

PROGRAMOVÁNÍ V C

URČENO PRO VZDĚLÁVÁNÍ V AKREDITOVANÝCH STUDIJNÍCH PROGRAMECH

ROSTISLAV FOJTÍK

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
OPATŘENÍ: 7.2
ČÍSLO OBLASTI PODPORY: 7.2.2

INOVACE VÝUKY INFORMATICKÝCH PŘEDMĚTŮ VE STUDIJNÍCH PROGRAMECH OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.2.00/28.0245

OSTRAVA 2013

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzent: Doc. Ing. František Huňka, CSc.

Název: Programování v C
Autor: Rostislav Fojtík
Vydání: druhé, 2013
Počet stran: 145

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

© Rostislav Fojtík
© Ostravská univerzita v Ostravě

Obsah

Úvod 3
1. Základní informace o jazyku C 4
Stručná historie jazyka C 4
Jednoduché datové typy 6
Typová konverze 7
Jednoduché operátory 7
Příklady 9
2. Vytváříme první projekt 12
Nastavení překladače 12
3. Základní řídící struktury 22
Vstup a výstup 22
Bloky příkazů 23
Podmínky 24
Vícenásobná podmínka 25
Cykly 25
Příkazy skoků 26
Příklady 27
Shrnutí učiva 32
4. Funkce 34
Rekurzivní funkce 35
Funkce návratovým typem void 35
Parametry funkcí 36
Příklady 36
Shrnutí učiva 39
Rekurze 40
5. Preprocesor 42
Vkládání souborů 42
Makra bez parametrů 42
Makra s parametry 43
Podmíněný překlad 44
Hlavičkové soubory 45
Příklady 45
Opakovací test 49
Shrnutí učiva 50
6. Preprocesor 52
Funkce a pointery 53
Pointerová aritmetika 54
Dynamická alokace paměti 54
Příklady 55
Shrnutí učiva 59
7. Jednorozměrné pole 60
Vstupní test 60
Statické pole 61
Dynamické pole 61
Pole jako parametr funkce 62
Příklady 63
Opakovací test 68
Shrnutí učiva 69

8. Vícerozměrná pole 70
Dvourozměrné statické pole....70
Pole pointerů 71
Pointer na pole....71
Pointer na pointer....71
Dvourozměrné pole jako parametr funkce....72
Inicializace polí 72
Příklady 73
Opakovací test....76
Shrnutí učiva 77
9. Práce s řetězci 78
Základní informace o práci s řetězci 78
Funkce pro práci s řetězci....79
Čtení a výpis řetězců 79
Příklady 79
Opakovací test....83
Shrnutí učiva 84
10. Struktury, unionya výčtové typy....86
Struktury 86
Sturktury a funkce 88
Union 88
Výčtový typ 89
Příklady 90
Opakovací test....94
Shrnutí učiva 95
11. Oddělený překlad. Paměťové třídy. 96
Paměťové třídy 97
Opakovací test....100
Shrnutí učiva 101
12. Práce se soubory....102
Příklady 106
Opakovací test....109
Shrnutí učiva 111
13. Další možnosti jazyka C....112
Parametry funkce main 112
Funkce s proměnným počtem argumentů 113
Bitové operace....114
Bitová pole....115
Opakovací test....118
Shrnutí učiva 119
14. Třídící algoritmy 122
15. Dynamické datové struktury 130
16. Další příklady k procvičení 136
Literatura....145

Vysvětlivky k používaným symbolům


Průvodce studiem – vstup autora do textu, specifický způsob, kterým se studentem komunikuje, povzbuzuje jej, doplňuje text o další informace


Příklad – objasnění nebo konkretizování problematiky na příkladu ze života, z praxe, ze společenské reality, apod.


Pojmy k zapamatování.


Shrnutí – shrnutí předcházející látky, shrnutí kapitoly.


Literatura – použitá ve studijním materiálu, pro doplnění a rozšíření poznatků.


Kontrolní otázky a úkoly – prověřují, do jaké míry studující text a problematiku pochopil, zapamatoval si podstatné a důležité informace a zda je dokáže aplikovat při řešení problémů.


Úkoly k textu – je potřeba je splnit neprodleně, neboť pomáhají dobrému zvládnutí následující látky.


Korespondenční úkoly – při jejich plnění postupuje studující podle pokynů s notnou dávkou vlastní iniciativy. Úkoly se průběžně evidují a hodnotí v průběhu celého kurzu.


Úkoly k zamyšlení.


Část pro zájemce – přináší látku a úkoly rozšiřující úroveň základního kurzu. Pasáže a úkoly jsou dobrovolné.


Testy a otázky – ke kterým řešení, odpovědi a výsledky studující najdou v rámci studijní opory.



Řešení a odpovědi – vážou se na konkrétní úkoly, zadání a testy.

Úvod



Úvodní lekce slouží studentům k orientaci ve výukovém kurzu „Programování v C“. Kurz je určen studentům oborů zaměřených na informatiku a výpočetní techniku, kteří studují distanční, kombinovanou nebo prezenční formou studia.

Cílem kurzu „Programování v C" je seznámit se základními rysy strukturovaného programování a jeho praktickým využitím v jazyce C. Součásti kurzu je rovněž tématika analýzy a tvorby programů.

Po absolvování kurzu by student měl být schopen:

- tvořit programy v jazyku C
- správně analyzovat a navrhovat řešení programů
- využívat standardní knihovny jazyka C
- znát základní architekturu programů v jazyce C

1. ZÁKLADNÍ INFORMACE O JAZYKU C

Cílem této lekce je seznámit se se základními informacemi o programovacím jazyku C tak, aby jste byli schopni sestavit jednoduchý program.

Budete:

- znát historii programovacího jazyka C
- umět napsat jednoduchý zdrojový text v jazyku C
- znát jednoduché datové typy a operátory jazyka C
- vědět, jakým způsobem pracuje překladač jazyka C

Klíčová slova této kapitoly:
programovací jazyk C, datové typy, historie jazyka

Čas potřebný k prostudování učiva kapitoly:
2 hodiny



Stručná historie jazyka C

Programovací jazyk C patří dnes mezi velmi populární jazyka, zvlášť mezi profesionály. Za autory jazyka jsou považování Dennis Ritchie a Ken Thompson, kteří jej sestavili na počátku sedmdesátých let. První standard jazyka je popsán v knize D. Ritchieho a B. W. Kernighama The C Programming Language z roku 1978. Tento standard je často označován jako K&R. Dnes již jsou stanoveny mezinárodní standardy (ANSI, ISO), které zajišťují přenositelnost na jiné typy počítačů.

Obecné charakteristiky jazyka C:

- C je obecně použitelný programovací jazyk (Fortran - vědeckotechnické výpočty; Cobol - úlohy z oblasti obchodu; Pascal, Scheme - pro výuku; Assembler - systémové programování ...)
- jedná se o strukturovaný jazyk
- většinou je jazyk C implementován jako překládač
- překladač jazyka C je rozsahem malý a lze ho poměrně snadno přenést na jiné platformy
- programy vytvořené pomocí jazyka C mají poměrně krátkou dobu provádění, nízké nároky na paměť a dají se lehce přenášet na jiné počítače (portabilita)

Jazyk C byl dále upravován a rovněž byl použit jako základ při vývoji programovacího jazyka C++, který je dílem Bjarne Stroustrupa, nebo jazyka Objective-C, který se využívá pro vývoj aplikací v operačních systémech Mac OS X a iOS.

Způsob zpracování programu

Zpracování programu probíhá v několika fázích. Nejprve je potřeba v editoru napsat zdrojový soubor, který má většinou příponu C. Dále přichází na řadu preprocesor, který bývá součástí překladače a který dále zpracovává zdrojový text. Například vkládá hlavičkové soubory, rozvíjí makra, atd. V další fázi compiler (překladač, kompilátor) provádí překlad upraveného zdrojového textu do relativního kódu (někdy kódu relativních adres) a vytvoří soubor s příponou „obj“ nebo „o“. V této fázi nejsou ještě známy adresy proměnných a funkcí. Následuje spojování (sestavování) programu pomoci linkeru, který zajišťuje nahrazení relativních adres adresami absolutními a provede odkazy na příslušné knihovní funkce. Výsledkem této činnosti je již spustitelný soubor, například s příponou EXE (případně COM).

Mnoho dnešních překladačů je v rámci daného integrovaného prostředí spojeno s překladačem programovacího jazyka C++. Zde může nastat určitý problém při nedbalé práci. Obvykle je vývojové prostředí nastaveno tak, že implicitně je spouštěn překladač C++. Proto je nutné před kompilaci zdrojový soubor nejprve uložit a to nejlépe s příponou C a přenastavit typ překladače. Nehledě k tomu, že spouštět neuložený program je hazardování z textem, který programátor v potu tváře vytvořil!

Během kompilace programu je potřeba nejprve soubor uložit. Některá vývojová prostředí totiž obsahují kompilátory jazyka C i C++ (což jsou do značné míry rozdílné programovací jazyky). Nástroje většinou implicitně připojují ke zdrojovému souboru koncovku cpp a spouští kompilátor C++. Takto například pracuje starší rozšířený kompilátor Borland C/C++ 3.1.

Po napsání a uložení zdrojového souboru se nesnažte program okamžitě spustit. Ale nejprve soubor překompilujte a zkontrolujte si errors i warnings. Warnings (upozornění) nemusí ještě stoprocentně znamenat chybu, ale upozorňují na podezřelou nebo neobvyklou konstrukci.

Základní informace

Programovací jazyk C důsledně rozlišuje velká a malá písmena. Proto Nazev, nazev a NAZEV jsou jednoznačně různé identifikátory.

Číselné konstanty mohou být v desítkové, osmičkové nebo šestnáctkové soustavě. Desítkové celé číslo je vyjádřeno posloupnosti číslic, z niž první nesmí být nula. Např. 165, 12, 1. V osmičkové (oktalové) začíná posloupnost číslic nulou: 061, 07. Číslo zapsáno v šestnáctkové (hexadecimální) soustavě je vytvořené posloupnosti nuly, malého či velkého znaku x a hexadecimálními znaky (0 - 9, A -F). U záporných čísel se píše znaménko mínus.

Reálné konstanty jsou implicitně typu double zapisují se s desetinou tečkou. Například: 3.14 , 0.65 , .65 , 12. , 5e12, 1E5. Konstanty typu float se ukončují písmenem F nebo f (2.14f). Typ long double zakončuje písmeno L nebo l (4e12L).

Znakové konstanty jsou ohraničeny apostrofy: 'a' , 'X', '[', '2'. Znaky z počátku ASCII tabulky se zapisují ve tvaru '\ooo', kde písmena ooo jsou nahrazeny posloupností tří oktálových číslic.

Příklad: '\010', '\001', '\013'. Je možné také použít hexadecimálního zápisu.

Příklad: '\0x0B', '\0x01', '\0x1f'. Některé obvykle používané znaky mají své znakové vyjádření:

\n \0x0A nová řádka
\a \0x07 pípnutí
\r \0x0D návrat na začátek řádky
\t \0x0C tabulátor
\b \0x08 posun doleva
\0 \0x00 nulový znak

Řetězcové konstanty jsou ohraničeny uvozovkami. Příklad: "Nejaky text".

Komentáře a poznámky jsou ohraničeny mezi znaky /* a */.

Příklad:

```
/* mezi temito znaky je nejaky komentar */
```

Pozor na vložené komentáře. Poznámka je vždy ukončena první dvojicí znaků */.

Příklad:

```
/* mezi temito znaky je nejaky komentar
  /* zde je nejaky vlozeny komentar */
  tato cast jiz není povazovana za komentar!
*/
```

Jednoduché datové typy

Podobně jako jazyk Pascal má programovací jazyk C jednoduché datové typy, které určují množinu přípustných hodnot. Celočíselné ordinální typy jsou: short int (zkráceně short), int, long int (zkráceně long), char. Mohou být buď unsigned nebo signed, to jest neznaménkové (tedy jen kladné) nebo se znaménkem (rozumí se unární '+' nebo '-'). Neordinální typy (reálné čísla s pohyblivou řadovou tečkou) jsou float, double a long double. Jednotlivé rozsahy jsou určeny příslušnými překladači a dají se zjistit aplikací operátoru sizeof. Příklad: sizeof(int).

Definice proměnných se v jazyce C provádí následujícím způsobem:

```
int a,b;
unsigned int c;
float f,g,h;
long l;
```

Definice proměnných může být spojená s její inicializací na počáteční hodnotu.:

```
int a = 1, b = 12, c;
```



Kontrolní otázka:

Zjistěte jaký rozsah mají datové typy char, int, long, float, double a long double ve vašem kompilátoru jazyka C.

Typová konverze

Jazyk C umožňuje implicitní (automatickou) a explicitní (požadovanou) typovou konverzi (převod mezi datovými typy).

K implicitní konverzi dochází:

1) V přiřazovacích výrazech je typ na pravé straně konvertován na typ na levé straně výrazu.

2) Jsou-li dva operandy ve výrazu různých typů, pak se operand s nižší prioritou konvertuje na typ s prioritou vyšší. Podle schématu int => unsigned int => long => unsigned long => float=> double => long double

3) Typ char a short int se konvertují automaticky na int

Explicitní konverzi využíváme v případě, že chceme změnit určitý datový typ (přetypovat) a nenastane samovolná konverze. Provádí se tak, že do kulatých závorek umístěných před konvertovanou proměnnou či výrazem uvedeme nový datový typ.



Příklady:

```
(int)char_vyraz  Převede char výraz na int
(float)int_vyraz Převede int výraz na float
(int)float_vyraz Převede float výraz na int, odřízne
desetinnou část.
```

Pozor, však na přetečení rozsahu čísla!

Jednoduché operátory

Binární operátory

+ sčítání
- odečítání
* násobení
/ celočíselné dělení
/ reálné dělení
% modulo

O tom, zda dělení bude celočíselné nebo reálné rozhoduje typ operandů. Je-li alespoň jeden z nich typu float, double nebo long double bude dělení realné. Budou-li oba operandy celá čísla, jedná se o dělení celočíselné.

Unární operátory

Zde patří běžné unární plus + a unární mínus -. Dále se v jazyku C používají speciální operátory inkrementace ++ a dekrementace --. Tyto operátory zvětšují (případně zmenšují) výraz o jedničku. Podle toho zda tyto operátory leží před nebo za operandem se buď nejprve operand upraví o jedničku a pak se tato hodnota použije ve výrazu nebo naopak.

Příklad:

```
int a = 5, b = 2 , c;

b++;  /* b má hodnotu 3 */
c = a + b++; /* c bude 8, a bude 5, b bude 4*/
c = ++a + b; /* c bude 10, a bude 6, b bude 4*/
```

Přiřazovací operátory

Pro přiřazení se používá znaku =. Kromě jednoduchého přiřazení umožňuje jazyk C použití dalších přiřazovacích operátorů. Například:

```
l-hodnota += výraz; znamená   l-hodnota = l-hodnota
+ výraz;
l-hodnota -= výraz; znamená l-hodnota = l-hodnota - výraz;
l-hodnota *= výraz; znamená l-hodnota = l-hodnota * výraz;
l-hodnota /= výraz; znamená l-hodnota = l-hodnota / výraz;
l-hodnota %= výraz; znamená l-hodnota = l-hodnota % výraz;

a += 12;   /* to samé jako a = a + 12;  */
```

Kontrolní otázka:



Jakou hodnotu bude mít proměnná x v následujícím příkladu?

```
int y = 20;
int x = 10;
x++;              // 1.
x += 5;           // 2.
x = y / 3         // 3.
```



Řešení:

1. x = 21
2. x = 26
3. x = 6

Ukázka programu

Jednoduchý příklad zdrojového textu:

```
/*
 * Ukazkovy zdrojovy text
 * Rostislav Fojtik, Ostrava, 2012
 */

/* vlozeni hlavickovych souboru */
#include <stdio.h>

/* deklaracni cast */
/* definice globalnich promennych */
int glob = 100;

/* hlavicky funkci */
int TretiMoc(int);

/* hlavni funkce */
int main( )
{
```

```
	/* definice lokalnich promennych */
	int lokalni;

	lokalni = TretiMoc(glob);
	printf("%d", lokalni);

	return 0;
}

/* definice funkci */
int TretiMoc(int x)
{
	return (x * x * x);
}
```

Definice mohou být uváděny před hlavní funkci main. Vzhledem k lepší přehlednosti, je však doporučuji psát až za tuto funkci.

Příklady



Příklad č.1

Krokujte následující program, v okně watch sledujte hodnoty jednotlivých proměnných.

```
int main()
{
 int a=10, b=5, c;

 a++;
 c = a + b;
 c = a + b++;
 c = --a + b;
 c = a+++b;

 return 0;
}
```



Příklad č. 2

Krokujte následující program, v okně *watch* sledujte hodnoty jednotlivých proměnných.

```
int main()
{
 int a=10, b=4, c;
 float f=3.1, g;

 c = a / b;
 c = a % b;
 g = a / b;
 g = a / f;
 g = a / (float)b;
 return 0;
}
```

Shrnutí kapitoly



• Obecné charakteristiky jazyka C:
- C je obecně použitelný programovací jazyk
- jedná se o strukturovaný jazyk
- většinou je jazyk C implementován jako překládač
- překladač jazyka C je rozsahem malý a lze ho poměrně snadno přenést na jiné platformy
- programy vytvořené pomocí jazyka C mají poměrně krátkou dobu provádění, nízké nároky na paměť a dají se lehce přenášet na jiné počítače (portabilita)

Mezinárodní standard jazyka C - ISO 9899 - 1998

• Zpracování programu probíhá ve fázích:
1. V editoru se napíše zdrojový soubor, který má většinou příponu C.
2. Spustí se *preprocesor.*
3. V další fázi pracuje *compiler* (překladač, kompilátor) provádí překlad upraveného zdrojového textu do relativního kódu (někdy kódu relativních adres) a vytvoří soubor s příponou OBJ.
4. Následuje spojování (sestavování) programu pomoci *linkeru*, který zajišťuje nahrazení relativních adres adresami absolutními a provede odkazy na příslušné knihovní funkce. Výsledkem této činnosti je již spustitelný soubor s příponou EXE (případně COM).

• Programovací jazyk C důsledně rozlišuje velká a malá písmena.

• Definice proměnných se v jazyce C provádí následujícím způsobem:

```
unsigned int c;
float f,g,h;
```

Definice proměnných může být spojená s její inicializací na počáteční hodnotu.:

```
int a = 1, b = 12, c;
```

• Jazyk C umožňuje *implicitní* (automatickou) a *explicitní* (požadovanou) typovou konverzi (převod mezi datovými typy).

2. VYTVÁŘÍME PRVNÍ PROJEKT

Cílem této lekce je seznámit se s vytvářením projektů ve vývojovém prostředí.

Budete:

- umět správně vytvořit projekt pro jazyk C
- vědět, jak pracovat s vývojovým prostředím

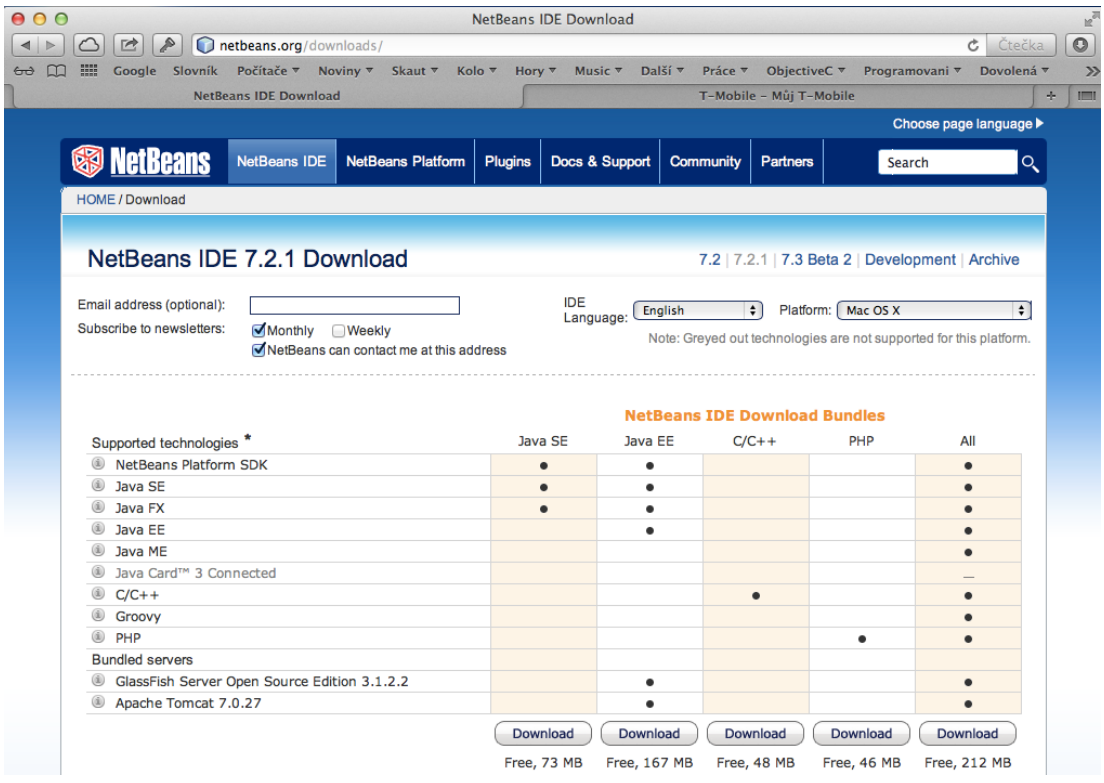
Klíčová slova této kapitoly:
kompilátor, gcc, vývojové nástroje, NetBeans, XCode, MS visual Studio



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Nastavení překladače

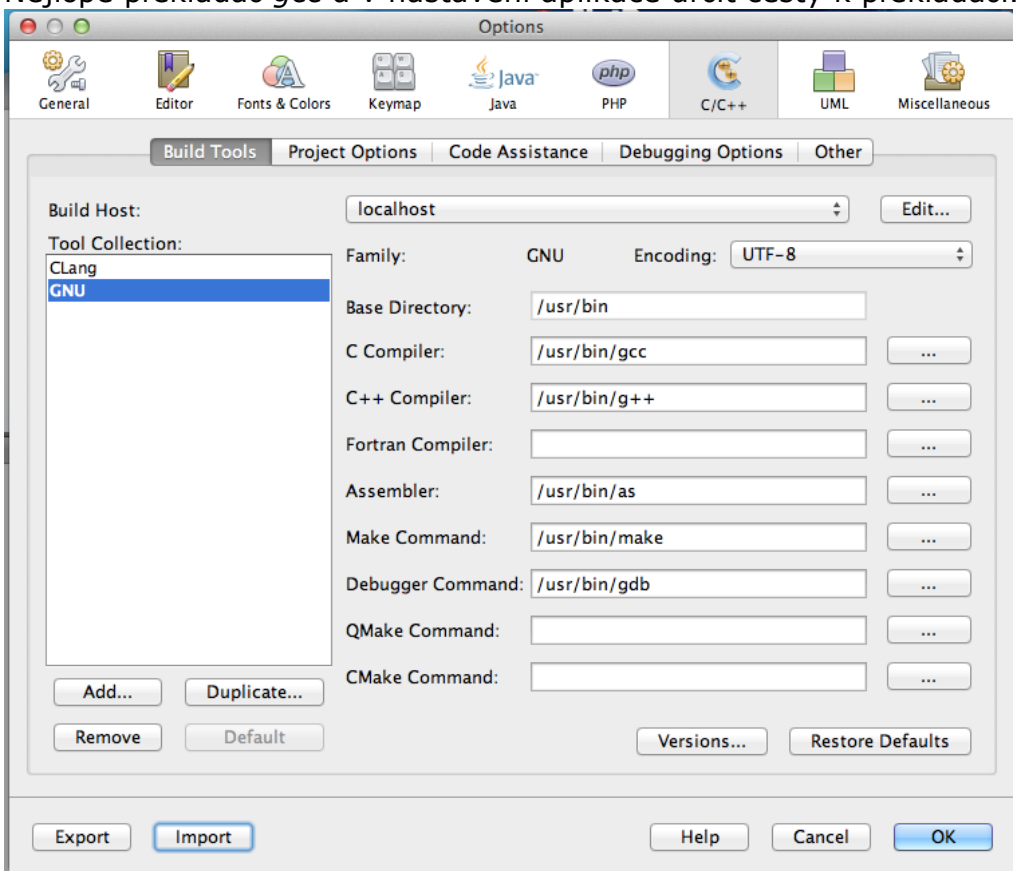
Současné vývojové nástroje obvykle obsahují překladače dva: jazyka C a jazyka C++. Je potřeba si uvědomit, že i přes velkou podobnost v některých ohledech, se jedná o dva rozdílné programovací jazyky. Proto při vytváření programů, které mají odpovídat specifikací jazyka C, je potřeba nastavit příslušný překladač. Obvykle je jako implicitní zvolen překladač jazyka C++, a proto je na uživateli, aby provedl změnu.



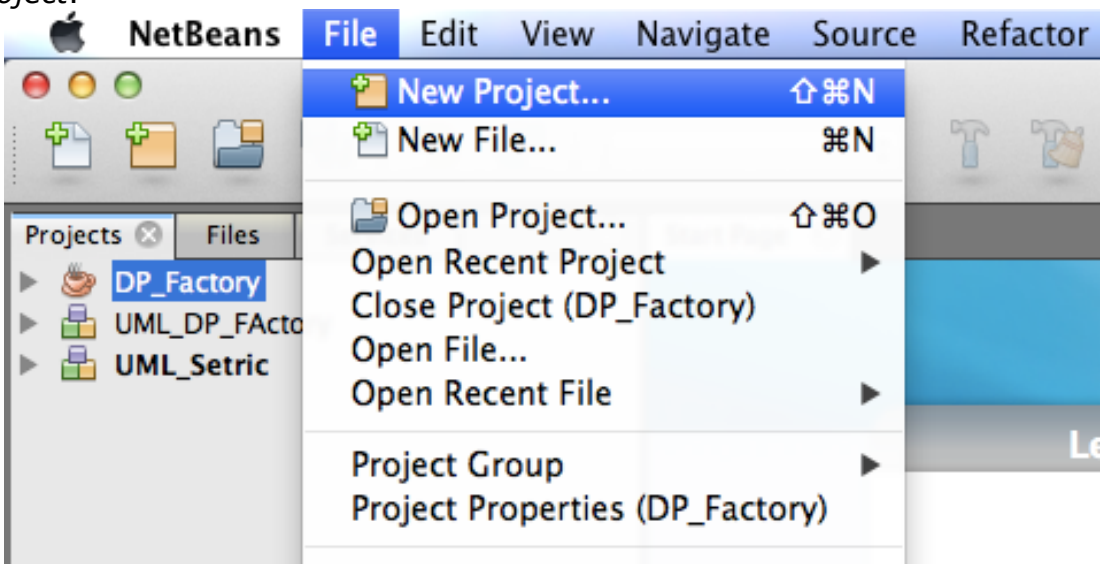
Vytvoření projektu v prostředí NetBeans

Vývojové prostředí NetBeans je primárně určeno sice pro vývoj programů v jazyku Java, ale ze stránek www.netbeans.org lze bezplatně stáhnout verzi s podporou pro vývoj v jazyicích PHP, Ruby i C/C++.

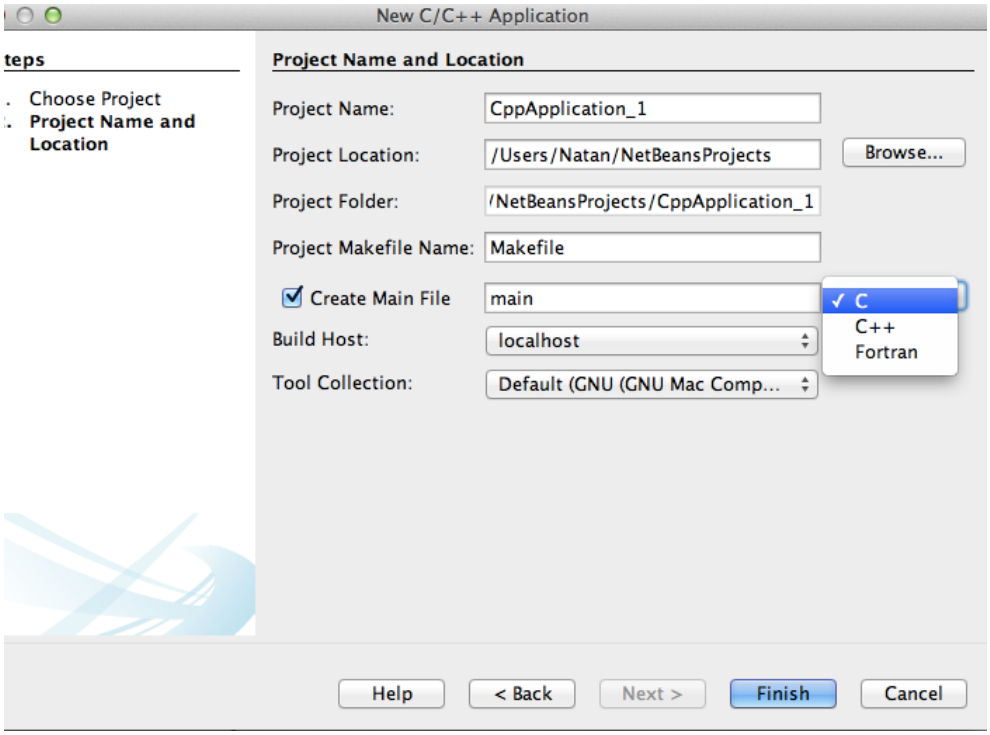
Je však nutné do počítače doinstalovat překladač jazyka C a C++. Nejlépe překladač *gcc* a v nastavení aplikace určit cesty k překladači.



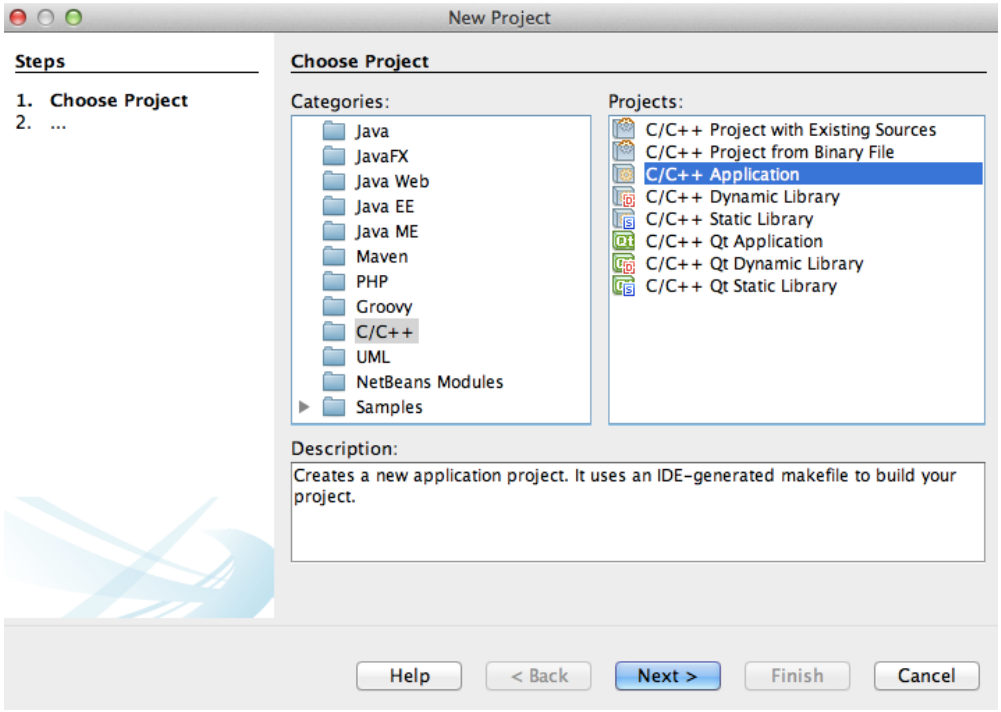
Při vytváření nového programu si v menu *File* vybereme volbu *New Project*.

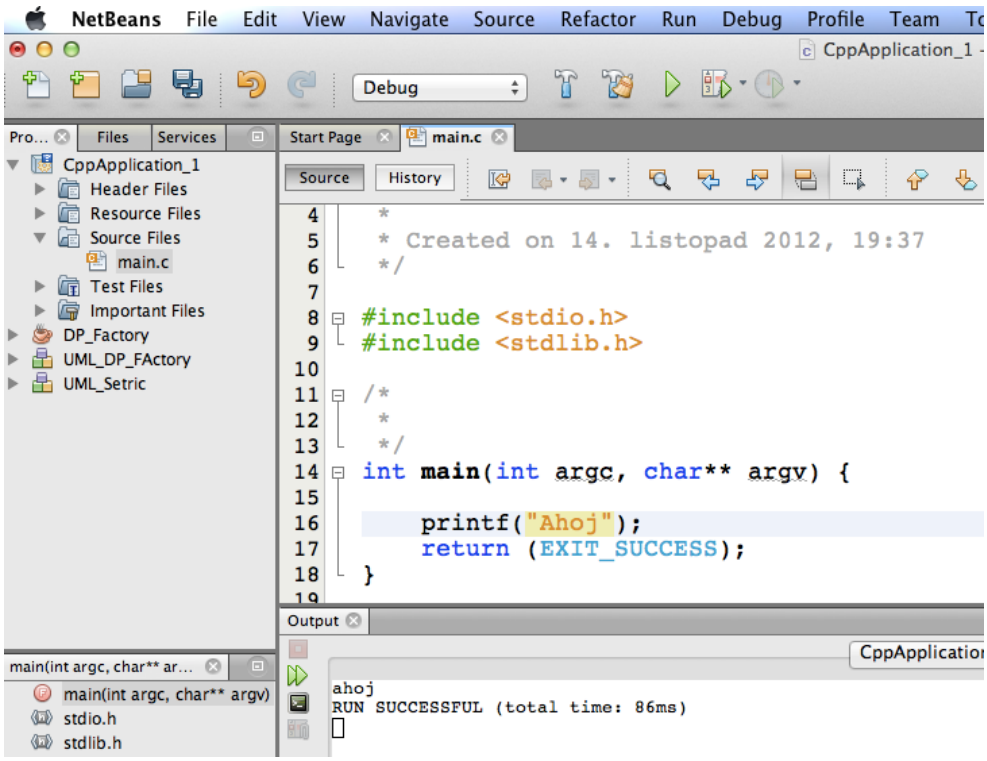


Nejprve určíme název a umístění projektu a pak je nutné vybrat vhodný překladač, tedy překladač jazyka C.



V dalším kroku zvolím typ aplikace. Pro studium stačí volit konzolovou aplikaci.

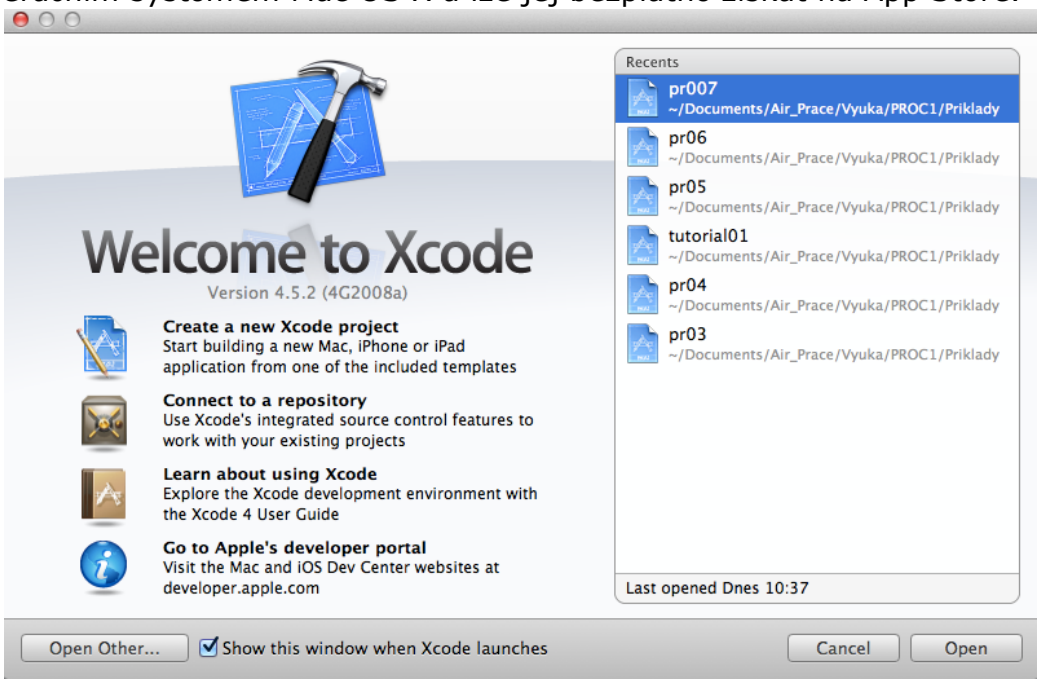




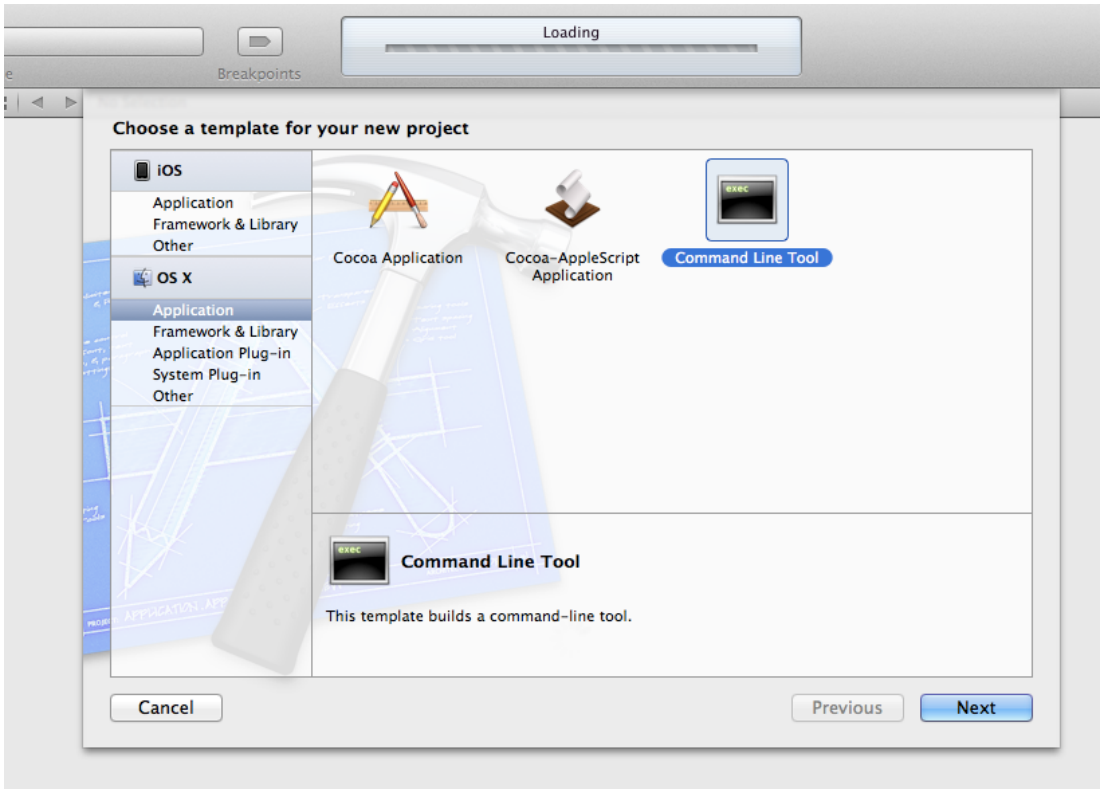
Výstupy aplikace lze sledovat v okně Output.

Vytvoření projektu v XCode

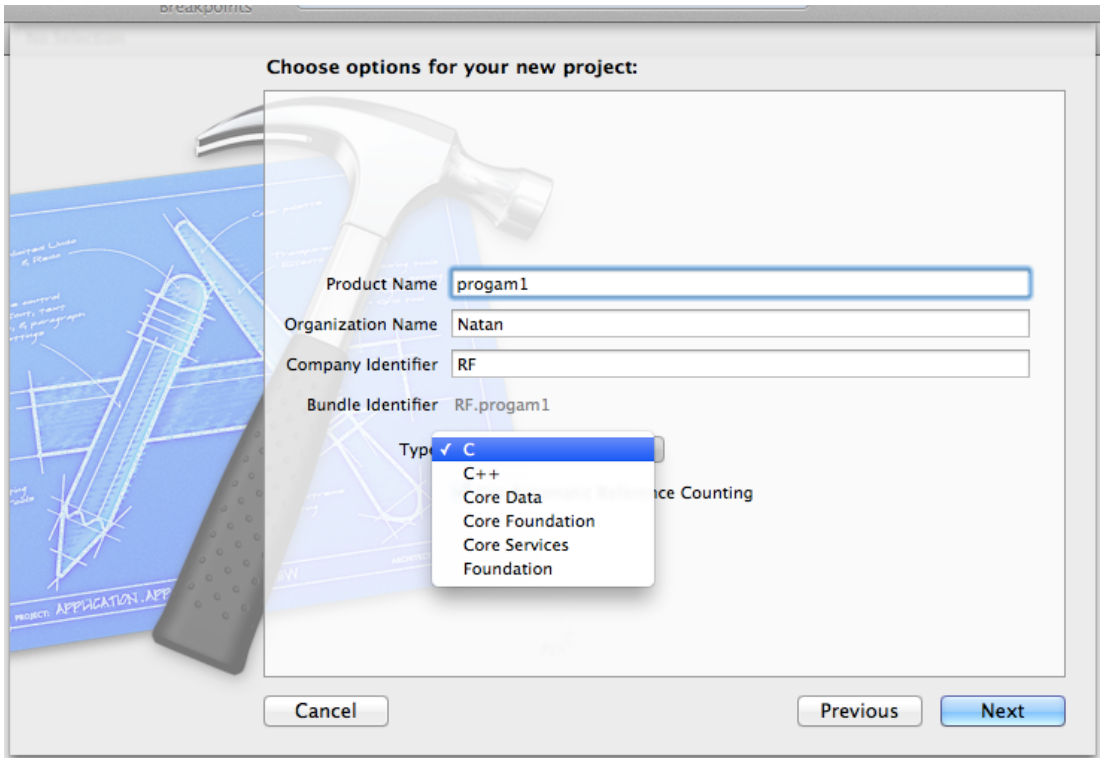
Vývojové prostředí XCode je určeno pro počítače firmy Apple s operačním systémem Mac OS X a lze jej bezplatně získat na App Store.



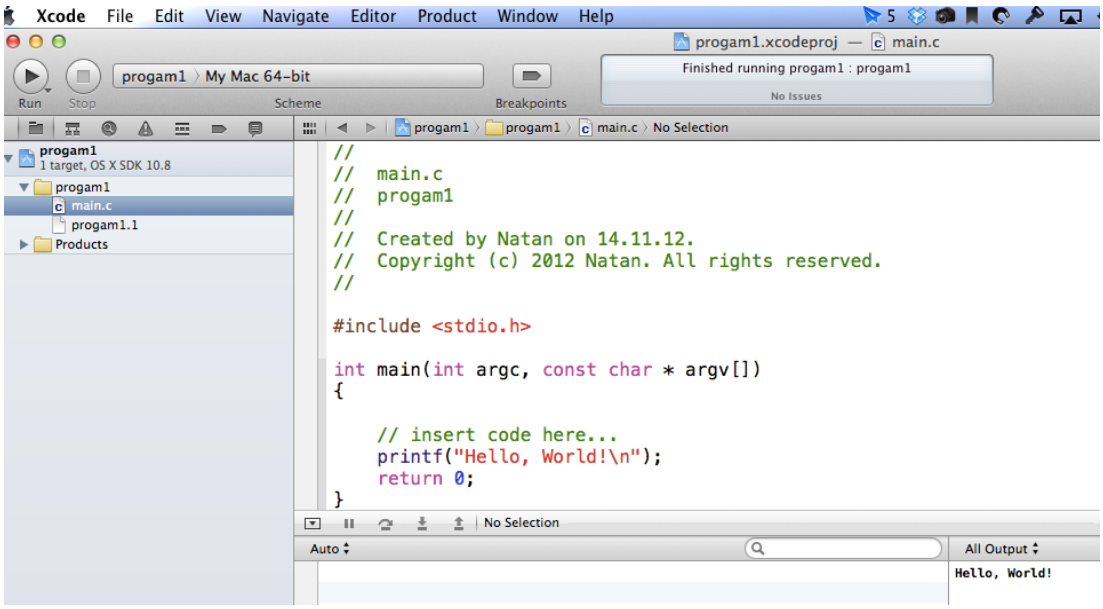
Pro vývoj programů v rámci výuky stačí zvolit aplikaci pro příkazový řádek.



Jakmile určíme jméno a umístění projektu, je potřeba zvolit překladač jazyka C.

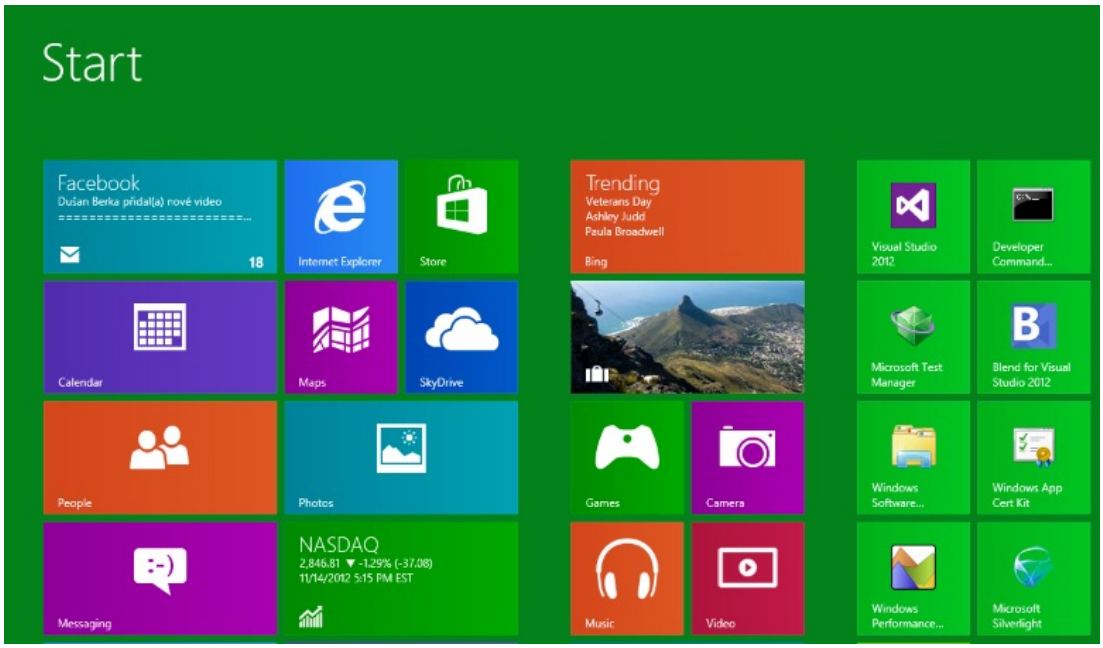


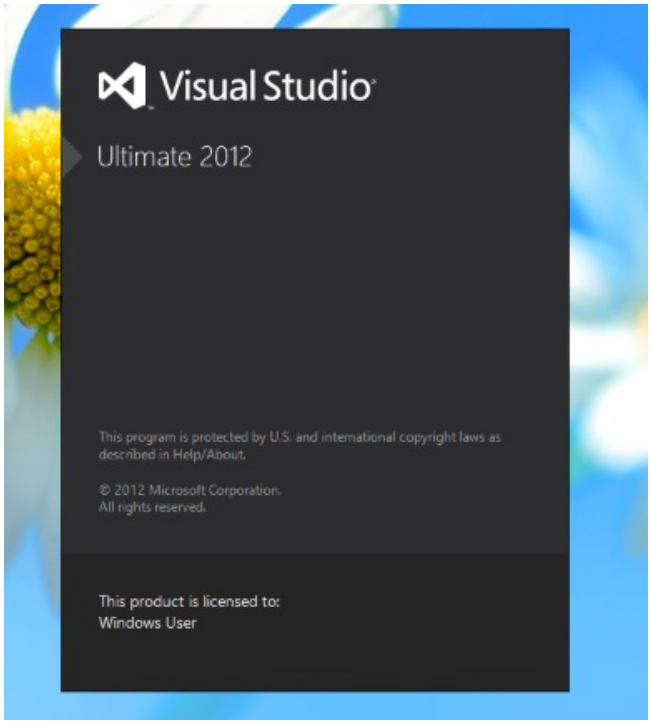
Výstupy programů lze podobně jako u NetBeans sledovat v okně Output.



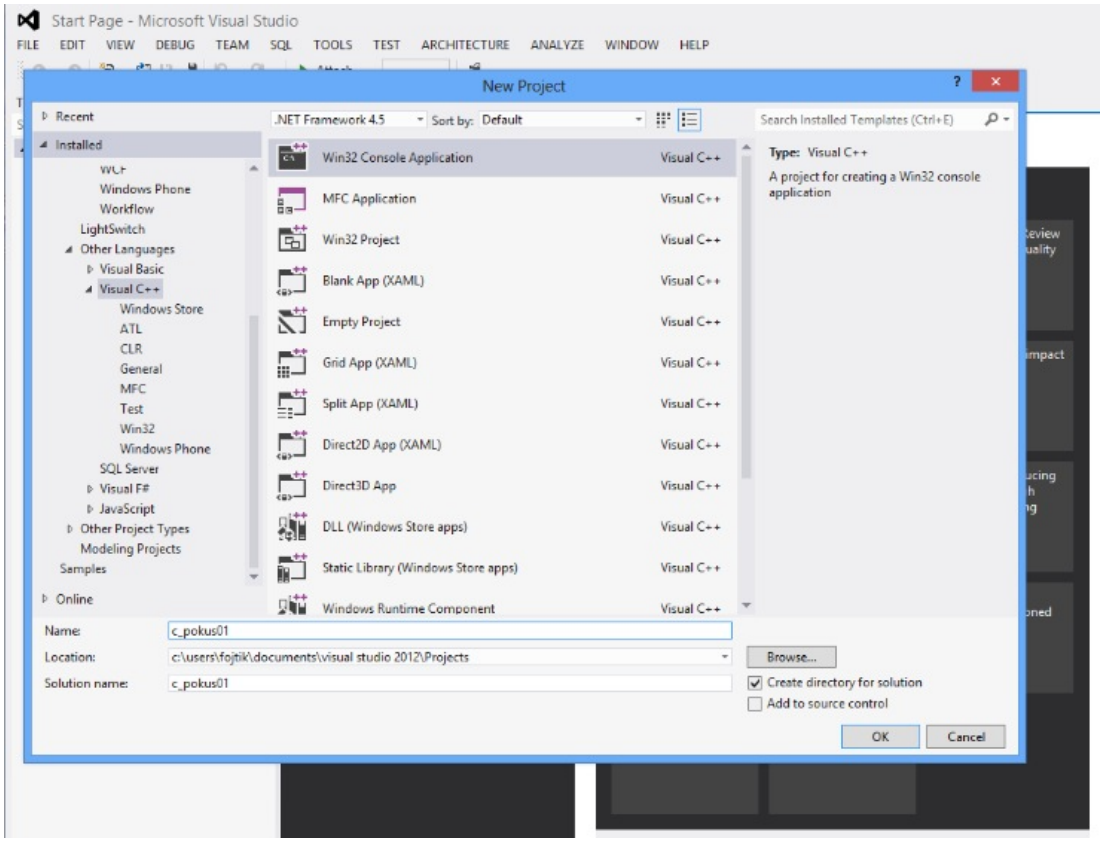
MS Visual Studio

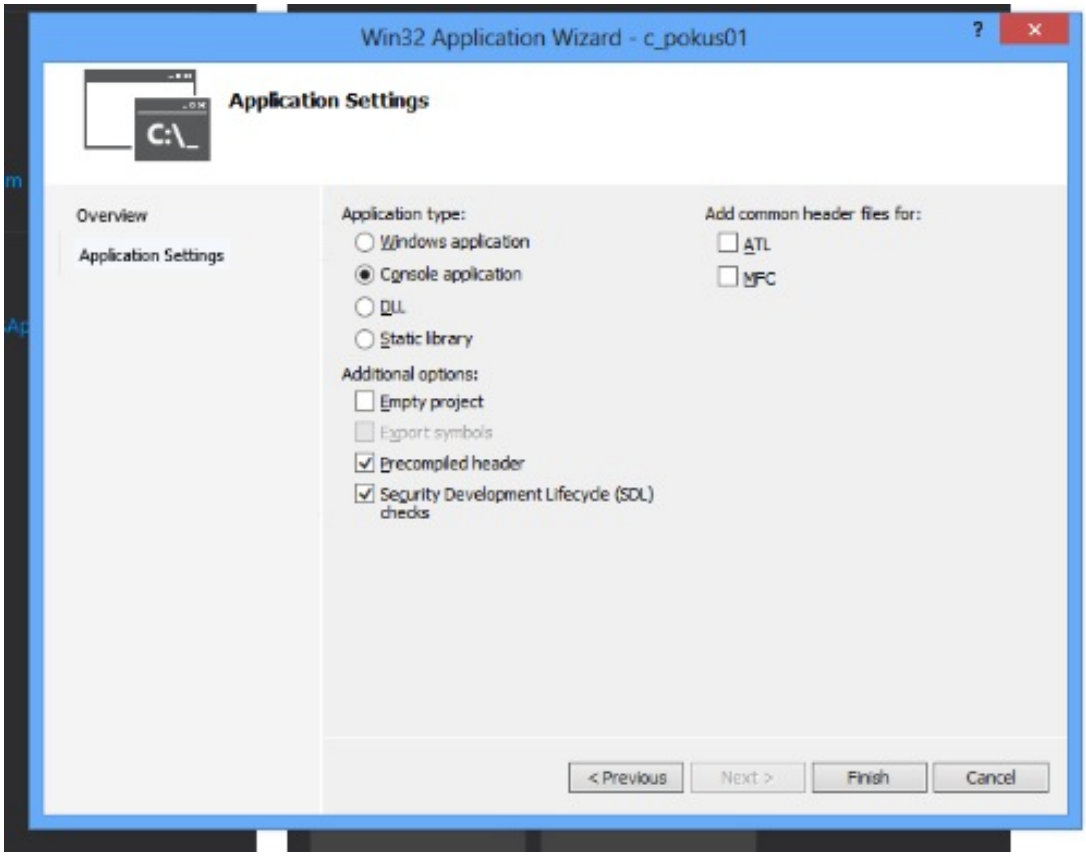
Vývojové prostředí firmy Microsoft lze rovněž využít pro tvorbu programů v jazyku C.



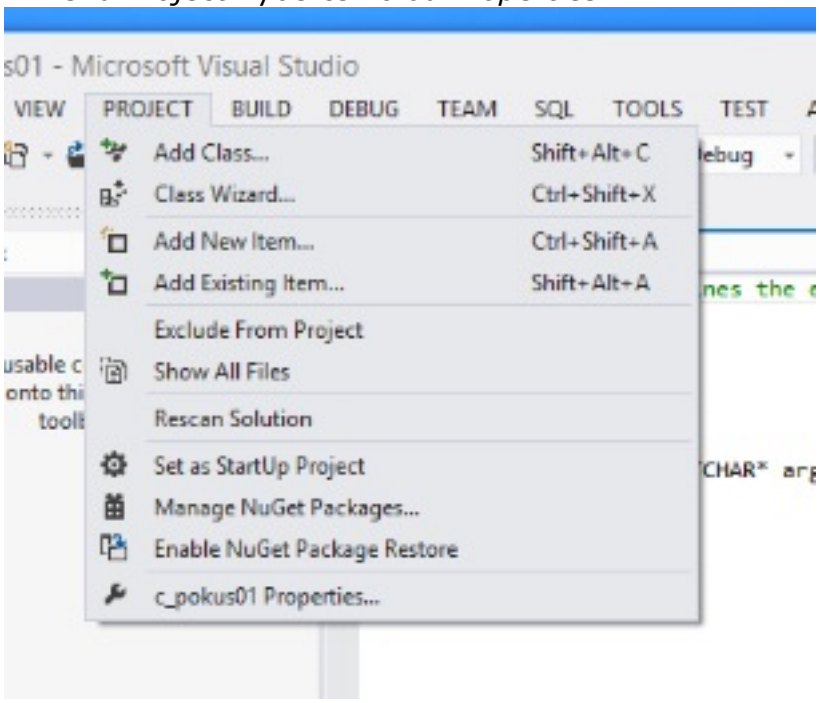


Vytvořte nový projekt jako konzolovou aplikaci typu Win 32 Console Application.

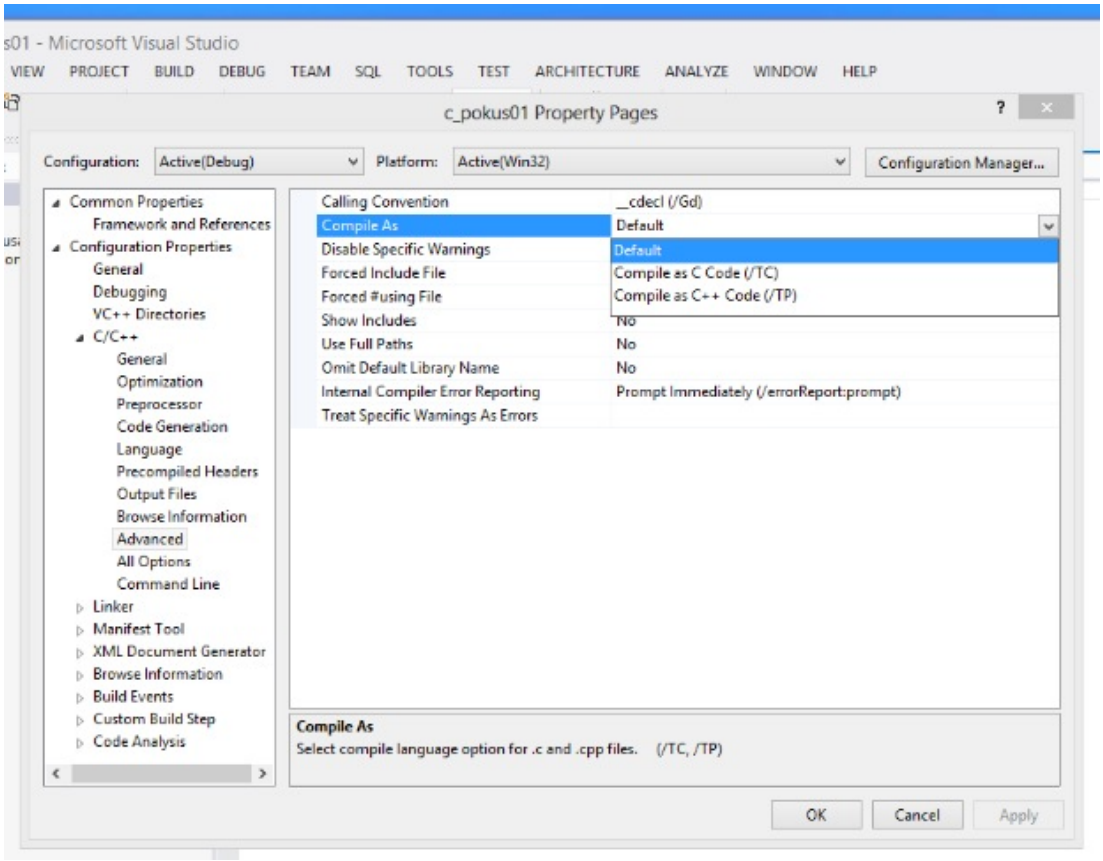




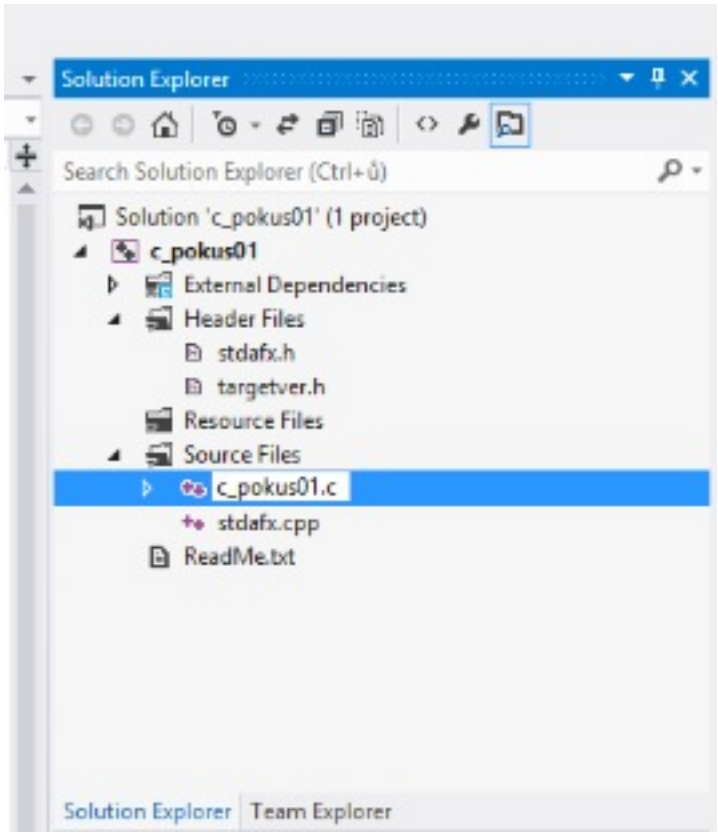
V menu *Project* vyberte volbu *Properties*.



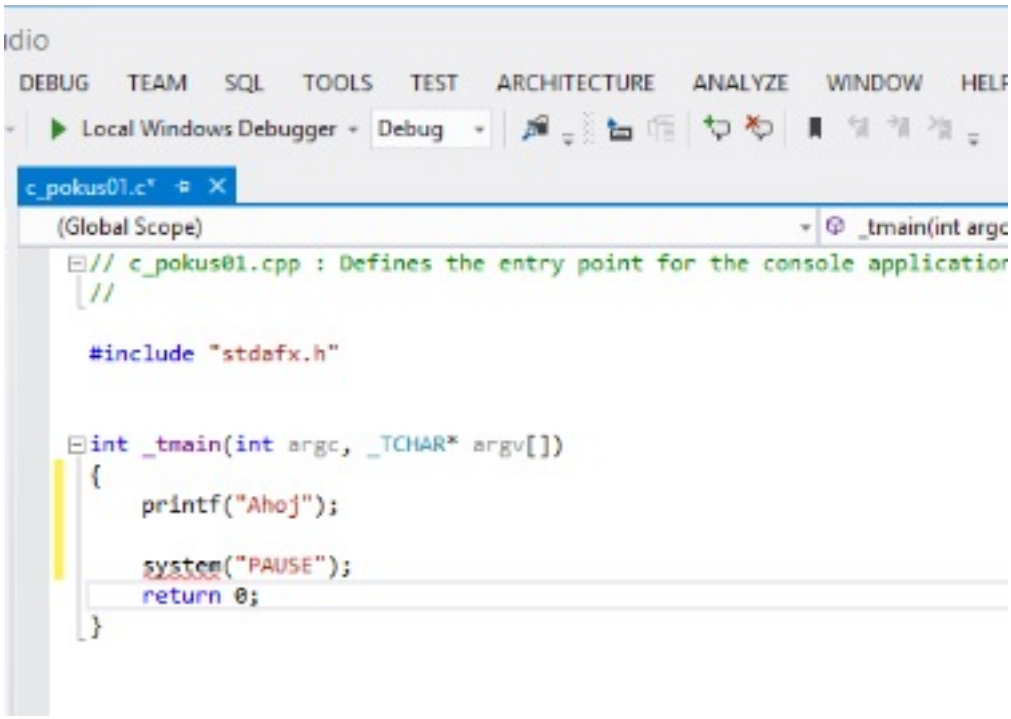
V nastavení kompilátoru změňte *Deafault* na *Compile as C Code*,

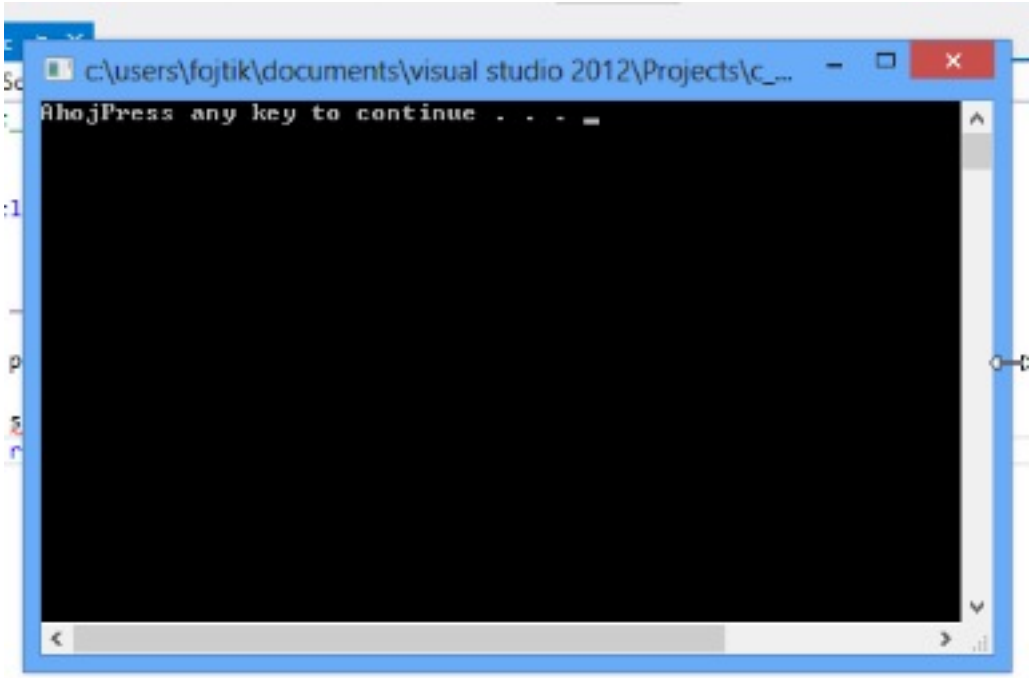


Přejmenujte hlavní zdrojový soubor tak, že příponu *cpp* změňte na příponu *c*.



Ve funkci *_tmain* před příkaz return vložte příkaz *system("PAUSE")*, který zajistí, že se okno příkazového řádku (konzola) zavře až po kliknuti na libovolnou klávesu.





3. ZÁKLADNÍ ŘÍDÍCÍ STRUKTURY

V každém programovacím jazyce je potřeba umět definovat základní řídící struktury, jako jsou bloky příkazů, podmínky, cykly. Kromě toho se v této lekci naučíte základní funkce pro formátovaný vstup a výstup dat v programu.

Budete:

- umět vytvářet bloky příkazů
- umět zapisovat podmínky
- umět definovat a řídit cykly s podmínkou na začátku i na konci
- používat jednoduché funkce pro čtení a zápis hodnot do proměnných

Klíčová slova této kapitoly:
podmínka, cyklus, blok příkazů, funkce

Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Vstup a výstup

Protože programovací jazyk C má malé jádro, musí většinu funkcí připojovat. K tomu je však potřeba do zdrojového souboru přidat příslušnou hlavičku funkce, které jsou nejčastěji umístěny v hlavičkových souborech s příponou H. Pro formátovaný vstup a výstup musíme přidat do programu příkaz:

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

Funkce, které formátovaný vstup a výstup zajišťují jsou:

pro vstup `scanf( )`

pro výstup `printf( )`

První parametr, který je ohraničen úvozovkami, určuje formát vstupu či výstupu. Dále následují jména proměnných, která se načítají či vypisují.

Příklad:

```
scanf("%d", &a); /* načte dekadické celé číslo a uloží na
adresu proměnné 'a' */
printf("%d", a); /* vypíše v dekadickém tvaru obsah proměnné
'a' */
```

Příklad:



Program načte z klávesnice dvě celá čísla, uloží jejich hodnoty do proměnných. Proměnné vypíše. Dále vypíše jejich součin a rozdíl.

```
#include <stdio.h>
int main( )
{
 int a,b;

 scanf("%d", &a);
 scanf("%d", &b);
```

```
 /* nesmite zapomentou uvest adresni operator '&'   jinak se
nactena hodnota ulozi "nekam" do pameti   a ne na adresu
urcene promenne */

 printf("a = %d ", a);
 printf("b = %d", b);
 /* nebo printf("a = %d b = %d", a,b);  */

 printf("Soucin cisel %d a %d je %d\n",a, b, a * b);
 printf("Rozdil cisel %d a %d je %d\n",a, b, a - b);

 return 0;
}
```

Formátové specifikace:

c	znak
d	desítkové číslo typu signet int
ld	desítkové číslo typu long int
u	desítkové číslo typu unsignet int
lu	desítkové číslo typu unsignet long int
f	float
Lf	long double
lf	double
x	hexadecimální číslo (např. 1a2)
X	hexadecimální číslo (např. 1A2)
o	osmičkové číslo
s	řetězec

Vstup a výstup jednotlivých znaků je zajištěn pomocí funkcí `getchar( )` a `putchar( )`.

Příklad: program načte z klávesnice znak a následně jej vypíše na obrazovku

```
#include <stdio.h>
int main( )
{
 int znak; /* obe funkce pracuji s promennymi typu int */

 znak = getchar( );
 putchar( znak );

 return 0;
}
```

Bloky příkazů

Bloky příkazů jsou ohraničeny složenými závorkami { a }. V programovacím jazyku C je možné na začátku každého bloku definovat lokální proměnné.

Příklad:

```
int main()
{
 int a;
```

```
 {
 int b;
 }
 /* zde již proměnná 'b' nelze použít */
}
```

Podmínky

Nejdříve je potřeba seznámit se s následujícími operátory:

rovnost == (dvě znaménka rovná se)
nerovnost !=
logický součin &&
logický součet ||
negace !
menší <
menší nebo rovno <=
větší >
větší nebo rovno >=

Pro správnou práci z uvedenými operátory je potřeba důkladně si prohlédnout tabulku priority jednotlivých skupin operátorů (například v nápovědě překládače).

Podmíněný příkaz má tvar:

```
if (vyraz) prikaz;
```

nebo

```
if (vyraz) prikaz1;  /* Zde musí být středník! */
 else prikaz2;
```

V případě bloku příkazu by předcházející zápisy vypadaly následovně:

```
if (vyraz)
{
 prikaz1;
 prikaz2;
}     /* Není středník! */
else
{
 prikaz3;
 prikaz4;
}
```

Navíc jazyk C umožňuje podmíněný výraz, který má tvar

```
(vyraz) ? prikaz1 : prikaz2;
```

Jedná se o ternární operátor, který může být přiřazen nějaké proměnné. Příklad:

```
int x;
x = (x > 10) ? 1 : 0;
```

Příklad:



Načtěte znak a v případě, že se jedná o malý znak, jej převeďte na znak velký.

```
#include <stdio.h>
int main()
{
```

```
 int znak;
 znak = getchar( );
 znak =(znak >= 'a' && znak <= 'z') ? znak-('a'-'A') : znak;
 putchar(znak);
 return 0;
}
```

Častá chyba!

Mezi časté chyby začátečníků při programování v C je, že vytvoří následující podmínku:

```
if (x=10) prikaz;
```

místo

```
if (x==10) prikaz;
```

První zápis je sice syntakticky správně, ale znamená:

```
if ((x=10)!=0) prikaz;
```

Vícenásobná podmínka

Pro vícenásobné větvení používá programovací jazyk C příkaz **switch**, který se zapisuje například v následujícím tvaru:

```
switch (vyraz)
{
 case hodnota1: prikaz1; break;
 case hodnota2: prikaz2; break;
 case hodnota3: prikaz3; break;
 case hodnota4: prikaz4; break;
 default : prikaz5; break;
}
```

Výraz, podle něhož se vybírají jednotlivé větve, musí být jednoznačně typu int. Pozor, příkaz **break** je velmi důležitý. Jeho neuvedení znamená, že se provádějí všechny příkazy od větve s hledanou hodnotou , až po první příkaz **break**. To znamená, že v níže uvedeném příkladu se při vyhodnocení výrazu na hodnotu 2, se provedou příkazy prikaz2, prikaz3 a prikaz4. Větev **default**, která nemusí být nutně uvedena jako poslední, se vybírá v případě, že žádná z ostatních větví nevyhovuje.

```
switch (vyraz)
{
 case 1: prikaz1;
 case 2: prikaz2;
 case 3: prikaz3;
 case 4: prikaz4; break;
 default : prikaz5; break;
}
```

Cykly

Podobně jako v jiných jazycích lze pracovat v jazyce se dvěma základními druhy cyklu. Cyklus s podmínkou na konci vypadá následovně:

```
do
{
 prikazy;
} while (vyraz_podminka);
```

Naproti tomu cyklus s podmínkou na začátku používá následující zápis:

```
while (vyraz_podminka)
{
 prikazy;
}
```

V obou případech se do těla cyklu vrací, je-li podmínka vyhodnocena jako splněna (srovnej s cyklem s podmínkou na konci v Pascalu).

Pro cyklus s podmínkou na začátku, u kterého je znám počet opakování, se používá zápisu:

```
for (i = 1; i <= 10; i++) prikazy;
```

Kde první část závorky představuje inicializaci počáteční hodnoty proměnné, která představuje čítač cyklu. Prostřední část vyjadřuje podmínku, při které se cyklus opakuje, a poslední část vyjadřuje krok o kolik se čítač mění. Předcházející zápis by samozřejmě šel napsat i následovně:

```
i = 1;
while (i <= 10)
{
      prikazy;
i++;
}
```

Všechny cykly jdou ukončit nebo přerušit pomocí příkazů **break** a **continue**. Kdy **break** ukončuje nejvnitřnější cyklus. Naopak **continue** přerušuje provádění těla cyklu, ale vrací se zpět k podmínce cyklu a teprve podle jejího vyhodnocení se rozhodne, zda bude cyklus pokračovat či nikoliv.

```
i = 1;
a = 1;
while (i <= 10)
{
 a = a * i;
 if (a <= 20) break;
 i++;
}
```

Příkazy skoků

Kromě skokových příkazů **break** a **continue**, které určeným způsobem upravují provádění cyklu, existují další příkazy na odskok z daného místa programu na jiné. Známým příkazem, hlavně z jiných programovacích jazyků, je příkaz **goto**. Tomu je ale vhodné, pokud je to jen trochu možné, se raději vyhnout, neboť znepřehledňuje zdrojový kód.

Příklad použití:

```
......
goto navesti;
/* odtud program odskočí až do míst uvozených návěštím */
......
navesti:   /* zde bude program pokračovat */
prikazy;
```

Dalším skokovým příkazem je **return**, který ukončí provádění dané funkce, ve které je zapsán a vrátí určenou hodnotu. Zatím jsme tento příkaz používali u funkce main( ).

Příklady



Příklad č. 1

Načtěte znak a vypište jeho desítkovou, osmičkovou a hexadecimální hodnotu.

```
#include <stdio.h>

int main()
{
    int znak;

    scanf("%c", &znak);
    printf("%d  %o %x ", znak, znak, znak);

    return 0;
}
```



Příklad č. 2

Vypište hodnoty ASCII tabulky od 33 do 128 znaku ve znakové, desítkové, hexadecimální a oktalové podobě.

```
#include <stdio.h>

int main()
{
    int i;

    for (i = 33; i <= 128; i++)
    printf("%5c- %4d%4o%4x ", i, i, i, i);
    /* cisla mezi % a formatem vypisu znamenaji, kolik ma byt
                  v danem pripade vytisteno znaku   */

    return 0;
}
```



Příklad č. 3

Načtěte tři celá čísla typu int a vypište je na obrazovku sestupně.

```
#include <stdio.h>

int main()
{
  int a,b,c;

  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%d", &c);

  if ( a > b )
   if ( b > c ) printf(" %d > %d > %d ", a, b, c);
```

```
    else
      if ( a > c ) printf(" %d > %d > %d ", a, c, b);
       else printf(" %d > %d > %d ", c, a, b);
  else
    if (a > c ) printf(" %d > %d > %d ", b, a, c);
     else
        if ( c > b) printf(" %d > %d > %d ", c, b, a);
          else printf(" %d > %d > %d ", b, c, a);
    return 0;
}
```

Příklad č. 4



Vypočtěte součet celých čísel od načtené dolní po načtenou horní hranici. Výsledek vypište na obrazovku.

Příklad upravte tak, aby součet číselné řady neobsahoval čísla, která jsou bezezbytku dělitelná zadaným číslem.

```
#include <stdio.h>

int main()
{
  int dolni, horni;   /* hranice ciselne rady */
  int i;   /* citac cyklu */
  int pom; /* pomocna promenna pro vymenu dolni a horni */
  long int suma=0L;   /* nutne vynulovat */

  scanf("%d", &dolni); /* nezapomenout adresni operator */
  scanf("%d", &horni);

  if (dolni > horni )
  {
    pom = dolni;
    dolni = horni;
    horni = pom;
  }
  for (i = dolni; i <= horni; i++) suma += i;
  printf("Soucet ciselne rady od %d do %d je
%ld",dolni,horni,suma);

  return 0;
}
```

Příklad č.5



```
/*
 * Jednoduché operátory
 * Krokujte program a sledujte, jak se mění proměnné
 */

int main()
{
  int a, b=13, c=5;
  float f, g=26.1, h=12.0;

  b++;
  c--;
```

```
  a=b+c;
  a=b/c;
  a=a%c;
  a=b+++c;
  a=--b+c;
  f=a/c;
  f=(float)a/c;
  f=g/h;
  f=(int)g/h;
  f=(int)g/(int)h;
  a=10;
  a-=c;
  a*=b;

  return 0;
}
```



Příklad č.6

```
/*
 * Jednoduché operátory a konverze datových typů.
 * Předem si na jednotlivé řádky do poznámek zapište,
 * jaké hodnoty budou proměnné mít.
 * Krokujte program a sledujte, jak se mění proměnné.
 * Označte "nebezpečné" kroky, kdy konverze může změnit
 * hodnoty proměnných.
 */

int main()
{
  unsigned int a=35458;
  int b=10;
  float f=3.14;
  long l=100;

  a++;
  a-=b;
  a+=f;
  l*=b;
  b=a;
  l=f;
  f*=1000000000;
  l=f;

  return 0;
}
```



Příklad č.7

```
/*
 * Načtěte znak a v případě, že se nejedná o písmeno ani
 * číslici vypište na obrazovku text "Nejedna se o pismenko
 * ani cislici"
 */

#include <stdio.h>

int main()
```

```
{
  char znak;

  znak = getchar();

  if ((znak>='a' && znak<='z') || (znak>='A' && znak<='Z') ||
(znak>='0' && znak<='9'));
   else printf("Nejedna se o pismenko ani cislici\n");

  /* nebo */
  if (znak<'a' || znak>'z')
    if (znak<'A' || znak>'Z')
      if (znak<'0' || znak>'9')
        printf("Nejedna se o pismenko ani cislici\n");

  return 0;
}
```

Příklad č.8



```
/*
 * Načtěte deset celých čísel typu int a vypočtěte jejich
 * součet a průměr.
 * Vypište průměr na dvě desetinná místa.
 */

#include <stdio.h>

int main()
{
  int i;
  int cislo;
  long suma=0; /* nezapomeňte vynulovat */
  float prumer;

  for (i=1;i<=10;i++)
  {
    printf("Zadej cele cislo: ");
    scanf("%d",&cislo);
    suma += cislo;
  }
  prumer = (float)suma / (i-1);

  printf("Soucet nactenych cisel je %ld\n",suma);
  printf("Prumer z nactenych cisel je %10.2f\n",prumer);

  return 0;
}
```

Příklad č.9



```
/*
 * Obměna příkladu č.4
 * Načtěte celá čísla typu int a vypočtěte jejich součet
 * a průměr. Načítaní se zastaví v okamžiku, kdy načtu nulu.
 * Nulu do průměru nepočítejte!
 * Vypište průměr na dvě desetinná místa.
 */
```

```
#include <stdio.h>

int main()
{
  int i=0; /* nezapomeňte vynulovat */
  int cislo;
  long suma=0; /* nezapomeňte vynulovat */
  float prumer;

  do
  {
    printf("Zadej cele cislo: ");
    scanf("%d",&cislo);
    suma += cislo;
    i++;
  }while (cislo != 0);

  if ((i-1) != 0)  /* pozor na deleni nulou */
  {
      prumer = (float)suma / (i-1);

      printf("Soucet nactenych cisel je %ld\n",suma);
      printf("Prumer z nactenych cisel je %10.2f\n",prumer);
  }

  return 0;
}
```

Shrnutí učiva



• Pro formátovaný vstup a výstup musíme přidat do programu příkaz:
```
#include <stdio.h>
```
Funkce, které formátovaný vstup a výstup zajišťují jsou:

pro vstup `scanf( )`

pro výstup `printf( )`

• Bloky příkazů jsou ohraničeny složenými závorkami { a }. V programovacím jazyku C je možné na začátku každého bloku definovat lokální proměnné.

• Podmíněný příkaz má tvar:
```
 if (vyraz) prikaz;
```
nebo
```
 if (vyraz) prikaz1;  /* Zde musí být středník! */
 else prikaz2;
```
Navíc jazyk C umožňuje podmíněný výraz, který má tvar
```
(vyraz) ? prikaz1 : prikaz2;
```
• Pro vícenásobné větvení používá programovací jazyk C příkaz **switch**, který se zapisuje například v následujícím tvaru:
```
switch (vyraz)
{
case hodnota1: prikaz1; break;
case hodnota2: prikaz2; break;
case hodnota3: prikaz3; break;
      case hodnota4: prikaz4; break;
      default :  prikaz5; break;
}
```
Výraz, podle něhož se vybírají jednotlivé větve, musí být jednoznačně typu int.

• Cyklus s podmínkou na konci vypadá následovně:
```
do
{
      prikazy;
} while (vyraz_podminka);
```
Naproti tomu cyklus s podmínkou na začátku používá následující zápis:
```
while (vyraz_podminka)
{
      prikazy;
}
```
V obou případech se do těla cyklu vrací, je-li podmínka vyhodnocena jako splněna.

Pro cyklus s podmínkou na začátku, u kterého je znám počet opakování, se používá zápisu:
```
for (i = 1; i <= 10; i++) prikazy;
```
Kde první část závorky představuje inicializaci počáteční hodnoty proměnné, která představuje čítač cyklu. Prostřední část vyjadřuje podmínku, při které se cyklus opakuje, a poslední část vyjadřuje krok o kolik se čítač mění.

• Příkazy skoku v jazyku C: **break, continue, goto, return**.

4. FUNKCE

Cílem lekce naučit se vytvářet funkce v jazyce C.

Budete:

- umět zapisovat hlavičky funkcí
- umět vytvářet rekurzivní funkcí
- umět definovat parametry funkcí

Klíčová slova této kapitoly:
Hlavička funkce, návratový typ funkce, parametry funkce



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Program v jazyku C vždy obsahuje alespoň jednu funkci, která se jmenuje `main()`. Jejím voláním program začíná a končí při jejím ukončení. Tato funkce je v programu právě jedna! Ostatní funkce slouží k vytváření podprogramů. Na rozdíl od některých jiných programovacích jazyků (Pascal), nelze v rámci funkce definovat další funkci. Deklarace funkce se děje pomocí její hlavičky a vypadá například takto:

```
long Rada(int dol, int hor);
navratovy_typ JmenoFunkce(typ p1, typ p2);
```

Deklarace funkcí se umisťují před hlavní funkci `main()` nebo ještě lépe do příslušných hlavičkových souborů. Za jménem funkce se vždy píší kulaté závorky a to i přesto, že funkce nemá ani jeden parametr! Při deklaraci není nutné u parametrů uvádět jejich jména - identifikátory, stačí pouze jejich datový typ.

```
long Rada(int, int);
```

Naopak definice jednotlivých funkcí (tedy jejich kód) se obvykle uvádí až za hlavní funkci. Je možné kód funkcí uvádět již před hlavní funkci **`main`**`()`, ale z hlediska přehlednosti se první způsob jeví jako vhodnější. Definice funkce může pak vypadat například následovně:

```
long Rada(int dol, int hor)/*zde nesmí být středník */
{
 int i; /* lokální proměnné */
 long vys=0;

 for (i=dol; i<=hor; i++) vys += i;
 return vys; /* ukončení funkce a vracení hodnoty */
} /* funkce vypočte součet čísel od čísla dol do hor */
```

Volání funkce se pak zapíše například takto:

```
x = Rada(1,10);
```

Rekurzivní funkce

Funkce mohou být rovněž rekurzivní. Ukážeme si to na klasickém příkladu výpočtu faktoriálu.



```
#include <stdio.h>

int Faktorial(int cis)
{
   return (cis <=1) ? 1 : cis * Faktorial(cis-1);
}

int main()
{
      int x=5,vys;

      vys=Faktorial(x);
      printf("Faktorial cisla %d je %d\n",x,vys);

      return 0;
}
```

Rekurze může být:

- *přímá* - rekurzivní funkce volá sama sebe
- *nepřímá* - například funkce Fce1 volá ve svém těle funkci Fce2 a ta zpětně volá Fce1.



Častá chyba!

Nezapomeňte, že rekurze musí mít jasně stanovenou podmínku pro ukončení. V opačném případě dojde k přetečení zásobníku.

Funkce návratovým typem void

Procedury v jazyku C sice neexistují, ale některé typy funkcí se defakto jako procedury chovají. To se děje díky datovému typu s názvem **void**, což je tzv. prázdný datový typ. Potom nemusíme ve funkci uvádět příkaz **return**. V případě, že funkci v určitém případě potřebuje ukončit dřív než na jejím konci, uvedeme v daném místě příkaz **return**, ale bez návratové hodnoty. Pokud je datový typ **void** uveden v závorkách za jménem funkce, znamená to, že funkce nemá žádný parametr. Pozor! Pokud neuvedeme v kulatých závorkách hlavičky funkce žádný datový typ, předpokládá kompilátor, že funkce obsahuje parametr typu **int**. Nejedná se tedy o deklaraci funkce bez parametrů!

Příklad:



```
void Tisk(void)
{
  printf(" At zije programovaci jazyk C ");
}

void NejakaFunkce();
/* Pozor! Tato funkce má jeden parametr a to typu int */
```

Parametry funkcí

Parametry funkcí jsou v jazyce C předávány pouze hodnotou. To znamená, že nemohou být skutečné parametry měněny uvnitř funkce. Tento nedostatek je řešen pomocí pointerů.

Typ skutečného parametrů funkce by měl souhlasit s typem formálního parametru. Jinak se provede typová konverze na typy uvedené ve funkčním prototypu.

Příklad:



```
float Exponent(float x, int n)
{
 //kód funkce
}

int main()
{
 int a=10, b=3;
 float vys;
 //...
 vys = Exponent(a,b);
 /*první parametr kude konvertován na typ float*/
 vys = Exponent(a);
 /*Chyba! Chybí druhý parametr!*/
 return 0;
}
```

Kontrolní úkol:

Zopakujte si, jakým způsobem je obsazován zásobník v paměti, při ukládání mezivýpočtů při volání rekurzivní funkce Faktorial.

Příklady

Příklad č.1

Vytvořte funkci, která vrátí menší ze dvou parametrů funkce.



```
#include <stdio.h>

int Mensi(int, int);  /* deklarace funkce */

int main(void)
{
  int a,b;

  printf("Zadej první cele cislo: ");
  scanf("%d", &a);
  printf("Zadej druhe cele cislo: ");
  scanf("%d", &b);

  printf("Vetsi ze dvou cisel %d a %d je %d\n",a, b,
Mensi(a,b));
  return 0;
}

int Mensi(int x1, int x2)
{
```

```
  return (x1 < x2) ? x1 : x2;
}
```

Příklad č.2



Napište funkci, která vypočte n-tou mocninu čísla x. (x^n)

```
#include <stdio.h>

float Mocnina(int, int);    /* deklarace funkce */
int Abs(int);

int main(void)
{
  int mc,mt;

  printf("Zadej mocnenec: ");
  scanf("%d", &mc);
  printf("Zadej mocnitel: ");
  scanf("%d", &mt);

  printf(" %d-ta mocnina cisla %d je %f\n",cis,
Mocnina(mc,mt));
  return 0;
}

float Mocnina(int x, int n)
{
  int i;
  float vys=1;

  for (i=1; i<=Abs(n); i++)
      vys *=x;
  if (n<0) vys = 1/vys;
  return vys;
}

int Abs(int c)
/* funkci je mozne definovat az za funkci Mocnina, nebot je
nazacatku jeji deklarace */
{
  return (c < 0) ? c * (-1) : c;
}
```



Příklad č.3

Vytvořte program s funkcí, která zjistí největší společný dělitel dvou čísel. Použijte například Euklidova algoritmu:

Obměňte program tak, že funkci napište pomocí rekurze.

```
#include <stdio.h>

int NSD(int, int);    /* deklarace funkce */

int main(void)
{
  int a, b;

  printf("Zadej prvni cele cislo: ");
```

```
  scanf("%d", &a);
  printf("Zadej druhe cele cislo: ");
  scanf("%d", &b);

  printf(" Nejvetsi spolecn delitel cisel %d a %d je
%d\n",a,b,NSD(a,b));
  return 0;
}

int NSD( int x, int y)
{
  while (x != y)
   if (x > y) x -= y;
     else y -= x;
  return x;
}
```

Příklad č.4



Vytvořte rekurzivní funkci pro výpočet sumy čísel v rozmezí od zadané *dolní* do *horní* hranice včetně. Například dolní hranice je 5, horní 11, pak suma je 5+6+7+8+9+10+11.

```
#include <stdio.h>

int Suma(int, int);

int main(void)
{
  int a, b;

  printf("Zadej dolni hranici: ");
  scanf("%d", &a);
  printf("\nZadej horni hranici: ");
  scanf("%d", &b);
  printf(" Soucet cisel od %d do %d je %d\n",a,b,Suma(a,b));
  return 0;
}

int Suma (int dol, int hor)
{
  if (dol == hor) return dol;
    else return (Suma(dol, hor-1) + hor);
}
```

Příklad č.5



Vytvořte funkci pro výpočet faktoriálu nejprve pomocí cyklu a pak i pomocí rekurze.

Shrnutí učiva



• Program v jazyku C musí obsahovat alespoň jednu funkci, která se jmenuje `main()`. Jejím voláním program začíná a končí při jejím ukončení. Tato funkce je v programu právě jedna! Ostatní funkce slouží k vytváření podprogramů. V rámci funkce nelze definovat další funkci.

Deklarace funkce:

```
navratovy_typ JmenoFunkce(typ p1, typ p2);
```

• Funkce mohou být rovněž rekurzivní.

```
int Faktorial(int cis)
{
   return (cis <=1) ? 1 : cis * Faktorial(cis-1);
}
```

• Parametry funkcí jsou v jazyce C předávány pouze hodnotou. To znamená, že nemohou být skutečné parametry měněny uvnitř funkce. Tento nedostatek je řešen pomocí pointerů.

Typ skutečného parametrů funkce by měl souhlasit s typem formálního parametru. Jinak se provede typová konverze na typy uvedené ve funkčním prototypu.

Rekurze

Rekurze je mechanizmus, kdy funkce volají samy sebe. Můžeme definovat rekurzi *přímo*, kde konkrétní funkce vola sama sebe. Druhou možností je rekurze *nepřímá*, kde funkce se volají vzájemně. Například funkce A volá ve svém kódu funkci B a naopak funkce B volá funkci A.

Základním předpokladem správně vytvořené rekurze je stanovit jednoznačnou podmínku ukončení. Je potřeba si uvědomit, že rekurze je relativně náročná na paměť. Při mnohonásobném rekurzivním volání může dojít až k vyčerpání přidělené paměti.



Příklad

Vytvořte funkci pro výpočet n-tého členu Fibonacciho řady, která je dána předpisem `fibo(0)=1, fibo(1)=1, fibo(n)=fibo(n-1)+fibo(n-2).`

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

int fibo(int n)
{
    if (n>1) return (fibo(n-1) + fibo(n-2));
     else return (1);
}

int main(int argc, char *argv[])
{
  int i, cislo;

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

  printf("Fibonacciho rada:\n");
  printf("%d",fibo(cislo));

  system("PAUSE");
  return 0;
}
```

Příklad



Sestavte rekurzivní funkci `combin(n,k)` pro výpočet hodnoty kombinačního čísla podle pravidel Pascalova trojúhelníka: `combin(n,0)=1, combin(n,n)=1, combin(n,k)= combin(n-1,k-1) + combin(n-1,k).`

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

int combin(int n, int k)
{
  if ((k>0) && (k<n))
     return(combin(n-1,k-1) + combin(n-1,k));
   else return 1;
```

```
}

int main(int argc, char *argv[])
{
  int n,k;

  printf("Zadej dve cele cisla: ");
  scanf("%d",&n);
  scanf("%d",&k);

  printf("Pascaluv trojuhelnik:\n");
  printf("%d",combin(n,k));

  return 0;
}
```

Příklad



Pomocí rekurze vyřešte problém hanojských věží. Jsou tři tyče. Na první jsou navlečeny kotouče tak, že největší je dole a nejmenší nahoře. Úkolem je přesunout všechny kotouče na třetí tyč. Při přesouvání je potřeba přenášet najednou jen jeden kotouč a větší kotouč se nesmí položit na menší.

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

void presun(int vyska, int zdroj, int cil, int pom)
{
  static int i=0;

  if (vyska>0)
  {
     presun(vyska-1, zdroj, pom, cil);
     i++;
     if (i % 2) printf("\n");
     printf("%d.krok - presun disk %d na %d ",i,zdroj,cil);
     if (!(i % 20))
     {
        printf("\n");
     }
     presun(vyska-1, pom, cil, zdroj);
  }
}

int main(int argc, char *argv[])
{
  int pocet;

  printf("Zadej pocet kotoucu: ");
  scanf("%d",&pocet);

  presun(pocet, 1,2,3);
  return 0;
}
```

5. PREPROCESOR

Cílem této výukové lekce je naučit se vytvářet a využívat příkazy preprocesoru. V lekci se dozvíte, jak dělat makra bez i s parametry, vkládat soubory či zapisovat příkazy pro podmíněný překlad.

Budete:

- umět vytvářet hlavičkové soubory
- umět vytvářet makra
- umět zpřehlednit své zdrojové kódy pomocí maker
- umět ladit program pomocí podmíněného programu

Klíčová slova této kapitoly:
Makra, podmíněný překlad, symbolické konstanty, preprocesor



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Preprocesor je nástroj, který upravuje zdrojový text před samotnou kompilací programu. Provádí vkládání hlavičkových souborů, podmíněný překlad, rozvíjí makra, vypouští komentáře. Příkazy preprocesoru začínají znakem "#".

Úpravy prováděné preprocesorem nejsou součástí jazyka C a provádí se ještě před kompilací. Veškeré úpravy se týkají pouze textu. To znamená, že jednu posloupnost znaků nahrazují posloupností jinou (jeden text zamění za druhý). Preprocesor nezná příkazy jazyka C, datové typy, proměnné atd.

Vkládání souborů

Soubory se do zdrojového textu vkládají pomocí příkazu `#include`. Za tímto příkazem se napíše jméno souboru. V případě, že se jméno souboru nachází mezi znaky menší větší, pak je hlavičkový soubor umístěn ve standardním adresáři (např. INCLUDE).

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

Je-li jméno hlavičkového souboru mezi uvozovkami, pak je jeho umístění v aktuálním nebo uvedeném adresáři.

```
#include "muj.h"
#include "c:\programy\funkce.h"
```

Makra bez parametrů

Makra bez parametrů se občas nazývají *symbolické konstanty*. Jejich účelem je nahrazení posloupnosti znaků jiným řetězcem. K vytvoření maker slouží příkaz `#define`.

```
#define MAX   100
```

```
#define TISK  { printf("Ahoj!\n");}
```

Kdekoliv se v programu objeví řetězec MAX bude nahrazen posloupnosti znaků 100. Pozor! Preprocesor skutečně nechápe hodnotu 100 jako číslo, ale jako posloupnost znaků. Jako číslo bude chápana až kompilátorem. Podobně bude pracováno s ostatními makry. Jména maker se většinou píší velkými znaky. Je-li tělo makra dlouhé a nevejde-li se na jeden řádek, uvede se na jeho konci obrácené lomítko.

```
#define TEXT Text, který se nevejde na jeden radek \
     a pokracuje na druhem radku.
```

Pozor! Makro se nerozvine uvnitř řetězce!

```
#define JMENO Karel Novak
printf("Jeho jmeno je JMENO");
```

Na obrazovce se objeví výpis "`Jeho jmeno je JMENO`" a neobjeví se "`Jeho jmeno je Karel Novak`", jak bychom možná mylně očekávali.

Makra s parametry

Makra mohou mít rovněž parametry. Na rozdíl od funkcí, se však parametrům u maker neurčuje datový typ.

```
#define soucin(a,b)    ((a)*(b))
```

Za jménem makra nesmí být mezera, jinak by preprocesor řádek chápal jako makro bez parametrů. Makra mohou v některých případech nahrazovat činnost funkcí. Na rozdíl od funkcí je provádění maker rychlejší, ale prodlužují zdrojový text. Je však mít stále na paměti, že preprocesor pracuje s tělem makra jako s řetězcem a ne s proměnnými a hodnotami určitého datového typu. Proto se rozvoj makra může provést jinak, než jsme předpokládali. Například, zapomeneme-li u předcházejícího makra závorky, můžeme se dostat do komplikací.

```
#define soucin(a,b)    a*b
```

předcházející makro se po volání

```
vys = soucin(x+3,y+z);
```

se rozvine následujícím způsobem

```
vys = x + 3 * y + z;
```

Je samozřejmé, že výsledek bude jiný, než programátor předpokládal.

Naproti tomu funkce

```
int FceSoucin(int a,int b)
{
 return (a*b);
}
```

se při následujícím volání vyhodnotí správně. Výrazy v parametrech se nejprve vyhodnotí a teprve pak vloží do těla funkce.

```
vys = FceSoucin(x+3,y+z);
```

Pokud vynecháme vnější závorky kolem výrazu, opět můžeme narazit na problémy.

```
#define soucin(a,b)    (a)*(b)
```

Při volání následujícího výrazu, se makro nevyhodnotí správně.

```
vys = 100/soucin(2,5);
```

Makro se rozvine následujícím způsobem

```
vys = 100 / 2 * 5
```

Výsledek bude mít hodnotu 250 místo 10!

Pozor! Makro neprovádí vyhodnocení svých parametrů před svým rozvinutím!

Makra s parametry nevyhodnocují typ parametrů, což se dá s úspěchem využít v okamžiku, kdy chci funkci, která řeší algoritmus pro více typů. Například chceme funkci, která vrátí absolutní hodnotu čísla. V jazyku C budeme muset napsat tolik funkcí, kolik je typů čísel a navíc každá funkce musí mít jiné jméno. Takže dostaneme např. funkce:

```
int Abs_i(int x);
long Abs_l(long x);
float Abs_f(float x);
```

...atd. Všechny funkce využívají stejný algoritmus.

Pomocí makra si situaci můžeme zjednodušit:

```
#define Abs(x) (((x) < 0) ? (x)*(-1) : (x))
```

Makro pak mohu aplikovat na jakýkoliv číselný typ.

Podmíněný překlad

Podmíněný překlad nám může značně ulehčit fázi ladění a testování programu. V rámci kódu se mohou objevit příkazy sloužící pouze pro ladění. Bez možnosti využít podmíněného překladu bychom museli ladící části ručně vymazat nebo vložit do poznámek. Takováto činnost je však velmi náročná a zdlouhavá. Pomocí příkazů preprocesorů však můžeme ladící části používat jen v případě ladění, aniž bychom museli zasahovat do zdrojového textu. Ladící kód zůstává neustále ve zdrojovém souboru, kdykoliv se k němu můžeme znovu vrátit a zbytečně nám neprodlužuje program.

Možnosti zápisu podmíněného překladu jsou následující:

```
#if konstantni_vyraz
 prikazy
#else  /* tato část jde vynechat */
 prikazy
#endif

#ifdef MMM /*jestliže je definovano makro MMM */
 prikazy1;
#else
 prikazy2;
#endif

#if defined MMM /*jestliže je definovano makro MMM */
 prikazy1;
#else
 prikazy2;
```

```
#endif

#ifndef MMM /*jestliže není definovano makro MMM */
 prikazy1;
#else
 prikazy2;
#endif

#ife !defined MMM
/*jestliže není definovano makro MMM */
 prikazy1;
#else
 prikazy2;
#endif
```

Hlavičkové soubory

Hlavičkové soubory obsahují deklarace funkcí (funkční prototypy) a deklarace proměnných, datových typů, konstant, maker. Proti vícenásobnému vložení určitého hlavičkového souboru do zdrojového textu je potřeba opatřit text příkazy podmíněného překladu:

```
#ifndef __MUJ_H
 #define __MUJ_H
  obsah_hlavickoveho_souboru
#endif
```

Kontrolní úkol:

Prohlédněte si standardní hlavičkové soubory (např. stdio.h, io.h, stdlib.h) v adresáři INCLUDE a podívejte se na ošetření proti vícenásobnému vložení.

Příklady



Příklad č.1

Vytvořte makro TISK, které bude vypisovat vaše jméno, a makro POCET, udávající počet výpisů.

```
#include <stdio.h>
#define POCET  10
#define TISK   { printf("Rostislav Fojtik\n");}

int main( )
{
      int i;

      for (i=1; i<=POCET; i++) TISK

      return 0;
}
```

Příklad č.2



Vytvořte makro UpCase(c), které zvětší malé písmeno na velké.

```
#include <stdio.h>
#define UpCase(c)  ((c)>='a' && (c)<='z')?(c)-('a'-'z'):(c)
```

```
int main( )
{
  int znak;

  znak = getchar( );
  znak = UpCase(znak);
  putchar( znak );

  return 0;
}
```

Příklad č.3

Vytvořte makro NaTreti(x), které vypočte třetí mocninu x.



```
#include <stdio.h>
#define NaTreti(x) ((x)*(x)*(x))

int main( )
{
     int cis, vys;

printf("Zadej cislo: ");
scanf("%d", &cis);
vys = NaTreti(cis)
printf("Vysledek je %d\n", vys);

return 0;
}
```

Příklad č.4

Vytvořte hlavičkový soubor FUNKCE.H, který bude obsahovat hlavičky funkcí Abs a NSD (viz. minulá lekce o funkcích). Prohlédněte si strukturu standardních hlavičkových souborů.



```
#ifndef __FUNKCE_H
  #define __FUNKCE_H

  int NSD( int , int );
  int Abs( int );

#endif
```

Příklad č.5



```
/*
 * Využití maker bez parametrů
 */
#include <stdio.h>

#define POCET 10
    //Makro zpřehlední kód programu.
#define VypisUdaju {printf("Karel Novak \n");}
    //Složené závorky kolem makra jsou vhodné.
#define VypisUdaju2 { printf("Karel Novak\n");\
                      printf("Na vrsku 100\nHorni dolni\n");}
/*Složené závorky kolem makra jsou NUTNÉ! Podívejte se na
uplatnění makra. */
```

```
/*Odstraňte složené závorky a podívejte se, jak se pak program
provede. */

int main()
{
  int i;
  for (i=1;i<=POCET;i++)
    printf("%8d",i);

  //Výpis hodnoty makra POCET
  printf("\nMakro ma hodnotu %d\n",POCET);

  VypisUdaju

  for (i=1;i<=POCET;i++) VypisUdaju2;

  return 0;
}
```



Příklad č.6

```
/*
 * Využití maker s parametry
 * Makro změní malé písmenko na velké.
 */
#include <stdio.h>

#define UpCase(c) ( ((c)>='a' && (c)<='z')?(c)-('a'-'A'):(c) )
   //Nezapomínejte závorkovat!

int main()
{
  char znak,u_znak;

  printf("Zadej znak: ");
  znak=getchar();
  u_znak=UpCase(znak);
  printf("Upraveny znak %c\n",u_znak);

  u_znak=UpCase(znak) + 32;
  printf("Upraveny znak %c\n",u_znak);
  /* Odstraňte vnější závorky kolem těla makra a odzkoušejte
změnu jeho chování. Číslo třicet se přičte jen není-li
podmínka splněna! */
  u_znak=UpCase(znak) + 32;
  printf("Upraveny znak %c\n",u_znak);

  return 0;
}
```

Uvědomujte si stále, že makro své parametry nevyhodnocuje. Pouze nahradí posloupnosti znaků. Proto je vhodné všechny parametry v těle makra dát do závorek. Do závorek dávejte rovněž celé výrazy. Za výrazy v těle makra nedávejte středník. Vaše makro by nešlo využít v dalších výrazech.

Příklad č.7



```
/*
 * Využití maker s parametry
 * Vytvořte makra pro výpočet obvodu a obsahu
 * obdélníka a kruhu
 */
#include <stdio.h>

#define ObvodObdelnika(a,b) ( ((a)*2)+((b)*2) )
#define ObsahObdelnika(a,b) ( (a)*(b) )
#define PI 3.14156
#define ObvodKruhu(r) ( 2 * PI * (r) )
#define ObsahKruhu(r) ( PI * (r) * (r) )
   //Nezapomínejte závorkovat!

int main()
{
  int a=10,b=5,r=3;

  printf("Obvod obdelniku o stranach %d a %d je %d\n",
          a,b,ObvodObdelnika(a,b));
  printf("Obsah obdelniku o stranach %d a %d je %d\n",
          a,b,ObsahObdelnika(a,b));
  printf("Obvod kruhu s polomerem %d je
%10.3f\n",r,ObvodKruhu(r));
  printf("Obvod kruhu s polomerem %d je
%10.3f\n",r,ObsahKruhu(r));

  return 0;
}
```

Příklad č.4



```
/*
 * Podmíněný překlad
 * Program bude zpracovávat příkazy jen za určitých podmínek
 */
#include <stdio.h>
#define PC_DOS 1

int main()
{

  //Podmíněný překlad:
  #ifdef PC_DOS
   printf("Program pracuje na stroji PC se systémem DOS,
            muzeme pouzit funkce pro DOS\n");
  //Následují příkazy, které pracují jen na PC se systémem DOS
  #else
   printf("Jiná plaforma!\n");
   //Následují jen standardní příkazy
  #endif

  /* V čem se předcházející kód liší od následujícího? */
  //Klasická podmínka jazyka C:
  if (PC_DOS)
```

```
  {
   printf("Program pracuje na stroji PC se systémem DOS,
           muzeme pouzit funkce pro DOS\n");
  //Následují příkazy, které pracují jen na PC se systémem DOS
  }
  else
  {
    printf("Jiná plaforma!\n");
   //Následují jen standardní příkazy
  }
  /* U klasické podmínky jazyka C se kompilují obě větve
     podmínky. Ta se vyhodnocuje až v době běhu programu!
     Podmíněný překlad vybere jen jednu větev již při
     Překladu. Druhá větev podmínky se v programu vůbec
     nebude nacházet!
  */

  return 0;
}
```



Opakovací test

1) Který z následujících nástrojů se spouští při převádění zdrojového souboru na program jako první?

- preprocesor
- linker
- kompilátor
- všechny nástroje se spouštějí najednou

2) Který z následujících výroků není pravdivý?

- makro nemůže vracet hodnotu
- při volání makra se jeho parametry konvertují na určený typ
- makra se nerozvinou, jsou-li součásti řetězce
- makra s parametry se prováději rychleji než funkce

3) Který z následujících výroků je pravdivý?

- makra převádí a vyhodnocuje kompilátor
- makra se rozvinou a vyhodnotí před kompilací
- makra nelze psát na více než jeden řádek
- názvy maker se musí psát jen velkými písmeny

4) Které z následujících maker je nejvhodnější pro výpočet třetí mocniny čísla?

```
#define NaTreti(x) ((x)*(x)*(x));
#define NaTreti(x) ((x)*(x)*(x))
#define NaTreti(x)  (x)*(x)*(x)
#define NaTreti(x) {((x)*(x)*(x))}
```

Shrnutí učiva



• Preprocesor je nástroj, který upravuje zdrojový text před samotnou kompilací programu. Příkazy preprocesoru začínají znakem "#". Veškeré úpravy se týkají pouze textu. To znamená, že jednu posloupnost znaků nahrazují posloupností jinou.

• Soubory se do zdrojového textu vkládají pomocí příkazu `#include`.

```
#include
#include <stdio.h>
#include "muj.h"
#include "c:\programy\funkce.h"
```

• Makra bez parametrů se občas nazývají *symbolické konstanty*. Jejich účelem je nahrazení posloupnosti znaků jiným řetězcem. K vytvoření maker slouží příkaz `#define`.

```
#define MAX   100
#define TISK  { printf("Ahoj!\n");}
#define TEXT Text, který se nevejde na jeden radek \
          a pokracuje na druhem radku.
```

• Makra mohou mít rovněž parametry. Na rozdíl od funkcí, se však parametrům u maker neurčuje datový typ. Makra oproti funkcím částečně zrychlují provádění programu, ale zvětšují celkový kód.

```
#define soucin(a,b)   ((a)*(b))
```

• Podmíněný překlad nám může značně ulehčit fázi ladění a testování programu. V rámci kódu se mohou objevit příkazy sloužící pouze pro ladění. Pomocí příkazů preprocesorů můžeme ladící části používat jen v případě ladění, aniž bychom museli zasahovat do zdrojového textu. Ladící kód zůstává neustále ve zdrojovém souboru, kdykoliv se k němu můžeme znovu vrátit a zbytečně nám neprodlužuje program.

• Hlavičkové soubory obsahují deklarace funkcí (funkční prototypy) a deklarace proměnných, datových typů, konstant, maker. Proti vícenásobnému vložení určitého hlavičkového souboru do zdrojového textu je potřeba opatřit text příkazy podmíněného překladu:

```
#ifndef __MUJ_H
 #define __MUJ_H
  obsah_hlavickoveho_souboru
#endif
```

6. PREPROCESOR

Cílem této výukové lekce je seznámit se s pointery v programovacím jazyce C. Dozvíte se, jaké obtíže a případné chyby vás čekají při práci s pointery.

Budete:

- umět definovat pointery
- umět inicializovat pointery
- umět vytvářet dynamické proměnné
- umět alokovat a dealokovat paměť pro dynamické proměnné
- umět pracovat s adresami proměnných

Klíčová slova této kapitoly:
Alokace, dealokace, dynamická proměnná, reference, dereference



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Pointery tvoří velmi důležitou součást jazyka C. Teprve s nimi se programy stávají opravdovými "céčkovskými" programy. Práce s pointery - ukazateli je mocný nástroj, ale také velmi nebezpečný v případě špatného použití.

Pointer (ukazatel) je proměnná, která místo obvyklé hodnoty obsahuje adresu na určité paměťové místo. Teprve na této adrese se skrývá hodnota, se kterou budeme pracovat. Definice proměnné typu pointer vypadá následovně:

```
datovy_typ *jmeno_poinetru;
int *p;    /* ukazatel na datový typ int */
```

Operátor hvězdička * se nazývá *dereferenční operátor* a pomocí něj můžeme získat hodnotu uloženou na adrese, kam ukazuje příslušný pointer. Opačný význam má *referenční operátor*, který se zapisuje pomocí znaku & a pomocí kterého získáváme adresu určité proměnné.

```
int pocet=10; /* definice proměnné typu int a její
inicializace */
int *p = &pocet; /* definice proměnné typu ukazatel na int a
jeji inicializace na adresu proměnné pocet*/
*p = 100; /* na adresu, na kterou ukazuje proměnna p
zapíše hodnotu 100, tedy v tomto případě ji zapíše do
proměnné pocet */
pocet = 20; /* tento příkaz přepíše hodnotu proměnné pocet,
stejný efekt by se docílil příkazem *p = 20;  */
p = 20; /* tento příkaz změní adresu, na kterou pointer
ukazuje. Adresa bude mít hodnotu 20, nevhodný postup! */
```

Při práci s pointery je potřeba dbát na některé zásady, jinak se můžeme dostat do značných potíží, které mohou vést ke zhroucení programu a dokonce i některých případech operačního systému. V žádném případě nepracujte s pointery, do které jste nevložili určitou adresu. Pointer pak může ukazovat kdekoliv do paměti.

```
int *p;
*p = 100; /* nevhodná a nebezpečná operace, neboť hodnotu 100
jsme zapsali na neznamé místo v paměti, pointer neobsahuje
konkrétní přidělenou adresu paměti */
```

V předcházejícím příkladu chybí přiřazení adresy do pointeru. To se provádí buď přiřazením adresy již existující proměnné nebo dynamickou alokací pamětí pomocí příslušných funkcí. Pokud potřebujeme zajistit, aby pointer neukazoval do určitého místa v paměti (ani náhodného), přiřadíme mu hodnotu symbolické konstanty (makra bez parametrů) NULL. Tuto hodnotu je možné přiřadit ukazateli na jakýkoliv typ bez nutnosti přetypování.



Pozor! Nikdy nepracujte s ukazatelem, který jste nenasměrovali na konkrétní smysluplnou adresu!



Kontrolní úkol:

Jakou hodnotu má mít pointer, který nevyužíváme (nemá ukazovat na konkrétní místo v paměti)?

Funkce a pointery

U kapitoly o parametrech funkcí jsem se zmínil, že jazyk C nezná pojem parametr volaný odkazem, ale pouze hodnotou. Toto omezení se dá obejít pomocí parametrů, které jsou pointery.

Parametry, u kterých chceme, aby se chovaly jako volané odkazem (změna formálních parametrů uvnitř funkce způsobí změnu skutečných parametrů), definujeme jako ukazatele na příslušný typ. Běžné formální parametry při zavolání funkce vytvoří přesnou kopií parametrů skutečných. A protože se jedná o pouhou kopií, tak změna kopie (formálních parametrů) samozřejmě nezpůsobí změnu originálu (skutečných parametrů). Naopak pointer jako parametr pouze ukazuje na skutečný parametr, tedy netvoří se kopie. Pak jakákoliv změna formálního parametru uvnitř dané funkce se ve skutečnosti provádí s parametrem skutečným. Musíte si dále uvědomit, že ukazatele jsou proměnné, které obsahují adresy, proto při volání funkce musíme použít adresního operátoru.

Program obsahující funkci pro výměnu obsahu dvou proměnných by pak vypadal následovně:



```
#include <stdio.h>
void swap(int *, int *);    /* funkce pro výměnu */

int main()
{
  int x = 10,y = 20;

  swap(&x, &y);
  /* zde musí být adresy skutečných parametrů */
  printf(" x = %d, y = %d\n",a, b);
  return 0;
}
```

```
void swap(int *a, int *b)
{
  int pom;
  pom = *a;
  *a = *b;
  *b = pom;
}
```

Pointerová aritmetika

Součet pointeru a celého čísla

Přičtení celého čísla k pointeru znamená posunutí o určitý počet bytů od hodnoty adresy, na kterou pointer ukazuje. Hodnota celého čísla neznamená automaticky počet bytů posunutí, ale tato hodnota se musí vynásobit velikostí jakou zabírá typ, na který pointer ukazuje. Tuto velikost získáme pomocí operátoru `sizeof(*p)`.

`p + cislo` má význam `(char*)p + sizeof(*p) * cislo`

Odečítání celého čísla od pointeru

Tato aritmetická operace má podobnou funkci, jako minulá operace. Ovšem s tím rozdílem, že posun se provádí na druhou stranu, tedy k nižším hodnotám adres.

`p - cislo` má význam `(char*)p - sizeof(*p) * cislo`

Porovnávaní pointerů

Použití relačních operátorů pro porovnávání pointerů má význam jen v případě, že oba pointery jsou stejného typu a ukazují do stejného úseku paměti (například pole).

Odečítání pointerů

Tato aritmetická operace zjistí počet prvků mezi danými ukazateli. Tato operace má smysl jen u celistvého bloku paměti (např. pole).

`p1 - p2` má význam `((char*)p1 - (char*)p2) / sizeof(*p1)`

Dynamická alokace paměti

V úvodu kapitoly o ukazatelích jsem předeslal, že není možné pracovat s pointery, které nemají přiděleny správné adresy. Jedním ze způsobů, jak toho docílit je možnost dynamické alokace paměti. Proměnné, které nazýváme *dynamické*, můžeme vytvářet a rušit za běhu programu. Což je velmi vhodné z hlediska šetření s pamětí. Dynamické proměnné, které vytváříme v *heapu* - haldě, ovšem nemají své identifikátory a pro práci s nimi slouží právě pointery.

Standardní funkcí pro přidělovaní je malloc( ), která vrací ukazatel na datový typ void. Je vhodné přetypovat na určený datový typ. V případě, že se přidělení paměti nezdaří vrátí se nulová adresa NULL. Pro uvolnění pamětí slouží funkce free( ). Pozor! Funkce pouze označí uvolněnou paměť jako znovu použitelnou pro jinou alokaci. V daném paměťovém prostoru však mohou zůstat všechny hodnoty a adresa

daného paměťového prostoru je neustále v určeném pointeru. Proto je vhodné pointer nastavit na NULL.



Příklad použití:

```
int *p;
/* alokace pameti */
if ((p = (int *)malloc(10)) == NULL)
printf("Chyba! Malo pameti!");
else
  ..... /* práce s dynamickou promennou */
/* uvolneni pameti */
free(p);
p = NULL;  /* zruseni uvolnene adresy */
```

Příklady

Příklad č.1



Odkrokujte si následující program a sledujte: i,&i, p1, *p1, &p1, p2, *p2 a &p2.

```
int main( )
{
 int i=10;
 int *p1=&i;
 int *p2;

 i=20;
 *p2=30; /* Chyba! Pointer nenese adresu! */
 p2=p1;
 *p2=40;
 p1=50;

 return 0;
}
```

Příklad č.2



Vytvořte program, ve kterém vyalokujete paměť pro dynamickou proměnnou typu `long double` a vložte do ní hodnotu. Vypište adresu dynamické proměnné a pointeru.

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

int main( )
{
  long double *p;

  if ((p=(long double*)malloc(sizeof(long double))) == NULL)
  {
    printf("Chyba! Malo pameti!");
    return 0;
  }
  *p=1000.55;
  printf("Adresa pointeru je %p\n",&p);
  printf("Adresa dynamicke promenne je %p\n",p);
  printf("Hodnota ulozena na dynamicke promenne je %Lf\n",*p);
```

```
  /* uvolneni pameti */
  free(p);
  p=NULL; /* zajisti preruseni spojeni mezi pointerem a
uvolnenou pameti */

  return 0;
}
```

Příklad č.3

Vytvořte program a v něm funkci, která bude načítat znaky z klávesnice. Funkce se ukončí zadáme-li číslici '0'. Funkce vrátí přes parametry počet zadaných malých písmen, velkých písmen a ostatních znaků. Budeme pracovat jen se znaky do ordinální hodnoty 127.

Příklad č.4



```
/*
 * Pointery
 */

#include <stdio.h>

int main(int argc, char *argv[])
{
  int j=10;
  int *p1,*p2;
  /* Nezapomeňte před p2 rovněž napsat hvězdičku. V opačném
     případě p2 bude proměnná typu int a ne ukazatel na int!
   */

  p1=&j; //Inicializace pointeru p1. Pointer ukazuje na 'j'.
  *p1=100; //Hodnota 100 se zapsala do proměnné 'j'

  printf("Pointer p1 obsahuje hodnotu %p\n",p1);
  printf("Adresa promene j je %p\n",&j);
  printf("Adresa pointeru p1 je %p\n",&p1);
  printf("Hodnota mista, na kterou p1 ukazuje je %d\n",*p1);
  printf("Hodnota promenne j je %d\n",j);

  p2=p1;
  //Pointer p2 ukazuje na stejné místo jako p1, tedy na j
  p2=300;

  printf("Hodnota promenne, na kterou p2 ukazuje je
%d\n",*p2);
  printf("Hodnota promenne, na kterou p1 ukazuje je
%d\n",*p1);
  printf("Hodnota promenne j je %d\n",j);

  return 0;
}
```



Příklad č.5

```
/*
 * Pointery
 * Vytvoření a zrušení dynamické proměnné
 */

#include <stdio.h>

int main(int argc, char *argv[])
{
  float *pf;

  //Následuje alokace dynamické proměnné
  if ((pf=(float*)malloc(sizeof(float))) == NULL)
  {
    printf("Alokace dynamicke promenne se nezdarila");
    return 1;
  }
  *pf=1234.56789;
  printf("Hodnota dynamicke promenne je %f\n",*pf)

  free(pf);//Uvolnění paměti
  pf=NULL;

  return 0;
}
```

Příklad č.6



```
/*
 * Pointery
 * Vytvořte funkci, která bude načítat znaky tak dlouho dokud
 * nezadáme tečku. Funkce vrátí celkový počet znaků ve větě a
 * zároveň počet malých písmen, velkých písmen, číslic a
 * mezer. Uvažujeme jen znaky první části ASCII tabulky.
 */
#include <stdio.h>
#include <malloc.h>

int PoctyVeVete(int *pm,int *pv,int *pmz,int *pc);

int main(int argc, char *argv[])
{
  int CP,PM,PV,PMZ,PC;

  CP=PoctyVeVete(&PM,&PV,&PMZ,&PC);
  printf("Celkovy pocet znaku je %d\n",CP);
  printf("Celkovy pocet malych pismen je %d\n",PM);
  printf("Celkovy pocet velkych pismen je %d\n",PV);
  printf("Celkovy pocet mezer je %d\n",PMZ);
  printf("Celkovy pocet cislic je %d\n",PC);

  return 0;
}
```

```
int PoctyVeVete(int *pm,int *pv,int *pmz,int *pc)
{
  char znak;
  int pocet=0;
  *pm=*pv=*pmz=*pc=0;
  //Nezapomeňte vynulovat jednotlivé počty!

  do
  {
    znak=getchar();
    pocet++;
    if (znak>='a' && znak<='z') *pm+=1; //nebo (*pm)++;
    /* Pozor! Zápis *pm++ je chybný, neboť zvětšuje adresu
       a ne požadovanou hodnotu! */
    if (znak>='A' && znak<='Z') (*pv)++;
    if (znak==' ') (*pmz)++;
    if (znak>='0' && znak<='9') (*pc)++;

  }while (znak!='.');
  return pocet;
}
```

Opakovací test



1) Který z následujících operátorů je *dereferenční operátor*?

- &
- *
- ^
- @

2) Který z následujících operátorů je *referenční operátor*?

- &
- *
- ^
- @

3) Která z následujících funkcí uvolní dynamicky alokovanou paměť?

- malloc
- free
- alloc
- mfree

4) Který z následujících vztahů vyjadřuje součet pointeru a celého čísla? Platí:

```
typ *p; int cislo;
```

- p + sizeof(p) * cislo
- (char*)p + sizeof(*p) * cislo
- (char*)p + sizeof(p) * cislo
- (char*)p + cislo



Shrnutí učiva

• Pointer - ukazatel je proměnná, která místo obvyklé hodnoty obsahuje adresu na určité paměťové místo. Teprve na této adrese se skrývá hodnota, se kterou budeme pracovat. Definice proměnné typu pointer:
datovy_typ *jmeno_poinetru;
int *p; /* ukazatel na datový typ int */

• Operátor hvězdička * se nazývá *dereferenční operátor* a pomocí něj můžeme získat hodnotu uloženou na adrese, kam ukazuje příslušný pointer. Opačný význam má *referenční operátor*, který se zapisuje pomocí znaku & a pomocí kterého získáváme adresu určité proměnné.

• V žádném případě nepracujte s pointery, do které jste nevložili určitou adresu. Pointer pak může ukazovat kdekoliv do paměti.
int *p;
p = 100; / nevhodná a nebezpečná operace, neboť hodnotu 100 jsme zapsali na neznamé místo v paměti, pointer neobsahuje konkrétní přidělenou adresu paměti */

• Parametry funkcí volané odkazem, definujeme jako ukazatele na příslušný typ. Běžné formální parametry při zavolání funkce vytvoří přesnou kopií parametrů skutečných. A protože se jedná o pouhou kopií, tak změna kopie (formálních parametrů) samozřejmě nezpůsobí změnu originálu (skutečných parametrů). Naopak pointer jako parametr pouze ukazuje na skutečný parametr, tedy netvoří se kopie. Pak jakákoliv změna formálního parametru uvnitř dané funkce se ve skutečnosti provádí s parametrem skutečným. Musíte si dále uvědomit, že ukazatele jsou proměnné, které obsahují adresy, proto při volání funkce musíme použít adresního operátoru.

• Pointerová aritmetika
- Součet pointeru a celého čísla
p + cislo má význam (char*)p + sizeof(*p) * cislo
- Odečítání celého čísla od pointeru
p - cislo má význam (char*)p - sizeof(*p) * cislo
- Porovnávaní pointerů
Použití relačních operátorů pro porovnávání pointerů má význam jen v případě, že oba pointery jsou stejného typu a ukazují do stejného úseku paměti (například pole).
- Odečítání pointerů
p1 - p2 má význam ((char*)p1 - (char*)p2) / sizeof(*p1)

• Proměnné, které nazýváme *dynamické*, můžeme vytvářet a rušit za běhu programu. Dynamické proměnné, které vytváříme v *heapu* - haldě, ovšem nemají své identifikátory a pro práci s nimi slouží právě pointery.
Standardní funkcí pro přidělovaní je malloc( ), která vrací ukazatel na datový typ void. V případě, že se přidělení paměti nezdaří vrátí se nulová adresa NULL. Pro uvolnění pamětí slouží funkce free( ).

7. JEDNOROZMĚRNÉ POLE

Cílem lekce je naučit se vytvářet proměnné typu jednorozměrné pole, a to jak statické, tak i dynamické. Naučíte se některé zvláštností a omezení při definici polí.

Budete:

- vědět, jak definovat proměnnou typu jednorozměrné statické pole
- vědět, jak definovat proměnnou typu jednorozměrné dynamické pole
- vědět, jak se vyhnout chybám při práci s pamětí
- umět využívat pole ve svých programech

Klíčová slova této kapitoly:
Jednorozměrné pole, dynamické pole, statické pole, adresa

Čas potřebný k prostudování učiva kapitoly:
2 hodiny



Vstupní test



1) Ukázka testovacích otázek
- všechny prvky pole musí být stejného datového typu
- datový typ 'int'
- prvky mohou být libovolného typu
- prvky musí mít velikost menší než 4B

2) Jak je reprezentováno jednorozměrné statické pole v paměti počítače?
- jako souvislý blok dat
- jako samostatné bloky jednotlivých prvků pole
- jako několik samostatných bloků dat
- prvky jsou uloženy na libovolném místě v paměti

3) Jakou činnost provede funkce `free(p)`?
- uvolní dynamicky alokovanou paměť pro novou alokaci
- uvolní paměť a vynuluje pointer
- uvolní a vyčistí paměť od hodnot
- vynuluje pointer

Pole je datový typ obsahující určitý počet prvků stejného typu. V paměti se pole nachází v celistvém bloku.

Statické pole

Definice pole vypadá následovně:

```
datovy_typ jmeno_pole[pocet_prvku];
```

Například:

```
#define MAX 100
int pole1[MAX];
int pole2[] = {12,50,33}; /* inicializace pole */
```

Vše uvedené pole `pole1` obsahuje prvky typu *int* a první prvek má index 0 a poslední MAX-1, tedy 99. Přístup k jednotlivm prvkům pole se provádí indexů:

```
pole[0] = 100;  /* naplnění prvního prvku pole */
pole[12] = 50;
pole[99] = 45;  /* naplnění posledního prvku pole */
pole[100] = 20; /* Chyba! Zapsání hodnoty mimo pole! */
```

Jednotlivé prvky pole jsou *l-hodnotami*, to znamená že mají svou adresu, se kterou můžeme pracovat. Například adresa prvního prvku v našem poli se získá pomocí &pole[0]. Adresa i-tého prvku pole je pak: &pole[i]. Díky tomu, že každé pole začíná indexem 0, pak platí, že adresa libovolného prvku v poli se dá vyjádřit vztahem:

```
&pole[i] = bázová_adresa pole + i * sizeof(typ)
```

Následující dvojice jsou analogické:

```
&pole[i]   odpovídá   pole + i
pole[i]    odpovídá   *(pole + i)
&pole[0]   odpovídá   pole + 0   odpovídá   pole
pole[0] = 100;  odpovídá   *pole = 100;
pole[12] = 50;  odpovídá   *(pole + 12) = 50;
pole[99] = 45;  odpovídá   *(pole + 99) = 45;
```

Tato vlastnost občas někoho svádí k závěru, že pole a pointery jsou jedno a totéž, což samozřejmě není pravda! Pointer je proměnná, která obsahuje pouze adresu a paměti zabírá tolik bitů kolik je potřeba na vyjádření adresy v daném systému (16 bitů, 32 bitů...). Název pole sice také představuje adresu. Ovšem u statického pole je to neměnná hodnota (nejedná se o *l-hodnotu*) a velikost paměti je určena součinem počtu prvků a velikosti datového typu prvku.

```
int i=1000;
int *p1, *p2=&i;
int pole1[MAX], pole2[MAX];

p1 = p2;   /* správný zápis */
pole1 = pole2;  /* Nelze! Snaha o přiřazení adresy do adresy.
Například FF12 = FE00! Kopírování pole se musí
provádět po jednom prvku!*/
```

Pozor! Pole a pointer jsou dvě různé proměnné. Název pole představuje jeho adresu.

Dynamické pole

Dynamické pole vytvoříme pomocí pointeru a následné alokace paměti.

```
#define POCET 100
int i;
int *dp;
```

```
/* alokace paměti v heapu */
if ((p=(int*)malloc(POCET * sizeof(int))) == NULL)
{
   printf("Chyba! Malo pameti!");
   return 0;
}
/* naplnění pole hodnotami */
for (i=0;i<POCET;i++)
     *(dp+i)=i; /* nebo dp[i]=i; */
/* uvolněn paměti */
free(dp);
dp = NULL;
```

Jak je vidět z minulého příkladu, není rozdíl mezi práci s dynamickým a statickým polem. V obou případech můžeme použit jak přístupu pomocí indexu v hranatých závorkách, tak pointerové aritmetiky.

Výhodou dynamického pole je, že můžeme měnit jeho velikost, tedy počet prvků v poli.

Na rozdíl od statického pole nelze zjistit velikost (nebo počet prvků) dynamického pole. Pro zjištění velikosti a počtu prvků statického pole stačí napsat:

```
typ spole[N];
velikost = sizeof(spole);
pocet = sizeof(spole)/sizeof(typ);
```

U dynamicky vytvořeného pole však výše uvedený postup nelze použít!

```
typ *dpole;
/*následuje alokace paměti */
dpole=(typ*)malloc(POCET * sizeof(typ));
velikost = sizeof(dpole);
```

Tento zápis zjistí jen velikost pointeru, tedy velikost adresy pole!

Pole jako parametr funkce

Je-li pole parametrem funkce, pak se skutečný parametr předává pouze odkazem, tedy pomocí pointerů. Což má nesporné výhody. Zaprvé v paměti se nemusí vytvářet lokální kopie skutečných parametrů, což šetří paměť. Prvky se nemusí po jednom kopírovat, což naopak šetří čas.

```
int Minimum(int pole[], int pocet)
{
 int i;
 int min=pole[0];
 for (i=1;i<pocet;i++)
  if (min>pole[i]) min=pole[i];
 return min;
}
```

Hlavička funkce může vypadat i takto:

```
int Minimum(int *pole, int pocet);
```

Volání funkce pak bude vypadat:

```
#define POCET 100
int main()
{
 int pp[POCET];
 .....
 x = Minimum(pole, POCET);
 .....
}
```



Kontrolní úkol:
Pole jako parametr funkce se předává odkazem nebo hodnotou?

Příklady



Příklad č.1

Vytvořte statické pole o N prvcích, naplněte náhodnými hodnotami typu int a vypište na obrazovku.

```
/*
 * Vytvořte statické pole o N prvcích, naplňte náhodnými
 * hodnotami typu int a vypište na obrazovku.
 */
#include <stdio.h>
#include <stlib.h>
#include <time.h>

#define N 10
int main( )
{
  int pole[N];
  randomize();
  for (i=0;i<N;i++)
   pole[i]=random(100); /* nebo  *(pole + i)=random(100);  */
  for (i=0;i<N;i++)
   printf("%4d",pole[i]); /* nebo printf("%4d",*(pole+i); */

  return 0;
}
```

Příklad č.2



Vytvořte dynamické pole o N prvcích, naplněte náhodnými hodnotami typu int, vypište na obrazovku a následně uvolněte paměť.

```
/*
 * Vytvořte dynamické pole o N prvcích, naplněte náhodnými
 * hodnotami typu int, vypište na obrazovku a následně
uvolněte paměť.
 */
#include <stdio.h>
#include <stlib.h>
#include <time.h>

#define N 10
int main( )
{
  int *pole;
  randomize();
  if ((pole=(int*)malloc(N * sizeof(int))) == NULL)
  {
   printf("Chyba! Malo pameti!");
   return 0;
  }
  for (i=0;i<N;i++)
   pole[i]=random(100); /* nebo  *(pole + i)=random(100);  */
  for (i=0;i<N;i++)
   printf("%4d",pole[i]); /* nebo printf("%4d",*(pole+i); */
```

```
  free(pole);
  pole=NULL;

  return 0;
}
```

Příklad č.3



Vytvořte dynamické pole o N prvcích, naplňte náhodnými hodnotami typu int a vypište na obrazovku. Pak pole zvětšete o M prvků, doplňte hodnoty a celé pole vypište.

```
/*
 * Vytvořte dynamické pole o N prvcích, naplněte náhodnými
 * hodnotami typu int a vypište na obrazovku. Pak pole
 * zvětšete o M prvků, doplňte hodnoty a celé pole vypište.
 */
#include <stdio.h>
#include <stlib.h>
#include <time.h>

#define N 10
#define M 5

int main( )
{
  int *pole, *nove_pole;
  randomize();

  if ((pole=(int*)malloc(N * sizeof(int))) == NULL)
  {
    printf("Chyba! Malo pameti!");
    return 0;
  }
  for (i=0;i<N;i++)
    pole[i]=random(100); /* nebo  *(pole + i)=random(100);  */
  for (i=0;i<N;i++)
    printf("%4d",pole[i]); /* nebo printf("%4d",*(pole+i); */

  if ((nove_pole=(int*)malloc(N * M * sizeof(int))) == NULL)
  {
    printf("Chyba! Malo pameti!");
    return 0;
  }
  for (i=0;i<N;i++) /* kopirovani pole */
    nove_pole[i]=pole[i];
  for (i=N;i<M;i++) /* doplneni hodnot do novych prvku*/
    nove_pole[i]=random(100);
  free(pole);    /* uvolneni stareho pole */
  pole = nove_pole;/* presunuti ukazatele na novou adresu */
  nove_pole=NULL;

  for (i=0;i<N+M;i++)
    printf("%4d",pole[i]); /* nebo printf("%4d",*(pole+i); */

  free(pole);
```

```
  pole=NULL;

  return 0;
}
```

Příklad č.4



```
/*  Jednorozměrné pole
 *  Vytvořte jednorozměrné statické pole o 20 prvcích typu int
 *  a naplňte je celými čísly od 1 do 20. Dále vytvořte
 *  dynamické pole s prvky typu int.
 *  Překopírujte obsah statického pole do pole dynamického.
 *  Pro naplnění, výpis a kopírování pole vytvořte funkce.
 */
#include <stdio.h>
#include <malloc.h>

#define N 20

void NaplnPole(int p[], int pocet);
void VypisPole(int p[], int pocet);
void KopirujPole(int p1[],int p2[], int pocet);

int main(int argc, char *argv[])
{
  int s_pole[N];
  int *d_pole;

  NaplnPole(s_pole,N);
  printf("Obsah statickeho pole:\n");
  VypisPole(s_pole,N);

  //alokace dynamického pole
  if ( (d_pole=(int*)malloc(N*sizeof(int)))==NULL )
  {
    printf("Chyba pri alokaci pameti!");
    return 0;
  }
  KopirujPole(d_pole,s_pole,N);
  printf("\nObsah statickeho pole:\n");
  VypisPole(d_pole,N);

  free(d_pole);

  return 0;
}

void NaplnPole(int p[], int pocet)
{
  int i;
  for(i=0;i<pocet;i++) p[i]=i+1;
}

void VypisPole(int p[], int pocet)
{
  int i;
  for(i=0;i<pocet;i++)
```

```
    printf("%d.prvek = %d, ",i,p[i]);
}

void KopirujPole(int p1[],int p2[], int pocet)
{
  int i;
  for(i=0;i<pocet;i++)
    p1[i]=p2[i];
}
```

Příklad č.5



```
/*
 * Vytvořte jednorozměrné dynamické pole o 20 prvcích typu int
 * a naplňte je celými čísly od 1 do 20. Prvky pole vypište.
 * Dále dynamické pole zvětšete o 10 prvků a doplňte o
 * hodnoty 21 až 30. Prvky pole vypište.
 */

#include <stdio.h>
#include <malloc.h>

#define N 20
#define M 10

int main(int argc, char *argv[])
{
  int *d_pole;
  int i;
  int *pom; //pomocný ukazatel pro dealokaci

  //1. Alokujeme pole o 20 prvcích a naplníme jej hodnotami
  if ((d_pole=(int*)malloc(N*sizeof(int)))==NULL)
  {
    printf("Chyba pri alokaci!");
    return 0;
  }
  for (i=0;i<N;i++)
  {
    d_pole[i]=i+1;
    printf("%d ", d_pole[i]);
  }
  //2. Poznamenáme si adresu pole
  pom=d_pole;
  //3. Vyalokujeme si novou velikost pole
  if ((d_pole=(int*)malloc((N+M)*sizeof(int)))==NULL)
  {
    printf("Chyba pri alokaci!");
    return 0;
  }
  //4. Překopírujeme prvky ze starého do nového pole
  for (i=0;i<N;i++)
  {
    d_pole[i]=pom[i];
  }
  //5. Přidáme nové hodnoty
  for (i=N;i<N+M;i++)
```

```
  {
    d_pole[i]=i+1;
  }
  //6. Vypíšeme pole
  printf("\nRealokovane pole:\n");
  for (i=0;i<N+M;i++)
  {
     printf("%d ", d_pole[i]);
  }
  //7. Uvolníme staré pole
  free(pom);
  pom=NULL;

  free(d_pole);

  return 0;
}
```



Příklad č.6

```
/*
 * Vytvořte jednorozměrné dynamické pole o 5 prvcích typu
 * float a naplněte je čísly. Prvky pole vypište.
 * Vytvořte funkci, která setřídí vzestupně prvky podle
 * velikosti. Prvky pole vypište.
 */

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

#define N 5

void SetridPole(float p[], int pocet);

int main(int argc, char *argv[])
{
  float *d_pole;
  int i;

  if ((d_pole=(float*)malloc(N*sizeof(float)))==NULL)
  {
    printf("Chyba pri alokaci!");
    return 0;
  }
  d_pole[0]=3.1;
  d_pole[1]=3.23;
  d_pole[2]=13.21;
  d_pole[3]=4.51;
  d_pole[4]=9.67;
  printf("Nesetrizene pole:\n");
  for (i=0;i<N;i++)
    printf("%10.2f ", d_pole[i]);

  SetridPole(d_pole,N);
  printf("\nSetrizene pole:\n");
  for (i=0;i<N;i++)
```

```
    printf("%10.2f ", d_pole[i]);

  free(d_pole);
  system("PAUSE");
  return 0;
}

void SetridPole(float p[], int pocet)
{
  int i,j;

  for(j=0;j<pocet-1;j++)
   for(i=0;i<pocet-1;i++)
    if (p[i]>p[i+1])
    {
      float pom=p[i];
      p[i]=p[i+1];
      p[i+1]=pom;
    }
 }
```

Opakovací test



1) Jaký index musí mít první prvek jednorozměrného statického pole?

0
1
-1
index může mít libovolnou ordinální hodnotu

2) Jaký index musí mít první prvek jednorozměrného dynamického pole?

0
1
-1
index může mít libovolnou ordinální hodnotu

3) Jaký index musí mít poslední prvek jednorozměrného pole? Pole je definováno
typ pole[MAX];

MAX – 1
MAX
100
index může mít libovolnou ordinální hodnotu

4) Který z následujících zápisu nepřiřadí do druhého prvku pole hodnotu 10? Pole je definováno `int pole[MAX];`

```
pole[2] = 10;
pole[1] = 10;
*(pole + 1) = 10;
i = 1; pole[i] = 10;
```



Shrnutí učiva

- Pole je datový typ obsahující určitý počet prvků stejného typu. V paměti se pole nachází v celistvém bloku. Na rozdíl od Pascalu je pole v jazyku C indexováno vždy od nuly.
- Definice pole vypadá následovně:

```
datovy_typ jmeno_pole[pocet_prvku];
```

Například:

```
#define MAX 100
int pole[MAX];
int pole2[3] = {12,50,33}; /* inicializace pole */
```

- Adresa prvního prvku v našem poli se získá pomocí &pole[0]. Adresa i-tého prvku pole je pak: &pole[i]. Díky tomu, že každé pole začíná indexem 0, pak platí, že adresa libovolného prvku v poli se dá vyjádřit vztahem:

```
&pole[i] = bázová_adresa pole + i * sizeof(typ)
```

Název pole pole představuje jeho adresu.

- Dynamické pole vytvoříme pomocí pointeru a následné alokace paměti.

```
#define POCET 100
int i;
int *dp;
/* alokace paměti v heapu */
p=(int*)malloc(POCET * sizeof(int))
/* naplnění pole hodnotami */
...
/* uvolněn paměti */
free(dp);
dp = NULL;
```

- U dynamického i statického jednorozměrného pole můžeme použit jak přístupu pomocí indexu v hranatých závorkách, tak pointerové aritmetiky. Výhodou dynamického pole je, že můžeme měnit jeho velikost, tedy počet prvků v poli. Na rozdíl od statického pole nelze zjistit velikost a počet prvků dynamického pole.

- Je-li pole parametrem funkce, pak se skutečný parametr předává pouze odkazem, tedy pomocí pointerů.

```
int Minimum(int pole[], int pocet);
```

nebo

```
int Minimum(int *pole, int pocet);
```

8. VÍCEROZMĚRNÁ POLE

Cílem této výukové lekce je seznámit se s možnostmi využití a definování vícerozměrných polí v programovacím jazyku C.

Budete:

- umět definovat vícerozměrná pole
- umět vytvářet statická i dynamická vícerozměrná pole
- umět využít vícerozměrná pole jako parametry funkcí



Klíčová slova této kapitoly:
Alokace, dealokace, dynamická proměnná, reference, dereference

Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Vícerozměrná pole můžeme v jazyku C vytvořit několika různými způsoby, které se dosti znatelně liší uložením prvků v paměti. Vytváření jednotlivých vícerozměrných polí bude vysvětleno na polích o dvou rozměrech. Složitější struktury si jistě odvodíte sami.

Vždy si však můžeme představit, že vícerozměrné pole je specifickým případem pole jednorozměrného, jehož prvky jsou opět typu pole.

Dvourozměrné statické pole

Takové pole je v paměti alokováno na začátku bloku jako souvislý blok paměti. Například:

```
int mat[2][4];
```

Uvedené pole má celkem 2 * 4 tedy 8 prvků a to ve dvou řádcích po čtyřech sloupcích. Jednotlivé prvky se popisují následujícím způsobem:

`mat[0][0] mat[0][1] mat[0][2] mat[0][3] mat[1][0] mat[1][1] mat[1][2] mat[1][3]`

Vzhledem k tomu, že pole je v tomto případě souvislý blok paměti jsou následující způsoby zápisu platné, ovšem matoucí a nedoporučují je používat.

`mat[0][4]` to samé jako `mat[1][0]`
`mat[1][4]` Pozor! Již jsme mimo pole!

Stejně jako u jednorozměrného pole, lze i zde zjistit adresy jednotlivých prvků, zde i jednotlivých řádků. Například:

`&mat[0][3] &mat[1][1] &mat[1][0]`
`&mat[1]` adresa druhého řádku, řádku s indexem 1
`&mat[2]` adresa třetího řádku, řádku s indexem 2
`&mat[0][0] &mat[0] mat` Všechny tři zápisy představují stejnou adresu.
Jedná se o adresu prvního prvků, prvního řádku, celého pole.
Všechny adresy jsou konstantní, vytvářejí se na začátku programového bloku.
Nejedná se tedy o *l-hodnoty* - nelze je změnit.



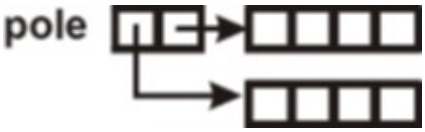
Pole pointerů

Jedná se o jednorozměrné pole, jehož prvky jsou pointery. Po alokaci paměti může každý z pointeru ukazovat na dynamicky vytvořené pole, takže celková struktura bude mít dva rozměry, řádky a sloupce. Oproti statickému dvourozměrnému poli, které má v každém řádku stejný počet sloupců, můžeme u pole pointerů mít řádky různě dlouhé (různý počet sloupců) a můžeme jejich velikost dynamicky měnit v průběhu programu.

Pokud bychom chtěli pomoci pole pointerů vytvořit pole o dvou řádcích po čtyřech sloupcích, museli bychom postupovat následovně.

```
int *pp[2];
/* nyní musí následovat alokace jednotlivých řádků */
pp[0]=(int*)malloc(4*sizeof(int));
pp[1]=(int*)malloc(4*sizeof(int));
/* práce s jednotlivými prvky */
pp[0][0] = 10;
pp[1][1] = 50;
/* uvolnění paměti */
free(pp[0]);
free(pp[1]);
```

Pozor! Jednotlivé řádky nemusí být v paměti alokovány za sebou, nemusí spolu fyzicky sousedit.



Pointer na pole

Pointer na pole na umožňuje dynamicky vytvářet počet řádků, kde všechny řádky mají stejný počet sloupců. V paměti se nejprve vytvoří pouze ukazatel a dynamickou alokací pak souvislý blok, ve kterém budou všechny prvky.

Příklad pro dva sloupce po čtyřech řádcích:

```
int (*pnp)[4];
pnp=(int (*)[])malloc(2*4*sizeof(int));
pnp[0][0] = 10;
pnp[1][1] = 50;
free(pnp);
```



Pointer na pointer

Pointer na pointer představuje nejsložitější způsob, který umožňuje dynamicky vytvářet a upravovat počty řádků i počty jejich sloupců. Opět uvedu příklad pro dva sloupce po čtyřech řádcích:

```
int **ppp;
/*alokace paměti pro pole pointeru, určeni počtu řádků */
ppp=(int**)malloc(2*sizeof(int*));
/*alokace paměti pro jednotlivé řádky, určení počtu sloupců */
ppp[0]=(int*)malloc(4*sizeof(int));
```

```
ppp[1]=(int*)malloc(4*sizeof(int));
/* práce s jednotlivými prvky */
ppp[0][0] = 10;
ppp[1][1] = 50;
/* uvolnění paměti, nejprve jednotlivých řádků */
free(ppp[0]);
free(ppp[1]);
/* uvolnění pole pointerů */
free(ppp);
```



Všimněte si, že práce s jednotlivými prvky je ve všech čtyřech uvedených případech stejná. Je samozřejmé, že místo zápisu indexu v hranatých závorkách jsme mohli používat pointerové aritmetiky.

Dvourozměrné pole jako parametr funkce

U formálních parametrů se musí udávat druhý rozměr (počet sloupců).

```
float Maximum(float p[][5], int radky);
```

Příklad funkce, která vrátí největší hodnotu dvourozměrného pole.

```
float Maximum(float p[][5], int radky)
{
 float Max=p[0][0];
 int i, j;
 for (i=0;i<radky;i++)
  for (j=0;j<5;j++)
   if (p[i][j]>Max) Max=p[i][j];
 return Max;
}
```

Inicializace polí

Inicializace polí se provádí následujícím způsobem:

```
int pole[4] = {1,2,3,4};
int pole[] = {1,2,3,4};
```

Počet prvků u jednorozměrného pole nemusí být uveden, neboť kompilátor si jej sám doplní. U dvourozměrného pole nemusí být uveden počet řádků, musí se však zapsat počet sloupců.

```
int matice[][4] ={ {1,2,3,4}, {5,6,7,8},
{9,10,11,12} };
```

Kontrolní úkol:

Který rozměr u dvourozměrného pole, které je definováno strukturou pointer na pointer, je možné v průběhu programu měnit?



Příklady



Příklad č.1

```
/*
 * Vícerozměrná pole
 * Vytvořte matici 5 krát 5 prvků typu int. Naplňte ji
 * náhodnými čísly. Matici vypište
 * a vypište rovněž součet prvků v hlavní diagonále.
 * Matici vytvořte jako statické dvourozměrné pole,
 * pole pointerů, pointer na pole a pointer na pointer.
 * Pomocí podmíněného překladu řiďte druh použité matice.
 */

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

#define R 5 //řádky
#define S 5 //sloupce


#define STAT_POLE
//#define POLE_POINTERU
//#define POINTER_NA_POLE
//#define POINTER_NA_POINTER

int main(int argc, char *argv[])
{
  int i, j;
  long soucet=0;
  time_t t;


  #ifdef STAT_POLE
   int mat[R][S];
   printf("Staticke dvourozmerne pole:\n");
  #endif
  #ifdef POLE_POINTERU
   int *mat[R];
   for(i=0;i<R;i++)
     if ((mat[i]=(int*)malloc(S*sizeof(int)))==NULL)
     {
       printf("Chyba pri alokaci!");
       return 0;
     }
   printf("Pole pointeru:\n");
  #endif
  #ifdef POINTER_NA_POLE
   int (*mat)[S];
   if ((mat=(int*)malloc(R*S*sizeof(int)))==NULL)
     {
       printf("Chyba pri alokaci!");
       return 0;
     }
```

```
   printf("Pointer na pole:\n");
  #endif
  #ifdef POINTER_NA_POINTER
   int **mat;
   if ((mat=(int**)malloc(R*sizeof(int*)))==NULL)
     {
       printf("Chyba pri alokaci!");
       return 0;
     }
   for(i=0;i<R;i++)
    if ((mat[i]=(int*)malloc(S*sizeof(int)))==NULL)
     {
       printf("Chyba pri alokaci!");
       return 0;
     }
   printf("Pointer na pointer:\n");
  #endif

  //Inicializace generátoru náhodných hodnot
  srand((unsigned) time(&t));
  //Naplnění matice náhodnými hodnotami
  for(i=0;i<R;i++)
   for(j=0;j<S;j++)
     mat[i][j]=rand();
  //Výpis matice
  for(i=0;i<R;i++)
  {
   for(j=0;j<S;j++)
     printf("%8d ",mat[i][j]);
   printf("\n");
  }
  //Výpočet součtu hodnot v hlavní diagonále
  for(i=0;i<R;i++)
    soucet+=mat[i][i];
  printf("\nSoucet v hlavni diagonale je %ld\n ",soucet);

  #ifdef POLE_POINTERU
   for(i=0;i<R;i++)
     free(mat[i]);
  #endif
  #ifdef POINTER_NA_POLE
   free(mat);
  #endif
  #ifdef POINTER_NA_POINTER
   for(i=0;i<R;i++)
     free(mat[i]);
   free(mat);
  #endif

  return 0;
}
```

Příklad č.2



```
/*
 * Vytvořte matici R krát S prvků typu int. Naplňte ji
 * náhodnými čísly. Matici vypište.
 * Vytvořte funkci, která vrátí největší hodnotu matice.
 * Vytvořte funkci, která vrátí součet všech hodnot matice.
 */

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

#define R 4 //řádky
#define S 5 //sloupce

int MaximumMatice(int m[][S], int radky);
long SoucetMatice(int m[][S], int radky);

int main(int argc, char *argv[])
{
  int i, j;
  int maximum;
  long soucet=0;
  time_t t;



  int mat[R][S];

  //Inicializace generátoru náhodných hodnot
  srand((unsigned) time(&t));
  //Naplnění matice náhodnými hodnotami
  for(i=0;i<R;i++)
   for(j=0;j<S;j++)
     mat[i][j]=rand();
  //Výpis matice
  for(i=0;i<R;i++)
  {
   for(j=0;j<S;j++)
     printf("%8d ",mat[i][j]);
   printf("\n");
  }

  maximum=MaximumMatice(mat,R);
  printf("\nNejvetsi hodnota matice je %8d \n",maximum);

  soucet=SoucetMatice(mat,R);
  printf("\nSoucet vsech hodnot matice je %12ld \n",soucet);

   return 0;
}

int MaximumMatice(int m[][S], int radky)
{
  int i,j;
```

```
  int mx=m[0][0];

  for(i=0;i<radky;i++)
   for(j=0;j<S;j++)
     if (m[i][j]>mx) mx=m[i][j];
  return mx;
}

long SoucetMatice(int m[][S], int radky)
{
  int i,j;
  long souc=0;

  for(i=0;i< radky;i++)
   for(j=0;j<S;j++)
     souc+=m[i][j];
  return souc;
}
```

Opakovací test



1) Kolik prvků má následující vícerozměrné statické pole?
typ pole[3][4][5];
- 60
- 12
- nelze zjistit
- libovolný počet

2) Jak lze zjistit velikost dvourozměrného statického pole?
sizeof(typ_pole) * (počet prvků)
sizeof(typ_pole)
(typ_pole) * (počet prvků)
nelze zjistit

3) Jak lze zjistit velikost dvourozměrného dynamického pole definovaného pomocí pointeru na pointer?
sizeof(typ_pole) * (počet prvků)
sizeof(typ_pole)
(typ_pole) * (počet prvků)
nelze zjistit

4) Který z následujících případů definuje pointer na pole?
typ (*p)[N];
typ *p[N];
typ **p;
typ p[N][M];

5) Který z následujících případů definuje pole pointerů?
typ (*p)[N];
typ *p[N];
typ **p;
typ p[N][M];



Shrnutí učiva

- Vícerozměrná pole můžeme v jazyku C vytvořit několika různými způsoby:

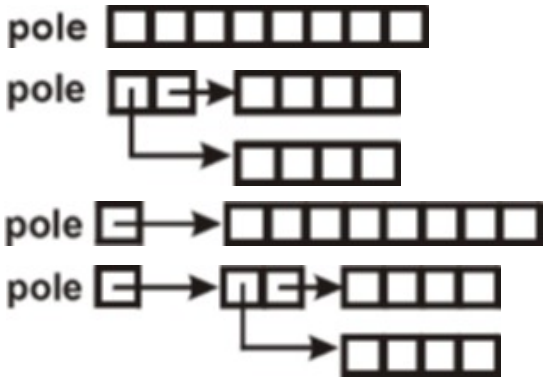
1) První obrázek představuje dvourozměrné statické pole (Na obrázku 2 x 4 prvky).
2) Druhý obrázek představuje pointer na pole.
3) Třetí obrázek představuje pole pointerů.
4) Čtvrtý obrázek znázorňuje pointer na pointer.

- Vždy si však můžeme představit, že vícerozměrné pole je specifickým případem pole jednorozměrného, jehož prvky jsou opět typu pole.
- Inicializace polí se provádí následujícím způsobem:

```
int pole[4] = {1,2,3,4};
int pole[] = {1,2,3,4};
int matice[][4] ={ {1,2,3,4},{5,6,7,8}};
```

- U formálních parametrů se musí udávat druhý rozměr (počet sloupců).

```
float Maximum(float p[][5], int radky);
```



9. PRÁCE S ŘETĚZCI

Cílem této výukové lekce je přiblížit způsob definovaní řetězců v programovacím jazyku C.

Budete:

- umět definovat řetězce v programovacím jazyce C
- umět využívat knihovní funkce pro práci s řetězci

Klíčová slova této kapitoly:
Řetězce, string.h, nulovací znak, NULL, `'\0'`

Čas potřebný k prostudování učiva kapitoly:
2 hodiny



Základní informace o práci s řetězci

Řetězec je vlastně jednorozměrné pole, jehož prvky jsou proměnné typu **char**. V daném poli je však nutné nějakým způsobem určit ukončení řetězce. V jazyku C se toto provádí ukončovacím znakem `'\0'`. Hodnoty za tímto znakem se při standardní práci s řetězci nevyužívají. Definice řetězcové proměnné je stejná jako u jednorozměrných polí.

```
char str[5];     /* staticky */
char *pstr;      /* dynamicky */
pstr=(char*)malloc(5);
```

Stejně jako u jiných polí, můžeme i řetězcové proměnné inicializovat:

```
char str[] = "Karel";
```

V předcházejícím příkladu se vytvoří pole, které bude mít šest prvků. Z toho pět znaků ze slova Karel a jeden znak na ukončení řetězce. Stále je si potřeba uvědomovat, že řetězce jsou vlastně pole. Název pole představuje konkrétní adresu a z toho vyplývá, proč nelze napsat následující příkaz:

```
str = "Karel"; /* Chyba! */
```

Abychom do proměnné `str` vložili konkrétní řetězec znaků, musíme je překopírovat po jednotlivých prvcích pole.

```
str[0]='K';
str[1]='a';
str[2]='r';
str[3]='e';
str[4]='l';
str[5]= '\0';
```

Pozor! Pokud bychom zapsat zakončující znak `'\0'`, za řetězec by se považovaly všechny znaky v paměti až do prvního výskytu ukončujícího znaku! Kopírování samozřejmě nemusíme vypisovat tak nepěkně, jak je naznačeno. Je možné použit standardní funkce `strcpy()`, pomocí které tuto operaci provedeme mnohem elegantněji.

```
strcpy(str, "Karel");
```

Funkce pro práci s řetězci

Pro využití standardních funkcí pro práci s řetězci je potřeba připojit hlavičkový soubor `string.h`. Zde je několik základních funkcí, na zbylé funkce se podívejte do nápovědy překladače.

```
int strlen(char * s);
```
- vrací délku řetězce bez ukončovacího znaku '\0'

```
char * strcpy(char *s1, char *s2);
```
- zkopíruje obsah řetězce s2 do řetězce s1

```
char * strcat(char *s1, char *s2);
```
- připojí řetězec s2 na konec řetězce s1, řetězce se spojí

```
int strcmp(char *s1, char *s2);
```
- lexikografické porovnání řetězců s1 a s2

```
int atoi(char *s);
int atol(char *s);
int atof(char *s);
```
- provádí konverzi řetězce na číslo typu int, long, případně float

Čtení a výpis řetězců

Formátované čtení provádíme pomocí funkce `scanf( )`. Její nevýhodou je, že načítá do řetězce znaky do prvního výskytu bílého znak (např. mezery). To znamená, že z řetězce "Ferda mravence" uloží do proměnné jen "Ferda". Pro načítání celého řádku, včetně mezer je vhodnější používat funkci `gets( )`.

```
scanf("%s", str);
gets(str);
```

Pro výpis řetězcové proměnné můžeme použít například funkci `printf`.

```
printf("Retezec je %s\n", str);
```



Kontrolní úkol:

Jak se ukončují řetězce v jazyce C?

Příklady

Příklad č.1

```
/*
 * Vytvořte funkci MyStrLen, která vrátí délku řetězce
 * bez koncové nuly
 */

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

long MyStrLen(char str[]);

int main(int argc, char *argv[])
{
  char s[10];

  printf("Zadej text: ");
  scanf("%s",s);
  printf("\nDelka textu je %ld\n", MyStrLen(s));
```

```
  system("PAUSE");
  return 0;
}

long MyStrLen(char str[])
{
  long delka=0;
  while(str[delka]!= 0)
  {
   delka++;
  }
  return delka;
}
```

Příklad č.2

```
/*
 * Vytvořte funkci MyStrCpy, která řetězec s2
 * překopíruje do řetězce s1.
 * Pro zjednodušení bude mít funkce návratový typ void.
 */

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

long MyStrLen(char *);
void MyStrCpy(char *s1, char *s2);

int main(int argc, char *argv[])
{
  char s1[10],s2[10];

  printf("Zadej text: ");
  scanf("%s",s1);
  printf("\nPrvni retezec \"%s\". \n",s1);
  MyStrCpy(s2,s1);
  printf("\nDruhy retezec \"%s\". \n",s2);

  system("PAUSE");
  return 0;
}

void MyStrCpy(char *s1, char *s2)
{
  long delka2=MyStrLen(s2);
  long i;

  for (i=0;i<=delka2;i++)
    s1[i]=s2[i];
  //Nezapomeňte překopírovat koncovou nulu!!!
}

long MyStrLen(char str[])
{
  long delka=0;
  while(str[delka]!= 0)
```

```
  {
   delka++;
  }
  return delka;
}
```

Příklad č.3



Vytvořte projekt, který se bude skládat ze dvou modulů. První bude obsahovat funkce pro práci s řetězci, druhý modul bude obsahovat hlavní funkci, ve které odzkoušíte jednotlivé funkce. Funkce pro práci s řetězci

```
long MyStrLen(char *); - zjistí délku řetězce
void MyStrCpy(char *s1, char *s2); - kopíruje řetězec s2 do s1
void MyStrCat(char *s1, char *s2); - spojí dva řetězce
void MyStrRev(char *s); - převrátí řetězec
```

Nezapomeňte:

• Vytvořit hlavičkový soubor a vložit jej do příslušných zdrojových souborů. Uvědomte si ale, že hlavičkový soubor není součásti projektu (nevkládáme jej do projektu)!

Soubor main.c

```
/*
 * Modul s hlavní funkci - main.c
 */

#include <stdio.h>

#include "MyString.h"
//Připojení hlavičkového souboru

int main(int argc, char *argv[])
{
  char s1[10],s2[10];

  printf("Zadej text: ");
  scanf("%s",s1);
  printf("\nDelka retezce: \"%ld\". \n",MyStrLen(s1));
  printf("\nPrvni retezec \"%s\". \n",s1);
  MyStrCpy(s2,s1);
  printf("\nDruhy retezec \"%s\". \n",s2);
  MyStrCat(s1,s2);
  printf("\nSpojeny retezec \"%s\". \n",s1);
  MyStrRev(s1);
  printf("\nPrevraceny retezec \"%s\". \n",s1);

  return 0;
}
```

Soubor MyString.c

```
/*
 * Modul s funkcemi pro práci s řetezci - MyString.c
 */

#include "MyString.h"
```

```
void MyStrRev(char *s)
{
  long i;
  long delka=MyStrLen(s);

  for(i=0;i<delka/2;i++)
  {
    char pom=s[i];
    s[i]=s[(delka-i)-1];
    s[(delka-i)-1]=pom;
  }
}

void MyStrCat(char *s1, char *s2)
{
  long delka1=MyStrLen(s1);
  long delka2=MyStrLen(s2);
  long i;

  for (i=0;i<=delka2;i++)
    s1[delka1+i]=s2[i];

}
void MyStrCpy(char *s1, char *s2)
{
  long delka2=MyStrLen(s2);
  long i;

  for (i=0;i<=delka2;i++)
    s1[i]=s2[i];
  //Nezapomeňte překopírovat koncovou nulu!!!
}

long MyStrLen(char str[])
{
  long delka=0;
  while(str[delka]!= 0)
  {
   delka++;
  }
  return delka;
}
```

Soubor MyString.h

```
/*
 * Hlavičkový soubor - MyString.h
 */

 #ifndef _MYSTRING_H
 #define _MYSTRING_H

 long MyStrLen(char *);
 void MyStrCpy(char *s1, char *s2);
 void MyStrCat(char *s1, char *s2);
 void MyStrRev(char *s);

#endif
```



Opakovací test

1) Která z následujících funkcí překopíruje obsah jednoho řetězce do druhého?

char * strcpy(char *s1, char *s2);
char * strcat(char *s1, char *s2);
char * strstr(char *s1, char *s2);
char * strchr(char *s1, char s2);

2) Která z následujících funkcí připojí obsah jednoho řetězce k druhému řetězci?

char * strcpy(char *s1, char *s2);
char * strcat(char *s1, char *s2);
char * strstr(char *s1, char *s2);
char * strchr(char *s1, char s2);

3) Který z následujících zápisů je správný? Platí char str[10];

str = "Ahoj";
strcpy(str, "Ahoj");
strcpy(str,'Ahoj');
str[5] = "Ahoj";

4) Který znak zakončuje řetězec?

‘\0’
‘0’
0
48

Shrnutí učiva



• Řetězec je vlastně jednorozměrné pole, jehož prvky jsou proměnné typu **char**. V daném poli je však nutné nějakým způsobem určit ukončení řetězce. V jazyku C se toto provádí ukončovacím znakem `'\0'`. Hodnoty za tímto znakem se při standardní práci s řetězci nevyužívají. Definice řetězcové proměnné je stejná jako u jednorozměrných polí.

```
char str[5];  /* staticky */
char *pstr;   /* dynamicky */
pstr=(char*)malloc(5);
```

• Pozor! Pokud bychom zapsat zakončující znak `'\0'`, za řetězec by se považovaly všechny znaky v paměti až do prvního výskytu ukončujícího znaku!

• Funkce pro práci s řetězci:

```
int strlen(char * s);
char * strcpy(char *s1, char *s2);
char * strcat(char *s1, char *s2);
int strcmp(char *s1, char *s2);
int atoi(char *s);
int atol(char *s);
int atof(char *s);

scanf("%s", str);
gets(str);
```

10. STRUKTURY, UNIONYA VÝČTOVÉ TYPY

Cílem lekce je přiblížit práci s dalšími datovými typy a stukturou, unionem a výčtovým typem. Naučíte se typy definovat, vytvářet statické i dynamické proměnné daných typů, tvořit parametry funkcí z daných typů.

Budete:

- vědět, jak definovat datový typ struct, union a enum
- umět pracovat s typy struct, union a enum
- umět využívat ve svých programech datové typy struct, union a enum
- vědět jak pracovat s výše uvedenými typy ve funkcích

Klíčová slova této kapitoly:
Struct, union, enum



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Struktury

Datový typ *stuktura* částečně odpovídá datovému typu *record* (záznam) z Pascalu. Položky *strukutry* mohou být heterogenní, tedy různých datových typů. Strukturu můžeme v programovacím jazyku definovat několika různými způsoby. Klíčové slovo pro definování struktury je *struct*.

```
struct
{
char jmeno[15];
char adresa[30];
int vek;
}Jirka, Adam, Pavel;
```

Tento způsob má několik nevýhod. Struktura nemá své jméno, a proto ji není již možné použít pro definovaní dalších proměnných. Identifikátory `Jirka, Adam, Pavel` jsou názvy proměnných.

```
struct osoba
{
char jmeno[15];
char adresa[30];
int vek;
}Jirka, Adam, Pavel;
```

Identifikátory Jirka, Adam, Pavel jsou opět názvy proměnných, ale struktura má již své jméno a je tedy možné definovat další proměnné a to následujícím způsobem.

```
struct osoba Karel;
```

Klíčové slovo *struct* nesmí chybět!

```
typedef struct
{
  char jmeno[15];
  char adresa[30];
  int vek;
}OSOBA;
```

Identifikátor `OSOBA` je název nového datového typu a další proměnné se definují takto:

```
OSOBA Karel, Pavel;
```

Poslední možností definice je kombinace předcházejících dvou způsobů, například pro případ definování prvku pro dynamický spojového seznam.

```
typedef struct list
{
  int value;
  struct list * next;
}LIST;
```

K jednotlivým položkám proměnné typu struktura přistupujeme pomocí tečkové konvence. Například:

```
typedef struct
{
  char jmeno[15];
  char adresa[30];
  int vek;
}OSOBA;

OSOBA Karel, Pavel;

strcpy(Karel.adresa, "Ostrava");
Karel.vek = 18;
Pavel = Karel;
```

V případě definování ukazatele na strukturu je potřeba dávat pozor na prioritu operátorů.

```
OSOBA *s;
s = (OSOBA*)malloc(sizeof(OSOBA));
*s.vek = 13; /* špatně!!! */
(*s).vek = 13; /* správně */
```

Jazyk C umožňuje ještě jiný způsob zápisu, kterému je vhodné dávat přednost, neboť je přehlednější.

```
s->vek = 13;
```

Struktury lze podobně jako pole inicializovat na počáteční hodnoty jednotlivých položek. Příklad:

```
typedef struct
{
  char c;
  int i;
  float f;
}SH;
SH a = { 10, 's', 3.14 };
```

Velikost proměnné typu SH je rovna součtu `sizeof(char) +sizeof(int)+sizeof(float)`. Jednotlivé položky začínají na různých adresách a leží v paměti za sebou. Adresy se liší o velikost jednotlivých položek struktury.

```
&a rovno &a.c
&a.i rovno (&a.c + sizeof(char))
&a.f rovno (&a.i + sizeof(int))
```

Sturktury a funkce

Datový typ struktura může být jak datovým typem parametrů funkce, tak rovněž i návratovým typem funkce. Jako klasický vzorový příklad vytvořme funkci pro sčítání dvou komplexních čísel.

```
     typedef struct
{
  float re, im;
}COMPLEX;

COMPLEX soucet(COMPLEX c1, COMPLEX c2)
{
  COMPLEX pom;
  pom.re = c1.re + c2.re;
  pom.im = c1.im + c2.im;
  return pom;
}

void main(void)
{
  COMPLEX a = {10.5 , 1.0}, b = { 5.5 , 2.0}, c;
  c = soucet(a,b);
}
```

Funkci pro sčítání dvou komplexních čísel bychom mohli napsat i následovně:

```
void soucet(COMPLEX *c1, COMPLEX *c2, COMPLEX *vys)
{
  vys->re = c1->re + c2->re;
  vys->im = c1->im + c2->im;
}

void main(void)
{
  COMPLEX a = {10.5 , 1.0}, b = { 5.5 , 2.0}, c;
  soucet(&a,&b,&c);
}
```

Tento způsob není v případě sčítaní dvou komplexních čísel moc vhodný, neboť se nedá přímo využít ve složitějších výrazech. Jeho výhoda je však v šetření pamětí a to hlavně u rozsáhlejších datových struktur.

Union

Tento datový typ **union** odpovídá přibližně variantnímu záznamu v Pascalu. Na rozdíl od struktury, jejíž velikost se rovná součtu velikostí jednotlivých položek, je velikost unionu rovna velikosti největší položky. Definice se provádí podobným způsobem jako u struktury. Klíčové slovo je *union*.

```
typedef union
{
  char c;
  int i;
  float f;
```

```
}UH;

UH hod;
hod.c = 's';
hod.i = 123; /* přepíše hodnotu 's' */
hod.f = 4.5678; /* přepíše hodnotu 123 */
```

Všechny položky začínají na stejné adrese, která je shodná s adresou proměnné "hod". Velikost této proměnné je rovna velikosti největší položky, tedy `sizeof(float)`. Naproti tomu následující datový typ má velikost rovnu součtu `sizeof(char) +sizeof(int)+sizeof(float)`. Jednotlivé položky začínají na různých adresách a leží v paměti za sebou.

```
typedef struct
{
  char c;
  int i;
  float f;
}SH;
```

Výčtový typ

Výčtový typ slouží k definování seznamu symbolických konstant a definice se provádí podobně jako u struktury. Klíčovým slovem je *enum*.

```
typedef enum
{
  CERNA, MODRA, ZELENA, CERVENA, BILA
}BARVY;

BARVY b;
b = MODRA; /* b = 1; */
```

Hodnoty jednotlivých symbolických konstant začínají nulou a postupně se zvětšují o jedna. Lze je také nastavit na specifické celočíselné hodnoty.

```
typedef enum
{
  CERNA=1, MODRA=3, ZELENA=10, CERVENA, BILA
}BARVY;
```



Kontrolní úkol:

Zamyslete se nad možností použití

```
typedef enum
{
  false, true
}bool;
nebo
#define false 0
#define true 1
```

Příklady

Příklad č. 1



Vytvořte strukturu a union podle níže uvedeného zadání. Pro každý datový typ definujte proměnnou a na obrazovku vypište adresy proměnných a jejich jednotlivých položek.

```
typedef union
{
  char c;
  int i;
  float f;
}UH;

typedef struct
{
  char c;
  int i;
  float f;
}SH;
```

Příklad č. 2

Vytvořte strukturu `Osoba`, ve které budou položky `jmeno`, `prijmeni`, `vek`. Vytvořte jednorozměrné pole osob, zadejte několik údajů a pole setřiďte podle věku osob.

Příklad č. 3

Vytvořte program, který bude obsahovat funkce pro práci s komplexními čísly. Komplexní číslo je číslo ve tvaru: "reálna_složka + imaginární_složka * i". Pro definici komplexního čísla budou proměnné typu "float". Ve funkci main vyzkoušejte každou z vámi vytvořených funkcí. Vytvořte funkce:

`KOMPLEX Soucet(KOMPLEX c1, KOMPLEX c2);`

která vrátí součet dvou komplexních čísel. Matematický vztah zní:

`c1 + c2 = (re1 + re2) + (im1 + im2)*i, kde c1=re1+im1*i a c2=re2+im2*i`

`KOMPLEX Rozdil(KOMPLEX c1, KOMPLEX c2);`

která vrátí rozdíl dvou komplexních čísel. Matematický vztah zní:

`c1 - c2 = (re1 - re2) + (im1 - im2)*i, kde c1=re1+im1*i a c2=re2+im2*i`

Příklad č.4



```
/*
 * Vytvořte datový typ TStudent, který bude obsahovat jméno,
 * věk a studijní průměr studenta.
 * Vytvořte proměnnou, načtěte a vypište.
 */

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

```
typedef struct
{
  char jmeno[20];
  int vek;
  float studijni_prumer;
}TStudent;

int main(int argc, char *argv[])
{
  TStudent s1;

  printf("Zadej jmeno studenta: ");
  scanf("%20s",s1.jmeno);
  //zde adresní operátor nepatří, jmeno je pole
  printf("Zadej vek studenta: ");
  scanf("%d",&s1.vek);// nezapomeňte adresní operátor
  printf("Zadej studijni prumer studenta: ");
  scanf("%f",&s1.studijni_prumer);

  printf("%s, vek = %d, studijni prumer = %4.2f",s1.jmeno,
s1.vek,
                s1.studijni_prumer);
  system("PAUSE");
  return 0;
}
```



Příklad č.5

```
/*
 * Vytvořte datový typ TStudent, který bude obsahovat jméno,
 * věk a studijní průměr studenta.
 * Vytvořte funkce pro práci s jednoduchým seznamem studentů
 * umístěných ve statickém poli.
 */

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

typedef struct
{
  char jmeno[20];
  int vek;
  float studijni_prumer;
}TStudent;

#define POCET 3

void NacistStudenta(TStudent *);
void VypisStudenta(TStudent *);
void TriditPodleJmena(TStudent *, int);
void TriditPodlePrumeru(TStudent *, int);

int main(int argc, char *argv[])
{
  TStudent seznam[POCET];
  int i;
```

```
  printf("Nacteni studentu do pole:\n");
  for (i=0;i<POCET;i++)
    NacistStudenta(&seznam[i]);

  printf("\nVypis studentu z pole:\n");
  for (i=0;i<POCET;i++)
    VypisStudenta(&seznam[i]);

  TriditPodleJmena(seznam, POCET);
  printf("\nVypis setridenych studentu podle jmena:\n");
  for (i=0;i<POCET;i++)
    VypisStudenta(&seznam[i]);

  TriditPodlePrumeru(seznam, POCET);
  printf("\nVypis setridenych studentu podle studijniho
prumeru:\n");
  for (i=0;i<POCET;i++)
    VypisStudenta(&seznam[i]);

  system("PAUSE");
  return 0;
}

void NacistStudenta(TStudent *s)
{
  printf("Zadej jmeno studenta: ");
  scanf("%20s",s->jmeno); //zde adresní operátor nepatří,
jmeno
je pole
  printf("Zadej vek studenta: ");
  scanf("%d",&s->vek);// nezapomeňte adresní operátor
  printf("Zadej studijni prumer studenta: ");
  scanf("%f",&s->studijni_prumer);
}

void VypisStudenta(TStudent *s)
{
  printf("\nJmeno studenta: %s",s->jmeno);
  printf("\nVek studenta: %d",s->vek);
  printf("\nStudijni prumer studenta: %f",s->studijni_prumer);
}

void TriditPodleJmena(TStudent p[], int pocet)
{
  /* Nezapomeňte, že chcete třídit řetězce, tedy pole a není
   * možné přímo použít operátor menší či větší. Jinak bychom
   * porovnávali adresy. Použijeme funkci pro lexikografické
   * porovnání strcmp. Předpokládáme, že nepoužíváme české
   * znaky a není zmatek ve velkých a malých písmenech!
   */
  int i,j;
  for (i=0;i<pocet;i++)
   for (j=0;j<pocet-1;j++)
     if (strcmp(p[j].jmeno,p[j+1].jmeno)>0)
     {
```

```
      /* Nezapomeňte přehazovat celé struktury ne jen jména!
*/
       TStudent pom=p[j];
       p[j]=p[j+1];
       p[j+1]=pom;
     }
}
void TriditPodlePrumeru(TStudent p[], int pocet)
{
  int i,j;
  for (i=0;i<pocet;i++)
   for (j=0;j<pocet-1;j++)
     if (p[j].studijni_prumer > p[j+1].studijni_prumer)
     {
      /* Nezapomeňte přehazovat celé struktury ne jen jména!
*/
       TStudent pom=p[j];
       p[j]=p[j+1];
       p[j+1]=pom;
     }
}
```



Příklad č.6

```
/*
 * Prohlédněte si adresy položek struktury a unionu
 */

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

typedef struct
{
  char c;
  int i;
  float f;
}STRUCT;

typedef union
{
  char c;
  int i;
  float f;
}UNION;

int main(int argc, char *argv[])
{
  STRUCT s;
  UNION u;

  printf("Adresa promenne typu struktura je %p\n", &s);
  printf("Adresa prvni polozky struktury je %p\n", &s.c);
  printf("Adresa druhe polozky struktury je %p\n", &s.i);
  printf("Adresa treti polozky struktury je %p\n", &s.f);

  printf("Adresa promenne typu union je %p\n", &u);
  printf("Adresa prvni polozky unionu je %p\n", &u.c);
```

```
  printf("Adresa druhe polozky unionu je %p\n", &u.i);
  printf("Adresa treti polozky unionu je %p\n", &u.f);

  system("PAUSE");
  return 0;
}
```

Opakovací test



1) Čemu je rovná velikost datového typu struktura?
   - součtu velikostí jednotlivých položek
   - velikosti největší položky
   - počtu položek ve struktuře
   - součtu velikostí největší a nejmenší položky

2) Čemu je rovná velikost datového typu union?
   - součtu velikostí jednotlivých položek
   - velikosti největší položky
   - počtu položek ve struktuře
   - součtu velikostí největší a nejmenší položky

3) Čemu je rovná první položka výčtového typu, není-li určeno jinak?
   0
   1
   názvu položky
   libovolné hodnotě

4) Která z následujících variant je správná?
   - všechny prvky unionu začínají na stejné adrese
   - každý prvek unionu začíná na jiné adrese, prvky leží za sebou
   - každý prvek unionu začíná na jiné adrese, prvky neleží za sebou
   - všechny prvky struktury začínají na stejné adrese

5) Která z následujících variant je správná?
   - všechny prvky struktury začínají na stejné adrese
   - každý prvek struktury začíná na jiné adrese, prvky leží za sebou
   - každý prvek unionu začíná na jiné adrese
   - každý prvek struktury začíná na jiné adrese, prvky neleží za sebou

Shrnutí učiva



• Struktura je heterogenní datový typ - položky mohou být různého datového typu. Jednotlivé položky začínají na různých adresách a leží v paměti za sebou. Klíčovým slovem pro definování struktury je *struct*.
• Datový typ má velikost rovnu součtu `sizeof(char) +sizeof(int)+sizeof(float)`.

```
typedef struct
{
char c;
int i;
float f;
}SH;
```

• Klíčovým slovem pro definování unionu je *union*. Všechny položky unionu začínají
na stejné adrese, která je shodná s adresou proměnné "hod". Velikost této proměnné typu
union je rovna velikosti největší položky, tedy `sizeof(float)`.

```
typedef union
{
char c;
int i;
float f;
}UH;
```

• Výčtový typ slouží k definování seznamu symbolických konstant. Klíčovým slovem je *enum*.

```
typedef enum
{
CERNA, MODRA, ZELENA, CERVENA, BILA
}BARVY;
BARVY b;
b = MODRA; /* b = 1; */
```

Hodnoty jednotlivých symbolických konstant začínají nulou a postupně se zvětšují o jedna. Lze je také nastavit na specifické celočíselné hodnoty.

11. ODDĚLENÝ PŘEKLAD. PAMĚŤOVÉ TŘÍDY.

Cílem lekce je přiblížit práci s moduly u větších projektů. Další náplní lekce jsou paměťové třídy a jejich vztah k definování lokálních a globálních proměnných.

Budete:

- vědět, jak vytvářet a definovat jednotlivé moduly projektu
- umět správně vytvářet projekty z více částí
- **vědět, jak definovat lokální a globální proměnné**

Klíčová slova této kapitoly:
Auto, extern, static, register, paměťový modul, lokální a globální proměnná



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

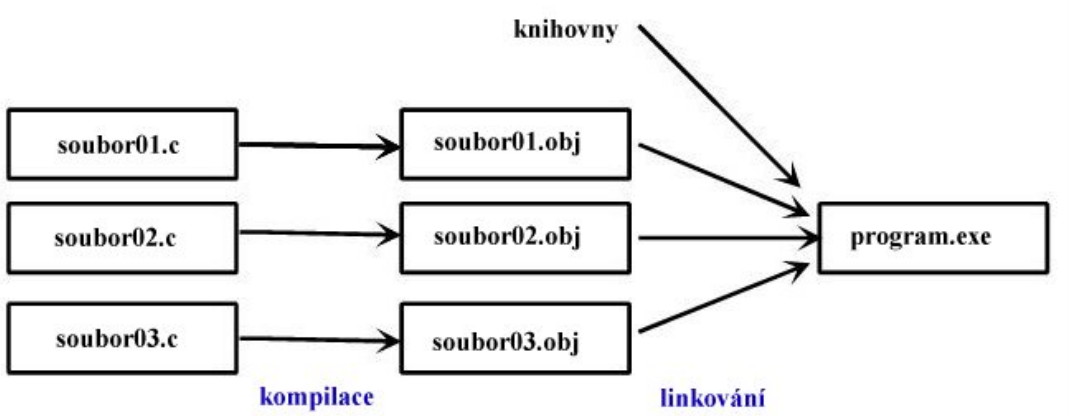
Programovací jazyk C umožňuje modulární přístup k tvorbě programů. Rozsáhlejší aplikace není vhodné z mnoha důvodů vytvářet v jednom zdrojovém souboru. Zdrojový kód se rozděluje na jednotlivé části – moduly, které jsou do jisté míry samostatné.

Výhody rozdělení programu na moduly:

- Kratší kód je přehlednější.
- Na aplikaci může pracovat více programátorů najednou.
- Zrychlí se překlad zdrojových kódů – překládá se pouze upravovaný zdrojový soubor. Soubory, které již byly opraveny a přeloženy se nemusí opětovně překládat.
- Jednotlivé moduly lze využít i v jiných projektech – aplikacích.

Dělení programu na několik souborů musí mít samozřejmě svou jasnou logiku. Nelze kód rozmisťovat nahodile do různých souborů. Jednotlivé moduly – zdrojové soubory musí obsahovat takové funkce, které spolu logicky souvisí (např. modul s matematickými funkcemi, modul pro grafickou podporu, modul pro vstupy a výstupy dat, modul s hlavní funkcí…).

Jednotlivé moduly (zdrojové soubory) se nejprve samostatně překompilují například do tvaru *obj*. Pak se slinkují do *exe* tvaru. Je potřeba zdůraznit, že pouze jeden z modulů může obsahovat funkci **main**. Téměř každý zdrojový soubor by měl mít příslušný hlavičkový soubor, ve kterém budou deklarace proměnných, typů a hlavičky funkcí.



Paměťové třídy

Auto

Paměťová třída auto je implicitní pro lokální proměnné. Klíčové slovo **auto** není nutné uvádět. Proměnné se definují v zásobníku a jsou přístupné pouze ve funkci, ve které byly definovány. Po ukončení provádění funkce, jsou proměnné zrušeny.

```
int funkce(void)
{
   auto int a;  //stejné jako int a;
   int i;  //stejné jako auto int i;
   . . .
}
```

Extern

Paměťová třída extern je implicitní pro globální proměnné. Klíčové slovo **extern** se uvádí pouze u deklaraci proměnných v ostatních modulech.

```
/*
 * modul, ve kterém se definuje globální proměnná
 *
 */
int global = 0; //definice globální proměnné

/*
 * modul, ve kterém se nedefinuje globální proměnná
 *
 */
extern int global; //deklarace globální proměnné
```

Static

Tato paměťová třída není implicitní pro žádný typ proměnných. Klíčové slovo se tedy musí vždy uvádět. Definujeme-li lokální proměnné jako *static*, pak se proměnné nevytváří v zásobníku jako běžné lokální proměnné, ale ve stejné oblasti paměti jako globální proměnné. Na rozdíl od nich však jsou však lokální proměnné viditelné jen ve funkcích.

Zpracujte následující program a podívejte se na jeho výpis.

Příklad



```
/*
 * lokální proměnné typu static
 *
 */
#include <stdio.h>

void VypisPromennych(void)
{
  auto int a = 0;
  static int s = 0;

  a++;
  s++;
  printf("a = %d, s =%d",a,s);
}

void main(void)
{
  int i;

  for (i=0;i<10;i++)
    VypisPromennych();
}
```

Definujeme-li jako **static** globální proměnné, pak jsou tyto proměnné použitelné jen v definovaném modulu. V rámci jednoho projektu můžeme definovat více globálních proměnných se stejným jménem. Toho se využívá v případě, že více programátorů pracuje na jednom projektu nebo definujeme-li globální proměnnou, kterou nechceme využívat v dalších modulech.

Registr

Paměťová třída registr se využívá u lokálních proměnných. Klíčové slovo **registr** je doporučuje překladači, aby umístil proměnnou do registru procesoru. Proměnná musí mít maximálně velikost jako registr (například 32bitů).

Příklad



Porovnejte časy provádění následujících programů. V prvním jsou proměnné globální a ve druhém jsou ukládány v registrech.

```
/*
 * Příklad č.1
 * globální proměnné
 */
#define POCET 5000

int i,j;
int main()
{
  for (i=0;i<=POCET;i++)
   for (j=0;j<=POCET;j++)
```

```
    {
      i=i;
      j=j;
    }
  return 0;
}
/*
 * Příklad č.2
 * lokální proměnné v registrech
 */
#define POCET 5000

int i,j;
int main()
{
  register int i,j;
  for (i=0;i<=POCET;i++)
   for (j=0;j<=POCET;j++)
    {
      i=i;
      j=j;
    }
  return 0;
}
```



Kontrolní úkol:

Jaké výhody má rozdělení velkého programu na více modulů?

Příklad

Vytvořte program, který se bude skládat ze dvou modulů. V prvním modulu definujte následující funkce:

int MyAbs(int cislo);

- funkce vrátí absolutní hodnotu čísla ‘cislo’

float Fakt(int cislo);

- funkce vrátí faktoriál čísla ‘cislo’

Druhý modul bude obsahovat funkci main, ve které načtete dvě čísla a vypíšete jejich absolutní hodnotu a velikost faktoriálu. K prvnímu modulu vytvořte i hlavičkový soubor.

Opakovací test

1) Která paměťová třída je implicitní pro lokální proměnné?
- auto
- static
- extern
- register

2) Která paměťová třída je implicitní pro globální proměnné?
- auto
- static
- extern
- register



3) Kterou paměťovou třídu nelze využít pro proměnnou typu double float?
- auto
- static
- extern
- register

4) Který z následujících zápisů není správně?
- auto int x = 1;
- extern int g;
- static int s = 0;
- extern float f = 0;



Shrnutí učiva

• Programovací jazyk C umožňuje modulární přístup k tvorbě programů. Rozsáhlejší aplikace je vhodné rozdělit na více samostatných zdrojových souborů.

• Výhody rozdělení na moduly:
  Kratší kód je přehlednější.
  Na aplikaci může pracovat více programátorů najednou.
  Zrychlí se překlad zdrojových kódů – překládá se pouze upravovaný zdrojový soubor. Soubory, které již byly opraveny a přeloženy se nemusí opětovně překládat.
  Jednotlivé moduly lze využít i v jiných projektech – aplikacích.

• Paměťové třídy jsou:
  auto – pro lokální proměnné
  extern – pro globální proměnné
  static – pro lokální i globální proměnné
  registr – doporučení pro umístění lokálních proměnných do registrů

12. PRÁCE SE SOUBORY

Cílem této lekce je seznámit se se základními postupy při manipulaci s binárními i textovými soubory.

Budete:

- vědět, jaký jsou základní rozdíly mezi textovými a binárními soubory
- umět pracovat s funkcemi pro formátované čtení a zápis dat v souboru
- umět pracovat s funkcemi a makry pro neformátované čtení a zápis dat v souboru
- umět využívat přímého přístupu u binárních souborů

Klíčová slova této kapitoly:
binární soubor, buffer, handle, stream, textový soubor



Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Jazyk C umožňuje pracovat se soubory dvěma způsoby:
a) pomocí proudu (stream) - ukazatel na strukturu FILE
b) pomocí handle - handle je celočíselná hodnota typu **int**, která udává symbolickou hodnotu souboru

Tak jako jiné programovací nástroje, rovněž jazyk C může pracovat s *textovými* nebo *binárními* soubory. Základní rozdíl mezi oběma typy souborů je v tom, že v binárním souboru jsou data ukládána stejným způsobem jako v paměti. Zatímco data do textového souboru se musí převést na posloupnost znaků. Z toho pak vyplývají další rozdíly:

- binární soubory s uloženými stejnými hodnotami jako textové jsou většinou kratší
- v binárním souboru máme přímý přístup k jednotlivým položkám, zatímco v textovém se musíme pohybovat sekvenčně
- práce s binárními soubory je rychlejší než s textovými, neboť nedochází k převodu

na posloupnost znaků a naopak

- naopak binární soubory se nedají číst bez přesné znalosti zapsaných datových typů

nebo bez speciálního editoru jako textové soubory

Jazyk C nedělá rozdíly mezi soubory a ostatními zařízeními a přistupuje ke všem pomocí *stream* (proudů). Před vlastním otevřením souboru je potřeba nadefinovat ukazatel na strukturu FILE, která je definována v hlavičkovém souboru stdio.h. Pak použijeme funkci fopen(), která soubor otevře v příslušném módu. Pro uzavření souboru se používá funkce fclose().Jednotlivé módy otevření:

"r" - otevře jen existující soubor a data se z něj dají odkudkoliv číst

"w" - pokud otevře existující soubor, ten bude vymazán. Neexistuje-li daný soubor, bude vytvořen. Data se lze kamkoliv zapsat.

"a" - existující soubor bude rozšiřován, neexistující soubor bude vytvořen. Data lze pouze zapisovat na konec.

"r+" - otevře jen existující soubor a data se z něj dají odkudkoliv číst, ale i kamkoliv zapsat.

"w+" - pokud otevře existující soubor, ten bude vymazán. Neexistuje-li dan soubor, bude vytvořen. Data se lze kamkoliv zapsat, ale rovněž lze data odkudkoliv číst.

"a+" - existující soubor bude rozšiřován, neexistující soubor bude vytvořen. Data lze zapisovat pouze na konec, ale rovněž odkudkoliv číst.

Přidáním písmene "b" zadáváme požadavek na otevření v binárním módu. Pro textový mód uvádíme buď písmeno "t" nebo zapíšeme jen základní tvar uveden v tabulce.

```
FILE *f;
f = fopen("Soubor.bbb","wb");
. . .
fclose(f);
```

Ošetření uvedených funkcí se provádí následujícími způsoby. Otevření se zdaří jen v případě, že funkce vrátí konkrétní adresu, v opačném případě vrací NULL. Při správném zavírání souboru se vrací nulová hodnota, při nezdařeném pokusu o uzavření vrací funkce EOF (konec souboru). Na funkci pro uzavření souboru není dobré zapomínat, neboť zajišťuje korektní vyprázdnění bufferů a uložení jejich obsahů do příslušného souboru.



Častá chyba!

Chybný zápis cesty k souboru:

```
f=fopen("C:\new\muj.txt","r");
```

Nezapomeňte, že v uvozovkách je potřeba zdvojit obrácené lomítko! Správně:

```
f=fopen("C:\\new\\muj.txt","r");
```

Formátované čtení a zápis dat

Nejčastěji formátované vstupy a výstupy provádíme pomocí funkcí `fprintf()` a `fscanf()`. Uvedený postup má samozřejmě význam převážně jen u textových souborů.

```
int fprintf (FILE* filePrintTo, const char* szFormat, ...);
int fscanf (FILE* fileReadFrom, const char* szFormat, ...);
```

Neformátované čtení a zápis dat

Neformátované vstupy a výstupy se dělí podle množství zpracovaných dat:

- zpracování po jednom znaku
- zpracování po blocích
- zpracování po řádcích

Zpracování po jednom znaku

Tento postup se používá jak pro binární, tak i pro textové soubory. Pro práci se používají následující funkce a makra:

```
int fgetc(FILE *f);
int fputc(int c, FILE *f);
int getc(FILE *f);
int putc(int c, FILE *f);
```

Příklad:



Překopírujte obsah jednoho souboru do druhého.

```
#include <stdio.h>
int main(void)
{
      FILE *fo,*fk;
      int c;
      fo = fopen("c:\\Original.txt","r");
      fk = fopen("c:\\Kopie.txt","w");
      while ((c = getc(fo)) != EOF)
            putc(c,fk);
      fclose(fo);
      fclose(fk);
      return 0;
}
```

Zpracování po blocích

Používá se pouze pro binární soubory otevřené v binárním módu. Blok je vlastně pole o určitém počtu prvků dané velikosti. Blokový způsob zpracování se používá v souborech, kde se zpracovávají rozměrná data. Pro čtení a zápis se používají následující funkce:

```
int fread(char *cil, size_t rozmer, size_t pocet, FILE *fr);
int fwrite(char *zdroj, size_t rozmer, size_t pocet,FILE *fw);
```

Příklad:



Překopírujte obsah jednoho souboru do druhého.

```
#include <stdio.h>
#include <stdlib.h>
#define VELIKOST 10
int main(void)
{
      FILE *fo,*fk;
      char blok[VELIKOST];
      int p;
      fo = fopen("Original.bbb","rb");
      fk = fopen("Kopie.bbb","wb");
      while ((p=fread(blok,sizeof(char),VELIKOSTfo)) ==
VELIKOST)
      fwrite(blok, sizeof(char), VELIKOST, fk);
      fclose(fo);
      fclose(fk);
      return 0;
}
```

Zpracování po řádcích

Používá se pouze pro textové soubory. Pro čtení a zápis se využívají následující funkce

```
char *fgets(char *str, int m, FILE *f);
int fputs(char *str, FILE *f);
```



Příklad:

Překopírujte obsah jednoho souboru do druhého.

```
#include <stdio.h>
#define DELKA 20
int main(void)
{
      FILE *fo,*fk;
      char radek[DELKA];
      fo = fopen("Original.txt","r");
      fk = fopen("Kopie.txt","w");
      while (fgets(radek,DELKA,fo) != NULL)
      fputs(radek,fk);
      fclose(fo);
      fclose(fk);
      return 0;
}
```

Přímý přístup

Přímý přístup do souboru se používá jen u binárních souborů, v textových souborech je potřeba se pohybovat sekvenčně. Využívají se funkce:

int fseek(FILE *f, long posun, int odkud);

long ftell(FILE *f);

Funkce ftell() zjišťuje aktuální pozici v bajtech v daném souboru a funkce fseek() přesune ukazatel na zaktuální pozice na novou pozici v souboru.. Pro parametr odkud jsou předdefinovány následující hodnoty:

SEEK_SET od začátku souboru

SEEK_CUR od aktuální pozice

SEEK_END od konce souboru

V případě úspěšného provedení funkce `fseek()`, se vrací nula, v opačném případě nenulová hodnota.

Možnosti bufferování

Pro urychlení práce se soubory se data nezapisují přímo po 1 B, ale nejprve se zapíší do speciální části paměti *buffer*. Teprve až v příhodné chvíli se obsah bufferu přesune na diskové médium. Tento přesun se děje například v okamžiku, kdy je buffer plný. Přesun dat se dále provádí v případě uzavření souboru funkci `fclose()` nebo při ukončení programu.

Pokud potřebujeme uložit obsah bufferu na disk, přestože nenastala žádná z výše uvedených variant, je možné použít funkci `fflush()`.

int fflush(FILE *f);

V případě úspěšného přesunu dat z bufferu na disk vrací funkce nulu, v opačném případě vrací konstantu EOF.

Programovací jazyk C umožňuje ukládat data na disk i bez použití bufferu. Tato varianta je vhodná v případě, pracujeme-li například s rozsáhlými datovými typy, které mají větší velikost než velikost bufferu. Přímý způsob zápisu na disk je pak rychlejší.

Kontrolní úkol:



Kdy je při práci se soubory rychlejší nebufferovaný přístup než využívání mezipaměti? Jaké přístupové doby mají pevné disky a jaké operační paměti?

Příklady

Příklad č.1



```
/*
 * Načtěte deset celých čísel a uložte je do souboru.
 */

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

#define POCET 10

int main(int argc, char *argv[])
{
  FILE *f;
  int i, cis;

  if ((f=fopen("soubor.bin","wb"))==NULL)
  {
    printf("Chyba pri praci se souborem!");
    return 0;
  }

  for (i=0;i<POCET;i++)
  {
    printf("Zadej cele cislo: ");
    scanf("%d", &cis);
    fwrite(&cis,sizeof(int),1,f);
  }

  fclose(f);

  system("PAUSE");
  return 0;
}
```

Příklad č.2



```
/*
 * Načtěte celá čísla ze souboru a vypište na obrazovku.
 */

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


int main(int argc, char *argv[])
{
  FILE *f;
  int i, cis;

  if ((f=fopen("soubor.bin","rb+"))==NULL)
  {
    printf("Chyba pri praci se souborem!");
    return 0;
  }

  i=1;
  while (fread(&cis,sizeof(int),1,f)==1)
  {
    printf("\n%d. cele cislo: %d", i, cis);
    i++;
  }

  fclose(f);

  system("PAUSE");
  return 0;
}
```

Příklad č.3



```
/*
 * Načtěte celá čísla ze souboru. Bude-li
 * číslo liché, přičteme k němu jedničku,
 * bude-li sudé, necháme jej bezezmněny.
 * Čísla zapíšeme zpět do souboru.
 */

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


int main(int argc, char *argv[])
{
  FILE *f;
  int cis;

  if ((f=fopen("soubor.bin","rb+"))==NULL)
  {
    printf("Chyba pri praci se souborem!");
    return 0;
```

```
  }

  while (fread(&cis,sizeof(int),1,f)==1)
  {
    if (cis % 2 != 0)
    {
      cis +=1;
      //liché číslo zvětšíme o 1
      fseek(f,-sizeof(int),SEEK_CUR);
      //přesuneme se zpět na adresu, kde je číslo uloženo
      fwrite(&cis,sizeof(int),1,f);
      //zapíšeme změněné číslo
      fseek(f,0L,SEEK_CUR);
      /* Tato zdánlivě zbytečná operace je nutná, jinak
         program nepracuje správně! */
    }
  }

  fclose(f);

  system("PAUSE");
  return 0;
}
```

Příklad č.4



```
/*
 * Načtěte celá čísla ze souboru. Bude-li
 * číslo liché, přičteme k němu jedničku,
 * bude-li sudé, necháme jej bezezměny.
 * Čísla zapíšeme zpět do souboru.
 * K souboru přistupujte pomocí handelu!
 */

#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
  int f; //handel
  int cis;

  if ((f=open("soubor.bin",O_CREAT|O_RDWR|O_BINARY))== -1)
  {
    printf("Chyba pri praci se souborem!");
    return 0;
  }

 while (read(f,&cis,sizeof(int))!=0)
  {
    if (cis % 2 != 0)
    {
      cis +=1;
      //liché číslo zvětšíme o 1
      lseek(f,-sizeof(int),SEEK_CUR);
```

```
      //přesuneme se zpět na adresu, kde je číslo uloženo
      write(f,&cis,sizeof(int));
      //zapíšeme změněné číslo
    }
  }

  close(f);

  system("PAUSE");
  return 0;
}
```



Příklad č.5

```
#include <stdio.h>

int main(int argc, const char * argv[])
{

    FILE *f;
    int i;
    int p[] ={123,130,111,96,215,119,134,144,200};
    char s[][30] ={"Karel", "Pavel", "Adam",
      "Alena","Ludmila","Jan","Jana","Petr","Dana"};
    f = fopen("cisla.bin", "w");
    for(i=0; i<9;i++)
        fwrite(&p[i], sizeof(int),1,f);

    fclose(f);
    f= fopen("jmena.txt","w");
    for(i=0; i<9;i++)
        fprintf(f,"%s\n",s[i]);
    fclose(f);

    f = fopen("cisla.bin","r");
    while(fread(&i, sizeof(int),1,f) == 1)
        printf("%d ",i);

    fclose(f);

    printf("sizeof int %ld\n", sizeof(int));
    return 0;
}
```



Opakovací test

1) Ve kterém hlavičkovém souboru je definována struktura FILE?

stdio.h
files.h
stream.h
file.h

2) Které z následujících čísel bude v textovém souboru zabírat méně bytů než v binárním souboru? Předpokládáme dvoubytový typ int.

```
int x = 1;
int x = 1000;
int x = 10000;
int x = 10;
```

3) Které z následujících zápisů je správný?

```
fo = fopen("c:\\novy\\soubor.txt","r");
fo = fopen("c:\novy\soubor.txt","r");
fo = open("c:\\novy\\soubor.txt","r");
fo = fopen("soubor.txt","rw");
```

Shrnutí učiva



• Jazyk C umožňuje pracovat se soubory dvěma způsoby:
a) pomocí proudu (stream) - ukazatel na strukturu FILE
b) pomocí handle - handle je celočíselná hodnota typu **int**, která udává symbolickou hodnotu souboru

• Programovací jazyk C může pracovat s *textovými* nebo *binárními* soubory. Základní rozdíl mezi oběma typy souborů je v tom, že v binárním souboru jsou data ukládána stejným způsobem jako v paměti. Zatímco data do textového souboru se musí převést na posloupnost znaků.

• Při práci se soubory za pomocí struktury FILE můžeme čtení a zápis v souboru
provádět formátovaně nebo neformátovaně. Při neformátovaném přístupu můžeme s daty manipulovat po znacích, blocích či řádcích.

• Pro urychlení práce se soubory se data nezapisují přímo po 1 B, ale nejprve se zapíší do speciální části paměti *buffer*. Teprve až v příhodné chvíli se obsah bufferu přesune na diskové médium. Tento přesun se děje například v okamžiku, kdy je buffer plný. Přesun dat se dále provádí v případě uzavření souboru funkci fclose()nebo při ukončení programu. Pokud potřebujeme uložit obsah bufferu na disk, přestože nenastala žádná z výše uvedených variant, je možné použít funkci fflush().

13. DALŠÍ MOŽNOSTI JAZYKA C

Cílem této lekce je seznámit se s dalšími možnostmi programovacího jazyka C, jako jsou parametry funkce main, definování funkcí s proměnným počtem argumentů a bitová pole.

Budete:

- umět definovat a využívat parametry u hlavní funkce main
- umět definovat funkce s proměnným počtem argumentů
- umět definovat a využívat bitová pole



Klíčová slova této kapitoly:
bitová pole, bitové operace, funkce s proměnným počtem argumentů, parametry funkce main, výpustky

Čas potřebný k prostudování učiva kapitoly:
2 hodiny

Parametry funkce main

Stejně jako ostatní funkce, může mít i hlavní funkce **main** své parametry. První z nich je celočíselného typu int a označuje se obvykle argc. Pokud je jeho hodnota větší než 1, pak to znamená, že kromě názvu programu je připojen i nějak parametr spuštění. Druhý parametr se obvykle nazývá argv a jedná se o pole ukazatelů na typ char. Parametr obsahuje řetězce parametrů funkce main.

```
/* Program spusteny s urcitym parametrem nastavi kurzor */
#include <conio.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int i;
  if (argc>1){

    for (i=1; i<argc;i++)
     {
     if (strcmp(argv[i],"b")==0)  _setcursortype(_NOCURSOR);
     else
     if (strcmp(argv[i],"n")==0)
_setcursortype(_NORMALCURSOR);
     else
     if (strcmp(argv[i],"s")==0)
_setcursortype(_SOLIDCURSOR);

     }
  }
  else  _setcursortype(_NORMALCURSOR);
  return 0;
}
```

Funkce s proměnným počtem argumentů

Doposud jsme tvořili funkce, u kterých byl předem pevně stanovený počet a typ argumentů (parametrů funkce). Existují však funkce, u kterých se typ případně počet argumentů mění. Typickým příkladem jsou funkce *scanf( )*, *printf()*. Programovací jazyk C poskytuje mechanizmus, který umožňuje takovýto typ funkcí vytvářet. Hovoříme o *funkcích s proměnným počtem argumentů*.

Už víme, že hodnoty skutečných argumentů jsou umisťovány do zásobníku. Je tedy potřeba provést jejich jednoznačné přiřazení. Problémem však je, že díky deklaraci i volající místo ví, jakého typu očekává volaná funkce argumenty. Podle toho je předá. Při proměnném počtu argumentů je předání nesnadné. ANSI C definuje několik maker a pravidel pro tvorbu funkcí s proměnným počtem argumentů. Je však nutné stanovit mít mezi argumenty funkce nějak pevný bod, poslední pevný argument pro získání správného ukazatele do zásobníku. Další argumenty budeme deklarovat jako proměnné. To se zapisuje pomocí tří teček - *výpustky*. Pak použijeme standardní makra. Pro jednoduchost zpracování jsou typy hodnot předávaných skutečných argumentů konvertovány na *int* a double.

Makra pro proměnn počet argumentů jsou tři:

void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);

- va_list formálně představuje pole proměnných argumentů
- va_start nastaví ap na první z proměnných argumentů předaných funkci (probíhá fixace na poslední neproměnný deklarovaný a předaný argument lastfix)
- va_arg se jako makro rozvine ve výraz stejného typu a hodnoty, jako je další očekávaný argument (type)
- va_end umožní volané funkci provést případné operace pro bezproblémový návrat zpět - return

Nezapomeňte, že při použití těchto maker musí být volány v pořadí va_start - před prvním voláním va_arg nebo va_end. Po ukončení načítání by měl být volán va_end. Dále platí, že va_end může ap změnit tak, že při jeho dalším použití musíme znovu volat va_start. Počet skutečně předaných hodnot může být předán jako poslední neproměnný argument, nebo technikou nastavení zarážky.

```
#include <stdio.h>
#include <stdarg.h>
const int konec = -12345;

double polynom(char *msg, ...)
{
 double hodnota = 0.0, x;
 va_list ap;
 int koef;
 va_start(ap, msg);
 x = va_arg(ap, double);
 koef = va_arg(ap, int);
```

```
 while (koef != konec)
 {
      hodnota = hodnota * x + koef;
      koef = va_arg(ap, int);
 }
 va_end(ap);
 return hodnota;
}

int main(void)
{
 double f;
 char *s;
 s = "polynom stupne 2";
 f = polynom(s, 2.0, 2, 3, 4, konec);
      /* 2x^2 + 3x + 4,x=2 -> 18      */
 printf("%s = %lf\n", s, f);
 s = "dalsi polynom";
 f = polynom(s, 3.0, 1, 0, 0, 0, konec);
      /* x^3          ,x=3 -> 27      */
 printf("%s = %lf\n", s, f);
 return 0;
}
```

Bitové operace

Bitový součin

Bitový součin provede logický součin na jednotlivých bitech. Neboli porovnává bity v obou operandech na i-té pozici a pokud oba budou mít hodnotu 1, zapíše se do výsledné proměnné na i-tou pozici 1. V opačné případě zde napíše 0.

```
unsigned c,a=5,b=2;
c = a & b;
/* c = 0000 0101 & 0000 0010; */
/* vyjde c = 0000 0000 */
```

Bitový součet

Bitový součet provede logický součet na jednotlivých bitech. Neboli porovnává bity v obou operandech na i-té pozici a pokud alespoň jeden z nich bude mít hodnotu 1, zapíše se do výsledné proměnné na i-tou pozici 1. V opačné případě zde napíše 0.

```
unsigned c,a=5,b=2;
c = a | b;
/* c = 0000 0101 | 0000 0010; */
/* vyjde c = 0000 0111 */
```

Bitový exkluzivní součet

Tato operace (XOR) porovnává i-té bity v obou operandech a pokud se jejich hodnoty vzájemně nerovnají bude na i-tém bitu ve výsledku 1. Operace se dá využít při porovnávání čísel.

```
if ( a ^ b ) /* cisla mají ruznou hodnotu */
```

Operace bitového posunu

Bitový posun je operace, kdy se bity posunují doleva, případně doprava.

```
a = a << n;
```

Předcházející operace bitového posunu posune hodnoty bitů o "n" pozic doleva. Zleva se bity ztrácí, zprava se doplňují nulami. Následující příkaz naopak posune hodnoty jednotlivých bitů doprava. Vpravo se bity ztrácejí a zleva doplňují nulami.

```
a = a >> n;
```

Operace bitového posunu se dají využít jako násobení, případně dělení mocninou dvou. Takovéto násobení (dělení) je provedeno rychleji než běžné operace násobení či dělení.

```
b = a << 3;     /* provede b = a * 8; */
b = a >> 4;     /* provede b = a / 16; */
```

Negace bit po bitu

Všechny bity v proměnné se postupně negují.

```
~ a
```

Bitová pole

Jedná se vlastně o strukturu, která má velikost stejnou jako datový typ **int**. Jednotlivé položky pak zaujímají určit počet bitů. V určitých případech můžeme díky bitovému poli ušetřit jinak nevyužitou paměť.

```
typedef struct
{
  unsigned int den;
  unsigned int mes;
  unsigned int rok;
}DATUM;
```

V takto definované struktuře, zabírá každá položka sizeof(unsigned int), což je vzhledem k požadavkům na hodnotu dne, měsíce a roku zbytečné.

```
typedef struct
{
  unsigned int den : 5;
  unsigned int mes : 4;
  unsigned int rok : 7;
}DATUM;
```

Nyní celá struktura zabírá pouze sizeof(unsigned int), v našem případě 2 B, tedy 16 bitů. K jednotlivým položkám přistupujeme stejně jako v jiných proměnných typu struktura.

```
DATUM dnes;
dnes.den = 12;
dnes.mes = 10;
dnes.rok = 1999 - 1980;
```

Příklady



Příklad č. 1

Vytvořte program, který vytvoří kopii určitého souboru. Jména originálního souboru i jeho kopie zadejte jako parametry programu při spuštění (parametry funkce main).

Příklad č. 2

Vytvořte funkci s proměnným počtem argumentů, která spojí řetězce do jednoho. Počet řetězců bude u funkce proměnný.

Příklad č. 3

Ukázky využití funkcí s proměnným počtem argumentů.

```
//  main.c
//  fceProm01
//
#include <stdio.h>
#include <stdarg.h>

const int konec = -1;

double polynom(char *msg, ...)
{
    double hodnota = 0.0, x;
    va_list ap;
    int koef;
    va_start(ap, msg);
    x = va_arg(ap, double);
    koef = va_arg(ap, int);
    while (koef != konec)
    {
        hodnota = hodnota * x + koef;
        koef = va_arg(ap, int);
    }
    va_end(ap);
    return hodnota;
}
int main (int argc, const char * argv[])
{
    double f;
    char *s;
    s = "polynom stupne 2";
    f = polynom(s, 2.0, 2, 3, 4, konec);
    /* 2x^2 + 3x + 4,x=2 -> 18      */
    printf("%s = %lf\n", s, f);
    s = "dalsi polynom";
    f = polynom(s, 3.0, 1, 0, 0, 0, konec);
    /* x^3          ,x=3 -> 27      */
    printf("%s = %lf\n", s, f);
    return 0;
}
```

```
//  main.c
//  fceProm02
//
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

void spojString(char *cil,...)
{
    va_list arg;
    char *str;
    cil[0]= '\0';

    va_start(arg, cil);

    while((str= va_arg(arg,char *)) != NULL)
        strcat(cil, str);

    va_end(arg);
}
int main (int argc, const char * argv[])
{
    char text[200];

    spojString(text,"prvni",NULL);
    printf("%s\n",text);

    spojString(text,"druhy ","treti ","ctvrty ",NULL);
    printf("%s\n",text);

    spojString(text,"neco
","jine","sssss","sddddd","ttttt",NULL);
    printf("%s\n",text);

    return 0;
}



//  main.c
//  fceProm03
//
#include <stdio.h>
#include <stdarg.h>


void obvodNuhelnik(int *obvod,...)
{
    va_list arg;
    int strana;
    *obvod = 0;

    va_start(arg, obvod);

    while((strana= va_arg(arg,int)) != 0)
```

```
        *obvod += strana;

    va_end(arg);
}
int obvodN2uhelnik(int pocet,...)
{
    va_list arg;
    int strana;
    int obvod = 0;

    va_start(arg, pocet);

    while(pocet > 0)
    {
        strana= va_arg(arg,int);
        obvod += strana;
        pocet--;
    }
    va_end(arg);
    return obvod;
}

int main (int argc, const char * argv[])
{
    int obvod;

    obvodNuhelnik(&obvod, 1,2,3,4,5,6,6,0);
    printf("Obvod je %d\n",obvod);

    obvod = obvodN2uhelnik(4,1,2,3,4);
    printf("Obvod je %d\n",obvod);

    return 0;
}
```

Opakovací test



1) Které z následujících maker při definic funkce s proměnným počtem argumentů musí být voláno jako první?

- va_start
- va_list
- va_arg
- va_end

2) Jaké typu je parametr funkce main, který se obvykle označuje jako argv?

- char *argv[ ]
- char argv[ ]
- int *argv[ ]
- int argv

3) Při které bitové operaci se zleva bity ztrácejí a zprava doplňují nulami?

- bitový posun do leva
- bitový posun do prava
- bitový součet
- bitový součin

4) Který z následujících příkladů může nahradit násobení proměnné x šestnácti?

x << 4
x << 16
x >> 4
x >> 2



Shrnutí učiva

· Stejně jako ostatní funkce, může mít i hlavní funkce **main** své parametry. První z nich je celočíselného typu int a označuje se obvykle argc. Pokud je jeho hodnota větší než 1, pak to znamená, že kromě názvu programu je připojen i nějak parametr spuštění. Druhý parametr se obvykle nazývá argv a jedná se o pole ukazatelů na typ char. Parametr obsahuje řetězce parametrů funkce **main**.

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

· Programovací jazyk C poskytuje mechanizmus, který umožňuje vytvářet funkce, kde počet a typ parametrů není předem jednoznačně určen. Hovoříme o *funkcích s proměnným počtem argumentů*.

Jazyk C definuje několik maker a pravidel pro tvorbu funkcí s proměnným počtem argumentů. Je však nutné stanovit mít mezi argumenty funkce nějak pevný bod, poslední pevný argument pro získání správného ukazatele do zásobníku. Další argumenty budeme deklarovat jako proměnné. To se zapisuje pomocí tří teček - *výpustky*. Pak použijeme standardní makra. Pro jednoduchost zpracování jsou typy hodnot předávaných skutečných argumentů konvertovány na *int* a double.

Makra pro proměnn počet argumentů jsou tři:

void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);

· Bitový součin provede logický součin na jednotlivých bitech. Neboli porovnává bity v obou operandech na i-té pozici a pokud oba budou mít hodnotu 1, zapíše se do výsledné proměnné na i-tou pozici 1. V opačné případě zde napíše 0.

c = a & b;

· Bitový součet provede logický součet na jednotlivých bitech. Neboli porovnává bity v obou operandech na i-té pozici a pokud alespoň jeden z nich bude mít hodnotu 1, zapíše se do výsledné proměnné na i-tou pozici 1. V opačné případě zde napíše 0.

c = a | b;

· Bitový exkluzivní součet (operace XOR) porovnává i-té bity v obou operandech a pokud se jejich hodnoty vzájemně nerovnají bude na i-tém bitu ve výsledku 1. Operace se dá využít při porovnávání čísel.

```
if ( a ^ b ) /* cisla mají ruznou hodnotu */
```

· Bitový posun je operace, kdy se bity posunují doleva, případně doprava.

```
a = a << n;
```

Předcházející operace bitového posunu posune hodnoty bitů o "n" pozic doleva. Zleva se bity ztrácí, zprava se doplňují nulami. Následující příkaz naopak posune hodnoty jednotlivých bitů doprava. Vpravo se bity ztrácejí a zleva doplňují nulami.

```
a = a >> n;
```

· Operace bitového posunu se dají využít jako násobení, případně dělení mocninou dvou. Takovéto násobení (dělení) je provedeno rychleji než běžné operace násobení či dělení.

```
b = a << 3; /* provede b = a * 8; */
b = a >> 4; /* provede b = a / 16; */
```

· Bitová negace - všechny bity v proměnné se postupně negují.

```
~ a
```

· Bitová pole - jedná se vlastně o strukturu, která má velikost stejnou jako datový typ int. Jednotlivé položky pak zaujímají určit počet bitů. V určitých případech můžeme díky bitovému poli ušetřit jinak nevyužitou paměť.

14. TŘÍDÍCÍ ALGORITMY

V této kapitole se dozvíte:

- Jak vytvářet různé druhy třídících funkcí v programovacím jazyce C.

Budete:

- Umět zapsat nejznámější typy třídících a algoritmů v jazyce C
- Vyzkoušíte si algoritmy SelectSort, InsertSort, binární zatřídění, BubleSort, ShakerSort, QuickSort, HeapSort, MergeSort

Klíčová slova této kapitoly:

Třídící algoritmy, SelectSort, InsertSort, binární zatřídění, BubleSort, ShakerSort, QuickSort, HeapSort, MergeSort

Čas potřebný k prostudování učiva kapitoly:
3 hodiny





Příklad

Vytvořte funkci pro třídění pole pomocí algoritmu SelectSort.

```
void SelectSort(int p[],int n)   //výběr maximem
{
  int k,i,j;
  int max;

  for (i=n-1;i>0;i--)
  {
    max=p[i];
    k=i;
    for (j=i-1;j>-1;j--)
      if (p[j]>max)
      {
     max=p[j];
     k=j;
      }
    p[k]=p[i];
    p[i]=max;
  }
}
```



Příklad

Vytvořte funkci pro třídění pole pomocí algoritmu InsertSort.

```
void InsertSort(int p[],int n)
{
  //zatřídění    - pole musí mít o jeden prvek více!
  //pole je nutné vypisovat od 1 do n!!!!
  int i,j;
  int pom;
```

```
  for (i=1;i<=n;i++)
  {
    pom=p[i];
    j=i-1;
    p[0]=pom;          //nastaveni nárazníku p[0]
    while (pom<p[j])
    {
      p[j+1]=p[j];
      j--;
    }
    p[j+1]=pom;
  }
}
```

Příklad



Vytvořte funkci pro třídění pole pomocí algoritmu binární zatřídění.

```
void bin_zatrideni(int p[],int n)
{
  int i,j,m;
  int L,R; //měl by být raději dvojnásobný rozsah
  int pom;

  for (i=1;i<n;i++)
  {
    pom=p[i];
    L=0;
    R=i-1;
    while (L<=R)
    {
      m=(L+R) / 2;
      if (pom<p[m]) R=m-1;
       else L=m+1;
    }
    for (j=i-1;j>L-1;j--) p[j+1]=p[j];
    p[L]=pom;
  }
}
```

Příklad



Vytvořte funkci pro třídění pole pomocí algoritmu BubleSort.

```
void BubleSort(int p[],int n)
{
  int i,j;
  int pom;

  for (j=n-1;j>-1;j--)
    for (i=0;i<j;i++)
      if (p[i]>p[i+1])
      {
     pom=p[i];
     p[i]=p[i+1];
```

```
      p[i+1]=pom;
       }
}


void BubleSort_2(int p[],int n)
{
  int i,j,k;
  int pom;

  k=n-2;
  do
  {
    j=0;
    for (i=0;i<k+1;i++)
      if (p[i]>p[i+1])
      {
     pom=p[i];
     p[i]=p[i+1];
     p[i+1]=pom;
     j=i;
      }
    k=j-1;
  }while (k!=-1);
}
```

Příklad



Vytvořte funkci pro třídění pole pomocí algoritmu ShakerSort.

```
void ShakerSort(int p[],int n)
{
  int i,j,L,R;
  int pom;

  L=1;
  R=n-1;
  j=n-1;
  do{
    for (i=R;i>L-1;i--)
    if (p[i-1]>p[i])
    {
      pom=p[i];
      p[i]=p[i-1];
      p[i-1]=pom;
      j=i;
    };
    L=j+1;
    for (i=L;i<R+1;i++)
    if (p[i-1]>p[i])
    {
      pom=p[i];
      p[i]=p[i-1];
      p[i-1]=pom;
      j=i;
    };
```

```
    R=j-1;
  }while (L<R);
}
```

Příklad



Vytvořte funkci pro třídění pole pomocí algoritmu QuickSort.

```
void quick_pom(int p[],int LL,int RR)
{  //pomocná funkce pro Quick Sort

  int L,R;
  int median,pom;

  median=p[(LL+RR) / 2];
  L=LL;
  R=RR;
  do{
    while (p[L]<median) L++;
    while (median<p[R]) R--;
    if (L<=R)
    {
      pom=p[L];
      p[L]=p[R];
      p[R]=pom;
      L++;
      R--;
    }
  }while (L<=R);
  if (LL<R) quick_pom(p,LL,R);
  if (L<RR) quick_pom(p,L,RR);
}

void QuickSort(int p[],int n)
{
  quick_pom(p,0,n-1);
}
```

Příklad



Vytvořte funkci pro třídění pole pomocí algoritmu HeapSort.

```
void heap_pom(int *Q,int *R,int *pom,int p[])
{ //pomocná funkce pro Heap Sort

  int i,j;

  i=*Q;
  j=2*i;
  if (i==0) j=1;
  *pom=p[i];
  while (j<=*R)
  {
    if (j<*R)
      if (p[j]<p[j+1]) j++;
    if (*pom>p[j]) break;
```

```
    p[i]=p[j];
    i=j;
    j=2*i;
  }
  p[i]=*pom;
}

void HeapSort(int p[], int n)
{
  int Q,R;
  int pom;

  Q=((n-1) / 2) +1;
  R=n-1;
  while (Q>0)
  {
    Q--;
    heap_pom(&Q,&R,&pom,pole);
  }
  while (R>0)
  {
    pom=p[0];
    p[0]=p[R];
    p[R]=pom;
    R--;
    heap_pom(&Q,&R,&pom,pole);
  }
}
```

Příklad

Vytvořte funkci pro třídění pole pomocí algoritmu MergeSort.



```
void MergeSort(int p[],int n)
{
  int i,j; //indexy prvků ve zdrojových polích
  int k,l; //indexy prvku v cílových polích
  int pocet;     //počet prvků p-tice
  int q,r; //pocty prvku, které zbývá sloučit v p-tici
  int m;   //kolik prvku zbývá celkem
  int h;   //směr ukládání v cílovém poli
  int pom;
  typedef enum { FALSE=0,TRUE=1}boolean;
  boolean nahoru;
  int *pom_pole; //pomocné pole

  if ((pom_pole=(int *) malloc(sizeof(int) * n*2)) == NULL)
  {
    printf("Chyba pri alokaci pole.");
    getch();
    exit(0);
  }

  for (i=0;i<n;i++) pom_pole[i]=p[i];
//kopírování pole do pomocného pole
  nahoru=TRUE;
  pocet=1;
```

```
do
{
  h=1;
  m=n;
  if (nahoru == TRUE)
  {
    i=0;j=n-1; //počáteční hodnoty indexu ve zdrojovém poli
    k=n;l=2*n-1;//počáteční hodnoty indexu v cílovém poli
  }
  else
  {
    k=0;l=n-1;//počáteční hodnoty indexu ve zdrojovém poli
    i=n;j=2*n-1;//počáteční hodnoty indexu v cílovém poli
  }
  do
  {
    if (m>=pocet) q=pocet;
   else q=m;
    m -=q;
    if (m>=pocet) r=pocet;
   else r=m;
    m -=r;
    //merge_pom
    while ((q != 0) && (r != 0))
    {
     if (pom_pole[i]<pom_pole[j])
     {
    pom_pole[k]=pom_pole[i];
    k += h;
    i++;
    q--;
     }
     else
     {
       pom_pole[k]=pom_pole[j];
       k += h;
       j--;
       r--;
     }
    }
    while (r != 0)
    {
   pom_pole[k]=pom_pole[j];
   k += h;
   j--;
   r--;
    }
    while (q != 0)
    {
      pom_pole[k]=pom_pole[i];
      k += h;
      i++;
      q--;
    }
    //merge_pom
    //změna směru ukládání v cílovém poli
```

```
      h=-h;
      pom=k;k=l;l=pom;
    }while (m != 0);
    nahoru= !nahoru;
    pocet *= 2;
  }while (pocet<n);
  //kopírování z pomocného pole
  if (nahoru == FALSE) for (i=0;i<n;i++) p[i]=pom_pole[i+n];
  else for (i=0;i<n;i++) p[i]=pom_pole[i];
}
```

Podívejte se na graf závislosti rychlosti třídění pole jednotlivými algoritmy na počtu prvku v poli. Měření bylo prováděno na starším počítači, aby rozdíly byly ještě více patrné.

15. DYNAMICKÉ DATOVÉ STRUKTURY

V této kapitole se dozvíte:

- Jak vytvářet různé druhy třídících funkcí v programovacím jazyce C.

Budete:

- vědět, jak vytvořit funkce pro dynamický zásobník a frontu
- umět využít dynamické datové struktury v praktických příkladech

Klíčová slova této kapitoly:
binární strom, fronta, FIFO, LIFO, spojový seznam, zásobník,



Čas potřebný k prostudování učiva kapitoly:
3 hodiny

Příklad



Vytvořte funkce pro práci s dynamickým zásobníkem. Uvědomte si, že zásobník je datová struktura, u které se nejprve odebírá prvek, který byl zařazen do zásobníku jako poslední.

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

typedef struct prvek{
      int Hod;
      struct prvek *Uk;
      }PRVEK;

void Vytvor(PRVEK **);
void PridejNaVrchol(PRVEK **,int );
void OdeberZVrcholu(PRVEK **,int *);
int Prazdny(PRVEK *);

/*************************************/
int main(int argc, char *argv[])
{
  PRVEK *vrchol;//promenna vrchol je ukazatel na typ prvek
  int i;

  Vytvor(&vrchol);
  for (i=1;i<=10;i++)
   PridejNaVrchol(&vrchol,i);
  while (Prazdny(vrchol)==0)
  {
    int h;
    OdeberZVrcholu(&vrchol,&h);
    printf("%d ",h);
```

```
  }
  system("PAUSE");
  return 0;

}
/************************************/
void Vytvor(PRVEK **v)
{
  *v=NULL;
}

void PridejNaVrchol(PRVEK **v,int H)
{  PRVEK *pom;
   pom=(PRVEK*)malloc(sizeof(PRVEK));
   pom->Hod=H;
   pom->Uk=*v;
   *v=pom;
}

void OdeberZVrcholu(PRVEK **v,int *H)
{  PRVEK *pom;
   pom=*v;
   *H=pom->Hod;//  *H=(*v)->Hod;
   *v=(*v)->Uk;    // *v=pom->Uk;
   free(pom);
}

int Prazdny(PRVEK *v)
{
 if (v==NULL) return 1;
    else return 0;
}
```



Příklad

Vytvořte funkce pro práci s dynamickou frontou. Uvědomte si, že fronta je datová struktura, u které se nejprve odebírá prvek, který byl zařazen do fronty jako první.

```
/*
 *  Řešení dynamické fronty.
 */

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

typedef struct element{
      int val;
      struct element *next;
      }ELEMENT;

typedef struct fifo{
      struct element *front;
      struct element *empty;
      }FIFO;

//deklarace funkcí pro práci s frontou
```

```
void VytvorFrontu (FIFO *q);
void ZrusitFrontu(FIFO *q);
void PridejNaKonec(FIFO *q, int x);
void OdebratZCela(FIFO * q, int *x);
int JePrazdna(FIFO *q);

int main(int argc, char *argv[])
{
  FIFO f1;
  int i;

  VytvorFrontu(&f1); //vytvoření fronty

  //přidání deseti prvků do fronty
  for(i=1;i<=10;i++)
    PridejNaKonec(&f1,i);

  printf("\nVypis\n");
  //odebírání a výpis všech prvků fronty
  while (JePrazdna(&f1)==0)
       {
         int hod;
         OdebratZCela(&f1,&hod);
         printf("%d ",hod);
       }

  system("PAUSE");
  return 0;
}

void VytvorFrontu (FIFO *q)
{
  if ((q->front = (ELEMENT *)
 malloc(sizeof(ELEMENT))) == NULL)
  {
    printf("Chyba pri alokaci!\n");
    exit(1);
  };
  q->empty = q->front;
}

void ZrusitFrontu(FIFO *q)
{
    q->front=q->empty=NULL;
}

int JePrazdna(FIFO *q)
{
    if (q->front==q->empty) return 1;
     else return 0;
}

void PridejNaKonec(FIFO *q, int x)
{
  q->empty->val = x;
  if ((q->empty->next = (ELEMENT *)
```

```
    malloc(sizeof(ELEMENT))) == NULL)
  { printf("Chyba pri alokaci!\n");
    exit(1);
  };
  q->empty = q->empty->next;
}

void OdebratZCela(FIFO * q,int *x)
{
  ELEMENT *pom;

  if (q->front == q->empty) printf("Chyba!\n");
   else
   {
     *x = q->front->val;
     pom = q->front;
     q->front = q->front->next;
     free(pom);
   }
}
```



Příklad - Josephus a jeho bojovníci

Podle legendy, za panování císaře Vespasiana, v době bojů Římanů a Židů, ukryl se do jeskyně Josephus (asi 37-100) se čtyřiceti svými druhy. Když viděli, že se nemohou zachránit, rozhodli se navzájem pobit. Všech 41 se postavilo do rady a každý třetí byl zabit (tj. čísla 3, 6, 9, ..., 39). Když se došlo na konec řady, pokračovalo se v počítání znovu od začátku (1, 5, 10, ...). Josephus nechtěl zemřít, proto se postavil tak, aby zůstal naživu.

Program můžeme vyřešit například pomocí fronty. Prvek, který nemá být vyřazen, odebereme z čela fronty a zařadíme jej na její konec. Prvek, který podle zadaného kroku má být vyřazen, z fronty vyřadíme. Tento postup budeme opakovat až do chvíle, kdy fronta bude prázdná.

```
/*
 *  Řešení Josephova problému pomocí fronty.
 */

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

typedef struct element{
      int val;
      struct element *next;
      }ELEMENT;

typedef struct fifo{
      struct element *front;
      struct element *empty;
      }FIFO;

//deklarace funkcí pro práci s frontou
```

```
void VytvorFrontu (FIFO *q);
void ZrusitFrontu(FIFO *q);
void PridejNaKonec(FIFO *q, int x);
void OdebratZCela(FIFO *q, int *x);
int JePrazdna(FIFO *q);

int main(int argc, char *argv[])
{
  FIFO f1;
  int i;
  int pocet, posun=1, krok, kdo, pom;

  VytvorFrontu(&f1); //vytvoření fronty

  printf("Zadej pocet vojaku : ");
  scanf("%d",&pocet);

  //vytvoření fronty vojáků
  for(i=1;i<=pocet;i++)
    PridejNaKonec(&f1,i);

  printf("Zadej krok vyrazovani vojaku: ");
  scanf("%d",&krok);


  while (JePrazdna(&f1)==0)
       {
         OdebratZCela(&f1,&kdo);
       if (posun != krok)
         {
           PridejNaKonec(&f1,kdo);
           posun++;
         }
         else
       {
            printf("%d ",kdo);
            posun = 1;
       }
       }

  system("PAUSE");
  return 0;
}

void VytvorFrontu (FIFO *q)
{
  if ((q->front = (ELEMENT *)
 malloc(sizeof(ELEMENT))) == NULL)
  {
    printf("Chyba pri alokaci!\n");
    exit(1);
  };
  q->empty = q->front;
}

void ZrusitFrontu(FIFO *q)
```

```
{
    q->front=q->empty=NULL;
}

int JePrazdna(FIFO *q)
{
    if (q->front==q->empty) return 1;
     else return 0;
}

void PridejNaKonec(FIFO *q, int x)
{
  q->empty->val = x;
  if ((q->empty->next = (ELEMENT *)
    malloc(sizeof(ELEMENT))) == NULL)
  { printf("Chyba pri alokaci!\n");
    exit(1);
  };
  q->empty = q->empty->next;
}

void OdebratZCela(FIFO *q,int *x)
{
  ELEMENT *pom;

  if (q->front == q->empty) printf("Chyba!\n");
   else
   {
     *x = q->front->val;
     pom = q->front;
     q->front = q->front->next;
     free(pom);
   }
}
```



Příklad

Příklad Josephova problému vyřešte pomocí kruhového spojového seznamu.

16. DALŠÍ PŘÍKLADY K PROCVIČENÍ

V této kapitole si můžete procvičit své znalosti a dovednosti na dalších řešených příkladech.

Čas potřebný k prostudování učiva kapitoly:
3 hodiny





Příklad – práce se soubory:

```
//**************************************
//  main.c
//  soubory
#include <stdio.h>
#include <stdlib.h>
#include "student.h"
#define N 3

int main(int argc, const char * argv[])
{
    Student s[N];
    int i;
    FILE *f;

    for(i=0;i<N;i++)
    {
        nactiStudenta(&s[i]);
    }

    for(i=0;i<N;i++)
    {
        vypisStudenta(&s[i]);
    }

    f = fopen("data.txt","w");
    if (f == NULL)
    {
        printf("Chyba souboru!");
        return 0;
    }

    for(i=0;i<N;i++)
    {
        fprintf(f,"%s\n",s[i].jmeno);
        fprintf(f,"%d\n",s[i].kredity);
    }
    fclose(f);
    //...

    f = fopen("data.txt","r");
    if (f == NULL)
    {
        printf("Chyba souboru!");
        return 0;
```

```
    }
    for(i=0;i<N;i++)
    {
        char pom[50];
        fscanf(f,"%s", pom);
        ziskejData("jmeno",pom,s[i].jmeno);

        fscanf(f,"%d", &s[i].kredity);
    }
    fclose(f);
    for(i=0;i<N;i++)
    {
        vypisStudenta(&s[i]);
    }


    return 0;
}

//**************************************
//  myxml.h
//  soubory

#ifndef soubory_myxml_h
#define soubory_myxml_h

int ziskejData(char *nod, char *radek,char *vys);


#endif

//**************************************
//  myxml.c
//  soubory

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

int ziskejData(char *nod, char *radek,char *vys)
{
    long delka;
    char pom[20];
    char text[50];

    pom[0]='\0';
    text[0]='\0';

    strcat(pom,"<");
    strcat(pom,nod);
    strcat(pom,">");

    if (strstr(radek,pom) != NULL)
    {
        char *konec;
        delka = strlen(pom);
        strcat(text, &radek[delka]);
```

```
        pom[0]='\0';
        strcat(pom,"</");
        strcat(pom,nod);
        strcat(pom,">");

        konec = strstr(text,pom);
        *konec = '\0';
        strcpy(vys,text);

        return 1;
    }
    else
    {
        return 0;
    }
}

//**************************************
//  student.c
//  soubory

#include <stdio.h>
#include "student.h"

void vypisStudenta(Student *s)
{
    printf("Jmeno %s, ",s->jmeno);
    printf("pocet kreditu %d.\n",s->kredity);
}

void nactiStudenta(Student *s)
{
    printf("Zadej jmeno ");
    scanf("%s",s->jmeno);
    printf("Zadej pocet kreditu ");
    scanf("%d",&s->kredity);
}

//**************************************
//  student.h
//  soubory

#ifndef soubory_student_h
#define soubory_student_h

typedef struct
{
    char jmeno[30];
    int kredity;
}Student;

void nactiStudenta(Student *s);
void vypisStudenta(Student *s);

#endif
```



Příklad – vytvořte funkci, která načte text ukončený tečkou a vypočte počet malých a velkých písmen, počet číslic a mezer.

```
//  soubor main.c
#include <stdio.h>
#include "veta.h"

int main(int argc, const char * argv[])
{
    int celkem, pomMalych=0, pomVelkych, pomCislic, pomMezer;

    celkem =
poctyVeVete(&pomMalych,&pomVelkych,&pomCislic,&pomMezer);

    pomMalych =0;
    celkem =
poctyVeVete(&pomMalych,&pomVelkych,&pomCislic,&pomMezer);

    printf("Celkovy pocet znaku je %d\n", celkem);
    return 0;
}


//  soubor veta.h
#ifndef cv03b_veta_h
#define cv03b_veta_h

int poctyVeVete(int *pm, int *pv, int *pc, int *pz);

#endif


// soubor veta.c
#include <stdio.h>
#include "veta.h"


int poctyVeVete(int *pm, int *pv, int *pc, int *pz)
{
    char znak;
    int pocet = 0;
    *pm = *pv = *pc = *pz = 0;

    do {
        //scanf(" %c",&znak);
        znak = getchar();
        pocet++;
        if ((znak>='a') && (znak<='z')) (*pm)++; //*pm = *pm +
1;
        if ((znak>='A') && (znak<='Z')) (*pv)++; //*pv += 1;
        if ((znak>='0') && (znak<='9')) (*pc)++;
        if (znak==' ') (*pz)++;
    } while (znak != '.');
    return pocet;
}
```

Příklad – práce s řetězci.

```
//  main.c

#include <stdio.h>
#include "mystring.h"

#define MAX 1000

int main (int argc, const char * argv[])
{
/*
    char str[MAX];
    int pSlova, pVety;
    long pZnaky;

    printf("Zadej text\n");
    gets(str);

    pZnaky = analyzujText(str, &pSlova, &pVety);

    printf("Pocet vsech znaku je %ld\n", pZnaky);
    printf("Pocet vsech slov je %d\n", pSlova);
    printf("Pocet vsech vet je %d\n", pVety);

*/

    char str[MAX], co[MAX], cim[MAX];
    int pozice;

    strcpy(str , "XXXfffghfg");
    strcpy(co , "XXX");
    strcpy(cim,"QQQQQQQ");
    printf("Neupraveny text je %s\n", str);

    pozice = nahraditText(str, co, cim);
    printf("Upraveny text je %s\n", str);
    printf("Pozice je %d\n", pozice);

    return 0;
}

//  mystring.h
#ifndef retezce01_mystring_h
#define retezce01_mystring_h

long analyzujText(char *str,int *pocetSlov, int * pocetVet);

int nahraditText(char *str, char *co, char *cim);

#endif

//  mystring.c
#include <string.h>
#include <stdlib.h>
#include "mystring.h"
```

```
long analyzujText(char *str,int *pocetSlov, int * pocetVet)
{
    long delka = strlen(str);
    int i;
    *pocetSlov = 0;
    *pocetVet = 0;
    for(i=0; i<delka; i++)
    {
        if ((str[i] == ' ' &&str[i-1] != '.')|| str[i] == '.')
*pocetSlov +=  1;
        if (str[i] == '.' || str[i] == '!' || str[i] == '?')
(*pocetVet)++;
    }
    return delka;
}

int nahraditText(char *str, char *co, char *cim)
{
    char *p;
    char *pom;
    if ((pom=(char*)malloc(sizeof(char)
             *(strlen(str)-strlen(co)) +1 )) == NULL)
    {
        return -2;
    }
    p = strstr(str,co);

    if (p == NULL) return -1;
    else
    {
        strcpy(pom, (p + strlen(co)));
        *p = '\0';
        strcat(str, cim);
        strcat(str,pom);
    }
    return (p - str);
}
```



Příklad – práce s polem.

```
//  main.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pole.h"

#define MAX 10

int main(int argc, const char * argv[])
{
    int pole1[MAX];
    int i;
    int pocet = 10;
    time_t t;
```

```
    int *p;
    srand((unsigned) time(&t));
      //nastavi generator nahodnych cisel

    if ((p=(int*)malloc(sizeof(int)*pocet)) == NULL)
    {
        printf("Chyba!");
        return 0;
    }
    for(i=0;i<pocet;i++)
        p[i] = rand() % 100;
        //*(p+i) = rand() % 100;
    printf("Dynamicke pole\n");
    vypisPole(p,pocet);
    setridPole(p,pocet);
    printf("\nSetrizene pole: ");
    vypisPole(p, pocet);

    free(p);
    p=NULL;

    printf("\nStaticke pole\n");
    for(i=0;i<pocet;i++)
        pole1[i] = rand() % 100;

    vypisPole(pole1, pocet);
    printf("minimum je %d\n",minimumPole(pole1, pocet));

    setridPole(pole1,pocet);
    printf("Setrizene pole: ");
    vypisPole(pole1, pocet);

    return 0;
}

//  pole.h
#ifndef pole01b_pole_h
#define pole01b_pole_h

void vypisPole(int p[], int pocet);
void setridPole(int p[], int pocet);
int minimumPole(int p[], int pocet);

#endif

//  pole.c
#include <stdio.h>
#include "pole.h"

void vypisPole(int p[], int pocet)
{
    int i;

    for(i=0;i<pocet;i++)
        printf("%d ",p[i]);
```

```
}

void setridPole(int p[], int pocet)
{
    int i, j;

    for (i=0;i<pocet-1;i++)
        for(j=0;j<pocet- 1 - i;j++)
            if (p[j]>p[j+1])
            {
                int pom = p[j];
                p[j] = p[j+1];
                p[j+1] = pom;
            }

}

int minimumPole(int p[], int pocet)
{
    int i, min = p[0];

    for(i=1;i<pocet;i++)
        if (min>p[i]) min = p[i];

    return min;
}
```



Příklad – práce se soubory pomocí handle

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

#define POCET 80
int main(int argc, char *argv[])
{
  int h;
  int i,cis;
  //zapis do souboru
  h = open("soubor07.txt",O_CREAT | O_RDWR);
  for(i=65;i<=POCET;i++)
    write(h,&i,sizeof(int));

  //cteni v souboru
  lseek(h,-tell(h), SEEK_CUR);
  for(i=65;i<=POCET;i++)
  {
     read(h,&cis,sizeof(cis));
     printf("%d ",cis);
  }

  close(h);
  system("PAUSE");
  return 0;
}
```



Příklad – práce s řetězci

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

void obrat(char s[]);

int main(int argc, char *argv[])
{
  char str[] ="nejakytext";
  obrat(str);
  printf("%s",str);

  system("PAUSE");
  return 0;
}

void obrat(char s[])
{
     int i;
     int pocet = strlen(s);
     for(i=0;i<pocet / 2;i++)
     {
        char pom;
        pom = s[i];
        s[i] = s[pocet-1-i];
        s[pocet-1-i] = pom;
     }
}
```



LITERATURA

BURKHAND, M. *C pro mikrokontrolery,* BEN, Praha 2003, ISBN 80-7300-077-6

HANÁK, J. *Programování v jazyce C*, Computer Media, Praha 2011, ISBN 978-80-7402-041-4

HEROUT, P. *Učebnice jazyka C 1. díl*, KOPP, 2009, ISBN 978-80-7232-383-8

HEROUT, P. *Učebnice jazyka C 2. díl*, KOPP, 2008, ISBN 978-80-7232-367-8

MATOUŠEK, D. *Programování v jazyce C v příkladech,* BEN, Praha 2011, ISBN 978-80-7300-4

PROKOP, J. *Algoritmy v jazyku C a C*++, Grada, Praha 2008, ISBN 978-80-247-2751-6