Jazyk C

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 8

A0B36PR2 – Programování 2

Část 1 – Jazyk C - syntax, proměnné, typy a řídicí struktury

Funkce

Proměnné, základní typy a literály

Výrazy a operátory

Příkazy a řízení běhu programu

Část 2 – Jazyk C - ukazatele, řetězce, funkce a volání funkcí

Pole

Ukazatele

Funkce a předávání parametrů

Ukazatele a pole

Část I

Část 1 – Jazyk C - syntax, proměnné, typy a řídicí struktury

Funkce

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

  Modulární–program je složen z více modulů/zdrojových souborů

- Každý spustitelný program v C obsahuje funkci `main()`
- Běh programu začíná na začátku funkce `main()`
- **Definice** funkce obsahuje **hlavičku funkce a její tělo**, syntax:

```
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.
- **Deklarace** se skládá z hlavičky funkce

  Odpovídá rozhraní v Javě

- Parametry se do funkce předávají **hodnotou** (call by value)

  *Parametrem může být i ukazatel (pointer), který pak dovoluje předávat parametry **odkazem**.*

Vlastnosti funkcí

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

Modul–samostatně překládaný soubor

- Funkce jsou implicitně deklarovány jako **extern**
- 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
- **C dovoluje rekurzi** – lokální proměnné jsou pro každé jednotlivé volání zakládány znovu na zásobníku

Kód funkce v C je reentrantní

- Funkce nemusí mít žádné vstupní parametry, zapisujeme:

```
fce(void)
```

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

Struktura programu / modulu

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

Proměnné

- Proměnné představují vymezenou oblast paměti a v C je můžeme rozdělit podle způsobu alokace
  - **Statická** alokace – provede se při deklaraci **statické** nebo globální proměnné. Paměťový prostor je alokován při startu programu a nikdy není uvolněn.
  - **Automatická** alokace – probíhá automaticky v případě lokálních proměnných (nebo argumentů funkce). Paměťový prostor je alokován na **zásobníku** a paměť proměnné je automaticky uvolněna s koncem platnosti proměnné.

    Např. po ukončení bloku funkce.
  - **Dynamická** alokace – není podporována přímo jazykem C, ale je přístupná knihovními funkcemi

    Např. `malloc()` *a* `free()` *z knihovny* `malloc.h`

    http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html

Proměnné – paměťová třída

- Specifikátory paměťové třídy (Storage Class Specifiers – SCS)
  - **auto** (lokální) – Definuje proměnnou jako dočasnou (automatickou). Lze použít pro lokální proměnné deklarované uvnitř funkce. Jedná se o implicitní nastavení, platnost proměnné je omezena na blok. Proměnná je v **zásobníku.**
  - **register** – Doporučuje překladači umístit proměnnou do registru procesoru (rychlost přístupu). Překladač může, ale nemusí vyhovět. Jinak stejné jako **auto**.

    Zpravidla řešíme překladem s optimalizacemi.
  - **static**
    - Uvnitř bloku `{...}` – deklaruje proměnnou jako statickou, která si **ponechává hodnotu i při opuštění bloku**. Existuje po celou dobu chodu programu. Je uložena v **datové oblasti**.
    - Vně bloku – kde je implicitně proměnná uložena v **datové oblasti** (statická) omezuje její viditelnost na modul.
  - **extern** – rozšiřuje viditelnost statických proměnných z modulu na celý program. Globální proměnné s `extern` jsou definované v **datové oblasti**.

Příklad deklarace proměnných

- Hlavičkový soubor vardec.h

```
1  extern int global_variable;
```

lec08/vardec.h

- Zdrojový soubor vardec.c

```
1  #include <stdio.h>
2  #include "vardec.h"
3
4  static int module_variable;
5  int global_variable;
6
7  void function(int p) {
8     int lv = 0; /* local variable */
9     static lsv = 0; /* local static variable */
10    lv += 1;
11    lsv += 1;
12    printf("func: p%d, lv %d, lsv %d\n", p, lv, lsv);
13 }
14 int main(void) {
15    int local;
16    function(1);
17    function(1);
18    function(1);
19    return 0;
20 }
```

- Výstup

```
1  func: p 1, lv 1, slv 1
2  func: p 1, lv 1, slv 2
3  func: p 1, lv 1, slv 3
```

lec08/vardec.c

Základní číselné typy

- Celočíselné typy – `int`, `long`, `short`, `char`
- Neceločíselné typy – `float`, `double`
  - **float** – 32-bit IEEE 754
  - **double** – 64-bit IEEE 754

- Celočíselné typy kromě počtu bajtů rozlišujeme na
  - **signed** – **znaménkový** (základní)

    Jako v Javě
  - **unsigned** – **neznaménkový**

    Proměnná neznaménkového typu nemůže zobrazit záporné číslo
  - Příklad (1 byte):

    `unsigned char`: 0 až 255

    `signed char`: -128 až 127
- Znak je typ `char`

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

Číselné typy a rozsahy

- Velikost paměti alokované příslušnou (celo)číselnou proměnnou se může lišit dle architektury počítače nebo překladače
- Aktuální velikost paměťové reprezentace lze zjistit operátorem **sizeof**, kde argumentem je jméno typu nebo proměnné.

```
unsigned int ui;
fprintf(stdout, "%lu\n", sizeof(unsigned int));
fprintf(stdout, "ui size: %lu\n", sizeof(ui));
```

lec08/types.c

- 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`

lec08/inttypes.c

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

Logická hodnota – Boolean

- V původním C není definována, hodnota *true* je libovolná hodnota typu `int` různá od 0
- Pro hodnoty *true* a *false* můžeme definovat makro preprocesoru

```
int a = 10;
if (a) {
   /* true */
} else {
   /* false */
}
```

```
#define FALSE 0
#define TRUE 1
if (a != FALSE) {
   /* true */
} else {
   /* false */
}
```

- V C verzi (**normě**) **C99** můžeme využít definice v hlavičkovém souboru `stdbool.h`

```
#include <stdbool.h>
int i = 10;
bool b = i == 10 ? true : false;
if (b) {
   fprintf(stdout, "b is true\n");
} else {
   fprintf(stdout, "b is false\n");
}
```

lec08/bool.c

Definice proměnných a operátor přiřazení

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

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

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

```
/* int c, i, j; */
i = j = 10;
if ((c = 5) == 5) {
   fprintf(stdout, "c is 5 \n");
} else {
   fprintf(stdout, "c is not 5\n");
}
```

lec08/assign.c

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
- Znaky pro řízení výstupních zařízení (**escape sequences**)
  - \t – tabelátor (tabular), \n – nový řádek (newline),
  - \a – pípnutí (beep), \b – backspace, \r – carriage return,
  - \f – form feed, \v – vertical space
- 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

  Např. \141, \x61 `lec08/esqdho.c`
- \0 – znak pro konec textového řetězce (null character)

Identifikátory

- 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`

Číselné datové typy

- 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` $\leq$ `int` $\leq$ `long`
  - `unsigned short` $\leq$ `unsigned` $\leq$ `unsigned long`
- Zápis 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`
- Neceločíselné datové typy jsou dané implementací, většinou se řídí standardem IEEE-754-1985 `float`, `double`

Literály

- Jazyk C má 6 typů konstant (literálů)
  - Celočíselné
  - Racionální
  - Znakové
  - Řetězcové
  - Výčtové

Enum

  - Symbolické – `#define NUMBER 10`

Preprocesor

Literály celých čísel - binární zápis 1/2

- C standardně nepodporuje binární zápis celých čísel (formou binárního literálu)

```
Lze využít šestnáctkový zápis např.  0x12, který má
k binárnímu zápis velmi blízko, jeden znak jeden byte.
```

- GNU gcc umožňuje zápis binárního literálu s prefixem 0b nebo 0B

```c
#include <stdio.h>

int main(int argc, char **argv) {
   int vh = 0x10;
   int vd = 16;
   int vb = 0b10000;

   fprintf(stdout, "v hex %d\n", vh);
   fprintf(stdout, "v dec %d\n", vd);
   fprintf(stdout, "v bin %d\n", vb);
   return 0;
}
```

```
gcc binary.c -o binary
./binary
v hex 16
v dec 16
v bin 16
```

lec08/binary_literal.c

https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html

- Makro z Boost knihovny (lze použít v C)

```c
# include <boost/utility/binary.hpp>
...
int vb = BOOST_BINARY(10000);
```

Literály celých čísel - binární zápis 2/2

- Alternativně je možné využít vlastní funkci a makro

```c
#include <stdio.h>

#define B(x) S_to_binary_(#x)

static inline unsigned long long
    S_to_binary_(const char *s) {
   unsigned long long i = 0;
   while (*s) {
      i <<= 1;
      i += *s++ - '0';
   }
   return i;
}

int main(int argc, char **argv) {
   int vh = 0x10;
   int vd = 16;
   int vb = B(10000);

   fprintf(stdout, "v bin %d\n", vb);
   return 0;
}
```

```
clang -O3 -S binary_function.c
....
movq __stdoutp(%rip), %rdi
movl $.L.str1, %esi
movl $16, %edx
xorl %eax, %eax
callq  fprintf
movq __stdoutp(%rip), %rdi
movl $.L.str2, %esi
movl $16, %edx
xorl %eax, %eax
callq  fprintf
movq __stdoutp(%rip), %rdi
movl $.L.str3, %esi
movl $16, %edx
xorl %eax, %eax
callq  fprintf
xorl %eax, %eax
popq %rbp
ret
```

lec08/binary_function.c

http://stackoverflow.com/questions/15114140/writing-binary-number-system-in-c-code

Informativní

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 = 10l;
```

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

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

Pole tak musí být vždy o 1 položku delší!

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,
    HEARDS,
    DIAMONDS
};
```

```
enum {
    SPADES = 10,
    CLUBS, /* the value is 11 */
    HEARDS = 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, HEARDS, 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, můžeme použít kulaté závorky pro zachování předepsaného pořadí vyhodnocení.

Výrazy a operátory

- Výraz se skládá z operátorů a operandů
  - Nejjednodušší výraz tvoří konstanta, proměnná nebo volání funkce
  - Výraz sám může být operandem
  - Výraz má **typ** a **hodnotu**

    Pouze výraz typu `void` *hodnotu nemám.*
  - Výraz zakončený středníkem `;` je příkaz
- Postup výpočtu výrazu s více operátory je dán prioritou operátorů
  - Postup výpočtu lze předepsat použitím kulatých závorek `(` a `)`
- Operátory
  - Aritmetické, relační, logické, bitové
  - Arita operátoru (počet operandů) – unární, binární, ternární

    Jediný ternární operátor je podmíněný příkaz `?` `:`
  - Pořadí vyhodnocení operátorů není definováno

    Mimo definované případy unárních operátorů a logického vyhodnocení

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

Aritmetické operátory

- Operandy aritmetických operátorů mohou být libovolného aritmetického typu

*Výjimkou je operátor zbytek po dělení % definovaný pro **int***

*	Násobení	`x * y`	Součin x a y
/	Dělení	`x / y`	Podíl x a y
%	Dělení modulo	`x % y`	Zbytek po dělení x a y
+	Sčítání	`x + y`	Součet x a y
-	Odčítaní	`x - y`	Rozdíl a y
+	Kladné znam.	`+x`	Hodnota x
-	Záporné znam.	`-x`	Hodnota -x
++	Inkrementace	`++x`/`x++`	Inkrementace před/po vyhodnocení výrazu x
--	Dekrementace	`--x`/`x--`	Dekrementace před/po vyhodnocení výrazu x

Relační operátory

- Operandy relačních operátorů mohou být aritmetického typu, ukazatele shodného typu nebo jeden z nich `NULL` nebo typ `void`

`<`	Menší než	`x < y`	1 pro x je menší než y, jinak 0
`<=`	Menší nebo rovno	`x <= y`	1 pro x menší nebo rovno y, jinak 0
`>`	Větší než	`x > y`	1 pro x je větší než y, jinak 0
`>=`	Větší nebo rovno	`x >= y`	1 pro x větší nebo rovno y, jinak 0
`==`	Rovná se	`x == y`	1 pro x rovno y, jinak 0
`!=`	Nerovná se	`x != y`	1 pro x nerovno y, jinak 0

Logické operátory

- Operandy mohou být aritmetické typy nebo ukazatele
- Výsledek 1 má význam `true`, 0 má význam `false`
- Ve výrazech `&&` a `||` se vyhodnotí nejdříve levý operand
- Pokud je výsledek dán levým operandem, pravý se nevyhodnocuje

Zkrácené vyhodnocování – složité výrazy

`&&`	Logické AND	`x && y`	1 pokud x ani y není rovno 0, jinak 0
`||`	Logické OR	`x || y`	1 pokud alespoň jeden z x, y není rovno 0, jinak 0
`!`	Logické NOT	`!x`	1 pro x rovno 0, jinak 0

Bitové operátory

- Bitové operátory vyhodnocují operandy bit po bitu
- Operátory bitového posunu posouvají celý bitový obraz o zvolený počet bitů vlevo nebo vpravo
  - Při posunu vlevo jsou uvolněné bity zleva plněny 0
  - Při posunu vpravo jsou uvolněné bity zprava
    - u čísel kladných nebo typu unsigned plněny 0
    - u záporných čísel buď plněny 0 (logical shift) nebo 1 (arithmetic shift right), dle implementace překladače.

`&`	Bitové AND	`x & y`	1 když x i y je rovno 1 (bit po bitu)
`|`	Bitové OR	`x | y`	1 když x nebo y je rovno 1 (bit po bitu)
`^`	Bitové XOR	`x ^ y`	1 pokud oba x a y jsou 0 nebo 1 (bit po bitu)
`~`	Bitové NOT	`~x`	1 pokud x je rovno 0 (bit po bitu)
`<<`	Posun vlevo	`x << y`	Posun x o y bitů vlevo
`>>`	Posun vpravo	`x >> y`	Posun x o y bitů vpravo

Příklad – bitových operací

```
uint8_t a = 4;
uint8_t b = 5;

a   dec: 4 bin: 0100
b   dec: 5 bin: 0101
a&b dec: 4 bin: 0100
a|b dec: 5 bin: 0101
a^b dec: 1 bin: 0001

a>>1 dec: 2 bin: 0010
a<<1 dec: 8 bin: 1000
```

lec08/bits.c

Operátory přístupu do paměti

- V C máme pro proměnné možnost přístupu k adrese paměti, kde je hodnota proměnné uložena
- Přístup do paměti je prostřednictvím ukazatele (*pointeru*)

Dává velké možnosti, ale také vyžaduje zodpovědnost.

Operátor	Význam	Příklad	Výsledek
`&`	Adresa proměnné	`&x`	Ukazatel (pointer) na x
`*`	Nepřímá adresa	`*p`	Proměnná (nebo funkce) adresovaná pointerem p
`[]`	Prvek pole	`x[i]`	`*(x+i)` – prvek pole x s indexem i
`.`	Prvek struct/union	`s.x`	Prvek x struktury s
`->`	Prvek struct/union	`p->x`	Prvek struktury adresovaný ukazatelem p

Operandem operátoru & nesmí být bitové pole a proměnná typu register.

*Operátor nepřímé adresy * umožňuje přístup na proměnné přes ukazatel.*

Ostatní operátory

- Operandem `sizeof()` může být jméno typu nebo výraz

`()`	Volání funkce	`f(x)`	Volání funkce `f` s argumentem `x`
`(type)`	Přetypování (cast)	`(int)x`	Změna typu `x` na `int`
`sizeof`	Velikost prvku	`sizeof(x)`	Velikost `x` v bajtech
`? :`	Podmíněný příkaz	`x ? y : z`	Proveď `y` pokud `x!=0` jinak `z`
`,`	Postupné vyhodnocení	`x, y`	Vyhodnotí `x` pak `y`

- Příklad použití operátoru čárka

```
for(c = 1, i = 0; i < 3; ++i, c += 2) {
    fprintf(stdout, "i: %d c: %d\n", i, c);
}
```

Operátor přetypování

- Změna typu za běhu programu se nazývá přetypování
- Explicitní přetypování (cast) zapisuje programátor uvedením typu v kulatých závorkách, např.

```
int i;
float f = (float)i;
```

- Implicitní přetypování provádí překladač automaticky při překladu
- Pokud nový typ může reprezentovat původní hodnotu, přetypování ji vždy zachová
- Operandy typů `char`, `unsigned char`, `short`, `unsigned short`, případně bitová pole, mohou být použity tam kde je povolen typ `int` nebo `unsigned int`. C očekává hodnoty alespoň typu int
  - Operandy jsou automaticky přetypovány na `int` nebo `unsigned int`.

Příklad signed a unsigned

- V případě znaménkových typů je rozsah rozdělen na kladná a záporná čísla

```
unsigned char uc = 129;
char c = 129;
fprintf(stdout, "uc: %d\n c: %d\n", uc, c);
int i;
for(i = 0, c = 120, uc = c; i < 10; ++i, ++c, ++uc) {
   fprintf(stdout, "i: %d uc: %d; c: %d\n", i, uc, c);
}

for(i = 0, uc = 250; i < 10; ++i, ++uc) {
   fprintf(stdout, "i: %d uc: %d\n", i, uc);
}
```

```
uc: 129
c: -127
```

lec08/signed.c

Příkaz a složený příkaz (blok)

- Příkaz je výraz zakončený středníkem

Příkaz tvořený pouze středníkem je prázdný příkaz

- Blok je tvořen seznamem deklarací a seznamem příkazů
- Uvnitř bloku musí deklarace předcházet příkazům

Záleží na standardu jazyka, platí pro ANSI C (C89, C90)

- Začátek a konec bloku je vymezen složenými závorkami { a }
- Bloky mohou být vnořené do jiného bloku

```
void function(void)
{ /* function block start */
   {/* inner block */
      for(i = 0; i < 10; ++i)
      {
      //inner for-loop block
      }
   }
}
```

```
void function(void) { /* function
     block start */
   { /* inner block */
      for(int i = 0; i < 10; ++i) {
      //inner for-loop block
      }
   }
}
```

Různé kódovací konvence

http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html
https://www.doc.ic.ac.uk/lab/cplus/cstyle.html
http://en.wikipedia.org/wiki/Indent_style

Příkazy řízení běhu programu

- Podmíněné řízení běhu programu
  - Podmíněný příkaz: `if ()` nebo `if () ... else`
  - Programový přepínač: `switch () case ...`
- Cykly
  - `for ()`
  - `while ()`
  - `do ... while ()`
- Nepodmíněné větvení programu
  - `continue`
  - `break`
  - `return`
  - `goto`

Podmíněný příkaz

- `if (vyraz) prikaz1; else prikaz2`
- Je-li hodnota výrazu `vyraz != 0` provede se příkaz `prikaz1` jinak `prikaz2`

Příkaz může být blok příkazů

- Část `else` je nepovinná
- Podmíněné příkazy mohou být vnořené, např.

```
int max;
if (a > b) {
   if (a > c) {
      max = a;
   }
}
```

- Podmíněné příkazy můžeme řetězit, např.

```
int max;
if (a > b) {
   ...
} else if (a < c) {
   ...
} else if (a == b) {
   ...
} else {
   ...
}
```

Programový přepínač

- Přepínač `switch(vyraz)` větví program do *n* směrů
- Hodnota `vyraz` je porovnávána s *n* konstantními výrazy typu `int` příkazy `case konst_x: ...`
- Hodnota `vyraz` musí být celočíselná a hodnoty `konst_x` musejí být navzájem různé
- Pokud je nalezena shoda, program pokračuje od tohoto místa dokud nenajde příkaz `break` nebo konec příkazu `switch`
- Pokud shoda není nalezena, program pokračuje nepovinnou sekcí `default`

Sekce `default` se zpravidla uvádí jako poslední

- Příkazy `switch` mohou být vnořené.

Programový přepínač – Příklad

```c
switch (v) {
    case 'A':
        printf("Upper A\n");
        break;
    case 'a':
        printf("Lower a\n");
        break;
    default:
        printf(
        "It is not A nor a\n");
        break;
}
```

```c
if (v == 'A') {
    printf("Upper A\n");
} else if (v == 'a') {
    printf("Lower a\n");
} else {
    printf(
    "It is not A nor a\n");
}
```

lec08/switch.c

Cyklus `for( ; ; )`

- Příkaz **for** cyklu má tvar `for ([vyraz1];[vyraz2];[vyraz3]) prikaz;`
- Cyklus **for** používá řídící proměnnou a probíhá následovně:
  1. `vyraz1` – Inicializace (zpravidla řídící proměnné)
  2. `vyraz2` – Test řídícího výrazu
  3. Pokud `vyraz2 !=0` provede se `prikaz`, jinak cyklus končí
  4. `vyraz3` – Aktualizace proměnných na konci běhu cyklu
  5. Opakování cyklu testem řídícího výrazu
- Výrazy `vyraz1` a `vyraz3` mohou být libovolného typu
- Libovolný z výrazů lze vynechat
- **break** – cyklus lze nuceně opustit příkazem `break`
- **continue** – část těla cyklu lze vynechat příkazem `continue`

  Příkaz přeruší vykonávání těla (blokového příkazu) pokračuje vyhodnocením `vyraz3`.
- Při vynechání řídicího výrazu `vyraz2` se cyklus bude provádět nepodmíněně

```
for (;;) {...}
```

Nekonečný cyklus

Cyklus `while ()`

- Příkaz **while** má tvar `while (vyraz1) prikaz;`
- Příkaz cyklu **while** probíhá
  1. Vyhodnotí se výraz `vyraz1`
  2. Pokud `vyraz1 != 0`, provede se příkaz `prikaz`, jinak cyklus končí
  3. Opakování vyhodnocení výrazu `vyraz1`
- Řídící cyklus se vyhodnocuje na začátku cyklu, cyklus se nemusí provést ani jednou
- Řídící výraz `vyraz1` se musí aktualizovat v těle cyklu, jinak je cyklus nekonečný

Cyklus `do...while ()`

- Příkaz **do...while ()** má tvar `do prikaz while (vyraz1);`
- Příkaz cyklu **do...while ()** probíhá
  1. Provede se příkaz `prikaz`
  2. Vyhodnotí se výraz `vyraz1`
  3. Pokud `vyraz1 != 0`, cyklus se opakuje provedením příkazu `prikaz`, jinak cyklus končí
- Řidící cyklus se vyhodnocuje na konci cyklu, tělo cyklu se vždy provede nejméně jednou
- Řídící výraz `vyraz1` se musí aktualizovat v těle cyklu, jinak je cyklus nekonečný

Příkaz continue

- Příkaz návratu na vyhodnocení řídicího výrazu – `continue`
- Příkaz `continue` lze použít pouze v těle cyklů
  - `for ()`
  - `while ()`
  - `do...while ()`
- Příkaz `continue` způsobí přerušení vykonávání těla cyklu a nové vyhodnocení řídicího výrazu
- Příklad

```
int i;
for (i = 0; i < 20; i++) {
   if (i % 2 == 0) {
      continue;
   }
   fprintf(stdout, "%d\n", i);
}
```

lec08/continue.c

Příkaz break

- Příkaz nuceného ukončení cyklu `break;`
- Příkaz `break` lze použít pouze v těle cyklů
  - `for()`
  - `while()`
  - `do...while()`
- a v těle programového přepínače `switch()`
- Příkaz `break` způsobí opuštění těla cyklu nebo těla `switch()`,
- program pokračuje následujícím příkazem, např.

```
int i = 10;
while (i > 0) {
    if (i == 5) {
        printf("i reaches 5, leave the loop\n");
        break;
    }
    i--;
    printf("End of the while loop i: %d\n", i);
}
```

lec08/break.c

Příkaz `return`

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

Příkaz goto

- Příkaz nepodmíněného lokálního skoku `goto`
- Syntax `goto navesti;`
- Příkaz `goto` lze použít pouze v těle funkce
- Příkaz `goto` předá řízení na místo určené návěstím `navesti`
- Skok `goto` nesmí směrovat dovnitř bloku, který je vnořený do bloku, kde je příslušné `goto` umístěno

```
1  int test = 3;
2  for(int i = 0; i < 3; ++i) {
3     for (int j = 0; j < 5; ++j) {
4        if (j == test) {
5           goto loop_out;
6        }
7        fprintf(stdout, "Loop i: %d j: %d\n", i, j);
8     }
9  }
10 return 0;
11 loop_out:
12 fprintf(stdout, "After loop\n");
13 return -1;
```

lec08/goto.c

Příkazy dlouhého skoku

- Příkaz `goto` je možné použít pouze v rámci jedné funkce
- Knihovna `<setjmp.h>` definuje funkce `setjmp` a `longjmp` pro skoky mezi funkcemi
- `setjmp` uloží aktuální stav registrů procesoru a pokud funkce vrátí hodnotu různou od 0, došlo k volání `longjmp`
- Při volání `longjmp` jsou hodnoty registrů procesoru obnoveny a program pokračuje od místa volání `setjmp`

Kombinaci `setjmp` a `longjmp` lze využít pro implementace ošetření výjimečných stavu podobně jako `try-catch`

```
1  #include <setjmp.h>
2  jmp_buf jb;
3  int compute(int x, int y);
4  void error_handler(void);
5  if(setjmp(jb) == 0) {
6     r = compute(x, y);
7     return 0;
8  } else {
9     error_handler();
10    return -1;
11 }
```

```
12 int compute(int x, int y) {
13    if(y == 0) {
14       longjmp(jb, 1);
15    } else {
16       x = (x + y * 2);
17       return (x / y);
18    }
19 }
20 void error_handler(void)  {
21    fprintf("Error\n");
22 }
```

Informativní

Část II

Část 2 – Jazyk C - ukazatele, řetězce, funkce a volání funkcí

Pole (array)

- Pole je posloupnost prvků **stejného typu**
- K prvkům pole se přistupuje pořadovým číslem prvku
- **Index prvního prvku je** vždy roven **0**
- Prvky pole mohou být proměnné libovolného typu

I strukturované typy, viz další přednáška

- Pole může být jednorozměrné nebo vícerozměrné

Pole polí (. . .) prvků stejného typu.

- Prvky pole určuje: **jméno**, **typ**, **počet prvků**
- Počet prvků statického pole musí být znám v době překladu
- Prvky pole zabírají v paměti souvislou oblast!
- Velikost pole (v bajtech) je dána počtem prvků pole *n* a `typ`em prvku, tj. `n * sizeof(typ)`
- Textový řetězec je pole typu `char`, kde poslední prvek je `'\0'`

C nekontroluje za běhu programu, zdali je index platný!

Pole – příklady

- Deklarace pole

```
/* jednorozmerne pole prvku typu char */
char simple_array[10];

/* dvourozmerne pole prvku typu int */
int two_dimensional_array[2][2];
```

- Inicializace pole

```
double d[] = {0.1, 0.4, 0.5};
char str[] = "hallo";
char s[] = {'h', 'a', 'l', 'l', 'o', '\0'};
int m[3][3] = { { 1, 2, 3}, {4 , 5 ,6}, {7, 8, 9}};
char cmd[][10] = { "start", "stop", "pause" };
```

- Přístup k prvkům pole

```
m[1][2] = 2*1;
```

Ukazatel (pointer)

- Ukazatel (pointer) je proměnná jejíž **hodnota je adresa** paměti jiné proměnné
  Analogie nepřímé adresy ve strojovém kódu.
- Pointer *ukazuje* na jinou proměnnou
- **Ukazatel má** též **typ** proměnné, na kterou může ukazovat
  - Ukazatel na hodnoty (proměnné) základních typů: `char`, `int`, ...
  - „Ukazatel na pole"; ukazatel na funkci; ukazatel na ukazatele
- Ukazatel může být též bez typu (**void**)
  - Pak může obsahovat adresu libovolné proměnné
  - Velikost proměnné nelze z vlastnosti ukazatele určit
- Prázdná adresa ukazatele je definovaná hodnotou konstanty **NULL**
  Textová konstanta makro preprocesoru definované jako „null pointer constant"
  C99 – lze též použít „int" hodnotu 0

C za běhu programu nekontroluje platnost adresy v ukazateli

- Ukazatel umožňuje:
  - Předávání parametru odkazem; práci s poli; pointer na funkci, pole funkcí; provádět s nimi aritmetické operace

Ukazatel (pointer) – příklady 1/2

```
int i = 10; /* promenna typu int */
            /* &i -- adresa promenne i */

int *pi;    /* deklarace promenne typu pointer */
            /* pi pointer na promenou typu int */
            /* *pi promenna typu int */

pi = &i;    /* do pi se ulozi adresa promenne i */

int b;      /* promenna typu int */

b = *pi;    /* do promenne b se ulozi obsah adresy
               ulozene v ukazeteli pi                */
```

Ukazatel (pointer) – příklady 2/2

```
printf("i: %d -- pi: %p\n", i, pi); // 10 0x7ffffffe8fc
printf("&i: %p -- *pi: %d\n", &i, *pi); // 0x7ffffffe8fc
    10
printf("*(&)i: %d -- &(*pi): %p\n", *(&i), &(*pi));

printf("i: %d -- *pj: %d\n", i, *pj);   // 10 10
i = 20;
printf("i: %d -- *pj: %d\n", i, *pj);   // 20 20

printf("sizeof(i): %ld\n", sizeof(i));  // 4
printf("sizeof(pi): %ld\n", sizeof(pi));// 8

long l = (long)pi;
printf("0x%lx %p\n", l, pi); /* print l as hex  -- %lx */
// 0x7ffffffe8fc 0x7ffffffe8fc

l = 10;
pi = (int*)l; /* possible but it is nonsense */
printf("l: 0x%lx %p\n", l, pi); // 0xa 0xa
```

lec08/pointers.c

Ukazatele (pointery), proměnné a jejich hodnoty

- Proměnné jsou názvy adres, kde jsou uloženy hodnoty příslušného typu
- Kompilátor pracuje přímo s adresami

  Přestože se v případě kompilace zpravidla jedná o adresy relativní.
- Ukazatel (pointer) je proměnná, ve které je uložena adresa. Na této adrese se pak nachází hodnota nějakého typu (např. `int`).
- Ukazatele realizují tzv. **nepřímé adresování** (**indirect addressing**)
- Dereferenční operátor `*` přistupuje na proměnnou adresovanou hodnotou ukazatele
- Operátor `&` vrací adresu, kde je uložena hodnota proměnné

Ukazatele (pointery) a kódovací styl

- Typ ukazatel se značí symbolem `*`
- `*` můžeme zapisovat u jména typu nebo jména proměnné
- Preferujeme zápis u proměnné, abychom předešli omylům

```
char* a, b, c;
```
Pointer je pouze `a`

```
char *a, *b, *c;
```
Všechny tři proměnné jsou ukazatele

- Zápis typu ukazatele na ukazatel `char **a;`
- Zápis pouze typu (bez proměnné): `char*` nebo `char**`
- Ukazatel na proměnnou prázdného typu zapisujeme jako

```
void *ptr
```

- Prokazatelně neplatná adresa má symbolické jméno `NULL`

Definovaná jako makro preprocesoru (C99 lze použít 0)

- Proměnné v C nejsou automaticky inicializovány a mohou ukazovat na neplatnou paměť, proto může být vhodné explicitně inicializovat na `NULL`

Např. `int *i = NULL;`

Funkce a předávání parametrů

- V C jsou **parametry funkce předávány hodnotou**
- Parametry jsou lokální proměnné funkce (alokované na zásobníku), které jsou inicializované na hodnotu předávanou funkci

```
void fce(int a, char *b) { /*
 a - je lokalni promena typu int (ulozena na zasobniku)
 b - je lokalni promena typu ukazatel na promenou
     typu char (hodnota je adresa a je take na zasobniku)
   */
}
```

- Lokální změna hodnoty proměnné neovlivňuje hodnotu proměnné vně funkce
- Při předání ukazatele, však máme přístup na adresu původní proměnné, kterou můžeme měnit
- **Ukazatelem tak realizujeme volání odkazem**

Funkce a předávání parametrů – příklad

- Proměnná a realizuje volání hodnotou
- Proměnná b realizuje volání odkazem

```
void fce(int a, char* b) {
   a += 1;
   (*b)++;
}
int a = 10;
char b = 'A';
printf("Before call %d %c\n", a, b);
fce(a, &b);
printf("After call %d %c\n", a, b);
```

- Výstup

```
Before call 10 A
After call 10 B
```

lec08/function_call.c

Funkce `main` a její tvary

- Základní tvar funkce `main`

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

- Alternativně pak také

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

- Argumenty funkce nejsou nutné

```
int main(void) { ... }
```

- Rozšířená funkce o nastavení proměnných prostředí

Pro Unix a MS Windows

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

Přístup k proměnným prostředí funkcí `getenv` z knihovny `stdlib.h`.

`lec08/main_env.c`

- Rozšířená funkce o specifické parametry Mac OS X

```
int main(int argc, char **argv, char **envp, char **apple);
```

Pointery a pole – Část 1

- Pointer ukazuje na vyhrazenou část paměti proměnné

  Předpokládáme správné použití

- Pole je označení souvislého bloku paměti

```
int *p; //ukazatel (adresa) kde je ulozena hodnota int
int a[10]; //souvisly blok pameti pro 10 int hodnot

sizeof(p); //pocet bytu pro ulozeni adresy (8 pro 64bit)
sizeof(a); //velikost alokovaneho pole je 10*sizeof(int)
```

- Obě proměnné odkazují na paměť, kompilátor s nimi však pracuje **rozdílně**
  - Proměnná typu pole je symbolické jméno pro místo v paměti, kde jsou uloženy hodnoty prvků pole

    Kompilátor nahrazuje jméno přímo paměťovým místem
  - Ukazatel obsahuje adresu, na které je příslušná hodnota (nepřímé adresování)
- **Při předávání pole jako parametru funkce je předáváno pole jako pointer (ukazatel)**

Viz kompilace souboru `main_env.c` *překladačem* `clang`

Příklad ukazatele a pole

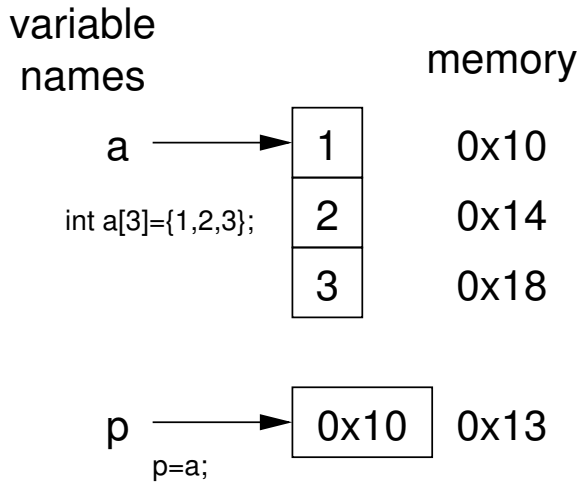
- Proměnná pole `int a[3] = {1,2,3};`

  `a` odkazuje na adresu prvního prvku pole

- Proměnná ukazatel `int *p = a;`

  ukazatel `p` obsahuje adresu prvního prvku pole

- Hodnota `a[1]` přímo reprezentuje hodnotu na adrese `0x14`.



- Oběma přístupy se dostaneme na příslušné prvky pole, způsob je však odlišný

http://eli.thegreenplace.net/2009/10/21/are-pointers-and-arrays-equivalent-in-c

Pointerová aritmetika

- S pointery lze provádět aritmetické operace + a -, kde operandy mohou být:
  - ukazatel (pointer) a celé číslo (int)
  - ukazatel stejného typu
- Aritmetické operace jsou užitečné pokud ukazatel odkazuje na více položek daného typu (souvislý blok paměti)
  - Např. pole položek příslušného typu
  - Dynamicky alokovaný souvislý blok paměti
- Přičtením hodnoty celého čísla k pointeru „posouváme" hodnotu pointeru na další prvek, např.

```
int a[10];
int *p = a;

int i = *(p+2); //odkazuje na hodnotu 3. prvku pole a
```

  - Podle typu ukazatele se hodnota adresy příslušně zvýší
  - `(p+2)` je ekvivalentní adrese `p + 2*sizeof(int)`
  - Příklad použití viz `lec08/pointers_and_array.c`

Shrnutí přednášky

Diskutovaná témata

- Jazyk C – Syntax, proměnné, základní typy a literály
- Jazyk C – Výrazy a operátory
- Jazyk C – Příkazy a řízení běhu programu
- Jazyk C – Pole, ukazatele, funkce a předávání parametrů
- Jazyk C – Ukazatele a pole

- Příště: struktury a uniony, definice typů, dynamická alokace paměti, práce se soubory a knihovní funkce