

# Struktury a uniony

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 06

**B0B36PRP – Procedurální programování**

# Přehled témat

- Část 1 – Struktury a uniony

Struktury – `struct`

Proměnné se sdílenou pamětí – `union`

Příklady

*S. G. Kochan: kapitola 9 a 17*

- Část 2 – Zadání 6. domácího úkolu (HW06)

*Appendix – Kódovací příklady*

# Část I

## Část 1 – Struktury a uniony

## Struktura – struct

- Struktura je konečná množina prvků (proměnných), které nemusí být stejného typu.
- Skladba struktury je definovaná uživatelem jako nový typ sestavený z již definovaných typů.
- K prvkům struktury **přístupujeme tečkovou notací**, např. `struct_proměnná.prvek`.
- K prvkům můžeme přistupovat přes ukazatel operátorem `->`, např.  
`proměnná_typu_ukazatel_na_struct->prvek`.
- **Pro struktury stejného typu je definován operátor přiřazení.**  
`var_struct1 = var_struct2;`
- Struktury (jako celek) **nelze** porovnávat relačním operátorem `==`.
- Struktura může být funkci předávána hodnotou i ukazatelem.
- Struktura může být návratovou hodnotou funkce.

## Příklad struct – Definice

- Bez zavedení nového typu (`typedef`) je nutné před identifikátor jména struktury uvádět klíčové slovo `struct`.
- Jméno struktury je ve jmenném prostoru složených typů (struktur).

```
1 struct record {
2     int number;
3     double value;
4 };

1 typedef struct {
2     int n;
3     double v;
4 } item;

1 record r; /* IT IS NOT ALLOWED! */
2         /* Type record is not known */
4 struct record r; /* Keyword struct is required */
5 item i;        /* type item defined using typedef */
```

- Zavedením nového typu `typedef` používáme definovaný typ a nemusíme používat (a ani definovat) jméno struktury. `lec06/struct.c`

## Definice jména struktury a typu struktury

- Uvedením `struct record` zavádíme nové jméno struktury `record`.

```
1 struct record {  
2     int number;  
3     double value;  
4 };
```

- Definujeme identifikátor `record` ve jmeném prostoru struktur.

- Definicí typu `typedef` zavádíme nové jméno typu `record`.

```
1 typedef struct record record;
```

- Definujeme globální identifikátor `record` jako jméno typu `struct record`.

- Obojí můžeme kombinovat v jediné definici jména a typu struktury.

```
1 typedef struct record {  
2     int number;  
3     double value;  
4 } record;
```

```
1 typedef struct record_struct_name {  
2     int number;  
3     double value;  
4 } record_type;
```

## Příklad struct – Inicializace

- Struktury:

```
1 struct record {
2     int number;
3     double value;
4 };
```

```
1 typedef struct {
2     int n;
3     double v;
4 } item;
```

- Proměnné typu struktura můžeme inicializovat prvek po prvku.

```
1 struct record r;
2 r.value = 21.4;
3 r.number = 7;
```

- Podobně jako pole lze inicializovat přímo při definici

```
1 item i = { 1, 2.3 };
```

- nebo pouze konkrétní položky (ostatní jsou nulovány).

```
1 struct record r2 = { .value = 10.4 };
```

[lec06/struct.c](#)

## Příklad struct jako parametr funkce

- Struktury můžeme předávat jako parametry funkcí hodnotou.

```
1 void print_record(struct record rec) {
2     printf("record: number(%d), value(%lf)\n",
3         rec.number, rec.value);
4 }
```

- Nebo hodnotou ukazatele

```
1 void print_item(item *v) {
2     printf("item: n(%d), v(%lf)\n", v->n, v->v);
3 }
```

- Při předávání parametru

- **hodnotou** se vytváří nová proměnná a původní obsah předávané struktury se kopíruje na zásobník (pro složený typ je definován operátor přiřazení);
- **hodnotou ukazatele** se kopíruje pouze hodnota ukazatele (adresa) a pracujeme tak s původní strukturou.

[lec06/struct.c](#)

## Složený typ, operátor přiřazení a pole jako prvek složeného typu 1/2

- Velikost složeného typu musí být známa během překlady, proto můžeme mít definovaný operátor přiřazení.  
*Nebo naopak, abychom mohli jednoduše přiřazovat, tak potřebujeme znát velikost typu.*
- Prvek složeného typu může být pole (definované velikosti) nebo ukazatel.

```
1 void print(const char *str, int n, int *a);    19     for (int i = 0; i < n; ++i) {
3 #define N 10 // We need named literal.      20         s1.a[i] = n - i;
5 int main(void)                               21     }
6 {                                             22     print("s1.a", n, s1.a);
7     const int n = N;                         23     print("s2.a", n, s2.a);
8     struct { // Anonymous struct            24     return 0;
9         int a[N]; // Defined size, no VLA    25 } // end main()
10 } s1, s2; // Two struct variables           27 void print(const char *str, int n, int *a) {
12 printf("s1 %p; s2 %p\n", &s1, &s2);         28     printf("%s %p: ", str, a);
13 for (int i = 0; i < n; ++i) {              29     for (int i = 0; i < n; ++i) {
14     s1.a[i] = i;                             30         printf("%d%s", a[i], i < (n-1) ? ", " : "\n");
15 }                                             31     }
16 print("s1.a", n, s1.a);                     32 }
17 s2 = s1; // Assignment
18 print("s2.a", n, s2.a);
```

lec06/demo-struct\_array.c

## Složený typ, operátor přiřazení a pole jako prvek složeného typu 2/2

### Příklad `lec06/demo-struct_array.c`

- Používáme anonymní složený typ - definice strukturu přímo v definici proměnných `s1` a `s2`.
- Musíme použít textový literál pro definici velikosti položky `a` jako pole definované délky.
- Ve funkci `print()` tiskneme hodnotu adresy, kde je alokované pole.

*V našem případě se shoduje s adresou, kde je struktura uložena. Struktura je „organizovaný“ pohled na blok paměti důležitý zejména pro zpřehlednění programu. Při běhu programu vlastně není nutné mít v paměti dílčí jména prvku složeného typu.*

```
s1 0x7fffffff840; s2 0x7fffffff818
s1.a 0x7fffffff840: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
s2.a 0x7fffffff818: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
s1.a 0x7fffffff840: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
s2.a 0x7fffffff818: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

- V příkladu si vyzkoušejte chování překladače a programu v případě použití VLA nebo konstantní proměnné definující velikost pole.
- Pole definované velikosti nahraďte dynamicky alokovaným polem.

## Příklad struct – Přiřazení

- Hodnoty proměnné **stejného typu** struktury můžeme přiřadit operátorem =.

```
1 struct record {
2     int number;
3     double value;
4 };
1 typedef struct {
2     int n;
3     double v;
4 } item;
```

```
1 struct record rec1 = { 10, 7.12 };
2 struct record rec2 = { 5, 13.1 };
3 item i;
4 print_record(rec1); /* number(10), value(7.120000) */
5 print_record(rec2); /* number(5), value(13.100000) */
6 rec1 = rec2;
7 i = rec1; /* IT IS NOT ALLOWED! */
8 // Different types, albeit with the same memory representation.
9 print_record(rec1); /* number(5), value(13.100000) */
```

lec06/struct.c

## Příklad struct – Přímá kopie paměti

- Jsou-li dvě struktury stejně veliké, můžeme přímo kopírovat obsah příslušné paměťové oblasti.

*Například funkcí `memcpy()` z knihovny `string.h`*

```
1 struct record r = { 7, 21.4};
2 item i = { 1, 2.3 };
3 print_record(r); /* number(7), value(21.400000) */
4 print_item(&i); /* n(1), v(2.300000) */
5 if (sizeof(i) == sizeof(r)) {
6     printf("i and r are of the same size\n");
7     memcpy(&i, &r, sizeof(i));
8     print_item(&i); /* n(7), v(21.400000) */
9 }
```

- V tomto případě je interpretace hodnot v obou strukturách identická, obecně tomu však být nemusí. Například v případě změny pořadí prvků typu `int` a `double`.

`lec06/struct.c`

## Struktura struct a velikost

- Vnitřní reprezentace struktury nutně nemusí odpovídat součtu velikostí jednotlivých prvků.

```
1 struct record {
2     int number;
3     double value;
4 };
1 typedef struct {
2     int n;
3     double v;
4 } item;
```

```
1 printf("Size of int: %lu size of double: %lu\n", sizeof(int), sizeof(double));
2 printf("Size of record: %lu\n", sizeof(struct record));
3 printf("Size of item: %lu\n", sizeof(item));
```

```
Size of int: 4 size of double: 8
Size of record: 16
Size of item: 16
```

lec06/struct.c

## Struktura struct a velikost 1/2

- Při kompilaci zpravidla dochází k zarovnání prvků na velikost slova příslušné architektury.

*Např. 8 bytů v případě 64-bitové architektury.*

*Jednotlivé prvky jsou na adrese v násobNapř. 8 bytů v případě 64-bitové architektury.*

- Můžeme explicitně předepsat kompaktní paměťovou reprezentaci, např. direktivou `__attribute__((packed))` překladačů `clang` a `gcc`.

```
1 struct record_packed {
2     int n;
3     double v;
4 } __attribute__((packed));
```

`lec06/struct.c`

## Struktura struct a velikost 2/2

### ■ Nebo

```
1 typedef struct __attribute__((packed)) {
2     int n;
3     double v;
4 } item_packed;
```

### ■ Příklad výstupu:

```
1 printf("Size of int: %lu size of double: %lu\n", sizeof(int), sizeof(double));
2 printf("record_packed: %lu\n", sizeof(struct record_packed));
3 printf("item_packed: %lu\n", sizeof(item_packed));
```

Size of int: 4 size of double: 8

Size of record\_packed: 12

Size of item\_packed: 12

[lec06/struct.c](#)

### ■ Zarovnání zpravidla přináší rychlejší přístup do paměti, ale zvyšuje paměťové nároky.

<http://www.catb.org/esr/structure-packing>

<https://stackoverflow.com/questions/4306186/structure-padding-and-packing>

## Proměnné se sdílenou pamětí – union

- **Union** je uživatelsky definovaný typ, který sdružuje více již existujících typů do jedné paměťové oblasti
- Prvky unionu sdílejí společně stejná paměťová místa. *Překrývají se*
- Velikost unionu je dána velikostí největšího z jeho prvků.
- K prvkům unionu se přistupuje tečkovou notací.
- Pokud nedefinujeme nový typ, je nutné k identifikátoru proměnné unionu uvádět klíčové slovo `union`. *Podobně jako u struktury `struct`.*

```
1 union Nums {
2     char c;
3     int i;
4 };
5 Nums nums; /* THIS IS NOT ALLOWED! Type Nums is not known! */
6 union Nums nums;
```

- Používá se v návrhovém vzoru **tagged union**.

*Např. měření sensorických dat, která mohou mít různý význam, počet průchodů nebo teplota, ale pořád se jedná o sensorická měření.*

## Příklad union 1/2

- Union složený z proměnných typu: `char`, `int` a `double`.

```
1  int main(int argc, char *argv[])
2  {
3      union Numbers {
4          char c;
5          int i;
6          double d;
7      };
8      printf("size of char %lu\n", sizeof(char));
9      printf("size of int %lu\n", sizeof(int ));
10     printf("size of double %lu\n", sizeof(double));
11     printf("size of Numbers %lu\n", sizeof(union Numbers));
12
13     union Numbers numbers;
14
15     printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);
```

- Příklad výstupu.

```
size of char 1
size of int 4
size of double 8
size of Numbers 8
Numbers c: 48 i: 740313136 d: 0.000000
```

lec06/union.c

## Příklad union 2/2

- Proměnné sdílejí paměťový prostor.

```
1 numbers.c = 'a';
2 printf("\nSet the numbers.c to 'a'\n");
3 printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);
4
5 numbers.i = 5;
6 printf("\nSet the numbers.i to 5\n");
7 printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);
8
9 numbers.d = 3.14;
10 printf("\nSet the numbers.d to 3.14\n");
11 printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);
```

- Příklad výstupu

```
Set the numbers.c to 'a'
Numbers c: 97 i: 1374389601 d: 3.140000

Set the numbers.i to 5
Numbers c: 5 i: 5 d: 3.139999

Set the numbers.d to 3.14
Numbers c: 31 i: 1374389535 d: 3.140000
```

## Příklad IPv4 adresy jako číslo nebo čtveřice bajtů

- IPv4 adresa je čtveřice bajtů, ale také `unsigned` 32-bitové číslo.

*Jedná se o příklad naznačující použití, v tom případě je také nutné zajistit správné pořadí bajtů.*

```
1 #include <stdio.h>
2 #include <stdint.h>
4 union IPv4Address {
5     uint32_t addr;    // adresa jako celé číslo (32 bitů)
6     uint8_t bytes[4]; // adresa po jednotlivých bytech
7 };
9 int main(void)
10 {
11     union IPv4Address ip;
13     // nastavení adresy "192.168.0.1"
14     ip.bytes[0] = 192;
15     ip.bytes[1] = 168;
16     ip.bytes[2] = 0;
17     ip.bytes[3] = 1;
19     printf("IPv4 address: %u.%u.%u.%u\n",
20           ip.bytes[0], ip.bytes[1], ip.bytes[2], ip.bytes
21           [3]);
22     printf("IPv4 as a number: %u\n", ip.addr);
23     return 0;
24 }
```

```
IPv4 address:
    192.168.0.1
IPv4 as a number:
    16820416
```

lec06/demo-union.c

## Inicializace union

- Proměnnou typu `union` můžeme inicializovat při definici.

```
1 union {  
2     char c;  
3     int i;  
4     double d;  
5 } numbers = { 'a' };
```

*Pouze první položka (proměnná) může být inicializována.*

- V C99 můžeme inicializovat konkrétní položku (proměnnou).

```
1 union {  
2     char c;  
3     int i;  
4     double d;  
5 } numbers = { .d = 10.3 };
```

## Příklad struktura, pole a výčtový typ 1/3

- Hodnoty (konstanty) výčtového typu jsou celá čísla, která mohou být použita jako indexy (pole).
- Také je můžeme použít pro inicializaci pole struktur.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  enum weekdays { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
6
7  typedef struct {
8      char *name;
9      char *abbr; // abbreviation
10 } week_day_s;
11
12 const week_day_s days_en[] = {
13     [MONDAY] = { "Monday", "mon" },
14     [TUESDAY] = { "Tuesday", "tue" },
15     [WEDNESDAY] = { "Wednesday", "wed" },
16     [THURSDAY] = { "Thursday", "thr" },
17     [FRIDAY] = { "Friday", "fri" },
18 };
```

lec06/demo-struct.c

## Příklad struktura, pole a výčtový typ 2/3

- Připravíme si pole struktur pro konkrétní jazyk (angličtina a čeština).
- Program vytiskne jméno a zkratku dne v týdnu dle čísla dne v týdnu.

*V programu používáme jednotné číslo dne bez ohledu na jazykovou mutaci.*

```
19 const week_day_s days_cs[] = {
20     [MONDAY] = { "Pondeli", "po" },
21     [TUESDAY] = { "Utery", "ut" },
22     [WEDNESDAY] = { "Streda", "st" },
23     [THURSDAY] = { "Ctvrtek", "ct" },
24     [FRIDAY] = { "Patek", "pa" },
25 };
27 int main(int argc, char *argv[], char **envp)
28 {
29     int day_of_week = argc > 1 ? atoi(argv[1]) : 1;
30     if (day_of_week < 1 || day_of_week > 5) {
31         fprintf(stderr, "(EE) File: '%s' Line: %d -- Given day of week out of range\n",
32             __FILE__, __LINE__);
33         return 101;
34     }
35     day_of_week -= 1; // start from 0
```

lec06/demo-struct.c

## Příklad struktura, pole a výčtový typ 3/3

- Detekci národního prostředí provedeme podle hodnoty proměnné prostředí.

*Pro jednoduchost detekujeme češtinu na základě výskytu řetězce "cs" v hodnotě proměnné prostředí LC\_CTYPE.*

```
35     _Bool cz = 0;
36     while (*envp != NULL) {
37         if (strstr(*envp, "LC_CTYPE") && strstr(*envp, "cs")) {
38             cz = 1;
39             break;
40         }
41         envp++;
42     }
43     const week_day_s *days = cz ? days_cs : days_en;
44     printf("%d %s %s\n", day_of_week,
45           days[day_of_week].name,
46           days[day_of_week].abbr
47         );
48     return 0;
49 }
50 }
```

lec06/demo-struct.c

**V programu jsme využili koncept definování datových struktur, které následně programově přepínáme a využíváme.**  
Alternativně můžeme data načítat ze souboru. *V programu se snažíme obecně pracovat s datovými strukturami.*

## Příklad - Zprávy různých typů v komunikaci

- Komunikace probíhá definovanými zprávami s hlavičkou (typ zprávy) a daty, která mohou mít různou podobu. `union` umožňuje uložit různá data do stejného bufferu.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MSG_LEN 32
5  typedef enum {
6      MSG_TEXT,
7      MSG_SENSOR,
8      MSG_COMMAND
9  } MsgType;
10
11 typedef struct {
12     MsgType type;
13     union {
14         char text[MSG_LEN];
15         struct {
16             int temperature;
17             int humidity;
18         } sensor;
19         int commandId;
20     } data;
21 } Message;
22
23
24
25 int main(void)
26 {
27     Message m1 = { .type = MSG_TEXT };
28     // Text je kratší než MSG_LEN
29     strcpy(m1.data.text, "Hello PRP!");
30
31     Message m2 = { .type = MSG_SENSOR };
32     m2.data.sensor.temperature = 23;
33     m2.data.sensor.humidity = 45;
34
35     Message m3 = { .type = MSG_COMMAND };
36     m3.data.commandId = 42;
37
38     if (m1.type == MSG_TEXT) {
39         printf("Text: %s\n", m1.data.text);
40     }
41     if (m2.type == MSG_SENSOR) {
42         printf("Sensor: %d°C, %d%%\n",
43             m2.data.sensor.temperature,
44             m2.data.sensor.humidity);
45     }
46     if (m3.type == MSG_COMMAND) {
47         printf("Command ID: %d\n",
48             m3.data.commandId);
49     }
50     return EXIT_SUCCESS;
51 }

```

```

$ clang union-msg.c -o msg
$ ./msg
Text: Hello PRP!
Sensor: 23°C, 45%
Command ID: 42

```

lec06/union-msg.c

## Kódovací příklad union – is\_big\_endian()

### Tisk hodnot v šestnáctkové soustavě

- Reprezentace `float` hodnot.
  - Hodnota 85.125 je `0x42aa4000`.
  - Hodnota 0.1 je sice `0x3dcccccc`, ale je kódována `0x3dcccccd`. *Protože chyba je absolutně menší.*
- Implementujeme funkci pro tisk paměťové reprezentace hodnoty typu `float` jako posloupnosti hodnot bajtů v šestnáctkové soustavě.
- Přístup k `float` jako posloupnosti bajtů a tisk hex hodnot `"%02x"` funkcí `printf()`.
  - Adresním operátorem `&` získáme adresu proměnné.
  - Přetypujeme adresu jako ukazatel na hodnotu `char`.
  - Použijeme nepřímý adresní operátor `*` k přístupu k hodnotě na adrese uložené v ukazateli.
- Datový typ `float` má vnitřní reprezentaci dle IEEE 754 definující neceločíselnou reprezentaci ve 32-bitech, které jsou v paměti uloženy jako 4 bajty dle architektury.

```
1  #include <stdio.h>
3  void print_float_hex(float v);
5  int main(void)
6  {
7      print_float_hex(85.125);
8      print_float_hex(0.1);
9      return 0;
10 }
12 void print_float_hex(float v)
13 {
14     ...
15 }
```

## Příklad – Tisk hodnot v šestnáctkové soustavě 1/3

- Získáme adresu proměnné `float v` operátorem `&v`.
- K hodnotám na adrese `&v` budeme přistupovat jako k bajtům, proto přetypujeme adresu na ukazatel (adresu) na hodnoty typu `char`.

```
unsigned char *p = (unsigned char*)&v;
```

- Hodnotu uloženou na adrese `p` získáme operátorem nepřímého adresování `*p`.
- Adresu následujícího bajtů za adresou uloženou v `p` získáme `p = p + 1`;

*Protože se jedná o ukazatel na `char`, probíhá inkrementace o `sizeof(char)`, tj. o 1 (ukazatelová aritmetika).*

- Vytisknuté hodnoty jsou v opačném než očekávaném pořadí **0x42aa4000** a **0x3dcccccd**.

```

1 int main(void)
2 {
3     print_float_hex(85.125);
4     print_float_hex(0.1);
5     ...
6 void print_float_hex(float v)
7 {
8     unsigned char *p = (unsigned char*)&v;
9     printf("Value %13.10f is 0x", v);
10    for (int i = 0; i < sizeof(float); ++i,
11         p = p + 1) {
12        printf("%02x", *p); // or use p[i]
13    }
14    putchar('\n');
```

```

1 $ clang floats.c -o floats && ./floats
2 Value 85.1250000000 is 0x0040aa42
3 Value  0.1000000015 is 0xcdcccc3d
```

## Příklad – Tisk hodnot v šestnáctkové soustavě 2/3

- Očekávaná reprezentace v šestnáctkové soustavě je pro 85.125 výstup **0x42aa4000** a pro 0.1 výstup **0x3dcccccd**. Namísto toho dostáváme 0x0040aa42 a 0xcdcccc3d.

- Výstup je závislý na reprezentaci více bajtových hodnot v paměti. Pro architekturu (amd64) je to tzv. little endian.

<https://en.wikipedia.org/wiki/Endianness>

- Proto potřebujeme detekovat, jak jsou hodnoty uloženy, například funkcí

```
_Bool is_big_endian(void);
```

- a případně vytiskneme hodnoty v opačném pořadí.

```

1 void print_float_hex(float v)
2 {
3     const _Bool big_endian = is_big_endian();
4     // cast pointer to float to pointer to char
5     unsigned char *p = (unsigned char*)&v
6         + (big_endian ? 0 : (sizeof(float) - 1));
7     printf("Value %13.10f is 0x", v);
8     for (int i = 0; i < sizeof(float); ++i) {
9         printf("%02x",
10             *(big_endian ? p++ : p--));
11     }
12     printf("\n");
13 }
14 
```

```

$ clang floats.c -o floats && ./floats
Value 85.1250000000 is 0x42aa4000
Value 0.1000000015 is 0x3dcccccd

```

## Příklad – Tisk hodnot v šestnáctkové soustavě 3/3

- Detekce uložení můžete být založena na různých principech.
- Intuitivně můžeme uložit definovanou hodnotu, která má pouze jeden bajt nenulový a ostatní nulové.
- Využijeme složeného typu `union`, ve kterém položky sdílejí paměť a umožňuje nám tak různý pohled na konkrétní block paměti.
  1. Definujeme celočíselnou proměnnou o čtyřech bajtech, např., `uint32_t` z knihovny `stdint.h`.
  2. Nastavíme hodnotu na `0x01 00 00 00`.
  3. Otestujeme první bajt paměťové reprezentace.

```
1 #include <stdint.h>
2
3 _Bool is_big_endian(void)
4 {
5     union {
6         uint32_t i;
7         char c[4]; // 4 is fine here as we use
8                     uint32_t
9     } e = { 0x01000000 };
10    return e.c[0];
11 }
```

## Část II

### Část 2 – Zadání 6. domácího úkolu (HW06)

## Zadání 6. domácího úkolu HW06

### Téma: Maticové počty

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

- **Motivace:** Získání zkušenosti s dvojrozměrným polem.
- **Cíl:** Osvojit si práci s polem variabilní délky a předávání ukazatelů.
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw06>
  - 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.

Využití `struct` a dynamické alokace může být výhodnou, není však nutné.

- **Termín odevzdání:** 06.12.2025, 23:59:59 PST.
- **Bonusová úloha:** 10.01.2026, 23:59:59 CET.

*PST – Pacific Standard Time*

*CET – Central European Time*

## Shrnutí přednášky

## Diskutovaná témata

- Struktury, způsoby definování, inicializace a paměťové reprezentace
- Uniony
  
- Příště: Paměť programu. Standární knihovny C. Rekurze.

## Část III

## Appendix

## Kódovací příklad – Textové řetězce – toupper() 1/2

- Implementujme funkci, která převede malá písmena na velká (ASCII znaky 'a'-'z'). Využijeme vlastní myMalloc().

```

1 #ifndef __MY_MALLOC_H__
2 #define __MY_MALLOC_H__
4 #include <stdlib.h>
6 void* myMalloc(size_t size, const char *filename, int line);
8 #endif
                                     my_malloc.h

```

```

1 #include <stdio.h>
2 #include <stdlib.h>
4 #include "my_malloc.h"
6 void* myMalloc(size_t size, const char *filename, int line)
7 {
8     void *ret = malloc(size);
9     if (!ret) {
10         fprintf(stderr, "ERROR: Malloc failed called at %s:%i!\n",
11             filename, line);
12         exit(-1);
13     }
14     return ret;
                                     my_malloc.c

```

```

1 #include <stdio.h>
2 #include <string.h>
4 #include "my_malloc.h"
6 int main(void)
7 {
8     const char *str = "I like prp!"; // Ukazatel na literál!
9     const size_t n = strlen(str); // Co se stane když str == NULL!
10    char *stru = myMalloc(
11        (n + 1) * sizeof(char), //+1 pro '\0'
12        __FILE__, __LINE__
13    );
14    for (int i = 0; i < n; ++i) {
15        stru[i] = (str[i] >= 'a' && str[i] <= 'z') ?
16            str[i] & 0xdf : str[i]; // 0xdf viz ASCII tabulka!
17    }
18    stru[n] = '\0'; // zajištění textového řetězce
19    printf("%s\n", str);
20    printf("%s\n", stru);
21    free(stru); // Volání je ok i v případě, že stru == NULL.
22    return EXIT_SUCCESS;
23 }

```

- V našem případě je `str` platný řetězec, proto je řádek 9 v pořádku.
- Přesto převod prepíšeme do funkce `toupper()`, kde tomu tak být nemusí.

## Kódovací příklad – Textové řetězce – toupper() 2/2

```

1 #include <stdio.h>
2 #include <string.h>
4 #include "my_malloc.h"
6 char* strtoupper(const char *str);
8 int main(void)
9 {
10     const char *str = "I like prp!";
11     char * const stru = strtoupper(str);
13     printf("%s\n", str);
14     printf("%s\n", stru);
15     free(stru); // Volání ok i pro str == NULL.
16     return EXIT_SUCCESS;
17 }

```

```

1 $ clang strtoupper.c my_malloc.c && ./a.out
2 I like prp!
3 I LIKE PRP!

```

- Volání funkce `strtoupper()` může být předán neplatný ukazatel `NULL`).
- Explicitně ošetřujeme, ikdyž například funkce `strlen()` předpokládá validní vstup a volání `strlen(NULL)` může skončit chybou programu.
- V našem programu, alokujeme ve funkci `strtoupper()` paměť dynamicky a to vždy nejméně pro jeden znak (`'\0'`).

```

1 char* strtoupper(const char *str)
2 {
3     char *ret = myMalloc( // Co se stane když malloc(0)?
4         // Ověříme, zdali je str platný ukazatel
5         (str ? strlen(str) + 1 : 1) * sizeof(char),
6         __FILE__, __LINE__
7     );
8     const char *cur = str; // kurzor vstupního řetězce
9     char *d = ret; // kurzor výstupního řetězce
10    while (cur && *cur) {
11        *d = *cur++;
12        if (*d >= 'a' && *d <= 'z') {
13            *d = *d - 'a' + 'A';
14        }
15        d += 1;
16    }
17    *d = '\0'; // ret je vždy nejméně 1 byte dlouhý.
18    return ret;
19 }
20 }

```

## Kódovací příklad – Textové řetězce – strrev() 1/2

- Implementujme funkci, která vrátí obrácený řetěz. Nejdříve však začneme pracovní verzi ve funkci `main()`.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     char *str = "I like prp!";
8     size_t j, n = strlen(str);
9
10    printf("%s\n", str);
11    for (size_t i = 0, j = n-1; i < n/2; ++i, --j) {
12        char t = str[i];
13        str[i] = str[j];
14        str[j] = t;
15    }
16    printf("%s\n", str);
17    return EXIT_SUCCESS;
18 }

```

- V cyklu využíváme operátor čárky k inicializaci a dekrementaci proměnné `j`.
- Opět v našem programu je řetězec `str` platný a můžeme tak bezpečně volat funkci `strlen(str)`.
- Nicméně po odladění obrácení řetězce, program přepíšeme s implementací naší nové funkce `strrev()`.

```

1 $ clang -g strrev.c && ./a.out
2 I like prp!
3
4 Command terminated
5
6 Command: ./a.out
7 ==10618==
8 I like prp!
9 ==10618==
10 ==10618== Process terminating with default action of signal 11 (SIGSEGV)
11 ==10618== Bad permissions for mapped region at address 0x20056D
12 ==10618== at 0x2019F9: main (strrev.c:13)

```

- Program však skončí chybou! Zapisujeme do paměti literálů!**

```

7 char str[] = "I like prp!";

```

- Nahrazením ukazatele na literál polem, program funguje.

```

1 $ clang -g strrev.c && ./a.out
2 I like prp!
3 !prp ekil I

```

- Program přepíšeme s využitím `myMalloc()`.

## Kódovací příklad – Textové řetězce – strrev() 2/2

```

1 #include <stdio.h>
2 #include <stdlib.h>
4 #include "my_malloc.h"
6 char* strrev(const char *str);
8 int main(void)
9 {
10     char *str = "I like prp!";
11     char *strr = strrev(str);
13     printf("%s\n", str);
14     printf("%s\n", strr);
16     free(strr);
17     return EXIT_SUCCESS;
18 }
```

- Funkce `strrev()` vytváří nový řetězec, proto můžeme bezpečně předat ukazatel na textový literál.
- Volání `strrev()` vrátí textový řetězec, nebo končí chybou (volání `myMalloc()`).
- Proměnná `strr` tak vždy ukazuje na paměť, ve které je nejméně jeden znak a to `'\0'`.
- Program tak v rámci `main()` vždy skončí úspěšně `EXIT_SUCCESS`.
- Ve funkci `main()` tak vlastně ani explicitně neřešíme návratové hodnoty volání.

```

20 char* strrev(const char *str)
21 {
22     size_t n = str ? strlen(str) : 0;
23     char *ret = myMalloc((n + 1) * sizeof(char), __FILE__, __LINE__);
24     const char *cur = str + n; // ukazatelová aritmetika
25     char *dst = ret;
26     while (str && cur != str) { // kontrola str!
27         *dst = *--cur;
28         dst += 1;
29     }
30     *dst = '\0'; //ret je vždy nejméně 1 byte dlouhý.
31     return ret;
32 }
```

- Ve funkci explicitně ověříme, že vstupní řetězec není `NULL`.
- V naší implementaci je prázdný (`NULL`) řetězec ekvivalentní s řetězcem o délce nula.
- Pokud je `str == NULL`, není hodnota `cur` validní.
- Proto ve `while` cyklu explicitně testujeme `str`.
- Z hlediska efektivity bychom mohli volání funkce v případě `str == NULL` ukončit dříve.
- Nicméně volíme přehlednost, menší počet řádků a jediný `return` ve funkci.

## Kódovací příklad – Textové řetězce – strwc() 1/2

- Implementujme funkci, která vrátí počet slov v řetězci.
- Slovo interpretujeme jako souvislou sekvenci znaků vyhovující funkci `isalpha()` z knihovny `ctype.h`.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <ctype.h>
5
6 int main(void)
7 {
8     int c, wc = 0;
9     bool inword = false;
10    while ((c = getchar()) != EOF) {
11        if (isalpha(c) {
12            if (!inword) {
13                inword = true;
14                wc += 1;
15            }
16        } else {
17            inword = false;
18        }
19    }
20    printf("Input contains %d words.\n", wc);
21    return EXIT_SUCCESS;
22 }

```

- Řádky 14–17 můžeme nahradit následujícím řádkem.

```
!inword && (wc++) && inword++;
```

```

1 $ cat in.txt
2 I like prp!
3
4 $ clang -g wc.c && ./a.out < in.txt
5 Input contains 3 words.

```

- Po počátečním odladění implementujeme funkci `strwc()`.

```

1 int strwc(const char *str)
2 {
3     int wc = 0;
4     bool inword = false;
5     const char *cur = str;
6     while (cur && *cur != '\0') {
7         if (isalpha(*cur) {
8             if (!inword) {
9                 inword = true;
10                wc += 1;
11            }
12        } else {
13            inword = false;
14        }
15        cur += 1;
16    }
17    return wc;
18 }

```

## Kódovací příklad – Textové řetězce – strwc() 2/2

- Čtení znaků ze `stdin` funkcí `getchar()` nahradíme voláním `getline()` z `stdlib.h`. *Viz man getline.*

```
ssize_t getline(char ** restrict linep, size_t * restrict linecap, FILE * restrict stream);
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5 #include <ctype.h>
6
7 int strwc(const char *str);
8
9 int main(void)
10 {
11     char *line = NULL; // nezbytné k alokaci v getline()
12     size_t cap = 0; // alokovaná kapacita v getline()
13     // getline vrací -1 při chybě, proto ssize_t
14     ssize_t l = getline(&line, &cap, stdin);
15     int wc = strwc(line);
16
17     fprintf(stderr, "DEBUG: Read line \"%s\" that is %lu long
18         stored in %lu bytes.\n", line, l, cap);
19     printf("Input contains %d words.\n", wc);
20     free(line); // proměnná je alokována dynamicky.
21     return EXIT_SUCCESS;
22 }
```

- Funkce `getline()` načítá řádek ze souboru, argument `FILE * restrict stream`, používáme `stdin`.
- Funkce načte řádek včetně oddělovače řádků, tj. `'\n'`.

```

1 $ clang -g wc-clean.c && ./a.out <in.txt
2 DEBUG: Read line "I like prp!"
3 " that is 12 long stored in 16 bytes.
```

- Načtený řetězec obsahuje 11 znaků, konec řádku, a `'\0'`.
- Celkem funkce `getline()` alokovala 16 bytů.
- Program můžeme upravit pro načítání souboru voláním `fopen()`.

```

1 int main(int argc, char *argv[])
2 {
3     char *line = NULL; // nezbytné k alokaci v getline()
4     FILE *fd = argc > 1 ? fopen(argv[1], "r") : NULL;
5     size_t cap = 0; // alokovaná kapacita v getline()
6     ssize_t l = getline(&line, &cap, fd ? fd : stdin);
```

```

1 $ clang -g wc-file.c && ./a.out in.txt
2 DEBUG: Read line "I like prp!"
3 " that is 12 long stored in 16 bytes.
4 Input contains 3 words.
```

- V uvedeném příkladu ztrácíme informaci o chybě načtení souboru.
- Je vhodné explicitně reagovat.
- V programu netestujeme interpunkční znaménka, která jsou součástí slov, ani předložky. Funkcionality implementujte!

## Kódovací příklad – Textové řetězce – strsplit() 1/2

- Implementujme funkci, která rozdělí daný řetězec na dva dle zadaného řetězce.

Všimněte si rozdílu ukazatelů!

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include "my_malloc.h"
6
7 int main(void)
8 {
9     const char *str = "I like programming and PRP especially!";
10    char *s1, *s2;
11    char *delim = "and";
12    char *s = strstr(str, delim);
13
14    s1 = s2 = NULL;
15
16    if (s) { // podřetězec (little) nalezen (v big)
17        fprintf(stderr, "D: str %lu\n", strlen(str));
18        fprintf(stderr, "D: delim %lu\n", strlen(delim));
19        fprintf(stderr, "D: s %lu\n", strlen(s));
20        fprintf(stderr, "D: (s - str) %lu\n", s - str);
21        // rozdíl ukazatelů. Oba odkazují do identického
22        // souvislého bloku paměti.
23        size_t n1 = strlen(str) - strlen(s);
24        size_t n2 = strlen(s);

```

- Začátek řetězce v řetězci najdeme funkcí strstr().

```
char* strstr(const char *big, const char *little)
```

Viz man strstr.

```

25     s1 = myMalloc( (n1 + 1) * sizeof(char), __FILE__, __LINE__);
26     s2 = myMalloc( (n2 + 1) * sizeof(char), __FILE__, __LINE__);
27
28     strncpy(s1, str, n1); // Kopírujeme nejvýše n1 znaků
29     strncpy(s2, s, n2); // Kopírujeme nejvýše n2 znaků (a '\0')
30 }
31
32 printf("String: \"%s\"\n", str); // Vstupní řetězec
33 printf("s1: \"%s\"\n", s1); // 1. část
34 printf("s2: \"%s\"\n", s2); // 2. část
35
36 free(s1); // volání free(NULL) je v pořádku
37 free(s2); // program končí, nemusíme nastavovat s1 = s2 = NULL
38
39 return EXIT_SUCCESS;
40 }

```

- Při implementaci použijeme ladící výstupy na stderr.
- Program odladíme a přepíšeme do funkce.

```

1 $ clang strsplit.c my_malloc.c && ./a.out
2 D: str 38
3 D: delim 3
4 D: s 19
5 D: (s - str): 19
6 String: "I like programming and PRP especially!"
7 s1: "I like programming "
8 s2: "and PRP especially!"

```

## Kódovací příklad – Textové řetězce – strsplit() 2/2

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <stdbool.h>
6 #include "my_malloc.h"
8 bool strsplit(const char *str, const char *delim, char **s1,
   char **s2);
10 int main(void)
11 {
12     const char *str = "I like programming and PRP especially!";
13     char *delim = "and";
14     char *s1, *s2;
15     strsplit(str, delim, &s1, &s2);
16     printf("String: \"%s\"\n", str);
17     printf("s1: \"%s\"\n", s1);
18     printf("s2: \"%s\"\n", s2);
20     free(s1); // it is ok to call free(NULL);
21     free(s2);
22     return EXIT_SUCCESS;
23 }

```

- Začátek řetězce v řetězci najdeme funkcí `strstr()`.

```
char* strstr(const char *big, const char *little)
```

Viz man `strstr`.

```

1 bool strsplit(const char *str, const char *delim, char **s1, char **s2)
2 {
3     char *s = NULL;
4     if (
5         !str || !delim || !s1 || !s2 // Inverze, podmínka na argumenty
6         || !(s = strstr(str, delim)) // Podřetězec nalezen.
7     ) {
8         return false;
9     }
11    size_t l2 = strlen(s); // Předpokládáme null-terminated řetězce.
12    size_t l1 = strlen(str) - l2; // strlen(str) >= l2
13    *s1 = myMalloc((l1 + 1) * sizeof(char), __FILE__, __LINE__);
14    *s2 = myMalloc((l2 + 1) * sizeof(char), __FILE__, __LINE__);
15    strncpy(*s1, str, l1);
16    strncpy(*s2, s, l2);
17    return true;
18 }

```

```

1 $ clang -g strsplit.c my_malloc.c && ./a.out
2 String: "I like programming and PRP especially!"
3 s1: "I like programming "
4 s2: "and PRP especially!"

```

- Při implementaci můžeme ladit programem `valgrind`.  
Nicméně ne vždy detekuje možné problémy správně.
- Funkci `strsplit()` můžeme dále doplnit, např. o rozdělení bez `delim`.

## Kódovací příklad – Knihovna – strings.h

- Implementované funkce `toupper()`, `strrev()`, `strwc()`, `strsplit()` vložíme do knihovny `strings.h` a `strings.c`.
- Do knihovny vložíme lokální verzi funkce `myMalloc()`, kterou definujeme jako `static` v souboru `strings.c`.

```

1 #ifndef __STRINGS_H__
2 #define __STRINGS_H__
4 #include <stdbool.h> // Protože bool v strsplit()
6 char* strtoupper(const char *str);
7 char* strrev(const char *str);
8 int strwc(const char *str);
9 bool strsplit(const char *str, const char *delim, char **s1,
10               char **s2);
11 #endif
                                     strings.h

```

```

1 #include <stdio.h>
2 #include <stdlib.h>
4 #include "strings.h"
6 int main(void)
7 {
8     char *line = NULL;
9     size_t cap = 0;
10    ssize_t l = getline(&line, &cap, stdin); //see getline
11    int wc = strwc(line);
12    fprintf(stderr, "DEBUG: Read line \"%s\" that is %lu long stored in %lu
13              bytes.\n", line, l, cap);
14    printf("Input contains %d words.\n", wc);
15    free(line);
16    return EXIT_SUCCESS;
                                     demo-wc.c

```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <ctype.h>
5 #include <stdbool.h>
7 #include "strings.h"
9 static void* myMalloc(size_t size, const char *filename, int
10                       line) { ... } // folded
11 char* strtoupper(const char *str) { ... } // folded
13 char* strrev(const char *str) { ... } // folded
15 int strwc(const char *str) { ... } // folded
                                     strings.c

```

```

1 $ clang -Wall -c strings.c -o strings.o
2 $ ar -rcs libstrings.a strings.o
3 $ clang demo-wc.c -lstrings -L. -o demo-wc
4 $ ./demo-wc < in.txt
5 DEBUG: Read line "I like prp!"
6 " that is 12 long stored in 16 bytes.
7 Input contains 3 words.

```

## Kódovací příklad – „String objekt“

- S využitím složeného typu a ukazatele na funkci implementujeme variantu objektu textového řetězce.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <stdbool.h>
5
6 #include "my_malloc.h"
7
8 typedef struct string {
9     char *str;
10    ssize_t len; // size_t vs. ssize_t (-1 might indicate error)
11    size_t (*getLength)(const char *);
12 } string;
13
14 bool string_create(struct string *s, const char *v);
15 void string_destroy(struct string *s);
16
17 int main(void)
18 {
19     string string = { .str = NULL, .len = -1, .getLength = &strlen };
20
21     string_create(&string, "I like PRP!");
22
23     printf("String str: \"%s\"\n", string.str);
24     printf("String length is %lu\n", string.getLength(string.str));
25     printf("strlen length is %lu\n", strlen(string.str));
26
27     string_destroy(&string);
28
29     return EXIT_SUCCESS;
30 }

```

```

33 bool string_create(struct string *s, const char *v)
34 {
35     if (!s) {
36         return false;
37     }
38     s->len = strlen(v); // it might 0 for ""
39     s->str = myMalloc((s->len + 1) * sizeof(char), __FILE__,
40                     __LINE__);
41     strncpy(s->str, v, s->len);
42     return true;
43 }
44
45 void string_destroy(struct string *s)
46 {
47     if (s) {
48         free(s->str);
49         s->len = -1; // -1 to indicate not allocated mem
50         s->str = NULL; // futher indications
51     }
52 }

```

```

1 $ clang strobj.c my_malloc.c && ./a.out
2 String str: "I like PRP!"
3 String length is 11
4 strlen length is 11

```