

# Ukazatele, paměťové třídy, volání funkcí

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 05

**B0B36PRP – Procedurální programování**

## Část I

### Část 1 – Ukazatele a dynamická alokace

## Přehled témat

### ■ Část 1 – Ukazatele a dynamická alokace

Modifikátor `const` a ukazatele

Dynamická alokace paměti

*S. G. Kochan: kapitoly 8 a 11*

*P. Herout: kapitoly 9 a 10*

### ■ Část 2 – Paměťové třídy a volání funkcí

Výpočetní prostředky a běh programu

Rozsah platnosti proměnných

Paměťové třídy

*S. G. Kochan: kapitola 8 a 11*

*P. Herout: kapitola 9*

### ■ Část 3 – Zadání 5. domácího úkolu (HW05)

## Modifikátor typu `const`

- Uvedením klíčového slova `const` můžeme označit proměnnou jako konstantu

*Překladač kontroluje přiřazení*

- Pro definici konstant můžeme použít např.

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

- Na rozdíl od symbolické konstanty

```
#define PI 3.14159265
```

- mají konstantní proměnné typ a překladač tak může provádět **typovou kontrolu**

*Připomínka*

## Ukazatele na konstantní proměnné a konstantní ukazatele

- Klíčové slovo **const** můžeme zapsat před jméno typu nebo před jméno proměnné
- Dostáváme 3 možnosti jak definovat ukazatel s **const**
  - (a) `const int *ptr;` – ukazatel na konstantní proměnnou
    - Nemůžeme použít pointer pro změnu hodnoty proměnné
  - (b) `int *const ptr;` – konstantní ukazatel
    - Pointer nemůžeme nastavit na jinou adresu než tu při inicializaci
  - (c) `const int *const ptr;` – konstantní ukazatel na konstantní hodnotu
    - Kombinuje předchozí dva případy

`lec05/const_pointers.c`

Další alternativy zápisu (a) a (c) jsou

- `const int *` lze též zapsat jako `int const *`
  - `const int * const` lze též zapsat jako `int const * const`
- `const` může být vlevo nebo vpravo od jména typu

- Nebo komplexnější definice, např. `int ** const ptr;` – konstantní ukazatele na ukazatel na int

## Příklad – Konstantní ukazatel

- Hodnotu konstantního ukazatele nelze po inicializaci měnit
- Zápis `int *const ptr;` můžeme číst zprava doleva
  - `ptr` – proměnná, která je
  - `*const` – konstantním ukazatelem
  - `int` – na proměnnou typu `int`

```

1 int v = 10;
2 int v2 = 20;
3 int *const ptr = &v;
4 printf("v: %d *ptr: %d\n", v, *ptr);
5
6 *ptr = 11; /* We can modify addressed value */
7 printf("v: %d\n", v);
8
9 ptr = &v2; /* THIS IS NOT ALLOWED! */

```

`lec05/const_pointers.c`

## Příklad – Ukazatel na konstantní proměnnou (hodnotu)

- Prostřednictvím ukazatele na konstantní proměnnou nemůžeme tuto proměnnou měnit

```

1 int v = 10;
2 int v2 = 20;
3
4 const int *ptr = &v;
5 printf("*ptr: %d\n", *ptr);
6
7 *ptr = 11; /* THIS IS NOT ALLOWED! */
8
9 v = 11; /* We can modify the original variable */
10 printf("*ptr: %d\n", *ptr);
11
12 ptr = &v2; /* We can assign new address to ptr */
13 printf("*ptr: %d\n", *ptr);

```

`lec05/const_pointers.c`

## Příklad – Konstantní ukazatel na konstantní proměnnou

- Hodnotu konstantního ukazatele na konstantní proměnnou nelze po inicializaci měnit a ani nelze prostřednictvím takového ukazatele měnit hodnotu adresované proměnné
- Zápis `const int *const ptr;` můžeme číst zprava doleva
  - `ptr` – proměnná, která je
  - `*const` – konstantním ukazatelem
  - `const int` – na proměnnou typu `const int`

```

1 int v = 10;
2 int v2 = 20;
3 const int *const ptr = &v;
4
5 printf("v: %d *ptr: %d\n", v, *ptr);
6
7 ptr = &v2; /* THIS IS NOT ALLOWED! */
8 *ptr = 11; /* THIS IS NOT ALLOWED! */

```

`lec05/const_pointers.c`

## Ukazatel na funkci

- Implementace funkce je umístěna někde v paměti a podobně jako na proměnnou v paměti může ukazatel odkazovat na paměťové místo s definicí funkce
- Můžeme definovat **ukazatel na funkci** a dynamicky volat funkci dle aktuální hodnoty ukazatele
- Součástí volání funkce jsou předávané parametry, které jsou též součástí typu ukazatele na funkci, resp. typu parametrů
- Funkce (a volání funkce) je identifikátor funkce a `()`, tj.
 

```
typ_návratové_hodnoty funkce(parametry funkce);
```
- Ukazatel na funkci definujeme jako
 

```
typ_návratové_hodnoty (*ukazatel)(parametry funkce);
```

## Příklad – Ukazatel na funkci 2/2

- V případě funkce vracející ukazatel postupujeme identicky
 

```
double* compute(int v);

double* (*function_p)(int v);
      ^----- substitute a function name

function_p = compute;
```
- Příklad použití ukazatele na funkci – `lec05/pointer_fnc.c`
- Ukazatele na funkce umožňují realizovat dynamickou vazbu volání funkce identifikované za běhu programu
 

*V objektově orientovaném programování je dynamická vazba klíčem k realizaci polymorfismu.*

## Příklad – Ukazatel na funkci 1/2

- Používáme dereferenční operátor `*` podobně jako u proměnných
 

```
double do_nothing(int v); /* function prototype */

double (*function_p)(int v); /* pointer to function */

function_p = do_nothing; /* assign the pointer */

(*function_p)(10); /* call the function */
```
- Závorky `(*function_p)` „pomáhají“ číst definici ukazatele
 

*Můžeme si představit, že závorky reprezentují jméno funkce. Definice proměnné ukazatel na funkci se tak v zásadě neliší od prototypu funkce.*
- Podobně je volání funkce přes ukazatel na funkci identické běžnému volání funkce, kde místo jména funkce vystupuje jméno ukazatele na funkci

## Definice typu – typedef

- Operátor `typedef` umožňuje definovat nový datový typ
- Slouží k pojmenování typů, např. ukazatele, struktury a uniony
 

*Struktury a uniony viz přednáška 6*
- Například typ pro ukazatele na `double` a nové jméno pro `int`:
 

```
1 typedef double* double_p;
2 typedef int integer;
3 double_p x, y;
4 integer i, j;
```
- je totožné s použitím původních typů
 

```
1 double *x, *y;
2 int i, j;
```
- Zavedením typů operátorem `typedef`, např. v hlavičkovém souboru, umožňuje systematické používání nových jmen typů v celém programu
 

*Viz např. `inttypes.h`*
- Výhoda zavedení nových typů je především u složitějších typů jako jsou ukazatele na funkce nebo struktury

## Dynamická alokace paměti

- Přídělení bloku paměti velikosti `size` lze realizovat funkcí

```
void* malloc(size); z knihovny <stdlib.h>
```

- Velikost alokované paměti je uložena ve správci paměti
- **Velikost není součástí ukazatele**
- Návrátová hodnota je typu `void*` – přetypování nutné
- **Je plně na uživateli (programátorovi), jak bude s pamětí zacházet**

- Příklad alokace paměti pro 10 proměnných typu `int`

```
1 int *int_array;
2 int_array = (int*)malloc(10 * sizeof(int));
```

- Operace s více hodnotami v paměťovém bloku je podobná poli
  - Používáme pointerovou aritmetiku

- **Uvolnění paměti**

```
void* free(pointer);
```

- Správce paměti uvolní paměť asociovanou k ukazateli
- Hodnotu ukazatele však nemění!

*Stále obsahuje předešlou adresu, která však již není platná.*

## Příklad alokace dynamické paměti 2/3

- Pro vyplnění hodnot pole alokovaného dynamicky nám postačuje předávat hodnotu adresy paměti pole

```
1 void fill_array(int* array, int size)
2 {
3     for (int i = 0; i < size; ++i) {
4         *(array++) = random();
5     }
6 }
```

- Po uvolnění paměti odkazuje ukazatel stále na původní adresu, proto můžeme explicitně nulovat

*Předání ukazatele na ukazatele je nutné, jinak nemůžeme nulovat.*

```
1 void deallocate_memory(void **ptr)
2 {
3     if (ptr != NULL && *ptr != NULL) {
4         free(*ptr);
5         *ptr = NULL;
6     }
7 }
```

lec05/malloc\_demo.c

## Příklad alokace dynamické paměti 1/3

- Alokace se nemusí nutně povést – testujeme návratovou hodnotu funkce `malloc()`
- Pro vyplnění adresy alokované paměti předáváme proměnnou jako ukazatel na proměnnou typu ukazatel na `int`

```
1 void* allocate_memory(int size, void **ptr)
2 {
3     // use **ptr to store value of newly allocated
4     // memory in the pointer ptr (i.e., the address the
5     // pointer ptr is pointed).
6
7
8     // call library function malloc to allocate memory
9     *ptr = malloc(size);
10
11    if (*ptr == NULL) {
12        fprintf(stderr, "Error: allocation fail");
13        exit(-1); /* exit program if allocation fail */
14    }
15    return *ptr;
16 }
```

lec05/malloc\_demo.c

## Příklad alokace dynamické paměti 3/3

- Příklad použití

```
1 int main(int argc, char *argv[])
2 {
3     int *int_array;
4     const int size = 4;
5
6     allocate_memory(sizeof(int) * size, (void**)&int_array);
7     fill_array(int_array, size);
8     int *cur = int_array;
9     for (int i = 0; i < size; ++i, cur++) {
10        printf("Array[%d] = %d\n", i, *cur);
11    }
12    deallocate_memory((void**)&int_array);
13    return 0;
14 }
```

lec05/malloc\_demo.c

## Část II

### Část 2 – Paměťové třídy, model výpočtu

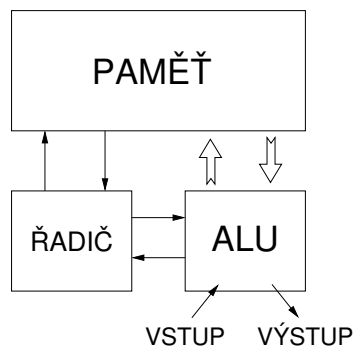
### Von Neumannova architektura

*V drtivě většině případů je program posloupnost instrukcí zpracovávající jednu nebo dvě hodnoty (uložené v nějakém paměťovém místě) jako vstup a generování nějaké výstupní hodnoty, kterou ukládá někam do paměti nebo modifikuje hodnotu PC (podmíněně řízení běhu programu).*

- **ALU** - Aritmeticko logická jednotka (Arithmetic Logic Unit)

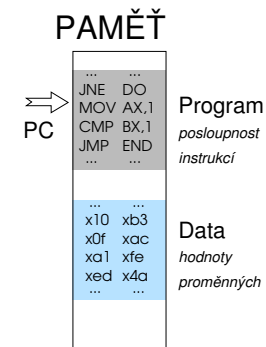
*Základní matematické a logické instrukce*

- **PC** obsahuje adresu kódu – při volání funkce tak jeho hodnotu můžeme uložit (na zásobník) a následně použít pro návrat na původní místo volání



### Paměť počítače s uloženým programem v operační paměti

- Posloupnost instrukcí je čtena z operační paměti
- Flexibilita ve tvorbě posloupnosti  
*Program lze libovolně měnit*
- Architektura počítače se společnou pamětí pro data a program
  - Von Neumannova architektura počítače  
*John von Neumann (1903–1957)*
  - Sdílí program i data ve stejné paměti
  - Adresa aktuálně prováděné instrukce je uložena v tzv. čítači instrukcí (Program Counter **PC**)



- Mimoto architektura se sdílenou pamětí umožňuje, aby hodnota ukazatele odkazovala nejen na data, ale také například na část paměti, kde je uložen program (funkce)

*Princip ukazatele na funkci*

### Základní rozdělení paměti

- Přidělenou paměť programu můžeme kategorizovat na 5 částí

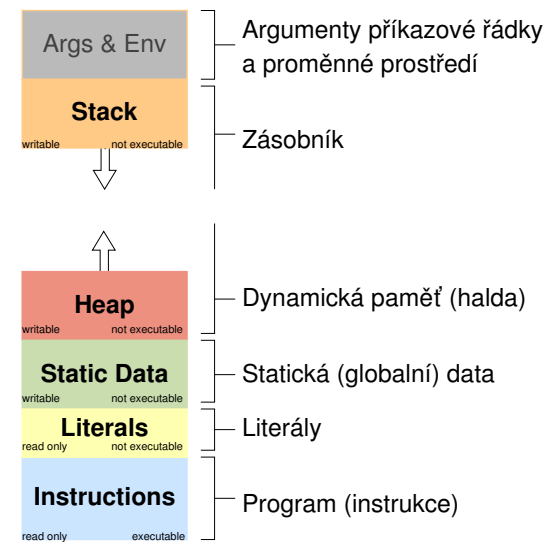
- **Zásobník** – lokální proměnné, argumenty funkcí, návratová hodnota funkce  
*Spravováno automaticky*

- **Halda** – **dynamická** paměť (`malloc()`, `free()`)  
*Spravuje programátor*

- **Statická** – globální nebo „lokální“ **static** proměnné  
*Inicializováno při startu*

- **Literály** – hodnoty zapsané ve zdrojovém kódu programu, např. textové řetězce  
*Inicializováno při startu*

- **Program** – strojové instrukce  
*Inicializováno při startu*



## Rozsah platnosti (scope) lokální proměnné

- Lokální proměnné mají rozsah platnosti pouze uvnitř bloku a funkce

```

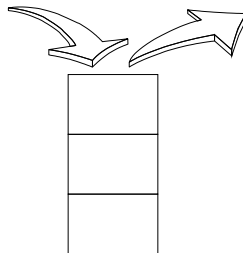
1 int a = 1; // globální proměnná
2
3 void function(void)
4 { // zde a ještě reprezentuje globální proměnnou
5   int a = 10; // lokální proměnná, zastíňuje globální a
6   if (a == 10) {
7     int a = 1; // nová lokální proměnná a; přístup
8               // na původní lokální a je zastíněn
9     int b = 20; // lokální proměnná s platností pouze
10               // uvnitř bloku
11     a = b + 10; // proměnná a má hodnotu 11
12   } // konec bloku
13   // zde má a hodnotu 10, je to lokální proměnná z řádku 5
14
15   b = 10; // b není platnou proměnnou
16 }
```

- Globální proměnné mají rozsah platnosti „kdekoliv“ v programu
  - Zastíněný přístup lze řešit modifikátorem `extern` (v novém bloku)
    - [http://www.tutorialspoint.com/cprogramming/c\\_scope\\_rules.htm](http://www.tutorialspoint.com/cprogramming/c_scope_rules.htm)

## Zásobník

- Úseky paměti přidělované lokálním proměnným a parametrům funkce tvoří tzv. **zásobník** (stack)
- Úseky se přidávají a odebírají
  - Vždy se odebere naposledy přidáný úsek

*LIFO – last in, first out*



- Na zásobník se ukládá „volání funkce“

*Na zásobník se také ukládá návratová hodnota funkce a také hodnota „program counter“ původně prováděné instrukce, před voláním funkce*

- Ze zásobníku se alokují proměnné parametrů funkce

*Argumenty (parametry) jsou de facto lokální proměnné*

**Opakovaným rekurzivním voláním funkce můžeme zaplnit velikost přiděleného zásobníku a program skončí chybou.**

## Přidělování paměti proměnným

- Přidělením paměti proměnné rozumíme určení paměťového místa pro uložení hodnoty proměnné (příslušného typu) v paměti počítače
- Lokálním proměnným a parametrům funkce se paměť přiděluje při volání funkce
  - Paměť zůstane přidělena jen do návratu z funkce
  - Paměť se automaticky alokuje z rezervovaného místa – **zásobník (stack)**
    - Při návratu funkce se přidělené paměťové místo uvolní pro další použití*
  - Výjimku tvoří lokální proměnné s modifikátorem `static`
    - Z hlediska platnosti rozsahu mají charakter lokálních proměnných
    - Jejich hodnota je však zachována i po skončení funkce / bloku
    - Jsou umístěny ve statické části paměti
- Dynamické přidělování paměti
  - Alokační paměti se provádí funkcí `malloc()`
    - Nebo její alternativou podle použité knihovny pro správu paměti (např. s garbage collectorem – boehm-gc)*
  - Paměť se alokuje z rezervovaného místa – **halda (heap)**

## Příklad rekurzivního volání funkce

```

#include <stdio.h>
void printValue(int v)
{
    printf("value: %i\n", v);
    printValue(v + 1);
}
int main(void)
{
    printValue(1);
}
lec05/demo-stack_overflow.c
```

- Vyzkoušejte si program pro omezenou velikost zásobníku

```

clang demo-stack_overflow.c
ulimit -s 1000; ./a.out | tail -n 3
value: 31730
value: 31731
Segmentation fault

ulimit -s 10000; ./a.out | tail -n 3
value: 319816
value: 319817
Segmentation fault
```

## Vsuvka – Kódovací styl `return` 1/2

- Předání hodnoty volání funkce je předepsáno voláním `return`

```
int doSomethingUseful() {
    int ret = -1;
    ...
    return ret;
}
```

- Jak často umisťovat volání `return` ve funkci?

<pre>int doSomething() {     if (         !cond1         &amp;&amp; cond2         &amp;&amp; cond3     ) {         ... do some long code ...     }     return 0; }</pre>	<pre>int doSomething() {     if (cond1) {         return 0;     }     if (!cond2) {         return 0;     }     if (!cond3) {         return 0;     }     ... some long code ....     return 0; }</pre>
--	---

<http://llvm.org/docs/CodingStandards.html>

## 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 knihovnými funkcemi

*Např. `malloc()` a `free()` z knihovny `<stdlib.h>` nebo `<malloc.h>`*

[http://gribblelab.org/CBootcamp/7\\_Memory\\_Stack\\_vs\\_Heap.html](http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html)

## Vsuvka – Kódovací styl `return` 2/2

- Volání `return` na začátku funkce může být přehlednější

*Podle hodnoty podmínky je volání funkce ukončeno*

- Kódovací konvence může také předepisovat použití nejvýše jedno volání `return`

*Má výhodu v jednoznačné identifikaci místa volání, můžeme pak například jednoduše přidat další zpracování výstupní hodnoty funkce.*

- Dále není doporučováno bezprostředně používat `else` za voláním `return` (nebo jiným přerušení toku programu), např.

<pre>case 10:     if (...) {         ...         return 1;     } else {         if (cond) {             ...             return -1;         } else {             break;         }     } }</pre>	<pre>case 10:     if (...) {         ...         return 1;     } else {         if (cond) {             ...             return -1;         }         break;     } }</pre>
--	---

## 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 stejně 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; lec05/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 {
9     int lv = 0; /* local variable */
10    static int lsv = 0; /* local static variable */
11    lv += 1;
12    lsv += 1;
13    printf("func: p%d, lv %d, lsv %d\n", p, lv, lsv);
14 }
15 int main(void)
16 {
17     int local;
18     function(1);
19     function(1);
20     function(1);
21     return 0;
22 }
```

- 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
```

lec05/vardec.c

## Část III

### Část 3 – Zadání 5. domácího úkolu (HW05)

## Definice proměnných a operátor přiřazení

- Proměnné definujeme uvedením typu a jména proměnné

- Jména proměnných volíme malá písmena
- Víceslovná jména zapisujeme s podtržítkem `_` nebo volíme tzv. *camelCase* <https://en.wikipedia.org/wiki/CamelCase>
- 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");
}
```

lec05/assign.c

## Zadání 5. domácího úkolu HW05

### Téma: Maticové počty

Povinné zadání: **3b**; Volitelné zadání: **3b**; Bonusové zadání: **5b**

- **Motivace:** Získání zkušenosti s dvojrozměrnými polem.
- **Cíl:** Osvojit si práci s polem variabilní délky a předávání ukazatelů
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw05>
  - Načtení vstupních hodnot dvou matic a znaku operace (`'*'` – násobení).
  - **Volitelné zadání** rozšiřuje úlohu o další operace s maticemi sčítání (`'+'`) a odčítání (`'-'`), které mohou být zapsány ve výrazu.
  - **Bonusové zadání** pak řeší zpracování celého výrazu, ve kterém jsou však jednotlivé matice uvedeny jako symboly, které jsou nejdříve definovány načtením hodnot matic ze standardního vstupu

- **Termín odevzdání:** **19.11.2016, 23:59:59 PST**

*PST – Pacific Standard Time*



## Shrnutí přednášky

## Diskutovaná témata

- Ukazatele a modifikátor `const`
- Dynamická alokace paměti
- Ukazatel na funkce
- Paměťové třídy
- Volání funkcí
  
- **Příště: Struktury a union, přesnost výpočtu a vnitřní reprezentace číselných typů.**