

# Pole, ukazatel, textový řetězec, vstup a výstup programu

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 05

**B0B36PRP – Procedurální programování**

# Přehled témat

- Část 1 – Pole, ukazatele a řetězce

Pole

Ukazatele

Funkce a předávání parametrů

Vstup a výstup programu

Ukazatele a pole

Textové řetězce

*S. G. Kochan: kapitoly 7, 10, 11*

*P. Herout: kapitola 10, 11, 12, 13*

- Část 2 – Zadání 4. domácího úkolu (HW04)

# Část I

## Pole a ukazatele

# Pole

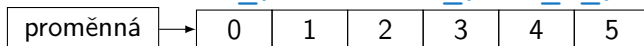
- Datová struktura pro uložení **více hodnot stejného typu**
- Slouží k reprezentaci posloupnosti hodnot v paměti

*Hodnoty uloženy v souvislém bloku paměti*

- Jednotlivé prvky mají identickou velikost a jejich relativní adresa vůči počátku pole je jednoznačně určena
  - Prvky můžeme adresovat pořadím prvku v poli

*Relativní „adresa“ vůči prvnímu prvku*

„adresa“ = velikost\_prvku \* index\_prvku\_v\_poli



- Proměnná typu pole reprezentuje adresu vyhrazeného paměťového prostoru, kde jsou hodnoty uloženy
 

*Adresa\_prvku = adresa\_prvního\_prvku + velikost\_typu \* index\_prvku\_v\_poli*
- Definicí proměnné dochází k alokaci paměti pro uložení definovaného počtu hodnot příslušného typu
- **Velikost pole statické délky nelze měnit**

*Garance souvislého přístupu k položkám pole*

## Definice pole

- Hodnota proměnné typu pole je odkaz (adresa) na místo v paměti, kde je pole uloženo
- Definice proměnné typu pole se skládá z typu prvků, jména proměnné a hranatých závorek []

**typ proměnná [];**

- Závorky [] slouží také k přístupu (adresaci) prvku

**proměnná \_typu \_pole [index \_prvku \_pole]**

Příklad definice proměnné typu pole hodnot typu `int`. Alokace paměti pro až 10 prvků pole.

```
int array[10];
```

*Tj.  $10 \times \text{sizeof}(\text{int})$*

```
printf("Size of array %lu\n", sizeof(array));
printf("Item %i of the array is %i\n", 4, array[4]);
```

```
Size of array 40
Item 4 of the array is -5728
```

*Hodnoty pole nejsou inicializovány!*

## 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**
- **Prvky pole tvoří v paměti souvislou oblast!**
- Velikost pole (v bajtech) je dána počtem prvků pole  $n$  a **typem** prvku, tj.  $n * \text{sizeof}(\text{typ})$
- Textový řetězec je pole typu **char**, kde poslední prvek je **'\0'**

**C nekontroluje za běhu programu, zdali je index platný!**

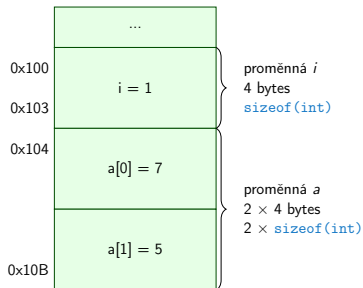
## Pole – Příklad vizualizace alokace přiřazení hodnot

- Proměnná typu pole odkazuje na začátek paměti, kde jsou alokovány jednotlivé prvky pole.
- Přístup k prvkům je prostřednictvím indexového operátoru `[]`, který určí adresu konkrétního prvku pole z typu proměnné.

```

1  int i;
2  int a[2];
3
4  i = 1;
5
6  a[1] = 5;
7  a[0] = 7;

```



*Pro účely vizualizace začíná alokace proměnných na adrese 0x100. Automatické proměnné na zásobníku jsou však zpravidla alokovány od horní adresy k adresám nižším.*

## Pole – Příklad 1/3

- Definice jednorozměrného a **dvourozměrného** pole

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

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

- Přístup k prvkům pole

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

- Příklad definice pole a tisk hodnot prvků

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int array[5];
6
7     printf("Size of array: %lu\n", sizeof(array));
8     for (int i = 0; i < 5; ++i) {
9         printf("Item[%i] = %i\n", i, array[i]);
10    }
11    return 0;
12 }
```

Size of array: 20  
Item[0] = 1  
Item[1] = 0  
Item[2] = 740314624  
Item[3] = 0  
Item[4] = 0

lec05/array.c



## Pole – Příklad 2/3

### ■ Příklad definice pole

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int array[10];
6
7      for (int i = 0; i < 10; i++) {
8          array[i] = i;
9      }
10
11     int n = 5;
12     int array2[n * 2];
13
14     for (int i = 0; i < 10; i++) {
15         array2[i] = 3 * i - 2 * i * i;
16     }
17
18     printf("Size of array: %lu\n", sizeof(array));
19     for (int i = 0; i < 10; ++i) {
20         printf("array[%i]=%+2i \t array2[%i]=%6i\n", i,
21             array[i], i, array2[i]);
22     }
23     return 0;
}
```

Size of array: 40

array[0]=+0	array2[0]=	0
array[1]=+1	array2[1]=	1
array[2]=+2	array2[2]=	-2
array[3]=+3	array2[3]=	-9
array[4]=+4	array2[4]=	-20
array[5]=+5	array2[5]=	-35
array[6]=+6	array2[6]=	-54
array[7]=+7	array2[7]=	-77
array[8]=+8	array2[8]=	-104
array[9]=+9	array2[9]=	-135

lec05/demo-array.c

## Pole – Příklad 3/3

### ■ Příklad definice pole s inicializací

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int array[5] = {0, 1, 2, 3, 4};
6
7      printf("Size of array: %lu\n", sizeof(array));
8      for (int i = 0; i < 5; ++i) {
9          printf("Item[%i] = %i\n", i, array[i]);
10     }
11     return 0;
12 }
```

Size of array: 20  
Item[0] = 0  
Item[1] = 1  
Item[2] = 2  
Item[3] = 3  
Item[4] = 4

lec05/array-init.c

### ■ Inicializace pole

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

## Pole variabilní délky

- C99 umožňuje definovat tzv. pole variabilní délky – délka pole je určena za běhu programu

*V předchozích verzích bylo nutné znát délku při kompilaci.*

- Délka pole tak může např. být argument funkce

```
void fce(int n)
{
    // int local_array[n] = { 1, 2 }; inicializace není dovolena
    int local_array[n]; // variable length array

    printf("sizeof(local_array) = %lu\n", sizeof(local_array));
    printf("length of array = %lu\n", sizeof(local_array) / sizeof(int));
    for (int i = 0; i < n; ++i) {
        local_array[i] = i * i;
    }
}

int main(int argc, char *argv[])
{
    fce(argc);
    return 0;
}
```

lec05/fce\_var\_array.c

- Pole variabilní délky však nelze v definici inicializovat

## Pole ve funkci a jako argument funkce

- Lokálně definované pole ve funkci má rozsah platnosti pouze v rámci funkce (bloku)

```
void fce(int n)
{
    int array[n];
    // počítání s array
    {
        int array2[n*2];
    } // po skončení bloku array2 automaticky zaniká
    // zde již není array2 přístupné
} // po skončení funkce, pole array automaticky zaniká
```

- Pole je automaticky vytvořeno a po skončení bloku (funkce) automaticky zaniká (paměť je uvolněna) *Více o paměťových třídách na 6. přednášce*
  - Lokální proměnné jsou ukládány na tzv. zásobník, který má zpravidla relativně malou velikost, proto pro velká pole může být vhodnější alokovat paměť dynamicky a použít **ukazatele**
- Pole může být argumentem funkce

```
void fce(int array[]);
```

hodnota je však předávána jako **ukazatel!**

## Ukazatel (pointer)

- Ukazatel (pointer) je proměnná jejíž **hodnota je adresa** paměti jiné proměnné
- Pointer *odkazuje* na jinou proměnnou  
*Odkazuje na oblast paměti, kde je uložena hodnota proměnné*
- Ukazatel má typ** proměnné, na kterou může ukazovat

*Důležité pro ukazatelovou aritmetiku*

- Ukazatel na hodnoty (proměnné) základních typů: `char`, `int`, ...
- „Ukazatel na pole“; ukazatel na funkci; **ukazatel na ukazatele**
- Ukazatel může být též bez typu (**void**)
  - Velikost proměnné nelze z vlastnosti ukazatele určit
  - Pak může obsahovat adresu libovolné proměnné
- Prázdná adresa ukazatele je definovaná hodnotou konstanty **NULL**  
Textová konstanta (makro) preprocesoru definovaná jako „*null pointer constant*“

**C99 – lze též použít „int“ hodnotu 0**

**C za běhu programu nekontroluje platnost adresy (hodnoty) ukazatele.**

*Ukazatele umožňují psát efektivní kódy, při neobezřetném používání mohou vést k chybám. Proto je důležité osvojit si princip nepřímého adresování a pochopit organizaci a přístup do paměti.*

## Referenční a dereferenční operátor

### ■ Referenční operátor – &

- Vrací adresu paměti, kde je uložena hodnota proměnné, před kterou je uveden **&proměnná**

### ■ Dereferenční operátor – \*

- Vrací **l-hodnotu** (*l-value*) odpovídající hodnotě na adrese ukazatele **\*proměnná\_typu\_ukazatel**

- Umožňuje číst a zapisovat hodnotu na adrese dané obsahem ukazatele, např. ukazatel na hodnotu typu `int` (tj. `int *p`)

```
*p = 10; // zápis hodnoty 10 na adresu uloženou v proměnné p
```

```
int a = *p; // čtení hodnoty z adresy uložené v p
```

- Pro tisk hodnoty ukazatele (adresy) lze ve funkci `printf()` použít řídicí řetězec `"%p"`

```
int a = 10;
int *p = &a;
```

```
printf("Value of a %i, address of a %p\n", a, &a);
printf("Value of p %p, address of p %p\n", p, &p);
```

```
Value of a 10, address of a 0x7fffffff95c
```

```
Value of p 0x7fffffff95c, address of p 0x7fffffff950
```

## Proměnné typu ukazatel (pointer) – příklady

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

int *pi;    /* definice promenne typu pointer
            pi -- pointer na promenu 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 */
```

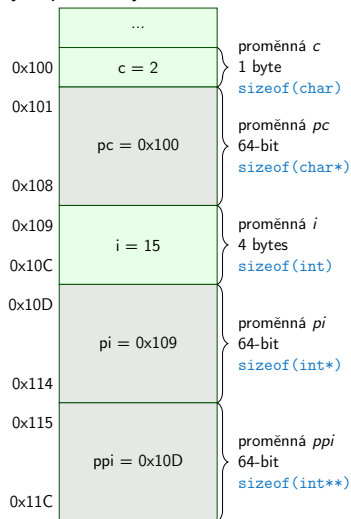
# Ukazatele – Příklad vizualizace alokace přiřazení hodnot

- Ukazatele jsou proměnné, které uchovávají adresy jiných proměnných

```

1 char c;
2
3 c = 10;
4
5 char *pc;
6
7 pc = &c;
8
9 int i = 17;
10 int *pi = &i;
11
12 *pi = 15;
13 *pc = 2;
14
15 int **ppi = &pi;

```



Pro účely vizualizace začíná alokace proměnných na adrese 0x100. Automatické proměnné na zásobníku jsou však zpravidla alokovány od horní adresy k adresám nižším.



## Ukazatel (pointer) – příklady 2/2

```
printf("i: %d -- pi: %p\n", i, pi); // 10 0x7fffffff8fc
printf("&i: %p -- *pi: %d\n", &i, *pi); // 0x7fffffff8fc
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): %lu\n", sizeof(i)); // 4
printf("sizeof(pi): %lu\n", sizeof(pi)); // 8

long l = (long)pi;
printf("0x%lx %p\n", l, pi); /* print l as hex -- %lx */
// 0x7fffffff8fc 0x7fffffff8fc

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

lec05/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



## 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 funkcí

*Více o volání funkcí a paměti v 6. přednášce*

```
void fce(int a, char *b)
{ /*
  a - je lokální proměna typu int (uložena na zásobníku)
  b - je lokální proměna typu ukazatel na proměnou
      typu char (hodnota je adresa a je také na zásobníku)
  */
}
```

- 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 a: %d b: %c\n", a, b);
fce(a, &b);
printf("After call a: %d b: %c\n", a, b);
```

- Výstup

Before call a: 10 b: A

After call a: 10 b: B

[lec05/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>`.*

*lec05/main\_env.c*

- Rozšířená funkce o specifické parametry Mac OS X

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

## Argumenty funkce `main`

- Základní tvar funkce `main`

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

- `argc` – obsahuje počet argumentů programu

*Včetně jména spouštěného programu*

- Argumenty jsou textové řetězce oddělené mezerou (bílým znakem)

- `argv` – pole ukazatelů na hodnoty typu `char`

*Typ „čteme“ zprava doleva*

- Pole `argv` má velikost (počet prvku) daný hodnotou `argc`
- Každý prvek pole `argv[i]` obsahuje adresu, kde je uložen textový řetězec argumentu (tj. typ `char*`)
- Textový řetězec (argument) je posloupnost znaků (typ `char`) zakončený znakem `'\0'`.  
*„null character“ – konec textového řetězce*
- Alokace paměti pro uložení argumentů (textových řetězců) je provedena při spuštění programu

*V případě programu pro OS zajišťuje zavaděč programu („loader“) a standardní knihovna C.*

## Předávání parametrů programu

- Při spuštění programu můžeme předat parametry programu prostřednictvím argumentů

```

1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("Number of arguments %i\n", argc);
6     for (int i = 0; i < argc; ++i) {
7         printf("argv[%i] = %s\n", i, argv[i]);
8     }
9     return argc > 0 ? 0 : 1;
10 }

```

clang demo-arg.c -o arg  
./arg one two three  
Number of arguments 4  
argv[0] = ./arg  
argv[1] = one  
argv[2] = two  
argv[3] = thre  
lec05/demo-arg.c

- Voláním `return` ve funkci `main()` vracíme z programu návratovou hodnotu, se kterou můžeme dále pracovat

*Např. v interpretu příkazů (shellu).*

```
./arg >/dev/null; echo $?
```

```
1
```

```
./arg first >/dev/null; echo $?
```

```
0
```

- Návratová hodnota programu je uložena v proměnné `$?`, kterou lze vypsát příkazem `echo`
- `>/dev/null` přesměruje standardní výstup do `/dev/null`



## Interakce programu s uživatelem

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

- Při spuštění programu lze předat parametry (textové řetězce)
- Při ukončení programu lze předat návratovou hodnotu

*Konvence 0 bez chyb, ostatní hodnoty chybový kód*

- Při běhu programu lze číst ze standardního vstupu a zapisovat na standardní výstup

*Např. `scanf()` nebo `printf()`*

- Při spuštění programu lze vstup i výstup přeměřovat z/do souboru  
*Program tak nečeká na vstup uživatele (stisk klávesy „Enter“)*

- Každý program (terminálový) má standardní vstup (`stdin`) a výstup (`stdout`) a dále pak standardní chybový výstup (`stderr`), které lze v shellu přeměřovat

```
./program <stdin.txt >stdout.txt 2>stderr.txt
```

- Alternativou k `scanf()` a `printf()` lze využít `fscanf()` a `fprintf()`.
  - Funkce mají první argument soubor jinak, je syntax identická
  - Soubory `stdin`, `stdout` a `stderr` jsou definována v `<stdio.h>`

## Příklad programu s výstupem na stdout a přesměrováním

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int ret = 0;
6
7      fprintf(stdout, "Program has been called as %s\n", argv[0]);
8      if (argc > 1) {
9          fprintf(stdout, "1st argument is %s\n", argv[1]);
10     } else {
11         fprintf(stdout, "1st argument is not given\n");
12         fprintf(stderr, "At least one argument must be given!\n");
13         ret = -1;
14     }
15     return ret;
16 }

```

lec05/demo-stdout.c

- Příklad výstupu – `clang demo-stdout.c -o demo-stdout`

```

./demo-stdout; echo $?
Program has been called as ./demo-
stdout
1st argument is not given
At least one argument must be given!
255

```

```

./demo-stdout 2>stderr
Program has been called as ./
demo-stdout
1st argument is not given

./demo-stdout ARGUMENT 1>
stdout; echo $?
0

```

## Pointery a pole

- 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 kompilace funkce s předáváním pole 1/2

- Argument funkce je pole

```
1 void fce(int array[])
2 {
3     int local_array[] = { 2, 4, 6 };
4     printf("sizeof(array) = %lu -- sizeof(local_array) = %lu\n",
5         sizeof(array), sizeof(local_array));
6     for (int i = 0; i < 3; ++i) {
7         printf("array[%i]=%i local_array[%i]=%i\n", i, array[i],
8             i, local_array[i]);
9     }
10    ...
11    int array[] = { 1, 2, 3 };
12    fce(array);
```

lec05/fce\_array.c

- Po překladu (`gcc -std=c99`) na `amd64`
  - `sizeof(array)` vrátí velikost **8 bajtů** (64-bitová adresa)
  - `sizeof(local_array)` vrátí velikost **12 bajtů** (3×4 bajty – `int`)
- Pole se funkcím předává jako ukazatel na první prvek

## Příklad kompilace funkce s předáváním pole 2/2

- Kompilátor `clang` (ve výchozím nastavení) upozorňuje na záměnu `int*` za `int[]`

```
clang fce_array.c
fce_array.c:7:16: warning: sizeof on array function parameter
will return size
of 'int *' instead of 'int []' [-Wsizeof-array-argument]
    sizeof(array), sizeof(local_array));
           ^
fce_array.c:3:14: note: declared here
void fce(int array[])
           ^
1 warning generated.
```

lec05/fce\_array.c

- Program lze zkompileovat, ale nelze se spoléhat na velikost `sizeof`
- Ukazatel nenese informaci o velikosti alokované paměti!

*Pole ano „hlídá za nás kompilátor“.*

# 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[0]` přímo reprezentuje hodnotu na adrese `0x10`.

- Hodnota `p` je adresa `0x10`, kde je uložena hodnota 1. prvku pole.

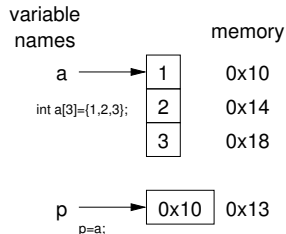
- Přřazení `p = a` je legitimní

*Kompilátor zajistí přřazení adresy prvního prvku do ukazatele.*

- Přístup k 2. prvku lze použít jak `a[1]` tak `p[1]`

- Oběma přístupy se dostaneme na příslušné prvky pole, způsob je však odlišný — Ukazatele využívají tzv. pointerovou aritmetiku

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



## Ukazatelová (pointerová) aritmetika

- S ukazateli (pointery) lze provádět aritmetické operace  $+$  a  $-$ , tj. přičítat nebo odčítat celé číslo
  - `ukazatel = ukazatel stejného typu + (nebo -) a celé číslo (int)`
  - Nebo lze používat zkrácený zápis např. `ukazatel += 1` a unární operátory např. `ukazatel++`
- 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 `lec05/pointers_and_array.c`

## Příklad ukazatele a pole

```
1 int a[] = { 1, 2, 3, 4 };
2 int b[] = { [3] = 10, [1] = 1, [2] = 5, [0] = 0 }; //initialization
3
4 // b = a; It is not possible to assign arrays
5 for (int i = 0; i < 4; ++i) {
6     printf("a[%i] =%3i    b[%i] =%3i\n", i, a[i], i, b[i]);
7 }
8
9 int *p = a; //you can use *p = &a[0], but not *p = &a
10 a[2] = 99;
11
12 printf("\nPrint content of the array 'a' with pointer arithmetic\n");
13 for (int i = 0; i < 4; ++i) {
14     printf("a[%i] =%3i    p+%i =%3i\n", i, a[i], i, *(p+i));
15 }

a[0] = 1    b[0] = 0
a[1] = 2    b[1] = 1
a[2] = 3    b[2] = 5
a[3] = 4    b[3] = 10

Print content of the array 'a' using pointer arithmetic
a[0] = 1    p+0 = 1
a[1] = 2    p+1 = 2
a[2] = 99   p+2 = 99
a[3] = 4    p+3 = 4
```

lec05/array\_pointer.c



## Příklad předání ukazatele na pole

- Předáním pole jako ukazatele nemáme informaci o počtu prvků
- Proto můžeme explicitně předat počet prvků v proměnné `n`

```
1 #include <stdio.h>
2
3 void fce(int n, int *array) // array je lokální proměnná
4 { // typu ukazatel, můžeme změnit obsah paměti proměnné
5     // definované v main()
6     int local_array[] = {2, 4, 6};
7     printf("sizeof(array) = %lu, n = %i -- sizeof(local_array) =
8         %lu\n",
9         sizeof(array), n, sizeof(local_array));
10    for (int i = 0; i < 3 && i < n; ++i) { //testujeme take n!
11        printf("array[%i]=%i local_array[%i]=%i\n", i, array[i],
12            i, local_array[i]);
13    }
14 }
15
16 int main(void)
17 {
18     int array[] = {1, 2, 3};
19     fce(array, sizeof(array)/sizeof(int)); // pocet prvku
20     return 0;
21 }
22
23                                     lec05/fce_pointer.c
```

- Přes ukazatel `array` v `fce()` máme přístup do pole z `main()`

## Příklad předání pole včetně velikosti využitím VLA

- **VLA (Variable Length Array)** – délka pole určena za běhu programu  
*Neznamená to, že můžeme měnit velikost již jednou alokovaného pole, ale že velikost při alokaci může být určena za běhu programu, např. výpočtem, uživatelským vstupem atd.*
- Při předání pole můžeme rovnou použít pro definici délky pole
- **Pole je však stále předáváno jako ukazatel**, získáváme tak především přehlednost kódu

```
1 void print_array(int n, int a[n])
2 {
3     printf("Size of the array a is %lu\n", sizeof(a));
4     for (int i = 0; i < n; ++i) {
5         printf("array[%i]=%i\n", i, a[i]);
6     }
7 }
8
9 int main(int argc, char *argv[])
10 {
11     int n = 10;
12     if (argc > 1 && sscanf(argv[1], "%d", &n) != 1) {
13         fprintf(stderr, "Warning: cannot parse number from argv[1]!\n");
14     }
15     printf("Size of the array is %lu\n", sizeof(array));
16     int array[n]; //vla array size depends on n
17     for (int i = 0; i < n; ++i) {
18         array[i] = 2*i;
19     }
20     print_array(n, array);
21     return 0;
22 }
```

lec05/fce\_vla.c

## Vícerozměrná pole

- Pole můžeme definovat jako vícerozměrná, např. 2D matice

```

int m[3][3] = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
};
printf("Size of m: %lu == %lu\n", sizeof(m), 3*3*sizeof(int));

for (int r = 0; r < 3; ++r) {
    for (int c = 0; c < 3; ++c) {
        printf("%3i", m[r][c]);
    }
    printf("\n");
}

```

lec05/matrix.c

## Vícerozměrná pole a vnitřní reprezentace

- Vícerozměrné pole je **vždy** souvislý blok paměti

*Např. `int a[3][3]`; reprezentuje alokovanou paměť o velikosti `9*sizeof(int)`, tj. zpravidla 36 bytů. Operátor `[]` nám tak převede vším zjednodušuje zápis programu.*

```
int *pm = (int *)m; // ukazatel na souvislou oblast m
printf("m[0][0]=%i m[1][0]=%i\n", m[0][0], m[1][0]); // 1 4
printf("pm[0]=%i pm[3]=%i\n", m[0][0], m[1][0]); // 1 4
```

`lec05/matrix.c`

- Dvourozměrné pole lze také definovat jako ukazatel na ukazatele (pole ukazatelů) na hodnoty konkrétního typu, např.
  - `int **a`; – ukazatel na ukazatele
  - V obecném případě však takový ukazatel nemusí odkazovat na souvislou oblast, kde jsou alokovány jednotlivé prvky.
  - Proto při přístupu jako do jednorozměrného pole `int *b = (int *)a`; nelze garantovat přístup do druhého řádku jako v přechodícím příkladě.

## Pole a vícerozměrná pole jako parametr funkce

- **Parametr funkce je ukazatel na pole**, např. typu `int`

```
int (*p)[3] = m; // pointer to array of int           Size of p: 8
                                                    Size of *p: 12
printf("Size of p: %lu\n", sizeof(p));
printf("Size of *p: %lu\n", sizeof(*p)); // 3 * sizeof(int) = 12
```

- Funkci nelze deklarovat s argumentem typu `[] []` např.

```
int fce(int a[] []);
```

neboť kompilátor nemůže určit adresu pro přístup na `a[i][j]`,  
neboť se používá adresová aritmetika odpovídající 2D poli

Pro `int m[row][col]` totiž `m[i][j]` odpovídá hodnotě na adrese `*(m + col * i + j)`

- Je však možné funkci deklarovat například jako
  - `int g(int a[]);` což odpovídá deklaraci `int g(int *a);`
  - `int fce(int a[][13]);` – je znám počet sloupců
  - nebo `int fce(int a[3][3]);`

## Inicializace pole

- Při definici můžeme hodnoty prvků pole inicializovat postupně nebo indexovaně

*2D pole jsou inicializována po řádcích*

- Při částečné inicializaci jsou ostatní prvky nastaveny na 0

```
#define ROWS 3
#define COLS 3
void print(int rows, int cols, int m[rows][cols])
{
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            printf("%4i", m[r][c]);
        }
        printf("\n");
    }
}

int m0[ROWS][COLS];
int m1[ROWS][COLS] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int m2[ROWS][COLS] = { 1, 2, 3 };
int m3[ROWS][COLS] =
    { [0][0] = 1, [1][1] = 2, [2][2] = 3 };

print(ROWS, COLS, m0);
print(ROWS, COLS, m1);
print(ROWS, COLS, m2);
print(ROWS, COLS, m3);
```

m0 - not initialized  
-584032767743694227  
0 1 0  
740314624 0 0

m1 - init by rows  
1 2 3  
4 5 6  
7 8 9

m2 - partial init  
1 2 3  
0 0 0  
0 0 0

m3 - indexed init  
1 0 0  
0 2 0  
0 0 3

lec05/array-inits.c

## 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 zakončena znakem `'\0'`

Např. řetězcová konstanta "word" je uložena jako

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

*Pole tak musí být vždy o 1 položku delší než je vlastní text!*

## Textový řetězec

- Textový řetězec můžeme inicializovat jako pole znaků, tj. `char []`

```
char str[] = "123";
char s[] = {'5', '6', '7' };

printf("Size of str %lu\n", sizeof(str));
printf("Size of s %lu\n", sizeof(s));
printf("str '%s'\n", str);
printf(" s '%s'\n", s);
```

Size of str 4  
Size of s 3  
str '123'  
s '567123'

lec05/array\_str.c

- Pokud není řetězec zakončen znakem `'\0'`, jako v případě proměnné `char s[]`, pokračuje výpis řetězce až do nejbližšího znaku `'\0'`**
- Na textový řetězec lze odkazovat ukazatelem na znak `char*`

```
char *sp = "ABC";
printf("Size of ps %lu\n", sizeof(sp));
printf(" ps '%s'\n", sp);
```

Size of ps 8  
ps 'ABC'

- Velikost ukazatele je 8 bytů (pro 64-bit OS)
- Textový řetězec musí být zakončen znakem `'\0'`

*Alternativně lze řešit vlastní implementací s explicitním uložením délky řetězce*



## Načítání textových řetězců

- Správnost alokace vstupních argumentů je zajištěna při spuštění

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

- Načtení textového řetězce funkcí `scanf()`

- Použitím `%s` může dojít k přepisu paměti

```
char str0[4] = "PRP"; // +1 \0
char str1[5]; // +1 for \0
printf("String str0 = '%s'\n", str0);
printf("Enter 4 chars: ");
scanf("%s", str1);
printf("You entered string '%s'\n", str1);
printf("String str0 = '%s'\n", str0);
```

Příklad výstupu programu:

```
String str0 = 'PRP'
```

```
Enter 4 chars: 1234567
```

```
You entered string '1234567'
```

```
String str0 = '67'
```

`lec05/str_scanf-bad.c`

- Načtení maximálně 4 znaků zajistíme řídicím řetězcem `"%4s"`

```
char str0[4] = "PRP";
char str1[5];
...
scanf("%4s", str1);
printf("You entered string '%s'\n", str1);
printf("String str0 = '%s'\n", str0);
```

Příklad výstupu programu:

```
String str0 = 'PRP'
```

```
Enter 4 chars: 1234567
```

```
You entered string '1234'
```

```
String str0 = 'PRP'
```

`lec05/str_scanf-limit.c`

## Zjištění délky textového řetězce

- Textový řetězec v C je pole (`char[]`) nebo ukazatel (`char*`) odkazující na část paměti, kde je uložena příslušná posloupnost znaků.
- Textový řetězec je zakončen znakem `'\0'`
- Délku textového řetězce lze zjistit sekvenčním procházením znak po znaku až k `'\0'`

```
int getLength(char *str)
{
    int ret = 0;
    while (str && (*str++) != '\0') {
        ret += 1;
    }
    return ret;
}
```

```
for (int i = 0; i < argc; ++i) {
    printf("argv[%i]: getLength = %i -- strlen = %lu\n",
        i, getLength(argv[i]), strlen(argv[i]));
}
```

`lec05/string_length.c`

*Nebo jen `while (*str++) ret +=1;`*

- Funkce pro práci s řetězci jsou ve standardní knihovně `<string.h>`
- Délka řetězce – `strlen()`
- **Dotaz na délku řetězce má lineární složitost  $O(n)$ .**

## Práce s textovými řetězci

- V C jsou řetězce pole znaků zakončené znakem `'\0'`
- Základní operace jsou definovány v knihovně `<string.h>`, například pro kopírování nebo porovnání řetězců
  - `char* strcpy(char *dst, char *src);`
  - `int strcmp(const char *s1, const char *s2);`
  - Funkce předpokládají dostatečný rozsah alokovaných polí
  - Funkce s explicitním limitem na maximální délku řetězců: `char* strncpy(char *dst, char *src, size_t len); int strncmp(const char *s1, const char *s2, size_t len);`
- Převod řetězce na číslo – `<stdlib.h>`
  - `atoi()`, `atof()` – převod celého a necelého čísla
  - `long strtol(const char *nptr, char **endptr, int base);`
  - `double strtod(const char *nptr, char **restrict endptr);`  
Funkce `atoi()` a `atof()` jsou „*obsolete*“, ale mohou být rychlejší
  - Alternativně také např. `sscanf()`  
Více viz `man strcpy`, `strcmp`, `strtol`, `strtod`, `sscanf`

## Část II

### Část 2 – Zadání 4. domácího úkolu (HW04)

# Shrnutí přednášky

## Diskutovaná témata

- Jednorozměrná a vícerozměrná pole a jejich inicializace
- Ukazatel
- Textový řetězec
- Rozdíl mezi polem a ukazatelem
- Předávání polí funkcím
- Vstup a výstup programu - argumenty programy a návratová hodnota
  
- **Příště: Ukazatele, paměťové třídy a volání funkcí**