveň budeme také testovat, zda při zpracování **gets()** nedošlo k chybě.

Příklad 7.2:

```
#include <stdio.h>

int main(void)
{
 char str[80];

 printf("Zadejte retezec: ");
 if(gets(str))  /* kontrola zda nedoslo k chybe */
    printf("Nacteny retezec: %s", str);

 return 0;
}
```

7.3 Formátovaný standardní výstup

Časová náročnost: 12 minut

Pro formátovaný výstup používáme funkci **printf**(). Přesná syntaxe je následující:

```
int printf (const char *format [, argument, ...]);
```

První argument format určuje formátovací řetězec. Ten může obsahovat popis formátu pro každý argument, nebo text, který bude zapsán do výstupu. Popis formátu vždy začíná znakem **%**. Chceme-li znak **%** použít jako text, zdvojíme jej: **%%**. **Návratová hodnota** reprezentuje počet znaků zapsaných do výstupu, nebo **EOF** v případě chyby. Plné určení formátu je poněkud obsáhlejší:

```
[flags] [width] [.prec] [l|L] type_char
```

Určení **formátu** ve funkci **printf**():

Položka	Význam
flags	zarovnání výstupu, zobrazení znaménka a desetinných míst u čísel, úvodní nuly, prefix pro osmičkový a šestnáctkový výstup;
width	minimální počet znaků na výstupu, mohou být uvedeny mezerami nebo nulami;
.prec	maximální počet znaků na výstupu, pro celá čísla minimum zobrazených znaků, pro racionální počet míst za desetinnou tečkou;
l-L	*l* indikuje dlouhé celé číslo, *L* long double;
type_char	povinný znak, určuje datový typ konverze.

Určení **datového typu** položky ve funkci **printf**():

Symbol	Význam
d	desítkové celé číslo se znaménkem;
u	desítkové celé číslo bez znaménka;
x, X	šestnáctkové celé číslo, číslice ABCDEF malé (x) nebo velké (X);
f	racionální číslo (float, double) bez exponentu, implicitně šest desetinných míst;
e, E	racionální číslo (float, double) v desetinném zápisu s exponentem, implicitně jedna pozice před desetinnou tečkou, šest za ní. Exponent uvozuje e, respektive E.
g, G	racionální číslo (float, double) v desetinném zápisu s exponentem nebo bez něj (podle absolutní hodnoty čísla). Nemusí obsahovat desetinnou tečku (nemá-li desetinnou část). Pokud je exponent menší než -4 nebo větší než počet platných číslic, je použit.
c	znak;
s	řetězec.

Na několika příkladech si ukážeme možnosti funkce **printf**(). V prvním příkladu si ukážeme možnosti tisku s určitou přesností.

Příklad 7.3:

```
#include <stdio.h>

int main(void)
{
 printf("%.5d\n",10);
 printf("$%.2f\n",54.32);
 printf("%.10s", "Ne vsechno co je napsane se vytiskne\n");

 return 0;
}
```

Výpis programu pak vypadá následovně:

```
00010
$54.32
Ne vsechno
```

Na dalším příkladu si ukážeme možnosti formátování. Vypíšeme hodnotu 25 čtyřmi různými způsoby: desítkově, osmičkově, šestnáctkově malými písmeny a šestnáctkově velkými písmeny. Vytiskneme také číslo v semilogaritmickém tvaru a malým ´ e´ a s velkým ´ E´ .

Příklad 7.4:

```
#include <stdio.h>

int main(void)
{
 printf("%d %o %x %X\n", 25, 25, 25, 25);
 printf("%e %E\n", 25.123, 25.123);

 return 0;
}
```

Výpis programu pak vypadá následovně:

```
25 31 19 19
2.512300e+01 2.512300E+01
```

7.4 Formátovaný standardní vstup

Časová náročnost: 12 minut

Pro formátovaný standardní vstup použijeme funkci **scanf()** s následující syntaxí.

```
int scanf (const char *format [, address, ...]);
```

První argument je formátovací řetězec. Obsahuje podobné možnosti popisu formátu, jako u funkce **printf**().

Druhý argument je pamě•ovou oblast (adresa pamě•ové oblasti), do níž bude odpovídající vstupní hodnota uložena. V praxi jde nejčastěji o adresu proměnné, nebo o ukazatel na pole znaků.

Čtení ze vstupu probíhá tak, že první formátovací popis je použit pro vstup první hodnoty, která je uložena na první adresu, druhý formátovací popis je použit pro vstup druhé hodnoty uložené na druhou adresu, ...

Návratová hodnota funkce **scanf** nás informuje kladným celým číslem o počtu bezchybně načtených a do paměti uložených položek, nulou o nulovém počtu uložených položek a hodnotou **EOF** o pokusu číst ze vstupu více položek, než v něm bylo k dispozici.

Určení **datového typu** položky ve funkci **scanf**():

Symbol	Význam
%c	čtení jednoho znaku
%d, %i	čtení desítkového celého čísla
%e, %f, %g	čtení čísla s pohyblivou řádovou čárkou
%o	čtení osmičkového čísla
%s	čtení řetězce
%x	čtení šestnáctkového čísla
%p	čtení ukazatele
%n	uloží celočíselnou hodnotu, která se rovná počtu dosud načtených znaků
%u	čtení celého čísla bez znaménka
%[]	čtení vybrané sady znaků

Následující dva programy ukáží možnosti formátovaného vstupu. V prvním programu načítáme sadu znaků, do které patří pouze malá a velká písmena. Jestliže zadáme nějaké znaky, poté třeba číslice a poté opět nějaké znaky a stiskneme ENTER budou vytištěna jenom malá a velká písmena zadaná předtím, než jste stiskli klávesu s nějakým jiným znakem než je písmeno.

Příklad 7.5:

```
#include <stdio.h>

int main(void)
{
 char str[80];

 printf("Zadejte znaky: \n");
 scanf("%[a-zA-Z]",str);

 printf(str);

 return 0;
}
```

Další program umožňuje uživateli zadat číslo následované operátorem, za kterým následuje další číslo, například 5*6. Program pak provede zadanou operaci s čísly a zobrazí výsledek.

Příklad 7.6:

```
#include <stdio.h>

int main(void)
{
 int i,j;
 char op;

 printf("Zadejte operaci: ");
 scanf("%d%c%d",&i, &op, &j);

 switch(op)
 {
  case '+': printf("%d",i+j);
                          break;
  case '-': printf("%d",i-j);
                          break;
  case '*': printf("%d",i*j);
                          break;
  case '/': printf("%d",i/j);
                          break;
 }

 return 0;
}
```

7.5 Vstupní a výstupní operace v paměti

Časová náročnost: 8 minut

V paměti počítače můžeme provádět prakticky stejné operace, jako při standardním vstupu a výstupu. Jen musíme specifikovat vstupní respektive výstupní řetězec. Tyto operace se provádějí pomocí funkcí **sprintf()** a **sscanf()** s následující syntaxí.

```
int sprintf(char *buffer, const char *format[,argument, ...]);

int sscanf(const char *buffer, const char *format[,address, ...]);
```

Vstupní a výstupní řetězec je **buffer**, ostatní argumenty mají stejný význam jako u funkcí pro formátovaný standardní vstup a výstup.

A příklad na závěr. Máme za úkol vypsat úhly sinus a kosinus. Uživatel je vyzván, aby zadal mez, do kolika stupňů mají být obě funkce tabelovány. Ke zpracování vstupu a přípravě výstupu používáme dostatečně dimenzovaný řetězec.

Příklad 7.7:

```
#include <stdio.h>
#include <math.h>

#define N 80
const od = 0;

int main(void)
{
 char b[N],
      *zahlavi = "\n x\tsin(x)\t  cos(x)";
 int x, po;
 double si, co, rad;

 printf("\nZadej do kolika stupnu:");
 gets(b);

 sscanf(b, "%d", &po);
 puts(zahlavi);
 for ( x = od; x < po; x++)
   {
    rad = x / 180.0 * M_PI;
    si = sin(rad);
    co = cos(rad);
    sprintf(b, "%2d %10.5f %10.5f", x, si, co);
    puts(b);
   }
 printf("\nKONEC!\a\n");
 return 0;
}
```

Po výzvě uživatele je jím žádaný vstup načten jako řetězec do připraveného znakového pole. Z něj je vstup zpracován pomocí funkce **sscanf**. Podobně je realizován výstup v těle cyklu. Řetězec je nejprve pomocí funkce **sprintf** naformátován do stejného pole, které již pro vstup nepotřebujeme. Pak je na výstup zapsán připravený řetězec pomocí funkce **puts**.

Následuje možný výstup:

```
zadej do kolika stupnu:3

 x      sin(x)    cos(x)
 0    0.00000    1.00000
 1    0.01745    0.99985
 2    0.03490    0.99939
```

```
KONEC!
```

Cvièení

1) Napište program, který zobrazí ASCII kódy znaků **A** až **Z** a **a** až **z**. Jak se od sebe liší kódy velkých a malých písmen? Řešení

2) Je tento program správný? Pokud ne, proč?

```
#include <stdio.h>

int main(void)
{
 char p, *q;

 printf("Zadejte retezec: ");
 p = gets(q);
 printf(p);

 return 0;
}
```

Řešení

3) Napište program, který vypisuje tabulku čísel. Každý řádek obsahuje číslo a jeho druhou a třetí mocninu. Tabulka začíná číslem 2 a končí číslem 100. Řešení

4) Jak byste vypsali tento řádek pomocí funkce **printf**()?

```
Sleva: 40% puvodni ceny
```

Řešení

5) Ukažte jak zobrazit číslo 1023.231 tak aby se vypisovala jen dvě desetinná čísla. Řešení

6) Napište program, který načte desetinné číslo a pak zvláš• zobrazí jeho celou a desetinnou část. Řešení

7) Je tento úsek správný? Pokud ne, proč?

```
char ch;
scanf("%2c", &ch);
```

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

6. Funkce

Časová náročnost kapitoly: 1 hodina

Funkce může, ale také nemusí, vracet návratovou hodnotu. Funkce může, ale také nemusí, mít argumenty. Již tedy začínáme chápat, že základní stavební kámen C - **funkce** - poskytuje množství variant. V této části si doplníme znalosti o proměnných. Upřesníme si, jakým způsobem se vytvářejí a kam se umis•ují. Vzápětí poté si ukážeme využití těchto nových informací. Vytvoříme funkci, která volá sama sebe - **rekurzivní funkci**.

Funkce představují základní programovou jednotku, která řeší nějaký problém. Pokud je problém příliš složitý, volá na pomoc další funkci či funkce. Z toho plyne, že by funkce neměla být příliš rozsáhlá. Pokud tomu tak je, stává se funkce nepřehlednou i obtížně modifikovatelnou.

Každý C program obsahuje alespoň jednu funkci - **main**(). ISO norma jazyka C určuje návratovou hodnotu funkce **main**() typu **int**. Co obsahuje tělo této funkce je věcí každého programátora. Ale je jisté, že v ní musí být zapsána nejméně kostra programu. Překladač jazyka C totiž určuje, že právě funkcí **main**() začíná provádění každého programu. Tato vlastnost činí popisovanou funkci výjimečnou. Jinak se ovšem jedná o funkci, kterou prostě musí programátor napsat.

Na záčátku této kapitoly si ještě ukážeme obecný format programu s více funkcemi.

```
/* zde vloži hlavièkové soubory */

/* zde umístit prototypy funkcí */

int main(void)
{
 /* .. .. .. .. */
}

návratový-typ funkce1(seznam-argumentù)
{
 /* .. .. .. .. */
}


návratový-typ funkce2(seznam-argumentù)
{
 /* .. .. .. .. */
```

```
}

.
.
.

návratový-typ funkceN(seznam-argumentù)
{
 /* .. .. .. .. */
}
```

Funkce mohou mít samozřejmě jména. Položka **návratový-typ** představuje typ dat vracených funkcí. Nevrací-li funkce žádnou hodnotu měl by být její návratový typ **void**. Nepoužívá-li funkce argumenty, měl by její **seznam-argumentů** obsahovat klíčové slovo **void**.

Konkrétnější informace o funkcích si řekneme dále v textu.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 5 - chápat jednotlivé prvky programů, orientovat se v problematice řízení chodu programu - příkazy if-else a switch. Chápat rozdíly a použití cyklů for, while a do.

6.1 Vytváření a dokumentace vlastních funkcí

Časová náročnost: 1 minuta

Uživatelské funkce jsou funkce, které jsme napsali a máme jejich zdrojové texty. Je dobrým zvykem naše funkce precizně dokumentovat a archivovat. A začít můžeme komentáři ve zdrojovém textu. To vše patří k dobrému programátorskému stylu. Když už jsme funkci jednou napsali a odladili, můžeme ji příště s důvěrou používat. A právě dokumentace nám při použití pomůže.

6.2 Pojmenování funkcí

Časová náročnost: 5 minut

Identifikátor funkce je prostě **jméno**, pod kterým se v programech budeme na funkci odvolávat. Současně se jménem funkce určujeme i **datový typ návratové hodnoty funkce**.

Nejjednodušší způsob, jak vytvořit funkci, se nazývá **definice funkce**. Při **definici** funkci najednou **pojmenujeme**, určíme **typ** její návratové hodnoty, **typ** a **názvy** jejích argumentů a současně napíšeme **kód**, který bude při každém volání funkce proveden. Tento **kód** zapisujeme v **těle** funkce. Argumenty uvedené při definici funkce označujeme jako **formální argumenty**. Tyto formální argumenty zastupují v těle funkce argumenty, se kterými byla funkce volána. Těchto skutečných argumentů může být mnoho a jejich identifikátory se jistě budou lišit.

Syntaxe:

```
typ jméno(argumenty)
   {
    tìlo funkce
   }
```

Tělo funkce umístěno v **bloku**. O bloku jsme se zmínili dříve. Vime tedy, že může obsahovat lokální proměnné. Po nich následuje posloupnost příkazů. Ty definují chování (vlastnosti) funkce. Při definici funkce vytváříme (definujeme) její kód.

Voláním funkce - říkáme, že chceme provést kód, který je funkcí definován. Při volání funkce musí být za jménem funkce **vždy** uvedeny závorky. V nich pak mohou být uvedeny argumenty, které při volání funkci předáváme. Pokud závorky za jménem funkce neuvedeme, jde o **adresu funkce**. Adresou funkce rozumíme adresu vstupního bodu funkce. V místě, odkud funkci voláme, jí předáváme **skutečné argumenty**, které představují identifikátory proměnných a konstant, nebo výrazy.

```
jméno(argumenty);
```

6.3 Návratová hodnota funkce

Časová náročnost: 12 minut

Když jsme funkci pojmenovávali, určili jsme přitom typ její **návratové hodnoty**. V těle funkce pak musíme určit, jakým způsobem návratovou hodnotu získáme. Návratová hodnota funkce musí být výsledkem výrazu uvedeného jako argument příkazu **return**. Typ tohoto výrazu se musí shodovat, nebo být alespoň slučitelný, s deklarovaným typem návratové hodnoty funkce.

Syntaxe návratové hodnoty:

```
return výraz-vhodného-typu;
```

Na typ funkce, přesněji typ její návratové hodnoty, nejsou kladena žádná omezení. Pokud nás návratová hodnota funkce nezajímá, tak ji prostě po volání nepoužijeme. Děláme to často, například funkce **scanf()** a **printf()** hodnotu vrací a my ji ne vždy použijeme. Pokud ovšem chceme deklarovat, že funkce nevrací žádnou hodnotu, pak použijeme klíčové slovo **void**.

V této chvíli si vyzkoušíme, zda jsme informace o funkcích pochopili správně. Vytvoříme program který na obrazovku vypíše číslice **1 2 3**.

Příklad 6.1:

```
#include <stdio.h>

/* Prototypy funkci */
void funkce1(void);
void funkce2(void);

int main(void)
{
 funkce2();
 printf("3");

 return 0;
}

void funkce2(void)
{
 funkce1();
 printf("2");
}

void funkce1(void)
{
 printf("1");
}
```

V tomto programu volá funkce **main()** nejprve **funkce2()**, která volá **funkce1()**. Po

vyvolání **funkce1**() vypíše **1**, předá řízení zpět **funkce2**(), která vypíše **2** a řízení předá zpět funkci **main**(), která vypíše **3**.

Následující program vypíše druhou mocninu čísla zadaného z klávesnice. Druhá mocnina se získá pomocí funkce **mocnina**().

Příklad 6.2:

```
#include <stdio.h>

/* Prototyp funkce */
int mocnina(void);

int main(void)
{
 int mocnina_cisla;

 mocnina_cisla = mocnina();
 printf("Druha mocnina: %d",mocnina_cisla);

 return 0;
}

int mocnina(void)
{
 int cislo;

 printf("Zadejte cislo: ");
 scanf("%d",&cislo);

 return cislo*cislo;
}
```

Příkaz **return** se v těle funkce nemusí vyskytovat pouze jedenkrát. Pokud to odpovídá větvení ve funkci, může se vyskytovat na konci každé větve. Rozhodně však příkaz **return** ukončí činnost funkce, umístí návratovou hodnotu na specifikované místo - programátor se o umístění obvykle nestará - a předá řízení programu bezprostředně za místem, z něhož byla funkce volána. Činnost funkce, jejíž návratový typ **není void, musí** vždy končit příkazem **return**. Funkce typu **void** končí rovněž dosažením konce bloku, který tvoří tělo funkce.

6.4 Argumenty funkcí a způsob jejich předávání

Časová náročnost: 15 minut

Před každý formální argument musíme při deklaraci či definici funkce uvést jeho datový typ. Současně s tím určujeme i způsob, jakým se budou hodnoty **skutečných argumentů** předávat **argumentům formálním**. Dále platí, že pořadí formálních a skutečných argumentů si odpovídá. Tedy že první skutečný argument je v těle funkce zastupován prvním formálním argumentem, druhý druhým, atd. Přirozeně se vyžaduje, aby jejich datové typy byly stejné, nebo alespoň slučitelné. Přesná pravidla uvádí norma ISO C.

Podstatný je tedy **způsob** předávání argumentů funkci.

- Kopie hodnoty skutečných argumentů jsou předány do zásobníku. Formální argumenty se odkazují na odpovídající místa v zásobníku, kde jsou kopie argumentů skutečných. Díky tomu se změna hodnoty formálního argumentu nepromítne do změny hodnoty argumentu skutečného! Tomuto způsobu předávání argumentů se říká **předávání hodnotou**.

- Může ovšem nastat situace, kdy potřebujeme, aby funkce vrátila více než jednu hodnotu. Obdobná situace vzniká při předávání rozsáhlého pole. Představa kopírování jeho obsahu do zásobníku je strašná, a u rozsáhlých polí by mohla vést až k nestabilitě aplikace. Řešení pomocí globálních proměnných kategoricky vylučuje dobrý programátorský styl! Tvorba nového strukturovaného datového typu jako návratové hodnoty nemusí být vždy přirozeným řešením. Řešení je předávání nikoli hodnot skutečných argumentů, ale jejich **adres**. Formální argumenty pak budou ukazatele na příslušný datový typ. Důležitý je ovšem fakt, že se tyto ukazatele budou odkazovat na místa původních, skutečných argumentů. Tím se naskýtá možnost změny hodnot skutečných argumentů. V jazyku C se hovoří o **volání adresou**.

Nyní si ukážeme jednoduchý program, který volá funkce a předává jim argumenty jak hodnotou tak adresou.

Příklad 6.3:

```
#include <stdio.h>

int nacti(int *a, int *b)
{
 printf("\nZadej dve cela cisla:");
 return scanf("%d %d", a, b);
}

float dej_podil(int i, int j)
{
 return((float) i / (float) j);
}
```

```
int main(void)
{
 int c1, c2;
 if (nacti(&c1, &c2) == 2)
   printf("Podil je : %f\n", dej_podil(c1, c2));
 return 0;
}
```

Funkce **nacti()** má vracet dvě načtené hodnoty. K tomu nemůžeme použít její návratovou hodnotu. Předáváme jí proto argumenty adresou. Pozor na **&** adresový operátor před skutečnými argumenty při volání! Bez něj adresy nezískáme. Ještě si však všimneme **návratové hodnoty** této funkce. Tato návratová hodnota slouží k tomu, aby potvrdila platnost (respektive neplatnost) hodnot argumentů, které předává adresou.

Ještě si ukažme jeden příklad. Pěkným příkladem volání odkazem bude funkce **swap()**, která zamění dvě celočíselné hodnoty.

Příklad 6.4:

```
#include <stdio.h>

void swap(int *i, int *j);

int main(void)
{
 int num1=50, num2=20;

 printf("Cislo1: %d, Cislo2: %d\n",num1, num2);
 swap(&num1, &num2);
 printf("Cislo1: %d, Cislo2: %d\n",num1, num2);

 return 0;
}

void swap(int *i, int *j)
{
 int temp;

 temp=*i;
 *i=*j;
 *j=temp;
}
```

Jelikož jsou funci předány ukazatele na dvě celá čísla, zamění se hodnoty, na které tyto ukazatele ukazují.

6.5 Pamě•ové třídy

Časová náročnost: 7 minut

Při deklaraci proměnné můžeme nepovinně uvést i klíčové slovo, určující **pamě•ovou třídu**. Pamě•ová třída určuje překladači náš požadavek na umístění proměnné. Pokud pamě•ovou třídu neurčíme při deklaraci, platí implicitní pravidla pro určení pamě•ové třídy.

Pamě•ová třída	Výklad
auto	Umístění na zásobník, neinicializované
extern	Nevytvářet, bude připojen z jiného modulu
register	Umístit do registru procesoru, neinicializované
static	Umístění do datového segmentu, inicializované nulou
typedef	Nemá pamě•ový význam, pojmenovává konstrukci definice

Modifikátor	Význam
const	Vyjadřuje neměnitelnost
volatile	Vyjadřuje neustálou proměnnost - nekešovat

Pamě•ovou třídu můžeme upřesnit ještě tak zvanou **typovou částí**. Jedná se o klíčové slovo **const** a klíčové slovo **volatile**. První možnost určuje neměnnost, zatímco druhá deklaruje její opak. Jak to chápeme? Představme si dvě současně běžící úlohy se sdíleným pamě•ovým prostorem. Pokud úlohy sdílejí nějakou společnou proměnnou, kterou navíc mění, nemůže procesor jakkoli optimalizovat přístup k této proměnné.

Následující tabulka uvádí nejdůležitější pravidla pro implicitní určení pamě•ové třídy. Tedy ukazuje, jakou pamě•ovou třídu budou proměnné jazyka C.

Objekt	Pamě•ová třída, výklad, umístění
Globální proměnné	*static* a *extern*, inicializovány nulou, datový segment
Lokální proměnné	*auto*, neinicializovány, zásobník
Formální argumenty	*auto*, neinicializovány, zásobník
Definice funkce	*extern*, definice dle kódu, kódový segment

6.6 Rekurze

Časová náročnost: 7 minut

Rekurze je proces, ve kterém je něco definováno samo sebou. Aplikujeme-li to na počítačový jazyk, rekuze znamená, že funkce může volat sama sebe. Ne všechny programovací jazky podporují rekurzi. Jazyk C však ano.

Je důležité vědět, že neexistují vícenásobné kopie rekurzivní funce. Existuje pouze jediná. Když je volána funkce, vyhradí se v zásobníku místo pro její argumenty a lokální proměnné. Když je tedy funkce volána rekurzivně, začíná pracovat s novou sadou argumentů a lokálních proměnných, ale kód, který tvoří funkci, zůstává stejný.

Nejčastějším příkladem užití rekurze je výpočet faktoriálu. I my si ukážeme jak vypočíst faktoriál na základě rekurzivní funkce.

Příklad 6.5:

```
#include <stdio.h>

int fakt(int n)
{
 return (( n <= 0 ) ? 1 : n * fakt(n-1));
}

int main()
{
 int i;

 printf("\nZadej cele cislo: ");
 scanf("%d",&i);
 printf("Faktorial cisla %d je: %d",i,fakt(i));
 return 0;
}
```

Nyní si ukážeme ještě jeden jednoduchý program abychom rekurzi dostatečně porozuměli. Naším úkolem bude vypsat na obrazovku čísla od 9 do 0. A použijeme k tomu rekurzivní funkci.

Příklad 6.6:

```
#include <stdio.h>

void rekurze(int i);

int main(void)
{
 rekurze(0);

 return 0;
}

void rekurze(int i)
{
 if(i<10)
 {
  rekurze(i+1);
  printf("%d ",i);
 }
}
```

Krátké vysvětlení:

Nejprve je volána funkce **rekurze()** s argumentem **0**. Jelikož je **0** menší než **10** pak volá sama sebe s hodnotou **i + 1**. To způsobí opětovné vyvolání funkce **rekurze()** tentokrát s argumentem **1**. Tento proces se opakuje dokud není **rekurze()** volána s argumentem 10. To způsobí návrat z funkce **rekurze()**. Jelikož se funkce vrací na místo svého vyvolání, provede se příkaz **printf()** - vytiskne se číslo **9** a vrací se. Vrátí se do místa svojí předchozí aktivace a vytiskne **8**. Atd.

6.7 Cizí funkce

Časová náročnost: 2 minuty

Z běžného použití známe **standardní** funkce, **uživatelské** funkce, **podpůrné** a **nádstavbové** funkce. Nejjednodušší možné členění zní: **funkce standardní** a **funkce ostatní**.

Standardním funkcím se zpravidla říká **knihovní funkce**. Jejich deklarace je popsána ve standardních **hlavičkových souborech**. Deklaracím funkcí se někdy říká **prototyp**.

Norma ISO C **vyžaduje** prototyp každé funkce, kterou chceme použít. Tento požadavek výrazně zvyšuje bezpečnost - umožňuje typovou kontrolu. Proto musíme začleňovat hlavičkové soubory tehdy, když používáme standardní funkce - hlavičkové soubory obsahují jejich deklarace. U deklarací nevadí, zahrneme-li shodnou deklaraci funkce vícekrát.

6.8 Opakování

Časová náročnost: 5 minut

Krátké zopakování základů spravných pochopení funkcí.

Syntaxe deklarace funkce:

```
typ jméno(seznam argumentù);
```

kde:

- **typ** představuje typ návratové hodnoty
- **jméno** je identifikátor, který funkci dáváme
- ( ) je povinná dvojice závorek, vymezující deklaraci argumentů
- **seznam argumentů** je nepovinný - funkce nemusí mít žádné argumenty, může mít jeden nebo více argumentů, nebo také můžeme určit, že funkce má proměnný počet argumentů. Pokud je argumentů více, oddělujeme je navzájem čárkami. Každý argument musí mít samostatně určen datový typ.

Deklarace popisuje vstupy a výstupy, které funkce poskytuje, ale nedefinuje posloupnost příkazů, které má funkce vykonávat. Deklarace určuje **rozhraní funkce**. Funkce **nemá** provádět akce s jinými daty, než která jí předáme jako argumenty. Současně výstupy z funkce **mají** probíhat jen jako její návratová hodnota. Pokud se funkce nechová uvedeným způsobem říkáme, že má vedlejší účinky, efekty.

Pokud uvedeme pouze definici funkce, na kterou se později v souboru odvoláváme, slouží tato definice současně jako **deklarace**. **Definici** smíme uvést jen jednou. Při případné druhé definici by překladač nevěděl, která z nich je platná.

Deklarace funkcí se umis•ují do **hlavičkových souborů**. V případě neshody deklarace a definice funkce ohlásí překladač chybu.

Cvièení

1) Napište program, který bude mít alespoň dvě funkce a vytiskne **Co se v mládí naučíš, ve stáří zapomeneš**. Řešení

2) Napište program, který použije funkci **convert()**, která vyzve uživatele k zadání částky v korunách a převede ji na dolary (Použijte směnný kurs např. 40 Kč za dolar). Vypište převedenou částku v dolarech. Řešení

3) Co je v tomto programu chybné?

```
#include <stdio.h>

int f1(void);

int main(void)
{
 double odpoved;

 odpoved = f1();
 printf("%f",odpoved);

 return 0;
}

int f1(void)
{
 return 100;
}
```

Řešení

4) Co je chybné v této funkci?

```
void func(void)
{
 int i;
 printf("Zadejte cislo: ");
 scanf("%d", &i);

 return i;
}
```

Řešení

5) Je tento program správný? Pokud ne, proč?

```
#include <stdio.h>

myfunc(int num, int min, int max);

int main(void)
```

```
{
 int i;

 printf("Zadejte cislo mezi 1 a 10: ");
 myfunc(&i, 1, 10);

 return 0;
}

void myfunc(int num, int min, int max)
{
 do
 {
   scanf("%d", num);
 }
 while(*num<min || *num>max);
}
```

Řešení

6) Vysvětlete rozdíl mezi voláním funce hodnotou a voláním funce odkazem.
Řešení

7) Co je špatně v této rekurzivní funkci?

```
void f(void)
{
 int i;

 printf("in f() \n");

 /* volani f() 10krat */
 for(i=0; i<10; i++) f();
}
```

Řešení

8) Napište program, který vypisuje na obrazovku řetězec znak po znaku pomocí rekurzivní funkce. Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

5. Řízení vykonávání programu

Časová náročnost kapitoly: 1 hodina 40 minut

V této části si ukážeme, jak nechat vykonat jen část kódu, případně jak některou část kódu nevykonat vůbec. Také si ukážeme jak vybrat některou variantu, podle aktuální hodnoty přepínače. V neposlední řadě se naučíme několik způsobů, jak opakovaně vykonávat skupiny příkazů.

Všechny tyto naše zvolené záměry budou vykonány za použití **řídicích konstrukcí**. S jejich pomocí bude náš program provádět právě ty příkazy, které požadujeme.

K řízení chodu programu máme k dispozici tyto příkazy:

1. Výrazový příkaz
2. Blok
3. Přepínač
4. Cyklus
5. Skok

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 4 - umět napsat jednoduché programy s pomocí výrazů. Orientovat se v problematice operátorů a znát jejich prioritu.

5.1 Výrazový příkaz

Časová náročnost: 1 minuta

Výrazem je nejen aritmetický výraz, prostý výskyt konstantní hodnoty, konstanty či proměnné, ale i volání funkce a přiřazení. Jestliže výraz ukončíme symbolem **;** (středník), získáme **výrazový příkaz**. Zkráceně mu budeme i dále říkat **příkaz**.

5.2 Prázdný příkaz

Časová náročnost: 1 minuta

Prázdný příkaz je tvořen samotným středníkem. Dává nám v některých případech možnost, umístit nadbytečný středník ; do zdrojového textu.

5.3 Blok příkazů

Časová náročnost: 5 minut

V jazyku C můžeme spojit dva nebo více příkazů dohromady. Nazýváme to **blok příkazů**, nebo **složený příkaz**. Pro vytvoření bloku příkazů ohraničte příkazy v bloku otevírací a uzavírací složenou závorkou. Když to uděláte, budou příkazy v bloku představovat jednu logickou jednotku, kterou lze použít všude tam, kde lze použít jeden příkaz.

Blok příkazu si nejlépe ukážeme na často používaném příkazu **if**.

Příklad:

```
if(výraz)
{
 pøíkaz1;
 pøíkaz2;
    .
    .
    .
 pøíkazN;
}
else
{
 pøíkaz1;
 pøíkaz2;
    .
    .
    .
 pøíkazN;
}
```

Pokud je výraz vyhodnocen jako pravdivý provedou se všechny příkazy uvedené v bloku za **if**. Budou prováděny v **pořadí** v jakém byly **zapsány**. Je-li výraz nepravdivý provedou se všechny příkazy bloku následující za **else**.

Blok může obsahovat nejen **lokální proměnné**, ale obecně v něm mohou být libovolné **lokální deklarace a definice**. Ovšem pouze na začátku bloku. Jejich platnost je omezena na blok a případné další vnořené bloky. Lokální proměnné v bloku mají pamě•ovou třídu **auto**. Vnořený blok **nemusí** být ukončen středníkem. Je vhodné si uvědomit, že **tělo**

každé funkce je blokem. Proto můžeme v těle funkce **main(void)** definovat a používat lokální proměnné. Blok tedy může obsahovat **žádnou**, **jednu** či **více deklarací**. Dále blok může obsahovat jednotlivé příkazy.

5.4 Oblast platnosti identifikátorů

Časová náročnost: 4 minuty

Identifikátor, který deklarujeme či definujeme, si ponechává svůj význam (platnost) v programu či bloku, v němž je deklarován či definován. Jeho jméno je v tomto rozsahu viditelné, není-li překryto:

- Na úrovni souboru začíná platnost deklarace místem deklarace a končí na konci souboru.
- Deklarace na úrovni argumentu funkce má rozsah od místa deklarace argumentu v rámci definice funkce až do ukončení vnějšího bloku definice funkce. Pokud se nejedná o definici funkce, končí rozsah deklarace argumentu s deklarací funkce.
- V rámci bloku je deklarace platná do konce bloku.

Jméno **makra** je platné od jeho **definice** (direktivou define) až do místa, kdy je **definice odstraněna** (direktivou undef), pokud vůbec odstraněna je. Jméno makra nemůže být maskováno. To vyplývá z toho, že makra zpracovává **preprocesor**.

5.5 Příkaz If - else

Časová náročnost: 15 minut

Příkaz **if-else** je jedním z příkazů jazyka C pro **větvení programu**. Často je také nazýván **podmíněný příkaz**. Jeho činnost je určena výsledkem testu podmínky, která je vyhodnocena jako pravdivá či nepravdivá.

Syntaxe příkazu if-else je jednoduchá, je však nutné zvýraznit, že **výraz** (podmínka splnění) musí být uzavřen **v kulatých závorkách**.

```
if (výraz) pøíkaz1 else pøíkaz2
```

Je-li podmínka splněna, tedy **výraz** dává **nenulovou hodnotu**, vykoná se **příkaz1**, pokud podmínka splněna není, tedy výsledkem **výrazu** je **nula**, vykoná se **příkaz2**. Po vykonání jednoho z příkazů pokračuje chod programu za podmíněným příkazem.

Pro názorné pochopení příkazu if-else si ukažme jeho činnost na programu. Naším úkolem bude přečíst znak a pokud to bude velké písmeno vypsat jeho ordinální hodnotu, v opačném případě vypsat hlášení o tom, že nejde o velké písmeno.

Příklad 5.1:

```
#include <stdio.h>

int main(void)
{
 int c;

 printf("Zadej znak: ");
 c = getchar();          /* nacteni znaku */
 if (c >='A' && c<='Z')
  printf("%d\n",c);      /* vytisteni ordinalni hodnoty */
 else
  printf("Male pismeno nebo jiny znak\n");
 return 0;
}
```

Další příklad požádá o zadání dvou čísel a vytiskne jejich podíl. V případě, že se budeme snažit dělit nulou vytiskne chybové hlášení.

Příklad 5.2:

```
#include <stdio.h>

void main(void)
{
  int a, b;

  printf("Zadejte dve cisla: ");
  scanf("%d%d", &a, &b);

  if(b)
    printf("%d\n", a/b);
  else
    printf("Nelze delit nulou!\n");
}
```

Další ilustrační jednoduché příkazy použití If-else:

Příklad:

```
if (i > 3)
 j = 5;
```

Příklad:

```
if (i > 3)
 j = 5;
else
 j = 1;
```

Složený příkaz je nutné uzavřít do bloku za pomoci závorek **{** a **}**. Tento blok však **nesmí** být ukončen středníkem ;

Příklad:

```
if (i > 3)
{
 j = 5;
 i = 7;
}  /* zde nesmí být žádný støedník */
else if (i < 1)
 j = 1;
else
 j = 0;
```

Samozřejmě že není nutné používat pouze konstrukce If-else. Můžeme použít jak zjednodušenou variantu se **syntaxí**:

```
if (výraz) pøíkaz1
```

případně variantu více rozvitou, se **syntaxí**:

```
if (výraz1) pøíkaz1;
 else if (výraz2) pøíkaz2;
 else if (výraz3) pøíkaz3;
 ...
 else if (výrazN) pøíkazN;
 else pøíkazN+1;
```

V programech napsaných v jazyce C se často setkáváme že místo příkazu:

```
if (vyraz != 0) se píše if (vyraz)
```

a místo příkazu:

```
if (vyraz == 0) se píše if (!vyraz)
```

5.6 Přepínač - switch

Časová náročnost: 20 minut

Ikdyž je příkaz **if** dobrý pro výběr ze dvou možností, stává se těžkopádným, když je potřeba pracovat s několika možnostmi. Jazyk C řeší tento problém příkazem **switch**. Příkaz **switch** je příkaz pro **vícenásobný výběr**. Používá se pro volbu **jedné z několika** variant a pracuje následovně. Hodnota je postupně testována podle seznamu celočíselných nebo znakových konstant. Když je nalezena shoda, provede se posloupnost příkazů spojená s touto hodnotou.

Obecný tvar příkazu **switch** je následující:

```
switch (celoèíselný výraz)
{
          case hodnota1: pøíkaz1;
          case hodnota2: pøíkaz2;
             .              .
             .              .
             .              .
          case hodnotaN: pøíkazN;
          default: pøíkazD;
}
```

Nyní si popíšeme jaké konstrukce nám **přepínač** umožňuje vytvořit. Po klíčovém slově **switch** následuje v závorkách uzavřený **celočíselný výraz**. Za ním obecně následuje příkaz. Příkaz přepínače je tvořen **blokem**, nebo• jinak bychom jen těžko mohli hovořit o přepínači mezi více variantami. V bloku jsou jednotlivé příkazy, **příkaz1, příkaz2, ..., příkazN** a **příkazD**. V tomtéž bloku mohou být po některé příkazy označeny speciálně umístěnou celočíselnou konstantou - mezi klíčovým slovem **case** a dvojtečkou.

Posloupnost příkazů pro **default** se provádí, když není nalezena žádná shoda. Část **default** je **nepovinná**. Pokud všechna porovnání selžou a **default** chybí, neprovede se žádná činnost.

A nyní si pokusíme vysvětlit význam **přepínače switch**. Během chodu programu je vyhodnocen **celočíselný výraz**. Podle jeho konkrétní hodnoty, je v bloku přepínače nalezena odpovídající hodnota, **hodnota1, hodnota2, ... hodnotaN** za kterou se přenese další chod programu. Provede se první **příkaz** za označeným místem, pak druhý, třetí, ..., dokud se nenarazí na konec bloku přepínače, nebo na zarážku **break**. Tím je ukončeno vykonávání přepínače a pokračuje se příkazem, který je v programu zapsán za příkazem přepínač.

Nyní, po trošce teorie, si ukážeme použití switch v praxi.

Naším úkolem je napsat program, ve kterém rozpoznáváme stisknuté klávesy 1 - 3 a vytiskneme název zadané klávesy. To znamená že po stisku **2** vypíše program **dvě**. Pokud zadáme jiné klávesy, než ty které zachytáváme vypíše program hlášení: **Jiná klávesa**.

Příklad 5.3:

```
#include <stdio.h>

int main(void)
{
 int i;

 printf("Stiskni klavesu: ");
 scanf("%d",&i);
 switch (i)
 {
 case 1:
          printf ("Stiskl jsi klavesu 1\n");
          break;
 case 2:
          printf ("Stiskl jsi klavesu 2\n");
          break;
 case 3:
          printf ("Stiskl jsi klavesu 3\n");
          break;
 default:
          printf ("Jina klavesa\n");
          break;
 }

 return 0;
}
```

Zamysleme se chvíli na tím, co by se stalo, kdyby v části kde kontrolujeme zda byla stisknuta klávesa 1 nebyl příkaz **break**. V tom případě by zápis vypadal následovně:

Příklad 5.4:

```
#include <stdio.h>

int main(void)
{
 int i;

 printf("Stiskni klavesu: ");
 scanf("%d",&i);
 switch (i)
 {
 case 1:
          printf ("Stiskl jsi klavesu 1\n");
 case 2:
          printf ("Stiskl jsi klavesu 2\n");
          break;
 case 3:
          printf ("Stiskl jsi klavesu 3\n");
          break;
 default:
          printf ("Jina klavesa\n");
          break;
 }

 return 0;
}
```

V tomto případě by při stisku klávesy 1 bylo vytisknuto:

```
Stiskl jsi klávesu 1
Stiskl jsi klávesu 2
```

Jak je vidět klíčové slovo **break** je velice důležité.

Nyní se podívejme na možné problémy přepínače **switch**.

- Program vyhodnotí výraz za **switch** a jeho hodnotu porovnává s každým z **case** návěští přepínače. Návěští **case** může být obsaženo uvnitř jiných příkazů (v rámci přepínače), kromě případného vnořeného přepínače.
- V jednom přepínači se nesmí používat dvě návěští se stejnou hodnotou.
- Příkaz **break** může být v rámci přepínače umístěn v libovolných příkazech, kromě případných vnořených **do**, **for**, **switch** nebo **while** příkazech. Tyto příkazy mohou **break** obsahovat. Jeho význam pak ovšem bude spojen s nimi, nikoliv s vnějším příkazem **case**.
- Častou chybou je opomenutí **case**. Samotná konstanta s dvojtečkou je pak chápána jako návěští skoku. Syntakticky o chybu nejde, ale význam je zcela jiný.

A ještě se podívejme na rozdíly v konstrukcích mezi přepínačem **switch** a **If-else**.

- Rozhodovací výraz **if** může testovat hodnoty jakéhokoliv typu (záleží jen na výsledném "nula - nenula"), zatímco rozhodovací výraz příkazu **switch** musí být výhradně celočíselným výrazem a jeho hodnoty se rozlišují podrobněji.
- V konstrukci **if-else-if** se provede nejvýše (či podle použití právě) jeden z příkazů. U přepínače se nemusí provést žádný z příkazů v bloku přepínače. Konstantní hodnota po slově **case** určuje pozici prvního příkazu v bloku přepínače. Pokud chceme, oddělíme následující příkazy příkazem **break**. Pokud **break** neuvedeme, provedou se všechny příkazy až do konce bloku přepínače.
- V přepínači se návěští **default** může vyskytovat kdekoliv. Odpovídající "default" varianta v konstrukci **if-else-if** může být umístěna pouze na konci.

5.7 Cykly

Časová náročnost: 2 minuty

Cyklus je část programu, která je v závislosti na podmínce prováděna opakovaně. U cyklu obvykle rozlišujeme **řídicí podmínku cyklu** Řídicí podmínka cyklu určuje, bude-li provedeno tělo cyklu, či bude-li řízení předáno za příkaz cyklu. **Tělo cyklu** je příkaz, zpravidla zapsaný v podobě bloku.

Cykly můžeme rozdělit podle toho, provede-li se tělo **alespoň jedenkrát**, a cykly, kde tělo **nemusí** být provedeno **vůbec**. Výběr typu cyklu ponechejme na vhodnosti použití v dané situaci i na zvyklostech programátora.

5.8 Cyklus while

Časová náročnost: 15 minut

Podmínka cyklu **while** se testuje **před** průchodem cyklu, což znamená, že cyklus tedy **vůbec nemusí proběhnout**. Programový kód pro **výraz** musí být uzavřen v kulatých závorkách.

Syntaxe příkazu while je následující:

```
while (výraz) pøíkaz
```

Začátečníci se často mylně domnívají, že když se při vykonávání těla cyklu změní hodnoty, na jejichž základě se vyhodnocuje řídcí podmínka cyklu, tedy **výraz**, projeví se to okamžitě změnou řízení chodu programu. Skutečnost je taková, že se **dokončí** vykonávání těla cyklu teprve **poté je** znovu **vyhodnocena řídicí podmínka** cyklu.

Do těla cyklu můžeme opět uvádět nepovinné příkazy **break** a **continue**. Jejich význam je následující:

- Příkaz **break** uvedený v těle cyklu ukončí provádění příkazů těla cyklu a přenese chod programu za probíhající cyklus.
- Příkaz **continue** rovněž ukončí provádění příkazů těla cyklu. Řízení ovšem předá na řídicí podmínku cyklu. U cyklu typu **while** jde o první příkaz za **while**. Podmínka je vyhodnocena a podle výsledku bude opět vykonáno tělo cyklu, nebo je řízení předáno za cyklus.

Správné pochopení cyklu while si nyní ukážeme na příkladu. V prvním příkladu budeme číst znaky z klávesnice do té doby dokud nestiskneme klávesu 'z' a všechny tisknutelné znaky budeme opět vypisovat na obrazovku.

Příklad 5.5:

```
#include <stdio.h>

int main()
{
 int c;

 printf("Zadavejte znaky: ");
 while ((c=getchar()) != 'z')
 {
  if (c >= ' ') /* tiskneme pouze viditelne znaky */
        putchar(c);
 }
 printf("Nacitani znaku bylo ukonceno\n");
 return 0;
}
```

Nyní si ukážeme tentýž příklad pouze s použitím **nekonečného cyklu while** a s použitím příkazů **break** a **continue**.

Příklad 5.6:

```
#include <stdio.h>

int main()
{
 int c;

 printf("Zadavejte znaky: ");
 while (1)               /* nekonecna smycka */
 {
  if ((c=getchar()) <= ' ')
        continue;       /* zahazujeme neviditelne znaky */
  if (c == 'z')
```

```
        break;          /* zastaveni po nacteni znaku z */
 putchar(c);            /* tisk znaku */
}
printf("Nacitani znaku bylo ukonceno\n");
return 0;
}
```

Pozn.: Tělo cyklu while může být také prázdné, například pokud chceme přeskočit všechny oddělovače na vstupu:

5.9 Cyklus for

Časová náročnost: 20 minut

Cyklus **for** se používá pro **zadaný počet opakování** příkazu nebo bloku příkazu. Má následující syntaxi:

```
for(inicializace; test-podmínky; inkrementace) pøíkaz;
```

Část **inicializace** se používá pro zadání počáteční hodnoty proměnné, která řídí průběh cyklu. Tato hodnota se obvykle označuje jako **řídící proměnná cyklu**. Část inicializace se provádí pouze jednou před začátkem cyklu. Část **test-podmínky** testuje řídící proměnnou cyklu na koncovou hodnotu. Je-li test podmínky vyhodnocen jako pravdivý, cyklus se opakuje. Je-li nepravdivý, cyklus se ukončí a zpracování programu pokračuje příkazem následujícím za cyklem. Test podmínky se provádí na začátku cyklu neboli před každým opakováním těla cyklu. Část **inkrementace** cyklu **for** se provádí na konci cyklu. To znamená, že část inkrementace se provádí poté, co se provede příkaz nebo blok, který tvoří tělo cyklu. Účelem inkrementace je zvyšovat (nebo snižovat) řídící proměnnou cyklu o určitou hodnotu.

Část **inkrementace** je nepovinná. Typicky se jedná o přípravu další iterace cyklu. Pokud neuvedeme testovací výraz **test-podmínky**, použije překladač hodnotu jedna, a tedy bude provádět nekonečný cyklus. Naštěstí můžeme použít příkazy **break** a **continue** se stejným významem, jaký jsme popsali u cyklu **while**.

Jako první jednoduchý příkaz si ukážeme program který vypíše čísla od 1 do 10.

Příklad 5.7:

```
#include <stdio.h>

int main(void)
{
 int i, cislo;

 for(i=1; i<=10; i++)
    printf("%d ",i);

 return 0;
}
```

Program pracuje následovně:

Nejprve se inicializuje řídící proměnná cyklu **i** na hodnotu 1. Pak se vyhodnotí příkaz **i<=10** a jelikož je pravdivý začne běžet cyklus for. Po vytisknutí čísla se **i** zvětší o 1 a znovu se vyhodnotí test podmínky. Tento proces pokračuje do té doby, dokud není i rovno 11. Když k tomu dojde, cyklus for se zastaví a ukončí.

Mějme na paměti, že se podmínka cyklu vyhodnocuje **na začátku** každého opakování. To znamená, že je-li test na začátku cyklu nepravdivý, neprovede se tělo cyklu ani jednou.

Zkusme si další příklad, který vypočítá součet a součin čísel od 1 do 5.

Příklad 5.8:

```
#include <stdio.h>

int main(void)
{
 int i, soucet, soucin;
 soucet=0;
 soucin=1;

 for(i=1; i<6;i++)
 {
  soucet = soucet + i;
  soucin = soucin * i;
 }
 printf("Soucet cisel 1 az 5 je: %d",soucet);
 printf("Soucin cisel 1 az 5 je: %d",soucin);

 return 0;
}
```

Nyní si ukážeme některé další možné použití cyklu for. Navážeme na náš předchozí

příklad a opět se budeme snažit tisknout čísla od 1 do 10

Příklad:

1. Klasické a doporučované použití **for**

```
int i = 0;
for (i=0; i<11; i++)
        printf("%d",i);
```

2. Využití inicializace i při definici - ne příliš vhodné řešení, protože není vše pohromadě

```
int i = 0;
for ( ; i<11; i++)
        printf("%d",i);
```

3. Řídící proměnná je měněna v těle cyklu - opět nevhodné řešení

```
int i = 0;
for ( ; i<11; )
        printf("%d",i++);
```

4. Využití operátoru čárka - časté, ale opět ne zcela vhodné řešení

```
int i = 0;
for ( ; i<11; printf("%d",i), i++)
```

5. Využití operátoru čárka při inicializaci (i = 1, sum = 0) je vhodné, ovšem při výpočtu (sum += i, i++) je nevhodné

```
int i, sum;
for ( i = 1, sum = 0; i<=11; sum +=i, i++)
        ;
```

6. Cyklus for může měnit řídící proměnnou libovolným způsobem

```
int i, soucin;
for ( i = 3, soucin = 1; i<=10; i+=2)
        soucin *=i;
```

Často se používá i **nekonečný cyklus**, který má tvar:

```
for ( ; ; ;)
```

5.10 Cyklus do

Časová náročnost: 7 minut

Příkaz **do** je jediným z cyklů, který zajiš•uje alespoň **jedno** provedení těla cyklu. Jinak řečeno, jeho testovací příkaz **příkaz** je testován až po průchodu tělem cyklu. Pokud je test, představovaný hodnotou získanou z **výrazu** splněn, provádí se tělo cyklu, tedy **příkaz**, který je typicky blokem.

Syntaxe:

```
do
{
 pøíkazy;
}while(výraz);
```

Opět provedeme naše již známé načítání znaků z klávesnice, ovšem tentokrát vytiskneme i koncový znak z.

Příklad 5.9:

```
#include <stdio.h>

int main(void)
{
 int c;

 do
 {
  if ((c=getchar()) >= ' ')   /* nacitani znaku */
        putchar(c);   /* tisk znaku */
 } while (c != 'z');

 return 0;
}
```

Mohlo by se zdát, že cyklus typu **do-while** je zbytečný. Totéž bychom ovšem mohli tvrdit i o cyklu **for**. Programátorská teorie totiž ukazuje, že pomocí cyklu typu **while** jsme schopni naprogramovat jakýkoliv cyklus. Programátorská realita nám ukazuje, že existence tří možných typů cyklů má své opodstatnění - to, co si programátor

oblíbí, to rád a správně používá.

5.11 Příkaz goto

Časová náročnost: 3 minuty

Jazyk C podporu příkaz nepodmíněného skoku nazvaný **goto**. Většina programátorů však příkaz **goto** nepoužívá, protože narušuje strukturu programu a pokud je používán často stává se napsaný kód nepřehledným. Později se může stát, že bude velice těžké pochopit správnou funkci takto napsaného programu.

Příkaz **goto** může provádět skok v rámci funkce. Nemůže však přeskakovat mezi dvěma funkcemi. Používá se spolu s návěstím.

Syntaxe:

```
návìští: pøíkaz;

goto návìští;
```

5.12 Opakování

Časová náročnost: 1 minuta

Uvědomte si, že:

- Pro ukončení smyčky se používá **break**
- Za každou větví příkazu **switch** by měl být **break** - pokud ovšem není mezi větvemi souvislost
- Vyhýbejte se komplikovaným zápisům a použití příkazu **goto**

Cvièení

1) Je tento úsek programu správný?

```
if(pocet < 100)
   printf("Cislo je mensi nez 100.\n");
   printf("Jeho druha mocnina je: %d", pocet * pocet);
}
```

Řešení

2) Napište program, který uživatele požádá o zadání celého čísla a pak vypíše zda je číslo liché nebo sudé. (Rada: použijte operátor modulo %). Řešení

3) Napište program, který si vyžádá dvě celá čísla a pak zobrazí podle volby uživatele buď součin nebo součet. Řešení

4) Co je špatně na této části programu?

```
float f;

scanf("%f",&f);

switch(f):
{
 case 10.05:
 .
 .
 .
}
```

Řešení

5) Napište program, který načte zadaný znak a rozhodne zda je souhláska či samohláska. Řešení

6) Napište program, který vytiskne čísla mezi 17 a 100, která budou beze zbytku dělitelná 17. Řešení

7) Napište program, který převádí galony na litry. Použijte cyklus do, aby mohl uživatel převody opakovat. (Jeden galon je asi 3,7854 litrů). Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

4. Operátory

Časová náročnost kapitoly: 1 hodina 10 minut

V této kapitole se seznámíme s operátory a jejich rozdělením. Zaměříme se zejména na operátory aritmetické. Ukážeme si jejich použití s celočíselnými i racionálními operandy a nezapomeneme ani na výrazy smíšené.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 a 3 - tedy znát strukturu programu, orientovat se v identifikátorech, klíčových slovech, komentářích, odsazovačích, číslech, řetězcích a konstantách (celočíselných, racionálních a znakových). A chápat také základy problematiky ukazatelů.

4.1 Výrazy

Časová náročnost: 2 minuty

S pomocí **operátorů** vytváříme **výrazy**. Zapíšeme-li například v matematice **a + b** hovoříme výrazu. Ten má dva **operandy a** a **b** a jeden **operátor +**.

S operandy v jazyce C to není jednoduché. Mohou být tvořeny **konstantami**, **identifikátory objektů**, **řetězci** a **voláním funkcí**. Operandy však můžeme vytvářet i pomocí **kombinací** zmíněných možností a dalších operátorů i závorek.

4.2 Přiřazení

Časová náročnost: 7 minut

Přiřazovací příkaz je nejčastěji používaným příkazem ve většině programovacích jazyků. **Operátorem přiřazení** v jazyce C je symbol **=**. Obecně musíme říci, že **vlevo** od = musí být výraz, odkazující se do paměti. Možná to zní nejasně, ale je jisté, že hodnotu zprava musíme někam zapsat. A kam jinam ji může program zapsat, než do paměti na adresu, kterou určí právě z levé strany přiřazovacího výrazu.

Adresu proměnné určí překladač jazyka C velmi snadno. Typ výrazu je určen typem operandů. Jsou-li operandy stejného typu, je výsledek téhož typu. Jinak je typu **největšího z uvedených operandů**.

Příklad:

```
j = 5;
d = 'z';
f = f + 3.14 *i;
```

Protože přiřazení je výraz je možné **několikanásobné přiřazení**.

Příklad:

```
int i,j,k;
i = j = k = 2;
```

Teorie zavedla pro pravou stranu přiřazení pojmenování **hodnotový výraz**, zatímco stranu levou pojmenovává **adresový výraz**. V anglické terminologii i v některých českých textech se používá zkrácené označení **rvalue** - r-hodnota, respektive **lvalue** - l-hodnota.

Platí tedy:

- výraz má vždy hodnotu (číslo)
- přiřazení je výraz a jeho hodnotou je hodnota na pravé straně
- přiřazení se stává příkazem je-li ukončeno středníkem

Časté chyby:

- C má přiřazení pouze pomocí = a ne pomocí := či ==
- Porovnání je v C == ne pouze =

ISO norma jazyka C dále **důrazně nedoporučuje** používat výrazy, kde se vlevo i vpravo od operátoru přiřazení mění stejná hodnota.

Příklad:

```
i = i++ *2;    /* takto NE */
```

4.3 Aritmetické výrazy

Časová náročnost: 7 minut

Aritmetické výrazy konstruujeme z operandů a aritmetických operátorů.

Operátor	Činnost
+	sčítání
-	odčítání
*	násobení
/	dělení
%	zbytek po celočíselném dělení

Příklad 4.1:

```
#include <stdio.h>

int main(void)
{
 int a = 7, b = 2;

 printf("cislo a = %d, cislo b = %d\n", a, b);
 printf("soucet %d + %d = %d\n", a, b, a+b);
 printf("rozdil %d - %d = %d\n", a, b, a-b);
 printf("soucin %d * %d = %d\n", a, b, a*b);
 printf("podil %d / %d = %d\n", a, b, a/b);
 printf("zbytek po celociselnem deleni %d %% %d = ", a, b, a%b);

 return 0;
}
```

Zkusme si ještě jeden příklad, kde argumenty aritmetických operací jsou celočíselné a levá strana je racionální.

Příklad 4.2:

```
#include <stdio.h>

int main(void)
{
 int i, j;
 float r, x;

 i = j = 5;

 j *= i;
 r = j / 3;
 x = j * 3;

 printf("i=%d\tj=%d\tr=%f\tx=%f\n", i, j, r, x);

 return 0;
}
```

V tomto příkladu jasně vidíme, že výpočet na pravé straně probíhá podle typů operandů pravé strany. Teprve je-li to potřeba je výsledek převeden na typ odpovídajíc straně levé.

Proto je výsledek tohoto příkladu: i=5 j=25 r=8.000000 x=75.000000

Pro určení pořadí vyhodnocování jednotlivých částí výrazů používáme **kulaté závorky** ().

4.4 Logické operátory

Časová náročnost: 5 minut

Logické hodnoty jsou dva, **pravda** a **nepravda**. Norma ISO C říká, že hodnota **nepravda** je představována **nulou**, zatímco **pravda jedničkou**. V případě hodnoty pravda se ovšem jedná pouze o doporučení. Nebo• užívaným anachronismem je považovat jakoukoliv **nenulovou hodnotu za pravdu**.

Operátor	Činnost
&	and (logický součet)
||	or (logický součin)
!	not (negace)

Pravidla pro určení výsledku známe z **Booleovy algebry**.

Pro demonstrování funkcí AND, OR a NOT si napíšeme jednoduchý příklad.

Příklad 4.3:

```
#include <stdio.h>

int main(void)
{
 int p,q;

 printf("Zadejte P (0 nebo 1): ");
 scanf("%d",&p);
 printf("Zadejte Q (0 nebo 1): ");
 scanf("%d",&q);
 printf("P AND Q: %d\n", p && q);
 printf("P OR Q: %d\n", p || q);
 printf("NOT Q: %d\n", !q);

return 0;
}
```

4.5 Relační operátory

Časová náročnost: 3 minuty

Máme následující relační operátory:

Operátor	Činnost
<	menší
>	větší
<=	menší nebo rovno
>=	větší nebo rovno
==	rovno
!=	nerovno

Tyto operátory jsou definovány pro operandy všech základních datových typů. Ovšem například pro ukazatele mají smysl pouze operátory rovnosti a nerovnosti. **Výsledkem** relačních operací jsou logické hodnoty **pravda** a **nepravda**.

4.6 Bitové operátory

Časová náročnost: 15 minut

Již podle názvu můžeme usuzovat, že nám bitové operátory umožňují provádět operace nad jednotlivými bity. Bitové operace je však možné provádět pouze s celočíselnými hodnotami.

Máme následující typy bitových operátorů:

Operátor	Činnost
<<	bitový posun vlevo
>>	bitový posun vpravo
&	bitový and (logický součin, konjunkce)
|	bitový or (logický součet, disjunkce)
~	bitový not (negace, inverze)
^	bitový xor (nonekvivalence)

Při **bitovém posunu vlevo (vpravo)** << (>>), se jednotlivé bity posouvají vlevo (vpravo), tedy do pozice s (binárně) vyšším (nižším) řádem. Na nejpravější (nejlevější) posunem vytvořenou pozici je umístěna nula. Posuny ovšem probíhají aritmeticky. To znamená, že uvedené pravidlo neplatí pro posun vpravo hodnoty celočíselného typu se znaménkem. V takovém případě se nejvyšší bit (znaménkový), zachovává. Takto se při posunu doplňuje do bitového řetězce nový bit. Naopak před posunem nejlevější (nejpravější) bit je odeslán do říše zapomnění.

Bitový posun o jeden (binární) řád vpravo, respektive vlevo, má stejný význam, jako celočíselné dělení, respektive násobení, dvěma. Je-li bitový posun o více než jeden řád, jedná se o násobení (dělení) příslušnou mocninou dvou. Přirozeně to platí pouze pro bitový posun celočíselných typů unsigned.

Obecný formát pro operátory bitového posunu vlevo (vpravo):

```
hodnota << poèet-bitù
hodnota >> poèet-bitù
```

Bitové and &, or |, a xor ^ provádí příslušnou binární operaci s každým párem odpovídajících si bitů. Výsledek je umístěn do pozice stejného binárního řádu výsledku. Výsledky operací nad jednotlivými bity jsou stejné, jako v Booleově algebře. **Bitové not ~** je operátorem unárním, provádí negaci každého bitu v bitovém řetězci jediného operandu. Tomuto operátoru se často říká bitový doplněk.

Pozn: XOR dává výsledek jedna jen v případě různosti hodnot operandů.

Nyní si ukážeme jednoduchý příklad použití bitových posunů.

Příklad 4.4:

```
#include <stdio.h>

int main()
{
 printf("1 << 1 = \t%d\t%#x\n", 1 << 1, 1 << 1);
 printf("1 << 7 = \t%d\t%#x\n", 1 << 7, 1 << 7);

 printf("-1 >> 1 = \t%d\t%#x\n", -1 >> 1, -1 >> 1);
 printf("512 >> 8 = \t%d\t%#x\n", 512 >> 8, 512 >> 8);

 printf("2 & 1 = \t%d\t%#x\n", 2 & 1, 2 & 1);
 printf("2 | 1 = \t%d\t%#x\n", 2 | 1, 2 | 1);
 printf("2 ^ 1 = \t%d\t%#x\n", 2 ^ 1, 2 ^ 1);

 return 0;
}
```

Výsledkem tohoto programu je následující výpis:

```
1 << 1 =        2       0x2
1 << 7 =        128     0x80
-1 >> 1 =       -1      0xffffffff
512 >> 8 =      2       0x2
2 & 1 =         0       0
2 | 1 =         3       0x3
```

```
2 ^ 1 =          3        0x3
```

Operace XOR má jednu zajímavou vlastnost. Máme-li dvě hodnoty A a B, pak je-li na výsledek operace A XOR B uplatněna znovu operace XOR s operandem B dostáváme operand A. Tuto vlastnost si demonstrujeme na příkladu.

Příklad 4.5:

```
#include <stdio.h>

int main (void)
{
 int a = 10, b = 20;

 printf("Pocatecni hodnota a: %d\n",a);

 a = a ^ b;
 printf("Po prvnim XOR: %d\n",a);

 a = a ^ b;
 printf("Po druhem XOR: %d\n",a);

 return 0;
}
```

4.7 Adresový operátor

Časová náročnost: 2 minuty

Adresový operátor & je unární. Jak již název adresový operátor napovídá, umožňuje získat adresu objektu, na nějž je aplikován. Adresu objektu můžeme použít v nejrůznějších situacích, obvykle je to v souvislosti s předáváním výsledků funkcí. Takto například můžeme přečíst hodnoty dvou proměnných jedinou funkcí pro formátovaný vstup.

Příklad:

```
int i;
float f;
scanf("%d %f",&i, &f);
```

4.8 Podmíněný operátor

Časová náročnost: 8 minut

Jazyk C obsahuje ternární (trojitý) operátor **?**. Ternární operátor potřebuje tři operandy. Operátor **?** se používá pro náhradu příkazu typu:

```
if(podmínka) pøíkaz1;
else pøíkaz2;
```

Obecný formát operátoru **?** je:

```
promìnná = podmínka ? výraz1 : výraz2
```

Podmínka je výraz, který je vyhodnocen jako pravdivý nebo nepravdivý. Je-li pravdivý dosadí se do **proměnné** hodnota **výrazu1**. Je-li **nepravdivý**, dosadí se do proměnné hodnota **výrazu2**. Důvodem použití operátoru **?** je to, že překladač jazyka C může v tomto případě vytvářet mnohem efektivnější kód než při použití příkazu **if-else**.

Ukažme si použití podmíněného operátoru na příkladu. Mějme program, který načte číslo a v případě že je větší nebo rovno nule jej převede na jedničku a v případě, že zadané číslo bylo záporné, na mínus jedničku.

Příklad 4.6:

```
#include <stdio.h>

int main(void)
{
 int i;

 printf("Zadejte cislo: ");
 scanf("%d",&i);
 i = i>=0 ? 1: -1;
 printf("Vysledek: %d",i);

 return 0;
}
```

4.9 Operátor čárka

Časová náročnost: 5 minut

Operátor čárka má zcela jedinečnou funkci. Říká překladači **udělej tohle a tohle a tohle**. To znamená, že se čárka používá pro **zřetězení několika operací**.

V následujícím cyklu se čárka používá v inicializační části pro inicializaci dvou řídících proměnných cyklu a v části pro inkrementaci **i** a **j**.

Pozn.: O cyklu for si více řekneme v následující kapitole.

Příklad:

```
for(i=0, j=0; i+j<pocet; i++, j++) .......
```

Hodnotu seznamu výrazů oddělených čárkami určuje výraz, který stojí **nejvíce vpravo**. V následujícím příkladu se do proměnné hodnota přiřadí číslo 100.

Příklad:

```
hodnota = (pocet, 90, 50, 100);
```

Závorky jsou zde potřebné, nebo• operátor čárka má nižší prioritu než přiřazovací operátor.

4.10 Přetypování výrazu

Časová náročnost: 5 minut

Někdy potřebujeme dočasně změnit typ proměnné. Můžeme například chtít používat pro nějaký výpočet hodnotu v pohyblivé řádové čárce, ale někde pro ni chceme zase použít operátor celočíselného dělení. Jelikož však lze tento operátor použít pouze na celá čísla máme problém. Jedním z možných řešení je vytvořit celočíselnou proměnnou, která se bude používat pro celočíselné dělení a ve vhodné chvíli ji přiřadit hodnotu proměnné s pohyblivou řádovou čárkou. To je však poněkud neelegantní řešení. Jiným řešení je použít **přetypování**, které způsobí dočasnou změnu typu.

Přetypování má obecný tvar:

```
(typ) hodnota
```

kde **typ** je jméno platného typu jazyka C.

Příklad:

```
float f;
f = 100.2;
printf("%d", (int) f);   /* pøetypování na celé èíslo */
```

4.11 Priorita operátorů

Časová náročnost: 5 minut

Následující přehled všech operátorů ukazuje jejich prioritu od **nejvyšší** po **nejnižší**.

Operátory	**Asociativita**
( ) [ ] ->.	zleva doprava
! ~ + - ++ -- (přetypování) *(*pointer*) &(*adresní operátor*) sizeof	zprava doleva
* / %	zleva doprava
+ -	zleva doprava
<< >>	zleva doprava
< <= > >=	zleva doprava
== !=	zleva doprava
&	zleva doprava
^	zleva doprava
|	zleva doprava
&&	zleva doprava
||	zleva doprava
?:	zprava doleva
= += -= *= /= %= >>= <<= &= |= ^=	zprava doleva
,	zleva doprava

Cvièení

1) Napište program, který načte dvě desetinná čísla (použijte typ **float**) a pak vytiskněte jejich součet. Řešení

2) Napište program, který vypočte objem kvádru. Program se bude ptát uživatele na jednotlivé rozměry a pak vypíše objem. Řešení

3) Napište program, který vypočítá počet sekund v nepřestupném roce. Řešení

4) Je tento výraz pravdivý?

!(10==9)

Řešení

5) Dávají tyto dva výrazy stejný výsledek?

a. 0&&1||1
b. 0&&(1||1)

Řešení

6) Jelikož hodnota v pohyblivé řádové čárce nemůže být použita spolu s operátorem %, jak můžete upravit tento příkaz?

```
x = 123.123 % 3 ;
```

Řešení

7) Jedním zvláště dobrým příkladem použití operátoru **?** je ochrana před dělením nulou. Napište program, který přečte dvě celá čísla zadaná uživatelem a vypíše výsledek dělení prvního čísla druhým. Použijte **?** abyste zabránili dělením nulou. Řešení

8) Převeďte následující příkaz na jeho ekvivalentní příkaz pomocí operátoru **?**.

```
if(a>b) pocet = 5;
else pocet = 10;
```

Řešení

9) Jaká je hodnota **i** po provedení následujícího kroku?

```
i = (1, 2, 3);
```

Řešení

10) Jaký je výsledek těchto operací?

a. 10100011 & 01011101
b. 01011101 | 11111011
c. 01010110 ^ 10101011

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

 Obsah  

3. Základní typy, konstanty a proměnné

Časová náročnost kapitoly: 1 hodina 15 minut

V této části se seznámíme s identifikátory, klíčovými slovy, komentáři. Řekneme si něco o číslech, konstantách - a to celočíselných, racionálních a znakových a nezapomeneme ani na řetězce, proměnné a ukazatele.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 - tedy znát základní strukturu programu v jazyce C a alespoň intuitivně chápat vstup a výstup programů v jazyce C.

3.1 Identifikátory

Časová náročnost: 2 minuty

Identifikátory jsou jména, která dáváme například proměnným, funkcím a typům. Délka identifikátoru je v normě ISO C omezená 31 znaky.

Pro vytváření identifikátorů platí:

- Prvním symbolem smí být pouze **písmeno** nebo **podtržítko**
- Poté následuje libovolná kombinace písmen, číslic a podtržítek
- **POZOR!** Rozlišují se malá a velká písmena!

Příklad:

```
identifikátor a Identifikátor
```

jsou **dva** různé identifikátory

3.2 Klíčová slova

Časová náročnost: 5 minut

Klíčová slova jsou zvláštní identifikátory, které mají speciální význam pro jazyk C. Proto je **nesmíme** používat v jiném významu, než jak určuje norma ISO C, což znamená, že nesmí být použita jako jména proměnných nebo funkcí.

Použití malých písmen v klíčových slovech je také důležité. Například **RETURN** nebude považováno za klíčové slovo **return**.

V jazyce C jsou definovány následující **klíčová slova**:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

3.3 Komentáře

Časová náročnost: 5 minut

Komentář je poznámka, kterou přidáváme do zdrojového kódu programu. Všechny komentáře jsou překladačem ignorovány. Komentáře se používají hlavně pro popis významu a účelu zdrojového kódu. Je to část programu umístěná mezi dvojici párových symbolů **/* */**. V tomto případě jde o komentář, který může zasahovat i přes více řádků.

Příklad:

```
printf("Ahoj");  /* toto je jednoøádkový komentáø */

/*
Toto je
viceradkovy
komentar
*/
```

V jazyku C může být **komentář** umístěn kdekoliv, jen ne uprostřed klíčového slova, jména funkce nebo jména proměnné.

Komentář lze také použít pro dočasné vynechání řádků zdrojového textu. Stačí uzavřít vynechávanou část symboly pro začátek a konec komentáře.

Pozn: Jazyk Java zavádí dokumentační komentář, který je podkladem pro generovaní dokumentace.

Pozn: Jazyk C++ zavádí jednořádkový komentář uvozený dvojicí lomítek \\ - více viz Komentáře v jazyce C++.

3.4 Odsazovače

Časová náročnost: 3 minuty

Odsazovače jinak také **bílé znaky**, jsou následující symboly:

mezera	tabulátor
nový řádek	posun řádku
návrat vozíku	vertikální tabulátor
nová stránka	

V obvyklém zdrojovém textu se nejčastěji můžeme setkat s prvními třemi představiteli odsazovačů. Odsazovače spolu s operátory a oddělovači stojí mezi identifikátory, klíčovými slovy, řetězci a konstantami použitými ve zdrojovém textu. Všechny uvedené pojmy se společně označují jako **lexikální symboly**. Pro překladač představují dále nedělitelné celky. Překladač považuje rovněž komentář za odsazovač.

3.5 Čísla

Časová náročnost: 8 minut

Čísla jsou v počítači uloženy ve formě nul a jedniček - tedy základní soustavou není soustava desítková, ale **dvojková**. Čísla jsou proto v paměti počítače uložena takovým způsobem, že jen celá čísla jsou uložena přesně - ovšem jen ve stanoveném rozsahu hodnot - zatímco u čísel reálných je k dispozici nejvýše jistý počet platných číslic plus exponent. Z výše uvedených skutečností musí vycházet každý programovací jazyk. V jazyce C dělíme základní typy dat na:

- celočíselné
- racionální
- znaky
- ukazatele

Úvodní dva typy společně nazýváme **aritmetické datové typy**.

Pozn: Racionálním typům se často říká dle jejich základního představitele float, správně se však jmenují čísla v pohyblivé řádové čárce.

Celočíselné datové typy mohou být takzvaně **bezznaménkové**, což znamená, že použijeme modifikátor **unsigned**. Tatáž vlastnost u znaků, například **unsigned char**, může být zajímavá kvůli kódování neanglických abeced v různých operačních systémech.

Základní typy dat:

Datový typ	**Počet bitů**	**Význam**
char	8	znak
short	16	krátké celé číslo
int	32	celé číslo
long	32	dlouhé celé číslo
enum	32	výčtový typ
float	32	racionální číslo
double	64	racionální číslo s dvojitou přesností
long double	80	ještě delší racionální číslo
pointer	32	ukazatel

Dále platí:

short <= int <= long

float <= double <= long double

char vyžaduje 8 bitů

Nebudete-li si jisti rozsahem hodnot jednotlivých aritmetických typů, podívejte se ve vašem systému do souboru **limits.h** - pro celočíselné typy, respektive do souboru **float.h** - pro typy racionální. V nich najdete nejmenší případně i největší možné hodnoty, které příslušný překladač připouští.

3.6 Pojmenovávání objektů

Časová náročnost: 2 minuty

S konstantami a proměnnými jsou úzce spjaty dva pojmy - **deklarace** a **definice**.

Deklarací určujeme typ objektu. Informace o typu je překladačem používána při typové kontrole, typových konverzích apod.

V místě **definice** definujeme hodnotu proměnné či posloupnost příkazů funkce.

3.7 Konstanty

Časová náročnost: 5 minut

Konstanty jsou pevně dané hodnoty, které nesmí být programem změněny.

Konstanty definujeme po klíčovém slově **const** následovaném typem konstanty, jejím identifikátorem a po rovnítku její hodnotou ukončenou středníkem. Jedná-li se o vektor (jednorozměrné pole), následuje za identifikátorem dvojice hranatých závorek, zpravidla obsahující jeho dimenzi. První prvek pole má vždy **index 0**. Nejlépe si možné definice konstant vysvětlíme na příkladech.

Příklad:

```
const int konstanta = 15;
const konstanta = 21; /* neuvedeme-li typ je implicitnì pøiøazen typ int */
const char pismeno = 'r';
const char *retezec = "Jazyk C";
const float pole[3] = {-1, 0, 1};

/*
pokud uvedeme všechny hodnoty pole na jeho pravé stranì nemusíme uvádìt dimenzi
*/
const char pole[] = {'a', 'b'}
```

Překladač jazyka C přidělí konstantám typ, který odpovídá přiřazované hodnotě. Z konstant odpovídajících si typů můžeme vytvářet **konstantní výrazy**. Tyto výrazy musí být vyhodnotitelné během překladu. Nesmí obsahovat žádný z následujících operátorů (nejsou-li použity mezi operandy operátoru **sizeof**):

- přiřazení
- inkrementace a dekrementace
- funkční volání
- čárka

3.8 Celočíselné konstanty

Časová náročnost: 3 minuty

Celočíselné konstanty jsou specifikovány jako čísla bez desetinné části. Jazyk C umožňuje použít tři typy celočíselných konstant:

- **desítkové** - posloupnost číslic z nichž první nesmí být nula
- **osmičkové** (oktalové) - číslice nula následována posloupností osmičkových číslic 0-7
- **šestnáctkové** (hexadecimální) - číslice nula následována znakem x (nebo X) a posloupností hexadecimálních číslic 0-9, a-f, A-F

Příklad:

```
desítkové - 15, 7, -25
oktalové - 065, 01, 036
hexadecimální - 0x12, 0x3A, 0XCD
```

Typ konstanty je určen implicitně, její velikostí, nebo explicitně používáním přípony L (nebo l - pozor snadno zaměnitelné s číslicí jedna) jako **long**, například 1234L. Druhá přípona, která explicitně určuje typ celočíselné konstanty je U (nebo u), jako **unsigned**, například 129U. Samozřejmě, že můžeme naspat i 21UL. Pak jde o **unsigned long** číslo.

3.9 Racionální konstanty

Časová náročnost: 6 minut

Racionální datový typ ukládá čísla do paměti ve tvaru **mantisa** a **exponent**, obojí s případným znaménkem. Není-li uvedeno jinak, je typ racionální konstanty **double**. U konstant s **pohyblivou řádovou čárkou** je třeba použít desetinnou tečku následovanou desetinnou částí čísla. Jazyk C umožňuje zapisovat čísla s pohyblivou řádovou čárkou v semilogaritmickém tvaru.

Je však nutné dodržet následující formát:

```
mantisa E znaménko exponent
```

Příklad:

```
123.45E1
```

Na výše uvedeném příkladu jsme pomocí semilogaritmické notace zapsali číslo 1234,5. **Znaménko** je nepovinné.

Norma ISO C říká, že rozsah exponentů pro všechny racionální datové typy je 10^{-38} až 10^{+38} a že přesnost typu float je nejméně šest platných číslic, přesnost typů **double** a **long double** je pak nejméně deset platných číslic.

Přesněji viz tabulka:

Typ	Počet bitů	Mantisa	Exponent	Rozsah absolutních hodnot
float	32	24	8	$3.4*10^{-38}$ - $3.4*10^{+38}$
double	64	53	11	$1.7*10^{-308}$ - $1.7*10^{+308}$
long double	80	64	15	$3.4*10^{-4932}$ - $3.4*10^{+4932}$

Má-li naše číslo větší absolutní hodnotu, než je hodnota pro typ tohoto čísla uvedená v tabulce dochází k **přetečení** a nastává chyba. V opačném případě, tedy když je absolutní hodnota čísla pro daný typ menší než je hodnota uváděná v tabulce, dochází k **podtečení** a výsledkem výrazu je nula.

3.10 Znakové konstanty

Časová náročnost: 4 minuty

Znakové konstanty jsou tvořeny požadovaným znakem uzavřeným mezi **apostrofy**.

Příklad:

```
'a'  'A'  'Š'  'ò'  '}'  '"'
```

Někdy však musíme jako znakovou konstantu použít i speciální znaky (řídící symboly, znaky nenacházející se na klávesnici). Zde si pomáháme symbolem zpětného lomítka a nejméně jedním dalším znakem. Těmto posloupnostem říkáme **escape sekvence**. Escape sekvence mohou být jednoduché - kdy zpětné lomítko následuje jediný znak. Nebo následuje osmičkový či šestnáctkový kód znaku. Tak jsme schopni zadat i znak, který se na klávesnici nenachází, ale jehož číselný kód je nám znám.

Příklad:

Posloupnost	Jméno	Význam
\a	Alert	pípnutí
\b	Backspace	návrat o jeden znak
\f	Formfeed	odstránkování

\n	New line	na začátek nového řádku
\r	Carriage return	na začátek aktuálního řádku
\t	Horizontal tab	na další tabelační pozici
\\	Backslash	zpětné lomítko
\000		znak zadaný osmičkově
\xHH		znak zadaný šestnáctkově

3.11 Řetězce

Časová náročnost: 2 minuty

Řetězec je posloupnost (pole) jednoho či více znaků. Začátek a konec řetězce jsou vymezeny **uvozovkami**.

Příklad:

```
"Jazyk C"
"nìkolik znakù v øetìzci"
"r" /* øetìzcová konstanta tvoøená øetìzcem délky jeden znak = r */
"Jazyk C\n" /* speciální symboly zapisujeme opìt pomocí escape sekvencí */
```

3.12 Proměnné

Časová náročnost: 5 minut

Proměnné jsou pamě•ová místa přístupná prostřednictvím **identifikátoru**. Hodnotu proměnných můžeme během výpočtu měnit. Tím se proměnné zásadně odlišují od **konstant**, které mají po celou dobu chodu programu **hodnotu neměnnou** - konstantní.

Proměnné mohou obsahovat písmena, číslice 0-9 a podtržítko. Jména však **nesmí** začínat číslicí.

Během provádění programu se veškeré informace nacházejí v paměti. Tamtéž jsou umístěny pomocné hodnoty, mezivýsledky i výsledky. Rozhodně by nebylo vůbec příjemné k takovým hodnotám přistupovat prostřednictvím jejich adresy. Proto je vhodné si odpovídající pamě•ová místa pojmenovat a pak se na ně odkazovat jménem. Nejprve tedy deklarujeme typy a identifikátory proměnných, čímž pro ně vyhrazujeme v paměti počítače místo a současně si překladač spojuje s identifikátorem proměnné informaci o jejím umístění v paměti (adrese).

Proměnné **deklarujeme** uvedením **datového typu**, za kterým následuje **identifikátor**, nebo **seznam identifikátorů**, navzájem oddělených **čárkami**. Deklarace **končí středníkem**. Současně s deklarací proměnné můžeme, ale nemusíme, definovat i její počáteční hodnotu:

Příklad:

```
int a, b, c;
float pi=3.14;
```

3.13 Ukazatele

Časová náročnost: 20 minut

Ukazatel je proměnná, která obsahuje pamě•ovou adresu jiného objektu. Například obsahuje-li proměnná nazvaná **p** adresu jiné proměnné nazvané **q**, pak se **p** nazývá **ukazatel** na **q**. Je-li proměnná **q** uložena v paměti na adrese 100, pak by měla proměnná **p** hodnotu 100.

Pro deklaraci proměnné typu ukazatel použijeme tento obecný formát:

```
typ *jméno-promìnné;
```

Typ zde znamená **základní typ** ukazatele. Základní typ určuje typ objektu, na který může ukazatel ukazovat. Všimneme si hvězdičky před jménem proměnné. Ta říká překladači, že se vytváří ukazatelová proměnná.

Uveďme si jednoduchý příklad ukazatele na celé číslo:

Příklad:

```
int *p;
```

Jazyk C má dva speciální ukazatelové operátory: ***** a **&**. **Operátor &** vrací adresu proměnné, prřd kterou stojí. **Operátor *** vrací hodnotu uloženou na adrese, před kterou stojí.

Pozn.: Ukazatelový operátor * nemá nic společného s operátorem násobení, který používá stejný symbol.

Příklad 3.1:

```
#include <stdio.h>

int main (void)
{
 int *p, q;
 q = 199;   /* priradi q hodnotu 199 */
 p = &q;   /* priradi p adresu q */
 printf("%d", *p);              /* zobrazi hodnotu q pomoci ukazatele */
 return 0;
}
```

Tento program vypíše na obrazovku hodnotu **199**.

Projděme si však jednotlivé části programu.

První řádek

```
int *p, q;
```

nadefinuje dvě proměnné. **p** - ukazatel na celé číslo a **q** - což je celé číslo.

Poté je **q** přiřazena hodnota 199

```
q = 199;
```

Následně přiřadíme **p** adresu **q**. Tento řádek můžeme přečíst jako: Přiřaď p adresu q.

```
p = &q;
```

Nakonec vytiskneme hodnotu ukazatele **q**. Operátor * můžeme vyjádřit jako: na adrese a celý řádek pak přečíst jako: vypiš hodnotu na adrese **q**.

```
printf("%d", *p);
```

Nyní přepišme tento program tak, že hodnotu do **q** přiřadíme nepřímo přes ukazatel **p**.

Příklad 3.2:

```
#include <stdio.h>

int main (void)
{
 int *p, q;
 p = &q;   /* ziskani adresy q */
 *p = 199;         /* priradi q hodnotu pomoci ukazatele */
 printf("hodnota q je %d", q);
 return 0;
}
```

Nyní si ukážeme jak jednoduše lze udělat chybu.

Příklad:

```
int q;
double *p;

p = &q;
p = 100.2;
```

Ikdyž je tento úsek syntakticky v pořádku přesto není správný. Ukazateli **p** je přiřazena adresa čísla **int**. Tato adresa je pak použita na levé straně přiřazovacího příkazu pro přiřazení hodnoty s pohyblivou řádovou čárkou. Jelikož je však číslo typu **int** obvykle kratší než **double**, způsobí tento přiřazovací příkaz přepsání paměti sousedící s **q**. Proto jsou důležité základní **typy** ukazatelů.

Další možný chybný zápis vzniká tehdy, pokud se snažíme použít ukazatel dříve než je do něj přiřazena adresa proměnné. V takovém případě program pravděpodobně zhavaruje. Je důležité si uvědomit, že **deklarace** ukazatelové proměnné pouze vytvoří proměnnou schopnou pojmout adresu paměti. **Nedá** ji však žádnou smysluplnou hodnotu.

Příklad:

```
int *p;

*p = 10;  /* chyba - ukazatel p na nic neukazuje! */
```

Místa, na která se ukazatel odkazuje (jejichž adresu obsahuje) se mohou lišit. Takže můžeme například psát cykly, v nichž pomocí jediného ukazatele postupně přistupujeme ke všem prvkům pole. Je nemyslitelné, psát takový kód "natvrdo". Další možnosti použití ukazatelů se otevírají při dynamickém přidělování paměti.

Je zde i jedno nebezpečí. Spočívá vtom, že **nelze** v programu zadávat absolutní adresu a očekávat, že bude použitelná jako pamě•ové místo.

Příklad:

```
int *p;
p = 1024;
*p = 102;  /* takže takto rozhodnì NE!! */
```

Na závěr poměrně rozsáhlé části o ukazatelích se pokusíme znázornit problematiku ukazatelů graficky.

Předpokládejme že máme tuto část kódu:

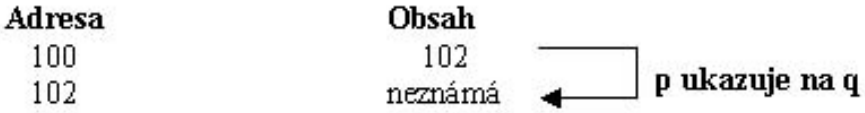
```
int *p, q;
```

Dále mějme zadáno že **q** je umístěno v paměti na adrese **102** a **p** na adrese **100**.

Poté následující příkaz

```
p = &q;
```

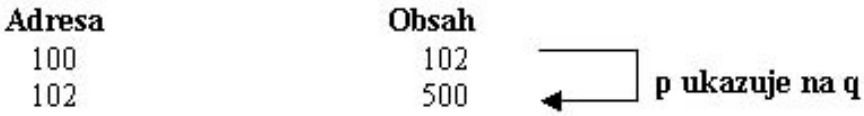
způsobí, že ukazatel p bude obsahovat hodnotu 102.



a po provedení příkazu

```
*p = 500;
```

bude obsah paměti vypadat takto



Cvièení

1) Co je chybné v těchto jménech proměnných?

a. pocet-hodin
b. $suma
c. black+white
d. 9krat

Řešení

2) Která z následujících slov nejsou klíčová?

a. auto
b. else
c. goto
d. void

Řešení

3) Co je chybně na této části programu?

```
/* timto se nacte cislo
scanf ("%d", &num);
```

Řešení

4) Datový typ char vyžaduje:

a. 8 bitů
b. 18 bitů
c. 6 bitů
d. 16 bitů

Řešení

5) Podle rozsahu hodnot je:

a. int = long
b. int = float
c. float > double
d. float <= double

Řešení

6) Napište program, který vytiskne **Mám rád jazyk C** - pomocí tří konstantních řetězců. Řešení

7) Mějme celočíselnou hodnotu **130L**. Jde o:

a. desítkovou hodnotu se znaménkem
b. dvojkovou hodnotu se znaménkem
c. desítkovou hodnotu typu long
d. desítkovou hodnotu typu long bez znaménka

Řešení

8) Napište program, který přečte a vytiskne hodnotu typu **long int**. Řešení

9) Co je to ukazatel? Řešení

10) Jaké jsou ukazatelové operátory a jaká je jejich funkce? Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

Obsah

2. Základy jazyka C

Časová náročnost kapitoly: 30 minut

V této části si řekneme něco o základní struktuře programů v jazyce C a napíšeme si první programy pro načítání a zobrazování čísel zadaných z klávesnice.

2.1 Prvky programu jazyka C

Časová náročnost: 7 minut

Všechny programy v jazyku C sdílejí určité základní součásti a vlastnosti. Všechny programy v jazyku C se skládají z jedné nebo více **funkcí**, z nichž každá obsahuje jeden nebo více **příkazů**. Funkce je v jazyku C pojmenovaný podprogram, který lze volat z jiných částí programu. Funkce jsou základními stavebními kameny jazyka C.

Všechny příkazy jazyka C končí **středníkem**. Jazyk C nepovažuje konec řádku za ukončení příkazu. Na jednom řádku tedy můžeme umístit i dva nebo více příkazů a jeden příkaz může pokračovat přes více řádků.

Obecný formát **funkce** v jazyku C je uveden zde:

```
návratový-typ jméno-funkce (seznam-argumentù)
{
 posloupnost pøíkazù
}
```

Položka **návratový-typ** určuje typ dat vrácených funkcí.

Položka **jméno-funkce** určuje jméno funkce.

Informace lze funkci předávat pomocí jejich argumentů, které jsou uvedeny v seznamu argumentů funkce **seznam-argumentů**.

Položka **posloupnost příkazů** může být jeden nebo více příkazů. Teoreticky by funkce nemusela obsahovat žádný příkaz, ale protože by v takovém případě neprováděla žádnou činnost, nemělo by to v praxi žádný význam.

2.2 První programy

Časová náročnost: 12 minut

Jelikož všechny programy v jazyku C sdílejí společné vlastnosti, pochopení jednoho programu nám pomůže pochopit mnoho dalších.

Příklad 2.1:

```
#include <stdio.h>

int main (void)
{
 printf("Jednoduchy program napsany v jazyce C");

 return 0;
}
```

Po přeložení a spuštění vypíše tento program na obrazovku počítače zprávu **Jednoduchy program napsany v jazyce C**.

Projděme si nyní tento program krok po kroku.

První řádek programu je:

```
#include <stdio.h>
```

To způsobí, že překladač jazyka C přečte soubor **stdio.h** a začlení jej do programu. Tento soubor obsahuje mimo jiné informace o funkci **printf()**.

Druhý řádek:

```
int main (void)
```

začíná funkci **main()**. Položka **int** určuje, že **main()** vrátí celočíselnou hodnotu. Položka **void** říká překladači, že **main()** nemá žádné argumenty.

Třetí řádek:

```
printf("Jednoduchy program napsany v jazyce C"};
```

Je příkaz jazyka C. Volá standardní knihovní funkci **printf()**, která zobrazí zadaný řetězec znaků.

Následující řádek:

```
return 0;
```

Podle konvencí indikuje nulová vrácená hodnota z funkce **main()** normální ukončení programu. Jakákoliv jiná hodnota představuje chybu. Operační systém může tuto hodnotu otestovat, aby zjistil, zda program proběhl úspěšně, nebo zda došlo k chybě. Položka **return** je jedním z klíčových slov jazyka C.

Nakonec je program formálně ukončen, když dojde na uzavírací závorku funkce **main()**.

Pro úplné pochopení základů jazyka C si uvedeme ještě jeden jednoduchý program:

Příklad 2.2:

```
#include <stdio.h>

int main (void)
{
 printf("Dalsi jednoduchy program ");
 printf("napsany v ");
 printf("jazyce C");

 return 0;
}
```

Tento program vypíše na obrazovku **Dalsi jednoduchy program napsany v jazyce C**.

2.3 Jednoduchý vstup a výstup

Časová náročnost: 8 minut

Přestože existuje několik způsobů jak číst hodnoty z klávesnice jedna z nejjednodušších variant je použít jednu ze standardních funkcí jazyka C nazvanou **scanf()**.

Pro jednoduchost si ukážeme jak přečíst celočíselnou hodnotu zadanou z klávesnice. Pro načtení této celočíselné hodnoty volejme funkci **scanf()** v obecném tvaru:

```
scanf("%d", &jmeno-celociselne-promenne);
```

kde **jmeno-celociselne-promenne** je jméno celočíselné proměnné do níž chceme uložit hodnotu. Prvním argumentem **scanf()** je řetězec, který určuje jak bude zpracován druhý argument. V našem případě **%d** určuje, že do druhého argumentu bude zadána celočíselná hodnota v desítkovém tvaru. Znak **&** je pro funkci **scanf()** velmi důležitý. Umožňuje totiž vložit hodnotu do jednoho z argumentů funkce.

Když zadáváme číslo z klávesnice, píšeme jednoduše řetězec číslic. Funkce **scanf()** čeká dokud nestiskneme Enter, pak převede řetězec do dvojkového formátu.

Vše si opět nejlépe demonstrujeme na příkladu:

Naším úkolem je načíst celé číslo. Po jeho zadání se na obrazovku vytiskne naše zadané číslo.

Příklad 2.3:

```
#include <stdio.h>

int main (void)
{
 int cislo;

 printf("Zadej cele cislo: ");
 scanf("%d",&cislo);

 printf("Zadal jsi: ");
 printf("%d", cislo);

 return 0;
}
```

Cvièení

1) Napište program, který načte dvě celá čísla a poté vytiskne jejich součet.

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

10. Soubory

Časová náročnost kapitoly: 1 hodina 20 minut

Soubor je posloupnost znaků (bajtů) ukončená nějakou speciální kombinací, která již k obsahu souboru nepatří - konec souboru, symbolicky **EOF**.

Textový soubor obsahuje **řádky textu**.

Binární soubor obsahuje hodnoty v **témže tvaru**, v jakém jsou uloženy v paměti počítače.

Soubor, s nímž můžeme v jazyce C pracovat, má své **jméno**. Tím rozumíme jméno na úrovni operačního systému. Někdy se v této souvislosti zavádí pojmy **vnější jméno souboru** a **vnitřní jméno souboru**. Tím prvním rozumíme jméno souboru na úrovni OS, druhý pojem chápeme jako jednoznačnou identifikaci souboru v rámci programu v jazyce C - nejčastěji jde o jméno proměnné, jejímž prostřednictvím se souborem pracujeme.

Operační systém neví nic o obsahu souboru, nezná jeho typ souboru. Přípony jmen souborů jsou většinou pouze doporučeními. **Binární soubor** obvykle nemá smysl vypisovat na terminál. **Textový soubor** můžeme snadno zobrazit a číst.

ISO norma jazyka C nám umožňuje pracovat se soubory technikou **datových proudů**. Pro manipulaci s datovým proudem je k dispozici řada funkcí, které poskytují vysoký komfort. Navíc, díky normám, můžeme v dobré víře očekávat plnou přenositelnost našich zdrojových textů na všechny platformy.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 9 - chápat a umět používat dříve probíranou teorii, rozumět problematice ukazatelů. Umět vytvářet pole (jedno i více rozměrná). Znát operace prováděné s řetězci a použití argumentů příkazového řádku.

10.1 Datové proudy

Časová náročnost: 6 minut

Než začneme probírat souborový I/O, musíme znát dva velice důležité pojmy: **datový proud** a **soubor**. O **souboru** jsme se již zmínili výše. Nyní se zaměříme na **datový proud**.

I/O systém jazyka C poskytuje programátorovi stále stejné rozhraní, bez ohledu na skutečně použité I/O zařízení. Aby se toho dalo dosáhnout, zavádí jazyk C mezi programátorem a zařízením (hardwarem) určitou úroveň abstrakce, která se nazývá **datový proud** (datový tok, stream). Skutečné zařízení provádějící I/O operace se nazývá soubor. Datový proud je tedy logickým rozhraním souboru. Za abstraktní soubor můžeme vzhledem k uvedenému považovat diskový soubor, obrazovku, klávesnici, port, tiskárnu a různé jiné. Nejčastějším typem je samozřejmě diskový soubor. Výhodou tohoto přístupu je, že pro programátora vypadá jedno hardwarové zařízení stejně jako druhé. Datový proud automaticky ošetřuje rozdíly.

Datový proud je připojen k souboru pomocí **operace otevření** (open) a odpojen od souboru pomocí **operace uzavření** (close).

ISO norma definuje dva režimy proudů - **textový** (rozlišuje řádky) a **binární**. Režim stanovíme při otevírání souboru.

Dalším důležitým pojmem je tzv. **aktuální pozice**. Je to místo v souboru, kde se bude provádět další operace se souborem. Například, je-li soubor dlouhý 100 byte a byla přečtena jeho polovina, pak bude další operace čtení probíhat od 50 bytu, což je aktuální pozice v souboru.

10.2 Základní datové proudy

Časová náročnost: 2 minuty

Základem pro přístup k proudu je datový typ **FILE**. Pro práci s datovými proudy musíme používat funkční prototypy umístěné v soubor **stdio.h**. Při každém spuštění programu máme otevřeny následující proudy:

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

10.3 Otevření a zavření proudu

Časová náročnost: 12 minut

Teprve po **otevření** můžeme s proudem pracovat. Při otevření proudu provádíme spojení mezi **vnitřním** a **vnějším** jménem souboru. Při otevření určujeme režim našeho přístupu k datům v proudu. **Uzavřením** proudu umožňujeme OS aktualizovat adresářové informace podle aktuálního stavu souboru, který byl s proudem spojen.

- `FILE *fopen(const char *filename, const char *mode);`

  Je funkce, vracející ukazatel na strukturu **FILE** v případě úspěšného otevření proudu. Při neúspěchu vrací hodnotu **NULL**. Konstantní řetězec **filename**, označuje jméno souboru podle konvencí příslušného operačního systému. Řetězec **mode** určuje režim práce se souborem i jeho **typ**.

- `int fclose(FILE *stream);`

  Je funkce uzavírající určený proud. V případě úspěchu vrátí hodnotu nula, jinak **EOF**. Uvolní pamě• vyhrazenou pro strukturu **FILE *** a vyprázdní případnou vyrovnávací pamě•.

- `ferror()`

  Informuje o chybách při práci s proudem.

- `perror()`

  Pošle řetězec chybového hlášení do standardního chybového proudu, tj. do **stderr**.

Pozn: Počet souborů, které můžeme z programu současně otevřít, je omezen operačním systémem (nejčastěji jeho konfigurací). Pro zjištění, jaký limit máme k dispozici, slouží makro **FOPEN_MAX**. Operační systém rovněž omezuje délku jména souboru. Rovněž tuto hodnotu můžeme zjistit pomocí makra, tentokráte však **FILENAME_MAX**. Konec souboru představuje makro **EOF**.

Režimy práce s datovým proudem:

Řetězec	**Význam - otevření pro**
r	čtení
w	zápis
a	připojení
r+	aktualizace - update - jako rw
w+	jako výše uvedené r+, ale existující proud ořízne na nulovou délku, jinak vytvoří nový
a+	aktualizace, pokud neexistuje tak jej vytvoří
t	textový režim
b	binární režim
rb	čtení v binárním režimu
wb	zápis v binárním režimu
ab	připojení v binárním režimu

Ikdyž většina souborových režimů nevyžaduje bližší vysvětlení, je vhodné uvést několik poznámek. Pokud soubor otevíráme jen pro čtení a on neexistuje, funkce **fopen()** selže. Když se otevírá neexistující soubor s režimem přidávání, pak bude vytvořen. Otevírá-li se již existující soubor v režimu pro přidávání, budou nová data zapisovány automaticky na konec souboru. Ke změně existujících dat nedojde. Když se otevírá neexistující soubor pro zápis, pak se vytvoří. Pokud soubor existuje, bude jeho původní obsah přepsán novým obsahem.

Jednoduchý příklad otevření souboru.

Příklad:

```
FILE *soubor;

if(soubor = fopen("mujsoubor", "r")) == NULL)
{
 printf("Chyba pri otevirani souboru");
 exit(1);
}
```

Výše popsaný, v tuto chvíli otevřený soubor, zavřeme jednoduše takto:

Příklad:

```
fclose(soubor);
```

10.4 Proud a vstup a výstup znaků

Časová náročnost: 10 minut

- `int fgetc(FILE *stream);`

  V případě úspěšného načtení znaku je jeho kód hodnotou návratovou. V případě chyby nebo dosažení konce proudu vrací hodnotu **EOF**.

- `int ungetc(int c, FILE *stream);`

  Je-li **c** různé od **EOF**, uloží jej do datového proudu **stream** a případně zruší příznak konce souboru. Následným čtením z tohoto proudu získáme námi zapsanou hodnotu. Je-li **c** rovno **EOF**, nebo nemůže-li zápis proběhnout, vrací funkce **EOF**. Jinak vrací kód vráceného znaku.

- `int fputc(int c, FILE *stream);`

  Zapíše znak **c** do proudu **stream**. Vrátí stejnou hodnotu, jako zapsala. V případě chyby, nebo dosažení konce proudu, vrací hodnotu **EOF**.

Pozn: Znak je po načtení z proudu konvertován bez znaménka na typ **int**. Obdobně je při zápisu do proudu konvertován opačným postupem. Tak máme ponechánu možnost rozlišit konec souboru od dalšího načteného znaku.

Na následujícím příkladu si vyzkoušíme základní funkce souborového systému. Nejprve otevřeme soubor MUJ pro zápis. Poté do něj zapíšeme: "Toto je cvicny soubor" a soubor zavřeme. Následně soubor otevřeme v režimu pouze pro čtení a vypíšeme jeho obsah na obrazovku a soubor opět zavřeme.

Příklad 10.1:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 char str[80] = "Toto je cvicny soubor";
 FILE *soubor;
 char ch, *p;

 /* otevreni souboru pro zapis */
 if((soubor = fopen("muj", "w")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 /* zapis do souboru a jeho ulozeni */
 p = str;
 while(*p)
  if(fputc(*p++, soubor) == EOF)
  {
        printf("Pri zapisu do souboru doslo k chybe\n");
        exit(1);
  }

 fclose(soubor);

 /* otevreni souboru pro cteni */
 if((soubor = fopen("muj", "r")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 /* cteni souboru */
 while((ch = fgetc(soubor)) != EOF)
  putchar(ch);

 fclose(soubor);

 return 0;
}
```

10.5 Proud a vstup a výstup řetězců

Časová náročnost: 10 minut

Z proudu nemusíme číst pouze jednotlivé znaky. Můžeme načítat i celé řádky. Jednotlivé řádky jsou ukončeny přechodem na nový řádek. Pro čtení musíme mít k dispozici dostatečně velkou **vyrovnávací pamě•**. Nejčastěji ji získáme pomocí znakového pole. Pro vyšší bezpečnost musíme při použití funkce pro čtení uvést velikost této vyrovnávací paměti. Při zápisu to pochopitelně nutné není. Do proudu se zapíše celý řetězec až po koncovou zarážku, ovšem bez ní.

- `char *fgets(char *s, int n, FILE *stream);`

  Načte řetězec (řádek až po jeho konec včetně znaku konce řádku) z proudu **stream** do vyrovnávací paměti **s**, nejvýše dlouhý **n-1** znaků. Vrátí ukazatel na tento řetězec (vyrovnávací pamě•), nebo, při chybě, **NULL**.

- `int fputs(const char *s, FILE *stream);`

  Zapíše řetězec **s** do datového proudu **stream**. V případě úspěchu vrátí počet zapsaných znaků (délku řetězce), jinak **EOF**.

A opět příklad. Naším úkolem bude číst řádky zadávané uživatelem a ukládat je do souboru jehož jméno bylo zadáno jako argument příkazového řádku. Když uživatel zadá prázdný řádek je vstup uzavřen a soubor uložen. Poté je soubor znovu otevřen a zadané řádky ze souboru jsou přečteny a vypsány na obrazovku.

Příklad 10.2:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 char str[80];
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc != 2)
 {
  printf("Chyba je nutne zadat jmeno souboru");
  exit(1);
 }

 /* otevreni souboru pro zapis */
 if((soubor = fopen(argv[1], "w")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 printf("Pro ukonceni zadejte prazdny radek\n");

 do
 {
  printf(": ");
  gets(str);
  strcat(str, "\n"); /* pridani noveho radku */
```

```
 if(*str != '\n')
       fputs(str, soubor);
}
while(*str != '\n');

fclose(soubor);

/* otevreni souboru pro cteni */
if((soubor = fopen(argv[1], "r")) == NULL)
{
 printf("Soubor nelze otevrit\n");
 exit(1);
}

/* cteni souboru */
do
{
 fgets(str, 79, soubor);
 if(!feof(soubor))
       printf(str);
}
while(!feof(soubor));


fclose(soubor);

return 0;
}
```

10.6 Proud a formátovaný vstup a výstup

Časová náročnost: 8 minut

Nová funkce **fprintf**(), známá **printf**() a podobně nová funkce **fscanf**() a známá **scanf**(). Je vidět, že nové funkce přidávají k odpovídajícím známým protějškům jen úvodní argument **stream**.

Pro srovnání:

```
int fprintf(FILE *stream, const char *format [,argument,...]);
int printf (              const char *format [,argument,...]);
int fscanf (FILE *stream, const char *format [,address, ...]);
int scanf  (              const char *format [,address, ...]);
```

Pomocí řetězcového a formátovaného výstupu vytvořme několikařádkový textový soubor. Ukážeme si při tom, jak do datového proudu zapíšeme řetězce a naformátované číselné hodnoty.

Příklad 10.3:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 FILE *soubor;
 int i;
 char *s, *jmeno = "soubor1.txt";
 if ((soubor = fopen(jmeno, "wt")) == NULL)
   return 1;
 s = "Toto je textovy soubor vytvoreny v jazyku C.\n";
 fputs(s, soubor);
 for (i = 0; i < 10; i++)
   {
    fprintf(soubor, "%5d", i);
   }
 fputs("\n", soubor);
 s = "Jeste pridat posledni radek na konec.\n";
 fputs(s, soubor);
 if (fclose(soubor) == EOF)     /* uzavreni proudu */
   return 1;
 return 0;
}
```

Výstup:

```
Toto je textovy soubor vytvoreny v jazyce C.
    0    1    2    3    4    5    6    7    8    9
Jeste pridat posledni radek na konec.
```

10.7 Další užitečné funkce

Časová náročnost: 4 minuty

- `int feof(FILE *stream);`

  Funkce, umožňující zjistit dosažení konce proudu. Její návratová hodnota je - true (tj. jednička, nebo jen hodnota různá od nuly), nacházíme-li se na konci proudu, nebo - false (nula) - jinak.

- `int fseek(FILE *stream, long offset, int whence);`

  Přenese aktuální pozici v proudu **stream** na stanovené místo. To je určeno posunem **offset** vzhledem k počátku, aktuální pozici, nebo konci souboru. Vztažný bod určuje argument **whence**.

Identifikátor	Význam
SEEK_SET	posun vůči aktuální pozici

SEEK_CUR	posun vzhledem počátku
SEEK_END	posun vzhledem ke konci

10.8 Binární soubory

Časová náročnost: 12 minut

Při práci s **binárním proudem** je nezbytný **blokový přenos dat**. Do binárního proudu totiž zapisujeme hodnoty nikoliv pěkně naformátované v textové podobě, ale v binárním tvaru. Tedy, tak jak jsou uloženy v paměti počítače. Výhodou je, že dopředu víme, kolik bajtů je pro který datový typ potřeba.

Skutečnost, že položky binárního souboru jsou stejně veliké, nám umožňuje vypočíst jejich polohu a přečíst, nebo zapsat třeba jedinou hodnotu na určenou pozici. Tomu říkáme **náhodný přístup**.

Funkce pro práci s binárními soubory:

- `fwrite(const void *ptr, size_t size, size_t n, FILE*stream);`

  Funkce, která provádí zápis položek do proudu. Má argumenty obdobného významu, jako předchozí funkce pro práci s datovými proudy.Typ **size_t**, je zaveden pro určení velikosti pamě•ových objektů a případně počtu.

- `fread(void *ptr, size_t size, size_t n, FILE *stream);`

  Přečte z proudu **stream** položky o velikosti **size** v počtu **n** jednotek do pamě•ové oblasti určené ukazatelem (adresou) **ptr**. V případě úspěchu vrátí počet načtených položek. Jinak vrátí počet menší (pravděpodobně to bude nula). Menší návratovou hodnotu, než je zadaný počet položek, můžeme získat například při dosažení konce proudu před načtením požadovaného počtu položek. I pak jsou načtené položky platné.

Ukažme si nyní program, který naplní pole o deseti prvcích čísly s pohyblivou řádovou čárkou, zapíše je do souboru a znovu přečte. Program zapisuje každý prvek pole zvláš•. Jelikož se binární data zapisují ve svém interním formátu, musí být soubor otevřen pro binární I/O operace.

Příklad 10.4:

```
#include <stdio.h>
#include <stdlib.h>

double d[10] = {10.23, 19.87, 100.2, 0.258,
11.15, 95.23, 21.21, 458.03, 73.321, 3.14};

int main(void)
{
 int i;
 FILE *soubor;

 if((soubor = fopen("soubor2", "wb")) == NULL)
```

```
{
 printf("Soubor nelze otevrit.\n");
 exit(1);
}

for(i=0; i<10; i++)
   if(fwrite(&d[i], sizeof(double), 1, soubor) != 1)
       {
        printf("Chyba pri zapisu.\n");
        exit(1);
       }
fclose(soubor);

if((soubor = fopen("soubor2", "rb")) == NULL)
{
 printf("Soubor nelze otevrit.\n");
 exit(1);
}

/* vymazani pole */
for(i=0; i<10; i++) d[i] = 0.0;

for(i=0; i<10; i++)
   if(fread(&d[i], sizeof(double), 1, soubor) != 1)
       {
        printf("Chyba pri zapisu.\n");
        exit(1);
       }
fclose(soubor);

/* vypis pole */
for(i=0; i<10; i++)
   printf("%f ", d[i]);

 return 0;
}
```

Následující příklad používá **fseek**() pro výpis hodnoty libovolného byte v souboru zadaném na příkazovém řádku.

Příklad 10.5:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 long pozice;
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc != 2)
 {
  printf("Chyba je nutne zadat jmeno souboru");
  exit(1);
 }

 /* otevreni souboru */
 if((soubor = fopen(argv[1], "rb")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
```

```
 exit(1);
}

/* vyhledavani */
printf("Zadejte cislo vyhledavaneho byte: \n");
scanf("%ld",&pozice);
if(fseek(soubor, pozice, SEEK_SET))
{
 printf("Chyba vyhledavani\n");
 exit(1);
}

printf("Hodnota na pozici %ld je %d", pozice, getc(soubor));

fclose(soubor);

return 0;
}
```

Cvièení

1) Napište program, který vypíše obsah textového souboru, zadaného jako argument příkazové řádky, na obrazovku. Řešení

2) Napište program, který kopíruje textový soubor. Jméno zdrojového a cílového souboru se zadává na příkazovém řádku. Pro kopírování souborů použijte funkce **fgets()** a **fputs()**. Proveďte úplnou kontrolu chyb. Řešení

3) Napište program, který umožňuje uživateli zadat libovolný počet hodnot **double** (max 32 767) a zapsat je do souboru tak, jak jsou zadávány. Tento soubor pojmenujte VALUES. Počet zadaných hodnot zapište do souboru COUNT. Řešení

4) Napište program, který používá soubory ze zadání **3)**. Program nejprve přečte ze souboru COUNT počet prvků v souboru VALUES. Poté čte a vypisuje hodnoty ze souboru VALUES. Řešení

5) Napište program, který vyhledává v souboru zadaném jako argument příkazového řádku určitou celočíselnou hodnotu zadanou také jako argument příkazového řádku. Je-li hodnota nalezena, nechejte program vypsat její bytovou pozici od začátku souboru. Řešení

6) Napište program, který spočítá počet bytů v souboru (textovém nebo binárním) a zobrazí výsledek. Jméno souboru zadá uživatel jako argument příkazové řádky. Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

11. Struktury a uživatelské typy dat

Časová náročnost kapitoly: 1 hodina 20 minut

V této části si ukážeme tvorbu a použití takových strukturovaných datových typů, jaké nám přináší život. Také si ukážeme definici výčtových typů, která umožní hodnoty nejen pojmenovat, ale provádět při překladu i jejich typovou kontrolu.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 10 - umět psát i složitější programy, chápat problematiku souborů (textových i binárních), umět do nich zapisovat a číst z nich data.

11.1 Uživatelský datový typ

Časová náročnost: 12 minut

Vyšší programovací jazyk má k dispozici takové základní datové typy, které pokryjí většinu potřeb. S jejich pomocí můžeme vytvářet rozsáhlejší homogenní datové struktury, jako například pole, či heterogenní, jako strukturu či unii (**struct** a **union**). Je však nutné musí mít k dispozici mechanismus, kterým si vytvoří datový typ podle svých potřeb. Tento mechanismus se nazývá **typedef** a má následující syntaxi:

```
typedef definice typu identifikátor;
```

Po klíčovém slově **typedef** následuje definice typu **definice typu**. Poté je novému typu určen identifikátor **identifikátor**.

A hned si ukažme několik příkladů. Nejprve si vytvořme nové jméno **smallint**, což bude jméno pro **signed char** a bude použito pro deklaraci **i**.

Příklad 11.1:

```
#include <stdio.h>

typedef signed char smallint;

int main(void)
{
 smallint i;

 for(i=0; i<10; i++)
    printf("%d ", i);

 return 0;
}
```

Další příklad je založen na kopírování řetězců. Pomocí konstrukce **typedef** vytvoříme nový datový typ **string**. Tím si zajistíme, že hvězdičky při deklaraci argumentů zmizí.

Příklad 11.2:

```
#include <stdio.h>

#define SIZE 80

typedef char * string;

void strcpy1(string d, string s)
{
 while ((*d++ = *s++) != 0) ;
}

void strcpy2(string d, string s)
{
 int i = 0;
 while ((d[i] = s[i]) != 0)
   i++;
}

int main()
{
 string s1 = "prvni (1.) retezec",
        s2 = "druhy (2.) retezec";
 char   d1[SIZE],
        d2[SIZE];
 strcpy1(d1, s1);
 strcpy2(d2, s2);
 printf("d1[]:%s\n", d1);
 printf("d2[]:%s\n", d2);
 return 0;
}
```

Musíme si však uvědomit **dvě základní věci**:

1. Typedef **nezpůsobí** deaktivaci původního jména.
2. Můžeme použít **několik příkazů** typedef pro vytvoření několika nových jmen pro stejný typ.

Příklad:

```
typedef int vyska;
typedef vyska delka;
typedef delka hloubka;

hloubka d;
```

Proměnná **d** je stále typu **int**.

11.2 Složitější typové deklarace

Časová náročnost: 15 minut

Nejprve si shrňme definice, které bychom už měli zvládnout.

Deklarace	Význam
typ jméno;	**typ**
typ jméno[];	(otevřené) pole typu **typ**
typ jméno[3];	pole (pevné velikosti) tří položek typu **typ** (jméno[0], jméno[1], jméno[2])
typ *jméno;	ukazatel na **typ**
typ *jméno[];	(otevřené) pole ukazatelů na **typ**
typ *(jméno[]);	(otevřené) pole ukazatelů na **typ**
typ (*jméno)[];	ukazatel na (otevřené) pole typu **typ**
typ jméno();	funkce vracející hodnotu typu **typ**
typ *jméno();	funkce vracející ukazatel na hodnotu typu **typ**
typ *(jméno());	funkce vracející ukazatel na hodnotu typu **typ**
typ (*jméno)();	ukazatel na funkci, vracející **typ**

Přesto se může stát, že si u některých definic přestáváme být jisti. Při deklaracích **typ (*jméno)[];** a **typ (*jméno)();** jsou závorky nezbytné, ale v případě **typ *(jméno[]);** jsou závorky naopak nadbytečné - zvyšují pouze čitelnost.

Zásady pro správnou interpretaci definic:

1. začněme u identifikátoru a hledejme vpravo kulaté nebo hranaté závorky (jsou-li nějaké)
2. interpretujme tyto závorky a hledejme vlevo hvězdičku
3. pokud narazíme na pravou závorku (libovolného stupně vnoření), vra•me se a aplikujme pravidla jedna a dva pro vše mezi závorkami
4. aplikujme specifikaci typu

Tento postup si nejlépe vysvětlíme na příkladu.

Příklad:

```
char  *(  *(  *var)  ()) [10];
```

A podle popsaných zásad postupujeme následovně:

1. identifikátor **var** je deklarován jako
2. ukazatel na
3. funkci vracející
4. ukazatel na
5. pole deseti prvků, které jsou
6. ukazateli na
7. hodnoty typu **char**

Další příklad, zde již popíšeme pouze výsledek.

Příklad:

```
double  (  *var (double (*)[3])) [3]
```

Tato funkce vrací ukazatel na pole tří hodnot typu double. Její argument, stejně jako návratová hodnota, je ukazatel na pole tří prvků typu **double**.

Argument předchozí funkce je konstrukce, která se nazývá **abstraktní deklarace**. Obecně se jedná o deklaraci bez identifikátoru. Deklarace obsahuje jeden či více ukazatelů, polí nebo modifikací funkcí. Pro zjednodušení a zpřehlednění abstraktních deklarací se používá konstrukce **typedef**.

Na závěr této části si uveďme několik příkladů abstraktních definic.

Příklad:

Deklarace	**Význam**
int *	ukazatel na typ **int**
int *[3]	pole tří ukazatelů na **int**

int (*)[5]	ukazatel na pole pěti prvků typu **int**
int *()	funkce bez specifikace argumentů vracející ukazatel na **int**
int (*) (void)	ukazatel na funkci nemající argumenty vracející **int**
int (*const []) (unsigned int, ...)	ukazatel na nespecifikovaný počet konstantních ukazatelů na funkce, z nichž každá má první argument **unsigned int** a nespecifikovaný počet dalších argumentů

11.3 Výčtový typ

Časová náročnost: 12 minut

V jazyku C můžeme definovat seznam pojmenovaných celočíselných konstant, nazvaných **výčet** (enumerace). Tyto konstanty lze pak použít všude, kde lze zadat celé číslo. Pro definici výčtového typu použijeme tento obecný formát:

```
enum jméno-výètu {seznam-položek} seznam-promìnných
```

Kde:

- **enum** - je klíčové slovo, zahajující definici hodnot výčtového typu
- **jméno-výčtu** - je nepovinná, dnes již nepožívaná "visačka" ve stylu K&R
- **{seznam-položek}** - je seznam konstant výčtového typu s možnou explicitně přiřazenou hodnotou, jinak nabývá první konstanta výčtového typu hodnoty nula, druhá hodnoty jedna, ..., každý následník má hodnotu o jedničku vyšší, než jeho předchůdce
- **seznam-proměnných** - je nepovinný seznam proměnných daného typu **enum**

Příklad:

```
enum {cervena, zelena, zluta} barva;
```

Výčtový typ se zde skládá z konstant cervena, zelena a zluta. Byla vytvořena jedna proměnná se jménem barva. Protože překladač přiřazuje konstantám výčtu celočíselnou hodnotu počínaje od nuly je **cervena** rovna **0**, **zelena** rovna **1** a **zluta** rovna **2**. Toto standardní přiřazení hodnot překladačem však můžeme změnit zadáním explicitní hodnoty konstanty.

Příklad:

```
enum {cervena, zelena=9, zluta} barva;
```

Je **cervena** zase rovna **0**, **zelena** je rovna **9** a **zluta** je rovna **10**.

Výčet je v podstatě celočíselný typ a výčtová proměnná může obsahovat libovolnou celočíselnou hodnotu, nejen tu, která je definována výčtem. Kvůli zachování přehlednosti bychom však měli používat výčtové proměnné jen pro uchování hodnot definovaných jejich

výčtovým typem.

Hlavním důvodem použití výčtových typů je napomoci vytváření samo vysvětlujícího a přehlednějšího zápisu programu.

Následující krátký program vytváří výčtový typ sestávající se z některých názvů částí počítače. Přiřadí **comp** hodnotu **CPU** (což je vlastně 1) a zobrazí ji.

Příklad 11.3:

```
#include <stdio.h>

enum {klavesnice, CPU, monitor, tiskarna} comp;

int main(void)
{
 comp = CPU;
 printf("%d", comp);

 return 0;
}
```

Všimněte si, že hodnoty výčtových typů **nelze** posílat na výstup ve tvaru, v jakém jsme je definovali. Můžeme je zobrazit pouze jako odpovídající **celá čísla**. Obdobně je můžeme číst ze vstupu. Výčtové konstanty se tedy ve své textové podobě nacházejí pouze ve zdrojovém tvaru programu. Přeložený program pracuje již jen s číselnými hodnotami výčtových konstant.

11.4 Typ struktura

Časová náročnost: 20 minut

Struktura je **sdružený** datový typ, který je složen ze dvou, nebo více proměnných nazvaných **prvky struktury**. Na rozdíl od pole, ve kterém jsou všechny prvky stejného typu, může mít každý prvek struktury svůj vlastní typ, který se může lišit od ostatních typů.

Struktury se v jazyku C definují podle následujícího obecného formátu:

```
struct jméno-typu
{
 typ prvek1[, prvek1, ....];
 typ prvek2[, prvek2, ....];
 typ prvek3[, prvek3, ....];
  .
  .
  .
 typ prvekN[, prvekN, ....];
} seznam-promìnných;
```

Kde:

- **struct** - říká překladači že se jedná o strukturovaný datový typ.
- **jméno-typu** - nepovinné pojmenování typu, které jako v případě výčtového typu nepoužíváme a zůstalo zachováno kvůli starší K&R definici.
- **typ prvek1[, prvek1, ...];** - blok definic položek struktury. Položky jsou odděleny středníkem. Jsou popsány identifikátorem typu, následovaným jedním, nebo více identifikátory prvků struktury. Ty jsou navzájem odděleny čárkami. Nic nám nebrání v tom, uvést místo jednoduchého identifikátoru pole položek určeného typu. Toto pole je pak součástí struktury stejně, jako ostatní položky struktury.
- **seznam-proměnných;** - proměnné nově definovaného typu.

Pro přístup k prvkům struktury používáme selektor struktury (záznamu) . "tečka". Tu umístíme mezi identifikátory proměnné typu struktura a identifikátor položky, s níž chceme pracovat. V případě, kdy máme ukazatel na strukturu, použijeme místo hvězdičky a nezbytných závorek raději operátor **->**.

Informace ve struktuře mají většinou nějaký logický vztah. Strukturu bychom mohli například používat pro uchování adresy osoby.

Příklad:

```
struct
{
 char jmeno[20];
 char prijmeni[30];
 char ulice[30];
 int cislo_popisne;
 char mesto[30];
 unsigned char psc;
} adresa;
```

Přiřazení hodnot může vypadat následovně:

Příklad:

```
adresa.jmeno="Petr";
adresa.prijmeno="Novak";
```

```
adresa.cislo_popisne=158;
```

Následující příklad nám předvede některé ze způsobů práce s prvky struktury.

Příklad 11.4:

```
#include <stdio.h>

struct
{
 int i;
 double d;
 char str[80];
} s;

int main(void)
{
 printf("Zadejte cele cislo: ");
 scanf("%d", &s.i);
 printf("Zadejte desetinne cislo: ");
 scanf("%lf", &s.d);
 printf("Zadejte retezec: ");
 scanf("%s", &s.str);

 printf("Bylo zadano: \n");
 printf("%d %lf %s", s.i, s.d, s.str);

 return 0;
}
```

Další příklad nám ukáže, že jména prvků struktury nebudou v konfliktu s jinými proměnnými používající stejná jména. Proto se taky na obrazovku vypíše **10 100 101**. Proměnná **i** a strukturovaný prvek **i** nemají mezi sebou žádný vztah.

Příklad 11.5:

```
#include <stdio.h>

struct
{
 int i;
 int j;
} s;

int main(void)
{
 int i;

 i = 10;
 s.i = 100;
 s.j = 101;
 printf("%d %d %d", i, s.i, s.j);

 return 0;
}
```

Syntaxe typu **FILE**

```
typedef struct {
  short          level;
  unsigned       flags;
  char           fd;
  unsigned char  hold;
  short          bsize;
  unsigned char *buffer, *curp;
  unsigned       istemp;
  short          token;
} FILE;
```

Typ **FILE** je jeden z typů, které již používáme dlouho dobu. Jeho definice je v hlavičkovém souboru **stdio.h**

Problém nastane v okamžiku, kdy potřebujeme definovat dvě struktury, které spolu navzájem souvisí. Přesněji řečeno, jedna obsahuje prvek typu té druhé. A naopak. Pravdou sice je, že se nejedná o častou situaci, nicméně se můžeme podívat na použití **neúplné deklarace**.

Příklad:

```
struct A;                /* neuplna */
struct B {struct A *pa};
struct A {struct B *pb}; /* dokonceni */
```

Vidíme, že u **neúplné deklarace** určíme identifikátoru **A** třídu **struct**. V těle struktury **B** se ovšem může vyskytovat pouze ukazatel na takto neúplně deklarovanou strukturu **A**. Její velikost totiž ještě není známa. Velikost nutná pro uložení ukazatele ovšem známa je. Proto je taková konstrukce možná.

11.5 Typ unie

Časová náročnost: 5 minut

Syntaxe:

```
union [<union type name>] {
  <type> <variable names> ;
  ...
} [<union variables>] ;
```

Již na první pohled je **unie** velmi podobná strukturám. S jedním podstatným rozdílem, který není zřejmý ze syntaxe, ale je dán sémantikou. Z položek unie lze používat v jednom okamžiku pouze **jednu**. Ostatní mají **nedefinovanou hodnotu**. Každý z prvků unie totiž začíná na jejím začátku. Můžeme si představit, že pamě•ově delší prvky překrývají ty kratší. Této skutečnosti můžeme někdy využít. Nevíme-li, jakého typu bude návratový argument, definujeme unii, mající položky všech požadovaných typů. Dalším argumentem předáme informaci o skutečném typu hodnoty. Pak podle ní provedeme přístup k správnému členu unie. Méně čitelné řešení předá vždy argument nejdelšího typu a poté jej podle potřeby přetypuje.

11.6 Bitová pole

Časová náročnost: 10 minut

Jazyk C má zvláštní druh struktury nazvaný **bitové pole**. Bitové pole se skládá z jednoho nebo více bitů. Pomocí bitového pole lze přistupovat prostřednictvím jména k jednomu nebo více bitům v bytu nebo ve slově. Pro definici bitového pole použijeme tento obecný formát:

```
typ jméno: délka;
```

Položka **typ** je buď **int** nebo **unsigned**. Zadáte-li bitové pole se znaménkem, pak je **nejvyšší bit** považován za **znaménkový bit**. Počet bitů v poli udává položka **délka**. Všimněte si, že je jméno bitového pole odděleno od od délky pole v bitech **dvojtečkou**.

Bitová pole se hodí, když potřebujeme natěsnat informace do co nejmenšího místa. Zde je například struktura, která používá bitové pole pro uložení inventárních informací.

Příklad:

```
struct
{
 unsigned department: 3 /* max. 7 oddeleni */
 unsigned instock: 1 /* 1 je na sklade, 0 neni na sklade */
 unsigned ordered: 1 /* 1 je objednana, 0 neni objednana */
 unsigned time: 3 /* doba vedeni objednavky v mesicich */
} inv[MAX_ITEM];
```

V tomto případě lze použít jeden byte k uložení informací o inventární položce, které by bez bitových polí zabraly normálně 4 byty. S bitovým pole se pracuje stejně jako s jinými prvky struktury.

Například následující příkaz přiřadí do bitového pole **department** v 10. položce pole **inv** hodnotu 3.

Příklad:

```
inv[9].department = 3;
```

Následující příkaz zjiš•uje, zda je 5. položka na skladě:

Příklad:

```
if(!inv[4].instock)
   printf("Neni na sklade");
else
   printf("Je na sklade");
```

Cvièení

1) Ukažte jak udělat u **UL** nové jméno pro **unsigned long**. Napište krátký program, který deklaruje proměnnou pomocí **UL**, přiřadí jí hodnotu a vypíše hodnotu proměnné na obrazovku. Řešení

2) Co je špatně na tomto zápisu?

```
typedef balance float;
```

Řešení

3) Vytvořte výčtový typ obsahující dny v týdnu. Řešení

4) Je tento úsek programu správný? Pokud ne, proč?

```
enum auta {Ford, Chrysler, GM} vyrobce;

vyrobce = GM;
printf("Auto vyrobil: %s",vyrobce);
```

Řešení

5) Co je špatně na tomto kousku programu?

```
stuct
{
 int i;
 long l;
 char str[80];
} s;
```

```
.
.
.
i = 10;
.
```

Řešení

6) Co je struktura a co je unie? Řešení

7) Co je špatně na tomto úseku programu?

```
struct
{
 int a;
 char b;
 float c;
} myvar, *p;

p = &myvar;
p.a = 10;
```

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

12. Dynamické datové struktury

Časová náročnost kapitoly: 1 hodina 10 minut

V této části si shrneme přednosti použití dynamických datových struktur. Ukážeme si způsob správy, získávání a vracení dynamické paměti. A popíšeme si jednu ze základních dynamických datových struktur - **seznam**.

Dynamické datové struktury představují jakýsi protiklad ke **statickým** datovým strukturám.

Statická data mají svou velikost přesně a neměně určenou v okamžiku překladu zdrojového textu.

Dynamická data nemají velikost při překladu určenou. Při překladu jsou vytvořeny pouze proměnné vhodného typu **ukazatel na**, které nám za chodu slouží jako pevné body pro práci s dynamickými daty. O požadovaný pamě•ový prostor ovšem musíme požádat OS. Jakmile prostor dostaneme, pracujeme s ním prostřednictvím ukazatelů.

Dynamická data snižují pamě•ové nároky proto, že požadují právě tolik paměti, kolik je v daný okamžik třeba. Tím umožňují i vícenásobné používání přidělené dynamické paměti, pokud jsou pamě•ové nároky vhodně rozloženy během činnosti programu. Místo, odkud se dynamické pamě• "bere", tedy kam se umis•ují dynamická data, se nazývá hromada či **halda** (heap).

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z kapitoly 2 až 11 - tedy rozumět doposud probírané látce. Umět vytvářet struktury a uživatelské typy dat, rozumět složitějším typovým deklaracím. Rozumět pojmům jako struktura, unie, výčtový typ nebo bitové pole.

12.1 Dynamická alokace paměti

Časová náročnost: 25 minut

Programovou podporu pro přidělování paměti z **haldy** a její navracení zpět, definuje ISO norma jazyka C pomocí několika funkcí. Jejich deklarace jsou umístěny v **stdlib.h**.

- `void *malloc(size_t size);`

  Funkce **malloc()** vrací ukazatel na první byte v oblasti paměti o velikost **size**, která byla alokována z haldy. Pokud pro uspokojení požadavku není v haldě dostatek paměti, vrací **malloc()** nulový ukazatel. Před použití vráceného ukazatele je důležité prověřit, zda není nulový. Použití nulového ukazatele obvykle způsobí havárii systému.

- `void *calloc(size_t num, size_t size);`

  Funkce **calloc()** vrací ukazatel na alokovanou pamě•. Velikost alokované paměti se rovná **num * size**. Funkce **calloc()** může tedy například alokovat dostatek paměti pro pole, které má **num** prvků o velikost **size**.

  Funkce **calloc()** vrací ukazatel na první byte alokované oblasti. Pokud pro uspokojení požadavku není dostatek paměti, vrací se nulový ukazatel.Před použití vráceného ukazatele je důležité prověřit, zda není nulový.

- `void free(void *ptr);`

  Funkce **free()** dealokuje (uvolňuje) pamě•, na kterou ukazuje **ptr**. Tím je pamě• k dispozici pro příští alokaci.

  Je nutné, aby byla funkce **free()** volána jen s ukazatelem, který byl předtím nastaven pomocí některé z funkcí systému dynamické alokace, například **calloc()** nebo **malloc()**. Použití nevhodného ukazatele při volání pravděpodobně naruší mechanismus správy paměti a způsobí havárii systému.

- `void *realloc(void *ptr, size_t size);`

  Funkce **realloc()** mění velikost alokované paměti, na kterou ukazuje **ptr**, na velikost určenou argumentem **size**. Hodnota **size** může být větší, nebo menší než původní velikost. Vrací se ukazatel na blok paměti, jelikož funkce **realloc()** musí někdy původní blok paměti přemístit, aby jej mohla zvětšit. Pokud se to stane, je obsah starého bloku překopírován do nového, takže nedojde k žádné ztrátě informací.

  Pokud v haldě není dostatek paměti pro alokaci **size** bytů, vrací se nulový ukazatel. To znamená, že je důležité si ověřit úspěšnost volání **realloc()**.

Tato funkce alokuje pamě• potřebnou pro uložení struktury typu **adr**.

Příklad:

```
#include <stdio.h>
#include <stdlib.h>

struct adr
{
 char jmeno[40];
 char ulice[40];
 char mesto[40];
```

```
 char stat[40];
};
.
.
.
struct adr *get_struct(void)
{
 struct adr *p;

 if((p = malloc(sizeof(struct adr))) == NULL)
 {
  printf("Chyba pri alokaci pameti");
  exit(1);
 }
 return p;
}
```

Následující funkce vrací ukazatel na dynamicky alokované pole 100 čísel typu float.

Příklad:

```
#include <stdio.h>
#include <stdlib.h>

float *get_mem(void)
{
 float *p;

 p = calloc(100, sizeof(float));
 if(!p)
 {
  printf("Chyba pri alokaci pameti");
  exit(1);
 }
 return p;
}
```

Následující program nejprve alokuje místo pro 10 uživatelem zadaných řetězců a pak je uvolní.

Příklad 12.1:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 char *str[10];
 int i;

 printf("Zadej 10 retezcu:\n");
 for(i=0; i<10; i++)
 {
  if((str[i] = malloc(128)) == NULL)
  {
   printf("Chyba pri alokaci");
   exit(1);
  }
```

```
 gets(str[i]);
}

/* uvolneni pameti */
for(i=0; i<10; i++)
 free(str[i]);
printf("Alokace i uvolneni pameti probehlo uspesne!");

return 0;
}
```

Poslední příklad nejprve alokuje 17 znaků, zkopíruje do tohoto místa řetězec "toto je 16 znaku" a pak použije **realloc()** pro zvětšení velikosti na 18, aby se dala na konec řetězce vložit tečka.

Příklad 12.2:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
 char *p;

 p = malloc(17);
 if(!p)
 {
  printf("Chyba pri alokaci pameti");
  exit(1);
 }
 strcpy(p, "toto je 16 znaku");

 p = realloc(p, 18);
 if(!p)
 {
  printf("Chyba pri realokaci pameti");
  exit(1);
 }
 strcat(p, ".");

 printf(p);
 free(p);

 return 0;
}
```

12.2 Seznam

Časová náročnost: 25 minut

Seznam je dynamická datová struktura, která mezi svými členy obsahuje kromě datové části i vazebný člen, který ukazuje na následující prvek seznamu, nebo **NULL**, je-li posledním prvkem. Minimum statických dat, které pro správu takového seznamu potřebujeme, je ukazatel na jeho první prvek. Dále přirozeně potřebujeme podpůrné funkce. Popsaný seznam se rovněž nazývá **jednosměrný seznam**. Postupně totiž můžeme projít od začátku seznamu až k jeho konci, nikoliv však naopak. Existuje ještě jedna varianta seznamu, mající vazebné členy dva. Jeden ukazuje opět na následníka, druhý na předchůdce.

Správné pochopení práce se seznamem si ukážeme na příkladu. Mějme za úkol spočítat četnost výskytu všech slov ve vstupním textu. Na závěr vytiskneme tabulku, v níž bude uvedena četnost výskytu a řetězec, představující slovo. Jména funkcí i proměnných v programu jsou volena s ohledem na jejich maximální popisnost. Vstupem může být soubor, je-li zadán jako první argument příkazového řádku. Jinak je vstupem standardní vstup.

Příklad 12.3:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define DELKA_RADKU 80
#define PAGE 24

typedef struct struct_info {
                int pocet;
                char *slovo;
                struct struct_info *dalsi;
               } info;

void pridej_slovo(info **prvni, char *s)
{
 info *prvek;
 prvek = *prvni;
 /* dokud neni na konci, porovnava nove s ulozenym */
 while (prvek != NULL)
   {
    /* shoda nalezena, pocet se zvetsil */
    if (strcmp(s, prvek->slovo) == 0)
      {
       (prvek->pocet)++;
       return;
      }
    prvek = prvek->dalsi;               /* prejdi na naslednika */
   }
 /* shoda nenalezena je nutno vytvorit novy prvek */
 prvek = malloc(sizeof(info));
 /* naalokovat i misto pro kopii slova */
 prvek->slovo = malloc(strlen(s) + 1);
 strcpy(prvek->slovo, s);
 /* nastavi pocet vyskytu na jedna,prida prvek na zacatek seznamu */
 prvek->pocet = 1;
 prvek->dalsi = *prvni;
 *prvni = prvek;
```

```
}

void tiskni(info *prvni)
{
 int vytisknuto = 0;
 while (prvni != NULL)  /* dokud nejsme na konci seznamu */
    {
    printf("%4d..%s\n", prvni->pocet, prvni->slovo);
    prvni = prvni->dalsi;
    vytisknuto++;
        /* tiskne stukturu prvek a pocita, kolikrat kontroluje
        totoz vystup na obrazovku kvuli pozastaveni vypisu */
    if ((vytisknuto % PAGE) == 0)
      {
       printf("pro pokracovani stiskni enter");
       getchar();
       printf("\n");
      }
   }
}

int main(int argc, char **argv)
{
 info *prvni = NULL;
 char radek[DELKA_RADKU + 1], *slovo;
 /* retezec obsahuje mezeru TAB NL a CR */
 char oddelovace[] = " \t\\n\\r";
 FILE *f;

 if (argc != 1)
   f = fopen(argv[1], "rt"); /* otevreni textoveho streamu */
 else
   f = stdin;  /* pouzije standardni vstup */
 if (f == NULL)
   return(1);
 /* dokud neni konec vstupu cte radek za radkem a vybira z nej slova */
 while (fgets(radek, DELKA_RADKU, f) != NULL)
    {
        /* nastaveni funkce,navrat prvniho slova z radku */
    slovo = strtok(radek, oddelovace);
    while (slovo != NULL)
      {
       pridej_slovo(&prvni, slovo);  /* slovo do seznamu */
       slovo = strtok(NULL, oddelovace); /* ziskani dalsiho slova z radku */
      }
   }
 tiskni(prvni);
 return 0;
}
```

Za **slova** považuje program řetězce oddělené specifikovanými znaky. Tuto skutečnost můžeme ovšem měnit úpravou řetězce **oddelovace**, který obsahuje mezeru a **escape sekvence** oddělovačů. V cyklu **while** je funkcí **gets()** načítán řádek za řádkem, dokud **gets()** nevrátí **NULL** a program postupuje dál. Načtený řádek je na slova rozdělen pomocí funkce **strtok**. Pro ni je prohledávanou oblastí řetězec **radek** a oddělovači slov jsou znaky z řetězce **oddelovace**. Poprvé je nutno načíst další **slovo** hned po načtení řádku a nastavit tak prohledávanou oblast. Další čtení až do konce řádku probíhají tak dlouho, dokud není **slovo** ukazatel na **NULL**. Vnitřní cyklus zpracování řádku je ukončen a probíhá načítání dalšího řádku.

Pro každé načtené slovo je zavolána funkce **pridej_slovo** s argumenty **prvni** a **slovo**. Proměnná **prvni** představuje ukazatel na seznam, přesněji na strukturu námi vytvořeného typu **struct_info**.

Při prvním volání funkce **pridej_slovo** je seznam prázdný, funkce obdrží hodnotu **NULL**, proto je prohledávání seznamu přeskočeno a pomocí funkce **malloc** je alokována pamě• pro položku velikosti **sizeof(struct_info)**. Jakmile je tato dynamická položka vytvořena, je v ní naalokován řetězec délky odpovídající slovu **s** a řetězcové zarážce, konkrétně **malloc(strlen(s)+1)**. Do vytvořeného prostoru je slovo **s** nakopírováno pomocí **strcpy**. Dosud jsme se stále odkazovali na slovo umístěné ve zpracovávaném řádku. Bez vytvoření kopie bychom jej načtením dalšího řádku ztratili. Kromě kopie řetězce musíme nastavit v nové položce seznamu počet výskytů na jedna.

Pak přichází klíčové místo. Připojení položky do seznamu. V našem případě připojíme stávající seznam přes vazební člen **prvek->dalsi** k nově vytvořenému prvku a ten se následně stává novým prvním prvkem seznamu, ***prvni = prvek**.

Když je v seznamu nejméně jeden prvek, musí být při přidávání slova nejprve prohlédnut stávající seznam. Provádíme to prvek za prvkem, kdy posouváme pomocný ukazatel **prvek** na další položku seznamu pomocí **prvek = prvek->dalsi**, dokud se nedostaneme na konec seznamu, indikovaný hodnotou **NULL** ve vazebním členu **dalsi**. Pokud při průchodu narazíme na výskyt stejného řetězce, jako je přidávané slovo, tedy je splněna podmínka **strcmp(s, prvek->slovo)**, zvýšíme počet výskytů slova v položce a opustíme funkci **pridej_slovo**.

Zkrácený výpis:

```
   1..POSLEDNI
   1..PRIDAT
   1..JESTE
   1..UMIM!
   1..SOUBOR
pro pokracovani stiskni enter
   1..JE
   1..TOTO
   1..WUER
   1..OIEWR
   2..83945
   2..EWIRIWR
```

12.3 Pole ukazatelů

Časová náročnost: 5 minut

Velmi pohodlnou dynamickou datovou strukturou je dynamické **pole ukazatelů**. Princip je jednoduchý. Požádáme o pamě• pro dostatečně veliké pole ukazatelů. Dále již pracujeme stejně, jako by pole bylo statické (až na ten ukazatel na pole navíc). K jednotlivým prvkům můžeme přistupovat pomocí indexu. Nesmíme zapomenout, že prvky jsou ukazatele, a že tedy musíme pamě•, na kterou ukazují, také alokovat. Díky funkci **realloc()** máme možnost snadno měnit počet prvků našeho pole ukazatelů. Navíc s tím, že naše dynamická data svou adresu nemění a funkce **realloc()** přenese správné (stále platné) adresy do nového (většího) bloku ukazatelů. My k nim přistupujeme pomocí stejné proměnné, indexy všech prvků zůstávají stejné, jen jich přibylo.

Cvièení

1) Napište program, který oznámí, kolik paměti je pro váš program přibližně k dispozici. (Rada: alokujte malé kousky paměti do té doby, dokud požadavek na alokaci neselže) Řešení

2) Co je špatně na této části programu?

```
char *p;

*p = malloc(10);

gets(p);
```

Řešení

3) Napište program, který dynamicky alokuje pamě• pro jedno číslo double. Přiřaďte tomuto číslu hodnotu 99.01, vypište je a pamě• uvolněte. Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

Závěr

Časová náročnost: 5 minut

Toto je závěrečná stránka seriálu o jazyku C. Nyní můžete pokračovat dále na úvodní stránku, kde můžete začít s výukou jazyka C++, případně si zopakovat získané znalosti o jazyku C.

Tato elektronická publikace vznikla jako součást diplomové práce s názvem:
Výuková podpora a testy pro C a OOP

Použitá literatura:

- Herout, Pavel: Učebnice jazyka C, III. upravené vydání, České Budějovice, KOPP, 1996, 269 s.
- Schildt, Herbert: Nauč se sám C, III. edice, Praha, SoftPress, 2001, 620 s.
- Šaloun, Petr: Jazyk C pro zelenáče, Praha, Neokortex, 1999, 208 s.
- URL http://www.zive.cz, Živě o počítačích a internetu, (únor 2002)
- URL http://www.builder.cz, Informační server o programování, (únor 2002)

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 char *str[10];
 int i;

 printf("Zadej 10 retezcu:\n");
 for(i=0; i<10; i++)
 {
  if((str[i] = malloc(128)) == NULL)
  {
   printf("Chyba pri alokaci");
   exit(1);
  }
  gets(str[i]);
 }

 /* uvolneni pameti */
 for(i=0; i<10; i++)
  free(str[i]);
 printf("Alokace i uvolneni pameti probehlo uspesne");

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
 char *p;

 p = malloc(17);
 if(!p)
 {
  printf("Chyba pri alokaci pameti");
  exit(1);
 }
 strcpy(p, "toto je 16 znaku");

 p = realloc(p, 18);
 if(!p)
 {
  printf("Chyba pri realokaci pameti");
  exit(1);
 }
 strcat(p, ".");

 printf(p);
 free(p);

 return 0;
}
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define DELKA_RADKU 80
#define PAGE 24

typedef struct struct_info {
                int pocet;
                char *slovo;
                struct struct_info *dalsi;
               } info;

void pridej_slovo(info **prvni, char *s)
{
 info *prvek;
 prvek = *prvni;
 /* dokud neni na konci, porovnava nove s ulozenym */
 while (prvek != NULL)
   {
    /* shoda nalezena, pocet se zvetsil */
    if (strcmp(s, prvek->slovo) == 0)
      {
       (prvek->pocet)++;
       return;
      }
    prvek = prvek->dalsi;               /* prejdi na naslednika */
   }
 /* shoda nenalezena je nutno vytvorit novy prvek */
 prvek = malloc(sizeof(info));
 /* naalokovat i misto pro kopii slova */
 prvek->slovo = malloc(strlen(s) + 1);
 strcpy(prvek->slovo, s);
 /* nastavi pocet vyskytu na jedna,prida prvek na zacatek seznamu */
 prvek->pocet = 1;
 prvek->dalsi = *prvni;
 *prvni = prvek;
}

void tiskni(info *prvni)
{
 int vytisknuto = 0;
 while (prvni != NULL)  /* dokud nejsme na konci seznamu */
    {
    printf("%4d..%s\n", prvni->pocet, prvni->slovo);
    prvni = prvni->dalsi;
    vytisknuto++;
        /* tiskne stukturu prvek a pocita, kolikrat kontroluje
        totoz vystup na obrazovku kvuli pozastaveni vypisu */
    if ((vytisknuto % PAGE) == 0)
      {
       printf("pro pokracovani stiskni enter");
       getchar();
       printf("\n");
      }
   }
}

int main(int argc, char **argv)
{
 info *prvni = NULL;
 char radek[DELKA_RADKU + 1], *slovo;
 /* retezec obsahuje mezeru TAB NL a CR */
 char oddelovace[] = " \t\\n\\r";
 FILE *f;
```

```
 if (argc != 1)
   f = fopen(argv[1], "rt"); /* otevreni textoveho streamu */
 else
   f = stdin;  // pouzije standardni vstup
 if (f == NULL)
   return(1);
 /* dokud neni konec vstupu cte radek za radkem a vybira z nej slova */
 while (fgets(radek, DELKA_RADKU, f) != NULL)
    {
        /* nastaveni funkce,navrat prvniho slova z radku */
    slovo = strtok(radek, oddelovace);
    while (slovo != NULL)
      {
       pridej_slovo(&prvni, slovo);  /* slovo do seznamu */
       slovo = strtok(NULL, oddelovace); /* ziskani dalsiho slova z radku */
      }
   }
 tiskni(prvni);
 return 0;
}
```

Řešení

Zadání: 1) Napište program, který oznámí, kolik paměti je pro váš program přibližně k dispozici. (Rada: alokujte malé kousky paměti do té doby, dokud požadavek na alokaci neselže)

Řešení:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 char *p;
 long l = 0;

 do
 {
  p = malloc(1000);
  if(p) l+=1000;
 }
 while(p);

 printf("Priblizna hodnota volne pameti je: %ld bytu", l);

 return 0;
}
```

Zpět

Zadání: 2) Co je špatně na této části programu?

```
char *p;

*p = malloc(10);

gets(p);
```

Řešení:

Příkaz

```
*p = malloc(10);
```

By měl správně být

```
p = malloc(10);
```

Chybí také kontrola, zda je hodnota vrácená funkcí **malloc()** platný ukazatel.

Zpět

Zadání: 3) Napište program, který dynamicky alokuje pamě• pro jedno číslo double. Přiřaďte tomuto číslu hodnotu 99.01, vypište je a pamě• uvolněte.

Řešení:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 double *p;

 p = malloc(sizeof(double));
 if(!p)
 {
  printf("Chyba pri alokaci pameti");
  exit(1);
 }

 *p = 99.01;
 printf("%f", *p);
 free(p);

 return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 char *p;
 long l = 0;

 do
 {
  p = malloc(1000);
  if(p) l+=1000;
 }
 while(p);

 printf("Priblizna hodnota volne pameti je: %ld bytu", l);

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 double *p;

 p = malloc(sizeof(double));
 if(!p)
 {
  printf("Chyba pri alokaci pameti");
  exit(1);
 }

 *p = 99.01;
 printf("%f", *p);
 free(p);

 return 0;
}
```

```c
#include <stdio.h>

typedef signed char smallint;

int main(void)
{
 smallint i;

 for(i=0; i<10; i++)
         printf("%d ", i);

 return 0;
}
```

```c
#include <stdio.h>

#define SIZE 80

typedef char * string;

void strcpy1(string d, string s)
{
 while ((*d++ = *s++) != 0) ;
}

void strcpy2(string d, string s)
{
 int i = 0;
 while ((d[i] = s[i]) != 0)
   i++;
}

int main()
{
 string s1 = "prvni (1.) retezec",
        s2 = "druhy (2.) retezec";
 char   d1[SIZE],
        d2[SIZE];
 strcpy1(d1, s1);
 strcpy2(d2, s2);
 printf("d1[]:%s\n", d1);
 printf("d2[]:%s\n", d2);
 return 0;
}
```

```c
#include <stdio.h>

enum {klavesnice, CPU, monitor, tiskarna} comp;

int main(void)
{
 comp = CPU;
 printf("%d", comp);

 return 0;
}
```

```c
#include <stdio.h>

struct
{
 int i;
 double d;
 char str[80];
} s;

int main(void)
{
 printf("Zadejte cele cislo: ");
 scanf("%d", &s.i);
 printf("Zadejte desetinne cislo: ");
 scanf("%lf", &s.d);
 printf("Zadejte retezec: ");
 scanf("%s", &s.str);

 printf("Bylo zadano: \n");
 printf("%d %lf %s", s.i, s.d, s.str);

 return 0;
}
```

```c
#include <stdio.h>

struct
{
 int i;
 int j;
} s;

int main(void)
{
 int i;

 i = 10;
 s.i = 100;
 s.j = 101;
 printf("%d %d %d", i, s.i, s.j);

 return 0;
}
```

Řešení

Zadání: 1) Ukažte jak udělat u **UL** nové jméno pro **unsigned long**. Napište krátký program, který deklaruje proměnnou pomocí **UL**, přiřadí jí hodnotu a vypíše hodnotu proměnné na obrazovku.

Řešení:

```
#include <stdio.h>

typedef unsigned long UL;

int main(void)
{
 UL cislo;

 cislo = 123;
 printf("%lu", cislo);

 return 0;
}
```

Zpět

Zadání: 2) Co je špatně na tomto zápisu?

```
typedef balance float;
```

Řešení:

Příkaz **typedef** má nesprávné pořadí argumentů. Správný formát **typedef** je:

```
typedef staré-jméno nové-jméno;
```

Zpět

Zadání: 3) Vytvořte výčtový typ obsahující dny v týdnu.

Řešení:

```
enum {pondeli, utery, streda, ctvrtek, patek, sobota, nedele} dny_tydne;
```

Zpět

Zadání: 4) Je tento úsek programu správný? Pokud ne, proč?

```

```

```
enum auta {Ford, Chrysler, GM} vyrobce;

vyrobce = GM;
printf("Auto vyrobil: %s",vyrobce);
```

Řešení:

Ne, protože výčtovou konstantu nelze vypisovat jako řetězec.

Zpět

Zadání: 5) Co je špatně na tomto kousku programu?

```
stuct
{
 int i;
 long l;
 char str[80];
} s;
.
.
.
i = 10;
.
```

Řešení:

Proměnná **i** je prvkem struktury **s**. Proto ji nelze použít samotnou. Musí být zpřístupněna pomocí s a tečkového operátoru takto:

```
s.i = 10;
```

Zpět

Zadání: 6) Co je struktura a co je unie?

Řešení:

Struktura je pojmenovaná skupina proměnných, které spolu nějak souvisejí. Unie definuje pamě•ovou oblast, která je sdílena dvěmi nebo více proměnnými různých typů.

Zpět

Zadání: 7) Co je špatně na tomto úseku programu?

```
struct
{
 int a;
 char b;
```

```
 float c;
} myvar, *p;

p = &myvar;
p.a = 10;
```

Řešení:

Jelikož je **p** ukazatel na strukturu, musíte pro odkaz na prvek použít šipkový a ne tečkový operátor.

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>

typedef unsigned long UL;

int main(void)
{
 UL cislo;

 cislo = 123;
 printf("%lu", cislo);

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 char str[80] = "Toto je cvicny soubor";
 FILE *soubor;
 char ch, *p;

 /* otevreni souboru pro zapis */
 if((soubor = fopen("muj", "w")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 /* zapis do souboru a jeho ulozeni */
 p = str;
 while(*p)
  if(fputc(*p++, soubor) == EOF)
  {
        printf("Pri zapisu do souboru doslo k chybe\n");
        exit(1);
  }

 fclose(soubor);

 /* otevreni souboru pro cteni */
 if((soubor = fopen("muj", "r")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }


 /* cteni souboru */
 while((ch = fgetc(soubor)) != EOF)
  putchar(ch);

 fclose(soubor);

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 char str[80];
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc != 2)
 {
  printf("Chyba je nutne zadat jmeno souboru");
  exit(1);
 }

 /* otevreni souboru pro zapis */
 if((soubor = fopen(argv[1], "w")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 printf("Pro ukonceni zadejte prazdny radek\n");

 do
 {
  printf(": ");
  gets(str);
  strcat(str, "\n"); /* pridani noveho radku */
  if(*str != '\n')
        fputs(str, soubor);
 }
 while(*str != '\n');

 fclose(soubor);

 /* otevreni souboru pro cteni */
 if((soubor = fopen(argv[1], "r")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 /* cteni souboru */
 do
 {
  fgets(str, 79, soubor);
  if(!feof(soubor))
        printf(str);
 }
 while(!feof(soubor));


 fclose(soubor);

 return 0;
}
```

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
 FILE *soubor;
 int i;
 char *s, *jmeno = "soubor1.txt";
 if ((soubor = fopen(jmeno, "wt")) == NULL)
   return 1;
 s = "Toto je textovy soubor vytvoreny v jazyce C.\n";
 fputs(s, soubor);
 for (i = 0; i<10; i++)
   {
    fprintf(soubor, "%5d", i);
   }
 fputs("\n", soubor);
 s = "Jeste pridat posledni radek na konec.\n";
 fputs(s, soubor);
 if (fclose(soubor) == EOF)
   return 1;
 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

double d[10] = {10.23, 19.87, 100.2, 0.258,
11.15, 95.23, 21.21, 458.03, 73.321, 3.14};

int main(void)
{
 int i;
 FILE *soubor;

 if((soubor = fopen("soubor2", "wb")) == NULL)
 {
  printf("Soubor nelze otevrit.\n");
  exit(1);
 }

 for(i=0; i<10; i++)
    if(fwrite(&d[i], sizeof(double), 1, soubor) != 1)
        {
         printf("Chyba pri zapisu.\n");
         exit(1);
        }
 fclose(soubor);

 if((soubor = fopen("soubor2", "rb")) == NULL)
 {
  printf("Soubor nelze otevrit.\n");
  exit(1);
 }

 /* vymazani pole */
 for(i=0; i<10; i++) d[i] = 0.0;

 for(i=0; i<10; i++)
    if(fread(&d[i], sizeof(double), 1, fp) != 1)
        {
         printf("Chyba pri zapisu.\n");
         exit(1);
        }
 fclose(soubor);

 /* vypis pole */
 for(i=0; i<10; i++)
    printf("%f ", d[i]);

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 long pozice;
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc != 2)
 {
  printf("Chyba je nutne zadat jmeno souboru");
  exit(1);
 }

 /* otevreni souboru */
 if((soubor = fopen(argv[1], "rb")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 /* vyhledavani */
 printf("Zadejte cislo vyhledavaneho byte: \n");
 scanf("%ld",&pozice);
 if(fseek(soubor, pozice, SEEK_SET))
 {
  printf("Chyba vyhledavani\n");
  exit(1);
 }

 printf("Hodnota na pozici %ld je %d", pozice, getc(soubor));

 fclose(soubor);

 return 0;
}
```

Řešení

Zadání: 1) Napište program, který vypíše obsah textového souboru, zadaného jako argument příkazové řádky, na obrazovku.

Řešení:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 char ch;
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc != 2)
 {
  printf("Chyba je nutne zadat jmeno souboru");
  exit(1);
 }

 /* otevreni souboru pro cteni */
 if((soubor = fopen(argv[1], "r")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 while((ch=fgetc(soubor)) != EOF)
         putchar(ch);


 fclose(soubor);

 return 0;
}
```

Zpět

Zadání: 2) Napište program, který kopíruje textový soubor. Jméno zdrojového a cílového souboru se zadává na příkazovém řádku. Pro kopírování souborů použijte funkce **fgets()** a **fputs()**. Proveďte úplnou kontrolu chyb.

Řešení:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 char str[128];
 FILE *zdroj, *cil;

 /* kontrola argumentu prikazoveho radku */
 if(argc < 3)
 {
  printf("Pouziti: cvi10_2 <zdroj> <cil>\n");
  exit(1);
 }

 /* otevreni zdrojoveho souboru */
 if((zdroj = fopen(argv[1], "r")) == NULL)
 {
  printf("Zdrojovy soubor nelze otevrit.\n");
  exit(1);
 }

 /* otevreni ciloveho souboru */
 if((cil = fopen(argv[2], "w")) == NULL)
 {
  printf("Cilovy soubor nelze otevrit.\n");
  exit(1);
 }

 while(!feof(zdroj))
 {
  fgets(str, 127, zdroj);
  if(ferror(zdroj))
  {
        printf("Chyba pri cteni.\n");
        break;
  }
  if(!feof(zdroj))
        fputs(str, cil);
  if(ferror(cil))
  {
        printf("Chyba pri zapisu.\n");
        break;
  }
 }

 if(fclose(zdroj) == EOF)
 {
  printf("Chyba pri zavirani zdrojoveho souboru.\n");
  exit(1);
 }

 if(fclose(cil) == EOF)
 {
```

```
 printf("Chyba pri zavirani ciloveho souboru.\n");
 exit(1);
}

 return 0;
}
```

Zpět

Zadání: 3) Napište program, který umožňuje uživateli zadat libovolný počet hodnot **double** (max 32 767) a zapsat je do souboru tak, jak jsou zadávány. Tento soubor pojmenujte VALUES. Počet zadaných hodnot zapište do souboru COUNT.

Řešení:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 FILE *soubor1, *soubor2;
 double d;
 int i;

 if((soubor1 = fopen("values", "wb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 if((soubor2 = fopen("count", "wb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 d = 1.0
 for(i=0; d!=0.0 && i<32767; i++)
 {
  printf("Zadejte cislo (0 = konec): ");
  scanf("%lf", &d);
  fwrite(&d, sizeof d, 1, soubor1);
 }

 fwrite(&i, sizeof i, 1, soubor2);

 fclose(soubor1);
 fclose(soubor2);

 return 0;
}
```

Zpět

Zadání: 4) Napište program, který používá soubory ze zadání **3)**. Program nejprve přečte ze souboru COUNT počet prvků v souboru VALUES. Poté čte a vypisuje hodnoty ze souboru VALUES.

Řešení:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 FILE *soubor1, *soubor2;
 double d;
 int i;

 if((soubor1 = fopen("values", "rb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 if((soubor2 = fopen("count", "rb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 fread(&i, sizeof i, 1, soubor2);

 for(; i>0; i--)
 {
  fread(&d, sizeof d, 1, soubor1);
  printf("%f\n",d);
 }

 fclose(soubor1);
 fclose(soubor2);

 return 0;

}
```

Zpět

Zadání: 5) Napište program, který vyhledává v souboru zadaném jako argument příkazového řádku určitou celočíselnou hodnotu zadanou také jako argument příkazového řádku. Je-li hodnota nalezena, nechejte program vypsat její bytovou pozici od začátku souboru.

Řešení:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 unsigned char ch, val;
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc < 3)
 {
  printf("Pouziti: cvi10_5 <soubor> <hodnota>\n");
  exit(1);
 }

 if((soubor = fopen(argv[1], "rb")) == NULL)
 {
  printf("Soubor nelze otevrit.\n");
  exit(1);
 }

 val = atoi(argv[2]);

 while(!feof(soubor))
 {
  ch = fgetc(soubor);
  if(ch == val)
        printf("Hodnota nalezena na %ld\n", ftell(soubor));
 }

 fclose(soubor);

 return 0;
}
```

Zpět

Zadání: 6) Napište program, který spočítá počet bytů v souboru (textovém nebo binárním) a zobrazí výsledek. Jméno souboru zadá uživatel jako argument příkazové řádky.

Řešení:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 FILE *soubor;
 unsigned velikost;

 if(argc!=2)
 {
  printf("Chyba je nutne zadat jmeno souboru.\n");
  exit(1);
 }

 if((soubor = fopen(argv[1], "rb")) == NULL)
 {
  printf("Soubor nelze otevrit.\n");
  exit(1);
 }

 velikost = 0;
 while(!feof(soubor))
 {
  fgetc(soubor);
  if(ferror(soubor))
  {
        printf("Chyba souboru.\n");
        exit(1);
  }
  velikost++;
 }

 printf("Velikost souboru je %u bytu.",velikost-1);
 fclose(soubor);

 return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 char ch;
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc != 2)
 {
  printf("Chyba je nutne zadat jmeno souboru");
  exit(1);
 }

 /* otevreni souboru pro cteni */
 if((soubor = fopen(argv[1], "r")) == NULL)
 {
  printf("Soubor nelze otevrit\n");
  exit(1);
 }

 while((ch=fgetc(soubor)) != EOF)
         putchar(ch);


 fclose(soubor);

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 char str[128];
 FILE *zdroj, *cil;

 /* kontrola argumentu prikazoveho radku */
 if(argc < 3)
 {
  printf("Pouziti: cvi10_2 <zdroj> <cil>\n");
  exit(1);
 }

 /* otevreni zdrojoveho souboru */
 if((zdroj = fopen(argv[1], "r")) == NULL)
 {
  printf("Zdrojovy soubor nelze otevrit.\n");
  exit(1);
 }

 /* otevreni ciloveho souboru */
 if((cil = fopen(argv[2], "w")) == NULL)
 {
  printf("Cilovy soubor nelze otevrit.\n");
  exit(1);
 }

 while(!feof(zdroj))
 {
  fgets(str, 127, zdroj);
  if(ferror(zdroj))
  {
        printf("Chyba pri cteni.\n");
        break;
  }
  if(!feof(zdroj))
        fputs(str, cil);
  if(ferror(cil))
  {
        printf("Chyba pri zapisu.\n");
        break;
  }
 }

 if(fclose(zdroj) == EOF)
 {
  printf("Chyba pri zavirani zdrojoveho souboru.\n");
  exit(1);
 }

 if(fclose(cil) == EOF)
 {
  printf("Chyba pri zavirani ciloveho souboru.\n");
  exit(1);
 }

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 FILE *soubor1, *soubor2;
 double d;
 int i;

 if((soubor1 = fopen("values", "wb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 if((soubor2 = fopen("count", "wb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 d = 1.0
 for(i=0; d!=0.0 && i<32767; i++)
 {
  printf("Zadejte cislo (0 = konec): ");
  scanf("%lf", &d);
  fwrite(&d, sizeof d, 1, soubor1);
 }

 fwrite(&i, sizeof i, 1, soubor2);

 fclose(soubor1);
 fclose(soubor2);

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 FILE *soubor1, *soubor2;
 double d;
 int i;

 if((soubor1 = fopen("values", "rb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 if((soubor2 = fopen("count", "rb")) == NULL)
 {
  printf("Nelze otevrit soubor.\n");
  exit(1);
 }

 fread(&i, sizeof i, 1, soubor2);

 for(; i>0; i--)
 {
  fread(&d, sizeof d, 1, soubor1);
  printf("%f\n",d);
 }

 fclose(soubor1);
 fclose(soubor2);

 return 0;

}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 unsigned char ch, val;
 FILE *soubor;

 /* kontrola argumentu prikazoveho radku */
 if(argc < 3)
 {
  printf("Pouziti: cvi10_5 <soubor> <hodnota>\n");
  exit(1);
 }

 if((soubor = fopen(argv[1], "rb")) == NULL)
 {
  printf("Soubor nelze otevrit.\n");
  exit(1);
 }

 val = atoi(argv[2]);

 while(!feof(soubor))
 {
  ch = fgetc(soubor);
  if(ch == val)
        printf("Hodnota nalezena na %ld\n", ftell(soubor));
 }

 fclose(soubor);

 return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 FILE *soubor;
 unsigned velikost;

 if(argc!=2)
 {
  printf("Chyba je nutne zadat jmeno souboru.\n");
  exit(1);
 }

 if((soubor = fopen(argv[1], "rb")) == NULL)
 {
  printf("Soubor nelze otevrit.\n");
  exit(1);
 }

 velikost = 0;
 while(!feof(soubor))
 {
  fgetc(soubor);
  if(ferror(soubor))
  {
        printf("Chyba souboru.\n");
        exit(1);
  }
  velikost++;
 }

 printf("Velikost souboru je %u bytu.",velikost-1);
 fclose(soubor);

 return 0;
}
```

```c
#include <stdio.h>

#define VELIKOST 10

int main(void)
{
 int sqr[VELIKOST];
 int i;

 for(i=0; i<11; i++)
    sqr[i] = i*i;

 for(i=0; i<11; i++)
    printf("Druha mocnina cisla %d je %d\n",i,sqr[i]);

 return 0;
}
```

```c
#include <stdio.h>

#define NEJVICE 31

int main(void)
{
 int temp[NEJVICE];
 int i, dny, min, max, prum = 0;

 /* zjisteni poctu dnu */
 printf("Kolik dnu ma tento mesic? ");
 scanf("%d",&dny);

 /* nacteni teplot v jednotlivych dnech */
 for(i=0; i<dny; i++)
 {
  printf("Zadejte teplotu pro den %d: ",i+1);
  scanf("%d",&temp[i]);
 }

 /* zjisteni prumerne teploty */
 for(i=0; i<dny; i++)
    prum = prum + temp[i];
 printf("\nPrumerna teplota: %f",(float)prum/dny);

 /* nalezeni minimalni a maximalni teploty */
 min = max = temp[0];
 for(i=0; i<dny; i++)
 {
  if(min>temp[i]) min = temp[i];
  if(max<temp[i]) max = temp[i];
 }

 printf("\nMinimalni teplota: %d",min);
 printf("\nMaximalni teplota: %d",max);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 char *cp, c;
 int *ip, i;
 float *fp, f;
 double *dp, d;

 cp = &c;
 ip = &i;
 fp = &f;
 dp = &d;

 /* vytiskneme aktualni hodnoty */
 printf("%p %p %p %p\n", cp, ip, fp, dp);

 /* inkrementace */
 cp++;
 ip++;
 fp++;
 dp++;

 /* vytiskneme zmenene hodnoty */
 printf("%p %p %p %p\n", cp, ip, fp, dp);

 return 0;
}
```

```c
#include <stdio.h>

int main()
{
 char text[] = "world", *new1 = text, *new2 = text;
 printf("%s\t%s\t%s\n", text, new1, new2);
 /* menime hodnotu ukazatele, nedochazi ke kopirovani retezcu */
 new1 = "hello";
 printf("%s\t%s\t%s\n", text, new1, new2);
 printf("%s\n", "hello world" + 2);     /* posun retezce */

 return 0;
}
```

```c
#include <stdio.h>

int main()
{
 int pole2[4][5];
 int i,j;

 for(i=0; i<4; i++)
 {
    for(j=0; j<5; j++)
        {
           pole2[i][j] = i*j;
         }
 }

 for(i=0; i<4; i++)
 {
    for(j=0; j<5; j++)
        {
           printf("%d ", pole2[i][j]);
        }
        printf("\n");
 }

 return 0;
}
```

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

#define POCET 10000
#define RND_START 1234

int float_sort (const float *a, const float *b)
{
 return (*a - *b);
}

void test (float *p, unsigned int pocet)
/* otestuje zda dane pole je setrideno vzestupne */
{
 int chyba = 0;
 for (; !chyba && --pocet > 0; p++)
   if (*p > *(p+1))
     chyba=1;
 puts ((chyba) ? "\npole neni setrideno\n" : "\npole je setrideno\n");
}

void vypln(float *p, int pocet)
/* vypni pole dane adresou a pocte prvku, nahodnymi cisly pomoci generatoru nahodnych cisel */
{
 srand(RND_START);
 while (pocet-- > 0)
    *p++ = (float) rand();
}

int main (void)
{
 static float pole [POCET];

 vypln (pole, POCET);
 clock(); /* zacatek pocitani casu */
 qsort(pole, POCET, sizeof(*pole), (int (*) (const void *, const void *)) float_sort);
 /* uplynuly cas v ticich preveden na sekundy */
 printf ("Trideni qsort trvalo %.2fs\n", ((float) clock()) / CLOCKS_PER_SEC);
 test (pole, POCET);

 return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define N 13

int qsort_stringlist(const void *e1, const void *e2)
{
 return strcmp(*(char **)e1, *(char **)e2);
}

int main()
{
 char *name[N] =
  {
   "chyba", "leden", "unor", "brezen", "duben",
   "kveten", "cerven", "cervenec", "srpen",
   "zari", "rijen", "listopad", "prosinec"
  };
 int i;
 puts("jmena mesicu v roce (podle poradi):");
 for (i = 0; i < N; i++)
        printf("%2d. mesic je : %s\n", i, name[i]);
 puts("\nsetrideno:");
 qsort(name, N, sizeof(char *), qsort_stringlist);

 for (i = 0; i < N; i++)
   printf("%2d. %s\n", i, name[i]);
 return 0;
}
```

```
#include <stdio.h>
#include <string.h>

char *p[][2] = {
"Red Delicious", "cervene",
"Golden Delicious", "zlute",
"Winesap", "cervene",
"Gala", "oranzove",
"Lodi", "zelene",
"Mutsu", "zlute",
"Cortland", "cervene",
"Jonathan", "cervene",
"", "" /* ukonceni tabulky nulovymi retezci */
};

int main(void)
{
 int i;
 char jablko[80];

 printf("Zadejte jmeno odrudy jablka: ");
 gets(jablko);

 for(i=0; *p[i][0]; i++)
 {
  if(!strcmp(jablko, p[i][0]))
     printf("%s je %s\n", jablko, p[i][1]);
 }

 return 0;
}
```

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
 int i;
 for (i = 0; i < argc; i++)
   printf("%d\t%s\n", i, argv[i]);

 return 0;
}
```

Řešení

Zadání: 1) Napište program s cyklem **for**, který počítá od 0 do 9 a vypisuje čísla na obrazovku. Čísla vypisujte pomocí ukazatele.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int i, *p;

 p = &i;

 for(i=0; i<10; i++)
    printf("%d ",*p);

 return 0;
}
```

Zpět

Zadání: 2) Ukažte jak deklarovat ukazatel na **double**.

Řešení:

```
double *p;
```

Zpět

Zadání: 3) Napište program, který přiřazuje hodnotu proměnné nepřímo, pomocí ukazatele na tuto proměnnou.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int i, *p;

 p = &i;
 *p = 10;
 printf("%d",i);

 return 0;
}
```

Zpět

Zadání:

4) Co je špatně na tomto úseku programu?

```
#include <stdio.h>

int main(void)
{
 int i, count[10];

 for(i=0; i<100; i++)
 {
  printf("Zadejte cislo: ");
  scanf("%d",&count[i]);
 }
.
.
.
}
```

Řešení:

Program překračuje meze pole **count**. To má pouze 10 prvků, ale program s ním pracuje, jako by mělo prvků 100.

Zpět

Zadání: 5) Napište program, který načte deset čísel zadaných uživatelem a ohlásí zda jsou některá z nich shodná.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int i[10], j, k, souhlas;

 printf("Zadejte 10 cisel: \n");
 for(j=0; j<10; j++)
    scanf("%d", &i[j]);

 /* kontrola shody */
 for(j=0; j<10; j++)
 {
  souhlas = i[j];
  for(k=j+1; k<10; k++)
  {
   if(souhlas == i[k])
         printf("Nalezena shoda v cisle %d\n",souhlas);
  }
 }

 return 0;
}
```

Zpět

Zadání: 6) Co je špatně na tomto úseku programu?

```
int *p, i;
p = &i;
p = p * 8;
```

Řešení:

Ukazatel nelze násobit.

Zpět

Zadání: 7) Můžete k ukazateli přičíst neceločíselnou hodnotu?

Řešení:

Ne, můžete přičítat a odečítat pouze celočíselné hodnoty.

Zpět

Zadání: 8) Napište program, který přečte řetězec a pak jej vypíše na obrazovku pozpátku.

Řešení:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 char str[80];
 int i;

 printf("Zadejte retezec: ");
 gets(str);

 for(i=strlen(str-1); i>=0; i--)
    printf("%c", str[i]);

 return 0;
}
```

Zpět

Zadání: 9) Co je špatně na tomto programu?

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 char str[5];

 strcpy(str, "tento text");
 printf(str);

 return 0;
}
```

Řešení:

Řetězec **str** není dost dlouhý, aby se do něj vešel řetězec "tento text".

Zpět

Zadání: 10) Napište program, který definuje trojrozměrné pole velikosti 3x3x3 a naplní jej čísly 1 až 27.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int tri[3][3][3];
 int i, j, k, x;

 x = 1;
 for(i=0; i<3; i++)
    for(j=0; j<3; j++)
           for(k=0; k<3; k++)
           {
                tri[i][j][k] = x;
                x++;
                printf("%d ", tri[i][j][k]);
           }

 return 0;
}
```

Zpět

Zadání: 11) Napište program, který přebírá dva argumenty z příkazového řádku a zobrazte jejich součet.

Řešení:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 if(argc != 3)
 {
  printf("Musite zadat dve cisla!");
  exit(1);
 }
 printf("%f", atof(argv[1]) + atof(argv[2]));

 return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>

int main(void)
{
 int i, *p;

 p = &i;

 for(i=0; i<10; i++)
    printf("%d ",*p);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i, *p;

 p = &i;
 *p = 10;
 printf("%d",i);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i[10], j, k, souhlas;

 printf("Zadejte 10 cisel: \n");
 for(j=0; j<10; j++)
    scanf("%d", &i[j]);

 /* kontrola shody */
 for(j=0; j<10; j++)
 {
  souhlas = i[j];
  for(k=j+1; k<10; k++)
  {
   if(souhlas == i[k])
         printf("Nalezena shoda v cisle %d\n",souhlas);
  }
 }

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int tri[3][3][3];
 int i, j, k, x;

 x = 1;
 for(i=0; i<3; i++)
    for(j=0; j<3; j++)
           for(k=0; k<3; k++)
           {
                tri[i][j][k] = x;
                x++;
                printf("%d ", tri[i][j][k]);
           }

 return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 if(argc != 3)
 {
  printf("Musite zadat dve cisla!");
  exit(1);
 }
 printf("%f", atof(argv[1]) + atof(argv[2]));

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 printf("Preklad: %s, radek: %d, dne: %s, v: %s", __FILE__, __LINE__,
 __DATE__, __TIME__);

 return 0;
}
```

```c
#include <stdio.h>

/* Definujem CHAR_SET jako 256 nebo 128 */
#define CHAR_SET 256

int main(void)
{
 int i;
#if CHAR_SET == 256
 printf("Zobrazeni uplne sady ASCII znaku plus rozsireni.\n");
#else
 printf("Zobrazeni jen samotne sady ASCII znaku.\n");
#endif

 for(i=0;i<CHAR_SET;i++)
    printf("%c ",i);

 return 0;
}
```

Zadání: 1) Co je **__FILE__** a co představuje?

Řešení:

__FILE__ je předdefinované makro, které obsahuje jméno právě překládaného souboru.

Zpět

Zadání: 2) Pomocí **#ifdef** ukažte, jak podmíněně přeložit tento úsek programu v závislosti na tom, zda je nebo není definováno **DEBUG**.

```
if(!(j%2))
{
 printf("j = %d\n",j);
 j = 0;
}
```

Řešení:

```
#ifdef DEBUG
if(!(j%2))
{
 printf("j = %d\n",j);
 j = 0;
}
#endif
```

Zpět

Zadání: 3) Je tento úsek správný? Pokud ne, ukažte jak jej opravit.

```
#define NUMBER

#ifdef !NUMBER
.
.
.
#endif
```

Řešení:

Ne. U **#ifdef** nelze použít výraz jako je !NUMBER. Musíme tuto část opravit, například takto:

```
#ifndef NUMBER
.
.
.
#endif
```

Zpět

Zadání: 4) Jaký je rozdíl mezi použitím uvozovek a úhlových závorek u direktivy **#include**?

Řešení:

Když zadáme jméno souboru v **úhlových závorkách**, překladač hledá soubor způsobem, který je závislý na implementaci - ve standardním adresáři pro include. Když uzavřeme jméno souboru do **dvojitých uvozovek**, překladač se nejprve snaží nalézt soubor jiným způsobem, který je opět závislý na implementaci - v aktivním, pracovním adresáři. Pokud se mu to nepodaří, začne jej hledat, jako bychom jméno souboru zadali v úhlových závorkách.

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>

int main(void)
{
 char ch;

 ch = getchar();   /* nacitani znaku */
 printf("Zadali jste: %d",ch);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 char str[80];

 printf("Zadejte retezec: ");
 if(gets(str))  /* kontrola zda nedoslo k chybe */
    printf("Nacteny retezec: %s", str);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 printf("%.5d\n",10);
 printf("$%.2f\n",54.32);
 printf("%.10s", "Ne vsechno co je napsane se vytiskne\n");

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 printf("%d %o %x %X\n", 25, 25, 25, 25);
 printf("%e %E\n", 25.123, 25.123);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 char str[80];

 printf("Zadejte znaky: \n");
 scanf("%[a-zA-Z]",str);

 printf(str);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i,j;
 char op;

 printf("Zadejte operaci: ");
 scanf("%d%c%d",&i, &op, &j);

 switch(op)
 {
  case '+': printf("%d",i+j);
                        break;
  case '-': printf("%d",i-j);
                        break;
  case '*': printf("%d",i*j);
                        break;
  case '/': printf("%d",i/j);
                        break;
 }

 return 0;
}
```

```c
#include <stdio.h>
#include <math.h>

#define N 80
const od = 0;

int main(void)
{
 char b[N],
      *zahlavi = "\n x\tsin(x)\t  cos(x)";
 int x, po;
 double si, co, rad;

 printf("\nZadej do kolika stupnu:");
 gets(b);

 sscanf(b, "%d", &po);
 puts(zahlavi);
 for ( x = od; x < po; x++)
   {
    rad = x / 180.0 * M_PI;
    si = sin(rad);
    co = cos(rad);
    sprintf(b, "%2d %10.5f %10.5f", x, si, co);
    puts(b);
   }
 printf("\nKONEC!\a\n");
 return 0;
}
```

Řešení

Zadání: 1) Napište program, který zobrazí ASCII kódy znaků **A** až **Z** a **a** až **z**. Jak se od sebe liší kódy velkých a malých písmen?

Řešení:

```
#include <stdio.h>

int main(void)
{
 char ch;

 for(ch='A'; ch<='Z'; ch++)
    printf("%d ", ch);

 printf("\n");

 for(ch='a'; ch<='z'; ch++)
    printf("%d ", ch);

 return 0;
}
```

Zpět

Zadání: 2) Je tento program správný? Pokud ne, proč?

```
#include <stdio.h>

int main(void)
{
 char p, *q;

 printf("Zadejte retezec: ");
 p = gets(q);
 printf(p);

 return 0;
}
```

Řešení:

Ne. Program je chybný, protože **p** musí být pointer.

Zpět

Zadání: 3) Napište program, který vypisuje tabulku čísel. Každý řádek obsahuje číslo a jeho druhou a třetí mocninu. Tabulka začíná číslem 2 a končí číslem 100.

Řešení:

```
#include <stdio.h>

int main(void)
{
 unsigned long i;

 for(i=2; i<=100; i++)
    printf("%-10lu %-10lu %-10lu\n", i, i*i, i*i*i);

 return 0;
}
```

Zpět

Zadání: 4) Jak byste vypsali tento řádek pomocí funkce **printf**()?

```
Sleva: 40% puvodni ceny
```

Řešení:

```
printf("Sleva: 40%% puvodni ceny");
```

Zpět

Zadání: 5) Ukažte jak zobrazit číslo 1023.231 tak aby se vypisovala jen dvě desetinná čísla.

Řešení:

```
printf("%.2f", 1023.231);
```

Zpět

Zadání: 6) Napište program, který načte desetinné číslo a pak zvláš• zobrazí jeho celou a desetinnou část.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int i,j;

 printf("Zadejte desitinne cislo: ");
 scanf("%d.%d",&i,&j);
 printf("Cela cast: %d, desetinna cast: %d",i,j);

 return 0;
}
```

Zpět

Zadání: 7) Je tento úsek správný? Pokud ne, proč?

```
char ch;
scanf("%2c", &ch);
```

Řešení:

Ne. Znak může mít maximální výstupní délku 1.

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>

int main(void)
{
 char ch;

 for(ch='A'; ch<='Z'; ch++)
         printf("%d ", ch);

 printf("\n");

 for(ch='a'; ch<='z'; ch++)
         printf("%d ", ch);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 unsigned long i;

 for(i=2; i<=100; i++)
    printf("%-10lu %-10lu %-10lu\n", i, i*i, i*i*i);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i,j;

 printf("Zadejte desitinne cislo: ");
 scanf("%d.%d",&i,&j);
 printf("Cela cast: %d, desetinna cast: %d",i,j);

 return 0;
}
```

```c
#include <stdio.h>

/* Prototypy funkci */
void funkce1(void);
void funkce2(void);

int main(void)
{
 funkce2();
 printf("3");

 return 0;
}

void funkce2(void)
{
 funkce1();
 printf("2");
}

void funkce1(void)
{
 printf("1");
}
```

```c
#include <stdio.h>

/* Prototyp funkce */
int mocnina(void);

int main(void)
{
 int mocnina_cisla;

 mocnina_cisla = mocnina();
 printf("Druha mocnina: %d",mocnina_cisla);

 return 0;
}

int mocnina(void)
{
 int cislo;

 printf("Zadejte cislo: ");
 scanf("%d",&cislo);

 return cislo*cislo;
}
```

```c
#include <stdio.h>

int nacti(int *a, int *b)
{
 printf("\nZadej dve cela cisla:");
 return scanf("%d %d", a, b);
}

float dej_podil(int i, int j)
{
 return((float) i / (float) j);
}

int main(void)
{
 int c1, c2;
 if (nacti(&c1, &c2) == 2)
   printf("Podil je : %f\n", dej_podil(c1, c2));
 return 0;
}
```

```c
#include <stdio.h>

void swap(int *i, int *j);

int main(void)
{
 int num1=50, num2=20;

 printf("Cislo1: %d, Cislo2: %d\n",num1, num2);
 swap(&num1, &num2);
 printf("Cislo1: %d, Cislo2: %d\n",num1, num2);

 return 0;
}

void swap(int *i, int *j)
{
 int temp;

 temp=*i;
 *i=*j;
 *j=temp;
}
```

```c
#include <stdio.h>

int fakt(int n)
{
 return (( n <= 0 ) ? 1 : n * fakt(n-1));
}

int main()
{
 int i;

 printf("\nZadej cele cislo: ");
 scanf("%d",&i);
 printf("Faktorial cisla %d je: %d",i,fakt(i));
 return 0;
}
```

```c
#include <stdio.h>

void rekurze(int i);

int main(void)
{
 rekurze(0);

 return 0;
}

void rekurze(int i)
{
 if(i<10)
 {
  rekurze(i+1);
  printf("%d ",i);
 }
}
```

Řešení

Zadání: 1) Napište program, který bude mít alespoň dvě funkce a vytiskne **Co se v mládí naučíš, ve stáří zapomeneš**.

Řešení:

```
#include <stdio.h>

void cast1(void);
void cast2(void);

int main(void)
{
 cast1();
 cast2();

 return 0;
}

void cast1(void)
{
 printf("Co se v mladi naucis, ");
}

void cast2(void)
{
 printf("ve stari zapomenes.");
}
```

Zpět

Zadání: 2) Napište program, který použije funkci **convert()**, která vyzve uživatele k zadání částky v korunách a převede ji na dolary (Použijte směnný kurs např. 40 Kč za dolar). Vypište převedenou částku v dolarech.

Řešení:

```
#include <stdio.h>

int convert();

int main(void)
{
 printf("Dolary: %d", convert());

 return 0;
}

int convert()
{
 int koruny;

 printf("Zadejte castku v korunach: ");
 scanf("%d", &koruny);
 return koruny/40;
}
```

Zpět

Zadání: 3) Co je v tomto programu chybné?

```
#include <stdio.h>

int f1(void);

int main(void)
{
 double odpoved;

 odpoved = f1();
 printf("%f",odpoved);

 return 0;
}

void f1(void)
{
 return 100;
}
```

Řešení:

Program je teoreticky v pořádku. Funkce **f1()** však vrací celočíselnou hodnotu, ale ta se přiřazuje proměnné typu **double**.

Zpět

Zadání: 4) Co je chybné v této funkci?

```
void func(void)
{
 int i;
 printf("Zadejte cislo: ");
 scanf("%d", &i);

 return i;
}
```

Řešení:

Funkce je deklarována s návratovým typem **void** a nemůže tudíž vracet žádnou hodnotu.

Zpět

Zadání: 5) Je tento program správný? Pokud ne, proč?

```
#include <stdio.h>

myfunc(int num, int min, int max);

int main(void)
{
 int i;

 printf("Zadejte cislo mezi 1 a 10: ");
 myfunc(&i, 1, 10);

 return 0;
}

void myfunc(int num, int min, int max)
{
 do
 {
   scanf("%d", num);
 }
 while(*num<min || *num>max);
}
```

Řešení:

Ne, funkce **myfunc()** se volá s ukazatelem na první argument místo se samotným argumentem.

Zpět

Zadání: 6) Vysvětlete rozdíl mezi voláním funkce hodnotou a voláním funkce odkazem.

Řešení:

Při volání hodnotou se funkci předává hodnota argumentu. Při volání odkazem se funkci předává adresa argumentu.

Zpět

Zadání: 7) Co je špatně v této rekurzivní funkci?

```
void f(void)
{
 int i;

 printf("in f() \n");

 /* volani f() 10krat */
 for(i=0; i<10; i++) f();
}
```

Řešení:

Funkce volá sama sebe, dokud program nezhavaruje, protože zde není žádná podmínka, která zabraňuje neustálému opakování rekurze.

Zpět

Zadání: 8) Napište program, který vypisuje na obrazovku řetězec znak po znaku pomocí rekurzivní funkce.

Řešení:

```
#include <stdio.h>

void display(char *p);

int main(void)
{
 display("Zobraz tento text");

 return 0;
}

void display(char *p)
{
 if (*p)
 {
  printf("%c", *p);
  display(p+1);
```

```
 }
}
```

[Zpět]

Poslední změna: **26. 2. 2002**

[RNDr. Petr Šaloun Ph.D.], [Radek Šup], [VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky]

```
#include <stdio.h>

void cast1(void);
void cast2(void);

int main(void)
{
        cast1();
        cast2();

        return 0;
}

void cast1(void)
{
 printf("Co se v mladi naucis, ");
}

void cast2(void)
{
 printf("ve stari zapomenes.");
}
```

```c
#include <stdio.h>

int convert();

int main(void)
{
 printf("Dolary: %d", convert());

 return 0;
}

int convert()
{
 int koruny;

 printf("Zadejte castku v korunach: ");
 scanf("%d", &koruny);
 return koruny/40;
}
```

```c
#include <stdio.h>

void display(char *p);

int main(void)
{
 display("Zobraz tento text");

 return 0;
}

void display(char *p)
{
 if (*p)
 {
  printf("%c", *p);
  display(p+1);
 }
}
```

```c
#include <stdio.h>

int main(void)
{
 int c;


 printf("Zadej znak: ");
 c = getchar();  /* nacteni znaku */
 if (c >='A' && c<='Z')
        printf("%d\n",c);   /* vytisteni ordinalni hodnoty */
 else
        printf("Male pismeno nebo jiny znak\n");
 return 0;
}
```

```c
#include <stdio.h>

void main(void)
{
  int a, b;

  printf("Zadejte dve cisla: ");
  scanf("%d%d", &a, &b);

  if(b)
    printf("%d\n", a/b);
  else
    printf("Nelze delit nulou!\n");
}
```

```c
#include <stdio.h>

int main(void)
{
 int i;

 printf("Stiskni klavesu: ");
 scanf("%d",&i);
 switch (i)
 {
 case 1:
          printf ("Stiskl jsi klavesu 1\n");
          break;
 case 2:
          printf ("Stiskl jsi klavesu 2\n");
          break;
 case 3:
          printf ("Stiskl jsi klavesu 3\n");
          break;
 default:
          printf ("Jina klavesa\n");
          break;
 }

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i;

 printf("Stiskni klavesu: ");
 scanf("%d",&i);
 switch (i)
 {
 case 1:
          printf ("Stiskl jsi klavesu 1\n");
 case 2:
          printf ("Stiskl jsi klavesu 2\n");
          break;
 case 3:
          printf ("Stiskl jsi klavesu 3\n");
          break;
 default:
          printf ("Jina klavesa\n");
          break;
 }

 return 0;
}
```

```c
#include <stdio.h>

int main()
{
 int c;

 printf("Zadavejte znaky: ");
 while ((c=getchar()) != 'z')
 {
  if (c >= ' ') /* tiskneme pouze viditelne znaky */
        putchar(c);
 }
 printf("Nacitani znaku bylo ukonceno\n");
 return 0;
}
```

```c
#include <stdio.h>

int main()
{
 int c;

 printf("Zadavejte znaky: ");
 while (1)               /* nekonecna smycka */
 {
  if ((c=getchar()) <= ' ')
        continue;       /* zahazujeme neviditelne znaky */
  if (c == 'z')
        break;          /* zastaveni po nacteni znaku z */
  putchar(c);           /* tisk znaku */
 }
 printf("Nacitani znaku bylo ukonceno\n");
 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 int i, cislo;

 for(i=1; i<=10; i++)
         printf("%d ",i);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i, soucet, soucin;
 soucet=0;
 soucin=1;

 for(i=1; i<6;i++)
 {
  soucet = soucet + i;
  soucin = soucin * i;
 }
 printf("Soucet cisel 1 az 5 je: %d",soucet);
 printf("\nSoucin cisel 1 az 5 je: %d",soucin);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int c;

 do
 {
  if ((c=getchar()) >= ' ')   /* nacitani znaku */
        putchar(c);   /* tisk znaku */
 } while (c != 'z');

 return 0;
}
```

Řešení

Zadání: 1) Je tento úsek programu správný?

```
if(pocet < 100)
   printf("Cislo je mensi nez 100.\n");
   printf("Jeho druha mocnina je: %d", pocet * pocet);
}
```

Řešení:

Ne, chybí otevírací složená závorka.

Zpět

Zadání: 2) Napište program, který uživatele požádá o zadání celého čísla a pak vypíše zda je číslo liché nebo sudé. (Rada: použijte operátor modulo %).

Řešení:

```
#include <stdio.h>

int main(void)
{
 int i;

 printf("Zadejte cislo: ");
 scanf("%d",&i);
 if((i%2)==0) printf("Sude cislo");
 if((i%2}==1) printf("Liche cislo");

 return 0;
}
```

Zpět

Zadání: 3) Napište program, který si vyžádá dvě celá čísla a pak zobrazí podle volby uživatele buď součin nebo součet.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int a,b,c;

 printf("Zadejte prvni cislo: ");
 scanf("%d",&a);
 printf("Zadejte druhe cislo: ");
 scanf("%d",&b);
 printf("Zadejte 0 pro secteni, 1 pro vynasobeni: ");
 scanf("%d",&c);
 switch(c)
 {
        case 0: printf("Soucet cisel %d a %d je %d",a,b,a+b);
                                  break;

        case 1: printf("Soucin cisel %d a %d je %d",a,b,a*b);
                                  break;

        default: printf("Stisknuta jina klavesa nez 0 nebo 1");
 }
 return 0;
}
```

Zpět

Zadání: 4) Co je špatně na této části programu?

```
float f;

scanf("%f",&f);

switch(f):
{
 case 10.05:
 .
 .
 .
}
```

Řešení:

Pro řízení příkazu switch **nelze** použít hodnoty s pohyblivou řádovou čárkou.

Zpět

Zadání: 5) Napište program, který načte zadaný znak a rozhodne zda je souhláska či samohláska.

Řešení:

```
#include <stdio.h>

int main(void)
{
 char c;

 printf("Zadejte znak: ");
 scanf("%c",&c);

 switch(c)
 {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'y': printf("Samohlaska");
                          break;
        default: printf("Souhlaska");
 }
 return 0;
}
```

Zpět

Zadání: 6) Napište program, který vytiskne čísla mezi 17 a 100, která budou beze zbytku dělitelná 17.

Řešení:

```
#include <stdio.h>
{
 int i;

 for(i=17;i<101; i++)
 {
        if((i%17)==0) printf("\n%d",i);
 }
 return 0;
}
```

Zpět

Zadání: 7) Napište program, který převádí galony na litry. Použijte cyklus do, aby mohl uživatel převody opakovat. (Jeden galon je asi 3,7854 litrů).

Řešení:

```c
#include <stdio.h>

int main(void)
{
 float galony;

 printf("Zadejte galony: ");
 scanf("%f",&galony);

 do
 {
        printf("Litry: %f\n", galony * 3.7854);
        printf("Zadejte galony nebo 0 pro ukonceni: ");
        scanf("%f",&galony);
 }
 while(galony != 0);

 return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>

int main(void)
{
 int i;

 printf("Zadejte cislo: ");
 scanf("%d",&i);
 if((i%2)==0) printf("Sude cislo");
 if((i%2)==1) printf("Liche cislo");

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int a,b,c;

 printf("Zadejte prvni cislo: ");
 scanf("%d",&a);
 printf("Zadejte druhe cislo: ");
 scanf("%d",&b);
 printf("Zadejte 0 pro secteni, 1 pro vynasobeni: ");
 scanf("%d",&c);
 switch(c)
 {
        case 0: printf("Soucet cisel %d a %d je %d",a,b,a+b);
                                break;

        case 1: printf("Soucin cisel %d a %d je %d",a,b,a*b);
                                break;

        default: printf("Stisknuta jina klavesa nez 0 nebo 1");
 }

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 char c;

 printf("Zadejte znak: ");
 scanf("%c",&c);

 switch(c)
 {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'y': printf("Samohlaska");
                                break;
        default: printf("Souhlaska");
 }
 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int a = 7, b = 2;

 printf("cislo a = %d, cislo b = %d\n", a, b);
 printf("soucet %d + %d = %d\n", a, b, a+b);
 printf("rozdil %d - %d = %d\n", a, b, a-b);
 printf("soucin %d * %d = %d\n", a, b, a*b);
 printf("podil %d / %d = %d\n", a, b, a/b);
 printf("zbytek po celociselnem deleni %d %% %d = %d", a, b, a%b);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i, j;
 float r, x;

 i = j = 5;

 j *= i;
 r = j / 3;
 x = j * 3;

 printf("i=%d\tj=%d\tr=%f\tx=%f\n", i, j, r, x);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int p,q;

 printf("Zadejte P (0 nebo 1): ");
 scanf("%d",&p);
 printf("Zadejte Q (0 nebo 1): ");
 scanf("%d",&q);
 printf("P AND Q: %d\n", p && q);
 printf("P OR Q: %d\n", p ¦¦ q);
 printf("NOT Q: %d\n", !q);

return 0;
}
```

```c
#include <stdio.h>

int main()
{
 printf("1 << 1 = \t%d\t%#x\n", 1 << 1, 1 << 1);
 printf("1 << 7 = \t%d\t%#x\n", 1 << 7, 1 << 7);

 printf("-1 >> 1 = \t%d\t%#x\n", -1 >> 1, -1 >> 1);
 printf("512 >> 8 = \t%d\t%#x\n", 512 >> 8, 512 >> 8);

 printf("2 & 1 = \t%d\t%#x\n", 2 & 1, 2 & 1);
 printf("2 | 1 = \t%d\t%#x\n", 2 | 1, 2 | 1);
 printf("2 ^ 1 = \t%d\t%#x\n", 2 ^ 1, 2 ^ 1);

 return 0;
}
```

```c
#include <stdio.h>

int main (void)
{
 int a = 10, b = 20;

 printf("Pocatecni hodnota a: %d\n",a);

 a = a ^ b;
 printf("Po prvnim XOR: %d\n",a);

 a = a ^ b;
 printf("Po druhem XOR: %d\n",a);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i;

 printf("Zadejte cislo: ");
 scanf("%d",&i);
 i = i>=0 ? 1: -1;
 printf("Vysledek: %d",i);

 return 0;
}
```

Řešení

Zadání: 1) Napište program, který načte dvě desetinná čísla (použijte typ **float**) a pak vytiskněte jejich součet.

Řešení:

```
#include <stdio.h>

int main(void)
{
 float i,j;

 printf("Zadejte dve cisla: ");
 scanf("%f",&i);
 scanf("%f",&j);
 printf("Jejich soucet je: %f",i+j);

 return 0;
}
```

Zpět

Zadání: 2) Napište program, který vypočte objem kvádru. Program se bude ptát uživatele na jednotlivé rozměry a pak vypíše objem.

Řešení:

```
#include <stdio.h>

int main(void)
{
 float delka, sirka, vyska;

 printf("Zadej delku: ");
 scanf("%f",&delka);
 printf("Zadej sirku: ");
 scanf("%f",&sirka);
 printf("Zadej vysku: ");
 scanf("%f",&vyska);
 printf("Objem je %f", delka*sirka*vyska);

 return 0;
}
```

Zpět

Zadání: 3) Napište program, který vypočítá počet sekund v nepřestupném roce.

Řešení:

```
#include <stdio.h>

int main(void)
{
 printf("Pocet sekund v roce: ");
 printf("%f", 60.0 * 60.0 * 24.0 * 365.0);

 return 0;
}
```

Zpět

Zadání: 4) Je tento výraz pravdivý?

!(10==9)

Řešení:

Ano

Zpět

Zadání: 5) Dávají tyto dva výrazy stejný výsledek?

a. 0&&1||1
b. 0&&(1||1)

Řešení:

Ne, první je pravdivý, druhý je nepravdivý.

Zpět

Zadání: 6) Jelikož hodnota v pohyblivé řádové čárce nemůže být použita spolu s operátorem %, jak můžete upravit tento příkaz?

```
x = 123.123 % 3 ;
```

Řešení:

```
x = (int) 123.123 % 3 ;
```

Zpět

Zadání: 7) Jedním zvláště dobrým příkladem použití operátoru **?** je ochrana před dělením nulou. Napište program, který přečte dvě celá čísla zadaná uživatelem a vypíše výsledek dělení prvního čísla druhým. Použijte **?** abyste zabránili dělením nulou.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int i, j, vysledek;

 printf("Zadejte dve cela cisla: ");
 scanf("%d %d", &i, &j);

 vysledek = j ? i/j : 0;   /* pravda je jakakoliv nenulova hodnota */
 printf("%d", vysledek);

 return 0;
}
```

Zpět

Zadání: 8) Převeďte následující příkaz na jeho ekvivalentní příkaz pomocí operátoru **?**.

```
if(a>b) pocet = 5;
else pocet = 10;
```

Řešení:

```
pocet = a>b ? 5 : 10:
```

Zpět

Zadání: 9) Jaká je hodnota **i** po provedení následujícího kroku?

```
i = (1, 2, 3);
```

Řešení:

i = 3

Zpět

Zadání: 10) Jaký je výsledek těchto operací?

a. 10100011 & 01011101
b. 01011101 | 11111011
c. 01010110 ^ 10101011

Řešení:

a. 00000001
b. 11111111
c. 11111101

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>

int main(void)
{
 float i,j;

 printf("Zadejte dve cisla: ");
 scanf("%f",&i);
 scanf("%f",&j);
 printf("Jejich soucet je: %f",i+j);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 float delka, sirka, vyska;

 printf("Zadej delku: ");
 scanf("%f",&delka);
 printf("Zadej sirku: ");
 scanf("%f",&sirka);
 printf("Zadej vysku: ");
 scanf("%f",&vyska);
 printf("Objem je %f ", delka*sirka*vyska);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 printf("Pocet sekund v roce: ");
 printf("%f", 60.0 * 60.0 * 24.0 * 365.0);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 int i, j, vysledek;

 printf("Zadejte dve cela cisla: ");
 scanf("%d %d", &i, &j);

 vysledek = j ? i/j : 0;   /*pravda je jakakoliv nenulova hodnota */
 printf("%d", vysledek);

 return 0;
}
```

2. Jazyk C++

Časová náročnost kapitoly: 1 hodina 10 minut

V této části si vysvětlíme základní pojmy jako je objektově orientované programování, zapouzdření, polymorfismus a dědičnost. Také si zkusíme napsat první jednoduché programy. Zmíníme se o komentářích a klíčových slovech. Závěrem si řekneme něco o třídách a přetěžování funkcí.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je vhodné mít znalosti o jazyku C - tedy zvládat znalosti potřebné pro úspěšné programování v jazyce C.

2.1 Objektově orientované programování (OOP)

Časová náročnost: 5 minut

Objektově orientované programování je výkonný způsob, jak přistupovat k úloze programování. Již od svých ranných začátků bylo programování spojováno s rozličnými metodologiemi. První programy byly realizovány pouhým nastavením přepínačů na čelním panelu počítače. Tento postup však byl vhodný pouze pro velice malé programy.

Později vytvořený **jazyk symbolických instrukcí** již umožňoval psaní delších programů. K dalšímu vývoji došlo v roce 1955, kdy byl vytvořen **programovací jazyk vysoké úrovně** - FORTRAN. S využitím programovacích jazyků vysoké úrovně byl programátor schopen psát dlouhé a složité programy. Další rozvoj přinesl vývoj **strukturovaných programovacích jazyků** - ALGOL a PASCAL. Strukturované programování se opírá o dobře definované řídící struktury, bloky kódu, vyloučení příkazu GOTO (nebo jeho minimalizace), funkce, rekurzi a lokální proměnné. Podstatou strukturovaného programování je začlenění programu do jeho základních vymezovacích prvků. S využitím strukturovaného programování může průměrný programátor vytvořit a udržovat programy až o délce několika tisíc řádků.

Když však program přesáhl určitou velikost, zklamalo i strukturované programování. Bylo potřeba vytvořit nový přístup k programování, který by dovolil psát složitější programy. K tomuto účelu bylo vytvořeno **objektově orientované programování**. OOP vzalo nejlepší myšlenky včleněné do strukturovaného programování a zkombinovalo je s výkonnými

novými koncepty, které umožňovaly organizovat programy mnohem efektivněji. Objektově orientované programování podněcuje k rozkladu problému na **elementární prvky**. Každá **komponenta** se stává samostatným a **nezávislým objektem**, který obsahuje své **vlastní instrukce a data**, vztahující se k tomuto objektu.

Všechny objektově orientované jazyky sdílejí následující tři vlastnosti:

1. **Zapouzdření** (*encapsulation*)
2. **Polymorfismus** - schopnost výskytu v mnoha formách (*polymorphism*)
3. **Dědičnost** (*inheritance*)

2.2 Zapouzdření

Časová náročnost: 3 minuty

Zapouzdření je mechanismus, který svazuje dohromady kód a data a zabezpečuje je před vnějšími zásahy či zneužitím. V OOP může být kód s daty slučován takovým způsobem, že vznikají tzv. nezávislé "černé skříňky". **Spojením kódu s daty vzniká objekt**. Jinými slovy lze říci, že objekt je instrument, který podporuje zapouzdření.

Uvnitř objektu může být kód nebo data nebo obojí, jednak jako **privátní** (*private*) vzhledem k objektu, nebo jako **veřejná** (*public*).

Privátní kód nebo data jsou známá a dostupná pouze pro jinou část daného objektu. Znamená to, že privátní kód nebo data **nejsou dostupné** z jiné části programu mimo objekt.

Když jsou kód nebo data **veřejná**, mohou k nim přistupovat i jiné části programu. Typicky jsou veřejné prvky objektu využity k zajištění řízeného rozhraní k privátním elementům objektu.

2.3 Polymorfismus

Časová náročnost: 5 minut

Polymorfismus je vlastnost, která umožňuje, aby bylo jedno jediné jméno použito pro dva nebo více souvisejících, ale technicky různých účelů. Ve vztahu k OOP umožňuje polymorfismus určit jedním jménem celou obecnou třídu procesů. Uvnitř obecné třídy procesů je pak volba konkrétního procesu dána typem dat.

Obecně lze polymorfismus charakterizovat jako: "jedno rozhraní, mnoho metod". Výhodou polymorfismu je, že omezuje přílišnou složitost tím, že určením **obecné třídy procedury** povolí jediné rozhraní. Je pak záležitostí překladače, aby vybral **konkrétní proceduru**, vhodnou pro danou situaci.

Příklad:

V jazyce C byly definovány funkce abs(), labs() a fabs() které vracely absolutní hodnotu z čísel integer, long integer a float. V C++ který podporuje polymorfismus mohou být všechny tyto funkce volány pod jediným jménem abs(). Tato vlastnost polymorfismu se nazývá **vícenásobná definice funkce**.

Polymorfismus může být použit také na operátory.

Příklad:

V jazyce C je znaménko + užito ke sčítání integer, long integer, znaku nebo hodnoty v pohyblivé řádové čárce. Ve všech případech překladač pozná, který typ aritmetiky má použit. V C++ je možné tento koncept podle vlastního uvážení rozšířit na další typy dat. Tato vlastnost polymorfismu se nazývá **vícenásobná definice operátoru**.

Polymorfismus tedy dovoluje vytvářet standardní rozhraní k příslušným procesům.

2.4 Dědičnost

Časová náročnost: 3 minuty

Dědičnost je proces, při němž může jeden objekt získat vlastnosti jiného objektu. Přesněji řečeno objekt může zdědit obecnou sadu vlastností a do ní může přidat takové vlastnosti, které jsou specifické pouze pro něj. Dědičnost je důležitá, protože dovoluje objektu podporovat koncept **hierarchické klasifikace**.

Příklad:

Popis **domu**. Dům je částí obecné třídy nazvané **budova**. Budova je zase částí obecnější třídy nazvané **stavba**, která je opět součástí obecné třídy objektů nazvané **zhotovené člověkem**.

Třída potomka dědí veškeré vlastnosti spojené s **rodiči** a přidává si k nim své vlastní charakteristiky. Bez využití uspořádané klasifikace, by měl každý objekt definovány všechny charakteristiky, které se k němu vztahují. Prostřednictvím dědičnosti je možné zadat třídu (či třídy) do níž daný objekt patří a přiřadit mu specifické vlastnosti, které mají všechny objekty dané třídy.

2.5 První program

Časová náročnost: 12 minut

Protože je C++ rozšířenou množinou jazyka C, jsou všechny prvky jazyka C obsaženy také v C++. Snad nejběžnějším prvkem specifickým pro C++ je jeho přístup ke konzoli I/O. Můžeme stále používat funkce **printf**() a **scanf**(), ale C++ nám nabízí nový a lepší způsob jak provádět tyto operace a to pomocí **I/O operátorů**. Výstupní operátor je **<<** a vstupní **>>**. Pro výstup na obrazovku proto použijeme příkaz:

Příklad:

```
cout << "Prikaz jazyka C++";
```

který způsobí, že se na obrazovku počítače vypíše řetězec. **Cout** je předefinovaný datový proud (stream), který je při spuštění programu v C++ automaticky připojen ke konzoli. Je to podobné jako **stdout** v jazyce C. Použitím výstupního operátoru << lze provést výstup jakéhokoliv základního typu jazyka C++.

Příklad:

```
cout << 25.125;
```

Obecný formát:

```
cout << výraz;
```

Pro vstup hodnot z klávesnice poté používáme vstupní operátor >>. Následující příkaz například načte hodnotu typu integer do **cislo**.

Příklad:

```
int cislo;
cin >> cislo;
```

Obecný formát:

```
cin >> výraz;
```

Následující program požádá uživatele o zadání celého čísla. Poté vytiskne zadané číslo a jeho dvojnásobek a druhou mocninu.

Příklad 2.1:

```
#include <iostream.h>

int main(void)
{
 int i;

 cout << "Zadejte celociselnou hodnotu: ";
 cin >> i;
 cout << "Zadana hodnota byla " << i << "\n";
 cout << "Jeji dvojnasobek je " << i*2 << " a druha mocnina je " << i*i;

 return 0;
}
```

Následující příklad ilustruje, že v jednom příkazu může vystupovat libovolný počet položek různého typu.

Příklad 2.2:

```
#include <iostream.h>

int main(void)
{
 int i;
 char str[80];

 cout << "Zadejte celociselnou hodnotu a retezec: ";
 cin >> i >> str;
 cout << "Zadana data: " << i << " " << str;

 return 0;
}
```

2.6 Komentáře

Časová náročnost: 3 minuty

V jazyce C++ můžeme vkládat poznámky do textu stejným způsobem jako v jazyce C. Tedy buď pomocí párových značek /* */ :

Příklad:

```
/*
komentar
*/
```

Nebo pomocí // za nimiž je vše až do konce řádku považováno za komentář.

Příklad:

```
int a, b, c;  // celociselne promenne
```

Pozn: Jazyk Java zavádí dokumentační komentář, který je podkladem pro generovaní dokumentace.

Pozn: Omezení platné v jazyce C si můžete prohlédnout v části Komentáře jazyka C

2.7 Klíčová slova

Časová náročnost: 3 minuty

Při použití **klíčových slov** platí stejná pravidla jako v jazyce C. Proto je tedy **nesmíme** používat v jiném významu, než jak určuje norma ISO, což znamená, že nesmí být použita jako jména proměnných nebo funkcí.

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

2.8 Úvod do tříd

Časová náročnost: 20 minut

Snad nejdůležitějším prvkem jazyka C++ je **třída**. Je to mechanismus používaný k vytváření objektů. Jako taková je třída srdcem mnoha prvků jazyka C++.

Třída je deklarována klíčovým slovem **class**. Syntaxe deklarace class vypadá v obecném tvaru takto:

```
class jméno-tøídy
{
    // privátní funkce a promìnné
 public:
    // veøejné funkce a promìnné
} seznam-objektù
```

Seznam objektů je v deklaraci třídy nepovinný. Stejně jako strukturu, můžeme deklarovat objekty třídy později - až budou potřeba. Zatímco **jméno třídy** je také technicky nepovinné, z praktického hlediska je vždy potřeba. Je to proto, že jméno třídy se stává novým typem jména použitého k deklaraci objektů třídy.

Funkce a proměnné deklarovány uvnitř deklarace třídy jsou označovány jako **členy** této třídy. To znamená, že jsou přístupné pouze pro ostatní členy třídy. Pro deklaraci členů veřejné třídy se použije klíčové slovo **public:**. Všechny funkce a proměnné takto deklarované jsou přístupné pro členy třídy a i pro další části programu, který obsahuje třídu.

Příklad jednoduché deklarace třídy.

Příklad:

```
class mojetrida
{
    int a; // privatni promenna
 public:
    void nastav_a(int num);
        int cti_a();
};
```

Tato třída má pouze jednu privátní proměnnou **a** a dvě veřejné funkce **nastav_a(int num)** a **cti_a()**. Funkce které jsou deklarovány uvnitř třídy se nazývají **členské funkce**.

Jelikož **a** je privátní není dostupná pro žádný kód vně **mojetrida**. Ovšem veřejné funkce mohou být volány každou částí programu, která **mojetrida** obsahuje.

Ačkoliv jsou funkce **nastav_a(int num)** a **cti_a()** deklarovány, nejsou ještě definovány. Abychom definovaly členskou funkci musíme spojit typové jméno třídy se jménem funkce a to pomocí **dvojice dvojteček ::**. Dvojice dvojteček se nazývá **operátor rozlišení oblasti**.

Ukažme si jak mohou být funkce definovány:

Příklad:

```
void mojetrida::nastav_a(int num)
{
 a = num;
}

int mojetrida::cti_a()
{
 return a;
}
```

Obecný tvar pro definici členské funkce:

```
return-type jméno-tøídy::jméno-funkce(seznam-argumentù)
{
 // tìlo funkce
}
```

Deklarace **mojetrida** nedefinueje žádný objekt typu **mojetrida**. Definuje pouze typ objektu, když bude deklarován.

Následující řádek deklaruje dva objekty typu mojetrida.

Příklad:

```
mojetrida obj1, obj2;
```

Poté co vytvoříme objekt třídy může se program odkazovat na jeho veřejné členy pomocí **tečkových operátorů**.

Příklad:

```
obj1.nastav_a(10);
obj2.nastav_a(5);
```

Tyto příkazy nastaví kopii **obj1** na 10, kopii **obj2** na 5. Každý objekt obsahuje **vlastní** kopii všech dat deklarovaných uvnitř třídy. Tedy **obj1** je odlišné od **a** vázaného na **obj2**.

Deklarace třídy je pouze logická abstrakce, jež definuje nový typ. Určuje, jak bude objekt daného typu vypadat. Teprve deklarace objektu vytváří fyzickou entitu daného typu. Objekt totiž zabírá pamě•, ale definiční typ ne.

Každý objekt třídy má svou vlastní kopii každé z proměnných deklarovaných uvnitř třídy.

Následující příklad předvádí třídu **mojetrida** popsanou výše.

Příklad 2.3:

```
#include <iostream.h>

class mojetrida
{
  int a;
public:
  void nastav_a(int num);
  int cti_a();
};

void mojetrida::nastav_a(int num)
{
  a = num;
}

int mojetrida::cti_a()
{
  return a;
}

int main()
{
  mojetrida obj1, obj2;

  obj1.nastav_a(10);
  obj2.nastav_a(5);

  cout << obj1.cti_a() << "\n";
  cout << obj2.cti_a() << "\n";

  return 0;
}
```

2.9 Úvod do přetěžování

Časová náročnost: 10 minut

Po třídách je jednou z nejdůležitějších částí C++ **přetěžování funkcí**. Přetěžování funkcí nejen poskytuje mechanismus, jímž C++ poskytuje jeden typ polymorfismu, ale také utváří základ, na němž může být programovací prostředí dynamicky rozšiřováno.

V C++ mohou dvě nebo více funkcí sdílet stejné jméno tak dlouho, pokud se liší typy jejich argumentů, nebo se liší počet jejich argumentů, nebo obojí. Když dvě nebo více funkcí sdílí stejné jméno označují se jako **přetížené**.

Je velmi snadné přetížit funkci. Prostě deklarujeme a definujeme všechny požadované verze. Správnou verzi vybere překladač podle typu nebo počtu argumentů, použitých pro volání funkce.

V C++ však lze přetěžovat i operátory.

Nyní si přetěžování funkcí ukážeme na příkladu. V jazyce C máme funkce **abs**(), **labs**(), **fabs**(), které vracejí absolutní hodnotu z hodnot integer, long integer a float. Naším úkolem bude napsat přetíženou funkci **abs**(), která vrátí absolutní hodnotu zadaného čísla.

Příklad 2.4:

```
#include <iostream.h>

int abs(int n);
long abs(long n);
double abs(double n);

int main()
{
  cout << "Absolutni hodnota -10: " << abs(-10) << "\n";
  cout << "Absolutni hodnota -10L: " << abs(-10L) << "\n";
  cout << "Absolutni hodnota -10.01: " << abs(-10.01) << "\n";

  return 0;
}

// abs() pro int
int abs(int n)
{
  cout << "Abs() pro Integer\n";
  return n<0 ? -n : n;
}

// abs() pro long
long abs(long n)
{
  cout << "Abs() pro long\n";
  return n<0 ? -n : n;
}

// abs() pro doubles
double abs(double n)
{
  cout << "Abs() pro double\n";
  return n<0 ? -n : n;
}
```

Cvièení

1) Napište program, který převádí metry na centimetry. Program bude opakovat tento proces, dokud uživatel nezadá nulu. Řešení

2) Převeďte tento program napsaný v jazyce C tak, aby používat I/O styly z C++.

```c
#include <stdio.h>

int main(void)
{
  int a, b, d, min;

  printf("Zadej dve cisla: ");
  scanf("%d%d", &a, &b);
  min = a > b ? b : a;
  for(d = 2; d<=min; d++)
    if(((a%d)==0) && ((b%d)==0)) break;
  if((d-1)==min) {
    printf("Zadny spolecny delitel\n");
    return 0;
  }
  printf("Nejmensi spolecny delitel je %d\n", d);
  return 0;
}
```

Řešení

3) Vytvořte funkci **min**(), která vrací menší ze dvou zadaných numerických argumentů, použitých ve volání funce. Přeložte **min**() tak, aby jako své argumenty přijímala znaky, integer a double. Řešení

4) Mějme program napsaný podle konvencí C++. Předveďte jak je změnit do formy pro jazyk C.

```c
#include <iostream.h>

int f(int a);

int main()
{
  cout << f(10);

  return 0;
}

int f(int a)
{
  return a * 3.1416;
}
```

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

1. Úvod

Časová náročnost: 5 minut

C++ je reakcí programátorů jazyka C na **objektově orientované programování**. Jazyk C++ je vytvořen na pevných základech jazyka C. Přináší nám podporu objektově orientovaného programování (a mnoha dalších nových prvků). Přitom však nebyla obětována ani původní výkonnost jazyka C, ani jeho elegance a pružnost.

C++ byl vytvořen **Bjarne Stroustrupem** v roce 1979 v Bellových laboratořích v Murray Hill v New Jersey. Původně se jmenoval *C s třídami*. Jeho jméno bylo teprve v roce 1983 změněno na *C++*. Od té doby prošel jazyk C++ třemi zásadními revizemi. První z nich byla v roce 1985 , druhá v roce 1990 a třetí proběhla během standardizačního procesu. Práce na standardizaci C++ započala před několika lety. Hrubý koncept plánovaného standardu byl vytvořen 25. ledna 1994. V tomto konceptu se komise ANSI / ISO C++ (American National Standards Institute a International Standards Organization) držela zásad definovaných Stroustrupem a přidala některá další. V podstatě však zůstal původní návrh C++ zachován.

Brzy po dokončení hrubého konceptu standardu došlo k události, která způsobila, že byl standard podstatně rozšířen - bylo to vytvoření knihovny standardních šablon STL (Standard Template Library) Alexandrem Stěpanovem. STL je sada generických rutin, které se mohou používat pro manipulaci s daty. Jsou výkonné a elegantní, bohužel však poněkud rozsáhlé. Následně po vytvoření hrubého konceptu schválila komise začlenění STL do normy C++, čím se původní rozsah C++ podstatně zvětšil. Nejzávažnější ovšem bylo, že zahrnutí STL mimo jiné zpomalilo proces standardizace C++.

Je pravdou, že standardizace C++ trvala mnohem déle, než kdokoliv čekal. Nicméně 14. listopadu 1997 prošel hotový koncept komisí a standard pro C++ se tak stal skutečností.

1.1 Krátké shrnutí

Časová náročnost: 1 minuta

```
C++ = C + podpora Objektovì Orientovaného
Programování
                        + Další pøíjemné vlastnosti
```

Některé z vlastností jazyka C++ si nyní shrneme:

- Třídy
- Konstruktory, destruktory
- Dědičnost
- Přetěžování funkcí, operátorů
- Vyjímky
- a mnoho dalších

To, že vám v tuto chvíli tyto pojmy nic neříkají, nevadí. Budou podrobněji vysvětleny v dalších kapitolách.

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

Obsah

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

Rejstřík

E

F

G

CH

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

Glosář

A

ABSTRAKTNÍ TŘÍDA Třída (class) v objektově orientovaném programovacím prostředí, která je navržena pouze jako nadřízená třída, ze které se odvozují vlastní funkční podtřídy (sub-class). Sama se nepoužívá; může dokonce obsahovat neúplnou sadu základních vlastností, ty jsou dodávány podtřídami.

ADRESA 1. Číslo označující umístění objektu v paměti počítače. 2. Adresa účastníka počítačové sítě.

ANSI Americká instituce, která vyvíjí americké průmyslové standardy ve shodě s mezinárodními standardy ISO.

APLIKACE Obecný pojem pro počítačový program či soubor programů, které plní daný účel. Pojem aplikace v mnohém splývá s pojmem program; obecně se program staví "níže", tedy do oblasti plnění menších, univerzálních a spíše pomocných funkcí, zatímco aplikace jsou větší programové balíky, plnící komplexnější úkoly. Aplikace se dělí podle oboru, druhu operačního systému, pod kterým pracují a někdy rovněž podle stupně komplexnosti.

AT&T American Telephone and Telegraph - Americká telefonní a telegrafní společnost - Založena v r. 1885, 1. 1. 1984 z rozhodnutí federálního soudu USA rozdělena na více nezávislých společností.

ASCII (Americký standardizovaný kód pro výměnu informací). Standardní sada znaků, definovaná a vytvořená v roce 1968. Definuje znaky, číslice a speciální písmena, použitelná v datové komunikaci. ASCII kód obsahuje 256 znaků, přičemž prvních 128 je zcela standardizováno prakticky na všech hardwarových i operačních platformách a dalších 128 je určeno pro další použití výrobci hardware či software. Velmi často druhých 128 znaků používá pro znaky lokálních abeced. Používá se dnes prakticky ve všech oborech výpočetní techniky.

B

BIT Základní jednotka informace vyjadřující dva stavy: ano - ne, pravda - nepravda, nízké napětí - vysoké napětí atd. Vyjadřuje se číslicemi 0 nebo 1. Pomocí nich lze tvořit čísla.

BYTE Jednotka informace složená z osmi bitů. Bajt je nejmenším adresovatelným prvkem v paměti počítače, přestože se lze odkazovat i na jeho jednotlivé bity.

BINÁRNÍ SOUSTAVA (binární soustava, dvojková číselná soustava) Polyadická poziční číselná soustava se základem 2. Stejně jako jsou ve známé desítkové soustavě čísla vyjadřována posloupnostmi číslic 0 až 9, jsou ve dvojkové soustavě užívány k vyjádření čísel pouze dvě číslice: 0 a 1.

BUFFER (vyrovnávací pamě•) Pamě•ový prostor pro přechodné uložení dat přesouvaných z rychlejšího pamě•ového média na pomalejší výstupní zařízení. Buffer plní úlohu mezičlánku zadržujícího přenášená data tak, aby je pomalé výstupní zařízení mělo okamžitě k dispozici a aby zároveň zdrojové médium nebylo zdržováno zdlouhavým přenosem. Je-li kapacita bufferu větší než objem přemís•ovaných dat, přesunou se všechna data do bufferu a zatížení zdrojového média je minimální. Vzhledem k tomu, že zdrojovým médiem je nejčastěji počítač (a cílovým tiskárna), je tato situace výhodná. Proto platí pravidlo: efektivita roste s velikostí bufferu, ne však nad určitou mez.

C

C Programovací jazyk vyšší úrovně, vyvinutý Denisem Hallem v roce 1972 pro implementaci operačního systému UNIX. Kombinuje prostředky vyšších jazyků s možnostmi assembleru. Obsahuje základní malé jádro, které je závislé na použitém počítači. Další funkce jsou pak uloženy v knihovnách, dostupných pomocí jazyka C. Jazyk C má pečlivě propracovanou standardizaci, a je proto implementován na většině systémů. Díky svým možnostem a hutné syntaxi je oblíbený zejména mezi profesionálními programátory.

C++ Nová verze programovacího jazyka C. (operátor ++ znamená v jazyku C následníka). Ke standardnímu jazyku C přidává nově objektově orientované programování a zlepšenou práci s funkcemi.

Č

ČÍSELNÁ SOUSTAVA Druh zápisu dat pomocí hierarchické soustavy. Podstatný je základ číselné soustavy, což je číslo, které určuje maximální počet prvků jednoho řádu. Kromě běžné desítkové soustavy se vyskytuje soustava binární (základ 2), osmičková (základ 8) a šestnáctková (hexadecimální, základ 16).

ČÍSLO S PLOVOUCÍ ŘÁDOVOU ČÁRKOU Číslo v počítači složené z celé a desetinné části. Je uloženo v pamě•ové buňce tak, že je umožněna změna jeho přesnosti - počtu desetinných míst v jistém rozsahu. Výhodou této interpretace desetinných čísel je vysoká přesnost při matematických operacích, nevýhodou jejich výpočtová složitost. K urychlení operací s plovoucí řádovou čárkou a pro práci s čísly s vysokým počtem desetinných míst slouží matematické koprocesory.

H

HARDWARE Souhrn hmotných technických prostředků umožňujících nebo rozšiřujících provozování počítačového systému. Hardware je sám počítač, jeho komponenty (pamě•, (viz) základní deska s obvody, záznamová média, periférie, vstupně/výstupní zařízení, přídavné karty atd.), tiskárny, sítě, speciální zařízení. hardware je vše kromě programového vybavení (software). Na hranici mezi oběma skupinami leží firmware.

HEXADECIMÁLNÍ SOUSTAVA Hexadecimální číselná soustava je číselná soustava se základem 16. Tomuto číslu odpovídá i počet cifer nutných k vyjádření čísla v hexadecimálním tvaru. Vzhledem k tomu, že běžně používaná desítková soustava má pouze deset cifer 0 až 9, musí hexadecimální soustava používat dalších šest cifer, za něž byly zvoleny písmena abecedy A až F. Skutečná hodnota jednotlivých hexadecimálních cifer je pro 0 až 9 totožná s desítkovou soustavou, nadstavené cifry A až F pak nabývají hodnot 10 až 15. Aby se odlišilo číslo zapsané v hexadecimálním tvaru neobsahující cifry A až F (a tedy na první pohled vypadající jako zápis decimálního čísla), používá se u hex. čísla speciální předpony nebo přípony (např. 10H, 10h, $0010 apod.). Výhodou hexadecimálního zápisu čísla v oblasti počítačového zpracování je skutečnost, že jím lze jednoduše vyjadřovat násobky hodnot bajtů. Např. čtyřbitové binární číslo 1110 (desítkově 14) lze vyjádřit jedinou hexadecimální cifrou E, tedy celý bajt (číslo s hodnotou v rozsahu 0 - 255) můžeme vyjádřit jako pouhé dvě hexadecimální cifry, dále pak slovo (číslo v rozsahu 0 - 65535 nebo podobně) lze vyjádřit čtyřmi hexadecimálními ciframi. Příklad: desítkové číslo 49534 se zapíše v hexadecimální soustavě jako C17E (tedy správněji C17Eh).

I

ISO Mezinárodní autoritativní organizace pro zavádění celosvětových standardů. V této organizaci jsou zastoupeny všechny významné státy světa svými normativními ústavy; ISO definuje a přijímá standardy z celé průmyslové oblasti včetně výpočetní techniky a komunikací. Více informací na www.iso.ch.

J

JAVA Sí•ově orientovaný interpretovaný programovací jazyk vyvinutý firmou Sun Microsystems. Byl vyvinut pro používání v prostředí Internetu a umožňuje tvorbu malých programů zvaných applety, které je možné načíst z Internetu a okamžitě spouštět; obecně však lze v Javě vytvořit cokoli, od operačního systému po podnikový informační systém. Díky tomu, že je Java interpretovaný jazyk, může běžeet na jakékoli platformě (tj. může být na tuto platformu portována velice rychle). Pomalost daná touto technologií by měla být vyvažována tím, že mikrokód interpretující Javu by měl být "zadrátován" jednak ve speciálních čipech, jednak (snad) i v budoucích verzí procesoru Pentium, a tím se dosáhne běžně rychlého chodu aplikací v Javě. Více viz www.java.sun.com.

JEDNOUŽIVATELSKÝ SYSTÉM Systém zkonstruovaný pro používání jedním uživatelem. Opakem je víceuživatelský systém.

K

KÓD Obecně jakýkoli druh zápisu dat, který dodržuje daná pravidla. Pojem používá se zejména u programů, kde je právě absolutně nutné dodržovat daná pravidla zápisu.

KOMPILACE Proces překladu zdrojového kódu programu (source code) z vyššího programovacího jazyka do strojového kódu (kódu instrukcí procesoru). Kompilaci provádí automaticky samostatný program - kompilátor.

KOMPILÁTOR Programový modul, který převádí instrukce ze zdrojového textu programu přímo do instrukcí strojového kódu procesoru. Kompilace se provede jen jednou a pak při každém dalším spuštění jsou pak přímo vykonávány strojové instrukce bez nutnosti překladu.

L

LEXIKÁLNÍ ANALÝZA První stupeň odlaďování při zpracování kódu v programovacím jazyce. Znaky zdrojového kódu jsou postupně čteny a seskupovány do funkčních celků (zvané lexemy či tokeny) - jsou to příkazy jazyka, identifikátory, citované řetězce atd. Tyto celky jsou pak předány parseru.

M

MAKRO Uživatelská definice posloupnosti více operací (např. stlačení kláves) nebo sekvence několika psaných příkazů. Makro se spouští jako jediný příkaz a je vlastně zkratkou zjednodušující uživateli provádění zdlouhavých posloupností kroků jejich nahrazením nějakou globální funkcí, která dokáže jednotlivé kroky provést automaticky. Makro může vystupovat např. v jednoduché formě náhrady některých delších řetězců nebo textů kratšími alternativami, které se při zpracování příslušného úseku textu rozvinou do původní podoby; makra se uplatňují také v aplikacích pro nahrazení posloupnosti opakovaných činností stlačením jediné kombinace kláves nebo např. v tabulkovém procesoru může makro nahradit definici složitějšího matematického výrazu.

MANTISA 1. Z matematického hlediska jde o část (viz) dekadického (viz) logaritmu čísla za desetinou tečkou. Např. dekadický logaritmus čísla 50 je přibližně 1.6989, odtud mantisa zaokrouhlená na čtyři desetinná místa odpovídá číslu 0.6989. 2. V počítačové terminologii je mantisa součástí popisu čísla s (viz) plovoucí desetinnou čárkou . Např. číslo 521 000 se v této notaci vyjádří jako 5.21 E+05, kde 5.21 je mantisa a E+05 je exponent (znamená 10^5).

O

OBJEKT 1. objektový kód, module . 2. Ohraničený, samostatný prvek, se kterým je možno manipulovat, obvykle graficky a manuálně. Objekty jsou důležitými stavebními i ovládacími prvky moderních programů, protože činí programy průhlednějšími při tvorbě a jednoduššími v ovládání.

OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ Způsob programování, při kterém je program složen z kompaktních prvků - objektů. OOP způsobilo určitou revoluci programování svou výbornou strukturovaností, modularitou a přehledností. OOP nesmírně usnadňuje používání knihoven - objektových modulů a tím umožňuje jednodušším způsobem tvořit rozsáhlé programové systémy.

OBJEKTOVÝ KÓD) Kód, který vytváří kompilátor nebo assembler ze zdrojového kódu programu. Objektový kód již obvykle obsahuje instrukce procesoru počítače nebo assembleru.

OPERAČNÍ SYSTÉM Podstatné softwarové vybavení počítače, které provádí základní řízení veškerých zdrojů počítače a komunikaci s uživatelem - je nenahraditelným rozhraním mezi počítačem (hardware) a buď uživatelem přímo nebo dalšími programy. Bez operačního systému není možné počítač používat - veškeré příkazy uživatele jsou operačním systémem přijímány a zpracovávány, a rovněž veškeré programy využívají ke své činnosti služeb operačního systému. Operační systém je přítomen na pevném disku počítače a načítá se do paměti při startu počítače - tomuto procesu se říká bootování.

P

PROCEDURA Část programu, která je tvořena pojmenovanou posloupností příkazů. Díky své ucelenosti a pojmenovanosti může být procedura volána svým jménem jinou procedurou nebo hlavním jádrem programu. Procedura může rovněž přebírat parametry a vracet návratové hodnoty.

PROGRAM Ucelený souhrn instrukcí (příkazů), pomocí kterých provádí počítač určitou činnost. Program je tvořen souborem nebo více soubory, které jsou v úhrnu dostatečně schopné provádět předepsanou činnost. Příbuznými termíny, mezi kterými lze těžko vymezit ostrou hranici, jsou: - aplikace , čímž se označuje obvykle komplexnější souhrn často i několika programů, které plní úkoly dané oblasti - software , čímž se označuje jakékoli programové vybavení počítače, které je ucelené spíše svým vnějším zjevem.

PŘEKLADAČ Programový prostředek určený pro převod posloupnosti příkazů (zdrojového kódu programu) do přímo a samostatně spustitelné podoby. Programovací jazyky jako C++, Pascal, Modula a další jsou navrženy jako kompilátory. Na druhé straně existují jazyky, které lze používat i v interpretované formě (např. Basic).

S

SOFTWARE Obecně jakékoli programové vybavení. Oblast software zahrnuje programy od základních vstupně/výstupních systémů (BIOS), přes operační systémy a veškeré aplikace, od jednoduchých utilit až po komplexní programové systémy. Software je obecně série programových instrukcí, uložená v přirozených celcích (souborech) na záznamovém médiu či v paměti počítače. Software sám je vždy "nehmotný", ke svému šíření a používání vždy potřebuje hardware.

U

UNIX Víceuživatelský a multiprocesingový operační systém vyvinutý (zejména pány Thompsonem a Ritchiem) v laboratořích firmy AT&T Bell v roce 1969 a poprvé komerčně použitý v počítačích PDP firmy Digital. Většina dalšího vývoje pak byla provedena na univerzitě v Berkeley, čímž vznikla edice UNIXu nazvaná BSD (Berkeley SOFTWARE Distribution) UNIX. Další vývoj UNIXu přebírá firma UNIX System Laboratories (USL), kterou pohltila později firma Novell, která ji však později předala nezávislé organizaci X/Open. Existuje široká škála implementací (forem, podob) operačního systému UNIX. Všechny však mají přibližně shodnou vrstevnatou strukturu, založenou na silném jádře. Přínos operačního systému UNIX je mj. v tom, že programy jsou v rámci jednotlivých implementací přenositelné, nebo• jsou obvykle programovány v jazyce C. Slovo UNIX se vyvinulo z výrazu Uniplexed Information and Computing System, což byl jakýsi oponent neúspěšného všezahrnujícího systému Multics, Multiplexed Information and Computing System. Ze slova Unics se pak vyvinul název UNIX.

V

VÍCEUŽIVATELSKÝ SYSTÉM Počítačový systém schopný zpracovávat požadavky více uživatelů současně. Může a nemusí jít vždy o systém na bázi počítačové sítě. Počítač, na němž pracuje střídavě více uživatelů v průběhu delšího časového intervalu, bývá považován spíše za jednouživatelský systém.

Z

ZDROJOVÝ KÓD Programové příkazy, napsané v programovacím jazyku, určené k interpretaci nebo kompilaci a následnému spouštění.

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

Rejstřík Obsah  Vysvětlivky Úvod

Vysvětlivky

Na této stránce uvádím strukturu stránek, typografickou konvenci a seznam použitých značek se kterými se můžete setkat na následujících stránkách.

Struktura stránek

- Na začátku každé kapitoly je uváděna časová náročnost kapitoly zaokrouhlená na desítky minut:

  Časová náročnost kapitoly: 1 hodina 10 minut
- Poté následuje stručný popis látky, která bude v kapitole probírána.
- Dále časové náročnosti jednotlivých částí kapitoly:

  Časová náročnost: 15 minut
- V části

  ### Potřebné znalosti

  jsou uváděny základní znalosti nutné k bezproblémovému zvládnutí látky probírané v této kapitole.
- Následuje probíraná látka ukončená částí nazvanou:

  Cvièení

  ve které jsou uváděny příklady (a samozřejmě i jejich řešení) pro procvičení právě probrané látky.
- V místech, kde je vhodné zopakovat (připomenout) si některé aspekty probírané látky, následuje část nazvaná:

  ### Opakování

Typografické konvence

V textu této elektronické publikace můžeme nalézt velké množství **příkladů**. Každý příklad je uveden slovem:

Příklad:

Pokud je tento příklad určen ke stažení (není to tedy jenom fragment) je uveden také svým **pořadovým číslem**:

Příklad 4.1:

Aby došlo k jednoznačnému odlišení příkladu od ostatního textu - je příklad **barevně zvýrazněn**:

```
int i,j,k;
i = j = k = 2;
```

Pokud po zdrojovém kódu nějakého programu následuje jeho funkční výpis je zapsán **neproporcionálním písmem**:

```
1 << 1 =         2        0x2
1 << 7 =         128      0x80
```

Protože je nutné zdůraznit i chyby, které se často vyskytují, jsou **chybné příklady** uvozeny následujícím způsobem:

Chybný příklad:

Použité značky

Šipka ukazující vpravo slouží k přechodu na **další** stránku (kapitolu) v probírané látce.

Šipka ukazující vlevo slouží k přechodu na **předcházející** stránku (kapitolu) v probírané látce.

Šipka ukazující nahoru slouží k přechodu na **vrchol** (top) aktuální stránky.

Šipka ukazující dolů slouží k přechodu na **patu** aktuální stránky.

Symbol kužele upozorňuje na ty části textu, které vyžadují zvýšenou pozornost. Zpravidla jsou takto označována místa s **důležitými informacemi**.

Symbolem diskety (floppy disku) jsou označeny výpisy zdrojových textů určených **ke stažení**. V případě Jazyka C půjde o stažení souborů s příponou *.c, v případě Jazyka C++ půjde o stažení souborů s příponou *.cpp.

Symbolem tužky je označovaná část **cvičení**. V této části jsou uváděny příklady pro samostudium, na kterých si je vhodné vyzkoušet správné pochopení probírané látky.

Symbol písmene C++ nás informuje o tom, že získáváme znalosti o **Jazyku C++**. Jazyk C++ je přece velice zajímavý, tak to dáme najevo :-)

Značka **Obsah** nás přesouvá na obsahovou část této publikace. Můžeme si zde vybrat část která nás zajímá, případně se přesunout na Rejstřík, Glosář, Vysvětlivky či Úvodní stránku.

Značka **Rejstřík** nás přesouvá na část, která nám pomáhá se snadněji zorientovat v používaných příkazech a funkcích. Umožňuje nám nalézt konkrétní kapitoly (či jejich části), příklady ve kterých jsou příkazy či funkce použity.

Značka **Glosář** nám pomáhá se snadněji orientovat v používaných termínech, které se snaží jednoduchou formou vysvětlit.

Značka **Úvod** nás přesouvá na úvodní stránku této elektronické publikace. Zde si můžeme vybrat zda se budeme vzdělávat v Jazyku C či C++.

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

3. Třídy

Časová náročnost kapitoly: 1 hodina

Tato část představí třídy a objekty. Řekneme si něco o konstruktorech a destruktorech, dědičnosti a in-line funkcích.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 jazyka C++ - tedy znát základní teoretické znalosti o zapouzdření, polymorfismu a dědičnosti. Získat základní znalosti o třídách a přetěžování.

3.1 Konstruktory a destruktory

Časová náročnost: 25 minut

Určitě víte, že v programech je zcela běžné, že některé jeho části vyžadují inicializaci. Potřeba inicializace je mnohem častější v případě, když pracujeme s objekty. Ve skutečnosti vyžadují určitý typ inicializace všechny objekty, kterém vytváříme. K ošetření této situace má C++ funkci **konstruktor**, která může být vložena do deklarace třídy. Konstruktor třídy je volán vždy, když je vytvářen objekt dané třídy. Všechny inicializace, které je nutno na objektu provést, může vykonat právě konstruktor.

Konstruktor má **stejné jméno jako třída**, jejíž je součástí a **nemá návratový typ**. Následující příklad ukazuje krátkou třídu, která obsahuje konstruktor.

Příklad 3.1:

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(); // konstruktor
  void show();
};

myclass::myclass()
{
  cout << "V konstruktoru\n";
  a = 10;
}

void myclass::show()
{
  cout << a;
}

int main()
{
  myclass ob;

  ob.show();

  return 0;
```

```
}
```

V tomto příkladu je hodnota **a** inicializována konstruktorem **myclass()**. Konstruktor je volán při vytváření objektu **ob**. Objekt je tedy vytvářen tehdy, když se provádí jeho deklarace. V C++ je deklarační příkaz proměnné vlastně příkazem činnosti.

Všimněte si, že **myclass()** nemá definovaný návratový typ. Vzhledem k formálním syntaktickým pravidlům **není** v C++ povoleno, aby měl konstruktor návratový typ.

Pro **globální objekty** je konstruktor objektu volán jen jednou, když se program začíná poprvé spouštět. Pro **lokální objekty** je konstruktor volán pokaždé, když je prováděn deklarační příkaz. Doplňkem konstruktoru je **destruktor**. Tato funkce je volána, když je objekt rušen. Když pracujete s objekty, je běžné, že se musí provést v souvislosti s rušením objektu nějaké akce. A právě o provádění těchto akcí se stará destruktor.

Jméno **destruktoru** je stejné jako **jméno jeho třídy**, ale je uvozeno znakem **~**.

Přidáme do předcházejícího příkladu destruktor.

Příklad 3.2:

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(); // konstruktor
  ~myclass(); //destruktor
  void show();
};

myclass::myclass()
{
  cout << "V konstruktoru\n";
  a = 10;
}

myclass::~myclass()
{
 cout << "V destruktoru\n";
}

void myclass::show()
{
  cout << a;
}

int main()
{
  myclass ob;

  ob.show();

  return 0;
}
```

Destruktor třídy je volán, když je **objekt rušen**. **Lokální objekty** jsou rušeny, když odcházejí **mimo oblast**. **Globální objekty** jsou rušeny, když **program končí**. **Není** možné získat **adresu** konstruktoru a destruktoru.

Následující program využívá objekt třídy **timer** ke zjištění délky intervalu mezi vytvořením zrušením objektu a zobrazuje uplynulý čas.

Příklad 3.3:

```
#include <iostream.h>
#include <time.h>

class timer {
  clock_t start;
public:
  timer(); // konstruktor
  ~timer(); // destruktor
};

timer::timer()
{
  start = clock();
}

timer::~timer()
{
  clock_t end;

  end = clock();
  cout << "Interval: " << (end-start) / CLOCKS_PER_SEC << "\n";
}

int main()
{
  timer ob;
  char c;

  // zpozdeni ...
  cout << "Stiskni klavesu a pote ENTER: ";
  cin >> c;

  return 0;
}
```

Funkci konstruktoru není možné předávat argumenty. Aby to bylo možné, je třeba přidat do deklarace a definice funkce konstruktoru vhodné argumenty. Pak, když deklarujeme objekt, specifikujeme argumenty.

Příklad 3.4:

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(int x); // konstruktor
  void show();
};

myclass::myclass(int x)
{
  cout << "V konstruktoru\n";
  a = x;
}
void myclass::show()
{
  cout << a << "\n";
}

int main()
{
  myclass ob(4);

  ob.show();

  return 0;
```

```
}
```

Konstruktor zde přebírá jeden argument. Hodnota předávaná do **myclass()** je využita k inicializace **a**. Hodnota **4** ve funkci **main()** specifikována v závorkách za **ob** je argumentem, který se předává do **x**, což je argument **myclass()** použitý v deklaraci **a**.

Na rozdíl od konstruktoru **nemůže** mít **destruktor** žádné argumenty. Příčina je snadno pochopitelná - neexistuje totiž mechanismus, který by umožnil předávat argument rušenému objektu.

3.2 Dědičnost

Časová náročnost: 17 minut

Dědičnost nám v C++ umožňuje, aby jedna třída zdědila vlastnosti jiné třídy. Je však nutné si definovat některé základní termíny. Když je třída děděná jinou, pak je třída, která je děděna, nazývána **základní třídou**. Dědící třída je pak nazvána **odvozenou třídou**. Obecně začíná proces dědění definicí základní třídy. Základní třída definuje všechny vlastnosti, které budou společné pro všechny odvozené třídy. Odvozená třída pak tyto obecné rysy dědí a přidává k nim vlastnosti, které jsou specifické pro danou třídu.

Na následujícím příkladu si ilustrujeme mnoho klíčových vlastností dědičnosti.

```
// definice základní tøídy
class base {
  int i;
public:
  void set_i(int n);
  int get_i();
};

// definice odvozené tøídy
class derived : public base {
  int j;
public:
  void set_j(int n);
  int mul();
};
```

Po jménu třídy **derived** je dvojtečka následována klíčovým slovem **public** a jménem třídy **base**. Překladači se tím říká, že třída **derived** bude dědit všechny prvky třídy **base**. Klíčové slovo **public** překladači říká, že **base** bude děděno tak, že všechny veřejné prvky základní třídy budou rovněž veřejnými prvky odvozené třídy. Ovšem všechny privátní prvky základní funkce zůstanou jejími privátními prvky a nebudou přímo dosažitelné žádnou odvozenou třídou.

Příklad 3.5:

```
#include <iostream.h>

// Zakladni trida
class base {
  int i;
public:
  void set_i(int n);
  int get_i();
};

// Odvozena trida
class derived : public base {
  int j;
public:
  void set_j(int n);
```

```
  int mul();
};

// nastaveni hodnoty i v zakladni tride
void base::set_i(int n)
{
  i = n;
}

// vraceni hodnoty i v zakladni tride
int base::get_i()
{
  return i;
}

// nastaveni hodnoty j v odvozene tride
void derived::set_j(int n)
{
  j = n;
}

// vraceni hodnoty j v odvozene tride
int derived::mul()
{
  // odvozena trida muze volat verejne funkce zakladni tridy
  return j * get_i();
}

int main()
{
  derived ob;

  ob.set_i(10);
  ob.set_j(4);

  cout << ob.mul();

  return 0;
}
```

Všimněte si, že **mull()** volá funkci **get_i()**, která je členem základní třídy **base**, nikoliv třídy **derived** bez propojením s nějakým konkrétním objektem. Je to možné, protože veřejné členy **base** se staly veřejnými členy **derived**.

Obecný tvar pro dědění základní třídy je následující:

```
class jméno-odvozené-tøídy : pøístupový-specifikátor jméno-základní-tøídy {
  .
  .
  .
};
```

Příkazový specifikátor musí být jedno z následujících klíčových slov: **public**, **private** nebo **protected**.

3.3 Ukazatelé objektu

Časová náročnost: 5 minut

Zatím jsme pracovali s objekty pomocí tečkových operátorů. Ke členu objektu je možné přistupovat přes ukazatel objektu. Když využíváme přístup pomocí ukazatele je vhodnější použít šipkový operátor ->

Ukazatele objektu definujeme stejným způsobem jako definujeme ukazatel jakéhokoliv jiného typu proměnné. Tedy pomocí *. K získání adresy objektu poté využijeme znaku **&**

Příklad 3.6:

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(int x); // konstruktor
  int get();
};

myclass::myclass(int x)
{
  a = x;
}

int myclass::get()
{
  return a;
}

int main()
{
  myclass ob(120);  // vytvareni objektu
  myclass *p;  // vytvareni ukazatele na objekt

  p = &ob; // vlozeni adresy ob do p

  cout << "Hodnota objektu: " << ob.get();
  cout << "\n";

  cout << "Hodnota pointeru: " << p->get();

  return 0;
}
```

3.4 In-line funkce

Časová náročnost: 10 minut

V C++ lze definovat funkce, které vlastně nejsou volány, přesněji řečeno jsou rozšiřovány dle místa každého volání. Je to stejný způsob jako makra s argumenty v jazyce C. Výhoda vkládaných in-line funkcí je, že s voláním funkce, ani s návratovým mechanismem není asociován žádný overhead. Neboli, že funkce in-line mohou být prováděny mnohem rychleji než normální funkce. Nevýhodou in-line funkcí je to, že jsou-li veliké a jsou-li volány často, program se podstatně zvětší. Proto jsou jako in-line funkce deklarovány pouze krátké funkce. Když definujeme in-line funkci prostě před její identifikátor vložíme klíčové slovo **inline**.

Příklad 3.7:

```
#include <iostream.h>

inline int even(int x)
{
  return !(x%2);
}

int main()
{
  if(even(10)) cout << "10 je sude\n";
  if(even(11)) cout << "11 je sude\n";

  return 0;
}
```

Dále platí:

1. Každá in-line funkce musí být definována před jejím prvním voláním. Jestliže tomu tak není, překladač nemá způsob jak by poznal, že se má rozšířit na in-line
2. Funkce in-line může být překladačem optimalizována mnohem důkladněji než makro.
3. Specifikátor **inline** není příkaz, ale požadavek pro překladač. Když z nějakých důvodů není překladač schopen tento požadavek splnit je funkce přeložena jako normální a požadavek inline je ignorován.

Následující příklad nám umožní třikrát přetížit **min()** a přitom je tato funkce deklarována jako **inline**.

Příklad 3.8:

```
#include <iostream.h>

inline int min(int a, int b)
{
  return a<b ? a : b;
}

inline long min(long a, long b)
{
  return a<b ? a : b;
}

inline double min(double a, double b)
{
  return a<b ? a : b;
}

int main()
{
  cout << min(-10, 10) << "\n";
  cout << min(-10.01, 100.002) << "\n";
  cout << min(-10L, 12L) << "\n";

  return 0;
}
```

Jestliže je definice členské funkce dostatečně krátká, může být tato definice zahrnuta přímo do deklarace třídy. Tímto způsobem se funkce automaticky stane in-line funkcí, je-li to možné. Je-li funkce definována uvnitř deklarace třídy není již klíčové slovo **inline** nutné.

Příklad 3.9:

```
#include <iostream.h>

class samp {
  int i, j;
public:
  samp(int a, int b);
  int divisible() { return !(i%j); }  // inline funkce
};

samp::samp(int a, int b)
{
  i = a;
  j = b;
}

int main()
{
  samp ob1(10, 2), ob2(10, 3);

  // pravda
  if(ob1.divisible()) cout << "10 je beze zbytku delitelne 2\n";

  // nepravda
  if(ob2.divisible()) cout << "10 je beze zbytku delitelne 3\n";

  return 0;
}
```

Cvièení

1) Co je špatného na konstruktoru v následujícím fragmentu?

```
class priklad {
   double a, b, c;
public:
   double priklad();   //chyba, proc?
};
```

Řešení

2) Vytvořte třídu **t_and_d**, která předává aktuální systémový čas a datum, jako argument svému konstruktoru, když je vytvářena. A• třída obsahuje členskou funkci, která zobrazí čas a datum na obrazovce (Rada: Pro nalezení a zobrazení data a času použijte standardní funkce ze standardní knihovny) Řešení

3) Je dána následující třída. Vytvořte dvě odvozené třídy **rectangle** a **isosceles**. A• každá třída obsahuje funkci pojmenovanou **area()** která dle zadání vrací plochu čtyřúhelníku nebo rovnoramenného trojúhelníku. Pro inicializaci **height** a **width** použijte konstruktor s argumenty.

```
class area_c1 {
public:
   double height;
   double width;
};
```

Řešení

4) Co zobrazí tento program?

```
#include <iostream.h>

int main()
{
  int i = 10;
  long l = 1000000;
  double d = -0.0009;

  cout << i << ' ' << l << ' ' << d;
  cout << "\n";

  return 0;
}
```

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---


Obsah

4. Objekty

Časová náročnost kapitoly: 45 minut

V této části se dozvíme něco o přiřazování objektů, předávání objektů funkcím a vracení objektů z funkcí. Také si něco povíme o tzv. spřátelených funkcích.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 a 3 jazyka C++ - znát a používat konstruktory a destruktory, dědičnost a in-line funkce.

4.1 Přiřazování objektů

Časová náročnost: 8 minut

Pokud existují dva objekty **stejného typu**, může být jeden objekt přiřazen druhému. Při přiřazování jednoho objektu druhému, se vytváří **bitová kopie** všech datových členů. Když je například objekt **obj1** přiřazen objektu **obj2**, pak je obsah všech dat objektu **obj1** kopírován do ekvivalentních členů objektu **obj2**. Viz následující příklad.

Příklad 4.1:

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
  myclass obj1, obj2;

  obj1.set(10, 4);

  // prirazeni obj1 do obj2
  obj2 = obj1;

  obj1.show();
```

```
  obj2.show();

  return 0;
}
```

Objekt **obj1** má nastaveny proměnné **a**, **b** na hodnoty 10 a 4. Přiřazení způsobí, že aktuální hodnota **obj1.a** je přiřazena do **obj2.a** a **obj1.b** je přiřazena do **obj2.b**

Při přiřazení mezi dvěma objekty zůstávají **data** v těchto objektech **identická**. **Objekty** však jsou stále **oddělené**.

Pokud však objekty nejsou stejného typu, pak se při kompilaci objeví chybové hlášení. Nestačí aby byly objekty stejného typu. Musí si také být fyzicky podobné - jejich typová jména musí být shodná.

V následujícím příkladě jsou **myclass** a **yourclass** fyzicky stejné, ale mají různá typová jména a proto je překladač bere jako rozdílné typy a zobrazí chybové hlášení.

Chybný příklad:

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};

/* Tato trida je podobna myclass, ale pouziva jine jmeno tridy a jevi
   se tedy prekladaci jako jiny typ
*/
class yourclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
  myclass o1;
  yourclass o2;

  o1.set(10, 4);

  o2 = o1; // ERROR, objekty nejsou stejneho typu

  o1.show();
  o2.show();

  return 0;
```

```
}
```

4.2 Předávání objektů funkcím

Časová náročnost: 12 minut

Objekty mohou být jako argumenty **předávány funkcím** stejným způsobem, jako jsou předávána jiná data. Prostě se deklarují argumenty funkce jako **typ třídy** a pak se objekt dané třídy použije jako argument při volání funkce.

Stejně jako jiné typy proměnných může být do funkce předána také **adresa objektu**. Předává se tak, že argument použitý ve volání může být modifikován funkcí.

Volání hodnotou:

Příklad 4.2:

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* nastavi o.i na jeho ctverec. Nema to vliv na objekt
   pouzity k volani sqr_it()
*/
void sqr_it(samp o)
{
   o.set_i(o.get_i() * o.get_i());

  cout << "Kopie a ma hodnotu i " << o.get_i();
  cout << "\n";
}

int main()
{
  samp a(10);

  sqr_it(a);  //a predavano hodnotou

  cout << "a.i zustava v main() nezmeneno: ";
  cout << a.get_i();

  return 0;
}
```

Předání adresy:

Příklad 4.3:

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* Nastavi o.i na jeho ctverec. To ma vliv na
   volajici argument
*/
void sqr_it(samp *o)
{
   o->set_i(o->get_i() * o->get_i());

  cout << "Kopie a ma hodnotu i: " << o->get_i();
  cout << "\n";
}

int main()
{
  samp a(10);

  sqr_it(&a); // predava adresu

  cout << "a.i ve funkci main() ma zmenenou hodnotu: ";
  cout << a.get_i();

  return 0;
}
```

Když je při předávání objektu do funkce vytvořena kopie objektu, znamená to, že začíná existovat **nový objekt**. Když je pak funkce, do níž byl objekt předán ukončena, je **kopie argumentu zrušena**.

Když je vytvářena kopie pro volání funkce, **není konstruktor volán**. Konstruktor se používá pro inicializaci určitých vlastností objektu a nemusí být proto volán při vytváření kopie existujícího objektu pro předávání funkci. Kdyby se volal, **změnil** by se obsah objektu.

Když funkce končí a kopie je rušena, pak se **destruktor volá**. Je to proto, že objekt mohl provést nějakou operaci, která musí být zrušena.

Na následujícím příkladě si výše uvedené skutečnosti ověříme.

Příklad 4.4:

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) {
    i = n;
    cout << "Konstruktor\n";
  }
  ~samp() { cout << "Destruktor\n"; }
  int get_i() { return i; }
};

int sqr_it(samp o)
{
  return o.get_i() * o.get_i();
}

int main()
{
  samp a(10);

  cout << sqr_it(a) << "\n";

  return 0;
}
```

Výstup:

```
Konstruktor
Destruktor
100
Destruktor
```

4.3 Vracení objektů funkcemi

Časová náročnost: 8 minut

Stejně jako lze předávat objekty funkcím, mohou funkce **objekty vracet**. Aby to bylo možné musíme nejprve deklarovat funkci tak, aby vracela třídu. Pak vrátí objekt takového typu, který určí příkaz **return**.

Když je funkcí vrácen objekt, vytváří se **dočasný objekt**, v němž je uložena návratová hodnota. Tento objekt je funkcí skutečně vrácen a jakmile je hodnota vrácena je dočasný objekt **zrušen**. Zrušení dočasného objektu však může mít některé nechtěné efekty - viz následující příklad.

Při vracení objektů z funkcí musíme být velice opatrní, jestliže tyto objekty obsahují destruktor a vrácený objekt odchází mimo rozsah, jakmile je volající rutině vrácená hodnota. Jestliže má objekt vrácený funkcí svůj vlastní destruktor, uvolní se dynamicky přidělená pamě•, a to přesto, že objekt přiřazený k vrácené hodnotě je stále používán.

Příklad 4.5:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class samp {
  char *s;
public:
  samp() { s = '\0'; }
  ~samp() { if(s) free(s); cout << "Uvolnuji s\n"; }
  void show() { cout << s << "\n"; }
  void set(char *str);
};

// Nacitani retezce
void samp::set(char *str)
{
  s = (char *) malloc(strlen(str)+1);
  if(!s) {
    cout << "Error\n";
    exit(1);
  }

  strcpy(s, str);
}

// Vraceni objektu
samp input()
{
  char s[80];
  samp str;

  cout << "Zadej retezec: ";
  cin >> s;

  str.set(s);
  return str;
}

int main()
{
  samp ob;
```

```
  ob = input();  // zde vznika chyba
  ob.show();

  return 0;
}
```

Výstup:

```
Zadej retezec: petr
Uvolnuji s
Uvolnuji s
¹_gÛÔôÒ
Uvolnuji s
```

Destruktor je volán **3x**. Poprvé když lokální objekt **str** odchází mimo rozsah při návratu **input()**. Podruhé, když je rušen dočasný objekt vrácený od **input()**. Potřetí když program končí, je volán destruktor objektu **ob** v **main()**

4.4 Spřátelené funkce

Časová náročnost: 6 minut

Někdy můžeme chtít, aby měla funkce přístup k privátním členům třídy bez toho, aby byla členem třídy. Proto C++ definuje **spřátelené funkce**. Spřátelená funkce není členem třídy, ale má přístup k jejím privátním prvkům.

Spřátelená funkce je definována jako řádná funkce bez členů. Uvnitř deklarace třídy, jejímž bude přítelem, je zahrnut také její prototyp s klíčovým slovem **friend**.

V následujícím příkladu deklarujeme **myclass()** uvnitř její konstruktor a jejího přítele. Protože jde o přítele má přístup k jejím privátním členům.

Příklad 4.6:

```
#include <iostream.h>

class myclass {
  int n, d;
public:
  myclass(int i, int j) { n = i; d = j; }
  friend int isfactor(myclass ob); //spratelena funkce
};

int isfactor(myclass ob)
{
  if(!(ob.n % ob.d)) return 1;
  else return 0;
```

```
}

int main()
{
  myclass ob1(10, 2), ob2(13, 3);

  if(isfactor(ob1)) cout << "10 je delitelne 2\n";
  else cout << "10 neni delitelne 2\n";

  if(isfactor(ob2)) cout << "13 je delitelne 3\n";
  else cout << "13 neni delitelne 3\n";

  return 0;
}
```

Musíme si uvědomit, že spřátelená funkce **není** členem třídy jejímž je přítelem. Není tedy možné volat spřátelenou funkci s využitím jména objektu a tečkového či šipkového operátoru. Přestože tedy spřátelená funkce zná privátní prvky třídy, s níž je spřátelená, může k nim přistupovat pouze přes objekty třídy.

Spřátelená funkce **není** dědičná. Proto, když základní třída obsahuje spřátelenou funkci, pak tato třída není spřátelena s odvozenou funkcí. Dále platí že spřátelená funkce **může být** spřátelena s více třídami.

Cvièení

1) Co je špatného v následujícím fragmentu?

```
#include <iostream.h>

class cl1 {
  int i, j;
public:
  cl1(int a, int b) { i = a; j = b; }
  // ...
};

class cl2 {
  int i, j;
public:
  cl2(int a, int b) { i = a; j = b; }
  // ...
};

int main()
{
  cl1 x(10, 20);
  cl2 y(0, 0);
  x = y;

  // ...
}
```

Řešení

2) Když je funkci předáván objekt, je vytvářena jeho kopie. Při návratu této funkce je volán destruktor kopie. Co je potom špatné v následujícím příkladu?

```
#include <iostream.h>
#include <stdlib.h>

class dyna {
  int *p;
public:
  dyna(int i);
  ~dyna() { free(p); cout << "Uvolnuji \n"; }
  int get() { return *p; }
};

dyna::dyna(int i)
{
  p = (int *) malloc(sizeof(int));
  if(!p) {
    cout << "Allocation failure\n";
    exit(1);
  }

  *p = i;
}

// vraceni zaporne hodnoty
int neg(dyna ob)
{
  return -ob.get();
}

int main()
{
  dyna o(-10);

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  dyna o2(20);
  cout << o2.get() << "\n";
  cout << neg(o2) << "\n";

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  return 0;
}
```

Řešení

3) Vytvořte třídu **who**. Nech• její konstruktor přebírá jednoznakový argument, který

bude použit pro identifikaci objektu. A• konstruktor při vytváření objektu vypisuje:

```
Vytvarim who #x
```

Kde **x** je identifikační znak každého objektu. Když je objekt rušen, a• se zobrazuje zpráva:

```
Rusim who #x
```

Nakonec vytvořte funkci **make_who()**, která vrací objekt **who**. Každému objektu přiřaďte unikátní jméno. Řešení

4) Může být adresa objektu využita funkcí jako argument? Řešení

5) Co jsou to spřátelené funkce? Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

Obsah

5. Pole, ukazatele a odkazy

Časová náročnost kapitoly: 1 hodina

Tato kapitola přináší informace o polích objektů a ukazatelích objektů. Na závěr se zmíníme o důležité inovaci C++ odkazech.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 až 4 jazyka C++ - tedy umět používat objekty, předávat je funkcím a získávat je z funkcí zpět, mít znalosti o friend funkcích.

5.1 Pole objektů

Časová náročnost: 10 minut

Objekty jsou také proměnné a mají tytéž možnosti a atributy jako jiné typy proměnných. Proto je také možné, aby byly sdružovány do polí. Syntaxe deklarace pole je zcela stejná jako syntaxe polí jiných typů proměnných. Pole objektů jsou přístupná jako pole jiných typů proměnných.

Následující program vytvoří čtyř prvkové pole objektů třídy **samp** a pak plní každý prvek **a** hodnotou z rozsahu 0 - 3.

Příklad 5.1:

```
#include <iostream.h>

class samp {
  int a;
public:
  void set_a(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4];
```

```
  int i;

  for(i=0; i<4; i++) ob[i].set_a(i);

  for(i=0; i<4; i++) cout << ob[i].get_a( );

  cout << "\n";

  return 0;
}
```

Jestliže třída obsahuje konstruktor, může být pole objektů inicializováno. Viz následující příklad.

Příklad 5.2:

```
#include <iostream.h>

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4] = { -1, -2, -3, -4 };
  int i;

  for(i=0; i<4; i++) cout << ob[i].get_a() << ' ';

  cout << "\n";

  return 0;
}
```

Samozřejmě můžeme mít i vícerozměrná pole objektů. Následující příklad vytváří dvourozměrné pole objektů.

Příklad 5.3:

```
#include <iostream.h>

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
    5, 6,
    7, 8
  };
  int i;


  for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][1].get_a() << "\n";
  }

  cout << "\n";

  return 0;
}
```

5.2 Ukazatele objektů a aritmetika ukazatelů

Časová náročnost: 5 minut

Když je použit ukazatel objektu, pak je na jeho členy ukazováno namísto tečkového operátoru přes šipkový **operátor**.

Aritmetika ukazatelů využívající ukazatel objektu je stejná jako u ostatních typů dat - provádí se ve vztahu k typu objektu. Když je například ukazatel objektu dekrementován, ukazuje na předcházející objekt.

Následující příklad ukazuje použití aritmetiky ukazatelů. Pokaždé, když bude ukazatel inkrementován, bude ukazovat na následující prvek pole.

Příklad 5.4:

```
#include <iostream.h>

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

int main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
    samp(7, 8)
  };
  int i;

  samp *p;
  p = ob; // pocatecni adresa pole

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    cout << p->get_b() << "\n";
    p++;  // nasledujici objekt
  }

  cout << "\n";

  return 0;
}
```

5.3 Ukazatel this

Časová náročnost: 6 minut

Ukazatel **this** je speciálním ukazatelem jazyka C++. Je to ukazatel na objekt vytvářející volání a automaticky se předává funkci při jejím prvním volání. Také platí, že ukazatel **this** je předáván pouze členským funkcím. Spřátelené funkce ukazatel **this** mít nebudou.

Následující příklad vytváří jednoduchou třídu **inventory** a využívá ukazatele **this**.

Příklad 5.5:

```
#include <iostream.h>
#include <string.h>

class inventory {
  char item[20];
  double cost;
  int on_hand;
public:
  inventory(char *i, double c, int o)
  {
     strcpy(this->item, i); // pristup clenu
     this->cost = c;  // pres ukazatel this
     this->on_hand = o;
   }
   void show();
};

void inventory::show()
{
  cout << this->item; // pro pristup k clenum
  cout << ": $" << this->cost;
  cout << "  pocet kusu: " << this->on_hand << "\n";
}

int main()
{
  inventory ob("cena", 4.95, 4);

  ob.show();

  return 0;
}
```

Nyní jsou členské proměnné dosažitelné přímo přes pointer **this**. V rámci **show()** jsou tedy tyto řádky totožné:

```
const = 12.25;
this->const = 12.25;
```

Ukazatel **this** má velké možnosti použití, například u přetěžování. Je **důležité** si uvědomit, že standardně všechny členské funkce automaticky přebírají ukazatel na volající objekt.

5.4 New a Delete

Časová náročnost: 17 minut

V jazyce C existují standardní funkce **malloc()** a **free()** pro dynamickou alokaci. Tyto funkce jsou samozřejmě dostupné i v jazyce C++, ale existuje zde i mnohem bezpečnější a jednodušší způsob přidělování a uvolňování paměti. V C++ můžeme přidělit pamě• pomocí **new** a uvolnit ji pomocí **delete**.

Tyto operátory mají následující formát:

```
privátní-promìnná = new typ
delete privátní-promìnná
```

Typem je míněn specifikátor objektu, jemuž chceme přidělit pamě•. **Privátní proměnná** je ukazatelem k tomuto typu.

Operátor **new** vrací ukazatel do dynamicky přidělované paměti, která je dostatečně velká pro uložení objektu typu **typ**. Operátor **delete** uvolňuje pamě•. Může být volán pouze s ukazatelem, který byl předtím přidělen při volání **new**. Při volání **new**, nebo **delete** se špatným ukazatelem může dojít k havárii systému.

Není-li dostatek paměti pro provedení alokačního příkazu, operátor **new** vrátí prázdný ukazatel nebo vyjímku. O vyjímkách se zmíníme v některé z dalších kapitol.

Ačkoliv provádějí **new** a **delete** činnosti podobné **malloc()** a **free()**, mají několik výhod:

1. **New** automaticky přiděluje dostatek paměti pro uložení objektu daného typu. Nemusíme tedy pro výpočet potřebného místa používat například operátoru **sizeof()**.
2. **New** automaticky vrací ukazatel určeného typu.
3. **New** i **delete** mohou být přetěžovány.
4. Je možné inicializovat dynamicky alokovaný objekt.

Následující jednoduchý příklad nám ukáže použití operátorů **new** a **delete** pro přidělení a uvolnění paměti pro hodnotu integer.

Příklad 5.6:

```
#include <iostream.h>

int main()
{
  int *p;

  p = new int; // alokace mista pro int
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }

  *p = 1000;

  cout << "Hodnota integer: " << *p << "\n";

  delete p;  // uvolneni pameti

  return 0;
}
```

Počáteční hodnotu můžeme dynamicky alokovaným objektům přidělit podle následujícího formátu:

```
privátní-promìnná = new typ (poèáteèní-hodnota)
```

Pro alokaci jednorozměrného pole můžeme použít tento formát:

```
privátní-promìnná = new typ [rozmìr]
```

Ve výše uvedeném případě ukáže **privátní proměnná** na začátek pole daného **rozměrem**.

Pozn.: Nelze inicializovat dynamicky alokované pole.

Ke zrušení dynamicky alokovaného pole lze použít následující formát **delete**:

```
delete [] privátní-promìnná
```

Výše uvedená syntaxe způsobí, že se zavolá destruktor pro každý prvek pole. **Nezpůsobí** to, že by byla privátní-proměnná uvolněná vícekrát.

Následující program alokuje pole pro hodnoty typu integer.

Příklad 5.7:

```
#include <iostream.h>

int main()
{
  int *p;

  p = new int [5]; // alokujeme misto pro pole 5 integeru
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }

  int i;

  for(i=0; i<5; i++) p[i] = i;

  for(i=0; i<5; i++) {
    cout << "Hodnota integer v p[" << i << "]: ";
    cout << p[i] << "\n";
  }

  delete [] p;  // uvolneni pameti

  return 0;
}
```

Následující program vytvoří dynamické pole objektů a nakonec volá destruktor pro každý prvek pole.

Příklad 5.8:

```
#include <iostream.h>

class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  ~samp() { cout << "Destruktor...\n"; }
  int get_product() { return i*j; }
};

int main()
{
  samp *p;
  int i;

  p = new samp [10]; // alokace objektu pole
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }
```

```
  for(i=0; i<10; i++)
    p[i].set_ij(i, i);

  for(i=0; i<10; i++) {
    cout << "Produkt [" << i << "] is: ";
    cout << p[i].get_product() << "\n";
  }

  delete [] p;
  return 0;
}
```

5.5 Odkazy

Časová náročnost: 5 minut

Odkaz je implicitní ukazatel, který ve všech významech a účelech vystupuje jako další jméno proměnné. Jsou tři způsoby použití odkazu:

1. Odkaz může být funkci předán.
2. Odkaz může být funkcí vrácen.
3. Vytvoření nezávislého odkazu.

První výhodou odkazu je to, že již nemusíme pamatovat na předávání adresy argumentu. Když je použit argument odkazu, je adresa předána automaticky. Používání odkazu nabízí čitelnější a elegantnější rozhraní, než explicitní mechanismus ukazatele. Další výhodou je, že při předávání objektu funkci jako odkaz se **nevytváří** kopie. Tímto způsobem se můžeme vyhnout problémům spojeným s tím, že destruktor poškodí argumentu něco užitečného jinde v programu.

Existuje omezení vůči všem typům odkazů. Nemůžeme se **odkazovat na jiný odkaz**. Nemůžeme získat **adresu odkazu**. Nemůžeme vytvářet **pole odkazů**. Nemůžeme mít odkazy **bitového pole**. Odkazy musí být **inicializovány**, aniž by byly členy tříd, nebo vracely hodnotu, nebo byly argumenty funkcí.

Klasickým případem předávání argumentů pomocí odkazů je funkce, která navzájem vymění hodnoty dvou argumentů, pomocí niž je volána. V našem případě půjde o výměnu dvou integer argumentů.

Příklad 5.9:

```
#include <iostream.h>

void swapargs(int &x, int &y);

int main()
{
  int i, j;

  i = 10;
  j = 19;

  cout << "Pred zamenou: ";
  cout << "i: " << i << ", ";
  cout << "j: " << j << "\n";

  swapargs(i, j);

  cout << "Po zamene: ";
  cout << "i: " << i << ", ";
  cout << "j: " << j << "\n";

  return 0;
}

void swapargs(int &x, int &y)
{
  int t;

  t = x;
  x = y;
  y = t;
}
```

5.6 Předávání odkazů objektům

Časová náročnost: 5 minut

Když je funkci předáván objekt standardním způsobem s voláním hodnotou vytváří se kopie objektu. Ačkoliv není volán konstruktor je při návratu z funkce volán destruktor.

Když předáváme objekt odkazem **nevytváří** se žádná kopie a ani při návratu z funkce **není volán** destruktor. Ovšem změny provedené na objektu v rámci funkce **ovlivní** i objekt použitý jako argument.

Následující program demonstruje předávání objektu odkazem.

Příklad 5.10:

```
#include <iostream.h>

class myclass {
  int who;
public:
  myclass(int n) {
    who = n;
    cout << "Konstruktor " << who << "\n";
  }
  ~myclass() { cout << "Destruktor " << who << "\n"; }
  int id() { return who; }
};

// predavani odkazem
void f(myclass &o)
{
  //  . operator je stale pouzivan
  cout << "Prijato " << o.id() << "\n";
}

int main()
{
  myclass  x(1);

  f(x);

  return 0;
}
```

Když přistupujeme ke členům nějakého objektu pomocí odkazu používejme **tečkový operátor**!

5.7 Vracení odkazů

Časová náročnost: 4 minuty

Funkce může **vrátit odkaz**. Vrácení odkazu může být použito při přetěžování určitých typů operátorů. Zároveň může být použito pro povolení, zda smí být funkce použita na levé straně přiřazovacího příkazu.

Opět si vracení odkazů demonstrujeme na příkladu.

Příklad 5.11:

```
#include <iostream.h>

int &f(); // return ukazatel
int x;

int main()
{
  f() = 100; // prirazeni 100 odkazu vracenemu f()

  cout << x << "\n";

  return 0;
}

int &f()
{
  return x;  // vraci odkaz na x
}
```

Funkce **f()** vrací odkaz na integer. Proto taky příkaz

```
return x;
```

nevrací hodnotu globální proměnné **x**, ale vrací **adresu** ve formátu **odkazu**.

5.8 Nezávislé odkazy

Časová náročnost: 3 minuty

Nezávislý odkaz je odkazová proměnná, která je ve všech důsledcích dalším jménem pro další proměnnou. Protože nemohou být odkazy přiřazeny k nové hodnotě, musí být **nezávislý odkaz** při své deklaraci **inicializován**.

Zde je uveden program, který obsahuje nezávislý odkaz.

Příklad 5.12:

```
#include <iostream.h>

int main()
{
  int x;
  int &ref = x; // nezavisly odkaz

  x = 10;    // tyto dva prikazy
  ref = 10;  // jsou funkcne shodne

  ref = 100;
  // 100 se vytiskne 2x
  cout << x << ' ' << ref << "\n";

  return 0;
}
```

Cvièení

1) S využitím deklarované třídy vytvořte desetiprvkové pole a inicializujte prvek **ch** hodnotami **A** až **J**. Následně toto pole vytiskněte.

```
#include <iostream.h>

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};
```

Řešení

2) Přepište v následujícím programu všechny příslušné odkazy na explicitní ukazatel **this**.

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int n, int m) { a = n; b = m; }
  int add() { return a+b; }
  void show();
};

void myclass::show()
{
  int t;

  t = add();
  cout << t << "\n";
}

int main()
{
  myclass ob(10, 14);

  ob.show();

  return 0;
}
```

Řešení

3) Napište program, který bude při použití **new** dynamicky alokovat **float**, **double** a **char**. Zadejte do těchto dynamických proměnných hodnoty a zobrazte je. Nakonec uvolněte celou dynamicky přidělenou pamě• pomocí **delete**. Řešení

4) Převeďte následující kód na jeho ekvivalent, který používá **new**.

```
char *p;

p = (char *) malloc(100);
// ...
strcpy(p, "Test");
```

Řešení

5) Co je v následujícím programu chybné?

```
#include <iostream>

void triple(double &num);

int main()
{
```

```
  double d = 7.0;

  triple(&d);

  cout << d;

  return 0;
}

void triple(double &num)
{
  num = 3 * num;
}
```

Řešení

6) Co je to ukazatel **this**? Řešení

7) Co je **odkaz**? Jaká je výhoda použití argumentu odkazu? Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

6. Přetěžování funkcí

Časová náročnost kapitoly: 40 minut

V této části si povíme něco o přetěžování funkcí, přetěžování konstruktorů, o tom, jak dávat funkcím standardní argumenty a jak se vyhnout nejednoznačnosti při přetěžování.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 až 5 jazyka C++ - tedy rozvinout své znalosti o objektech - umět vytvářet jejich pole, chápat aritmetiku jejich ukazatelů. Znát ukazatel this a používat operátory new, delete a odkazy.

6.1 Přetěžování konstruktorů

Časová náročnost: 12 minut

Bez větších potíží lze **přetížit konstruktor** třídy. Je to výhodné hned z několika důvodů.

- Získání větší flexibility
- Podpora polí
- Vytváření kopírovacích konstruktorů

Nejčastějším použitím přetíženého konstruktoru je zajištění možnosti voleb - zda inicializovat či ne.

V následujícím příkladu si uvedeme jak přetěžovat konstruktor. Všimněte si, že pokud odstraníme konstruktor s argumenty program se nepřeloží, protože mu bude tento konstruktor chybět. Jsou tedy nutné oba dva.

Příklad 6.1:

```
#include <iostream.h>

class myclass {
  int x;
public:
  myclass() { x = 0; } // konstruktor bez inicilizace
  myclass(int n) { x = n; } // inicializovany konstruktor
  int getx() { return x; }
};

int main()
{
  myclass o1(10); // deklarace s pocatecni hodnotou
  myclass o2; // deklarace bez pocatecni hodnoty

  cout << "o1: " << o1.getx() << '\n';
  cout << "o2: " << o2.getx() << '\n';

  return 0;
}
```

Další důvod pro přetěžování konstruktorů je, aby se v programu mohly objevit jak samostatné objekty, tak i pole objektů.

Příklad 6.2:

```
#include <iostream.h>

class myclass {
  int x;
public:
  myclass() { x = 0; } // konstruktor bez inicializace
  myclass(int n) { x = n; } // inicializovany konstruktor
  int getx() { return x; }
};

int main()
{
  // deklarace pole bez inicializace
  myclass o1[10];
  // declare pole s inicializaci
  myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  int i;

  for(i=0; i<10; i++) {
    cout << "o1[" << i << "]: " << o1[i].getx();
    cout << '\n';
    cout << "o2[" << i << "]: " << o2[i].getx();
    cout << '\n';
  }
```

```
  return 0;
}
```

Na závěr mějme situaci, kdy potřebujeme přetížit konstruktor třídy při alokování dynamického pole této třídy. Jak jsme si již dříve uvedli, dynamické pole nemůže být inicializováno. Proto, když třída obsahuje konstruktor, který přebírá inicializační prvek, musíme přidat přetíženy konstruktor, který inicializační prvek nepřebírá.

Příklad 6.3:

```
#include <iostream.h>

class myclass {
  int x;
public:
  myclass() { x = 0; } // konstruktor bez inicializace
  myclass(int n) { x = n; } // konstruktor s inicializaci
  int getx() { return x; }
  void setx(int n) { x = n; }
};

int main()
{
  myclass *p;
  myclass ob(10);  // inicializace jednoduche promenne

  p = new myclass[10]; // nelze pouzit inicializatory
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }

  int i;

  for(i=0; i<10; i++) p[i] = ob;

  for(i=0; i<10; i++) {
    cout << "p[" << i << "]: " << p[i].getx();
    cout << '\n';
  }

  return 0;
}
```

6.2 Použití standardních argumentů

Časová náročnost: 10 minut

V jazyce C++ existuje prvek, který má vztah k přetěžování funkcí. Tento prvek se jmenuje **standardní argument** a umožňuje nám dát argumentům standardní hodnotu, když při volání funkce není specifikován odpovídající argument.

Pro zadání standardního argumentu do argumentu funkce za argument vložíme **rovnítko** a **hodnotu**, která má být použita jako standardní, když se při volání funkce neobjeví odpovídající argument.

Například následující funkce přiřazuje svým dvěma argumentům standardní hodnotu 1.

Příklad:

```
void fce(int a=1, int b=1);
```

Na první pohled je tato syntaxe podobná inicializaci proměnné. U funkcí je to však trochu jiné. Funkce může být volána třemi způsoby:

1. Bez argumentů
2. S jedním argumentem
3. Se dvěma argumenty

V prvním případě mají **a** i **b** hodnotu **1**.

Příklad:

```
fce();
```

Ve druhém případě má **a** hodnotu **5** a **b** má hodnotu **1**.

Příklad:

```
fce(5);
```

Ve třetím případě má **a** hodnotu **5** a **b** má hodnotu **10**.

Příklad:

```
fce(5, 10);
```

Když vytváříme funkci, která má jeden nebo více standardních argumentů, musí být tyto argumenty specifikovány pouze jednou, a to buď v **prototypu funkce** nebo v **její definici**. Standardní hodnoty nemohou být současně specifikovány v prototypu a v definici funkce.

Standardní argumenty musí být na **pravé straně** každého argumentu, který nemá standardní hodnotu. Když již začneme definovat standardní argumenty, nemůžeme pak specifikovat nějaký argument, který by standardní hodnotu neměl.

Standardní argumenty musí být **konstantami** nebo **globálními proměnnými**. Nemohou to být lokální proměnné nebo jiné argumenty.

Následující program má za úkol vypočítat plocha čtyřúhelníka v případě že jsou zadány dva rozměry a čtverce v případě že je zadán pouze jeden rozměr.

Příklad 6.4:

```
#include <iostream.h>

double rect_area(double length, double width = 0)
{
  if(!width) width = length;
  return length * width;
}

int main()
{
  cout << "Obsah ctyruhelnika daneho rozmery 10 x 5.8 je: ";
  cout << rect_area(10.0, 5.8) << '\n';

  cout << "Obsah ctyruhelnika daneho rozmery 10 x 10 je: ";
  cout << rect_area(10.0) << '\n';

  return 0;
}
```

Pokud nezadáme druhou hodnotu je jako standardní argument použita **0**. Pokud je jako druhá hodnota načtena **0** považujeme daný čtyřúhelník za čtverec a jeho obsah vypočteme jako obsah čtverce.

6.3 Přetěžování a nejednoznačnost

Časová náročnost: 8 minut

Nejednoznačnost způsobená přetěžováním může být do programu vnesena **převodem typů**, přes **argumenty odkazů** nebo přes **standardní argumenty**.

Následující příklad obsahuje chybu nejednoznačnosti. V tomto případě není jasné, zda je volána funkce s argumentem **float** nebo **double**.

Chybný příklad:

```
#include <iostream.h>

float f(float i)
{
  return i / 2.0;
}

double f(double i)
{
  return i / 3.0;
}

int main()
{
  float x = 10.09;
  double y = 10.09;

  cout << f(x); // jednoznacne - float
  cout << f(y); // jednoznace - double

  cout << f(10); // nejednoznacen - float nebo double?

  return 0;
}
```

Nejednoznačnost může vzniknout, když přetížíme funkci, v níž některé přetížené funkce používají standardní argumenty.

Chybný příklad:

```
#include <iostream.h>

int f(int a)
{
  return a*a;
}

int f(int a, int b = 0)
{
  return a*b;
}

int main()
{
```

```
  cout << f(10, 2); // vola funkci f(int, int)
  cout << f(10); // nejednozncne - vola f(int) nebo f(int, int)?

  return 0;
}
```

6.4 Adresy přetěžovaných funkcí

Časová náročnost: 4 minuty

Stejně jako v jazyce C můžeme přiřadit adresu funkce ukazateli a přistupovat pak k funkci přes ukazatel. Adresa funkce je získána vložením jejího jména na pravou stranu přiřazovacího příkazu, ovšem bez závorek či argumentů. Je-li **funkce**() funkcí, pak můžeme získat adresu této funkce takto:

Příklad:

```
adresa = funkce;
```

Když získáme adresu přetěžované funkce, pak způsob deklarace ukazatele určí, od které přetěžované funkce získáváme adresu. V podstatě se deklarace musí shodovat s přetěžovanou funkcí. Funkce, s jejíž deklarací se pak shodne, je funkce, jejíž adresa je použita.

Cvièení

1) Udejte dva důvody, proč můžeme chtít přetížit konstruktor třídy. Řešení

2) Co je v následujícím fragmentu špatně?

```
class samp {
  int a;
public:
  samp(int i) { a = i; }
  // ...
};

// ...

int main()
{
  samp x, y(10);

  // ...
}
```

Řešení

3) Stručně popište standardní argument. Řešení

4) Co je v tomto prototypu funkce špatně?

```
char *f(char *p, int x = 0, char *q);
```

Řešení

5) Co je špatně v tomto prototypu, který používá standardní argument?

```
int f(int pocet, int max = pocet);
```

Řešení

6) Mějme tyto dvě přetěžované funkce. Ukažte jak získat adresu každé z nich.

```
int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}
```

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---


Obsah

7. Dědičnost

Časová náročnost kapitoly: 50 minut

O dědičnosti jsme se částečně zmínili již dříve. Vzhledem k tomu, že dědičnost je jedním ze 3 základních stavebních kamenů jazyka C++ podíváme se na ni v této kapitole podrobněji.

Zmíníme se o řízení přístupu k základní třídě, přístupu protected, dědění více základních tříd, předávání argumentů konstruktorům základních tříd a na závěr si něco řekneme o virtuálních základních třídách.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 až 6 jazyka C++ - umět přetěžovat konstruktory, používat standardní argumenty a získavat adresy přetěžovaných funkcí.

7.1 Řízení přístupu k základní třídě

Časová náročnost: 12 minut

Při dědění se používá tato syntaxe:

```
Class jméno-odvozené-tøídy:access jméno-základní tøídy
{
   // tìlo tøídy
}
```

Kde na místo **access** dosadíme jedno z následujících klíčových slov: **public**, **private** nebo **protected**.

To jakým způsobem ke třídě přistupujeme, určuje, jak jsou elementy základní třídy děděny odvozenou třídou.

Pokud použijeme klíčové slovo:

- **public** pak se všechny obecné členy základní třídy stanou privátními členy odvozené třídy.
- **private** pak se všechny veřejné členy základní třídy stanou privátními členy odvozené třídy. Všechny privátní členy základní třídy zůstávají privátními a nejsou pro odvozenou třídu přístupné.
- **protected** pak platí stejná pravidla jako u klíčového slova **private**, pouze s tím rozdílem, že chráněné členy základní třídy jsou přístupné členům kterékoliv třídy, odvozené z této základní třídy. Vně základní nebo odvozené třídy nejsou chráněné členy přístupné.

Jestliže není specifikátor **access** uveden a je-li odvozená třída **class**, pak je brán jako standard **private** Je-li odvozenou třídou **struct** pak je jako standard brán **public**.

V zájmu srozumitelnosti je **vhodné** specifikátor uvádět.

V následujícím příkladu si ukážeme základní třídu a odvozenou třídu, která ji dědí jako veřejná.

Příklad 7.1:

```
#include <iostream.h>

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// dedi jako verejnou
class derived : public base {
  int y;
public:
  void sety(int n) { y = n; }
  void showy() { cout << y << '\n'; }
};

int main()
{
  derived ob;

  ob.setx(10); // pristup clena zakladni tridy
  ob.sety(20); // pristup clena odvozene tridy

  ob.showx(); // pristup clena zakladni tridy
  ob.showy(); // pristup clena odvozene tridy

  return 0;
}
```

Pokud bychom však v tomto režimu chtěli přistupovat k proměnné **x** došlo by k chybě a to z toho důvodů, že pokud odvozená třída dědí jako **veřejná**, pak nemá přístup k privátním členům základní třídy.

Na dalším programu si ukážeme, že se veřejné členy základní třídy stávají privátními členy odvozené třídy a jsou stále přístupné uvnitř odvozené třídy pokud odvozená třída dědí třídu základní jako **private**.

Příklad 7.2:

```
#include <iostream.h>

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// Dedime zakladni tridu jako private
class derived : private base {
  int y;
public:
  // setx a showx jsou pristupne z odvozene tridy
  void setxy(int n, int m) { setx(n); y = m; }
  void showxy() { showx(); cout << y << '\n'; }
};

int main()
{
  derived ob;

  ob.setxy(10, 20);
```

```
  ob.showxy();

  /* ob.setx(10);
     je chyba, protoze funkce setx() se stavaji privatnimi k derived
     a nejsou z vnejsku pristupne. */

  return 0;
}
```

Jak je v poznámce správně uvedeno, pokud dědíme základní třídu jako **private** pak se veřejné členy základní třídy stanou privátními členy odvozené třídy a nejsou tudíž z vnějšku přístupné.

7.2 Specifikátor protected

Časová náročnost: 6 minut

Specifikátor **protected** se může vyskytovat kdekoliv v deklaraci třídy, typicky se však vyskytuje po deklaraci privátních členů, ale před členy veřejnými. Viz následující syntaxe:

```
class jméno-tøídy {
   // privátní-èleny
protected:   // volitelné
   // chránìné èleny
public:
   // veøejné èleny
};
```

Když je chráněný člen základní třídy děděn odvozenou třídou jako veřejný, stane se chráněným členem odvozené třídy. Jestliže je děděna základní třída jako privátní, chráněný člen základní třídy se stane privátním členem odvozené třídy.

Základní třída může být také děděna odvozenou třídou jako chráněná. V takovém případě se veřejné a chráněné členy základní třídy stanou chráněnými členy odvozené třídy.

Specifikátor **protected** je možné použít také u struktur.

Následující program ukazuje jak jsou chráněné členy děděny jako veřejné.

Příklad 7.3:

```
#include <iostream.h>

class base {
protected:
  int a, b;
public:
  void setab(int n, int m) { a = n; b = m; }
};

class derived : public base {
  int c;
public:
  void setc(int n) { c = n; }

  // tato funkce ma pristup k a i b ze zakladni tridy
  void showabc() {
    cout << a << ' ' << b << ' ' << c << '\n';
  }
```

```
};

int main()
{
  derived ob;

  ob.setab(1, 2);
  ob.setc(3);

  ob.showabc();

  return 0;
}
```

7.3 Konstruktory, destruktory a dědičnost

Časová náročnost: 9 minut

Když má základní i odvozená třída konstruktor a destruktor jsou konstruktory spouštěny při odvozování. Destruktory funkcí jsou poté spouštěny v opačném pořadí. Konstruktor základní třídy je spouštěn před konstruktorem odvozené třídy. Pro destruktory je to naopak.

Pokud chceme předat argumenty konstruktoru odvozené třídy, předáváme mu argumenty obvyklým způsobem popsaným v jedné z předcházejících kapitol. Pokud chceme předat argumenty konstruktoru základní třídy, je zaveden tzv. **řetězec předávaných argumentů**. Nejdříve se veškeré argumenty základní i odvozené třídy předají konstruktoru odvozené třídy. Použitím rozšířeného formátu deklarace konstruktoru odvozené třídy dopředně předáme příslušné argumenty do třídy základní. Viz syntaxe:

```
odvozený-konstruktor(seznam-argumentù):báze(seznam-argumentù) {
   // tìlo-konstruktoru-odvozené-tøídy
};
```

Kde **báze** je název základní třídy. Je přípustné, aby odvozená i základní třída užívaly stejný argument. Je také možné, aby třída ignorovala všechny argumenty a pouze je dopředně předala základní třídě.

Tento program ukazuje, jak přebírá konstruktor odvozené třídy i konstruktor základní třídy argument. V tomto případě jsou argumenty stejné.

Příklad 7.4:

```
#include <iostream.h>

class base {
  int i;
public:
  base(int n) {
    cout << "Constructing base class\n";
    i = n;
  }
  ~base() { cout << "Destructing base class\n"; }
  void showi() { cout << i << '\n'; }
};

class derived : public base {
  int j;
public:
  derived(int n) : base(n) { // predani argumentu do zakladni tridy
    cout << "Constructing derived class\n";
```

```
    j = n;
  }
  ~derived() { cout << "Destructing derived class\n"; }
  void showj() { cout << j << '\n'; }
};

int main()
{
  derived o(10);

  o.showi();
  o.showj();

  return 0;
}
```

Následující příklad ukazuje jak konstruktoru a destruktoru předat různé argumenty. V takovém případě musíme předat konstruktoru odvozené třídy všechny argumenty potřebné pro obě třídy. Potom odvozená třída předá argumenty nutné pro základní třídu dopředně.

Příklad 7.5:

```
#include <iostream.h>

class base {
  int i;
public:
  base(int n) {
    cout << "Constructing base class\n";
    i = n;
  }
  ~base() { cout << "Destructing base class\n"; }
  void showi() { cout << i << '\n'; }
};

class derived : public base {
  int j;
public:
  derived(int n, int m) : base(m) { // predani argumentu zakladni tride
    cout << "Constructing derived class\n";
    j = n;
  }
  ~derived() { cout << "Destructing derived class\n"; }
  void showj() { cout << j << '\n'; }
};

int main()
{
  derived o(10, 20);

  o.showi();
  o.showj();

  return 0;
}
```

7.4 Vícenásobná dědičnost

Časová náročnost: 12 minut

Existují dva způsoby, jak mohou odvozené třídy zdědit více než jednu základní třídu.

1. Odvozená třídy může být použita jako základní třída pro další odvozenou třídu, vytvořením víceúrovňové hierarchie třídy.
2. Odvozená třída může zdědit více než jednu základní třídu.

Ad 1) Původní základní třída je označena za **nepřímou** (indirect) základní třídu druhé odvozené třídy. V tomto případě platí, že jakákoliv třída, bez ohledu na to, jak byla vytvořena může být základní třídou. Dále platí, že v tomto případě, jsou konstruktory funkcí všech tříd nazývány **podle pořadí** odvozování. Také destruktory jsou volány v opačném pořadí.

Ad 2) Dvě nebo více základních tříd je zkombinováno tak, aby pomohly vytvořit odvozenou třídu. V tomto případě používáme následující syntaxi:

```
class jméno-odvozené tøídy:access base1, access base2, ..., access baseN
{
   // tìlo-tøídy
}
```

base1, ..., baseN jsou jména základních tříd. **Access** je přístupový specifikátor, který může být jiný pro každou třídu. Konstruktory jsou v tomto případě spouštěny zleva doprava, neboli v tom pořadí v jakém jsou zapsány. Destruktory jsou spouštěny v opačném pořadí.

Následující program ukazuje odvozenou třídu, která dědí od jiné odvozené třídy.

Příklad 7.6:

```
#include <iostream.h>

class B1 {
  int a;
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// dedeni ze zakladni tridy
class D1 : public B1 {
  int b;
public:
  D1(int x, int y) : B1(y)
  {
    b = x;
  }
  int getb() { return b; }
};

// dedeni z odvozene tridy
class D2 : public D1 {
  int c;
public:
  D2(int x, int y, int z) : D1(y, z)
  {
    c = x;
  }

  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '\n';
  }
```

```
};

int main()
{
  D2 ob(1, 2, 3);

  ob.show();
   cout << ob.geta() << ' ' << ob.getb() << '\n';

  return 0;
}
```

Následující program ukazuje předcházejí program přepracovaný tak, aby odvozená třída dědila přímo ze dvou základních tříd.

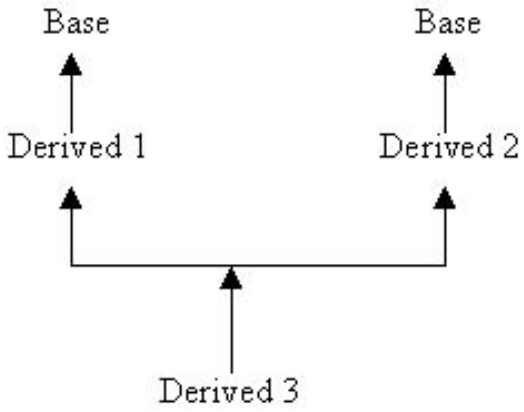
Příklad 7.7:

```
#include <iostream.h>

// prvni zakladni trida
class B1 {
  int a;
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// druha zakladni trida
class B2 {
  int b;
public:
  B2(int x)
  {
    b = x;
  }
  int getb() { return b; }
};

// dedim ze dvou zakladnich trid
class D : public B1, public B2 {
  int c;
public:
  D(int x, int y, int z) : B1(z), B2(y)
  {
    c = x;
  }

  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '\n';
  }
};

int main()
{
  D ob(1, 2, 3);

  ob.show();

  return 0;
}
```

7.5 Virtuální třídy

Časová náročnost: 7 minut

Mějme následující příklad:



V tomto příkladu je základní třída **Base** děděna jak **Derived 1**, tak **Derived 2**. **Derived 3** přímo dědí obě třídy **Derived**. To ale znamená, že **Base** je děděna dvakrát. Jednou přes **Derived 1** a podruhé přes **Derived 2**. Tím vzniká určitá nejednoznačnost.

C++ obsahuje mechanismus, který zajistí, aby k podobné nejednoznačnosti nedocházelo. Tato vlastnost se nazývá **virtuální třída**. Zabrání se tím, aby se dvě nebo více kopii základní třídy objevily ve třídě odvozené, která dědí nepřímo základní třídu. Abychom určili, že třída je chápána jako virtuální uvedeme jako specifikátor klíčové slovo **virtual**.

Výše popsaný příklad si nyní ukážeme.

Příklad 7.8:

```
#include <iostream.h>

class base {
public:
  int i;
};

// dedim zakladni tridu jako virtualni
class derived1 : virtual public base {
public:
  int j;
};

// dedim zakladni tridu jako virtualni
class derived2 : virtual public base {
public:
  int k;
};

// v toto tride je base dedena pouze jednou
class derived3 : public derived1, public derived2 {
public:
  int product() { return i * j * k; }
};

int main()
{
  derived3 ob;
```

```
  ob.i = 10;
  ob.j = 3;
  ob.k = 5;

  cout << "Product is " << ob.product() << '\n';

  return 0;
}
```

Cvièení

1) Mějme definován tento kód.

```
#include <iostream>

class mybase {
  int a, b;
public:
  int c;
  void setab(int i, int j) { a = i; b = j; }
  void getab(int &i, int &j) { i = a; j = b; }
};

class derived1 : public mybase {
// ...
};

class derived2 : private mybase {
// ...
};

int main()
{
  derived1 o1;
  derived2 o2;
  int i, j;

  // ...
}
```

Které z následujících příkazů jsou v rámci funkce **main**() platné?

```
o1getab(i, j);
o2getab(i, j);
o1.c = 10;
o2.c = 10;
```

Řešení

2) Mějme základní třídu **Base** a odvozenou třídu **Derived**. Napište program, který ukáže v jakém pořadí jsou spouštěny konstruktory a destruktory v případě, že **Derived** dědí třídu **Base**. Řešení

3) Co zobrazí následující program?

```
#include <iostream.h>

class A {
public:
  A() { cout << "Constructing A\n"; }
  ~A() { cout << "Destructing A\n"; }
};

class B {
public:
  B() { cout << "Constructing B\n"; }
  ~B() { cout << "Destructing B\n"; }
};

class C : public A, public B {
public:
  C() { cout << "Constructing C\n"; }
  ~C() { cout << "Destructing C\n"; }
};

int main()
{
  C ob;

  return 0;
}
```

Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---


Obsah

8. Vstupně / Výstupní systém

Časová náročnost kapitoly: 1 hodina

V této části si řekneme něco o základních pojmech vstupně / výstupního systému (**I/O**). Zaměříme se na soubory a to jak textové i binární a na závěr se dozvíme něco o přímém přístupu.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 až 7 jazyka C++ - umět používat modifikátory přístupu (public, private, protected), chápat pravidla při dědění konstruktorů a destruktorů. Mít znalosti o vícenásobné dědičnosti a virtuálních třídách.

8.1 Základní pojmy I/O

Časová náročnost: 6 minut

I/O systém jazyka C++ pracuje s **proudy** (stream). Proud dat je logické zařízení, které slouží ke vstupu, nebo výstupu dat. Proudy dat se chovají stejným způsobem, ikdyž se liší fyzické zařízení na které jsou připojeny. Například stejná funkce kterou využijeme pro výpis na obrazovku, může být využita pro výpis do souboru.

Při spuštění programu v jazyce C++ se automaticky otevírají tyto předdefinované proudy:

1. **Cin** - standardní vstup z klávesnice
2. **Cout** - standardní výstup na obrazovku
3. **Cerr** - standardní chybový výstup na obrazovku
4. **Clog** - bufferovaná verze Cerr s výstupem na obrazovku

Standardně jsou proudy dat užívány ke komunikaci s obrazovkou. Jelikož však prostředí C++ podporuje přesměrování, mohou být tyto datové proudy přesměrovány na jiná zařízení.

Jazyk C++ poskytuje pro svůj I/O systém informace v hlavičkovém souboru **iostream.h**. V tomto souboru je definována hierarchie tříd, která podporuje I/O operace. Třídy I/O začínají systémem **šablonových tříd**. Šablonové třídy definují druh tříd bez plné specifikace dat s nimiž mohou operovat. Jakmile jsou šablony tříd definovány mohou se z nich vytvářet standardní případy.

I/O systém jazyka C++ je postaven na dvou rozdílných hierarchiích šablonových tříd:

1. **basic_streambuf** - zajiš•ující základní I/O operace nízké úrovně a vytváří základ celého I/O systému

2. **basic_ios** - tato vysokoúrovňová třída I/O zajiš•uje formátování, kontrolu chyb a stavové informace související s proudy dat.

Třída **basic_ios** slouží jako základní třída pro odvozené třídy **basic_istream**, **basic_ostream** a **basic_iostream**. Tyto třídy slouží k vytváření proudů dat pro I/O.

8.2 Soubory

Časová náročnost: 15 minut

Pro práci se souborem musíme do svého programu vložit hlavičkový soubor **fstream.h**. Zde je definováno několik tříd včetně **ifstream**, **ofstream** a **fstream**. Tyto třídy jsou odvozeny od **istream** a **ostream**.

V jazyce C++ otevřeme soubor připojením k datovému proudu. Existují 3 typy proudů:

1. Vstupní
2. Výstupní
3. Vstupně-Výstupní

Než můžeme soubor otevřít musíme získat proud. Pro vytvoření vstupního proudu deklarujeme objekt typu **ifstream**. Pro vytvoření výstupního proudu deklarujeme objekt typu **ofstream**. Pro vytvoření vstupně-výstupního proudu deklarujeme objekt typu **fstream**.

Příklad:

```
ifstream vstup;
ofstream vystup;
fstream vstupvystup;
```

Výše uvedenou deklarací jsme deklarovali jeden vstupní, jeden výstupní a jeden vstupně-výstupní proud. Poté co jsme vytvořili proud, je jeden ze způsobů jeho přidružení k souboru příkaz **open**(). Viz následující syntaxe:

```
void ifstream::open(const char *jméno-souboru,
                    openmode mód=ios::in);
void ofstream::open(const char *jméno-souboru,
                    openmode mód=ios::out | ios::trunc);
void fstream::open(const char *jméno-souboru,
                    openmode mód=ios::in | ios::out);
```

Jméno-souboru může zahrnovat také specifikaci cesty k souboru. Hodnota **mód** určuje jak je soubor otevřen. Musí to být hodnota typu **openmode** která může být jedna z následujících:

- **ios::app** - všechny výstupy do souboru se připojují na konec
- **ios::ate** - při otevření souboru se hledá jeho konec, ale I/O operace může začít kdekoliv v souboru
- **ios::binary** - soubor otevřen v binárním módu
- **ios::in** - soubor je určen pro vstup

- **ios::out** - soubor je určen pro výstup
- **ios::trunc** - obsah souboru bude oříznut na nulovou délku

Standardně jsou všechny soubory otevírány jako textové. Když vytváříme výstupní soubor pomocí **ofstream** pak je automaticky oříznut na nulovou délku.

Jestliže funkce **open()** selže, pak bude proud v booleovském výrazu vyhodnocen jako **false**. Proto bychom se vždy měli přesvědčit, zda k otevření proudu opravdu došlo!

Také bychom si měli ověřovat, zda došlo k úspěšnému otevření souboru a to pomocí funkce **is_open()**, která je členem **fstream**, **ifstream** a **ofstream**. Má následující syntaxi:

```
bool is_open();
```

Je-li proud připojen k otevřenému souboru pak vrací **true**, jinak vrací **false**.

Pro uzavření souboru použijeme členskou funkci **close()**. Tato funkce nepřebírá žádné argumenty, ani nevrací žádnou hodnotu.

Pomocí funkce **eof()** si ověřujeme, zda byl dosažen konec souboru. Tato funkce má následující syntaxi:

```
bool eof();
```

Pokud je dosažen konec souboru vrací **true**, v opačném případě vrací **false**.

Pokud byl již soubor otevřen můžeme začít číst data, nebo data zapisovat. Stejně jako při práci s obrazovkou budeme používat operátory **<<** a **>>**.

Probranou teorii si ověříme na příkladu. Naším úkolem je vytvořit soubor, zapsat do něj informace a tyto informace opět přečíst.

Příklad 8.1:

```
#include <iostream.h>
#include <fstream.h>

int main()
{
  ofstream fout("test");  // vytvareni souboru pro zapis

  if(!fout) {
    cout << "Nemohu otevrit soubor pro zapis\n";
    return 1;
  }

  fout << "Test\n";
  fout << 100 << ' ' << hex << 100 << endl;

  fout.close();

  ifstream fin("test"); // otevreni souboru pro cteni
```

```
  if(!fin) {
    cout << "Nemohu otevrit soubor pro cteni\n";
    return 1;
  }

  char str[80];
  int i;

  fin >> str >> i;
  cout << str << ' ' << i << endl;

  fin.close();

  return 0;
}
```

8.3 Binární soubory

Časová náročnost: 15 minut

I/O funkce nejnižší úrovně jsou **get()** a **put()**. Funkce **get()** čte byt a funkce **put()** byt zapisuje. Nejpoužívanější syntaxe jsou uvedeny zde:

```
istream &get(char &ch);
ostraam &put(char ch);
```

Funkce **get()** čte jednotlivé znaky z proudu a vkládá je do **ch**. Funkce **put()** zapisuje **ch** do proudu a vrací odkaz na proud.

Pro čtení nebo zápis bloku dat do proudu se použijí funkce **read()** nebo **write()** s následující syntaxí:

```
istream &read(char *buf, streamsize poèet);
ostream &write(const char *buf, streamsize poèet);
```

Funkce **read()** čte daný počet bytů z vyvolaného proudu a vkládá je do vyrovnávací paměti (bufferu) odkazovaného přes **buf**. Funkce **write()** zapisuje daný počet bytů z vyrovnávací paměti odkazované přes **buf** do proudu. Typ **streamsize** je forma integer. Objekt typu **streamsize** je schopen pojmout největší počet bytů předávaných v jakékoliv I/O operaci.

Jestliže bylo dosaženo konce před tím, než byl přečten daný počet znaků, pak prostě funkce **read()** přestane číst a ve vyrovnávací paměti bude tolik znaků, kolik bylo přečteno. K tomu, abychom zjistili kolik znaků bylo přečteno můžeme použít funkci **gcount()**.

```
streamsize gcount();
```

V následujícím příkladu si ukážeme jak vytvářet jednoduchý binární soubor a zapisovat do něj data. Název souboru je zadán jako argument příkazové řády.

Příklad 8.2:

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: 8_2 nazev souboru\n";
    return 1;
  }

  ofstream out(argv[1], ios::out | ios::binary);

  if(!out) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  cout << "Pro ukonceni zadejte $\n";
  do {
    cin.get(ch);
    out.put(ch);
  } while (ch!='$');

  out.close();

  return 0;
}
```

Následující program vypíše obsah souboru.

Příklad 8.3:

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: 8_3 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }
```

```
  in.close();

  return 0;
}
```

Mimo výše uvedených formátů funkcí **get()** a **put()** je možné i tyto funkce přetížit a to následujícím způsobem:

```
istream &get(char *buf, streamsize poèet);
istream &get(char *buf, streamsize poète, char oddìlovaè);
int get();
```

První formát načítá do pole znaky dle ukazatele **buf**, pokud není přetečen buď počtem znaků **počet-1**, nebo se nenarazí na konec řádku, nebo na konec souboru. Jestliže se ve vstupním proudu objeví znak nového řádku **není** přijat. Zůstává v proudu až do další vstupní operace.

Druhý formát čte znaky z pole dle ukazatele **buf**, pokud není přečten počet znaků **počet-1**, nebo se nenarazí na znak určený **oddělovač**em nebo se nenarazí na konec souboru.Jestliže se ve vstupním proudu objeví znak oddělovače **není** přijat. Zůstává v proudu až do další vstupní operace.

Třetí formát vrací vždy následující znak z proudu. Při dosažení konce vrací EOF.

8.4 Přímý přístup

Časová náročnost: 8 minut

V I/O systémech jazyka C++ přístup pomocí funkcí **seekg()** a **seekp()** s následujícími syntaxemi:

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

off_type je typ integer definovaný **ios**, který je schopen pojmout největší platnou hodnotu, kterou může **offset** mít. Argument **seekdir** je jedna z následujících hodnot:

- **ios::beg** - hledání od začátku
- **ios::cur** - hledání od současné pozice
- **ios::end** - hledání od konce

I/O systém jazyka C++ používá dva typy ukazatelů.

1. **get ukazatel** - který ukazuje, kde v souboru dojde k další **výstupní** operaci
2. **put ukazatel** - který ukazuje, kde v souboru dojde k další **vstupní** operaci

Obecně platí, že soubory, k nimž přistupujeme pomocí **seekg()** a **seekp()**, by měly být otevřeny pro **binární** operace.

Existují i přetížené verze funkcí **seekg()** a **seekp()**, které přesunují ukazatel na pozici vrácenou funkcemi **tellg()** a **tellp()**. Funkce **tellg()** a **tellp()** nám vrací aktuální hodnotu pozice daného

ukazatele.

```
istream &seekg(pos_type position);
ostream &seekp(pos_type position);
```

pos_type je integer typ definovaný v **ios**, který je schopen uložit největší hodnotu, která definuje pozici v sobouru.

```
pos_type tellg();
pos_type tellp();
```

Následující program používá seekg() k nastavení ukazatele doprostřed souboru a pro zobrazení jeho obsahu. Jméno souboru a pozice jsou zadávány jako argumenty příkazového řádku.

Příklad 8.4:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Zadejte: 8_4 nazev_souboru pozice\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg);

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

8.5 Ověřování stavu I/O

Časová náročnost: 7 minut

I/O systém jazyka C++ udržuje stavovou informaci o každém výsledku I/O operace. Aktuální stav I/O proudu je popsán v následujících stavech:

- **goodbit** - nevyskytly se chyby
- **eofbit** - konec souboru dosažen
- **failbit** - došlo k ošetřitelné chybě
- **badbit** - došlo k fatální chybě

Existují dva způsoby jak tuto informaci získat:

1. Pomocí funkce **rdstate()**, která má následující syntaxi:

   ```
   iostate rdstate();
   ```

   Vrací aktuální stav chybového příznaku.

2. Použitím některé z následujících funkcí:

   ```
   bool bad(); // vrací true, je-li nastaven badbit
   bool eof(); // vrací true, pokud byl dosažen konec souboru
   bool fail(); // vrací true, je-li nastaven failbit
   bool good(); // vrací true, pokud se nevyskytly chyby
   ```

Když se objeví chyba, můžeme se pokusit odstranit její následky, než bude program pokračovat. K tomuto účelu slouží funkce **clear()** a má následující syntaxi:

```
void clear(iostate pøíznak=ios::goodbit);
```

Pokud je **příznak** nastaven na goodbit, jsou všechny chybové příznaky zrušeny. Jinak nastaví **příznak**y do požadovaného nastavení.

Následující program zobrazuje textový soubor a pro zobrazení chyb používá funkci **good()**. Název souboru je opět čten jako argument příkazové řádky.

Příklad 8.5:

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: 8_5 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1]);
```

```
  if(!in) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    if(!in.good() && !in.eof()) {
      cout << "I/O chyba\n";
      return 1;
    }
    cout << ch;
  }

  in.close();

  return 0;
}
```

Cvièení

1) Napište program, který zkopíruje textový soubor. A• program spočítá kopírované znaky a výsledek zobrazí. Řešení

2) Napište program, který zobrazí textový soubor pozpátku. Řešení

3) Napište program, který čte textový soubor a hlásí, kolikrát se v něm které písmeno vyskytlo. Řešení

4) Napište program, který kopíruje textový soubor a obrací velká písmena na malá a naopak. Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

9. Virtuální funkce

Časová náročnost kapitoly: 30 minut

V této části si řekneme něco o virtuálních funkcích a o polymorfismu. Začátkem si ale zopakujeme něco o ukazatelích odvozených tříd.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 až 8 jazyka C++ - chápat použití souborů v jazyce C++.

9.1 Ukazatele odvozených tříd

Časová náročnost: 3 minuty

Platí: Ukazatel deklarovaný jako ukazatel základní třídy může být použit jako ukazatel na třídu, která je ze základní třídy odvozená. Přesto platí určitá omezení.

- Můžeme se odkazovat pouze na členy odvozeného objektu, které byly zděděny ze základní třídy. Je to proto, že ukazatel základní třídy má údaje pouze o základní třídě. Neví nic o členech, které byly k odvozené třídě přidány.
- Není možné se odkazovat ukazatelem odvozené třídy na třídu základní.

Obecně platí, že aritmetika ukazatele závisí na deklaraci datového typu, na nějž ukazatel ukazuje.

9.2 Virtuální funkce

Časová náročnost: 15 minut

Virtuální funkce je členská funkce, která je deklarována v rámci základní třídy a je **redefinována** odvozenou třídou. Aby byla funkce považována za virtuální je nutné uvést před jejím jménem klíčové slovo **virtual**. Při dědění třídy, která obsahuje virtuální funkci, si odvozená třída virtuální funkci redefinuje dle sebe. Když je virtuální funkce redefinována odvozenou třídou není již klíčové slovo **virtual** nutné.

Virtuální funkce může být volána jako jakákoliv jiná členská funkce. Když ukazatel základní třídy ukazuje na odvozený objekt, který obsahuje virtuální funkci, a tato virtuální funkce je volána prostřednictvím ukazatele, pak se na základě **typu objektu odkazovaného** ukazatelem určí a to za **běhu programu**, která verze virtuální funkce bude spuštěna. Tímto způsobem se dociluje polymorfismu za běhu programu. Následně se třídě, která definuje virtuální funkci říká **polymorfická třída**.

V následujícím příkladu si ukážeme doposud probranou teorii virtuálních funkcí.

Příklad 9.1:

```
#include <iostream.h>

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Verze func() ze tridy base: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Verze func() ze tridy derived1: ";
    cout << i*i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
  void func()
  {
    cout << "Verze func() ze tridy derived2: ";
    cout << i+i << '\n';
  }
```

```
};

int main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // func() tridy base

  p = &d_ob1;
  p->func(); // func() tridy derived1

  p = &d_ob2;
  p->func(); // func() tridy derived2

  return 0;
}
```

Pokud odvozená třída nepřehodnotí virtuální funkci, pak je použita funkce definovaná v základní třídě.

Příklad 9.2:

```
#include <iostream.h>

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Verze func() ze tridy base: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Verze func() ze tridy derived1: ";
    cout << i*i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
};
```

```
int main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // func() tridy base

  p = &d_ob1;
  p->func(); // func() tridy derived1

  p = &d_ob2;
  p->func(); // func() tridy base

  return 0;
}
```

Může se stát, že ikdyž je virtuální funkce definována v základní třídě, tak nemusí provádět smysluplné operace. Někdy prostě jenom definuje základní sadu členských funkcí a proměnných, které si pak odvozené funkce upraví.

Jazyk C++ proto podporuje tzv. **čistě virtuální funkce**. Čistě virtuální funkce nemají definici vztaženou k základní třídě. Je v ní zahrnut pouze prototyp s následující syntaxí:

```
virtual typ jméno-funkce(seznam-argumentù) = 0;
```

Tato syntaxe říká překladači, že vzhledem k základní třídě nemá funkce tělo. Pokud je virtuální funkce definována jako čistá, pak to každou odvozenou třídu **nutí**, aby ji přehodnotila. Jestliže to odvozená třída **neudělá**, ohlásí se **chyba** překladu.

Pokud třída obsahuje nejméně jednu čistě virtuální funkci, pak je na ni odkazováno jako na **abstraktní třídu**. Abstraktní třídy tedy existují proto, aby mohly být děděny.

Následující příkaz nám ukazuje použití čisté virtuální funkce. Program vypočte obsah čtyřúhelníka a trojúhelníka na základě děděné čisté virtuální funkce.

Příklad 9.3:

```
#include <iostream.h>

class area {
  double dim1, dim2; // rozmery
public:
  void setarea(double d1, double d2)
  {
    dim1 = d1;
    dim2 = d2;
  }
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea() = 0; // cista virtualni funkce
};

class rectangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return d1 * d2;
  }
};

class triangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return 0.5 * d1 * d2;
  }
};

int main()
{
  area *p;
  rectangle r;
  triangle t;

  r.setarea(3.3, 4.5);
  t.setarea(4.0, 5.0);

  p = &r;
  cout << "Obsah ctyruhelnika je: " << p->getarea() << '\n';

  p = &t;
```

```
  cout << "Obsah trojuhelnika je: " << p->getarea() << '\n';

  return 0;
}
```

9.3 Polymorfismus

Časová náročnost: 7 minut

Polymorfismus je proces, jímž je běžné rozhraní aplikováno ve dvou nebo více podobných, ale technicky různých, situacích. Polymorfismus je důležitý proto, že může významně zjednodušit složité systémy. Polymorfismus v podstatě dovoluje, aby se logický vztah podobných operací stal běžným a tím pádem se program stane jednodušším na pochopení a údržbu.

Jsou definovány dva pojmy spojované s objektově orientovaným programováním. Jsou to:

- **Časná vazba** - vztahuje se k událostem, které mohou být známy během překladu. Speciálně se vztahuje k takovému volání funkcí, které může být **řešeno během překladu**. Časně vázané položky zahrnují "normální" funkce, přetěžované funkce a nevirtuální členy spřátelených tříd. Když jsou typy těchto funkcí kompilovány během překladu, jsou známy všechny potřebné adresní informace nutné k jejich volání. Hlavní **výhodou** časné vazby je jejich efektivnosti, rychlost komplice a jejich volání. Hlavní **nevýhodou** je nedostatek flexibility.

- **Pozdní vazba** - vztahuje se k událostem, které se vyskytují při běhu programu. Volání pozdně vázané funkce je volání, při němž adresa funkce, která se má volat, není známa dokud se program nespustí. Například virtuální funkce je pozdně vázaný objekt. Hlavní **výhodou** pozdní vazby je flexibilita při běhu programu. Hlavní **nevýhodou** je, že s voláním funkce je spojena větší režie. To způsobuje, že jsou tato volání pomalejší, než volání s časnou vazbou.

Cvièení

1) Vytvořte program, který bude obsahovat základní třídu **num**, která bude obsahovat virtuální funkci **shownum**(). Vytvořte dvě derivované třídy **fhex** a **foct**, které dědí **num**. A• odvozené třídy přehodnotí funkci shownum() a zobrazí číslo v šestnáctkové a osmičkové soustavě. Řešení

2) Proč nemůže být pomocí abstraktní funkce vytvořen objekt? Řešení

3) Je tento fragment správný?

```
class base {
public:
  virtual int f(int a) = 0;
  // ...
};

class derived : public base {
public:
  int f(int a, int b) { return a*b; }
  // ...
};
```

Řešení

4) Jsou virtuální vlastnosti dědičné? Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

---

10. Šablony a obsluha výjimek

Časová náročnost kapitoly: 45 minut

Tato část pojednává o důležitých rysech C++ a to o **šablonách** a **výjimkách**.

Šablony nám pomáhají vytvářet generické třídy a funkce. V generické třídě nebo funkci je typ zpracovávaných dat určen jako argument. To nám dovoluje používat jednu funkci pro různé typy dat, aniž bychom museli tuto funkci upravovat pro specifické typy dat.

Obsluha výjimek nám umožňuje obsloužit chyby, které se mohou objevit během běhu programu.

Potřebné znalosti

K zvládnutí obsahu této kapitoly je nutné mít znalosti z jazyka C a kapitoly 2 až 9 jazyka C++ - umět používat virtuální funkce a virtuální třídy. Rozumět pojmům jako včasná a pozdní vazba.

10.1 Generická funkce

Časová náročnost: 6 minut

Generická funkce definuje **obecnou** sadu operací, které mohou být použity na rozmanité typy dat. Určení typu zpracovávaných dat se generické funkci předává jako argument. Jakmile je generická funkce vytvořena, překladač automaticky generuje správný kód dle typu dat.

Generické funkce jsou vytvářeny pomocí klíčového slova **template** (česky šablona) s následující syntaxí:

```
template <class Typ> návratový-typ jméno-funkce(seznam argumentù)
{
 // tìlo funkce
}
```

Typ je jméno vlastníka prostoru pro typ dat použitý funkcí. Namísto klíčového slova **class** je možné použít i slovo **typename**.

Na příkladu si ukážeme použití generické funkce, jejíž úkolem je záměna dvou proměnných s nimiž je volána.

Příklad 10.1:

```
#include <iostream.h>

// genericka funkce
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;

  cout << "Originalni i, j: " << i << ' ' << j << endl;
  cout << "Originalni x, y: " << x << ' ' << y << endl;

  swapargs(i, j); // zamen ingeger
  swapargs(x, y); // zamen float

  cout << "Zamenene i, j: " << i << ' ' << j << endl;
  cout << "Zamenene x, y: " << x << ' ' << y << endl;

  return 0;
}
```

10.2 Generická třída

Časová náročnost: 8 minut

Generická třída vychází ze stejného principu jako generická funkce. Pokud tedy nadefinujeme nějakou třídu jako generickou, definujeme tím všechny algoritmy touto třídou použité, ale konkrétní typ obsluhovaných dat, bude specifikován až jako argument při vytváření objektu třídy.

Generické třídy jsou použitelné tehdy, pokud chceme nadefinovat zevšeobecnitelnou logiku.

Syntaxe je následující:

```
template <class Typ> class jméno-tøídy
{
 // tìlo tøídy
}
```

Typ je typové jméno vlastníka prostoru, které bude specifikováno, když bude třída konkretizována. Můžeme také v čárkami odděleném seznamu definovat více generických typů.

Pomocí následujícího formátu poté vytváříme specifické případy dané třídy:

```
jméno-tøídy <typ> ob;
```

Typ je tentokrát jméno typu dat, se kterým bude třída pracovat.

Členské funkce generické třídy jsou **automaticky** generické. Nemusí být proto uvozeny klíčovým slovem **template**.

Následující program vytvoří generickou třídu pro spojový seznam. Poté tuto třídu použije pro ukládání znaků.

Příklad 10.2:

```
#include <iostream.h>

template <class data_t> class list {
  data_t data;
  list *next;
public:
  list(data_t d);
  void add(list *node) {node->next = this; next = 0; }
  list *getnext() { return next; }
  data_t getdata() { return data; }
};

template <class data_t> list<data_t>::list(data_t d)
{
  data = d;
  next = 0;
}

int main()
{
  list<char> start('a');
  list<char> *p, *last;
  int i;

  // vytvareni seznamu znaku
  last = &start;
  for(i=1; i<26; i++) {
    p = new list<char> ('a' + i);
    p->add(last);
    last = p;
  }

  // vypis seznamu
```

```
  p = &start;
  while(p) {
    cout << p->getdata();
    p = p->getnext();
  }

  return 0;
}
```

10.3 Výjimky

Časová náročnost: 25 minut

V jazyce C++ existuje mechanismus, který slouží k obsluze chyb. Nazývá se **obsluha výjimek**. Obsluha výjimek je založena na třech klíčových slovech:

- **Try** - zde uvádíme všechny výjimky, které chceme monitorovat
- **Catch** - popisuje jak reagovat na vzniklé výjimky
- **Throw** - slouží k zahazování (odmítnutí) výjimek

Všechny příkazy programu, které chceme monitorovat se zapíší do bloku **try**. Příkaz který odmítá výjimku musí být v bloku **try** umístěn. Výjimka musí být zachycena příkazem **catch**, který následuje bezprostředně za příkazem **try**.

Syntaxe je následující:

```
try {
 // blok try
}
catch (typ1 argument) {
 // blok catch
}
catch (typ2 argument) {
 // blok catch
}
  .
  .
  .
catch (typN argument) {
 // blok catch
}
```

Blok **try** musí obsahovat tu část programu, kterou chceme monitorovat na výskyt chyb. Když je výjimka odmítnuta,je zachycena odpovídajícím příkazem **catch** přidruženým k **try**. Jaký příkaz **catch** to bude konkrétně určuje **typ výjimky** - přesněji se porovná **typ dat** specifikovaný v **catch** s typem ve výjimce a pokud je nalezena shoda, provede se posloupnost příkazů.

Syntaxe příkazu **throw** je následující:

```
throw výjimka;
```

Příkaz **throw** musí být spuštěn buď z bloku **try**, nebo z nějaké funkce, kterou kód v bloku volá. Výjimka je odmítnutá hodnota.

Jestliže odmítneme výjimku, k níž neexistuje odpovídající příkaz **catch** může dojít k abnormálnímu ukončení programu.

Na prvním příkladu si ukážeme jak jazyk C++ přistupuje k výjimkám.

Příklad 10.3:

```
#include <iostream.h>

int main()
{
  cout << "start\n";

  try { // blok try
    cout << "Uvnitr try\n";
    throw 10; // odmitnuti chyby
  }
  catch (int i) { // zachyceni chyby
    cout << "Zachycena chyba! Cislo je: ";
    cout << i << "\n";
  }

  cout << "end";

  return 0;
}
```

Jakmile je výjimka odmítnuta, předá se řízení výrazu **catch** a blok **try je ukončen**. Po provedení příkazu **catch** pokračuje řízení programu příkazem následujícím za **catch**.

Typ výjimky **musí** souhlasit s typem specifikovaným ve výrazu **catch**. Pokud tomu tak není, výjimka nebude zachycena a může dojít k chybnému ukončení programu.

Chybný příklad:

```
#include <iostream.h>

int main()
{
  cout << "start\n";

  try { // blok try
    cout << "Uvnitr try\n";
    throw 10; // odmitnuti chyby
  }
  // nebude fungovat pro vyjimku s datovym typem int
  catch (double i) {
    cout << "Zachycena chyba! Cislo je: ";
    cout << i << "\n";
  }
```

```
  cout << "end";

  return 0;
}
```

Někdy můžeme také chtít, abychom zachytávaly všechny výjimky - ne jen výjimky určitého typu. Použijeme následující formát **catch**:

```
catch(...) {
  // zpracování všech výjimek
}
```

Můžeme omezit typy výjimek, které může funkce odmítat zpět do jejich vyvolávače. Může také řídit, jaký typ výjimek, může funkce sama odmítat. Pro aplikaci určitých omezení musíme do definice funkce přidat klauzuli **throw**.

Syntaxe poté vypadá následovně:

```
návratový-typ jméno-funkce(argumenty) throw(seznam-typù)
{
 // tìlo funkce
}
```

Typy dat oddělené čárkou v poli **seznam-typů** mohou být funkcí odmítnuty. Pokud nechceme aby funkce zachytávala nějaké výjimky uvedeme **prázdný seznam**.

Následující příklad ukazuje zachycení všech výjimek - pouze u výjimky typu int bude vypsáno jiné hlášení.

Příklad 10.4:

```
#include <iostream.h>

void Xhandler(int test)
{
  try{
    if(test==0) throw test;   // odmita int
    if(test==1) throw 'a';    // odmita char
    if(test==2) throw 123.23; // odmita double
  }
  catch(int i) { // zachytava vyjimky typu int
    cout << "Zachycen int " << i << '\n';
  }
  catch(...) { // zachytava vsechny ostatni vyjimky
    cout << "Zachyceno neco jineho nez int\n";
  }
}

int main()
{
  cout << "start\n";

  Xhandler(0);
```

```
  Xhandler(1);
  Xhandler(2);

  cout << "end";

  return 0;
}
```

Následující příklad ukazuje jak lze omezit typy výjimek, které mohou být funkcí odmítnuty.

Příklad 10.5:

```
#include <iostream.h>

// Funkce odmita pouze int, char a double
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test;   // odmita int
  if(test==1) throw 'a';    // odmita char
  if(test==2) throw 123.23; // odmita double
}

int main()
{
  cout << "start\n";

  try{
    Xhandler(0);
  }
  catch(int i) {
    cout << "Zachyceno int\n";
  }
  catch(char c) {
    cout << "Zachyceno char\n";
  }
  catch(double d) {
    cout << "Zachyceno double\n";
  }

  cout << "end";

  return 0;
}
```

Cvièení

1) Vytvořte generickou funkci **min()**, která vrací menší ze dvou argumentů. Předveďte její funkci v programu. Řešení

2) Co je v tomto fragmentu programu špatně?

```
int main()
{
 throw 12.23;
 .
 .
 .
}
```

Řešení

3) Co je v tomto fragmentu programu špatně?

```
try {
 // ....
 throw 'a';
 // ...
}
catch(char *) {
 // ...
}
```

Řešení

4) Co je v tomto fragmentu programu špatně?

```
try {
 // ...
 throw 10;
}
catch(int *p) {
 // ...
}
```

Řešení

5) Ukažte nějaký způsob jak opravit chyby v předchozím fragmentu programu. Řešení

6) Vytvořte generický bubble sort. Řešení

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

Závěr

Časová náročnost: 5 minut

Toto je závěrečná stránka seriálu o jazyku C++. Nyní můžete pokračovat dále na úvodní stránku, kde můžete ukončit své vzdělávání, případně si zopakovat získané znalosti o jazycích C a C++.

Tato elektronická publikace vznikla jako součást diplomové práce s názvem:
Výuková podpora a testy pro C a OOP

Použitá literatura:

- Virius, Miroslav: Pasti a propasti jazyka C++, Praha, Grada Publishing, 1997, 251 s.
- Horstmann, Cay S.: Vyšší škola objektového návrhu v C++, Veletiny, Science, 1997, 370 s.
- Schildt, Herbert: Nauč se sám C++, III. edice, Praha, SoftPress, 2001, 620 s.
- Šešera, Ľubor, Mičovský Aleš: Objektovo-orientovaná tvorba systémov a jazyk C++, Bratislava, Perfekt, 1994, 375 s.
- URL http://www.zive.cz, Živě o počítačích a internetu, (únor 2002)
- URL http://www.builder.cz, Informační server o programování, (únor 2002)

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

// genericka funkce
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;

  cout << "Originalni i, j: " << i << ' ' << j << endl;
  cout << "Originalni x, y: " << x << ' ' << y << endl;

  swapargs(i, j); // zamen ingeger
  swapargs(x, y); // zamen float

  cout << "Zamenene i, j: " << i << ' ' << j << endl;
  cout << "Zamenene x, y: " << x << ' ' << y << endl;

  return 0;
}
```

```
#include <iostream.h>

template <class data_t> class list {
  data_t data;
  list *next;
public:
  list(data_t d);
  void add(list *node) {node->next = this; next = 0; }
  list *getnext() { return next; }
  data_t getdata() { return data; }
};

template <class data_t> list<data_t>::list(data_t d)
{
  data = d;
  next = 0;
}

int main()
{
  list<char> start('a');
  list<char> *p, *last;
  int i;

  // vytvareni seznamu znaku
  last = &start;
  for(i=1; i<26; i++) {
    p = new list<char> ('a' + i);
    p->add(last);
    last = p;
  }

  // vypis seznamu
  p = &start;
  while(p) {
    cout << p->getdata();
    p = p->getnext();
  }

  return 0;
}
```

```
#include <iostream.h>

int main()
{
  cout << "start\n";

  try { // blok try
    cout << "Uvnitr try\n";
    throw 10; // odmitnuti chyby
    cout << "Toto se nevytiskne";
  }
  catch (int i) { // zachyceni chyby
    cout << "Zachycena chyba! Cislo je: ";
    cout << i << "\n";
  }

  cout << "end";

  return 0;
}
```

```
#include <iostream.h>

void Xhandler(int test)
{
  try{
    if(test==0) throw test;   // odmita int
    if(test==1) throw 'a';    // odmita char
    if(test==2) throw 123.23; // odmita double
  }
  catch(int i) { // zachytava vyjimky typu int
    cout << "Zachycen int " << i << '\n';
  }
  catch(...) { // zachytava vsechny ostatni vyjimky
    cout << "Zachyceno neco jineho nez int\n";
  }
}

int main()
{
  cout << "start\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "end";

  return 0;
}
```

```
#include <iostream.h>

// Funkce odmita pouze int, char a double
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test;   // odmita int
  if(test==1) throw 'a';    // odmita char
  if(test==2) throw 123.23; // odmita double
}

int main()
{
  cout << "start\n";

  try{
    Xhandler(0);
  }
  catch(int i) {
    cout << "Zachyceno int\n";
  }
  catch(char c) {
    cout << "Zachyceno char\n";
  }
  catch(double d) {
    cout << "Zachyceno double\n";
  }

  cout << "end";

  return 0;
}
```

Řešení

Zadání: 1) Vytvořte generickou funkci **min()**, která vrací menší ze dvou argumentů. Předveďte její funkci v programu.

Řešení:

```
#include <iostream.h>

template <class X> X min(X a, X b)
{
  if(a<=b)
    return a;
  else
    return b;
}

int main()
{
  cout << min(0.2, 2.0);
  cout << "\n";
  cout << min(3, 4);
  cout << "\n";
  cout << min('c', 'a');
  return 0;
}
```

Zpět

Zadání: 2) Co je v tomto fragmentu programu špatně?

```
int main()
{
 throw 12.23;
 .
 .
 .
}
```

Řešení:

Throw je voláno ještě před průchodem bloku **try**.

Zpět

Zadání: 3) Co je v tomto fragmentu programu špatně?

```
try {
 // ....
 throw 'a';
 // ...
}
catch(char *) {
 // ...
}
```

Řešení:

Znaková vyjímka je odmítnuta, ale příkaz **catch** obslouží pouze znakový ukazatel - neexistuje odpovídající příkaz **catch**, který by obsloužil znakovou vyjímku.

Zpět

Zadání: 4) Co je v tomto fragmentu programu špatně?

```
try {
 // ...
 throw 10;
}
catch(int *p) {
 // ...
}
```

Řešení:

Pro **throw** neexistuje odpovídající **catch**.

Zpět

Zadání: 5) Ukažte nějaký způsob jak opravit chyby v předchozím fragmentu programu.

Řešení:

Je nutné vytvořit **catch** zachytávající int - **catch(int)** a nebo je možné zachytávat všechny vyjímky - **catch(...)**

Zpět

Zadání: 6) Vytvořte generický bubble sort.

Řešení:

```
#include <iostream.h>

template <class X> void bubble(X *data, int size)
{
  register int a, b;
  X t;

  for(a=1; a < size; a++)
    for(b=size-1; b >= a; b--)
      if(data[b-1] > data[b]) {
        t = data[b-1];
        data[b-1] = data[b];
        data[b] = t;
      }
}

int main()
{
  int i[] = {3, 2, 5, 6, 1, 8, 9, 3, 6, 9};
  double d[] = {1.2, 5.5, 2.2, 3.3};
  int j;

  bubble(i, 10); // setridene int
  bubble(d, 4);  // setridene double

  for(j=0; j<10; j++) cout << i[j] << ' ';
  cout << endl;

  for(j=0; j<4; j++) cout << d[j] << ' ';
  cout << endl;

  return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

template <class X> X min(X a, X b)
{
  if(a<=b)
    return a;
  else
    return b;
}

int main()
{
  cout << min(0.2, 2.0);
  cout << "\n";
  cout << min(3, 4);
  cout << "\n";
  cout << min('c', 'a');
  return 0;
}
```

```
#include <iostream.h>

template <class X> void bubble(X *data, int size)
{
  register int a, b;
  X t;

  for(a=1; a < size; a++)
    for(b=size-1; b >= a; b--)
      if(data[b-1] > data[b]) {
        t = data[b-1];
        data[b-1] = data[b];
        data[b] = t;
      }
}

int main()
{
  int i[] = {3, 2, 5, 6, 1, 8, 9, 3, 6, 9};
  double d[] = {1.2, 5.5, 2.2, 3.3};
  int j;

  bubble(i, 10); // setridene int
  bubble(d, 4);  // setridene double

  for(j=0; j<10; j++) cout << i[j] << ' ';
  cout << endl;

  for(j=0; j<4; j++) cout << d[j] << ' ';
  cout << endl;

  return 0;
}
```

```
#include <iostream.h>

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Verze func() ze tridy base: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Verze func() ze tridy derived1: ";
    cout << i*i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
  void func()
  {
    cout << "Verze func() ze tridy derived2: ";
    cout << i+i << '\n';
  }
};

int main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // func() tridy base

  p = &d_ob1;
  p->func(); // func() tridy derived1

  p = &d_ob2;
  p->func(); // func() tridy derived2

  return 0;
}
```

```
#include <iostream.h>

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
  {
    cout << "Verze func() ze tridy base: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) {}
  void func()
  {
    cout << "Verze func() ze tridy derived1: ";
    cout << i*i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) {}
};

int main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // func() tridy base

  p = &d_ob1;
  p->func(); // func() tridy derived1

  p = &d_ob2;
  p->func(); // func() tridy base

  return 0;
}
```

```
#include <iostream.h>

class area {
  double dim1, dim2; // rozmery
public:
  void setarea(double d1, double d2)
  {
    dim1 = d1;
    dim2 = d2;
  }
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea() = 0; // cista virtualni funkce
};

class rectangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return d1 * d2;
  }
};

class triangle : public area {
public:
  double getarea()
  {
    double d1, d2;

    getdim(d1, d2);
    return 0.5 * d1 * d2;
  }
};

int main()
{
  area *p;
  rectangle r;
  triangle t;

  r.setarea(3.3, 4.5);
  t.setarea(4.0, 5.0);

  p = &r;
  cout << "Obsah ctyruhelnika je: " << p->getarea() << '\n';

  p = &t;
  cout << "Obsah trojuhelnika je: " << p->getarea() << '\n';

  return 0;
}
```

Řešení

Zadání: 1) Vytvořte program, který bude obsahovat základní třídu **num**, která bude obsahovat virtuální funkci **shownum()**. Vytvořte dvě derivované třídy **fhex** a **foct**, které dědí **num**. A• odvozené třídy přehodnotí funkci shownum() a zobrazí číslo v šestnáctkové a osmičkové soustavě.

Řešení:

```
#include <iostream.h>

class num {
public:
  int i;
  num(int x) { i = x; }
  virtual void shownum() { cout << i << '\n'; }
};

class fhex : public num {
public:
  fhex(int n) : num(n) {}
  void shownum() { cout << hex << i << '\n'; }
};

class ofct : public num {
public:
  foct(int n) : num(n) {}
  void shownum() { cout << oct << i << '\n'; }
};

int main()
{
  oct o(10);
  hex h(20);

  o.shownum();
  h.shownum();

  return 0;
}
```

Zpět

Zadání: 2) Proč nemůže být pomocí abstraktní funkce vytvořen objekt?

Řešení:

Abstraktní třída obsahuje alespoň jednu čistě virtuální funkci. To znamená, že

vzhledem k této třídě nemá funkce žádné tělo. **Neexistuje** žádný způsob, jak by mohl být vytvořen objekt, poněvadž definice třídy není úplná.

Zpět

Zadání: 3) Je tento fragment správný?

```
class base {
public:
  virtual int f(int a) = 0;
  // ...
};

class derived : public base {
public:
  int f(int a, int b) { return a*b; }
  // ...
};
```

Řešení:

Fragment **není** správný, protože definice virtuální funkce musí mít stejný návratový typ a počet argumentů jako původní funkce. V tomto případě se liší počet argumentů.

Zpět

Zadání: 4) Jsou virtuální vlastnosti dědičné?

Řešení:

Ano

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

class num {
public:
  int i;
  num(int x) { i = x; }
  virtual void shownum() { cout << i << '\n'; }
};

class fhex : public num {
public:
  fhex(int n) : num(n) {}
  void shownum() { cout << hex << i << '\n'; }
};

class foct : public num {
public:
  foct(int n) : num(n) {}
  void shownum() { cout << oct << i << '\n'; }
};

int main()
{
  foct o(10);
  fhex h(20);

  o.shownum();
  h.shownum();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>

int main()
{
  ofstream fout("test");  // vytvareni souboru pro zapis

  if(!fout) {
    cout << "Nemohu otevrit soubor pro zapis\n";
    return 1;
  }

  fout << "Test\n";
  fout << 100 << ' ' << hex << 100 << endl;

  fout.close();

  ifstream fin("test"); // otevreni souboru pro cteni

  if(!fin) {
    cout << "Nemohu otevrit soubor pro cteni\n";
    return 1;
  }

  char str[80];
  int i;

  fin >> str >> i;
  cout << str << ' ' << i << endl;

  fin.close();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: 8_2 nazev souboru\n";
    return 1;
  }

  ofstream out(argv[1], ios::out | ios::binary);

  if(!out) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  cout << "Pro ukonceni zadejte $\n";
  do {
        cin.get(ch);
    out.put(ch);
  } while (ch!='$');

  out.close();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: 8_3 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Zadejte: 8_4 nazev_souboru pozice\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg);

  while(!in.eof()) {
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: 8_5 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Nelze otevrit soubor\n";
    return 1;
  }

  while(!in.eof()) {
    in.get(ch);
    if(!in.good() && !in.eof()) {
      cout << "I/O chyba\n";
      return 1;
    }
    cout << ch;
  }

  in.close();

  return 0;
}
```

Řešení

Zadání: 1) Napište program, který zkopíruje textový soubor. A• program spočítá kopírované znaky a výsledek zobrazí.

Řešení:

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Zadejte: cvi8_1 vstup vystup\n";
    return 1;
  }

  ifstream fin(argv[1]); // otevirani vstupniho souboru
  ofstream fout(argv[2]);  // vytvareni vystupniho souboru

  if(!fin) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  if(!fout) {
    cout << "Nelze vytvorit vystupni soubor\n";
    return 1;
  }

  char ch;
  unsigned count=0;

  fin.unsetf(ios::skipws);
  while(!fin.eof()) {
    fin >> ch;
    if(!fin.eof()) {
      fout << ch;
      count++;
    }
  }

  cout << "Pocet zkopirovanych znaku: " << count << '\n';

  fin.close();
  fout.close();

  return 0;
}
```

Zpět

Zadání: 2) Napište program, který zobrazí textový soubor pozpátku.

Řešení:

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Zadejte: cvi8_2 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  char ch;
  long i;

  in.seekg(0, ios::end);
  i = (long) in.tellg();
  i -= 2;

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

Zpět

Zadání: 3) Napište program, který čte textový soubor a hlásí, kolikrát se v něm které písmeno vyskytlo.

Řešení:

```
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

int alpha[26];

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: cvi8_3 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  // init alpha[]
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(isalpha(ch)) { // pokud bylo nalezeno pismono
      ch = toupper(ch); // normalizace
      alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, ...
    }
  };

  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] << '\n';
  }

  in.close();

  return 0;
}
```

Zpět

Zadání: 4) Napište program, který kopíruje textový soubor a obrací velká písmena na malá a naopak.

Řešení:

```
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Zadejte: cvi8_4 zdroj cil\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  ofstream out(argv[2]);

  if(!out) {
    cout << "Nelze otevrit vystupni soubor\n";
    return 1;
  }

  while(!in.eof()) {
    ch = in.get();
    if(!in.eof()) {
      if(islower(ch)) ch = toupper(ch);
      else ch = tolower(ch);
      out.put(ch);
    }
  };

  in.close();
  out.close();

  return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Zadejte: cvi8_1 vstup vystup\n";
    return 1;
  }

  ifstream fin(argv[1]); // otevirani vstupniho souboru
  ofstream fout(argv[2]);  // vytvareni vystupniho souboru

  if(!fin) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  if(!fout) {
    cout << "Nelze vytvorit vystupni soubor\n";
    return 1;
  }

  char ch;
  unsigned count=0;

  fin.unsetf(ios::skipws);
  while(!fin.eof()) {
    fin >> ch;
    if(!fin.eof()) {
      fout << ch;
      count++;
    }
  }

  cout << "Pocet zkopirovanych znaku: " << count << '\n';

  fin.close();
  fout.close();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Zadejte: cvi8_2 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  char ch;
  long i;

  in.seekg(0, ios::end);
  i = (long) in.tellg();
  i -= 2;

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

int alpha[26];

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Zadejte: cvi8_3 nazev souboru\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  // init alpha[]
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(isalpha(ch)) { // pokud bylo nalezeno pismono
      ch = toupper(ch); // normalizace
      alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, ...
    }
  };

  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] << '\n';
  }

  in.close();

  return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Zadejte: cvi8_4 zdroj cil\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Nelze otevrit vstupni soubor\n";
    return 1;
  }

  ofstream out(argv[2]);

  if(!out) {
    cout << "Nelze otevrit vystupni soubor\n";
    return 1;
  }

  while(!in.eof()) {
    ch = in.get();
    if(!in.eof()) {
      if(islower(ch)) ch = toupper(ch);
      else ch = tolower(ch);
      out.put(ch);
    }
  };

  in.close();
  out.close();

  return 0;
}
```

```
#include <iostream.h>

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// dedi jako verejnou
class derived : public base {
  int y;
public:
  void sety(int n) { y = n; }
  void showy() { cout << y << '\n'; }
};

int main()
{
  derived ob;

  ob.setx(10); // pristup clena zakladni tridy
  ob.sety(20); // pristup clena odvozene tridy

  ob.showx(); // pristup clena zakladni tridy
  ob.showy(); // pristup clena odvozene tridy

  return 0;
}
```

```
#include <iostream.h>

class base {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << '\n'; }
};

// Dedime zakladni tridu jako private
class derived : private base {
  int y;
public:
  // setx a showx jsou pristupne z odvozene tridy
  void setxy(int n, int m) { setx(n); y = m; }
  void showxy() { showx(); cout << y << '\n'; }
};

int main()
{
  derived ob;

  ob.setxy(10, 20);

  ob.showxy();

  /* ob.setx(10);
     je chyba, protoze funkce setx() se stavaji privatnimi k derived
     a nejsou z vnejsku pristupne. */

  return 0;
}
```

```
#include <iostream.h>

class base {
protected:
  int a, b;
public:
  void setab(int n, int m) { a = n; b = m; }
};

class derived : public base {
  int c;
public:
  void setc(int n) { c = n; }

  // tato funkce ma pristup k a i b ze zakladni tridy
  void showabc() {
    cout << a << ' ' << b << ' ' << c << '\n';
  }
};

int main()
{
  derived ob;

  ob.setab(1, 2);
  ob.setc(3);

  ob.showabc();

  return 0;
}
```

```
#include <iostream.h>

class base {
  int i;
public:
  base(int n) {
    cout << "Constructing base class\n";
    i = n;
  }
  ~base() { cout << "Destructing base class\n"; }
  void showi() { cout << i << '\n'; }
};

class derived : public base {
  int j;
public:
  derived(int n) : base(n) { // predani argumentu do zakladni tridy
    cout << "Constructing derived class\n";
    j = n;
  }
  ~derived() { cout << "Destructing derived class\n"; }
  void showj() { cout << j << '\n'; }
};

int main()
{
  derived o(10);

  o.showi();
  o.showj();

  return 0;
}
```

```
#include <iostream.h>

class base {
  int i;
public:
  base(int n) {
    cout << "Constructing base class\n";
    i = n;
  }
  ~base() { cout << "Destructing base class\n"; }
  void showi() { cout << i << '\n'; }
};

class derived : public base {
  int j;
public:
  derived(int n, int m) : base(m) { // predani argumentu zakladni tride
    cout << "Constructing derived class\n";
    j = n;
  }
  ~derived() { cout << "Destructing derived class\n"; }
  void showj() { cout << j << '\n'; }
};

int main()
{
  derived o(10, 20);

  o.showi();
  o.showj();

  return 0;
}
```

```
#include <iostream.h>

class B1 {
  int a;
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// dedeni ze zakladni tridy
class D1 : public B1 {
  int b;
public:
  D1(int x, int y) : B1(y)
  {
    b = x;
  }
  int getb() { return b; }
};

// dedeni z odvozene tridy
class D2 : public D1 {
  int c;
public:
  D2(int x, int y, int z) : D1(y, z)
  {
    c = x;
  }

  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '\n';
  }
};

int main()
{
  D2 ob(1, 2, 3);

  ob.show();
   cout << ob.geta() << ' ' << ob.getb() << '\n';

  return 0;
}
```

```
#include <iostream.h>

// prvni zakladni trida
class B1 {
  int a;
public:
  B1(int x) { a = x; }
  int geta() { return a; }
};

// druha zakladni trida
class B2 {
  int b;
public:
  B2(int x)
  {
    b = x;
  }
  int getb() { return b; }
};

// dedim ze dvou zakladnich trid
class D : public B1, public B2 {
  int c;
public:
  D(int x, int y, int z) : B1(z), B2(y)
  {
    c = x;
  }

  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '\n';
  }
};

int main()
{
  D ob(1, 2, 3);

  ob.show();

  return 0;
}
```

```
#include <iostream.h>

class base {
public:
  int i;
};

// dedim zakladni tridu jako virtualni
class derived1 : virtual public base {
public:
  int j;
};

// dedim zakladni tridu jako virtualni
class derived2 : virtual public base {
public:
  int k;
};

// v toto tride je base dedena pouze jednou
class derived3 : public derived1, public derived2 {
public:
  int product() { return i * j * k; }
};

int main()
{
  derived3 ob;

  ob.i = 10;
  ob.j = 3;
  ob.k = 5;

  cout << "Product is " << ob.product() << '\n';

  return 0;
}
```

Zadání: 1) Mějme definován tento kód.

```
#include <iostream>

class mybase {
  int a, b;
public:
  int c;
  void setab(int i, int j) { a = i; b = j; }
  void getab(int &i, int &j) { i = a; j = b; }
};

class derived1 : public mybase {
// ...
};

class derived2 : private mybase {
// ...
};

int main()
{
  derived1 o1;
  derived2 o2;
  int i, j;

  // ...
}
```

Které z následujících příkazů jsou v rámci funkce **main()** platné?

```
o1getab(i, j);
o2getab(i, j);
o1.c = 10;
o2.c = 10;
```

Řešení:

Platné příkazy jsou tyto:

```
o1getab(i, j);
o1.c = 10;
```

Zpět

Zadání: 2) Mějme základní třídu **Base** a odvozenou třídu **Derived**. Napište program,

který ukáže v jakém pořadí jsou spouštěny konstruktory a destruktory v případě, že **Derived** dědí třídu **Base**.

Řešení:

```
#include <iostream.h>

class base {
public:
  base() { cout << "Konstruktor tridy Base\n"; }
  ~base() { cout << "Destruktor tridy Base\n"; }
};

class derived : public base {
public:
  derived() { cout << "Konstruktor tridy Derived\n"; }
  ~derived() { cout << "Destruktor tridy Derived\n"; }
};

int main()
{
  derived o;

  return 0;
}
```

Zpět

Zadání: 3) Co zobrazí následující program?

```
#include <iostream.h>

class A {
public:
  A() { cout << "Constructing A\n"; }
  ~A() { cout << "Destructing A\n"; }
};

class B {
public:
  B() { cout << "Constructing B\n"; }
  ~B() { cout << "Destructing B\n"; }
};

class C : public A, public B {
public:
  C() { cout << "Constructing C\n"; }
  ~C() { cout << "Destructing C\n"; }
};

int main()
{
```

```
  C ob;

  return 0;
}
```

Řešení:

```
Constructing A
Constructing B
Constructing C
Destructing C
Destructing B
Destructing A
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

class base {
public:
  base() { cout << "Konstruktor tridy Base\n"; }
  ~base() { cout << "Destruktor tridy Base\n"; }
};

class derived : public base {
public:
  derived() { cout << "Konstruktor tridy Derived\n"; }
  ~derived() { cout << "Destruktor tridy Derived\n"; }
};

int main()
{
  derived o;

  return 0;
}
```

```
#include <iostream.h>

class A {
public:
  A() { cout << "Constructing A\n"; }
  ~A() { cout << "Destructing A\n"; }
};

class B {
public:
  B() { cout << "Constructing B\n"; }
  ~B() { cout << "Destructing B\n"; }
};

class C : public A, public B {
public:
  C() { cout << "Constructing C\n"; }
  ~C() { cout << "Destructing C\n"; }
};

int main()
{
  C ob;

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int x;
public:
  myclass() { x = 0; } // konstruktor bez inicilizace
  myclass(int n) { x = n; } // inicializovany konstruktor
  int getx() { return x; }
};

int main()
{
  myclass o1(10); // deklarace s pocatecni hodnotou
  myclass o2; // deklarace bez pocatecni hodnoty

  cout << "o1: " << o1.getx() << '\n';
  cout << "o2: " << o2.getx() << '\n';

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int x;
public:
  myclass() { x = 0; } // konstruktor bez inicializace
  myclass(int n) { x = n; } // inicializovany konstruktor
  int getx() { return x; }
};

int main()
{
  // deklarace pole bez inicializace
  myclass o1[10];
  // declare pole s inicializaci
  myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  int i;

  for(i=0; i<10; i++) {
    cout << "o1[" << i << "]: " << o1[i].getx();
    cout << '\n';
    cout << "o2[" << i << "]: " << o2[i].getx();
    cout << '\n';
  }

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int x;
public:
  myclass() { x = 0; } // konstruktor bez inicializace
  myclass(int n) { x = n; } // konstruktor s inicializaci
  int getx() { return x; }
  void setx(int n) { x = n; }
};

int main()
{
  myclass *p;
  myclass ob(10);  // inicializace jednoduche promenne

  p = new myclass[10]; // nelze pouzit inicializatory
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }

  int i;

  for(i=0; i<10; i++) p[i] = ob;

  for(i=0; i<10; i++) {
    cout << "p[" << i << "]: " << p[i].getx();
    cout << '\n';
  }

  return 0;
}
```

```
#include <iostream.h>

double rect_area(double length, double width = 0)
{
  if(!width) width = length;
  return length * width;
}

int main()
{
  cout  << "Obsah ctyruhelnika daneho rozmery 10 x 5.8 je: ";
  cout << rect_area(10.0, 5.8) << '\n';

  cout  << "Obsah ctyruhelnika daneho rozmery 10 x 10 je: ";
  cout << rect_area(10.0) << '\n';

  return 0;
}
```

Zadání: 1) Udejte dva důvody, proč můžeme chtít přetížit konstruktor třídy.

Řešení:

Například proto, abychom mohli v konstruktoru v závislosti na konkrétním volání provést danou inicializaci, nebo také proto, aby se v programu mohly objevit jak samostatné objekty tak i pole objektů.

Zpět

Zadání: 2) Co je v následujícím fragmentu špatně?

```
class samp {
  int a;
public:
  samp(int i) { a = i; }
  // ...
}:

// ...

int main()
{
  samp x, y(10);

  // ...
}
```

Řešení:

Třída **samp** definuje pouze jediný konstruktor a tento konstruktor požaduje hodnotu pro inicializaci. Je nesprávné deklarovat objekt typu **samp** bez něj.

Zpět

Zadání: 3) Stručně popište standardní argument.

Řešení:

Standardní argument je hodnota, která se přidělí argumentu funkce, když se při volání funkce neobjeví požadovaný počet argumentů.

Zpět

Zadání: 4) Co je v tomto prototypu funkce špatně?

```
char *f(char *p, int x = 0, char *q);
```

Řešení:

Všechny programy, které přebírají standardní argumenty se musí objevit napravo od toho, který nepřebírá. Jakmile totiž začneme předávat standardní argumenty, **všechny následující** argumenty musí být také standardní. **q** zde není standardní.

Zpět

Zadání: 5) Co je špatně v tomto prototypu, který používá standardní argument?

```
int f(int pocet, int max = pocet);
```

Řešení:

Standardní argument nemůže být další argument nebo lokální proměnná.

Zpět

Zadání: 6) Mějme tyto dvě přetěžované funkce. Ukažte jak získat adresu každé z nich.

```
int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}
```

Řešení:

```
#include <iostream.h>

int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}

int main()
{
  int (*p1)(int, int);
  float (*p2)(float, float);

  p1 = dif; // adresa dif(int, int)
  p2 = dif; // adresa dif(float, float);

  cout << p1(10, 5) << ' ';
  cout << p2(10.5, 8.9) << '\n';

  return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

int dif(int a, int b)
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}

int main()
{
  int (*p1)(int, int);
  float (*p2)(float, float);

  p1 = dif; // adresa dif(int, int)
  p2 = dif; // adresa dif(float, float);

  cout << p1(10, 5) << ' ';
  cout << p2(10.5, 8.9) << '\n';

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int a;
public:
  void set_a(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4];
  int i;

  for(i=0; i<4; i++) ob[i].set_a(i);

  for(i=0; i<4; i++) cout << ob[i].get_a( );

  cout << "\n";

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4] = { -1, -2, -3, -4 };
  int i;

  for(i=0; i<4; i++) cout << ob[i].get_a() << ' ';

  cout << "\n";

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
    5, 6,
    7, 8
  };
  int i;


  for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][1].get_a() << "\n";
  }

  cout << "\n";

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

int main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
    samp(7, 8)
  };
  int i;

  samp *p;
  p = ob; // pocatecni adresa pole

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    cout << p->get_b() << "\n";
    p++;  // nasledujici objekt
  }

  cout << "\n";

  return 0;
}
```

```
#include <iostream.h>
#include <string.h>

class inventory {
  char item[20];
  double cost;
  int on_hand;
public:
  inventory(char *i, double c, int o)
  {
     strcpy(this->item, i); // pristup clenu
     this->cost = c;  // pres ukazatel this
     this->on_hand = o;
   }
   void show();
};

void inventory::show()
{
  cout << this->item; // pro pristup k clenum
  cout << ": $" << this->cost;
  cout << "  pocet kusu: " << this->on_hand << "\n";
}

int main()
{
  inventory ob("cena", 4.95, 4);

  ob.show();

  return 0;
}
```

```
#include <iostream.h>

int main()
{
  int *p;

  p = new int; // alokace mista pro int
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }

  *p = 1000;

  cout << "Hodnota integer: " << *p << "\n";

  delete p;  // uvolneni pameti

  return 0;
}
```

```
#include <iostream.h>

int main()
{
  int *p;

  p = new int [5]; // alokujeme misto pro pole 5 integeru
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }

  int i;

  for(i=0; i<5; i++) p[i] = i;

  for(i=0; i<5; i++) {
    cout << "Hodnota integer v p[" << i << "]: ";
    cout << p[i] << "\n";
  }

  delete [] p;  // uvolneni pameti

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int i, j;
public:
  void set_ij(int a, int b) { i=a; j=b; }
  ~samp() { cout << "Destruktor...\n"; }
  int get_product() { return i*j; }
};

int main()
{
  samp *p;
  int i;

  p = new samp [10]; // alokace objektu pole
  if(!p) {
    cout << "Chyba pri alokaci\n";
    return 1;
  }

  for(i=0; i<10; i++)
    p[i].set_ij(i, i);

  for(i=0; i<10; i++) {
    cout << "Produkt [" << i << "] is: ";
    cout << p[i].get_product() << "\n";
  }

  delete [] p;
  return 0;
}
```

```
#include <iostream.h>

void swapargs(int &x, int &y);

int main()
{
  int i, j;

  i = 10;
  j = 19;

  cout << "Pred zamenou: ";
  cout << "i: " << i << ", ";
  cout << "j: " << j << "\n";

  swapargs(i, j);

  cout << "Po zamene: ";
  cout << "i: " << i << ", ";
  cout << "j: " << j << "\n";

  return 0;
}

void swapargs(int &x, int &y)
{
  int t;

  t = x;
  x = y;
  y = t;
}
```

```
#include <iostream.h>

class myclass {
  int who;
public:
  myclass(int n) {
    who = n;
    cout << "Konstruktor " << who << "\n";
  }
  ~myclass() { cout << "Destruktor " << who << "\n"; }
  int id() { return who; }
};

// predavani odkazem
void f(myclass &o)
{
  //  . operator je stale pouzivan
  cout << "Prijato " << o.id() << "\n";
}

int main()
{
  myclass  x(1);

  f(x);

  return 0;
}
```

```
#include <iostream.h>

int &f(); // return ukazatel
int x;

int main()
{
  f() = 100; // prirazeni 100 odkazu vracenemu f()

  cout << x << "\n";

  return 0;
}

int &f()
{
  return x;  // vraci odkaz na x
}
```

```
#include <iostream.h>

int main()
{
  int x;
  int &ref = x; // nezavisly odkaz

  x = 10;    // tyto dva prikazy
  ref = 10;  // jsou funkcne shodne

  ref = 100;
  // 100 se vytiskne 2x
  cout << x << ' ' << ref << "\n";

  return 0;
}
```

Řešení

Zadání: 1) S využitím deklarované třídy vytvořte desetiprvkové pole a inicializujte prvek **ch** hodnotami **A** až **J**. Následně toto pole vytiskněte.

```
#include <iostream.h>

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};
```

Řešení:

```
#include <iostream.h>

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};

int main()
{
  letters ob[10] = { 'a', 'b', 'c','d', 'e', 'f',
                     'g', 'h', 'i', 'j' };

  int i;

  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "\n";

  return 0;
}
```

Zpět

Zadání: 2) Přepište v následujícím programu všechny příslušné odkazy na explicitní ukazatel **this**.

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int n, int m) { a = n; b = m; }
  int add() { return a+b; }
  void show();
};

void myclass::show()
{
  int t;

  t = add();
  cout << t << "\n";
}

int main()
{
  myclass ob(10, 14);

  ob.show();

  return 0;
}
```

Řešení:

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int n, int m) { this->a = n; this->b = m; }
  int add() { return this->a + this->b; }
  void show();
};

void myclass::show()
{
  int t;

  t = this->add();
  cout << t << "\n";
}

int main()
{
  myclass ob(10, 14);
```

```
  ob.show();

  return 0;
}
```

Zpět

Zadání: 3) Napište program, který bude při použití **new** dynamicky alokovat **float**, **double** a **char**. Zadejte do těchto dynamických proměnných hodnoty a zobrazte je. Nakonec uvolněte celou dynamicky přidělenou pamě• pomocí **delete**.

Řešení:

```
#include <iostream.h>

int main()
{
  float *f;
  long *l;
  char *c;

  f = new float;
  l = new long;
  c = new char;

  if(!f || !l || !c) {
    cout << "Chyba pri alokaci.";
    return 1;
  }

  *f = 10.102;
  *l = 100000;
  *c = 'A';

  cout << *f << ' ' << *l << ' ' << *c;
  cout << '\n';

  delete f;
  delete l;
  delete c;

  return 0;
}
```

Zpět

Zadání: 4) Převeďte následující kód na jeho ekvivalent, který používá **new**.

```
char *p;

p = (char *) malloc(100);
// ...
strcpy(p, "Test");
```

Řešení:

```
char *p;

p = new char [100];
// ...
strcpy(p, "Test");
```

Zpět

Zadání: 5) Co je v následujícím programu chybné?

```
#include <iostream.h>

void triple(double &num);

int main()
{
  double d = 7.0;

  triple(&d);

  cout << d;

  return 0;
}

void triple(double &num)
{
  num = 3 * num;
}
```

Řešení:

Když je voláno **tripple()**, získává se explicitně adresa **d** operátorem **&**, což není povolené. Když je použit argument odkazu, není před argument vloženo **&**.

Zpět

Zadání: 6) Co je to ukazatel **this**?

Řešení:

Ukazatel **this** je ukazatel, který je automaticky předáván členské funkci a který ukazuje na objekt, který generoval volání.

Zpět

Zadání: 7) Co je odkaz? Jaká je výhoda použití argumentu odkazu?

Řešení:

Odkaz je v podstatě implicitní konstantní ukazatel, který má odlišné jméno než jiná proměnná nebo argument. Výhodou použití odkazu je, že nevzniká kopie argumentu.

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};

int main()
{
  letters ob[10] = { 'a', 'b', 'c','d', 'e', 'f',
                     'g', 'h', 'i', 'j' };

  int i;

  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "\n";

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int n, int m) { this->a = n; this->b = m; }
  int add() { return this->a + this->b; }
  void show();
};

void myclass::show()
{
  int t;

  t = this->add();
  cout << t << "\n";
}

int main()
{
  myclass ob(10, 14);

  ob.show();

  return 0;
}
```

```
#include <iostream.h>

int main()
{
  float *f;
  long *l;
  char *c;

  f = new float;
  l = new long;
  c = new char;

  if(!f || !l || !c) {
    cout << "Chyba pri alokaci.";
    return 1;
  }

  *f = 10.102;
  *l = 100000;
  *c = 'A';

  cout << *f << ' ' << *l << ' ' << *c;
  cout << '\n';

  delete f;
  delete l;
  delete c;

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int n, int m) { a = n; b = m; }
  int add() { return a+b; }
  void show();
};

void myclass::show()
{
  int t;

  t = add();
  cout << t << "\n";
}

int main()
{
  myclass ob(10, 14);

  ob.show();

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  void set(int i, int j) { a = i; b = j; }
  void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
  myclass obj1, obj2;

  obj1.set(10, 4);

  // prirazeni obj1 do obj2
  obj2 = obj1;

  obj1.show();
  obj2.show();

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* nastavi o.i na jeho ctverec. Nema to vliv na objekt
   pouzity k volani sqr_it()
*/
void sqr_it(samp o)
{
   o.set_i(o.get_i() * o.get_i());

  cout << "Kopie a ma hodnotu i " << o.get_i();
  cout << "\n";
}

int main()
{
  samp a(10);

  sqr_it(a);  //a predavano hodnotou

  cout << "a.i zustava v main() nezmeneno: ";
  cout << a.get_i();

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) { i = n; }
  void set_i(int n) { i = n; }
  int get_i() { return i; }
};

/* Nastavi o.i na jeho ctverec. To ma vliv na
   volajici argument
*/
void sqr_it(samp *o)
{
   o->set_i(o->get_i() * o->get_i());

  cout << "Kopie a ma hodnotu i " << o->get_i();
  cout << "\n";
}

int main()
{
  samp a(10);

  sqr_it(&a); // predava adresu

  cout << "a.i ve funkci main() ma zmenenou hodnotu: ";
  cout << a.get_i();

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int i;
public:
  samp(int n) {
    i = n;
    cout << "Konstruktor\n";
  }
  ~samp() { cout << "Destruktor\n"; }
  int get_i() { return i; }
};

int sqr_it(samp o)
{
  return o.get_i() * o.get_i();
}

int main()
{
  samp a(10);

  cout << sqr_it(a) << "\n";

  return 0;
}
```

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class samp {
  char *s;
public:
  samp() { s = '\0'; }
  ~samp() { if(s) free(s); cout << "Uvolnuji s\n"; }
  void show() { cout << s << "\n"; }
  void set(char *str);
};

// Nacitani retezce
void samp::set(char *str)
{
  s = (char *) malloc(strlen(str)+1);
  if(!s) {
    cout << "Error\n";
    exit(1);
  }

  strcpy(s, str);
}

// Vraceni objektu
samp input()
{
  char s[80];
  samp str;

  cout << "Zadej retezec: ";
  cin >> s;

  str.set(s);
  return str;
}

int main()
{
  samp ob;

  ob = input();  // zde vznika chyba
  ob.show();

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int n, d;
public:
  myclass(int i, int j) { n = i; d = j; }
  friend int isfactor(myclass ob); //spratelena funkce
};

int isfactor(myclass ob)
{
  if(!(ob.n % ob.d)) return 1;
  else return 0;
}

int main()
{
  myclass ob1(10, 2), ob2(13, 3);

  if(isfactor(ob1)) cout << "10 je delitelne 2\n";
  else cout << "10 neni delitelne 2\n";

  if(isfactor(ob2)) cout << "13 je delitelne 3\n";
  else cout << "13 neni delitelne 3\n";

  return 0;
}
```

Řešení

Zadání: 1) Co je špatného v následujícím fragmentu?

```
#include <iostream.h>

class cl1 {
  int i, j;
public:
  cl1(int a, int b) { i = a; j = b; }
  // ...
};

class cl2 {
  int i, j;
public:
  cl2(int a, int b) { i = a; j = b; }
  // ...
};

int main()
{
  cl1 x(10, 20);
  cl2 y(0, 0);
  x = y;

  // ...
}
```

Řešení:

Přiřazovací příkaz **x = y** je chybný, protože **cl1** a **cl2** jsou dva různé typy tříd a objekty různých typů tříd nemohou být přiřazeny.

Zpět

Zadání: 2) Když je funkci předáván objekt, je vytvářena jeho kopie. Při návratu této funkce je volán destruktor kopie. Co je potom špatné v následujícím příkladu?

```
#include <iostream.h>
#include <stdlib.h>

class dyna {
  int *p;
public:
  dyna(int i);
  ~dyna() { free(p); cout << "Uvolnuji \n"; }
  int get() { return *p; }
};
```

```
dyna::dyna(int i)
{
  p = (int *) malloc(sizeof(int));
  if(!p) {
    cout << "Allocation failure\n";
    exit(1);
  }

  *p = i;
}

// vraceni zaporne hodnoty
int neg(dyna ob)
{
  return -ob.get();
}

int main()
{
  dyna o(-10);

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  dyna o2(20);
  cout << o2.get() << "\n";
  cout << neg(o2) << "\n";

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  return 0;
}
```

Řešení:

Pamě• použitá pro uložení integer a odkazovaná přes **p** v objektu **o**, který je použit pro volání **neg()** je uvolněna při zrušení kopie **o**, když končí **neg()**, bez ohledu na to, zda ji ještě **o** potřebuje v **main()**.

Zpět

Zadání: 3) Vytvořte třídu **who**. Nech• její konstruktor přebírá jednoznakový argument, který bude použit pro identifikaci objektu. A• konstruktor při vytváření objektu vypisuje:

```
Vytvarim who #x
```

Kde **x** je identifikační znak každého objektu. Když je objekt rušen, a• se zobrazuje zpráva:

```
Rusim who #x
```

Nakonec vytvořte funkci **make_who()**, která vrací objekt **who**. Každému objektu přiřaďte unikátní jméno.

Řešení:

```
#include <iostream.h>

class who {
  char name;
public:
  who(char c) {
     name = c;
     cout << "Vytvarim who #";
     cout << name << "\n";
  }
  ~who() { cout << "Rusim who #" << name << "\n"; }
};

who makewho()
{
  who temp('B');
  return temp;
}

int main()
{
  who ob('A');

  makewho();

  return 0;
}
```

Zpět

Zadání: 4) Může být adresa objektu využita funkcí jako argument?

Řešení:

Ano

Zpět

Zadání: 5) Co jsou to spřátelené funkce?

Řešení:

Spřátelené funkce nejsou členské funkce, ale mají zajištěný přístup k privátním členům třídy s níž jsou spřáteleny.

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

class who {
  char name;
public:
  who(char c) {
     name = c;
     cout << "Vytvarim who #";
     cout << name << "\n";
  }
  ~who() { cout << "Rusim who #" << name << "\n"; }
};

who makewho()
{
  who temp('B');
  return temp;
}

int main()
{
  who ob('A');

  makewho();

  return 0;
}
```

```
#include <iostream.h>
#include <stdlib.h>

class dyna {
  int *p;
public:
  dyna(int i);
  ~dyna() { free(p); cout << "Uvolnuji \n"; }
  int get() { return *p; }
};

dyna::dyna(int i)
{
  p = (int *) malloc(sizeof(int));
  if(!p) {
    cout << "Allocation failure\n";
    exit(1);
  }

  *p = i;
}

// vraceni zaporne hodnoty
int neg(dyna ob)
{
  return -ob.get();
}

int main()
{
  dyna o(-10);

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  dyna o2(20);
  cout << o2.get() << "\n";
  cout << neg(o2) << "\n";

  cout << o.get() << "\n";
  cout << neg(o) << "\n";

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(); // konstruktor
  void show();
};

myclass::myclass()
{
  cout << "V konstruktoru\n";
  a = 10;
}

void myclass::show()
{
  cout << a;
}

int main()
{
  myclass ob;

  ob.show();

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(); // konstruktor
  ~myclass(); //destruktor
  void show();
};

myclass::myclass()
{
  cout << "V konstruktoru\n";
  a = 10;
}

myclass::~myclass()
{
 cout << "V destruktoru\n";
}

void myclass::show()
{
  cout << a;
}

int main()
{
  myclass ob;

  ob.show();

  return 0;
}
```

```
#include <iostream.h>
#include <time.h>

class timer {
  clock_t start;
public:
  timer(); // konstruktor
  ~timer(); // destruktor
};

timer::timer()
{
  start = clock();
}

timer::~timer()
{
  clock_t end;

  end = clock();
  cout << "Interval: " << (end-start) / CLOCKS_PER_SEC << "\n";
}

int main()
{
  timer ob;
  char c;

  // zpozdeni ...
  cout << "Stiskni klavesu a pote ENTER: ";
  cin >> c;

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(int x); // konstruktor
  void show();
};

myclass::myclass(int x)
{
  cout << "V konstruktoru\n";
  a = x;
}
void myclass::show()
{
  cout << a << "\n";
}

int main()
{
  myclass ob(4);

  ob.show();

  return 0;
}
```

```
#include <iostream.h>

// Zakladni trida
class base {
  int i;
public:
  void set_i(int n);
  int get_i();
};

// Odvozena trida
class derived : public base {
  int j;
public:
  void set_j(int n);
  int mul();
};

// nastaveni hodnoty i v zakladni tride
void base::set_i(int n)
{
  i = n;
}

// vraceni hodnoty i do zakladni tridy
int base::get_i()
{
  return i;
}

// nastaveni hodnoty j v odvozene tride
void derived::set_j(int n)
{
  j = n;
}

// vraceni hodnoty j v odvozene tride
int derived::mul()
{
  // odvozena trida muze volat verejne funkce zakladni tridy
  return j * get_i();
}

int main()
{
  derived ob;

  ob.set_i(10);
  ob.set_j(4);

  cout << ob.mul();

  return 0;
}
```

```
#include <iostream.h>

class myclass {
  int a;
public:
  myclass(int x); // konstruktor
  int get();
};

myclass::myclass(int x)
{
  a = x;
}

int myclass::get()
{
  return a;
}

int main()
{
  myclass ob(120);  // vytvareni objektu
  myclass *p;  // vytvareni ukazatele na objekt

  p = &ob; // vlozeni adresy ob do p

  cout << "Hodnota objektu: " << ob.get();
  cout << "\n";

  cout << "Hodnota pointeru: " << p->get();

  return 0;
}
```

```
#include <iostream.h>

inline int even(int x)
{
  return !(x%2);
}

int main()
{
  if(even(10)) cout << "10 je sude\n";
  if(even(11)) cout << "11 je sude\n";

  return 0;
}
```

```
#include <iostream.h>

inline int min(int a, int b)
{
  return a<b ? a : b;
}

inline long min(long a, long b)
{
  return a<b ? a : b;
}

inline double min(double a, double b)
{
  return a<b ? a : b;
}

int main()
{
  cout << min(-10, 10) << "\n";
  cout << min(-10.01, 100.002) << "\n";
  cout << min(-10L, 12L) << "\n";

  return 0;
}
```

```
#include <iostream.h>

class samp {
  int i, j;
public:
  samp(int a, int b);
  int divisible() { return !(i%j); }  // inline funkce
};

samp::samp(int a, int b)
{
  i = a;
  j = b;
}

int main()
{
  samp ob1(10, 2), ob2(10, 3);

  // pravda
  if(ob1.divisible()) cout << "10 je beze zbytku delitelne 2\n";

  // nepravda
  if(ob2.divisible()) cout << "10 je beze zbytku delitelne 3\n";

  return 0;
}
```

Řešení

Zadání: 1) Co je špatného na konstruktoru v následujícím fragmentu?

```
class priklad {
   double a, b, c;
public:
   double priklad();   //chyba, proc?
};
```

Řešení:

Konstruktor **nemůže** mít návratový typ.

Zpět

Zadání: 2) Vytvořte třídu **t_and_d**, která předává aktuální systémový čas a datum, jako argument svému konstruktoru, když je vytvářena. A• třída obsahuje členskou funkci, která zobrazí čas a datum na obrazovce (Rada: Pro nalezení a zobrazení data a času použijte standardní funkce ze standardní knihovny)

Řešení:

```
#include <iostream.h>
#include <time.h>

class t_and_d {
  time_t systime;
public:
  t_and_d(time_t t);
  void show();
};

t_and_d::t_and_d(time_t t)
{
  systime = t;
}

void t_and_d::show()
{
  cout << ctime(&systime);
}

int main()
{
  time_t x;

  x = time(NULL);
```

```
  t_and_d ob(x);

  ob.show();

  return 0;
}
```

Zpět

Zadání: 3) Je dána následující třída. Vytvořte dvě odvozené třídy **rectangle** a **isosceles**. A• každá třída obsahuje funkci pojmenovanou **area()** která dle zadání vrací plochu čtyřúhelníku nebo rovnoramenného trojúhelníku. Pro inicializaci **height** a **width** použijte konstruktor s argumenty.

```
class area_c1 {
public:
   double height;
   double width;
};
```

Řešení:

```
#include <iostream.h>

class area_cl {
public:
  double height;
  double width;
};

class rectangle : public area_cl {
public:
  rectangle(double h, double w);
  double area();
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w);
  double area();
};

rectangle::rectangle(double h, double w)
{
  height = h;
  width = w;
}

isosceles::isosceles(double h, double w)
{
  height = h;
  width = w;
}
```

```
double rectangle::area()
{
  return width * height;
}

double isosceles::area()
{
  return 0.5 * width * height;
}

int main()
{
  rectangle b(10.0, 5.0);
  isosceles i(4.0, 6.0);

  cout << "Ctyruhelnik: " << b.area() << "\n";
  cout << "Trojuhelnik: " << i.area() << "\n";

  return 0;
}
```

Zpět

Zadání: 4) Co zobrazí tento program?

```
#include <iostream.h>

int main()
{
  int i = 10;
  long l = 1000000;
  double d = -0.0009;

  cout << i << ' ' << l << ' ' << d;
  cout << "\n";

  return 0;
}
```

Řešení:

```
10 1000000 -0.0009
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>
#include <time.h>

class t_and_d {
  time_t systime;
public:
  t_and_d(time_t t);
  void show();
};

t_and_d::t_and_d(time_t t)
{
  systime = t;
}

void t_and_d::show()
{
  cout << ctime(&systime);
}

int main()
{
  time_t x;

  x = time(NULL);

  t_and_d ob(x);

  ob.show();

  return 0;
}
```

```
#include <iostream.h>

class area_cl {
public:
  double height;
  double width;
};

class rectangle : public area_cl {
public:
  rectangle(double h, double w);
  double area();
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w);
  double area();
};

rectangle::rectangle(double h, double w)
{
  height = h;
  width = w;
}

isosceles::isosceles(double h, double w)
{
  height = h;
  width = w;
}

double rectangle::area()
{
  return width * height;
}

double isosceles::area()
{
  return 0.5 * width * height;
}

int main()
{
  rectangle b(10.0, 5.0);
  isosceles i(4.0, 6.0);

  cout << "Ctyruhelnik: " << b.area() << "\n";
  cout << "Trojuhelnik: " << i.area() << "\n";

  return 0;
}
```

```
#include <iostream.h>

int main()
{
  int i = 10;
  long l = 1000000;
  double d = -0.0009;

  cout << i << ' ' << l << ' ' << d;
  cout << "\n";

  return 0;
}
```

```
#include <iostream.h>

int main(void)
{
 int i;

 cout << "Zadejte celociselnou hodnotu: ";
 cin >> i;
 cout << "Zadana hodnota byla " << i << "\n";
 cout << "Jeji dvojnasobek je " << i*2 << " a druha mocnina je " << i*i;

 return 0;
}
```

```
#include <iostream.h>

int main(void)
{
 int i;
 char str[80];

 cout << "Zadejte celociselnou hodnotu a retezec: ";
 cin >> i >> str;
 cout << "Zadana data: " << i << " " << str;

 return 0;
}
```

```
#include <iostream.h>

class mojetrida
{
  int a;
public:
  void nastav_a(int num);
  int cti_a();
};

void mojetrida::nastav_a(int num)
{
  a = num;
}

int mojetrida::cti_a()
{
  return a;
}

int main()
{
  mojetrida obj1, obj2;

  obj1.nastav_a(10);
  obj2.nastav_a(5);

  cout << obj1.cti_a() << "\n";
  cout << obj2.cti_a() << "\n";

  return 0;
}
```

```
#include <iostream.h>

int abs(int n);
long abs(long n);
double abs(double n);

int main()
{
  cout << "Absolutni hodnota -10: " << abs(-10) << "\n";
  cout << "Absolutni hodnota -10L: " << abs(-10L) << "\n";
  cout << "Absolutni hodnota -10.01: " << abs(-10.01) << "\n";

  return 0;
}

// abs() pro int
int abs(int n)
{
  cout << "Abs() pro Integer\n";
  return n<0 ? -n : n;
}

// abs() pro longs
long abs(long n)
{
  cout << "Abs() pro long\n";
  return n<0 ? -n : n;
}

// abs() pro doubles
double abs(double n)
{
  cout << "Abs() pro double\n";
  return n<0 ? -n : n;
}
```

Řešení

Zadání: 1) Napište program, který převádí metry na centimetry. Program bude opakovat tento proces, dokud uživatel nezadá nulu.

Řešení:

```
#include <iostream.h>

int main()
{
  double centimetry;

  do {
     cout << "Zadej pocet metru (0 = konec): ";
     cin >> centimetry;

     cout << centimetry * 100 << " centimetru\n";
  } while (centimetry != 0.0);

  return 0;
}
```

Zpět

Zadání: 2) Převeďte tento program napsaný v jazyce C tak, aby používat I/O styly z C++.

```
#include <stdio.h>

int main(void)
{
  int a, b, d, min;

  printf("Zadej dve cisla: ");
  scanf("%d%d", &a, &b);
  min = a > b ? b : a;
  for(d = 2; d<=min; d++)
    if(((a%d)==0) && ((b%d)==0)) break;
  if((d-1)==min) {
    printf("Zadny spolecny delitel\n");
    return 0;
  }
  printf("Nejmensi spolecny delitel je %d\n", d);
  return 0;
}
```

Řešení:

```
#include <iostream.h>

int main()
{
  int a, b, d, min;

  cout << "Zadej dve cisla: ";
  cin >> a >> b;

  min = a > b ? b : a;

  for(d = 2; d<=min; d++)
         if(((a%d)==0) && ((b%d)==0)) break;
  if((d-1)==min) {
    cout << "Zadny spolecny delitel\n";
    return 0;
  }
  cout << "Nejmensi spolecny delitel je " << d << ".\n";

  return 0;
}
```

Zpět

Zadání: 3) Vytvořte funkci **min()**, která vrací menší ze dvou zadaných numerických argumentů, použitých ve volání funkce. Přeložte **min()** tak, aby jako své argumenty přijímala znaky, integer a double.

Řešení:

```
#include <stdio.h>
#include <ctype.h>

char min(char a, char b);
int min(int a, int b);
double min(double a, double b);

int main()
{
  cout << "Min is: " << min('x', 'a') << "\n";
  cout << "Min is: " << min(10, 20) << "\n";
  cout << "Min is: " << min(0.2234, 99.2) << "\n";

  return 0;
}

// min() for chars
char min(char a, char b)
{
  return tolower(a)<tolower(b) ? a : b;
}

// min() for ints
int min(int a, int b)
```

```
{
  return a<b ? a : b;
}

// min() for doubles
double min(double a, double b)
{
  return a<b ? a : b;
}
```

Zpět

Zadání: 4) Mějme program napsaný podle konvencí C++. Předveďte jak je změnit do formy pro jazyk C.

```
#include <iostream.h>

int f(int a);

int main()
{
  cout << f(10);

  return 0;
}

int f(int a)
{
  return a * 3.1416;
}
```

Řešení:

```
#include <stdio.h>

int f(int a);

int main()
{
  printf("%d",f(10));

  return 0;
}

int f(int a)
{
  return a * 3.1416;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```
#include <iostream.h>

int main()
{
  double centimetry;

  do {
     cout << "Zadej pocet metru (0 = konec): ";
     cin >> centimetry;

     cout << centimetry * 100 << " centimetru\n";
  } while (centimetry != 0.0);

  return 0;
}
```

```
#include <iostream.h>

int main()
{
  int a, b, d, min;

  cout << "Zadej dve cisla: ";
  cin >> a >> b;

  min = a > b ? b : a;

  for(d = 2; d<=min; d++)
         if(((a%d)==0) && ((b%d)==0)) break;
  if((d-1)==min) {
    cout << "Zadny spolecny delitel\n";
    return 0;
  }
  cout << "Nejmensi spolecny delitel je " << d << ".\n";

  return 0;
}
```

```
#include <iostream.h>
#include <ctype.h>

char min(char a, char b);
int min(int a, int b);
double min(double a, double b);

int main()
{
  cout << "Min is: " << min('x', 'a') << "\n";
  cout << "Min is: " << min(10, 20) << "\n";
  cout << "Min is: " << min(0.2234, 99.2) << "\n";

  return 0;
}

// min() for chars
char min(char a, char b)
{
  return tolower(a)<tolower(b) ? a : b;
}

// min() for ints
int min(int a, int b)
{
  return a<b ? a : b;
}

// min() for doubles
double min(double a, double b)
{
  return a<b ? a : b;
}
```

```
#include <iostream.h>


int f(int a);

int main()
{
  cout << f(10);

  return 0;
}

int f(int a)
{
  return a * 3.1416;
}
```

```c
#include <stdio.h>

int main (void)
{
 int *p, q;
 q = 199;   /* priradi q hodnotu 199 */
 p = &q;   /* priradi p adresu q */
 printf("%d", *p);      /* zobrazi hodnotu q pomoci ukazatele */
 return 0;
}
```

```c
#include <stdio.h>

int main (void)
{
 int *p, q;
 p = &q;   /* ziskani adresy q */
 *p = 199;         /* priradi q hodnotu pomoci ukazatele */
 printf("hodnota q je %d", q);
 return 0;
}
```

Zadání: 1) Co je chybné v těchto jménech proměnných?

a. pocet-hodin
b. $suma
c. black+white
d. 9krat

Řešení: Jména proměnných jsou chybná protože:

a. Ve jménu proměnné nelze použít spojovník
b. Ve jménu proměnné nelze použít znak $
c. Ve jménu proměnné nelze použít znak +
d. Jméno proměnné nesmí začínat číslicí

Zpět

Zadání: 2) Která z následujících slov nejsou klíčová?

a. auto
b. else
c. goto
d. void

Řešení: Všechna uvedená slova jsou klíčová.

Zpět

Zadání: 3) Co je chybně na této části programu?

```
/* timto se nacte cislo
scanf ("%d", &num);
```

Řešení: Chybí ukončovací znak komentáře */.

Zpět

Zadání: 4) Datový typ char vyžaduje:

a. 8 bitů
b. 18 bitů

c. 6 bitů
d. 16 bitů

Řešení: Datový typ char vyžaduje **8** bitů. Správná odpověď je tedy **a.**

Zpět

Zadání: 5) Podle rozsahu hodnot je:

a. int = long
b. int = float
c. float > double
d. float <= double

Řešení: Podle rozsahu hodnot je **float <= double**. Správná odpověď je tedy **d.**

Zpět

Zadání: 6) Napište program, který vytiskne **Mám rád jazyk C** - pomocí tří konstantních řetězců.

Řešení:

```
#include <stdio.h>

int main(void)
{
 const char *prvni = "Mam rad";
 const char *druhy = "jazyk";
 const char *treti = "C";

 printf("%s %s %s", prvni, druhy, treti);

 return 0;
}
```

Zpět

Zadání: 7) Mějme celočíselnou hodnotu **130L**. Jde o:

a. desítkovou hodnotu se znaménkem
b. dvojkovou hodnotu se znaménkem
c. desítkovou hodnotu typu long
d. desítkovou hodnotu typu long bez znaménka

Řešení: Celočíselná hodnota **130L** je **desítková hodnota typu long**. Správná hodnota je tedy **c.**

Zpět

Zadání: 8) Napište program, který přečte a vytiskne hodnotu typu **long int**.

Řešení:

```
#include <stdio.h>

int main(void)
{
 long int i;

 printf("Zadejte cislo: ");
 scanf ("%ld", &i);
 printf("%ld", i);

 return 0;
}
```

Zpět

Zadání: 9) Co je to ukazatel?

Řešení: Ukazatel je proměnná, která obsahuje adresu jiné proměnné.

Zpět

Zadání: 10) Jaké jsou ukazatelové operátory a jaká je jejich funkce?

Řešení: Ukazatelové operátory jsou * a &. Operátor * vrací hodnotu objektu, na který ukazuje ukazatel, který před ním stojí. Operátor & vrací adresu proměnné, před kterou stojí.

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>;

int main(void)
{
 const char *prvni = "Mam rad";
 const char *druhy = "jazyk";
 const char *treti = "C";

 printf("%s %s %s", prvni, druhy, treti);

 return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
 long int i;

 printf("Zadejte cislo: ");
 scanf ("%ld", &i);
 printf("%ld", i);

 return 0;
}
```

```c
#include <stdio.h>

int main (void)
{
 printf("Jednoduchy program napsany v jazyce C");

 return 0;
}
```

```c
#include <stdio.h>

int main (void)
{
 printf("Dalsi jednoduchy program ");
 printf("napsany v ");
 printf("jazyce C");

 return 0;
}
```

```c
#include <stdio.h>

int main (void)
{
 int cislo;

 printf("Zadej cele cislo: ");
 scanf("%d",&cislo);


 printf("Zadal jsi: ");
 printf("%d", cislo);

 return 0;
}
```

Řešení

Zadání: 1) Napište program, který načte dvě celá čísla a poté vytiskne jejich součet.

Řešení:

```
#include <stdio.h>

int main(void)
{
 int a,b,c;

 printf("Zadej prvni cislo: ");
 scanf("%d", &a);
 printf("Zadej druhe cislo: ");
 scanf("%d", &b);
 c = a + b;
 printf("Soucet cisel %d + %d je %d",a,b,c);

 return 0;
}
```

Zpět

Poslední změna: **26. 2. 2002**

RNDr. Petr Šaloun Ph.D., Radek Šup, VŠB - TU Ostrava, fakulta Elektrotechniky a Informatiky

```c
#include <stdio.h>

int main(void)
{
 int a,b,c;

 printf("Zadej prvni cislo: ");
 scanf("%d", &a);
 printf("Zadej druhe cislo: ");
 scanf("%d", &b);
 c = a + b;
 printf("Soucet cisel %d + %d je %d",a,b,c);

 return 0;
}
```