První začátky s C

Struktura programu a základní prvky

Jazyk C/C++ , co se týče struktury souboru, je daleko volnější oproti jiným programovacím jazykům. V podstatě je jedno, kde deklarujete proměnnou nebo nový typ. V souboru se rozlišují pouze určité bloky a platnost proměnné začíná od místa deklarace a končí koncem bloku, v němž je deklarovaná.

```
{ // Začátek nějakého bloku
  příkaz1;
  příkaz2;
  ...
  příkazn;
} // Konec nějakého bloku
```

Každý příkaz je ukončen středníkem. Na jednom řádku může být za sebou i několik příkazů, všechny však musí být odděleny středníkem (příkaz1; příkaz2; příkaz3).

Základní termíny (1)

- **Klíčová slova**
  jsou jednoduše řečeno příkazy jazyka, které jsou přesně dány, nedají se předefinovat.
  Např. `for, while, if` atd…
- **Konstanty**
  dělí se na numerické, znakové a textové (literály), nenumerické, ... Je to obdoba proměnných, které mají ovšem po nadefinování neměnnou hodnotu.
- **Operátory**
  zajišťují provedení určité akce nad určitými operandy. Příkladem operátoru je operátor pro sečtení dvou operandů (+).
- **Identifikátory**
  jsou to označení patřící různým proměnným, typům, funkcím, objektům atd… daného programu. Musí začínat písmenem a další znaky mohou být čísla nebo písmena.

Základní termíny (2)

- **Návěští**
  určitý bod v programu, na který je "odskakováno" po zavolání příkazu skoku (`goto`). Jsou zde zařazena především z historického hlediska. V klasických metodách strukturovaného programovaní se používají velice zřídka.
- **Komentář**
  text, který můžeme vložit do programu. Nemá žádný vliv na běh programu. Je určen k zlepšení čitelnosti zdrojového textu a tak k rychlejší orientaci. Komentář je uvozen buď znaky // (dvě lomítka) nebo /* (lomítko a hned za ním hvězdička). Komentář uvozený // má platnost od uvedení lomítek až do konce řádku, komentář /* má platnost od tohoto začátku až do místa, kde jsou znaky */ (hvězdička a lomítko), které identifikují konec komentáře. Textu v komentářích si překladač vůbec nevšímá. A před vlastním spuštěním programu provede jeho odstranění (preprocesor).

Jednoduché datové typy a přiřazení (1)

C poskytuje podobné datové typy jako Pascal:

Pascal	**C**
`INTEGER`	**`int`**
	`long int` *též* **`long`**
	`short int` *též* **`short`**
`CHAR`	**`char`**
`REAL`	**`float`**
	`double`
	`long double`

Jednoduché datové typy a přiřazení (2)

- Typy **int**, **long int**, **short int** a **char** mohou být
  - **signed,** implicitní pro **int**, **long int**, **short int**, (pro **char** záleží na implementaci)
  - **unsigned,** (**unsigned int** se často zkracuje jen na **unsigned**)
  - Rozsah **signed** (znaménkový) a **unsigned** (neznaménkový) čísel:
    - Proměnné typu **unsigned** mají rozsah od 0 do 2^{n-1}, kde *n* je počet bitů proměnné. (**Unsigned** typ nemůže zobrazit záporné číslo)
    - rozsah **signed** proměnných je od -2^{n-1} do $+2^{n-1} - 1$ (poloviční rozsah typu **unsigned**).
      - Příklad pro typ **char**, který je vždy 1 Byte (8 bitů) dlouhý:
        - » **unsigned char** 0 až 255
        - » **signed char** -128 až +127
- C neposkytuje přímo typ Boolean. Booleovské hodnoty jsou reprezentovány pomocí celočíselných (**int**) hodnot, kde:
  - nulová hodnota (0) *FALSE*
  - nenulová hodnota (nejčastěji 1) *TRUE*
- Typ **double** má přesnost asi na 20 desetinných míst

Definice proměnných

Definice je příkaz, který přidělí proměnné určitého typu jméno a paměť

Deklarace je příkaz, který pouze udává typ proměnné a její jméno (nepřiděluje paměť!)

V C jsou definice v obráceném pořadí než v Pascalu:

Pascal	**C**
`Var i : INTEGER;`	`int i;`
`c,ch : CHAR;`	`char c,ch;`
`f,g : REAL;`	`float f,g;`

Pozn:

Definice proměnné vně funkce (globální proměnná) nebo uvnitř funkce (lokální proměnná)

```
int i;                /* globální proměnná */
int main()
{
  int j;              /* lokální proměnná */
}
```


Identifikátory

Jazyk C je *case sensitive* jazyk (rozlišuje malá a velká písmena).

`prom Prom PROM` jsou tři různé identifikátory!

Klíčová slova C (např. **`if`**, **`while`**, **`register`**, …) musí být psána malými písmeny. Jsou-li zapsána velkými nebo kombinací malých a velkých písmen, berou se jako identifikátory.

C dovoluje u identifikátorů používat znak podtržítko "_":

`_prom`	nepoužívat, znamená to systémový identifikátor
`prom_`	používat často, zpřehledňuje text
`prom_`	nepoužívat, na konci se často přehlédne

Délka identifikátoru není omezena, ale ANSI C rozeznává obecně pouze prvních 31 znaků (32. další jsou bezvýznamné)

Přiřazení

Často se pracuje s pojmem *l-hodnota (l-value)*. L-hodnota představuje adresu, tedy např. proměnná (x) je l-hodnotou, ale konstanta (123) nebo výraz (x+3) l-hodnotou nejsou.
L-hodnota je to, co může být na levé straně přiřazení.

česky	*anglicky*	*symbolicky*	*prakticky*
výraz	*expression*	*výraz*	`i*2+3`
přiřazení	assignment	*l-hodnota=výraz*	`j=i*2+3`
příkaz	statement	*l-hodnota=výraz;*	`j=i*2+3;`

Příklad - různé typy přiřazovacích příkazů:

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

Protože přiřazení je výraz, je možné několikanásobné přiřazení:
`k = j = i = 2;`
které se vyhodnocuje zprava doleva tedy: `k = (j = (i = 2));`

Hlavní program (1)

Hlavní program v C se jmenuje vždy `main` a musí být v programu uveden - je to první funkce volaná po spuštění programu.

Pascal
```
PROGRAM POKUS(INPUT, OUTPUT);
```
C
```
int main() bez středníku!
```

Funkce `main` je normální funkce v C, odlišuje se od ostatních pouze tím, že je vyvolána na začátku programu jako první

Pascal
```
PROGRAM POKUS(INPUT, OUTPUT);
VAR
 i, j : INTEGER;
BEGIN
 i := 5;
 j := -1;
 j := j + 2 * i;
END.
```
C
```
int main() /* bez středníku! */
{
  int i, j;

  i = 5;
  j = -1;
  j = j + 2 * i;
}
```


Hlavní program (2)

- Pascalovské **BEGIN** a **END** je v C nahrazeno znaky "{" a "}"
- Závorky "{ }" neuzavírají pouze složený příkaz, ale i blok
- Bezprostředně za každou "{" mohou být definice

Blok

seznam definicí následovaný seznamem příkazů
```
{
   int i;
   i = 5;
   j = 6;
}
```

Složený příkaz

pouze seznam příkazů
```
{
   i = 5;
   j = 6;
}
```

- Na rozdíl od Pascalu, umožňuje C inicializaci proměnných přímo v definici, takže předchozí příklad bude také správně když:

```
int main()
{
   int i = 5,
       j = 6;
   j = j + 2 * i;
}
```


Konstanty - celočíselné

- **desítkové** (decimální)
  - posloupnost číslic, z nichž první nesmí být nula
    - Příklad: `15, 0, 1`
- **osmičkové** (oktalové)
  - číslice 0 následovaná posloupností osmičkových číslic (0 - 7)
    - Příklad: `065, 015, 0, 01`
- **šestnáctkové** (hexadecimální)
  - číslice 0 následovaná znakem `x` (nebo `X`) a posloupností hexadecimálních číslic (`0 - 9`, `a - f`, `A - F`)
    - Příklad: `0x12, 0X3A, 0XCD, 0xCD, 0Xcd, 0x15, 0x0, 0x1`

Typ konstanty je určen implicitně její velikostí nebo explicitně použitím přípony:

Příklad: 12345678L - konstanta typu **long**
12345LU - konstanta typu **long unsigned**

Konstanty - reálné

- Tvoří se podle běžných zvyklostí
- Mohou začínat a končit desetinnou tečkou (ne čárkou!)
- Implicitně jsou typu **double**
  - Příklad: 15. 56.8 .84 3.14 5e6 7E23
- Reálná konstanta typu **float** se definuje pomocí přípony F (nebo f)
  - Příklad: 3.14f 3.14F
- Reálná konstanta typu **long double** pomocí L (nebo l)
  - Příklad: 12e3L 12E3L 12e3l

  (*malé l raději nepoužívat, může se lehce zaměnit za 1)*

Konstanty - znakové

- Stejně jako v Pascalu se uzavírají mezi apostrofy
  - Příklad: `'a', '*', '4'`
- Hodnota znakových konstant (ordinální číslo) je odvozena z odpovídající kódové tabulky – nejčastěji ASCII.
- Velikost znakové konstanty je **int** a ne **char** !
- Znaková konstanta z neviditelného znaku – `'\ddd'`, kde `ddd` je kód znaku složený ze tří oktalových číslic
  - Příklad: `'\012', '\007'`

Konstanty – řetězcové (literály)

Uzavírají se mezi uvozovky (narozdíl od Pascalu)

Příklad: `"Toto je retezcova konstanta"`

Aritmetické výrazy

Příkazem se stává výraz ukončený středníkem

Příklad:	`i = 2`	*výraz s přiřazením*
	`i = 2;`	*příkaz*

Samotný středník se používá pro prázdný příkaz (*null statement*), který se používá např. v cyklu **while** nebo **for**

Aritmetické výrazy – unární operátory

Unární –

Unární +

Oba operátory se používají v běžném významu

Aritmetické výrazy – binární operátory (1)

Z větší části mají stejný význam jako v Pascalu:

	Pascal	**C**
– *Sčítání*	`+`	`+`
– *Odčítání*	`-`	`-`
– *Násobení*	`*`	`*`
– *Reálné dělení*	`/`	`/`
– *Celočíselné dělení*	`DIV`	`/`
– *Dělení modulo*	`MOD`	`%`

Zda bude dělení celočíselné nebo reálné, závisí na typu operandů:

int / **int**	- celočíselné
int / **float**	- reálné
float / **int**	- reálné
float / **float**	- reálné

Pozn.: Pro **double** a **long double** platí totéž co pro **float**.

Aritmetické výrazy – binární operátory (2)

Příklad:

```
int i = 5, j = 13;
j = j / 4;          - celočíselné dělení, j bude 3
j = i % 3;          - dělení modulo, j bude 2
```


Aritmetické výrazy – speciální unární operátory (1)

V Pascalu nemají obdobu

inkrement	`++`
dekrement	`--`

Oba operátory se dají použít jako předpony (*prefix*) i jako přípony (*suffix*):

`++vyraz`

- inkrementování před použitím
- výraz je nejprve zvětšen o jedničku a pak je tato nová hodnota vrácena jako hodnota výrazu

`vyraz++`

- inkrementování po použití
- je vrácena původní hodnota výrazu a pak je výraz zvětšen o jedničku

Pozor: Výraz musí být l-hodnota (tedy proměnná).

`45++` nebo `--(i+j)` je chybné!

Aritmetické výrazy – speciální unární operátory (2)

Příklad:

Různá použití operátorů ++ a --

```
int i=5, j=1, k;
i++;                    - i bude 6
j=++i;                  - j  bude 7, i bude 7
j=i++;                  - j bude 7, i bude 8
k=--j+2;                - k bude 8, j bude 6, i bude 8
```


Aritmetické výrazy – přiřazovací operátory (1)

Operátor přiřazení - Pascalu: :=
- C: =

C má navíc celou řadu rozšířených přiřazovacích operátorů.
Místo přiřazení:

l-hodnota = l-hodnota operátor výraz

se velmi často používá zkrácený zápis:

l-hodnota operátor= výraz

Aritmetické výrazy – přiřazovací operátory (2)

Dají se použít následující přiřazení:

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

Pozn.:

Mezi operátorem a rovnítkem se nedává mezera.

ne `j + = 5;` ale `j += 5;`

Příklad:

```
int i = 4, j = 3;
j += i;                 j bude 7
j /= --i;               j bude 2, i bude 3
j *= i -2;              j=j*(i–2)=2
                        ne j=j*i-2=4
```


Komentáře (1)

Přestože jsou velmi často opomíjeny, jsou důležitou součástí programu:

- zpřehledňují, někdy na první pohled dost nepochopitelný, program
- slouží k tomu, aby se ve vašem programu vyznal někdo cizí nebo i vy sami, když se k němu vrátíte třeba za několik let
- doporučuje se komentovat své programy během vytváření, a ne až po odladění („*Až na to někdy zbude čas.*“)

```
/* toto je komentar */
/* viceradkovy
   komentar
*/
x=3*a+b;    /* popis prikazu */
```

Komentáře (2)

Někdy se můžeme setkat s jiným stylem psaní komentářů, který není v normě ANSI C a nazývá se **jednořádkový komentář**.

- Začíná dvojicí // a končí koncem řádku
- Byl zaveden pro C++. Jeho použití v programech C je sice technicky nesprávné, ale většina překladačů jej bude akceptovat.

```
x=3*a+b;    // popis prikazu
```

je totéž jako

```
x=3*a+b;    /* popis prikazu */
```


Terminálový vstup a výstup

V C se (narozdíl od Pascalu) nedefinuje žádná I/O (vstupně/výstupní – *Input/Output*) operace jako část jazyka.

Nezbytné vstupy a výstupy jsou řešeny tak, že standardní knihovna obsahuje několik funkcí, které I/O zajišťují.

Důvod:

Nejvíce strojově závislé akce jsou právě I/O a tímto se tedy důsledně oddělují strojově závislé a strojově nezávislé části jazyka.

Tato skutečnost je pak významným přínosem při vytváření kompilátoru pro jiný počítač.

Hlavičkový soubor `stdio.h a math.h`

Aby bylo možné správně používat všechny funkce pro vstup a výstup, je nutné na začátku programu připojit "popis" těchto funkcí. Ten se nachází v hlavičkovém (*header*) souboru `stdio.h` a do programu se připojí pomocí příkazu:

```
#include <stdio.h>
```
na konci není středník !!!

Od tohoto okamžiku je možné používat dále popisované funkce

Pro využívání standardních matematických funkcí (sin, cos, sqrt, apod.) existuje hlavičkový soubor `math.h`

```
#include <math.h>
... c=sqrt(a+b); ...
```


Vstup a výstup znaku

`putchar()` výstup jednoho znaku
`getchar()` vstup jednoho znaku

Obě funkce pracují s proměnnými typu **int** a ne **char**

Příklad:

Program přečte znak z klávesnice, vytiskne ho a odřádkuje.

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

  c = getchar();
  putchar(c);
  putchar('\n');
}
```

Formátovaný vstup a výstup (1)

	C	Pascal
Vstup	`scanf()`	READ
Výstup	`printf()`	WRITE

Základní použití:

- Příkaz: `scanf("%d",&i)`
  přečte z klávesnice celé číslo a uloží ho do proměnné `i`
  - `"%d"` určuje formát čtení (zde dekadický celočíslený)
  - `&` před `i` je nezbytně nutný (vynechání je častou chybou)
- Příkaz: `printf("%d",i)`
  vytiskne na obrazovku hodnotu proměnné `i`
  - `"%d"` určuje formát výpisu (zde dekadický celočíslený)
  - před `i` není `&`, což je rozdíl proti `scanf()`
    (je to hodnota a ne adresa)

Formátovaný vstup a výstup (2)

Program přečte z klávesnice dvě čísla, vytiskne je v obráceném pořadí a pak vytiskne jejich součet

```
#include <stdio.h>

int main()
{
  int i, j;
  scanf("%d", &i);
  scanf("%d", &j);
  printf("%d%d", j, i);
  printf("%d je soucet", i + j);
}
```


Formátovaný vstup a výstup (3)

Příklady (i = 4, j = 7):

1. `printf("Soucet je %d", i + j);`
   vypíše: `Soucet je 11`
2. `printf("Pracovali na 100%%");`
   vypíše: `Pracovali na 100%` neboť pro výpis znaku "%" je nutné tento znak zdvojit
3. `printf("Soucet je %d\tSoucin je %d\n", i + j, i * j);`
   vypíše: `Soucet je 11  Soucin je 28` a odřádkuje
4. `printf("\007Chyba, pokus o deleni nulou.\n");`
   pískne a vypíše: `Chyba, pokus o deleni nulou.` a odřádkuje

Vždy je nutné dodržet stejný počet parametrů (proměnných nebo výrazů) jako formátovacích specifikací (kolik je v řídicím řetězci znaků "%", tolik musí být dalších parametrů)

Formátovaný vstup a výstup (4)

Některé formátové specifikace řídicího řetězce formátu uváděné za znakem "%" použitelné jak pro `scanf()` tak i pro `printf()`:

`c`	- znak
`d`	- desítkové číslo typu **signed int**
`ld`	- desítkové číslo typu **signed long**
`u`	- desítkové číslo typu **unsigned int**
`lu`	- desítkové číslo typu **unsigned long**
`f`	- **float** (pro `printf()` také **double**)
`Lf`	- **long double** (*Pozor:* L musí být velké!)
`lf`	- **double** (*Pozor:* někdy nelze použít pro `printf()`)
`x`	- hexadecimální číslo malými písmeny, např. `1a2c`
`X`	- hexadecimální číslo velkými písmeny, např. `1A2C`
`o`	- osmičkové číslo
`s`	- řetězec

Časté chyby

• `main();`	za definicí funkce se nedělá středník
• `printf("%d",i,j);`	mnoho argumentů
• `printf("%d%d",i);`	málo argumentů
• `scanf("%d",i);`	chybí znak & tedy: `scanf("%d",&i);`
• `scanf(0"%d",&c);`	formát pro **char** je `%c scanf("%c",&c);`

Co je dobré si uvědomit

- Všechna klíčová slova musí být malými písmeny.
- Dělení (`/`) je operace závislá na typu operandů – pro celá čísla je to celočíselné dělení, jinak je to reálné dělení.
- Přiřazení je výraz a příkazem se stává až po ukončení středníkem.
- Počet výstupních výrazů v `printf()` nebo vstupních v `scanf()` musí přesně odpovídat počtu formátových specifikací.
- U proměnných ve `scanf()` je `&`.

Řídicí struktury

Booleovské výrazy

V C není implicitně typ **Boolean.** Místo něj se používá typ **int**, kde nulová hodnota (`0`) znamená *FALSE* a nenulová hodnota (nejčastěji `1`, ale není to podmínkou) je *TRUE.*

	Pascal	**C**
rovnost	`=`	`==`
nerovnost	`<>`	`!=`
logický součin	`AND`	`&&`
logický součet	`OR`	`||`
negace	`NOT`	`!`

další čtyři relační operátory mají stejnou syntaxi i význam

menší	`<`
menší nebo rovno	`<=`
větší	`>`
větší nebo rovno	`>=`

Zkrácené vyhodnocování logických výrazů

Zajímavou vlastností jazyka C je, že se logický součin a součet vyhodnocují ve **zkráceném vyhodnocení** (*short circuit*).

To znamená, že argumenty jsou vyhodnocovány zleva doprava a jakmile je možno určit konečný výsledek, vyhodnocování okamžitě končí.

Příklad:

C `if (y != 0 && x / y < z)`
nedojde k dělení nulou

Pascal `if ((y <> 0) AND (x / y < z))`
může dojít k dělení nulou

Priority vyhodnocování výrazů (1)

Tabulka priorit a způsobu vyhodnocování některých operátorů:

Operátor	*Směr vyhodnocení*
`!  ++  --  -  +` (*typ*)	zprava doleva
`*  /  %`	zleva doprava
`+  -`	zleva doprava
`<  <=  >=  <`	zleva doprava
`==  !=`	zleva doprava
`&&`	zleva doprava
`||`	zleva doprava
`?  :`	zleva doprava
`=  +=  -=  *=` atd.	zleva doprava
`,`	zleva doprava

Priority vyhodnocování výrazů (2)

V C mají aritmetické operátory a operátor porovnání vyšší prioritu než logické operátory, takže výraz:

```
if (c >= 'A' && c <= 'Z')
```

je správný, kdežto stejně napsaný pascalovský výraz:

```
if (c >= 'A' AND c <= 'Z')
```

je chybný.

Pozor:

Nezaměňovat && za & nebo || za |. Operátory & a | představují bitové operace, a použity nesprávně v logických výrazech dají nesprávné výsledky.

Blok

Syntaxe bloku je velice jednoduchá. Blok je uzavřen mezi dvěma složenými závorkami. Mezi těmito závorkami mohou být libovolné jiné příkazy včetně dalších bloků.

Kromě toho mohou být ještě na začátku bloku, tedy před prvním příkazem, lokální deklarace a definice. Proměnné (a další objekty) takto definované jsou pak viditelné pouze uvnitř bloku a v dalších vnořených blocích.

Blok je v C chápán jako jediný příkaz a proto se také dá použít všude tam, kde je možné použít příkaz. Blok se také často označuje pojmem **složený příkaz**.

```
{ int i;
  DelejNeco(i);
   {DelejJesteNeco(i);
   }
}
```

Podmíněný příkaz `if`

`if (výraz1) příkaz1`

je základní příkaz sloužící k **větvení** toku programu. Prvním krokem při vykonávání příkazu je vyhodnocení *výrazu1* (závorky kolem výrazu nelze vynechat !). Pokud je hodnota, vzniklá jeho vyhodnocením, nenulová, provede se *příkaz1*. V opačném případě, kdy je *výraz1* vyhodnocen jako nula, se *příkaz1* neprovede.

Příkaz if lze ještě doplnit o nepovinnou část `else`:

`if (výraz1) příkaz1 else příkaz2`

Rozdíl při použití `else` je v tom, že při nesplnění podmínky (*výraz1* je vyhodnocen jako 0) se provede *příkaz2*.

Příklad:

```
if (a>1) a=1; else b=1;
```

V příkazu `a=1;` (stejně tak v `b=1;`) je nutné použít středník, protože právě ten udělá z výrazu `a=1` příkaz.

V C znak středníku neplní oddělovací funkci, jak je tomu například v Pascalu, ale je přímo součástí příkazů.

Cyklus `while`

Příkaz `while` realizuje v C cykly s podmínkou na začátku.

`while (výraz1) příkaz1`

funguje tak, že se nejprve vyhodnotí *výraz1* a pak, je-li vyhodnocen jako nenulový, provede se *příkaz1*. Po jeho provedení se znovu vyhodnotí *výraz1* a případně se znovu provede *příkaz1*. Cyklus končí v okamžiku, kdy je podmínka vyhodnocena jako 0. *Příkaz1* se pak již neprovede a program pokračuje dalším příkazem po `while`.

```
int a=0;
while (a<10) a++; //desetkrát se provede inkrementace a.
```


Cyklus `do-while`

Podobný cyklu `while`, ale vyhodnocuje pokračovací podmínku (*výraz1*) až po provedení těla cyklu. To znamená, že minimálně jednou se tělo cyklu provede vždy.

`do příkaz1 while (výraz1)`

Nejprve se provede *příkaz1* a teprve po jeho provedení je vyhodnocen výraz podmínky. Pokud je nenulový, znovu se vykoná *příkaz1*. To pokračuje až do doby, kdy je podmínka vyhodnocena jako nulová.

```
a=0;
do { printf("ahoj\n");
}while (a++ != 10)
```

Na předchozím příkladu je vidět, že poslední řádek si lze, u složitějších programů, snadno splést se zápisem obyčejného cyklu `while`. Proto se ukončovací závorka bloku často píše před slovo `while`. Tak programátor naznačí sám sobě, ale i dalším lidem, kteří budou se zdrojákem pracovat, že nejde o cyklus `while`, ale `do-while`.

Cyklus `for` (1)

Cyklus for je vlastně jen trochu rozšířený příkaz `while`, pomocí kterého můžeme přehledněji zapisovat iterační cykly. Už na první pohled vypadá příkaz `for` odlišně od stejných příkazů v jiných jazycích. I jeho použití může být značně odlišné od toho, na jaké jsme zvyklí třeba z Pascalu. Nicméně hlavní způsob jeho uplatnění budou, stejně jako v jiných jazycích, právě iterační cykly.

```
for (inicializační_výraz; terminální_výraz; iterační_výraz)
příkaz1
```

příkaz for je prováděn v těchto krocích:

1. Je vyhodnocen *inicializační výraz*.
2. Je vyhodnocen *terminální výraz*. Pokud je vyhodnocen jako 0, je vykonávání cyklu for ukončeno a pokračuje se prvním příkazem uvedeným po cyklu. Pokud je ale vyhodnocen jako nenulový, pokračuje se krokem 3.
3. Je proveden *příkaz1*.
4. Vyhodnotí se *iterační výraz*. Pokračuje se znovu od bodu 2.

Cyklus for (2)

Typické použití cyklu for:

```
int i;
for(i=0; i<5; i++) printf("%d",i);
// 5-krát se provede příkaz printf, který
tiskne hodnotu proměnné i.
```

Jako inicializační a iterační výrazy mohou být použity výrazy jakéhokoliv typu. Terminální výraz by však měl být číselný.

Je také možné vynechat libovolný z těchto výrazů. Například vynecháním inicializačního a iteračního výrazu vlastně získáme jinak zapsaný cyklus `while`.

Vynecháním terminálního výrazu pak docílíme nekonečného cyklu, protože chybějící terminální výraz bude nahrazen nějakou nenulovou konstantou a ta, samozřejmě, nemůže být nikdy vyhodnocena jako 0.

Cyklus for (3)

Následující příklad demonstruje použití cyklu for pro neiterační cyklus.

```
for (;(a=getch())!=27; putch(a));
```

Inicializační výraz je vynechán, ale znak středníku je povinný a tudíž jej vynechat nelze. V terminálním výrazu je do proměnné a načítán znak z klávesnice a načtená hodnota je hned použita při porovnání s konstantou 27 (klávesa Escape). Cyklus je tedy ukončen stiskem klávesy <Esc>.

Tělem cyklu je prázdný příkaz ;.

Tisk znaku zajišťuje až vyhodnocení iteračního výrazu.

Příkaz skoku goto

Příkaz `goto` rozhodně nepatří mezi hojně užívané příkazy, tedy alespoň ne mezi programátory, kteří se drží zásad strukturovaného programování.

Použití `goto` se lze vždy vyhnout, a proto se používá pouze v ojedinělých případech, kdy by se program bez jeho použití značně znepřehlednil.

```
goto návěští;
.
.
návěští: příkaz;
```

Provedením příkazu `goto` se vykonávání programu přesune na příkaz, před kterým je uveden odpovídající identifikátor návěští následovaný dvojtečkou. Návěští nemusí být předem deklarováno, a protože pro návěští je vyhrazen vlastní jmenný prostor, mohou být jejich identifikátory shodné s identifikátory obyčejných proměnných

Příkaz break

```
break;
```

Příkaz `break` může být použit v tělech cyklů (`while`, `do-while`, `for`) a v těle příkazu `switch`, přičemž jeho použití způsobí okamžité opuštění cyklu (nebo příkazu `switch`). Jde tedy vlastně o skok na první příkaz za cyklem.

Příkaz continue

`continue;`

Příkaz `continue`, na rozdíl od `break`, může být použit pouze v tělech cyklů a způsobí okamžité započetí dalšího cyklu. Stejně jako u příkazu `break` se můžeme i na `continue` dívat jako na příkaz skoku, ale tentokrát se skočí za poslední příkaz těla cyklu.

```
while (1)
 { c = getch();
     // do c se načte znak z klávesnice
   if (c==27) break;
     // pokud je stisknuta klávesa ESC, je cyklus ukončen
   if (!isalpha(c)) continue;
     // není-li znak písmeno, začne se provádět znovu tělo cyklu,
     // takže už nedojde na...
   putch (c); // ...vytištění znaku
 }
```


Příkaz mnohonásobného větvení – switch (1)

Pokud potřebujeme tok programu větvit do více jak dvou směrů, můžeme místo několika do sebe vnořených příkazů `if-else` využít možností, které nám v C poskytuje příkaz `switch`.

```
switch (výraz1)
{
  case konstantní_výraz: příkazy
  case konstantní_výraz: příkazy
  .
  .
  default: příkazy
}
```

při provádění příkazu `switch` je nejdříve vyhodnocen výraz1 a pak se postupně vyhodnocují konstantní_výrazy v návěštích `case`. V případě, že je nalezeno návěští, kde je konstantní_výraz roven hodnotě výrazu1, začnou se provádět všechny příkazy uvedené za tímto návěštím až do konce příkazu `switch`. To ale znamená, že se provedou i všechny příkazy v následujících větvích. Pokud tedy chceme, aby se provedla vždy jen jedna větev, lze vykonávání příkazu `switch` okamžitě zastavit uvedením příkazu `break`. (Obdoba Pascalovského příkazu `case`).

Příkaz mnohonásobného větvení – switch (2)

Pokud není nalezena žádná vyhovující větev, skočí se na návěští default (pokud je použito), které může být uvedeno kdekoliv v těle příkazu `switch`.

```
int c;
.
.
.
switch (c)
 {
   case 1:
   case 2:
   case 3: printf ("Cislo 1, 2, nebo 3");break;
   case 4: printf ("Cislo 4"); break;
   default: printf ("Jine cislo, nez 1,2,3 nebo 4"); break;
 }
```


Podmíněný výraz - operátor "?"

Vedle podmíněného příkazu `if-else` existují v C ještě další způsoby, jak if-podmínky zapsat. Jedním z nich je použití operátoru "?", který je mimochodem jediný operátor v C mající tři operandy. Proto se pro něj často používá název **ternární** operátor.

`výraz1 ? výraz2 : výraz3`

Vyhodnocení výrazu s ternárním operátorem probíhá takto:

Nejdříve je vyhodnocen výraz1, který by měl být číselného typu. Je-li jeho hodnota určena jako nenulová, je následně vyhodnocen výraz2 a jeho hodnota je zároveň výslednou hodnotou celého výrazu. Výraz3 se vůbec nevyhodnotí a tudíž se neprovedou ani případné postranní efekty. V případě, že je výraz1 vyhodnocen jako nula, je naopak vyhodnocen výraz3 a jeho hodnota je také výsledkem celé operace.

Na rozdíl od příkazu if-else, použití operátoru "?" tvoří výraz a z toho také vyplývají odlišné možnosti jeho použití.

```
i = a>b?a:b;
// proměnné i se přiřadí vždy větší z čísel a a b
```

Operátor postupného vyhodnocování ","

```
výraz1 , výraz2
```

Operátor „čárka“ je jeden z mála operátorů, který zajišťuje pořadí vyhodnocení svých operandů. Nejdříve je vyhodnocen levý operand, tedy výraz1, jehož hodnota je ovšem zapomenuta, a který slouží především k vykonání postranních efektů. Jako druhý je pak vyhodnocen výraz2, jehož hodnota je pak i výslednou hodnotou celého výrazu.

Operátor čárky se často používá například v řídících částech příkazů for a while, kde jeho použití může vést ke zjednodušení zápisu, nebo kde slouží jako prostředek k vykonání dalších postranních efektů.

```
for (i=0, j=9; i<10; i++, j--) printf("%d %d\n", i, j);
//inkrementace i a dekrementace j
```


Zkrácené vyhodnocování logických operátorů

Při vyhodnocování výrazů s logickými operátory `&&` nebo `||` často stačí k určení výsledné hodnoty vyhodnotit pouze jeden z operandů.

Například u výrazů s operátorem logického součinu `&&`, kdy je první operand vyhodnocen jako nulový, nemůže již vyhodnocení druhého operandu nijak ovlivnit celkový výsledek, který bude také nulový.

V C se v takovém případě skutečně pravý operand nevyhodnocuje a tato skutečnost tedy může být využita jako další způsob podmíněného vyhodnocování.

```
i<0 && (i=0);
!(i<0) || (i=0);
```

Oba příklady mají stejný efekt: Je-li hodnota proměnné **i** menší než 0, bude tato hodnota nastavena na 0.

Funkce (1)

Funkce jsou nepostradatelné součásti všech strukturovaných jazyků, a tedy i jazyka C.

Definice funkce

```
návratový_typ identifikátor_funkce (seznam definicí formálních parametrů)
{ lokální deklarace a definice;
..
příkazy;
}
```

Definice funkce začíná tzv. hlavičkou funkce, což je v našem vzorovém příkladu první řádek. Návratový_typ určuje jakého typu bude hodnota, kterou bude funkce vracet. Jako další po identifikátoru typu píšeme identifikátor funkce následovaný seznamem definicí formálních parametrů. Formální parametry jsou pak proměnné nadefinované v tomto seznamu. Ty se definují stejně jako normální proměnné, jen s tím rozdílem, že nelze zkrátit zápis více proměnných stejného typu oddělením jednotlivých identifikátorů čárkou, jak ukazuje následující příklad:

```
int secti (int a,b)         //nelze
int secti (int a, int b)    //správný zápis
```


Funkce (2)

Po uvedení hlavičky funkce ještě následuje tělo funkce, které je tvořeno blokem. Jak už víme, na začátku bloku mohou být uvedeny lokální deklarace a definice, po kterých následují příkazy. Ty obvykle pracují s předanými parametry. Provádění funkce je ukončeno po vykonání posledního příkazu těla funkce, ale v takovém případě není možné odhadnout, jakou hodnotu funkce vrátí. Proto se používá příkaz `return`

```
return výraz1;
```

Tento příkaz způsobí okamžité opuštění funkce, ve které je použit. Hodnota nepovinného výrazu1 je pak návratovou hodnotou funkce. Pokud výraz1 neuvedeme, bude návratová hodnota předem neurčitelná. Pak bychom ale neměli tuto hodnotu nikde používat.

```
int secti (int a, int b)
{return a+b;}
```

V příkladu je nadefinována funkce, která vrací součet dvou čísel. Typ její návratové hodnoty je určen jako `int`, stejně jako typy obou parametrů funkce. Po předání řízení funkci se hned začne vykonávat příkaz `return`, který funkci ihned ukončí a jako výsledek po jejím volání vrátí součet hodnot předaných parametrů.

Funkce (3)

Deklarace funkce
Deklarace funkce je vlastně způsob jak dát překladači všechny potřebné údaje o funkci, aniž bychom ji museli celou definovat. Předtím, než funkci zavoláme, měla by být vždy předem definována, nebo deklarována. To proto, aby překladač znal všechny formální parametry, a tak mohl vytvořit správný kód. Pokud funkci nebudeme ještě před jejím voláním deklarovat ani definovat, bude překladač odhadovat formální parametry podle typu skutečných parametrů (parametry předané funkci při jejím volání), které ale nemusí odpovídat typu parametrů formálních. Výsledkem by pak byl nesprávně sestavený kód. Pokud ale z nějakého důvodu není funkce definována před svým použitím, měla by být alespoň deklarována.
Deklarace funkce vypadá takhle:

```
návratový_typ identifikátor_funkce (seznam definicí formálních parametrů);
```

Je to vlastně celá hlavička definice, která je ale zakončená středníkem.

```
int secti (int a, int b); // deklarace výše definované funkce.
```


Funkce (4)

Volání funkce

```
identifikátor_funkce (seznam parametrů)
```

Jednotlivé výrazy v seznamu parametrů jsou odděleny čárkou. Samotné parametry jsou pak výrazy, které jsou před předáním řízení funkci vyhodnoceny a jejich výsledné hodnoty jsou funkci předány. I když funkce nemá žádné skutečné parametry, závorky je nutné uvést vždy. Počet skutečných parametrů musí být vždy stejný nebo větší než počet parametrů formálních.

Samotné volání funkce je chápáno jako výraz a to určuje i místo jeho použití (např. operand některého operátoru, ve výrazovém příkazu, jako skutečný parametr jiné funkce).

Formální parametry jsou vlastně lokální proměnné, a tedy existují pouze po dobu vykonávání funkce a jsou viditelné pouze z těla funkce. Když vykonávání funkce skončí, je paměť vyhrazená pro tyto proměnné uvolněna a jako výsledek volání funkce se použije její návratová hodnota (většinou tedy hodnota výrazu za příkazem `return`).

```
a = secti(3,5); //do proměnné a se uloží výsledek po volání funkce secti(3,5)
```


Preprocesor jazyka C

Jistou zvláštností jazyka C je jeho preprocesor. Ten ještě před samotným překladem zdrojový soubor upraví a teprve **upravený soubor je předán překladači**. Mezi úpravy, které preprocesor provádí se řadí především substituce textu, odstraňování komentářů a podmíněný překlad.

Činnost preprocesoru řídíme pomocí tzv. **direktiv** preprocesoru. Každá direktiva je uvozena znakem #, který musí být uveden hned jako první znak na řádku.
Tak určíme, že zbytek řádku je určen preprocesoru a zápisy v něm se tedy řídí jeho syntaktickými pravidly, která nejsou totožná s těmi céčkovskými.

Direktiva `#include`

`#include <soubor>` nebo `#include "soubor"`

Pokud preprocesor narazí na výskyt direktivy `#include`, nahradí ji obsahem určeného souboru. To se nejčastěji používá pro vkládání tzv. **hlavičkových souborů** s deklaracemi funkcí apod., nebo přímo **jiných zdrojových souborů C**. Jak jste si jistě všimli, je možné v zápisu direktivy ohraničit jméno souboru buď lomenými závorkami, nebo uvozovkami. Pokud použijete zápisu se závorkami a nespecifikujete úplnou cestu k souboru, bude soubor hledán ve standardním adresáři pro ukládání hlavičkových souborů. Použijete-li zápis s uvozovkami, bude soubor hledán nejdříve v adresáři se zdrojovým souborem a pak teprve v adresáři s hlavičkovými soubory.

Direktiva #define

```
#define identifikátor_makra text_makra
```

Tato direktiva se používá pro vytváření tzv. **maker**.

Makra se často používají pro definování tzv. **symbolických konstant**, kdy místo konstanty používáme nějaké symbolické jméno. Ještě před překladem tak budou všechny výskyty tohoto symbolického jména nahrazeny skutečnou hodnotou.

```
#define PI 3.141592653
```

Předpokládejme, že máme ve svém zdrojáku nadefinována makro z předchozího příkladu a že tam máme také následující řádky:

```
double c;
printf ("Cislo PI ");
c = PI;
```

Po zpracování preprocesorem bude předchozí zápis vypadat takto:

```
double c;
printf ("Cislo PI ");
c = 3.141592653;
```

V parametru funkce `printf` nebylo nahrazení textem makra provedeno, protože v řetězcích se nahrazování neprovádí.

Pole (1)

Definice proměnné typu pole

- Proměnná typu pole se definuje podobně jako proměnná jednoduchého typu. Rozdílem je, že při definici pole se za jménem identifikátoru proměnné ještě uvádějí hranaté závorky, ve kterých určíme počet prvků pole.

```
bázový_typ identifikátor[počet_prvků];
```

- Bázový typ neurčuje typ proměnné, ale typ položek pole. Že jde o pole pozná překladač právě podle hranatých závorek. Počet_prvků je jakýkoliv konstantní výraz, tedy takový, který lze vyhodnotit již při překladu.

```
short int moje_pole[10];
```

- Nadefinovali jsme proměnnou typu pole, která obsahuje 10 položek typu short int. Znamená to tedy, že se pro naši proměnnou vyhradilo 20 bytů (10 * sizeof(short int)).

Pole (2)

Prvky pole lze inicializovat již při definici. Stačí za poslední hranatou závorku uvést znak '=' následovaný seznamem inicializačních výrazů, jak ukazuje následující příklad:

```
short int moje_pole[5]={1, 0, 443, -46, 987};
```

Inicializační výrazy se přiřazují postupně, tak jak jsou zapsány. Pole *moje_pole* bude naplněno hodnotami takto:

index:	0	1	2	3	4
pole:	1	0	443	-46	987

- V případě, že počet inicializačních výrazů je vyšší než počet položek pole, bude se při překladu hlásit chyba. Pokud je počet inicializátorů menší, chyba se nehlásí a zbylé položky jsou inicializovány buď nulovou hodnotou, nebo nejsou inicializovány vůbec.
- Pokud při definici zároveň inicializujeme prvky pole, nemusíme specifikovat velikost pole, ale stačí když uvedeme prázdné závorky. Překladač sám určí velikost pole podle počtu inicializačních výrazů.

```
short int moje_pole[]={1, 0, 443, -46, 987};
```


Pole (3)

- V jazyce C jsou všechny proměnné typu pole indexovány od nuly. To znamená, že první položka má vždy index 0, což není možné nijak ovlivnit. Budeme-li tedy chtít získat první položku pole, použijeme zápis:

```
moje_pole[0]
```

- Je důležité si uvědomit, že indexací nezískáváme pouze hodnotu prvku pole, ale přímo prvek samotný, tedy l-hodnotu. Je tedy možné použít tento zápis i na levé straně přiřazovacího výrazu:

```
int moje_pole[4];
moje_pole[0]=250;
```

Tímto zápisem jsme do první položky pole zapsali hodnotu 250.

Procházení pole

- Při procházení položek pole je nutné dát si pozor na to, abychom nikdy omylem nepřekročili hranice pole. Jazyk C totiž zásadně nekontroluje meze polí, a tak je bez problémů možné číst i zapisovat do paměti, která nám již nepatří. To v nejhorším případě může vést až ke zhroucení programu, nebo celého počítače. Poslední prvek pole o n položkách tedy bude mít index n-1.

Řetězce (1)

Jazyk C nemá implementovaný speciální datový typ pro řetězce. Ty jsou proto v C reprezentovány jako pole prvků typu `char`, kde je v každém prvku uložena **ascii hodnota** příslušného znaku řetězce. Jako poslední musí být vždy uveden znak **EOS (end of string)**, což je znak s ascii hodnotou 0. Ten označuje konec řetězce.

Z toho, jak jsou řetězce ukládány, je jasné, že velikost řetězce je omezena jedině velikostí paměti, kterou pro něj lze alokovat.

Řetězce (2)

Inicializace řetězce při definici

Řetězce se definují stejně jako pole. Pouze je třeba dát si pozor na to, abychom při definování velikosti nezapomněli na znak EOS. I pole prvků char lze samozřejmě inicializovat při definici, a to stejným způsobem jako obyčejná pole, tedy výčtem jednotlivých prvků:

```
char string[]={'p','o','l','e','\0'};
```

Jednotlivé znakové konstanty představují ascii hodnoty, které jsou do pole `string` uloženy. Jako poslední znak pak nesmíme zapomenout vložit znak EOS. Tento způsob inicializace ale není zrovna moc pohodlný, a tak je možné při definici inicializovat přímo řetězcovou konstantou:

```
char string[]="pole";
```

Tento zápis je zcela ekvivalentní s předchozím. Pro pole string je vyhrazeno 5 bytů, do kterých je uložen řetězec „pole“ včetně znaku EOS.

Kopírování řetězců

```
char *strcpy(char cil[], char zdroj[])
```

Pro kopírování jednoho řetězce do druhého je v C připravena funkce `strcpy`, která má dva argumenty, řetězce *cil* a *zdroj*.

Bez ohledu na obsah a velikost řetězce *cil* je do něho postupně, znak po znaku, kopírován obsah řetězce *zdroj*, a to až do doby kdy se narazí na znak EOS. Tento znak je posledním zkopírovaným znakem. Při používání funkce `strcpy` je nutné zajistit, aby pole *cil* mělo vždy dostatečnou velikost na to, aby se do něj řetězec *zdroj* vešel. To ale platí i pro většinu ostatních funkcí pracujících s řetězci. Návratovou hodnotou funkce `strcpy` je ukazatel na řetězec *cil*, ale protože o souvislosti polí s ukazateli si povíme až někdy jindy, můžeme prozatím tuto informaci pominout.

Spojování řetězců

Ke spojování řetězců můžeme použít funkci strcat:

```
char * strcat (char cil[], char zdroj[])
```

Tato funkce jednoduše připojí řetězec *zdroj* za řetězec *cil*. Stejně jako u funkce `strcpy`, ani u `strcat` se neberou ohledy na skutečnou velikost paměti alokované pro pole *cil*. To musí být dostatečně velké, aby pojalo oba řetězce *cil* a *zdroj* i se znakem EOS.

Zjišťování délky řetězce

`int strlen(char str[])`

Pro zjištění délky řetězce nám jazyk C nabízí funkci `strlen`, která jako svou návratovou hodnotu vrací délku řetězce, který jí byl předán parametrem.

Porovnávání dvou řetězců

`int strcmp (char str1[], char str2[])`

Funkce `strcmp` porovnává řetězce *str1* a *str2* a vrací zápornou hodnotu v případě, že řetězec *s1* je lexikograficky menší než řetězec *s2* a kladné číslo v případě, že *s1* je větší než *s2*. Jsou-li oba řetězce stejné, je funkcí vrácena hodnota 0.

Příklad:

```
char s1[]="retezec";
char s2[]="pole";
char s3[20];
strcpy(s3,s1);           // zkopíruje s1 do s3
strcat(s3, "a");         // připojí řetězec "a" k s3
strcat(s3, s2);          // připojí řetězec s2 k s3
                         // v proměnné s3 je teď uložen
                         // řetězec "retezec a pole";
printf("%d",strlen(s3)); // vypíše délku řetězce s3,
                         // číslo 14
```


Vícerozměrná pole (1)

Vícerozměrná pole se definují podobně jako jednorozměrná. Stačí pouze připojit další dvojici hranatých závorek, která poli přidá novou dimenzi.

`int a[10];` definice jednorozměrného pole o deseti prvcích

`int b[10][5];` definice dvourozměrného pole 10x5

Zápis definice proměnné a přečteme obvyklým způsobem: “a je pole desíti prvků, jejichž typ je int.”

Čtení druhé definice bude o trochu složitější. Jak už bylo řečeno, lze se na vícerozměrná pole dívat jako na pole prvků, kde tyto prvky jsou jiná pole.

Zápis druhé definice tedy přečteme takto: “b je pole desíti prvků, jejichž typ je pole pěti prvků typu int.”

Vícerozměrná pole (2)

Přístup k prvkům pole

Stejně jako při definici, i při indexaci pole stačí přidat další index (ve vlastních hranatých závorkách).

Vraťme se znovu k následující definici:

`int b[10][5];`

Opět využijeme způsobu nazírání jako na jednorozměrné pole. Použitím jednoho indexu získáme některý prvek pole b. Například zápisem `b[6]` získáme sedmý prvek pole b.

Tento prvek je ale sám pěti-prvkové pole.

Další indexací se tedy pohybujeme mezi prvky tohoto pole. Například `b[6][3]`.

Vícerozměrná pole (3)

Ukážeme si to na příkladu, ve kterém si vytvoříme pole reprezentující následující matici:

```
1  2  3      b[0],[0]   b[0],[1]  b[0],[2]
4  5  6      b[1],[0]   b[1],[1]  b[1],[2]
7  8  9      b[2],[0]   b[2],[1]  b[2],[2]
10 11 12     b[3],[0]   b[3],[1]  b[3],[2]
```

Nejprve vytvoříme nové neinicializované pole 4x3:

```
int b[4][3];
```

Pomocí prvního indexu tedy budeme vybírat řádek a druhým se budeme pohybovat po prvcích tohoto řádku. Teď můžeme pole naplnit příslušnými hodnotami:

```
int i,j, k=1;
for (i=0; i<4; i++)
   for (j=0; j<3; j++, k++) b[i][j]=k;
```


Vícerozměrná pole (4)

Inicializace při definici

I vícerozměrná pole se dají inicializovat již při definici, a to stejně jako pole jednorozměrná.

Opět si ale musíme uvědomit, že jednotlivými prvky pole jsou jiná pole, a tak jako inicializátory musíme uvádět další seznamy inicializačních výrazů.

```
int a[2][3]={{22,1,16},{112,0,4}};
int b[3][2]={{22,1},{16,112},{0,4};
```

Pole a a b z výše uvedených definic pak budou vypadat takto:

```
Pole a                Pole  b
 22  1  16           22    1
112  0   4           16  112
                       0    4
```


Vstup ze souboru a výstup do souboru

Z hardwarového hlediska je každý soubor posloupnost bajtů uložených na nějakém médiu (nejčastěji disku) v několika blocích. Jak se s bloky pracuje je záležitost operačního systému a nás to nemusí zajímat. Přístup k souboru je možný sekvenčně, tak i náhodně.

Základní datový typ pro práci se souborem v jazyce C:
`FILE *` - což je pointer na objekt typu `FILE`
(zatím nevíme co to znamená, viz později)

Definice proměnné `f` pro práci se souborem

```
FILE *f;
```

- Identifikátor `FILE` musí být velkými písmeny
- Proměnná `f` se dá použít jak pro čtení, tak i pro zápis do souboru
- Chceme-li definovat více proměnných, čili pracovat s více soubory najednou (např. pro čtení a zápis), musí se znak * opakovat, tj. `FILE *fr, *fw;`

Vstup ze souboru a výstup do souboru (2)

Otevření souboru pro čtení

Soubor POKUS bude možné jen číst.

```
f = fopen("Pokus", "r")
```
"r" jako read

Otevření souboru pro zápis

Do souboru POKUS bude možné jen zapisovat

```
f = fopen("Pokus", "w")
```
"w" jako write

Pozn.:

- Existují i další režimy otevření souboru (kromě "r" a "w")
- Některé kompilátory rozlišují režimy otevření pro **textový** nebo **binární** soubor. V dalším předpokládáme textový režim
- Obecně platí, že "w" nebo "r" bez dalšího písmene znamená otevření souboru v textovém režimu.

Základní operace s otevřeným souborem

Funkce ze standardní knihovny popsané ve `stdio.h`, které umožňují pracovat se souborem:
(Proměnná `f` je typu `FILE*`).

Čtení znaku ze souboru	`c = getc(f)`
Zápis znaku do souboru	`putc(c, f)`
Formátované čtení ze souboru	`fscanf(f, "formát", argumenty)`
Formátovaný zápis do souboru	`fprintf(f, "formát", argumenty)`

Pozor.:
U funkce `putc()` je první parametr zapisovaný znak a druhý soubor. To se často plete s funkcí `fprintf()`, kde je to obráceně)

Pro osvěžení paměti a také jako ukázku, že se práce se soubory příliš neliší od práce s obrazovkou a klávesnicí, je uveden i přehled korespondujících (již známých) funkcí

Čtení znaku z klávesnice	`c = getchar()`
Zápis znaku na obrazovku	`putchar(c)`
Formátované čtení z klávesnice	`scanf("formát",argumenty)`
Formátovaný zápis na obrazovku	`printf("formát",argumenty)`

Ukončení práce se souborem

Po skončení práce se souborem (už z něho nebudeme dále číst nebo do něho nebudeme dále zapisovat) je nutné tuto skutečnost operačnímu systému sdělit. Tato akce se jmenuje **uzavření souboru** a provádí se pomocí funkce `fclose(f)`, kde `f` je typu `FILE *`.

Příklady základní práce se soubory (1)

Program vytvoří soubor `POKUS.TXT` a zapíše do něho čísla od 1 do 10, každé na nový řádek.

```
#include <stdio.h>
main()
{
  FILE *fw;
  int  i;
  fw = fopen("POKUS.TXT", "w");
  for (i = 1;  i <= 10;  i++)
    fprintf(fw, "%d \n", i);
  fclose(fw);
}
```


Příklady základní práce se soubory (2)

Program přečte tři **double** čísla ze souboru `DATA.TXT` a vypíše na obrazovku jejich součet.

```
#include <stdio.h>
main()
{
  FILE *fr;
  double x, y, z;

  fr = fopen("DATA.TXT", "r");
  fscanf(fr, "%lf %lf %lf", &x, &y, &z);
  printf("%f\n", x + y + z);
  fclose(fr);
}
```

Příklady základní práce se soubory (3)

Program přečte dva znaky ze souboru ZNAKY.TXT a zapíše je do souboru KOPIE.TXT.

```
#include <stdio.h>
main()
{
  FILE *fr, *fw;
  int c;

  fr = fopen("ZNAKY.TXT", "r");
  fw = fopen("KOPIE.TXT", "w");

  c = getc(fr);              /* cteni prvniho znaku */
  putc(c, fw);               /* zapis prvniho znaku */
  putc(getc(fr), fw);        /* cteni a zapis druheho znaku */

  fclose(fr);
  fclose(fw);
}
```


Parametry funkce `main` (1)

Jak už víme, funkce `main` má mezi ostatními funkcemi v C výsadní postavení, neboť, kromě toho že musí být vždy definována, je automaticky spouštěna ihned po startu programu.

Parametry funkce main se v C využívají pro **získání argumentů**, které byly našemu programu předány při jeho spuštění. Dá se říct, že pokud program reaguje na jemu předané parametry, je to obecně velmi užitečná vlastnost. Ne vždy je totiž interakce programu s uživatelem vítaná, neboť mnohé úkoly lze zpracovávat dávkově, a tedy mnohem rychleji. Chceme-li tedy v našich programech využít možnost práce s parametry příkazové řádky, definujeme hlavičku funkce main typicky takto:

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

Prvním parametrem je zde proměnná `argc`, která v sobě nese informaci o počtu parametrů. Druhý parametr `argv` pak představuje pole řetězců, ve kterých jsou tyto jednotlivé parametry uloženy. Formální parametry funkce `main` se z historických důvodů pojmenovávají vždy právě jako `argc` a argv. Kromě samotných parametrů je v poli `argv`, jako jeho nultá položka, uložen i řetězec se jménem spouštěného programu. Hodnota parametru `argc` uvažuje i tento řetězec, a tak, pokud programu předáme například tři parametry, bude mít argc hodnotu 4.

Parametry funkce `main` (2)

Uvažujme, že máme funkci `main` nadefinovanou výše uvedeným způsobem, a přeložený program spustíme s dvěma parametry například takto:

```
program.exe pr1 pr2
```

Pak bude v proměnné `argc` uložena hodnota 3 a první tři položky pole budou obsahovat tyto řetězce:

```
argv[0] = "program.exe„
argv[1] = "pr1"
argv[2] = "pr2"
```


Parametry funkce `main` (3)

Jako příklad si vyzkoušíme jednoduchý program, který načte jemu předané parametry a spolu s údajem o jejich počtu je vypíše.

```
int main(int argc, char *argv[])
{
 int i;
  printf("Pocet parametru: %d\n", argc);
   for (i=0; i<argc; i++)
    printf("argv[%d] == \"%s\"\n", i,argv[i]);
 return 0;
}
```

Závěr a doporučená literatura

- Text této přednášky rozhodně není učebnicí programování v jazyku C, protože neobsahuje kompletní popis jazyka C
- Záměrem bylo pouze ukázat některé možnosti pro první kroky při výuce tohoto jazyka
- Pro bližší studium se doporučují následující publikace:
  - SCHILDT, H: *Nauč se sám C*, Softpress, Praha, 2001. ISBN 80-86497-16-X
  - ECKEL, B. *Myslíme v jazyku C++*. Praha: Grada Publishing, 2002. ISBN 8-0247-9009-2
  - VIRIUS, M. *Programovací jazyky C/C++*. Praha: Gcomp, 1992. ISBN 8-0901-0735-4.
  - HEROUT, P. *Učebnice jazyka C*. České Budějovice: Kopp, 1992. ISBN 8-0858-2821-9
  - HEROUT, P. *Učebnice jazyka C, 2.díl*. Č. Budějovice: Kopp, 1992. ISBN 80-85828-50-2
  - Internet:
    - Např. www.builder.cz