Základy programování v C

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 02

B0B36PRP – Procedurální programování

Přehled témat

Část I

Část 1 – Základy programování v C

Jazyk C

- Nízko-úrovňový programovací jazyk.
- Systémový programovací jazyk (operační systém).

  Jazyk pro vestavné (embedded) systémy — MCU, křížová (cross) kompilace.
- Téměř vše nechává na uživateli/ce (programátorovi/ce).

  Inicializace proměnných, uvolňování dynamické paměti.
- Má blízko k využití hardwarových zdrojů výpočetního systému.

  Přímé volání služeb OS, přímý zápis do registrů a portů.
- Klíčové pro správné fungování programu je zacházení s pamětí.

Cílem kurzu PRP je naučit se základním principům, které lze následně generalizovat též pro jiné programovací jazyky. Pochopení těchto principů je klíčem k efektivnímu psaní efektivních programů.

Je výhodné mít překlad programu plně pod kontrolou.

Přestože to může z počátku vypadat složitě, jsou základní principy relativně jednoduché. I proto je výhodné používat základní nástroje pro překlad programů a po jejich osvojení využít komplexnější vývojové prostředí.

Správnost programu

- Syntakticky i staticky sémanticky správný program neznamená, že dělá to co od něj požadujeme.
- Správnost a smysluplnost programu je dána očekávaným chováním při řešení požadovaného problému.
- V zásadě při spuštění programu mohou nastat následující události.
  - Program havaruje a dojde k chybovému výpisu.

    Mrzuté, ale výpis (report) je dobrý start řešení chyby (bug).
  - Program běží, ale nezastaví se a počítá v nekonečné smyčce.

    Zpravidla velmi obtížné detekovat a program ukončujeme po nějaké době, proto je vhodné mít představu o výpočetní náročnosti řešené úlohy a použitém přístup řešení (algoritmu).
  - Program včas dává odpověď.

    Je však dobré vědět, že odpověď je korektní.

Správnost programu je plně v režii programátorky nebo programátora, proto je důležité pro snadnější ověření správnosti, ladění a hledání chyby používat **dobrý programovací styl**.

Zdrojové soubory programu v C

- **Zdrojový** soubor s koncovkou **.c**.

  Zpravidla—základní rozlišení souborů, viz např. .C.
- **Hlavičkový** soubor s koncovkou **.h**.

  Jména souborů volíme výstižné (krátké názvy) a zpravidla zapisujeme malými písmeny.
- Zdrojové soubory jsou překládány do binární podoby překladačem a vznikají objektové soubory (**.o**) nebo spustitelný program.

  Objektový kód obsahuje relativní adresy proměnných a volání funkcí nebo pouze odkazy na jména funkcí, jejichž implementace ještě nemusejí být známy.
- Z objektových souborů (object files) se sestavuje výsledný program, ve kterém jsou již všechny funkce známy a relativní adresy se nahradí absolutními.

  Program se zpravidla sestavuje z více objektových souborů umístěných například v knihovnách.
- Zdrojový kód se skládá z volání **klíčových slov** a funkcí, které pracují s proměnnými a číselnými hodnotami (případně řetězci) nazývané také (**literály**).
  - **Klíčová slova** a jejich použití je definováno jazykem.
  - Jména funkcí a proměnných jsou **identifikátory**, které navrhuje programátor/ka.

    Vyhrazená jména jazyka a std. knihovny, např. `main`.
  - Hodnoty (**literály**) mají svůj typ, tj. velikost v paměti!

Definice programovacího jazyka

- **Syntax** – definice povolených výrazů a konstrukcí programu.

  Plná kontrola a podpora vývojových prostředí.
  - Příklad popisu výrazu gramatikou v Backus-Naurově formě
    <exp> ::= <exp> + <exp> | <exp> * <exp> | <exp> | <number> <number> ::= <number> <digit> | <digit> <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
- **Statická sémantika** – definuje jak jsou konstrukty používány.

  Částečná kontrola a podpora prostředí.
  - Příklad axiomatické specifikace: $\{P\}\ S\ \{Q\}$,
    - P- precondition,
    - Q-postcondition,
    - S - konstrukce jazyka.
- **Plná sémantika** – co program znamená a dělá, jeho smysluplnost.

  Kontrola a ověření správnosti je kompletně v režii programátora/ky.

Platné znaky pro zápis zdrojových souborů

- Malá a velká písmena, číselné znaky, symboly a oddělovače.

  ASCII – American Standard Code for Information Interchange.
  - a–z A–Z 0—9
  - ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~
  - mezera, tab, nový řádek.
- Escape sekvence pro symboly
  - \' – ', \" – ", \? – ?, \\– \.
- Escape sekvence pro tisk číselných hodnot v textovém řetězci
  - \o, \oo, kde o je osmičková číslice;
  - \xh, \xhh, kde h je šestnáctková číslice.

```
1 int i = 'a';
2 int h = 0x61;
3 int o = 0141;
4
5 printf("i: %i h: %i o: %i c: %c\n", i, h, o, i);
6 printf("oct: \141 hex: \x61\n");
```

Např. \141, \x61 `lec02/esqdho.c`
- \0 – znak pro konec textového řetězce (null character).

Identifikátory a klíčová slova

- **Identifikátory** jsou jména proměnných, funkcí, a vlastních typů.

  Vlastní typy a funkce viz další přednášky.
- Pravidla pro volbu identifikátorů.

  Názvy proměnných, typů a funkcí.
  - Znaky a–z, A–Z, 0–9 a _.
  - První znak není číslice.
  - Rozlišují se velká a malá písmena (case sensitive).
  - Délka identifikátoru není omezena.

    Prvních 31 znaků je významných – může se lišit podle implementace.
- Klíčová (rezervovaná) slova (keywords)$_{32}$:

auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while. C98

C99 dále rozšiřuje například o `inline`, `restrict`, `_Bool`, `_Complex`, `_Imaginary`.

C11 pak dále například o `_Alignas`, `_Alignof`, `_Atomic`, `_Generic`, `_Static_assert`, `_Thread_local`.

Funkce

- Funkce tvoří základní stavební blok **modulárního** jazyka C.

Modulární program je složen z více modulů/zdrojových souborů.

- Každý spustitelný program v C obsahuje *alespoň* jednu funkci a to funkci `main()`.
  - Běh programu začíná funkcí `main()`.
- **Deklarace** se skládá z hlavičky funkce.

```
typ_návratové_hodnoty jméno_funkce(seznam parametrů);
```

C používá **prototyp (hlavičku) funkce** k **deklaraci** informací nutných pro překlad tak, aby mohlo být přeloženo správné volání funkce i v případě, že **definice** je umístěna dále v kódu.

- **Definice** funkce obsahuje **hlavičku funkce a její tělo**, syntax:

```
typ_návratové_hodnoty jméno_funkce(seznam parametrů)
{
    //tělo funkce
}
```

Definice funkce bez předchozí deklarace je zároveň deklarací funkce.

Vlastnosti funkcí

- C nepovoluje funkce vnořené do jiných funkcí.
- Jména funkcí se mohou exportovat do ostatních modulů.

Modul je samostatně překládaný soubor.

- Funkce jsou implicitně deklarovány jako **extern**, tj. viditelné.
- Specifikátorem **static** před jménem funkce omezíme viditelnost jména funkce pouze pro daný modul.

Lokální funkce modulu

- Formální parametry funkce jsou **lokální proměnné**, které jsou inicializovány skutečnými parametry při volání funkce.

*Parametry se do funkce předávají **hodnotou** (**Call by Value**).*

- **C dovoluje rekurzi** – lokální proměnné jsou pro každé jednotlivé volání zakládány znovu na zásobníku.

Kód funkce v C je reentrantní ve smyslu volání funkce ze sebe sama.

- Funkce nemusí mít žádné vstupní parametry, zapisujeme klíčovým slovem `void`.

```
fce(void)
```

- Funkce nemusí vracet funkční hodnotu–návratový typ je `void`.

```
void fce(void)
```

`lec02/function.c`

Struktura programu / modulu

```
1  #include <stdio.h> /* hlavickovy soubor */
2  #define NUMBER 5 /* symbolicka (textova) konstanta */
3
4  int compute(int a); /* hlavicka/prototyp funkce */
5
6  int main(int argc, char *argv[])
7  { /* hlavni funkce */
8     int v = 10; /* definice promennych */
9     int r;
10    r = compute(v); /* volani funkce */
11    return 0; /* ukonceni hlavni funkce */
12 }
13
14 int compute(int a)
15 { /* definice funkce compute */
16   int b = 10 + a; /* telo funkce */
17   return b; /* navratova hodnota funkce */
18 }
```


Příkaz `return`

- Příkaz ukončení funkce `return vyraz;`.
- `return` lze použít pouze v těle funkce.
- `return` ukončí funkci, vrátí návratovou hodnotu funkce určenou hodnotou `vyraz` a předá řízení volající funkci.
- `return` lze použít v těle funkce vícekrát.

Kódovací konvence však může doporučovat nejvýše jeden výskyt return ve funkci.

- U funkce s prázdným návratovým typem, např. `void fce()`, nahrazuje uzavírací závorka těla funkce příkaz `return;`.

```
void fce(int a)
{
   ...
}
```


Překlad (kompilace) a spuštění programu

- Zdrojový soubor `program.c` přeložíme do spustitelné podoby kompilátorem např. `clang` nebo `gcc`

```
clang program.c
```

- Vznikne soubor `a.out`, který můžeme spustit např.

```
./a.out
```

Alternativně pouze jako `a.out` pokud je aktuální pracovní adresář nastaven v prohledávané cestě spustitelných souborů

- Program po spuštění vypíše text uvedený jako argument `printf()`

```
./a.out
I like B0B36PRP!
```

- Pokud nechce psát `./a.out` ale raději jen `a.out` lze přidat aktuální pracovní adresář do cest(y) definované proměnnou prostředí `PATH`

```
export PATH="$PATH:‘pwd‘"
```

Pracovních adresářů můžete mít více—používejte obezřetně.

- Příkaz `pwd` vytiskne aktuální pracovní adresář, více viz `man pwd`

Ano jde to, ale není dobrý nápad!

Důležité je mít povědomí, že existuje něco jako proměnná prostředí `PATH`

Struktura zdrojového souboru – Komentovaný zdrojový soubor `program.c`

```
1  /* komentar zapisujeme do dvojice vyhrazenych znaku */
2  // Nebo v C99 jako jednoradkovy
3  #include <stdio.h> /* vlozeni hlavickoveho souboru standardni knihovny
       stdio.h */
4
5  int main(void) // zjednodusena hlavicka funkce main()
6  {  // hlavni funkce program main()
7     printf("I like B0B36PRP!\n"); /* volani funkce printf() z knihovny
       stdio.h pro tisk textoveho retezce na standardni vystup. Znak \n
       definuje novy radek (odradkovani). */
8
9     return 0; /* ukonceni funkce a predani navratove hodnoty 0
       operacnimu systemu */
10 }
```


Zdrojové soubory

Proč psát program do více souborů?

- Rozdělení na zdrojové a hlavičkové soubory umožňuje rozlišit **deklaraci** a **definici**, která podporuje následující.
  - **Organizaci** zdrojových kódů v adresářové struktuře souborů.
  - **Modularitu**
    - Hlavičkový soubor obsahuje popis co modul nabízí, tj. popis (seznam) funkcí a jejich parametrů bez konkrétní implementace (**deklarace funkcí**).

Deklarování, že funkce existují a jaké mají rozhraní (vstup a výstup) argumenty a návratový typ udávající velikost paměti pro předávaná data.

  - **Znovupoužitelnost**
    - Pro využití binární knihovny potřebuje znát její „rozhraní“, které je deklarované v hlavičkovém souboru.

Pro jednoduché programy a domácí úkoly nedává moc smysl.

Vyplatí se především v HW08 a HW 10, případně HW06 (Matice)!

Překlad a sestavení programu

- Vytvoření programu `program` ze zdrojového souboru `program.c` příkazem

clang program.c -o program

slučuje překlad a sestavení programu do jediného volání příkazu (`clang` nebo `gcc`).

- Překlad se však skládá ze tří hlavních částí, které lze provést individuálně .

1. Textové předzpracování **preprocesorem** využívající makro jazyk (příkazy uvozeny znakem #)

Všechny odkazované hlavičkové soubory se vloží do jediného zdrojového souboru

2. Vlastní překlad zdrojového souboru do objektového souboru.

*Zpravidla jsou jména souborů zakončena příponou **.o**.*

clang -c program.c -o program.o

Příkaz kombinuje volání preprocesoru a kompilátoru.

3. Spustitelný soubor se sestaví z příslušných dílčích objektových souborů a odkazovaných knihoven, tzv. „linkováním“ (**linker**).

clang program.o -o program

nebo s vložením matematické knihovny **clang program.o -lm -o program**.

Např. pokud použijeme funkci `sqrt` z knihovny `math.h`.

Schéma překladu a sestavení programu

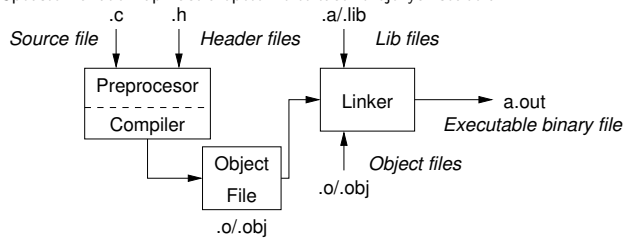
- Vývoj programu se skládá z editace zdrojových souborů (.c a .h).

Lidsky čitelných

  - Kompilace dílčích zdrojových souborů (.c) do objektových souborů (.o nebo .obj).

Strojově čitelných

  - Linkování přeložených souborů do spustitelného programu.
- Spouštění a ladění aplikace a opětovné editace zdrojových souborů.



Části překladu a sestavení programu

- **Preprocesor** – umožňuje definovat makra a tím přizpůsobit překlad aplikace kompilačnímu prostředí.
  Výstupem je textový („zdrojový") soubor.
- **Compiler** (**kompilátor**) – Překládá zdrojový (textový, lidsky čitelný) soubor do strojově čitelné (spustitelné) podoby.
  Nativní (strojový) kód platformy, bytecode, případně assembler.
- **Linker** (**sestavovací program**) – sestavuje program z objektových souborů do podoby výsledné aplikace.
  Stále může odkazovat na knihovní funkce (dynamické knihovny linkované při spuštění programu), může též obsahovat volání OS (knihovny).
  Linkerem také vytváříme knihovny a dynamicky linkované knihovny.
- Dílčí části **preprocesor**, **compiler**, **linker** jsou zpravidla „jediný" program, který se volá s příslušnými parametry.

Překladače jazyka C

- V rámci předmětu PRP budeme používat především překladače z rodin.
  - `gcc` – GNU Compiler Collection.
    `https://gcc.gnu.org`
  - `clang` – C language family frontend for LLVM.
    `http://clang.llvm.org`

  Pro win platformy pak odvozená prostředí cygwin https://www.cygwin.com/ nebo MinGW http://www.mingw.org/.*
- Základní použití (přepínače a argumenty) je u obou překladačů stejné.
  Clang je kompatibilní s gcc.
- Příklad použití
  - compile:
    `gcc -c program.c -o program.o`
  - link:
    `gcc program.o -o program`

Spustitelný program – `main()`

- Každý spustitelný program musí obsahovat jedinou definici funkce `main()`, jejíž vykonávání je zahájeno po spuštění programu.
- Funkce `main()` má základní tvar pro předání argumentů programu prostřednictví operačního systému.

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

- Případně s rozšířením o proměnné prostředí.

```
int main(int argc, char **argv, char **envp)
{
   ...
}
```


Argumenty funkce `main()`

- Při spuštění programu předává OS programu počet argumentů (`argc`) a argumenty (`argv`), jako pole textových řetězců (OS zajišťuje správnou alokaci paměti).
  - První argument je jméno programu.

```
1 int main(int argc, char *argv[])
2 {
3    int v;
4    v = 10;
5    v = v + 1;
6    return argc;
7 }
```

`lec02/var.c`

- Program je ukončen voláním `return` ve funkci `main()`.
- Návratová hodnota je předána OS, kde je možné ji dále použít.
  Např. v příkazovém interpretu pro řízení spouštění programů.

Příklad kompilace a spuštění programu

- Sestavení programu kompilátorem `clang` – automaticky dojde ke kompilaci a linkování programu do spustitelného souboru `a.out`.
  `clang var.c`
- Specifikací jména (výstupního – **o**utput) spustitelného souboru, např. `var`.
  `clang var.c -o var`
- můžeme program spustit v rámci aktuálního pracovního adresáře (viz `pwd`).
  `./var`
- Kompilaci a spuštění můžeme spojit do dvojice příkazů.
  `clang var.c -o var; ./var`
- nebo můžeme spuštění podmínit úspěšnou kompilací programu (`EXIT_SUCCESS`).
  `clang var.c -o var && ./var`

Návratová hodnota programu — 0 znamená OK, chyb může být více.
Logický operátor && závisí na příkazovém interpretu, např. `sh`, `bash`, `zsh`.

Příklad spuštění programu

- Navratová hodnota je uložena v proměnné **`$?`**.
  sh, bash, zsh
- Příklad spuštění s různým počtem argumentů.

```
./var

./var; echo $?
1

./var 1 2 3; echo $?
4

./var a; echo $?
2
```


Příklad zpracování zdrojového souboru preprocesorem

- Příznakem **`-E`** můžeme při „kompilaci" vyvolat pouze preprocesor.
  `gcc -E var.c`
  Nebo třeba `clang -E var.c`

```
1  # 1 "var.c"
2  # 1 "<built-in>"
3  # 1 "<command-line>"
4  # 1 "var.c"
5  int main(int argc, char **argv) {
6     int v;
7     v = 10;
8     v = v + 1;
9     return argc;
10 }
```

`lec02/var.c`

Příklad kompilace zdrojového kódu do Assembleru

- Příznakem **`-S`** kompilujeme zdrojový kód do Assembleru.
  `clang -S var.c -o var.s`

```
1    .file "var.c"
2    .text
3    .globl  main
4    .align  16, 0x90
5    .type main,@function
6  main:
       # @main
7    .cfi_startproc
8  # BB#0:
9    pushq %rbp
10 .Ltmp2:
11   .cfi_def_cfa_offset 16
12 .Ltmp3:
13   .cfi_offset %rbp, -16
14   movq  %rsp, %rbp
15 .Ltmp4:
16   .cfi_def_cfa_register %rbp
17   movl  $0, -4(%rbp)
18   movl  %edi, -8(%rbp)
19   movq  %rsi, -16(%rbp)
20   movl  $10, -20(%rbp)
21   movl  -20(%rbp), %edi
22   addl  $1, %edi
23   movl  %edi, -20(%rbp)
24   movl  -8(%rbp), %eax
25   popq  %rbp
26   ret
27 .Ltmp5:
28   .size main, .Ltmp5-main
29   .cfi_endproc
30
31
32   .ident  "FreeBSD clang version 3.4.1 (
       tags/RELEASE_34/dot1-final 208032)
       20140512"
33   .section  ".note.GNU-stack","",
       @progbits
```


Příklad kompilace do objektového souboru

- Pouze kompilace (nikoliv linkování) je specifikována přepínačem (příznakem) `-c` (compile).
  `clang -c var.c -o var.o`

```
% clang -c var.c -o var.o
% file var.o
var.o: ELF 64-bit LSB relocatable, x86-64, version 1 (FreeBSD), not
    stripped
```

Příkaz `file` identifikuje soubor.

- **Linkování** do spustitelného souboru.
  `clang var.o -o var`

```
% clang var.o -o var
% file var
var: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),
    dynamically linked (uses shared libs), for FreeBSD 10.1 (1001504),
    not stripped
```

dynamically linked
not stripped

Příklad spustitelného souboru v OS 1/2

- Ve výchozím nastavení jsou spustitelné soubory závislé na knihovně C a volání služeb OS.
- Závislosti na knihovnách lze zobrazit např. programem `ldd`.

ldd – list dynamic object dependencies

```
ldd var
var:
        libc.so.7 => /lib/libc.so.7 (0x2c41d000)
```

- Statické linkování lze povolit přepínačem `-static`.

```
clang -static var.o -o var
% ldd var
% file var
var: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),
    statically linked, for FreeBSD 10.1 (1001504), not stripped
% ldd var
ldd: var: not a dynamic ELF executable
```

Jaká je výsledná velikost binárních souborů?

Příklad spustitelného souboru v OS 2/2

- Zkompilovaný program (objektový soubor) obsahuje symbolická jména (ve výchozím nastavení).

Použitelná například pro ladění.

```
clang var.c -o var
wc -c var
    7240 var
```

wc – word, line, character, and byte count

-c – byte count

- Symboly lze odstranit programem `strip`.

```
strip var
wc -c var
    4888 var
```

Případně můžete velikost souboru zobrazit příkazem `ls -l`.

Příklad součtu hodnot dvou proměnných

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     int var1;
6     int var2 = 10; /* inicializace hodnoty promenne */
7     int sum;
8
9     var1 = 13;
10
11    sum = var1 + var2;
12
13    printf("The sum of %i and %i is %i\n", var1, var2, sum);
14
15    return 0;
16 }
```

- Proměnné `var1`, `var2` a `sum` reprezentují tři různá místa v paměti (automaticky přidělené), ve kterých jsou uloženy tři celočíselné hodnoty.

Základní číselné typy

- Celočíselné typy – `int`, `long`, `short`, `char`.

`char` – celé číslo v rozsahu jednoho bajtu nebo také znak.

  - Velikost paměti alokované příslušnou (celo)číselnou proměnnou se může lišit dle architektury počítače nebo překladače.

Typ `int` má zpravidla velikost 4 bajty a to i na 64-bitových systémech.

  - Aktuální velikost paměťové reprezentace lze zjistit operátorem **sizeof()**, kde argumentem je jméno typu nebo proměnné.

```
int i;
printf("%lu\n", sizeof(int));
printf("ui size: %lu\n", sizeof(i));
```

lec02/types.c

- Neceločíselné typy – `float`, `double`

Konretní reprezentace je dána implementací, většinou dle standardu IEEE-754-1985.

  - **float** – 32-bit IEEE 754
  - **double** – 64-bit IEEE 754

http://www.tutorialspoint.com/cprogramming/c_data_types.htm

Znaménkové a neznaménkové celočíselné typy

- Celočíselné typy kromě počtu bajtů rozlišujeme na
  - **signed** – **znaménkový** (základní);
  - **unsigned** – **neznaménkový**.

Proměnná neznaménkového typu nemůže zobrazit záporné číslo.

  - Příklad (1 byte):
    `unsigned char`: 0 až 255;
    `signed char`: -128 až 127.

```
1 unsigned char uc = 127;
2 char su = 127;
3
4 printf("The value of uc=%i and su=%i\n", uc, su);
5 uc = uc + 2;
6 su = su + 2;
7 printf("The value of uc=%i and su=%i\n", uc, su);
```

lec02/signed_unsigned_char.c

Znak – char

- Znak je typ `char`.
- Znak reprezentuje celé číslo (byte).

Kódování znaků (grafických symbolů), např. ASCII – American Standard Code for Information Interchange.

- Hodnotu znaku lze zapsat jako tzv. znakovou konstantu, např. `'a'`.

```
1 char c = 'a';
2
3 printf("The value is %i or as char '%c'\n", c, c);
```

lec02/char.c

```
clang char.c && ./a.out
The value is 97 or as char 'a'
```

- Pro řízení výstupních zařízení jsou definovány řídicí znaky.

*Tzv. **escape sequences***

  - \t – tabulátor (tabular), \n – nový řádek (newline),
  - \a – pípnutí (beep), \b – backspace, \r – carriage return,
  - \f – form feed, \v – vertical space

Logický datový typ (Boolean) – _Bool

- Ve verzi **C99** je zaveden logický datový typ `_Bool`.

```
_Bool logic_variable;
```

- Jako hodnota *true* je libovolná hodnota typu `int` různá od 0.
- Dále můžeme využít hlavičkového souboru `<stdbool.h>`, kde je definován typ **bool** a hodnoty `true` a `false`.

```
#define false 0
#define true  1

#define bool  _Bool
```

- V původním (ANSI) C explicitní datový typ pro logickou hodnotu není definován.
  - Můžeme však použít podobnou definici jako v `<stdbool.h>`.

```
#define FALSE 0
#define TRUE 1
```


Rozsahy celočíselných typů

- Rozsahy celočíselných typů v C nejsou dány normou, ale implementací.

Mohou se lišit implementací a prostředím 16 bitů vs 64 bitů.

- Norma garantuje, že pro rozsahy typů platí.
  - `short` ≤ `int` ≤ `long`
  - `unsigned short` ≤ `unsigned` ≤ `unsigned long`
- Pokud chceme zajistit definovanou velikost můžeme použít definované typy například v hlavičkovém souboru `<stdint.h>`.

IEEE Std 1003.1-2001

`int8_t`	`uint8_t`
`int16_t`	`uint16_t`
`int32_t`	`uint32_t`

lec02/inttypes.c

http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html

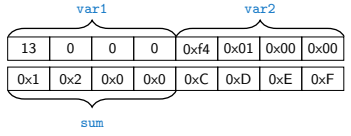
Přiřazení, proměnné a paměť – Vizualizace int

```
1  int var1;
2  int var2;
3  int sum;
4
5  // 00 00 00 13
6  var1 = 13;
7
8  // x00 x00 x01 xF4
9  var2 = 500;
10
11 sum = var1 + var2;
```

- Proměnné typu `int` alokují 4 bajty.

Zjistit velikost můžeme operátorem `sizeof(int)`.

- Obsah paměti není po alokaci definován.



500 (dec) je 0x01F4 (hex)
513 (dec) je 0x0201 (hex)

*V případě architektury Intel x86 a x86-64 jsou hodnoty uloženy v pořadí **little-endian**.*

Zápis hodnot datových typů

- Hodnoty datových typů označujeme jako **literály**.
- Zápis celých čísel (celočíselné literály).
  - dekadický 123 450932
  - šestnáctkový (hexadecimální) 0x12 0xFAFF (začíná `0x` nebo `0X`)
  - osmičkový (oktalový) 0123 0567 (začíná `0`)
  - unsigned 12345U (přípona `U` nebo `u`)
  - long 12345L (přípona `L` nebo `l`)
  - unsigned long 12345ul (přípona `UL` nebo `ul`)
- Není-li přípona uvedena, jde o literál typu `int`. Výchozí typ Cčka.
- Neceločíselné datové typy jsou dané implementací, většinou se řídí standardem IEEE-754-1985. `float`, `double`

Literály

- Jazyk C má 6 typů literálů (konstantních hodnot)
  - Celočíselné;
  - Racionální;
  - Znakové;
  - Řetězcové;
  - Výčtové – *pojmenovaná celá čísla typu* `int`; *Enum*
  - Symbolické – `#define NUMBER 10`. *Preprocesor*

Literály racionálních čísel

- Formát zápisu racionálních literálů:
  - S řádovou tečkou – `13.1`;
  - Mantisa a exponent – `31.4e-3` nebo `31.4E-3`.
- Typ racionálního literálu:
  - `double` – pokud není explicitně určen;
  - `float` – přípona `F` nebo `f`;

```
float f = 10f;
```

  - `long double` – přípona `L` nebo `l`.

```
long double ld = 101;
```


Znakové literály

- Formát – jeden (případně více) znaků v jednoduchých apostrofech
  `'A'`, `'B'` nebo `'\n'`.
- Hodnota – jednoznakový literál má hodnotu odpovídající kódu znaku
  `'0'`∼ 48, `'A'`∼ 65.
  Hodnota znaků mimo ASCII (větší než 127) závisí na překladači.
- Typ znakové konstanty
  - **znaková konstanta je typu** `int`.

Řetězcové literály

- Formát – posloupnost znaků a řídicích znaků (escape sequences) uzavřená v uvozovkách.
  `"Řetězcová konstanta s koncem řádku\n"`
  - Řetězcové konstanty oddělené oddělovači (white spaces) se sloučí do jediné, např.
    `"Řetězcová konstanta"  " s koncem řádku\n"`
    se sloučí do
    `"Řetězcová konstanta s koncem řádku\n"`.
- Typ
  - Řetězcová konstanta je uložena v poli typu `char` a zakončená znakem `'\0'`.
    Např. řetězcová konstanta `"word"` je uložena jako poslopnost znaků/bajtů (pole).

'w'	'o'	'r'	'd'	'\0'

Pole tak musí být vždy o 1 položku delší!
Více o textových řetězcích na 4. přednášce a cvičení.

Konstanty výčtového typu

- Formát
  - Implicitní hodnoty konstanty výčtového typu začínají od 0 a každý další prvek má hodnotu o jedničku vyšší.
  - Hodnoty můžeme explicitně předepsat.

```
enum {
   SPADES,
   CLUBS,
   HEARTS,
   DIAMONDS
};
```

```
enum {
   SPADES = 10,
   CLUBS, /* the value is 11 */
   HEARTS = 15,
   DIAMONDS = 13
};
```

Hodnoty výčtu zpravidla píšeme velkými písmeny.

- Typ – výčtová konstanta je typu `int`.
  - Hodnotu konstanty můžeme použít pro iteraci v cyklu.

```
enum { SPADES = 0, CLUBS, HEARTS, DIAMONDS, NUM_COLORS };
for (int i = SPADES; i < NUM_COLORS; ++i) {
   ...
}
```


Symbolické konstanty – `#define`

- Formát – konstanta je založena příkazem preprocesoru `#define`.
  - Je to makro příkaz bez parametru.
  - Každý `#define` musí být na samostatném řádku.

```
#define SCORE 1
```

Zpravidla píšeme velkými písmeny.

- Symbolické konstanty mohou vyjadřovat konstantní výraz.

```
#define MAX_1   ((10*6) - 3)
```

- Symbolické konstanty mohu být vnořené.

```
#define MAX_2   (MAX_1 + 1)
```

- **Preprocesor provede textovou náhradu definované konstanty za její hodnotu**.

```
#define MAX_2   (MAX_1 + 1)
```

Je-li hodnota výraz, jsou kulaté závorky nutné pro správné vyhodnocení výrazu, např. pro `5*MAX_1` *s vnějšími závorkami je 5*((10*6) - 3)=285 vs 5*(10*6) - 3=297.*

Proměnné s konstantní hodnotou – modifikátor (`const`)

- Uvedením klíčového slova **const** můžeme označit proměnnou jako konstantní.
  Překladač kontroluje přiřazení a nedovolí hodnotu proměnné nastavit znovu.
- Pro definici konstant můžeme použít např.
- Proměnné s konstatní hodnotou mají typ a paměť

```
const float pi = 3.14159265;
```

- na rozdíl od symbolické konstanty

```
#define PI 3.14159265
```

- reprezentující literál.

Výrazy

- **Výraz** předepisuje výpočet hodnoty určitého vstupu.
- Struktura výrazu obsahuje **operandy**, **operátory** a **závorky**.
- Výraz může obsahovat
  - literály,
  - proměnné,
  - konstanty,
  - unární a binární operátory,
  - volání funkcí,
  - závorky.
- Pořadí operací předepsaných výrazem je dáno **prioritou** a **asociativitou** operátorů.

Příklad

```
10 + x * y      // pořadí vyhodnocení 10 + (x * y)
10 + x + y      // pořadí vyhodnocení (10 + x) + y
```

** má vyšší prioritu než +*
+ je asociativní zleva

Základní rozdělení operátorů

- Operátory jsou vyhrazené znaky (nebo posloupnost znaků) pro zápis výrazů.
- Můžeme rozlišit čtyři základní typy **binárních operátorů**.
  - **Aritmetické** operátory – sčítání, odčítaní, násobení, dělení;
  - **Relační** operátory – porovnání hodnot (menší, větší, ...);
  - **Logické** operátory – logický součet a součin;
  - Operátor přiřazení - na levé straně operátoru = je proměnná.
- **Unární operátory**
  - indikující kladnou/zápornou hodnotu: + a −;
    Unární operátor minus − modifikuje znaménko výrazu za ním.
  - modifikující proměnou ++ a −−;
  - logický operátor doplněk !;
  - operátor přetypování (jméno typu).
- **Ternární operátor** – podmíněný výsledek výrazu ze dvou výrazů.

Proměnné, operátor přiřazení a příkaz přiřazení

- Proměnné definujeme uvedením typu a jména proměnné.
  - Jména proměnných volíme malá písmena.
  - Víceslovná jména zapisujeme s podtržítkem _.
    Nebo volíme CamelCase
  - Proměnné definujeme na samostatném řádku.

```
int n;
int number_of_items;
```

  - **Proměnné reprezentují data, proto zpravidla volíme podstatná jména.**
- Přiřazení je nastavení hodnoty proměnné, tj. uložení definované hodnoty na místo v paměti, kterou proměnná reprezentuje.
- Tvar přiřazovacího operátoru.

$$\langle \textbf{proměnná} \rangle = \langle \textbf{výraz} \rangle$$

Výraz je literál, proměnná, volání funkce, ...

- Příkaz přiřazení se skládá z operátoru přiřazení = a ;.
  - Levá strana přiřazení musí být **l-value – location-value, left-value**.
    Tj. musí reprezentovat paměťové místo pro uložení výsledku.
  - Přiřazení je výraz a můžeme jej použít všude, kde je dovolen výraz příslušného typu.

Základní aritmetické výrazy

- Pro operandy (ne)celočíselných typů `int`, `char`, `short` a `double` a `float` jsou definovány operátory:
  - unární operátor změna znaménka −;
  - binární sčítání + a odčítání −;
  - binární násobení * a dělení /.
- Pro operandy celočíselných typů pak dále
  - binární zbytek po dělení %.
- Pro oba operandy stejného typu je výsledek aritmetické operace stejného typu.
- V případě kombinace typů **int** a **double**, se `int` převede na **double** a výsledek je hodnota typu **double**.
  Implicitní typová konverze.
- Dělení operandů typu **int** je celá část podílu.
  Např. 7/3 je 2 a −7/3 je −2
- Pro zbytek po dělení platí $x\%y = x - (x/y) * y$.
  Např. 7 % 3 je 1 -7 % 3 je -1 7 % -3 je 1 -7 % -3 je -1
  *Pro záporné operandy je v C99 výsledek celočíselného dělení blíže 0, platí (a/b)*b + a%b = a.*
  Pro starší verze C závisí výsledek na překladači.

Další operátory příště.

Příklad – Aritmetické operátory 1/2

```
1  int a = 10;
2  int b = 3;
3  int c = 4;
4  int d = 5;
5  int result;
6
7  result = a - b; // rozdil
8  printf("a - b = %i\n", result);
9
10 result = a * b; // nasobeni
11 printf("a * b = %i\n", result);
12
13 result = a / b; // celociselne deleni
14 printf("a / b = %i\n", result);
15
16 result = a + b * c; // priorita operatoru
17 printf("a + b * c = %i\n", result);
18
19 printf("a * b + c * d = %i\n", a * b + c * d);         // -> 50
20 printf("(a * b) + (c * d) = %i\n", (a * b) + (c * d)); // -> 50
21 printf("a * (b + c) * d = %i\n", a * (b + c) * d);     // -> 350
```

lec02/arithmetic_operators.c

Příklad – Aritmetické operátory 2/2

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     int x1 = 1;
6     double y1 = 2.2357;
7     float x2 = 2.5343f;
8     double y2 = 2;
9
10    printf("P1 = (%i, %f)\n", x1, y1);
11    printf("P1 = (%i, %i)\n", x1, (int)y1);
12    printf("P1 = (%f, %f)\n", (double)x1, (double)y1); // operator pretypovani
       (double)
13    printf("P1 = (%.3f, %.3f)\n", (double)x1, (double)y1);
14
15    printf("P2 = (%f, %f)\n", x2, y2);
16
17    double dx = (x1 - x2); // implicitni konverze na float, resp.  double
18    double dy = (y1 - y2);
19
20    printf("(P1 - P2)=(%.3f, %0.3f)\n", dx, dy);
21    printf("|P1 - P2|^2=%.2f\n", dx * dx + dy * dy);
22    return 0;
23 }
```

lec02/points.c

Část II

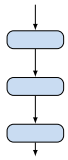
Část 2 – Řídicí struktury

Řídicí struktury

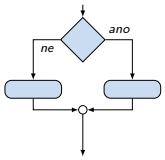
- Řídicí struktura je programová konstrukce, která se skládá z dílčích příkazů a předepisuje pro ně způsob provedení.
- Tři základní druhy řídicích struktur:
  - **Posloupnost** – předepisuje postupné provedení dílčích příkazů;
  - **Větvení** – předepisuje provedení dílčích příkazů v závislosti na splnění určité podmínky;
  - **Cyklus** – předepisuje opakované provedení dílčích příkazů v závislosti na splnění určité podmínky.

Typy řídicích struktur 1/2

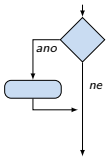
- **Sekvence**



- Podmínka **If**

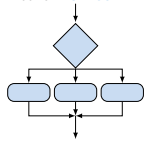


- Podmínka **If**

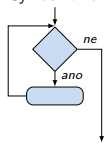


Typy řídicích struktur 2/2

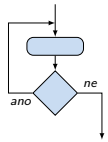
- Větvení **switch**



- Cyklus **for** a **while**



- Cyklus **do**



Složený příkaz a blok

- Řídicí struktury mají obvykle formu strukturovaných příkazů.
  - **Složený příkaz** – posloupnost příkazů.
  - **Blok** – posloupnost definic proměnných a příkazů.

```
{
   //blok je vymezen složenými závorkami
   int steps = 10;

   printf("No. of steps %i\n", steps);
}

steps += 1; //nelze - mimo rozsah platnosti bloku
```

Definice – alokace paměti podle konkrétního typu proměnné. Rozsah platnosti proměnné je lokální v rámci bloku.

- Budeme používat složené příkazy:
  - složený příkaz nebo blok pro posloupnost;
  - příkaz **if** nebo **switch** pro větvení;
  - příkaz **while**, **do** nebo **for** pro cyklus.

Podmíněné opakování bloku nebo složeného příkazu.

- **Funkce** je **pojmenovaný blok příkazů**, který můžeme **znovupoužít**.

Větvení **if**

- Příkaz **if** umožňuje větvení programu na základě podmínky.
- Má dva základní tvary.
  - **if** (*podmínka*) `příkaz`$_1$
  - **if** (*podmínka*) `příkaz`$_1$ **else** `příkaz`$_2$
- `podmínka` je logický výraz, jehož hodnota je logického (celočíselného) typu.

Tj. false (hodnota 0) nebo true (hodnota různá od 0).

- `příkaz` je příkaz, složený příkaz nebo blok.

Příkaz je zakončen středníkem ;

- Ukázka zápisu zjištění menší hodnoty z x a y.

Varianta zápisu 1

```
int min = y;
if (x < y) min = x;
```

Varianta zápisu 2

```
int min = y;
if (x < y)
   min = x;
```

Varianta zápisu 3

```
int min = y;
if (x < y) {
   min = x;
}
```

Která varianta splňuje kódovací konvenci a proč?

Příklad větvení **if**

Příklad: Jestliže $x < y$ vyměňte hodnoty těchto proměnných

Nechť proměnné x a y jsou definovány a jsou typu `int`.

Varianta 1

```
if (x < y)
   tmp = x;
   x = y;
   y = tmp;
```

Varianta 2

```
if (x < y)
   int tmp = x;
   x = y;
   y = tmp;
```

Varianta 3

```
int tmp;
if (x < y)
   tmp = x;
   x = y;
   y = tmp;
```

Varianta 4

```
if (x < y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

- Která varianta je správně a proč?

Příklad větvení **if**-then-**else**

Příklad: Do proměnné *min* uložte menší z čísel x a y a do *max* uložte větší z čísel.

Nechť proměnné `x`, `y`, `min` a `max` jsou definovány a jsou typu `int`.

Varianta 1

```
if (x < y)
   min = x;
   max = y;
else
   min = y;
   max = x;
```

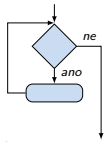
Varianta 2

```
if (x < y) {
   min = x;
   max = y;
} else {
   min = y;
   max = x;
}
```

- Která varianta odpovídá našemu zadání?

Cyklus `while ()`

- Příkaz **while** má tvar `while (vyraz) prikaz;`
- Příkaz cyklu **while** probíhá:
  1. Vyhodnotí se výraz `vyraz`;
  2. Pokud `vyraz != 0`, provede se příkaz `prikaz`, jinak cyklus končí;
  3. Opakování vyhodnocení výrazu `vyraz`.
- Řídicí cyklus se vyhodnocuje na začátku cyklu, cyklus se nemusí provést ani jednou.
- Řídicí výraz `vyraz` se musí aktualizovat v těle cyklu, jinak je cyklus nekonečný.



Příklad zápisu

```
int i = 0;
while (i < 5) {
   ...
   i += 1;
}
```


Příklad cyklu `while`

- Základní příkaz cyklu **while** má tvar
  while (*podmínka*) `příkaz`.

Příklad

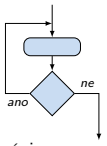
```
int x = 10;
int y = 3;
int q = x;

while (q >= y) {
   q = q - y;
}
```

- Jaká je hodnota proměnné q po skončení cyklu?

Cyklus `do...while ()`

- Příkaz `do...while ()` má tvar
  `do prikaz while (vyraz);`
- Příkaz cyklu `do...while ()` probíhá
  1. Provede se příkaz `prikaz`;
  2. Vyhodnotí se výraz `vyraz`;
  3. Pokud `vyraz != 0`, cyklus se opakuje provedením příkazu `prikaz`, jinak cyklus končí.
- Řídicí cyklus se vyhodnocuje na konci cyklu, tělo cyklu se vždy provede nejméně jednou.
- Řídicí výraz `vyraz` se musí aktualizovat v těle cyklu, jinak je cyklus nekonečný.



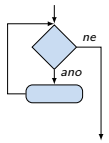
Příklad zápisu

```
int i = -1;
do {
   ...
   i += 1;
} while (i < 5);
```


Cyklus **for**

- Základní příkaz cyklu **for** má tvar **for** (*inicializace*; *podmínka*; *změna*) `příkaz`.
- Odpovídá cyklu while v následujícím tvaru.

inicializace;
while (*podmínka*) {
    `příkaz`;
    změna;
}



Příklad

```
for (int i = 0; i < 10; ++i) {
   printf("i: %i\n", i);
}
```

- Změnu řídicí proměnné lze zkráceně zapsat operátorem inkrementace $++$ nebo dekrementace $--$.
- Alternativně lze též použít zkrácený zápis přiřazení, např. $+=$.

Cyklus **for** – příklady

- Jak se změní výstup když použijeme místo prefixového zápisu $++i$ postfixový zápis $i++$.

```
for (int i = 0; i < 10; i++) {
   printf("i: %i\n", i);
}
```

- V cyklu můžeme také řídicí proměnou dekrementovat.

```
for (int i = 10; i >= 0; --i) {
   printf("i: %i\n", i);
}
```

Kolik program vypíše řádků?

- Kolik řádků vypíše program?

```
for (int i = 10; i > 0; --i) {
   printf("i: %i\n", i);
}
```

- Řídicí proměnná může být také nečeločíselného typu, např. **double**.

```
#include <math.h>

for (double d = 0.5; d < M_PI; d += 0.1) {
   printf("d: %f\n", d);
}
```

Část III

Část 3 – Zadání 2. domácího úkolu (HW02)

Zadání 2. domácího úkolu HW02

Téma: **První cyklus**

Povinné zadání: **2b**; Volitelné zadání: není; Bonusové zadání: není

- **Motivace**: „Automatizovat“ a zobecnit výpočet pro „libovolně“ dlouhý vstup.
- **Cíl**: Osvojit si využití cyklů jako základní programového konstrukce pro hromadné zpracování dat.
- **Zadání**: `https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw02`
  - Zpracování **libovolně dlouhé** posloupnosti celých čísel.
  - Výpis načtených čísel.
  - Výpis statistiky vstupních čísel.
    - Počet načtených čísel; Počet kladný a záporných čísel a jejich procentuální zastoupení na vstupu.
    - Četnosti výskytu sudých a lichých čísel a jejich procentuální zastoupení na vstupu.
    - Průměrná, maximální a minimální hodnota načtených čísel.
- Termín odevzdání: 15.10.2022, 23:59:59 PDT.

PDT – Pacific Daylight Time

Shrnutí přednášky

Diskutovaná témata

- Základy programování v C
  - Zápis programu v C
  - Program, zdrojové soubory a kompilace programu
  - Literály a konstantní hodnoty
  - Proměnné, základní číselné typy
  - Proměnné, přiřazení a paměť
  - Základní výrazy
  - Řídicí struktury
- Příště: Dokončení řídicích struktur, výrazy.