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

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

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 **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 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).

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

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

```c
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`.

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

  - Definujeme idenfikátor `record` ve jmeném prostoru struktur.
- Definicí typu `typedef` zavádíme nové jméno typu `record`.

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

  - Definujeme globální idenfikátor `record` jako jméno typu `struct record`.
- Obojí můžeme kombinovat v jediné definici jména a typu struktury.

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

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


Příklad `struct` – Inicializace

- Struktury:

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

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

- Proměnné typu struktura můžeme inicializovat prvek po prvku.

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

- Podobně jako pole lze inicializovat přímo při definici

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

- nebo pouze konkrétní položky (ostatní jsou nulovány).

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

lec06/struct.c

Příklad `struct` jako parametr funkce

- Struktury můžeme předávat jako parametry funkcí hodnotou.

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

- Nebo hodnotou ukazatele

```c
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 překladu, 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.

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

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 zpřehlední programu. Při běhu programu vlastně není nutné mít v paměti dílčí jména prvku složeného typu.

```
s1 0x7fffffffe840; s2 0x7fffffffe818
s1.a 0x7fffffffe840: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
s2.a 0x7fffffffe818: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
s1.a 0x7fffffffe840: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
s2.a 0x7fffffffe818: 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 velikosti nahraďte dynamicky alokovaným polem.

Příklad struct – Přiřazení

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

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

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

```c
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`

```c
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.

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

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

```c
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`.

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

lec06/struct.c

Struktura struct a velikost 2/2

- Nebo

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

- Příklad výstupu:

```c
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 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`.

```c
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 union 1/2

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

```c
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));
13   union Numbers numbers;
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.

```c
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);
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);
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
```

lec06/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>
5  enum weekdays { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7  typedef struct {
8     char *name;
9     char *abbr; // abbreviation
10 } week_day_s;
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", __FILE__,
         __LINE__);
32       return 101;
33    }
34    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;
45    printf("%d %s %s\n", day_of_week,
46          days[day_of_week].name,
47          days[day_of_week].abbr
48          );
49    return 0;
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 structurami.*

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;
    Záleží na způsobu reprezentace.
  - Množinu operací, které lze s hodnotami typu provádět.
- **Jednoduchý typ** je takový typ, jehož hodnoty jsou atomické, tj. z hlediska operací dále nedělitelné.

Příklad číselných typů a vnitřní reprezentace

- 32-bitový typ `int` umožňuje uložit celá čísla v intervalu $\langle -2147483648, 2147483647 \rangle$, 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 soustave
5 int sum = 10 + decI + 0x13; //pocatecni hodnota je vyraz
```

- 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. neceločíselné (float) typy a využití tzv. FPU (Floating Point Unit).

Příklad zápisů stejného čísla v různých soustavách.

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

Číselné soustavy

- Číselné soustavy – poziční číselné soustavy (polyadické) jsou charakterizovány bází 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 půllitrů.
- Binární soustava (bin) – 2 číslice 0 nebo 1.

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

- Desítková soustava (dec) – 10 číslic, znaky 0 až 9.

$$
\begin{aligned}
138,24_{(10)} &= \mathbf{1}\cdot 10^2 + \mathbf{3}\cdot 10^1 + \mathbf{8}\cdot 10^0 + \mathbf{2}\cdot 10^{-1} + \mathbf{4}\cdot 10^{-2} \\
&= \mathbf{1}\cdot 100 + \mathbf{3}\cdot 10 + \mathbf{8}\cdot 1 + \mathbf{2}\cdot 0,1 + \mathbf{4}\cdot 0,01
\end{aligned}
$$

- Šestnáctková soustava (hex) – 16 číslic, znaky 0 až 9 a A až F.

$$
\begin{aligned}
0x7D_{(16)} &= 7\cdot 16^1 + D\cdot 16^0 \\
&= 112 + 13 \\
&= 125
\end{aligned}
$$

Kódování záporných čísel

- Přímý kód – znaménko je určeno prvním bitem (zleva), snadné stanovení absolutní hodnoty. Reprezentace má dvě nuly.
  - $121_{(10)}$ $0111\ 1001_{(2)}$
  - $-121_{(10)}$ $1111\ 1001_{(2)}$
  - $0_{(10)}$ $0000\ 0000_{(2)}$
  - $-0_{(10)}$ $1000\ 0000_{(2)}$
- Inverzní kód – záporné číslo odpovídá bitové negaci kladné hodnoty čísla. Reprezentace má dvě nuly.
  - $121_{(10)}$ $0111\ 1001_{(2)}$
  - $-121_{(10)}$ $1000\ 0110_{(2)}$
  - $0_{(10)}$ $0000\ 0000_{(2)}$
  - $-0_{(10)}$ $1111\ 1111_{(2)}$
- **Doplňkový kód** – záporné číslo je uloženo jako hodnota kladného čísla po bitové negaci zvětšená o 1. Jediná reprezentace nuly.
  - $121_{(10)}$ $0111\ 1001_{(2)}$
  - $\tilde{}121_{(10)}$ $1000\ 0110_{(2)}$
  - $-121_{(10)}$ $1000\ 0111_{(2)}$
  - $127_{(10)}$ $0111\ 1111_{(2)}$
  - $-128_{(10)}$ $1000\ 0000_{(2)}$
  - $-1_{(10)}$ $1111\ 1111_{(2)}$

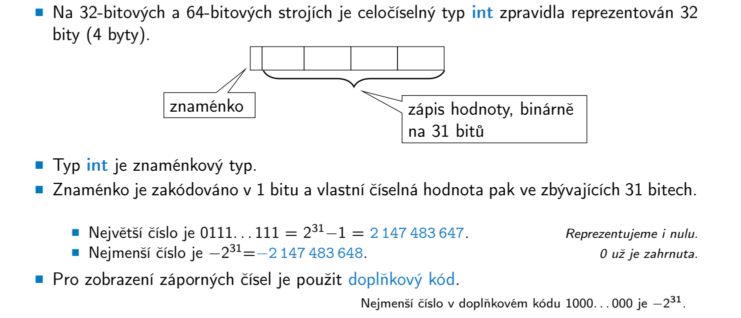
Více-bajtová reprezentace a pořadí bajtů

- 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 byty).



- Typ **int** je znaménkový typ.
- Znaménko je zakódováno v 1 bitu a vlastní číselná hodnota pak ve zbývajících 31 bitech.
  - Největší číslo je 0111...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 $r = 256$.

$$D(x) = \begin{cases} x & \text{pro } 0 \le x < \frac{r}{2} \\ r + x & \text{pro } -\frac{r}{2} \le x < 0 \end{cases} \quad (1)$$

- 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>
3  int main(void)
4  {
5     double a = 1e+10;
6     double b = 1e-10;
8     printf("a  : %24.12lf\n", a);
9     printf("b  : %24.12lf\n", b);
10    printf("a+b: %24.12lf\n", a + b);
12    return 0;
13 }
15 clang sum.c && ./a.out
16 a  : 10000000000.000000000000
17 b  :             0.000000000100
18 a+b: 10000000000.000000000000
```

lec06/sum.c

Necelá čísla a přesnost výpočtu 2/2

Příklad dělení dvou čísel

```
1  #include <stdio.h>
3  int main(void)
4  {
5     const int number = 100;
6     double dV = 0.0;
7     float fV = 0.0f;
9     for (int i = 0; i < number; ++i) {
10       dV += 1.0 / 10.0;
11       fV += 1.0 / 10.0;
12    }
14    printf("double value: %lf ", dV);
15    printf(" float value: %lf ", fV);
17    return 0;
18 }
20 clang division.c && ./a.out
21 double value: 10.000000  float value: 10.000002
```

lec06/division.c

Přesnost výpočtu - strojová přesnost

- Strojová přesnost ϵ_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 ϵ_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 vyhrazujeme omezený paměťový prostor.

  Příklad – zápis čísla $\frac{1}{3}$ v dekadické soustavě
  - $= 33333333\ldots3333$
  - $= 0,3\overline{3}$
  - $\approx 0,33333333333333333333$
  - $\approx 0,333$

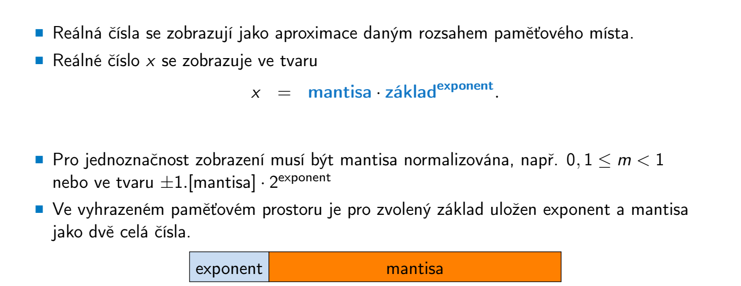
  V trojkové soustavě: $0 \cdot 3^1 + 0 \cdot 3^0 + 1 \cdot 3^{-1} = (0,1)_3$
- Nepřesnosti v zobrazení reálných čísel v konečné posloupnosti bitů způsobují
  - Iracionální čísla, např. e, π, $\sqrt{2}$;
  - Čísla, která mají v dané soustavě periodický rozvoj, např. $\frac{1}{3}$;
  - Čísla, která mají příliš dlouhý zápis.

Model reprezentace reálných čísel

- Reálná čísla se zobrazují jako aproximace daným rozsahem paměťového místa.
- Reálné číslo x se zobrazuje ve tvaru

$$x = \textbf{mantisa} \cdot \textbf{základ}^{\textbf{exponent}}.$$

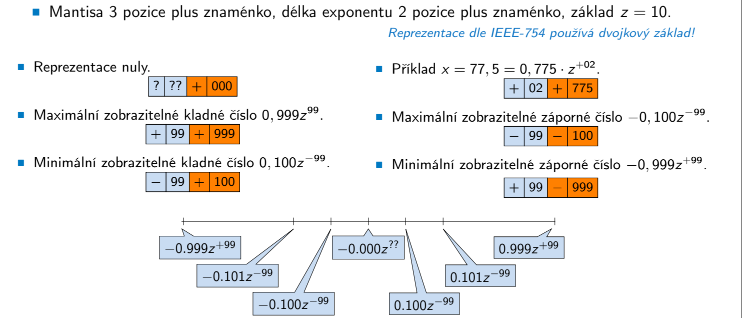
- Pro jednoznačnost zobrazení musí být mantisa normalizována, např. $0,1 \le m < 1$ nebo ve tvaru $\pm1.[\text{mantisa}] \cdot 2^{\text{exponent}}$
- Ve vyhrazeném paměťovém prostoru je pro zvolený základ uložen exponent a mantisa jako dvě celá čísla.



Příklad modelu reprezentace reálných čísel na 7 bajtů se základem 10

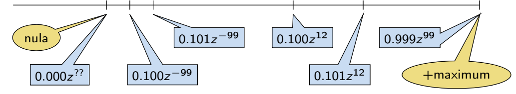
- Mantisa 3 pozice plus znaménko, délka exponentu 2 pozice plus znaménko, základ $z = 10$.
  Reprezentace dle IEEE-754 používá dvojkový základ!

- Reprezentace nuly.
  | ? | ?? | + | 000 |
- Příklad $x = 77,5 = 0,775 \cdot z^{+02}$.
  | + | 02 | + | 775 |
- Maximální zobrazitelné kladné číslo $0,999z^{99}$.
  | + | 99 | + | 999 |
- Maximální zobrazitelné záporné číslo $-0,100z^{-99}$.
  | − | 99 | − | 100 |
- Minimální zobrazitelné kladné číslo $0,100z^{-99}$.
  | − | 99 | + | 100 |
- Minimální zobrazitelné záporné číslo $-0,999z^{+99}$.
  | + | 99 | − | 999 |

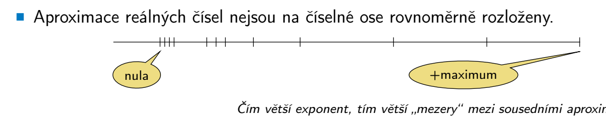


Model reprezentace reálných čísel a vzdálenost mezi aproximacemi

- 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 $\{-99, -98, \dots, 0\}$.



- Aproximace reálných čísel nejsou na číselné ose rovnoměrně rozloženy.



Čím větší exponent, tím větší „mezery" mezi sousedními aproximacemi čísel.

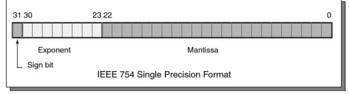
Reprezentace necelých čísel – IEEE 754

- Reálné číslo x se zobrazuje ve tvaru **Základ 2.**

$$x = (-1)^s \mathbf{mantisa} \cdot 2^{\mathbf{exponent}-\mathbf{bias}}.$$

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 ≈ 16,7 milionu možností.



- **double** – 64 bitů (8 bajtů).
  - s – 1 bit znaménko (+ nebo −).
  - **exponent** – 11 bitů, tj. 2048 možností.
  - **mantisa** – 52 bitů ≈ 4,5 biliardy možností (4 503 599 627 370 495).
- **bias** umožňuje reprezentovat exponent vždy jako kladné číslo.

Lze zvolit, např. bias $= 2^{eb-1} - 1$, *kde eb je počet bitů exponentu.*

http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky

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

Přiřazovací operátor a příkaz

- Slouží pro nastavení hodnoty proměnné.

Uložení číselné hodnoty do paměti, kterou proměnná reprezentuje.

- Tvar přiřazovacího operátoru.

$$\langle \textbf{proměnná} \rangle = \langle \textbf{výraz} \rangle$$

Výraz je literál, proměnná, volání funkce, …

- Zkrácený zápis

$$\langle \textbf{proměnná} \rangle \langle \textbf{operátor} \rangle = \langle \textbf{výraz} \rangle$$

- Přiřazení je výraz asociativní zprava.
- Přiřazovací příkaz – výraz zakončený středníkem ;

```
1 int x; //definice promenne x
2 int y; //definice promenne y
4 x = 6;
5 y = x = x + 6;
```

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


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.

Explicitní typové konverze

- Převod hodnoty typu **double** na **int** je třeba explicitně předepsat.
- 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

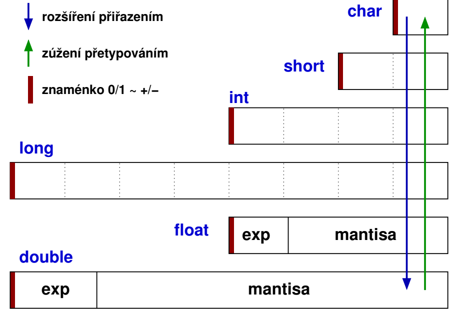
```
1 double d = 1e30;
2 int i = (int)d;
4 // i je -2147483648
5 // to je asi -2e9 misto 1e30
```

```
1 long l = 5000000000L;
2 int i = (int)l;
4 // i je 705032704
5 // (oříznutě 4 bajty)
```

lec06/demo-type_conversion.c

Konverze primitivních číselných typů

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



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 funkce 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`

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://cw.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.

- **Termín 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: Standarní knihovny C. Rekurze.

Část IV

Appendix

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

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

```c
#ifndef __MY_MALLOC_H__
#define __MY_MALLOC_H__

#include <stdlib.h>

void* myMalloc(size_t size, const char *filename,
   int line);

#endif
```
my_malloc.h

```c
#include <stdio.h>
#include <stdlib.h>

#include "my_malloc.h"

void* myMalloc(size_t size, const char *filename,
   int line)
{
   void *ret = malloc(size);
   if (!ret) {
      fprintf(stderr, "ERROR: Malloc failed called
   at %s:%i!\n", filename, line);
      exit(-1);
   }
   return ret;
}
```
my_malloc.c

```c
#include <stdio.h>
#include <string.h>

#include "my_malloc.h"

int main(void)
{
   const char *str = "I like prg!"; // Ukazatel na literál!
   const size_t n = strlen(str); // Co se stane když str == NULL!
   char *stru = myMalloc(
         (n + 1) * sizeof(char), //+1 pro '\0'
         __FILE__, __LINE__
         );
   for (int i = 0; i < n; ++i) {
      stru[i] = (str[i] >= 'a' && str[i] <= 'z') ?
         str[i] & 0xdf : str[i]; // 0xdf viz ASCII tabulka!
   }
   stru[n] = '\0';  // zajištění textového řetězce
   printf("%s\n", str);
   printf("%s\n", stru);
   free(stru); // Volání je ok i v případě, že stru == NULL.
   return EXIT_SUCCESS;
}
```

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

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

```c
#include <stdio.h>
#include <string.h>

#include "my_malloc.h"

char* strtoupper(const char *str);

int main(void)
{
   const char *str = "I like prg!";
   char * const stru = strtoupper(str);

   printf("%s\n", str);
   printf("%s\n", stru);
   free(stru); // Volání ok i pro str == NULL.
   return EXIT_SUCCESS;
}
```

```
$ clang strtoupper.c my_malloc.c && ./a.out
I like prg!
I LIKE PRG!
```

- Volání funkce `strtoupper()` může být předán neplatný ukazatel `NULL`).

```c
char* strtoupper(const char *str)
{
   char *ret = myMalloc( // Co se stane když malloc(0)?
   // Ověříme, zdali je str platný ukazatel
         (str ? strlen(str) + 1 : 1) * sizeof(char),
         __FILE__, __LINE__
         );
   const char *cur = str; // kurzor vstupního řetězce
   char *d = ret; // kurzor výstupního řetězce

   while (cur && *cur) {
      *d = *cur++;
      if (*d >= 'a' && *d <= 'z') {
         *d = *d - 'a' + 'A';
      }
      d += 1;
   }
   *d = '\0'; // ret je vždy nejméně 1 byte dlouhý.
   return ret;
}
```

- 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 – strrev() 1/2

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
   char *str = "I like prg!";
   size_t j, n = strlen(str);

   printf("%s\n", str);
   for (size_t i = 0, j = n-1; i < n/2; ++i, --j) {
      char t = str[i];
      str[i] = str[j];
      str[j] = t;
   }
   printf("%s\n", str);
   return EXIT_SUCCESS;
}
```

- 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()`.

```
$ clang -g strrev.c && ./a.out
I like prg!

Command terminated

Command: ./a.out
==10618==
I like prg!
==10618==
==10618== Process terminating with default action of signal 11 (
    SIGSEGV)
==10618==  Bad permissions for mapped region at address 0x20056D
==10618==    at 0x2019F9: main (strrev.c:13)
```

- Program však skončí chybou! Zapisujeme do paměti literálů!

```c
   char str[] = "I like prg!";
```

- Nahrazením ukazatele na literál polem, program funguje.

```
$ clang -g strrev.c && ./a.out
I like prg!
!prg ekil I
```

- Program přepíšeme s využitím `myMalloc()`.

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

```c
#include <stdio.h>
#include <stdlib.h>

#include "my_malloc.h"

char* strrev(const char *str);

int main(void)
{
   char *str = "I like prg!";
   char *strr = strrev(str);

   printf("%s\n", str);
   printf("%s\n", strr);

   free(strr);
   return EXIT_SUCCESS;
}
```

- Funkce `strrev()` vytváří nový řetězec, proto můžeme bezpečně předat ukazatel na textový literál.
- Volání `strrev()` vratí 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.

```c
char* strrev(const char *str)
{
   size_t n = str ? strlen(str) : 0;
   char *ret = myMalloc((n + 1) * sizeof(char), __FILE__,
   __LINE__);
   const char *cur = str + n; // ukazatelová aritmetika
   char *dst = ret;
   while (str && cur != str) { // kontrola str!
      *dst = *--cur;
      dst += 1;
   }
   *dst = '\0'; //ret je vždy nejméně 1 byte dlouhý.
   return ret;
}
```

- Ve funkci explicitně ověřujeme, že vstupní řetězec není `NULL`.
- V naší implementaci je prázndný (`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`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include <ctype.h>

int main(void)
{
   int c, wc = 0;
   bool inword = false;

   while ((c = getchar()) != EOF) {
      if (isalpha(c)) {
         if (!inword) {
            inword = true;
            wc += 1;
         }
      } else {
         inword = false;
      }
   }
   printf("Input contains %d words.\n", wc);
   return EXIT_SUCCESS;
}
```

- Řádky 14–17 můžeme nahradit následujícím řádkem.

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

```
$ cat in.txt
I like prg!

$ clang -g wc.c && ./a.out < in.txt
Input contains 3 words.
```

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

```c
int strwc(const char *str)
{
   int wc = 0;
   bool inword = false;
   const char *cur = str;
   while (cur && *cur != '\0') {
      if (isalpha(*cur)) {
         if (!inword) {
            inword = true;
            wc += 1;
         }
      } else {
         inword = false;
      }
      cur += 1;
   }
   return wc;
}
```

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 linecapp, FILE * restrict stream);`

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include <ctype.h>

int strwc(const char *str);

int main(void)
{
   char *line = NULL; // nezbytné k alokaci v getline()
   size_t cap = 0; // alokovaná kapacita v getline()
   // getline vrací -1 při chybě, proto ssize_t
   ssize_t l = getline(&line, &cap, stdin);
   int wc = strwc(line);
   fprintf(stderr, "DEBUG: Read line \"%s\" that is %lu
    long stored in %lu bytes.\n", line, l, cap);
   printf("Input contains %d words.\n", wc);
   free(line); // proměnná je alokovaná dynamicky.
   return EXIT_SUCCESS;
}
```

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

```
$ clang -g wc-clean.c && ./a.out <in.txt
DEBUG: Read line "I like prg!
" 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()`.

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

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

- V uvedeném příkladu ztrácíme informaci o chybě načtení souboru.
- Je vhodné explicitně reagovat.
- V programu netestujme 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!**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "my_malloc.h"

int main(void)
{
   const char *str = "I like programming and PRG
    especially!";

   char *s1, *s2;
   char *delim = "and";
   char *s = strstr(str, delim);

   s1 = s2 = NULL;
   if (s) { // podřetězec (little) nalezen (v big)
      fprintf(stderr, "D: str %lu\n", strlen(str));
      fprintf(stderr, "D: delim %lu\n", strlen(delim));
      fprintf(stderr, "D: s %lu\n", strlen(s));
      fprintf(stderr, "D: (s - str) %lu\n", s - str);
      // rozdíl ukazatelů.  Oba odkazují do identického
      // souvislého bloku paměti.
      size_t n1 = strlen(str) - strlen(s);
      size_t n2 = strlen(s);
      s1 = myMalloc( (n1 + 1) * sizeof(char), __FILE__, __LINE__);
      s2 = myMalloc( (n2 + 1) * sizeof(char), __FILE__, __LINE__);

      strncpy(s1, str, n1); // Kopírujeme nejvýše n1 znaků
      strncpy(s2, s, n2); // Kopírujeme nejvýše n2 znaků (a '\0')
   }

   printf("String: \"%s\"\n", str); // Vstupní řetězec
   printf("s1: \"%s\"\n", s1);      // 1.  část
   printf("s2: \"%s\"\n", s2);      // 2.  část

   free(s1); // volání free(NULL) je v pořádku
   free(s2); // program končí, nemusíme nastavovat s1 = s2 = NULL

   return EXIT_SUCCESS;
}
```

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

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

  Viz `man strstr`.

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

```
$ clang strsplit.c my_malloc.c && ./a.out
D: str 38
D: delim 3
D: s 19
D: (s - str): 19
String: "I like programming and PRG especially!"
s1: "I like programming "
s2: "and PRG especially!"
```


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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "my_malloc.h"

bool strsplit(const char *str, const char *delim, char
    **s1, char **s2);

int main(void)
{
   const char *str = "I like programming and PRG
    especially!";
   char *delim = "and";
   char *s1, *s2;
   strsplit(str, delim, &s1, &s2);
   printf("String: \"%s\"\n", str);
   printf("s1: \"%s\"\n", s1);
   printf("s2: \"%s\"\n", s2);

   free(s1); // it is ok to call free(NULL);
   free(s2);
   return EXIT_SUCCESS;
}
```

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

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

  Viz `man strstr`.

```c
bool strsplit(const char *str, const char *delim, char **s1, char **s2)
{
   char *s = NULL;
   if (
         !str || !delim || !s1 || !s2 // Inverze, podmínka na argumenty
         || !(s = strstr(str, delim)) // Podřetězec nalezen.
      ) {
      return false;
   }
   size_t l2 = strlen(s); // Předpokládáme null-terminated řetězce.
   size_t l1 = strlen(str) - l2; // strlen(str) >= l2
   *s1 = myMalloc((l1 + 1) * sizeof(char), __FILE__, __LINE__);
   *s2 = myMalloc((l2 + 1) * sizeof(char), __FILE__, __LINE__);
   strncpy(*s1, str, l1);
   strncpy(*s2, s, l2);
   return true;
}
```

```
$ clang -g strsplit.c my_malloc.c && ./a.out
String: "I like programming and PRG especially!"
s1: "I like programming "
s2: "and PRG 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`.

strings.h

```c
#ifndef __STRINGS_H__
#define __STRINGS_H__

#include <stdbool.h> // Protože bool v strsplit()

char* strtoupper(const char *str);
char* strrev(const char *str);
int strwc(const char *str);
bool strsplit(const char *str, const char *delim, char
    **s1, char **s2);

#endif
```

strings.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>

#include "strings.h"

static void* myMalloc(size_t size, const char *filename,
     int line) { ... } // folded

char* strtoupper(const char *str) { ... } // folded

char* strrev(const char *str) { ... } // folded

int strwc(const char *str) { ... } // folded
```

demo-wc.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "strings.h"

int main(void)
{
   char *line = NULL;
   size_t cap = 0;
   ssize_t l = getline(&line, &cap, stdin); //see getline
   int wc = strwc(line);
   fprintf(stderr, "DEBUG: Read line \"%s\" that is %lu long stored in
    %lu bytes.\n", line, l, cap);
   printf("Input contains %d words.\n", wc);
   free(line);
   return EXIT_SUCCESS;
}
```

```
$ 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 – „String objekt“

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "my_malloc.h"

typedef struct string {
   char *str;
   ssize_t len;
   size_t (*getLength)(const char *);
} string;

bool string_create(struct string *s, const char *v);
void string_destroy(struct string *s);

int main(void)
{
   string string = { .str = NULL, .len = 0, .getLength = &strlen };

   string_create(&string, "I like PRG!");

   printf("String str: \"%s\"\n", string.str);
   printf("String length is %lu\n", string.getLength(string.str));
   printf("strlen length is %lu\n", strlen(string.str));

   string_destroy(&string);

   return EXIT_SUCCESS;
}
```

```c
bool string_create(struct string *s, const char *v)
{
   if (!s) {
      return false;
   }
   s->len = strlen(v);
   s->str = myMalloc((s->len + 1) * sizeof(char),
    __FILE__, __LINE__);
   strncpy(s->str, v, s->len);
   return true;
}

void string_destroy(struct string *s)
{
   if (s) {
      free(s->str);
      s->len = 0;
   }
}
```

```
$ clang strobj.c my_malloc.c && ./a.out
String str: "I like PRG!"
String length is 11
strlen length is 11
```