

# Struktury a uniony, přesnost výpočtů a vnitřní reprezentace číselných typů

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 06

BAB36PRGA – Programování v C

## 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 **přistupujeme 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 funkcí předávána hodnotou i ukazatelem.
- Struktura může být návratovou hodnotou funkce.

## Příklad struct – Inicializace

- Struktury:

```
1 struct record {          1 typedef struct {
2     int number;          2     int n;
3     double value;        3     double v;
4 };                       4 } item;
```
- Proměnné typu struktura můžeme inicializovat prvek po prvu.

```
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 };
```

## Přehled témat

- Část 1 – Struktury a uniony  
Struktury – struct  
Proměnné se sdílenou pamětí – union  
Příklad  
Základní číselné typy a jejich reprezentace v počítači  
Typové konverze  
Matematické funkce  
*S. G. Kochan: kapitola 9 a 17*
- Část 2 – Přesnost výpočtů a vnitřní reprezentace číselných typů  
*S. G. Kochan: kapitola 14 (typové konverze)*
- Část 3 – Zadání 6. domácího úkolu (HW6)  
*Appendix – Kódovací příklady*

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

## 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`

## Část I

### Část 1 – Struktury a uniony

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

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

- Velikost složeného typu musí být známa během překladač, 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);  18 for (int i = 0; i < n; ++i) {
2 #define N 10 // We need named literal.      19     s1.a[i] = n - i;
3 int main(void)                               20 }
4 {                                             21 print("s1.a", n, s1.a);
5     struct { // Anonymous struct            22 print("s2.a", n, s2.a);
6         int a[N]; // Defined size, no VLA    23 return 0;
7     } s1, s2; // Two struct variables        24 // end main()
8     printf("s1 %p; s2 %p\n", &s1, &s2);     26 void print(const char *str, int n, int *a) {
9     for (int i = 0; i < n; ++i) {           27     printf("%s %p: ", str, a);
10        s1.a[i] = i;                          28     for (int i = 0; i < n; ++i) {
11    }                                           29         printf("%d%s", a[i], i < (n-1) ? ", " : "\n");
12    }                                           30 }
13    }                                           31 }
14    }                                           lec06/demo-struct_array.c
15    print("s1.a", n, s1.a);
16    s2 = s1; // Assignment
17    print("s2.a", n, s2.a);
```

## 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 zpřehlední programu. Při běhu programu vlastně není nutné mít v paměti dílčí jména prvků 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 překladu a programu v případě použití VLA nebo konstantní proměnné definující velikost pole.
  - Pole definované velikostí nahraďte dynamicky alokovaným polem.

## Struktura struct a velikost

- Vnitřní reprezentace struktury nutně nemusí odpovídat součtu velikostí jednotlivých prvků.
- ```
1 struct record {          1 typedef struct {
2   int number;           2   int n;
3   double value;        3   double v;
4 };                     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
```

## Proměnné se sdílenou pamětí – union

- **Union** je množina prvků (proměnných), které nemusí být stejného typu.
- Prvky unionu sdílejí společně stejná paměťová místa.  
*Překrývají se*
- Velikost unionu je dána velikostí největšího z jeho prvků.
- Skladba unionu je definována uživatelem jako nový typ sestavený z již definovaných typů.
- 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;
```

## Příklad struct – Přiřazení

- Hodnoty proměnné stejného typu struktury můžeme přiřadit operátorem `=`.
- ```
1 struct record {          1 typedef struct {
2   int number;           2   int n;
3   double value;        3   double v;
4 };                     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) */
```

## 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))` překladací `clang` a `gcc`.
- ```
1 struct record_packed {
2   int n;
3   double v;
4 } __attribute__((packed));
```

## 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  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
```

## Příklad struct – Přímá kopie paměti

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

## 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
```
- 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>

## 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
```

### 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 výtčový typ 1/3

- Hodnoty (konstanty) výtčového 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 výtčový 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] = { "Pondělí", "po" },
21   [TUESDAY] = { "Úterý", "ut" },
22   [WEDNESDAY] = { "Středa", "st" },
23   [THURSDAY] = { "Čtvrtek", "ct" },
24   [FRIDAY] = { "Pátek", "pa" },
25 };
26
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", __FILE__,
32             __LINE__);
33     return 101;
34   }
35   day_of_week -= 1; // start from 0
```

lec06/demo-struct.c

### Příklad struktura, pole a výtčový 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.*

### Datové typy

- Při návrhu algoritmu abstrahujeme od binární podoby paměti počítače.
- S daty pracujeme jako s hodnotami různých datových typů, které jsou uloženy v paměti předepsaným způsobem.
- Datový typ specifikuje
  - Množinu hodnot, které je možné v počítači uložit;
  - Množinu operací, které lze s hodnotami typu provádět.
- Jednoduchý typ** je takový typ, jehož hodnoty jsou atomické, tj. z hlediska operací dále nedělitelné.

*Záleží na způsobu reprezentace.*

### Příklad číselných typů a vnitřní reprezentace

- 32-bitový typ `int` umožňuje uložit celá čísla v intervalu  $(-2147483648, 2147483647)$ , pro která můžeme použít:
  - aritmetické operace `+`, `-`, `*`, `/` s výsledkem hodnota typu `int`;
  - relační operace `==`, `!=`, `>`, `<`, `>=`, `<=`.

- Inicializovat hodnotou dekadického nebo hexadecimálního literálu.

```
1 int i; // definice promenne typu int
2 int decI = 120; // definice spolu s prirazenim
3 int hexI = 0x78; //pocatecni hodnota v 16-kove soustavě
5 int sum = 10 + decI + 0x13; //pocatecni hodnota je vyráz
```

- Vnitřní reprezentace typů (např. `int`, `short`, `double`) umožňuje uložit čísla z definovaného rozsahu s různou přesností.
- Číselné datové typy lze vzájemně převádět implicitní nebo explicitní typovou konverzí.
- Při konverzi nemusí být hodnota zachována** – viz

lec06/demo-types.c

### Reprezentace dat v počítači

- V počítači není u datové položky určeno jaký konkrétní datový typ je v paměti uložen.
- Proto musíme přidělení paměti **definovat** s jakými typy dat budeme pracovat.
- Překladač tuto definici hlídá a volí odpovídající strojové instrukce pro práci s daty, např. jako s odpovídajícími číselnými typy.

*Např. necelocíselné (float) typy a využití tzv. FPU (Floating Point Unit).*

#### Příklad zápisů stejného čísla v různých soustavách.

- 0100 0001<sub>(2)</sub> – binární zápis jednoho bajtu (8-mi bitů);
- 65<sub>(10)</sub> – odpovídající číslo v dekadické soustavě;
- 41<sub>(16)</sub> – odpovídající číslo v šestnáctkové soustavě;
- Obsah paměťového místa 0100 0001<sub>(2)</sub> o velikosti 1 byte může být interpretován jako znak A.

### Číselné soustavy

- Číselné soustavy – poziční číselné soustavy (polyadické) jsou charakterizovány bázi udávající kolik číslic lze maximálně použít.  
 $x_d = \sum_{i=-n}^{i=m} a_i \cdot z^i$ , kde  $a_i$  je číslice a  $z$  je základ soustavy.
- Unární – např. počet vypitých pšlitrů.
- Binární soustava (bin) – 2 číslice 0 nebo 1.

$$\begin{aligned} 11010, 01_{(2)} &= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 + 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} \\ &= 26,25 \end{aligned}$$

- Desítková soustava (dec) – 10 číslic, znaky 0 až 9.  
 $138, 24_{(10)} = 1 \cdot 10^2 + 3 \cdot 10^1 + 8 \cdot 10^0 + 2 \cdot 10^{-1} + 4 \cdot 10^{-2} = 1 \cdot 100 + 3 \cdot 10 + 8 \cdot 1 + 2 \cdot 0,1 + 4 \cdot 0,01$

- Šestnáctková soustava (hex) – 16 číslic, znaky 0 až 9 a A až F.  
 $0x7D_{(16)} = 7 \cdot 16^1 + D \cdot 16^0 = 112 + 13 = 125$

### Kódování záporných čísel

- Přímý kód** – znaménko je určeno prvním bitem (zleva), snadné stanovení absolutní hodnoty. Reprezentace má dvě nuly.

|                      |                          |
|----------------------|--------------------------|
| 121 <sub>(10)</sub>  | 0111 1001 <sub>(2)</sub> |
| -121 <sub>(10)</sub> | 1111 1001 <sub>(2)</sub> |
| 0 <sub>(10)</sub>    | 0000 0000 <sub>(2)</sub> |
| -0 <sub>(10)</sub>   | 1000 0000 <sub>(2)</sub> |

- Inverzní kód** – záporné číslo odpovídá bitové negaci kladné hodnoty čísla. Reprezentace má dvě nuly.

|                      |                          |
|----------------------|--------------------------|
| 121 <sub>(10)</sub>  | 0111 1001 <sub>(2)</sub> |
| -121 <sub>(10)</sub> | 1000 0110 <sub>(2)</sub> |
| 0 <sub>(10)</sub>    | 0000 0000 <sub>(2)</sub> |
| -0 <sub>(10)</sub>   | 1111 1111 <sub>(2)</sub> |

- Doplňkový kód** – záporné číslo je uloženo jako hodnota kladného čísla po bitové negaci zvětšená o 1. Jediná reprezentace nuly.

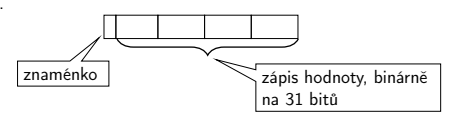
|                      |                          |
|----------------------|--------------------------|
| 121 <sub>(10)</sub>  | 0111 1001 <sub>(2)</sub> |
| -121 <sub>(10)</sub> | 1000 0110 <sub>(2)</sub> |
| -121 <sub>(10)</sub> | 1000 0111 <sub>(2)</sub> |
| 127 <sub>(10)</sub>  | 0111 1111 <sub>(2)</sub> |
| -128 <sub>(10)</sub> | 1000 0000 <sub>(2)</sub> |
| -1 <sub>(10)</sub>   | 1111 1111 <sub>(2)</sub> |

### Více-bajtová reprezentace a pořadí bajtů

- Číselné typy s více-bajtovou reprezentací mohou mít bajty uloženy v různém pořadí.
    - little-endian* – **nejméně** významný bajt se ukládá na nejnižší adresu. x86, ARM
    - big-endian* – **nejvíce** významný bajt se ukládá na nejnižší adresu. Motorola, ARM
  - Pořadí je důležité při přenosu hodnot z paměti jako posloupnosti bajtů a jejich následné interpretaci.
  - Network byte order** – je definován pro síťový přenos a není tak nutné řešit konkrétní architekturu.
    - Tj. hodnoty z paměti jsou ukládány a přenášeny v tomto pořadí bajtů a na cílové stanici pak zpětně zapsány do konkrétního nativního pořadí. big-endian
- Informativní*

### Příklad reprezentace celých čísel int

- Na 32-bitových a 64-bitových strojích je celočíselný typ **int** zpravidla reprezentován 32 bity (4 bajty).
 


- Typ **int** je znaménkový typ.
  - Znaménko je zakódováno v 1 bitu a vlastní číselná hodnota pak ve zbyvajících 31 bitech.
    - Největší číslo je  $0111 \dots 111 = 2^{31} - 1 = 2\,147\,483\,647$ . *Reprezentujeme i nulu.*
    - Nejmenší číslo je  $-2^{31} = -2\,147\,483\,648$ . *0 už je zahrnuta.*
  - Pro zobrazení záporných čísel je použit **doplňkový kód**. *Nejmenší číslo v doplňkovém kódu 1000...000 je  $-2^{31}$ .*

### Reprezentace záporných celých čísel

- Doplňkový kód –  $D(x)$ .
    - Pro 8-mi bitovou reprezentací čísel.
      - Můžeme reprezentovat  $2^8 = 256$  čísel.
      - Rozsah  $r = 256$ .
- $$D(x) = \begin{cases} x & \text{pro } 0 \leq x < \frac{r}{2} \\ r + x & \text{pro } -\frac{r}{2} \leq x < 0 \end{cases} \quad (1)$$
- Příklady
 

| Desítkově | Doplňkový kód                                  |
|-----------|------------------------------------------------|
| 0-127     | 0000 0000 – 0111 1111                          |
| 128       | nelze zobrazit na 8 bitů v doplňkovém kódu     |
| -128      | $D(-128) = 256 + (-128) = 128$ to je 1000 0000 |
| -1        | $D(-1) = 256 + (-1) = 255$ to je 1111 1111     |
| -4        | $D(-4) = 256 + (-4) = 252$ to je 1111 1100     |
- Informativní*

### Necelá čísla a přesnost výpočtu 1/2

- Ztráta přesnosti při aritmetických operacích.
- Příklad sčítání dvou čísel**
- ```

1 #include <stdio.h>
2 int main(void)
3 {
4     double a = 1e+10;
5     double b = 1e-10;
6     printf("a : %24.12lf\n", a);
7     printf("b : %24.12lf\n", b);
8     printf("a+b: %24.12lf\n", a + b);
9     return 0;
10 }

```
- ```

15 clang sum.c && ./a.out
16 a : 10000000000.000000000000
17 b : 0.00000000000100
18 a+b: 10000000000.000000000000

```
- lec06/sum.c*

### Necelá čísla a přesnost výpočtu 2/2

- Příklad dělení dvou čísel**
- ```

1 #include <stdio.h>
2 int main(void)
3 {
4     const int number = 100;
5     double dV = 0.0;
6     float fV = 0.0f;
7     for (int i = 0; i < number; ++i) {
8         dV += 1.0 / 10.0;
9         fV += 1.0 / 10.0;
10    }
11    printf("double value: %lf ", dV);
12    printf("float value: %lf ", fV);
13    return 0;
14 }

```
- ```

20 clang division.c && ./a.out
21 double value: 10.000000 float value: 10.000002

```
- lec06/division.c*

### Přesnost výpočtu - strojová přesnost


- Strojová přesnost  $\epsilon_m$  - nejmenší desetinné číslo, které přičtením k 1.0 dává výsledek různý od 1, pro  $|v| < \epsilon_m$ , platí
 
$$v + 1.0 == 1.0.$$

*Symbol == odpovídá porovnání dvou hodnot (test na ekvivalenci).*
- Zaokrouhlovací chyba - nejméně  $\epsilon_m$ .
- Přesnost výpočtu - aditivní chyba roste s počtem operací v řádu  $\sqrt{N} \cdot \epsilon_m$ .
  - Často se však kumuluje preferabilně v jedno směru v řádu  $N \cdot \epsilon_m$ .

### Reprezentace reálných čísel

- Pro uložení čísla vyhrazueme omezený paměťový prostor.
- Příklad – zápis čísla  $\frac{1}{3}$  v dekadické soustavě**
- $= 33333333 \dots 3333$
  - $= 0,33$
  - $\approx 0,33333333333333333333$
  - $\approx 0,333$
- V trojkové soustavě:  $0 \cdot 3^3 + 0 \cdot 3^0 + 1 \cdot 3^{-1} = (0,1)_3$*
- Nepřesnosti v zobrazení reálných čísel v konečné posloupnosti bitů způsobují
    - Iracionální čísla, např.  $e, \pi, \sqrt{2}$ ;
    - Čísla, která mají v dané soustavě periodický rozvoj, např.  $\frac{1}{3}$ ;
    - Čísla, která mají příliš dlouhý zápis.

### Model reprezentace reálných čísel

- Reálná čísla se zobrazují jako aproximace daným rozsahem paměťového místa.
  - Reálné číslo  $x$  se zobrazuje ve tvaru
 
$$x = \text{mantisa} \cdot \text{základ}^{\text{exponent}}$$
  - Pro jednoznačnost zobrazení musí být mantisa normalizována, např.  $0, 1 \leq m < 1$  nebo ve tvaru  $\pm 1.[\text{mantisa}] \cdot 2^{\text{exponent}}$
  - Ve vyhrazeném paměťovém prostoru je pro zvolený základ uložen exponent a mantisa jako dvě celá čísla.
- 

### Příklad modelu reprezentace reálných čísel na 7 bajtů se základem 10

- Mantisa 3 pozice plus znaménko, délka exponentu 2 pozice plus znaménko, základ  $z = 10$ . *Reprezentace dle IEEE-754 používá dvojkový základ!*
  - Reprezentace nuly.
 

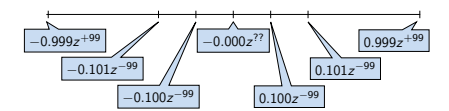
? ?? + 000

+ 02 + 775
  - Maximální zobrazitelné kladné číslo 0,999<sup>99</sup>.
 

+ 99 + 999
  - Maximální zobrazitelné záporné číslo -0,100z<sup>-99</sup>.
 

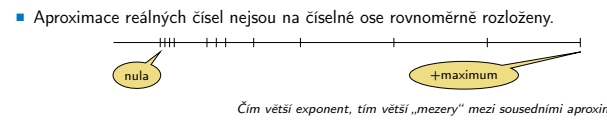
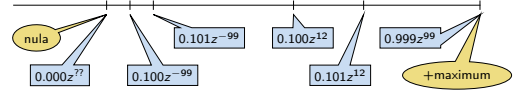
- 99 - 100
  - Minimální zobrazitelné kladné číslo 0,100z<sup>-99</sup>.
 

- 99 + 100
  - Minimální zobrazitelné záporné číslo -0,999z<sup>+99</sup>.
 

+ 99 - 999
- 

### Model reprezentace reálných čísel a vzdálenost mezi aproximacemi

- Rozsah hodnot pro konkrétní exponent je dán velikostí mantisy.
- Absolutní vzdálenost dvou aproximací tak záleží na exponentu.
  - Mezi hodnotou 0 a 1,0 je využit celý rozsah mantisy pro exponenty  $\{-99, -98, \dots, 0\}$ .



### Přirázovací operátor a příkaz

- Slouží pro nastavení hodnoty proměnné.
  - Uložení číselné hodnoty do paměti, kterou proměnná reprezentuje.
- Tvar přirázovacího operátoru.

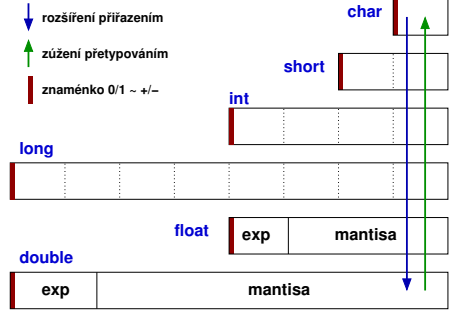
$(\text{proměnná}) = (\text{výraz})$   
 Výraz je literál, proměnná, volání funkce, ...

- Zkrácený zápis  $(\text{proměnná}) (\text{operátor}) = (\text{výraz})$
  - Přirazení je výraz asociativní zprava.
  - Přirázovací příkaz – výraz zakončený středníkem ;
- ```

1 int x; //definice promenne x      1 int x, y; //definice promennych x a y
2 int y; //definice promenne y      3 x = 10;
4 x = 6;                             4 y = 7;
5 y = x = x + 6;                      6 y += x + 10;
```

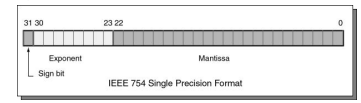
### Konverze primitivních číselných typů

- Primitivní datové typy jsou vzájemně nekompatibilní, ale jejich hodnoty lze převádět.



### Reprezentace necelých čísel – IEEE 754

- Reálné číslo  $x$  se zobrazuje ve tvaru  $x = (-1)^s \cdot \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$ .
  - IEEE 754, ISO/IEC/IEEE 60559:2011
- Mantisa je **normalizována** na první jedničku vlevo (v soustavě o dvojkovém základu).
- float** – 32 bitů (4 bajty):  $s$  – 1 bit znaménko (+ nebo -), **exponent** – 8 bitů, tj. 256 možností. **mantisa** – 23 bitů  $\approx 16,7$  milionu možností.



- double** – 64 bitů (8 bajtů).
  - $s$  – 1 bit znaménko (+ nebo -).
  - exponent** – 11 bitů, tj. 2048 možností.
  - mantisa** – 52 bitů  $\approx 4,5$  biliardy možností (4 503 599 627 370 495).
- bias** umožňuje reprezentovat exponent vždy jako kladné číslo.
  - Lze zvolit, např.  $\text{bias} = 2^{e-1} - 1$ , kde  $e$  je počet bitů exponentu.
  - <http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky>

### Typové konverze

- Typová konverze je operace převedení hodnoty nějakého typu na hodnotu typu jiného.
- Typová konverze může být
  - implicitní** – vyvolá se automaticky;
  - explicitní** – je nutné v programu explicitně uvést.
- Konverze typu **int** na **double** je implicitní.
  - Hodnota typu **int** může být použita ve výrazu, kde se očekává hodnota typu **double**, dojde k automatickému převodu na hodnotu typu **double**.

**Příklad**

```

1 double x;
2 int i = 1;
4 x = i; // hodnota 1 typu int se automaticky převede
5 // na hodnotu 1.0 typu double
```

- Implicitní konverze je bezpečná.

### Matematické funkce

- `<math.h>` – základní funkce pro práci s „reálnými“ čísly.
  - Výpočet odmocniny necelého čísla  $x$ . `double sqrt(double x);`, `float sqrtf(float x);`
  - V C funkci nepřetěžujeme, proto jsou jména odlišena.*
  - `double pow(double x, double y)`; – výpočet obecné mocniny.
  - `double atan2(double y, double x)`; – výpočet  $\arctan y/x$  s určením kvadrantu.
  - Symbolické konstanty – `M_PI`, `M_PI_2`, `M_PI_4`, atd.
    - `#define M_PI 3.14159265358979323846`
    - `#define M_PI_2 1.57079632679489661923`
    - `#define M_PI_4 0.78539816339744830962`
  - `isfinite()`, `isnan()`, `isless()`, ... – makra pro porovnání reálných čísel.
  - `round()`, `ceil()`, `floor()` – zaokrouhlování, převod na celá čísla.
- `<complex.h>` – funkce pro počítání s komplexními čísly. *ISO C99*
- `<fenv.h>` – funkce pro řízení zaokrouhlování a reprezentaci dle IEEE 754. *man math*

### Příklad reprezentace float hodnot dle IEEE 754

- Chyba reprezentace -256.75 vs -256.74.
- Infinity (0x7f800000), -Infinity (0xff800000), a NaN (0x7fffffff).
  - <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

### Explicitní typové konverze

- Převod hodnoty typu **double** na **int** je třeba **explicitně** předeepsat.
- Dojde k „odseknutí“ necelé části hodnoty **int**.

**Příklad**

```

1 double x = 1.2; // definice proměnné typu double
2 int i; // definice proměnné typu int
3 int i = (int)x; // hodnota 1.2 typu double se převede
4 // na hodnotu 1 typu int
```

- Explicitní konverze je potenciálně nebezpečná.

**Příklady**

```

1 double d = 1e30;
2 int i = (int)d;
4 // i je -2147483648
5 // to je asi -2e9 místo 1e30

1 long l = 5000000000L;
2 int i = (int)l;
4 // i je 705032704
5 // (oříznuté 4 bajty)
   lec06/demo-type_conversion.c
```

Část II

Část 3 – Zadání 6. domácího úkolu (HW6)

## Zadání 6. domácího úkolu HW6

### Téma: Maticové počty

Povinné zadání: **4b**; Volitelné zadání: **4b**; Bonusové zadání: **5b**

- **Motivace:** Získání zkušenosti s dvojrozměrným polem.
- **Cíl:** Osvojit si práci s polem variabilní délky a předávání ukazatelů.
- **Zadání:** <https://cv.fel.cvut.cz/wiki/courses/bab36prga/hw/hw6>
  - 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.

- **Termin odevzdání:** 20.04.2024, 23:59:59 PDT.
- **Bonusová úloha:** 24.05.2024, 23:59:59 CEST.

## Shrnutí přednášky

## Diskutovaná témata

- Struktury, způsoby definování, inicializace a paměťové reprezentace
- Uniony
- Přesnost výpočtu
- Vnitřní paměťová reprezentace celočíselných i neceločíselných číselných typů
- Knihovna `math.h`

- **Příště:** Standardní knihovny C. Rekurse.

## Část IV

## Appendix

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

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

```
1 #ifndef _MY_MALLOC_H_
2 #define _MY_MALLOC_H_
3 #include <stdio.h>
4 #include <stdlib.h>
5 void* myMalloc(size_t size, const char *filename,
6               int line);
7 #endif
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include "my_malloc.h"
12 void* myMalloc(size_t size, const char *filename,
13               int line)
14 {
15     void *ret = malloc(size);
16     if (ret) {
17         fprintf(stderr, "ERROR: Malloc failed called
18             at %s:%i\n", filename, line);
19         exit(-1);
20     }
21     return ret;
22 }
23 #endif
24 my_malloc.c
```

```
1 #include <stdio.h>
2 #include <string.h>
3 #include "my_malloc.h"
4 int main(void)
5 {
6     const char *str = "I like prg!"; // Ukazatel na literál!
7     const size_t n = strlen(str); // Co se stane když str == NULL!
8     char *stru = myMalloc(
9         (n + 1) * sizeof(char), //+1 pro '\0'
10        _FILE_, _LINE_);
11
12     for (int i = 0; i < n; ++i) {
13         stru[i] = (str[i] >= 'a' && str[i] <= 'z') ?
14             str[i] && toupper(str[i]); // Odeř viz ASCII tabulka!
15     }
16     stru[n] = '\0'; // zajištění textového řetězce
17     printf("%s\n", str);
18     printf("%s\n", stru);
19     free(stru); // Volání ok i pro str == NULL.
20     return EXIT_SUCCESS;
21 }
```

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

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

```
1 #include <stdio.h>
2 #include <string.h>
3 #include "my_malloc.h"
4 char* strtoupper(const char *str)
5 {
6     char *ret = myMalloc( // Co se stane když malloc(0)?
7         // Ověříme, zdali je str platný ukazatel
8         (str ? strlen(str) + 1 : 1) * sizeof(char),
9         _FILE_, _LINE_);
10
11     const char *cur = str; // kurzor vstupního řetězce
12     char *ret = str; // kurzor výstupního řetězce
13     while (cur && *cur) {
14         *ret = *cur++;
15         if (*ret >= 'a' && *ret <= 'z') {
16             *ret = *ret + 'A' - 'a';
17             d += 1;
18         }
19     }
20     return ret;
21 }
```

- 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').

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

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

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

```
1 $ clang -g strev.c && ./a.out
2 I like prg!
3 Command terminated
4 Command: ./a.out
5 --10618--
6 --10618-- Process terminating with default action of signal 11 (
7     SIGSEGV)
8 --10618-- Bad permissions for mapped region at address 0x20056D
9 --10618-- at 0x2019F9: main (strev.c:13)
10
11 ■ Program však skončí chybou! Zapisujeme do paměti literál!
12 char str[] = "I like prg!";
```

- 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 ovládnutí obrácení řetězce, program přepíšeme s implementací naší nové funkce `strev()`.

- Nahrazením ukazatele na literál polem, program funguje.
- 1 \$ clang -g strev.c && ./a.out
- 2 I like prg!
- 3 |prg ekil I
- Program přepíšeme s využitím `myMalloc()`.

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "my_malloc.h"
4 char* strev(const char *str);
5 int main(void)
6 {
7     char *str = "I like prg!";
8     char *strev = strev(str);
9     printf("%s\n", str);
10    printf("%s\n", strev);
11    free(str);
12    return EXIT_SUCCESS;
13 }
```

- Funkce `strev()` vytváří nový řetězec, proto můžeme bezpečně předat ukazatel na textový literál.
- Volání `strev()` vrátí textový řetězec, nebo končí chybou (volání `myMalloc()`).
- Proměnná `str` 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.

```
1 char* strev(const char *str)
2 {
3     size_t n = str ? strlen(str) : 0;
4     char *ret = myMalloc((n + 1) * sizeof(char), _FILE_,
5         _LINE_);
6     const char *cur = str + n; // ukazatelová aritmetika
7     char *dst = ret;
8     while (str && cur != str) { // kontrola str!
9         *dst = *--cur;
10        dst += 1;
11    }
12    *dst = '\0'; //ret je vždy nejméně 1 byte dlouhý.
13    return ret;
14 }
```

- Ve funkci explicitně ověřujeme, 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.
- Ve funkci `main()` tak vlastně ani explicitně neřešíme návratové hodnoty volání.
- Nicméně volíme přehlednou, menší počet řádků a jediný `return` ve funkci.

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

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

- Řádky 14–17 můžeme nahradit následujícím řádkem.  
`inword && (wc++) && inword++;`

```
1 int strev(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 – strvc() 2/2

■ Čtení znaků ze stdin funkcí `getchar()` nahradíme voláním `getline()` z `stdlib.h`. Viz man `getline`.  
**size\_t** `getline(char ** restrict linep, FILE * restrict stream);`

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <ctype.h>
5
6 int strvc(const char *str);
7 int main(void)
8 {
9     char *line = NULL; // nezbytné k alokaci v getline()
10    size_t cap = 0; // alokovaná kapacita v getline()
11
12    // getline vrací -1 při chybě, proto size_t
13    ssize_t l1 = getline(&line, &cap, stdin);
14    int wc = strvc(line);
15    fprintf(stderr, "DEBUG: Read line \"%s\" that is %ld\n", line, l1, cap);
16    printf("Input contains %d words.\n", wc);
17    free(line); // proměnná je alokována dynamicky.
18    return EXIT_SUCCESS;
19 }
    
```

■ Nactený 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; // alokována kapacita v getline()
6     ssize_t l1 = getline(&line, &cap, fd ? fd : stdin);
7
8     // clang -g wc-file.c && ./a.out in.txt
9     DEBUG: Read line "I like prg!"
10    * that is 12 long stored in 16 bytes.
11    Input contains 3 words.
12 }
    
```

■ 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 – Knihovna – strings

■ Implementované funkce `toupper()`, `strrev()`, `strvc()`, `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_
3 #include <stdbool.h> // Protože bool v strsplit()
4 char* strtoupper(const char *str);
5 char* strrev(const char *str);
6 bool strsplit(const char *str, const char *delim, char **s1, char **s2);
7 #endif
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <ctype.h>
13 #include <stdbool.h>
14 #include "strings.h"
15
16 static void* myMalloc(size_t size, const char *filename, int line) { ... } // folded
17 char* strtoupper(const char *str) { ... } // folded
18 char* strrev(const char *str) { ... } // folded
19 int strvc(const char *str) { ... } // folded strings.c
    
```

■ `clang -Wall -c strings.c -o strings.o`  
 ■ `ar -rcs libstrings.a strings.o`  
 ■ `clang demo-wc.c -lstrings -L. -o demo-wc`  
 ■ `./demo-wc < in.txt`  
 ■ `DEBUG: Read line "I like prg!"`  
 ■ `* that is 12 long stored in 16 bytes.`  
 ■ `Input contains 3 words.`

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

■ Implementujme funkci, která rozdělí daný řetězec na dva dle zadaného řetězce. **Všimněte si rozdílů ukazatelů!**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "my_malloc.h"
5
6 int main(void)
7 {
8     const char *str = "I like programming and PRG especially!";
9     char *s1, *s2;
10    char *delim = "and";
11    char *s = strstr(str, delim);
12    s1 = s2 = NULL;
13    if (s) { // podřetězec (little) nalezen (v big)
14        fprintf(stderr, "D: str %s\n", str);
15        fprintf(stderr, "D: delim %s\n", str);
16        fprintf(stderr, "D: s %s\n", s);
17        // rozdíl ukazatelů. Oba odkazují do identického
18        // souvislého bloku paměti.
19        size_t n1 = strlen(str) - strlen(s);
20        size_t n2 = strlen(s);
21
22        // Začátek řetězce v řetězci najdeme funkcí strstr().
23        char* strstr(const char *big, const char *little)
24            Viz man strstr.
25    }
26
27    s1 = myMalloc((n1 + 1) * sizeof(char), __FILE__, __LINE__);
28    s2 = myMalloc((n2 + 1) * sizeof(char), __FILE__, __LINE__);
29    strncpy(s1, str, n1); // Kopírujeme nejvýše n1 znaků
30    strncpy(s2, s, n2); // Kopírujeme nejvýše n2 znaků (a '\0')
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    return EXIT_SUCCESS;
39 }
    
```

■ Při implementaci použijeme ladící výstupy na `stderr`.  
 ■ Program oladíme 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 PRG especially!"
7 s1: "I like programming "
8 s2: "and PRG especially!"
    
```

■ Začátek řetězce v řetězci najdeme funkcí `strstr()`.  
 ■ Funkci `strsplit()` můžeme dále doplnit, např. o rozdělení bez `delim`.

## 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 #include "my_malloc.h"
6
7 typedef struct string {
8     char *str;
9     size_t len;
10    size_t (getLength)(const char *);
11 } string;
12
13 bool string_create(struct string *s, const char **);
14 void string_destroy(struct string *s);
15
16 int main(void)
17 {
18     string str = { .str = NULL, .len = 0, .getLength = &strlen };
19     string_create(&str, "I like PRG!");
20     printf("String str: \"%s\"\n", string.str);
21     printf("String length is %ld\n", string.getLength(string.str));
22     string_destroy(&str);
23     return EXIT_SUCCESS;
24 }
25
26 bool string_create(struct string *s, const char **);
27 void string_destroy(struct string *s);
28
29 if (!s) {
30     return false;
31 }
32 s->len = strlen(v);
33 s->str = myMalloc((s->len + 1) * sizeof(char), __FILE__, __LINE__);
34 strncpy(s->str, v, s->len);
35 return true;
36 }
37
38 void string_destroy(struct string *s)
39 {
40     if (s) {
41         free(s->str);
42         s->len = 0;
43     }
44 }
45
46 $ clang strobj.c my_malloc.c && ./a.out
47 String str: "I like PRG!"
48 String length is 11
49 String length is 11
    
```

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

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <stdbool.h>
5 #include "my_malloc.h"
6
7 bool strsplit(const char *str, const char *delim, char **s1, char **s2);
8 int main(void)
9 {
10    const char *str = "I like programming and PRG especially!";
11    char *delim = "and";
12    char *s1, *s2;
13    strsplit(str, delim, &s1, &s2);
14    printf("String: \"%s\"\n", str);
15    printf("s1: \"%s\"\n", s1);
16    printf("s2: \"%s\"\n", s2);
17
18    free(s1); // it is ok to call free(NULL);
19    free(s2);
20    return EXIT_SUCCESS;
21 }
22
23 $ clang strsplit.c my_malloc.c && ./a.out
24 String: "I like programming and PRG especially!"
25 s1: "I like programming "
26 s2: "and PRG especially!"
    
```

■ Začátek řetězce v řetězci najdeme funkcí `strstr()`.  
 ■ Funkci `strsplit()` můžeme dále doplnit, např. o rozdělení bez `delim`.  
 ■ Nicméně ne vždy detekuje možné problémy správně.