Programování (v C)

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 02

BAB36PRGA – Programování v C

Přehled témat

Část I

Část 1 – Programování v C

Zdrojové soubory programu v C

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

Zpravidla—základní rozlišení souborů, viz např. .C.

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.

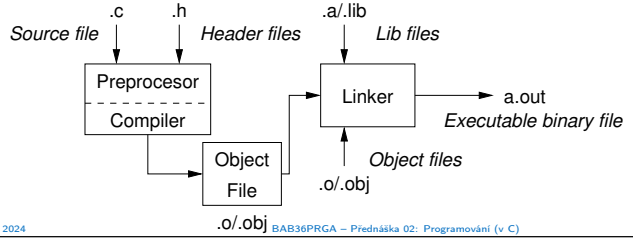
Program se zpravidla sestavuje z více objektových souborů umístěných například v knihovnách.

Schéma překladu a sestavení programu

- Vývoj programu se skládá z editace zdrojových souborů (.c a .h). *Lidsky čitelných*
  - Kompilace zdrojových souborů (.c) do objektových (.o nebo .obj). *Strojově čitelných*
    - **Preprocesor** – zpracování maker a přizpůsobení překladu kompilačnímu prostředí.
  - Linkování přeložených (objektových) souborů do spustitelného programu.

Také vytváření dynamicky linkovaných knihoven.

- Spouštění a ladění aplikace a opětovné editace zdrojových souborů.



Překladače jazyka C

- V PRGA používáme `gcc` a `clang` (C language family frontend for LLVM).

https://gcc.gnu.org http://clang.llvm.org

- Příklad použití

Základní použití (přepínače a argumenty) je u obou překladačů stejné.

  - compile: `gcc -c program.c -o program.o`
  - link: `gcc program.o -o program`
  - Sloučení překladu a sestavení v jediném příkazu `clang program.c -o program`
  - Linkování s vložením matematické knihovny `clang program.o -lm -o program`

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

- Informace o souboru (`file`) a závislosti na dynamických knihovnách (`ldd`).

```
% clang var.c -o var
% file var
var: ELF 64-bit LSB executable, x86-64, version 1 (
    FreeBSD), dynamically linked, interpreter /
    libexec/ld-elf.so.1, for FreeBSD 12.4
    (1204500), FreeBSD-style, with debug_info, not
    stripped
% ldd var
var:
        libc.so.7 => /lib/libc.so.7 (0x2c41d000)
```

```
% clang program.c -lm -o program
% ldd program
program:
        libm.so.5 => /lib/libm.so.5 (0x80024c000)
        libc.so.7 => /lib/libc.so.7 (0x800283000)

% clang program.c -lm -static -o program
% ldd program
ldd: program: not a dynamic ELF executable
```


Zdrojové soubory

Proč psát program do více souborů?

- Zdrojové a hlavičkové soubory umožňují rozlišit **deklaraci** a **definici** podporující:
  - **Znovupoužitelnost**
    - K využití binární knihovny potřebuje znát „rozhraní" funkcí (případně typů), které je deklarované v hlavičkovém souboru. *Např. funkce standardní knihovny C, `libc`.*
  - **Modularitu**
    - Hlavičkový soubor obsahuje popis co modul nabízí, tj. popis (seznam) funkcí a jejich parametrů (**deklarace funkcí**) bez konkrétní implementace.

*Implementace funkce je **definice funkce**.*

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

  - **Organizaci** zdrojových kódů v adresářové struktuře souborů.

Pro jednoduché programy a domácí úkoly nedává moc smysl.

Vyplatí se především v HW7, HW8 a HW9, případně HW6 (Matice)!

Příklad kompilace programu z více souborů

```
#ifndef __COMPUTE_H__
#define __COMPUTE_H__

// deklarace funkce (hlavička/prototyp)
int compute(int a);

#endif                          lec02/compute.h
```

```
#include "compute.h"

int compute(int a) // definice funkce
{
   int b = 10 + a; // tělo funkce
   return b;  // návratová hodnota funkce
}                               lec02/compute.c
```

```
#include "compute.h" // vložení deklarace funkce

int main(int argc, char *argv[])
{ // hlavní funkce
   int v = 10; // definice proměnné
   int r = compute(v); // volání funkce
   return 0; // ukončení hlavní funkce
}                               lec02/main-compute.c
```

```
clang -c compute.c
clang -c main-compute.c
clang main-compute.o compute.o -o compute
./compute
```

- Výsledný spustitelný soubor linkujeme s `main-compute.o` a `compute.o`, musí obsah právě jednu funkci `main()`.
- Linkování spustitelné aplikace pouze s `main-compute.o` skončí chybou.

```
% gcc main-compute.o -o compute
/usr/local/bin/ld: main-compute.o: in function 'main':
main-compute.c:(.text+0x21): undefined reference to 'compute'
collect2: error: ld returned 1 exit status
```


Spustitelný program – `main()`

- Spustitelný program musí obsahovat jedinou definici funkce `main()`, která má základní tvary předání argumentů programu.

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

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

- Při spuštění programu předává OS programu počet argumentů (`argc`) a argumenty (`argv`), jako pole textových řetězců. *První argument je jméno programu.*
- Návratová hodnota je předána OS, kde je možné ji dále použít.

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

- Návratová hodnota programu je 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
```

lec02/var.c

Příklad kompilace a spuštění programu

- Sestavení programu **clang var.c** automaticky dojde ke kompilaci a linkování programu do spustitelného souboru `a.out`. *Výchozí jméno programu.*
- Výstupní (output) soubor specifikujeme **clang var.c -o var** a spustíme, např. `./var`.
- Kompilaci a spuštění můžeme spojit do dvojice příkazů **clang var.c -o var; ./var**.
- Spuštění můžeme podmínit úspěšnou kompilací programu **clang var.c -o var && ./var**.

Návratová hodnota programu — 0 (`EXIT_SUCCESS`) znamená OK, chyb může být více.

*Logický operátor **&&** závisí na příkazovém interpretu, např. `sh`, `bash`, `zsh`.*

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

```
# 1 "var.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "var.c"
int main(int argc, char **argv) {
    int v;
    v = 10;
    v = v + 1;
    return argc;
}
```

lec02/var.c

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 (tj. konkrétní jméno souboru .c). **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 zásobník. *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

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 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)
{
   ...
}
```


Argumenty funkce

- Argumenty funkce se předávají hodnotou.

```
int main(void)
{
   int v1 = 10;
   int v2 = 20;

   printf("v1: %i v2: %i\n", v1, v2);
   swap0(v1, v2);
   printf("v1: %i v2: %i\n", v1, v2);
   swap(&v1, &v2); //předání paměťového místa
   printf("v1: %i v2: %i\n", v1, v2);
   return 0;
}
```

```
void swap0(int a, int b)
{
   int t = a; // dočasná proměnná
   a = b;
   b = t;
}

void swap(int *a, int *b)
{
   int t = *a; // dočasná proměnná
   *a = *b;
   *b = t;
}
```

lec02/swap.c

- Proto předáváme adresu paměťového místa (pointer/ukazatel) – `&v1` a `&v2`.

```
% clang swap.c -o swap && ./swap
v1: 10 v2: 20
v1: 10 v2: 20
v1: 20 v2: 10
```


Argumenty funkce a návratová hodnota

- K „vrácení" více hodnot, můžeme využít předání paměťových míst. *Podobně jako `scanf()`.*
- Příklad načtení celých čísel typu `int` a určení minimální a maximální hodnoty.

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

void min_max(int v, int *min, int *max);

int main(void)
{
   int ret = EXIT_SUCCESS;
   int min = INT_MAX; // limits
   int max = INT_MIN; // limits
   int c = 0;
   int v;

   while (scanf("%i", &v) == 1) {
      min_max(v, &min, &max);
      c = c + 1;
   }
   if (c > 0) {
      printf("Read %d numbers, min: %d, max: %d\n", c, min, max);
   } else {
      fprintf(stderr, "ERROR: No input given!\n");
      ret = EXIT_FAILURE;
   }
   return ret;
}

void min_max(int v, int *min, int *max)
{
   if (v < *min) *min = v;
   if (v > *max) *max = v;
}
```

lec02/min_max.c

`min_max()` – příklad volání

- Vytvoříme vstupní soubor s pěti náhodnými čísly od 1 do 99: `shuf -i 1-99 -n 5`.
- Standardní výstup programu `shuf` přesměrujeme do souboru `in.txt`.
- Standardní vstup našeho programu `minmax` přesměrujeme ze souboru `in.txt`.
- Vytiskneme návratovou hodnotu volání programu.

```
% clang min_max.c -o minmax
% shuf -i 1-99 -n 5 > in.txt
% ./minmax <in.txt
Read 5 numbers, min: 1, max: 9
% echo $?
0
```

lec02/min_max.c

- Vytvoříme alternativní (chybný) vstup, nebo zadáme ručně.

```
% echo "a" >in2.txt
% lec02 cat in.txt >>in2.txt
% ./minmax <in2.txt
ERROR: No input given!
% echo $?
1
```


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`

Konkretní 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.
```

```
unsigned char uc = 127;
char su = 127;

printf("The value of uc=%i and su=%i\n", uc, su);
uc = uc + 2;
su = su + 2;
printf("The value of uc=%i and su=%i\n", uc, su);
```

lec02/signed_unsigned_char.c

```
$ clang signed_unsigned.c && ./a.out
The value of uc=127 and su=127
The value of uc=129 and su=-127
```

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 z hlavičkového 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

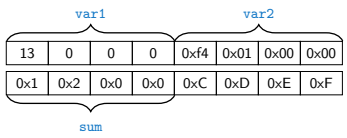
Přiřazení, proměnné a paměť – Vizualizace int

```
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**.*

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`.

Automatická konverze kódu ASCII znaku na typ `char`.

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 posloupnost 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 konstatní proměnné, symbolické konstanty (preprocesor) a v případě celočíselných hodnot (`int`) také `enum`.
- 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.

$$\texttt{výraz ? hodnota}_1 \texttt{ : hodnota}_2$$

Hodnota výrazu ternárního operátoru je buď druhý nebo třetí operand v závislosti na logické hodnotě prvního operandu.

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 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

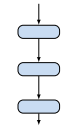
Část 2 – Řídicí struktury

Řídicí struktury

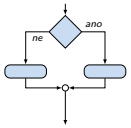
- Řídicí struktura je programová konstrukce, která se skládá z dílčích příkazů a předepisuje pro ně způsob provedení.
- Tři základní druhy řídicích struktur:
  - **Posloupnost** – předepisuje postupné provedení dílčích příkazů;
  - **Větvení** – předepisuje provedení dílčích příkazů v závislosti na splnění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

Typy řídicích struktur

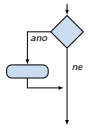
- **Sekvence**



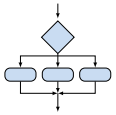
- Podmínka **If**



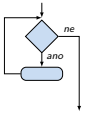
- Podmínka **If**



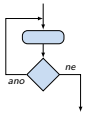
- Větvení **switch**



- 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;
```

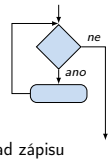
Varianta 2

```
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

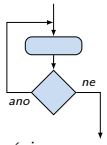
```
int x = 10;
int y = 3;
int q = x;

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.



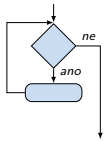
Příklad zápisu

```
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 neceločíselného 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 1. a 2. domácího úkolu (HW1 a HW2)

Zadání 1. domácího úkolu HW1

Téma: **Načítání vstupu**

Povinné zadání: 3b; 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/bab36prga/hw/hw1
  - 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: 16.03.2024, 23:59:59 PDT.

PDT – Pacific Daylight Time

Zadání 2. domácího úkolu HW2

Téma: **Kreslení (ASCII art)**

Povinné zadání: 3b; Volitelné zadání: není; Bonusové zadání: není

- **Motivace**: Zábavným a tvůrčím způsobem získat praktickou zkušenost s cykly a jejich parametrizací na základě uživatelského vstupu.
- **Cíl**: Osvojit si použití cyklů a vnořených cyklů.
- **Zadání**: https://cw.fel.cvut.cz/wiki/courses/bab36prga/hw/hw2
  - Načtení parametrizace pro vykreslení šroubovice s využitím vybraných ASCII znaků.
    https://en.wikipedia.org/wiki/ASCII_art
  - Ošetření vstupních hodnot.
- Termín odevzdání: 23.03.2024, 23:59:59 PDT.

PDT – Pacific Daylight Time

Shrnutí přednášky

Diskutovaná témata

- 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.

Část V

Appendix

Kódovací příklad – Zadání

- Implementujte program, který vytiskne vzor o sedmi řádcích.
- Výchozí šířka n je 27 znaků nebo je načtena jako první argument programu (je-li zadán).
- Šířka n musí být liché číslo, jinak program vrátí **100**.
- Platí $11 \leq n \leq 67$, jinak program vrátí **101**.
- Při úspěchu program vytiskne sedm řádků a vrátí **0** (EXIT_SUCCESS).
- Snažte se maximálně vyhnout použití "magic numbers" v programu.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * *
** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** **
*** *** *** *** *** *** *** *** *** *** *** *** *** ***
*****************************************************
*** *** *** *** *** *** *** *** *** *** *** *** *** ***
** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** **
* * * * * * * * * * * * * * * * * * * * * * * * * * *
```

- Argument programu argv[1] převeďte na číslo atoi(), jeli zadán.
- Dekomponujte program jako tisk 7× řádků.
- Implementujte „infrastrukturu“ programu.
- Následně řešte logiku jednotlivých řádků řízených **vhodně navrženým výrazem**.

Příklad kódování – Strategie implementace 1/4

- Definujeme návratové (chybové) hodnoty (0, 100, 101) využitím `enum`, aby byl „kód čistý“.
- Definujeme platný rozsah $\langle 11, 67 \rangle$, (`#define`).
- Zajistíme přístup k argumentům programu pouze tehdy, pokud jsou zadány.
- Kontrolujeme, že počet řádků *n* je platná hodnota, jinak program vrací chybu.
- Provádíme libovolnou operaci pouze v případě, že argumenty (hodnoty) jsou platné.
- Tisk 7-mi řádků rozdělíme do dvou `for` smyček, mezi smyčkami bude tisk plného `*` řádku.
- Implementujeme samostatnou funkci tisk vzoru řádku.

```
#include <stdio.h> //for putchar()
#include <stdlib.h> //for atoi()

enum {
   ERROR_OK = 0,
   ERROR_INPUT = 100,
   ERROR_RANGE = 101
};

#define MIN_VALUE 11
#define MAX_VALUE 67

#define LINES 3

// Print line of the with n using character
    in c and space; with k continuous
    characters c followed by space.
void print(char c, int n, int k);
```


Příklad kódování – Strategie implementace 2/4

- Definujeme návratové (chybové) hodnoty (0, 100, 101) využitím `enum`, aby byl „kód čistý“.
- Definujeme platný rozsah $\langle 11, 67 \rangle$, (`#define`).
- Zajistíme přístup k argumentům programu pouze tehdy, pokud jsou zadány.
- Kontrolujeme, že počet řádků *n* je platná hodnota, jinak program vrací chybu.
- Provádíme libovolnou operaci pouze v případě, že argumenty (hodnoty) jsou platné.
- Tisk 7-mi řádků rozdělíme do dvou `for` smyček, mezi smyčkami bude tisk plného `*` řádku.
- Implementujeme samostatnou funkci tisk vzoru řádku.

```
...
int main(int argc, char *argv[])
{
   int ret = ERROR_OK;
   int n = argc > 1 ? atoi(argv[1]) : 27; //
    convert argv[1] or use default value

   ret = n % 2 == 0 ? ERROR_INPUT : ret; //
    ensure n is odd number
   if (!ret &&
         (n < MIN_VALUE || n > MAX_VALUE)) {
      ret = ERROR_RANGE; //ensure n is in the
    closed interval [MIN_VALUE, MAX_VALUE]
   }
   ...
   return ret;
}
```


Příklad kódování – Strategie implementace 3/4

- Definujeme návratové (chybové) hodnoty (0, 100, 101) využitím `enum`, aby byl „kód čistý“.
- Definujeme platný rozsah $\langle 11, 67 \rangle$, (`#define`).
- Zajistíme přístup k argumentům programu pouze tehdy, pokud jsou zadány.
- Kontrolujeme, že počet řádků *n* je platná hodnota, jinak program vrací chybu.
- Provádíme libovolnou operaci pouze v případě, že argumenty (hodnoty) jsou platné.
- Tisk 7-mi řádků rozdělíme do dvou `for` smyček, mezi smyčkami bude tisk plného `*` řádku.
- Implementujeme samostatnou funkci tisk vzoru řádku.

```
// print a line with n characters with the
    pattern:  k-times c, then space.
// the line ends by new line character '\n'.
void print(char c, int n, int k);

int main(int argc, char *argv[])
{ ...
   if (!ret) { // only if ret == ERROR_OK
      for (int l = 1; l <= LINES; ++l) {
         print('*', n, l); // print l x '*'
      }
      print('*', n, n); // print n x '*'
      for (int l = LINES; l > 0 ; --l) {
         print('*', n, l); // print l x 'x'
      }
   }
   return ret;
}
```


Příklad kódování – Strategie implementace 4/4

- Definujeme návratové (chybové) hodnoty (0, 100, 101) využitím `enum`, aby byl „kód čistý“.
- Definujeme platný rozsah $\langle 11, 67 \rangle$, (`#define`).
- Zajistíme přístup k argumentům programu pouze tehdy, pokud jsou zadány.
- Kontrolujeme, že počet řádků *n* je platná hodnota, jinak program vrací chybu.
- Provádíme libovolnou operaci pouze v případě, že argumenty (hodnoty) jsou platné.
- Tisk 7-mi řádků rozdělíme do dvou `for` smyček, mezi smyčkami bude tisk plného `*` řádku.
- Implementujeme samostatnou funkci tisk vzoru řádku.

```
void print(char c, int n, int k)
{
   for (int i = 0; i < n; ++i) {
      putchar( (i+1) % (k+1) ? c : ' ');
   }
   putchar('\n');
}
```

- Řádek se skládá z `n` znaků, takže je třeba vypsat *n* znaků.
- Za každým `k`-tým znakem `c` je mezera.
- Násobek `k` lze zjistit ze zbytku po celočíselném dělení, operátor `%`.
- Ošetříme, že `i` začíná od 0.
- Mezera je každý (`k+1`)-tý znak.

Příklad kódování – Strategie implementace 4(b)/4

- Definujeme návratové (chybové) hodnoty (0, 100, 101) využitím `enum`, aby byl „kód čistý“.
- Definujeme platný rozsah $\langle 11, 67 \rangle$, (`#define`).
- Zajistíme přístup k argumentům programu pouze tehdy, pokud jsou zadány.
- Kontrolujeme, že počet řádků *n* je platná hodnota, jinak program vrací chybu.
- Provádíme libovolnou operaci pouze v případě, že argumenty (hodnoty) jsou platné.
- Tisk 7-mi řádků rozdělíme do dvou `for` smyček, mezi smyčkami bude tisk plného `*` řádku.
- Implementujeme samostatnou funkci tisk vzoru řádku.

```
void print(char c, int n, int k)
{
   int i, j;
   for (i = j = 0; i < n; ++i, ++j) {
      if (j == k) {
         putchar(' ');
         j = 0;
      } else {
         putchar(c);
      }
   }
   putchar('\n');
}
```

- Použijime extra proměnnou `j` pro tisk mezery, jako každý `k`-tý vytištěný znak.
- Využijeme operátor čárky k inkrementaci `j` v rámci smyčky **for**.