

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

S. G. Kochan: kapitola 9 a 17

- Část 2 – Přesnost výpočtů a vnitřní reprezentace číselných typů

Základní číselné typy a jejich reprezentace v počítači

Typové konverze

Matematické funkce

S. G. Kochan: kapitola 14 (typové konverze)

- Část 3 – Zadání 5. domácího úkolu (HW5)

Appendix – Kódovací příklady



Část I

Část 1 – Struktury a uniony



Obsah

Struktury – struct

Proměnné se sdílenou pamětí – union

Příklad



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

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 zřehlední 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 překladu a programu v případě použití VLA nebo konstantní proměnné definující velikost pole.
- Pole definované velikosti nahraďte dynamicky alokovaným polem.



Příklad struct – Přřazení

- 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) */
```



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



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>



Obsah

Struktury – struct

Proměnné se sdílenou pamětí – union

Příklad



Proměnné se sdílenou pamětí – union

- **Union** je množina prvků (proměnných), které nemusí být stejného typu.
- Prvky unionu sdílejí společně stejná paměťová místa.
- 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**.

Překrývají se

*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 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
```



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 };
```



Obsah

Struktury – struct

Proměnné se sdílenou pamětí – union

Příklad



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 };
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
32         range\n", __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.*



Část II

Část 2 – Přesnost výpočtů a vnitřní reprezentace číselných typů



Obsah

Základní číselné typy a jejich reprezentace v počítači

Typové konverze

Matematické funkce



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 prirazeni`
 - 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.

$$x_d = \sum_{i=-n}^{i=m} a_i \cdot z^i, \text{ kde } a_i \text{ je číslice a } z \text{ je základ soustavy.}$$

- Unární – např. počet vypitých půllitrů.
- 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.

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

- 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.
- **Inverzní kód** – záporné číslo odpovídá bitové negaci kladné hodnoty čísla. Reprezentace má dvě nuly.
- **Doplňkový kód** – záporné číslo je uloženo jako hodnota kladného čísla po bitové negaci zvětšená o 1. Jediná reprezentace nuly.

■ $121_{(10)}$	$0111\ 1001_{(2)}$
■ $-121_{(10)}$	$1111\ 1001_{(2)}$
■ $0_{(10)}$	$0000\ 0000_{(2)}$
■ $-0_{(10)}$	$1000\ 0000_{(2)}$
■ $121_{(10)}$	$0111\ 1001_{(2)}$
■ $-121_{(10)}$	$1000\ 0110_{(2)}$
■ $0_{(10)}$	$0000\ 0000_{(2)}$
■ $-0_{(10)}$	$1111\ 1111_{(2)}$
■ $127_{(10)}$	$0111\ 1111_{(2)}$
■ $-128_{(10)}$	$1000\ 0000_{(2)}$
■ $-1_{(10)}$	$1111\ 1111_{(2)}$



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.

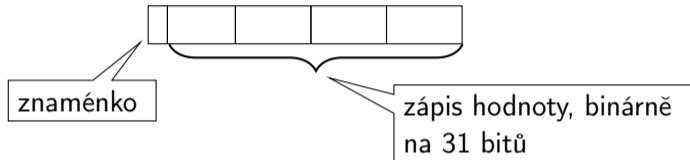
big-endian

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 $0111\dots111 = 2^{31} - 1 = 2\,147\,483\,647$.
 - Nejmenší číslo je $-2^{31} = -2\,147\,483\,648$.
- Pro zobrazení záporných čísel je použit **doplňkový kód**.

Reprezentujeme i nulu.

0 už je zahrnuta.

Nejmenší číslo v doplňkovém kódu $1000\dots000$ je -2^{31} .



Reprezentace záporných celých čísel

- Doplnkový 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	Doplnkový kód
0–127	0000 0000 – 0111 1111
128	nelze zobrazit na 8 bitů v doplnkovém 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



Necelá čísla a přesnost výpočtu 1/2

- Ztráta přesnosti při aritmetických operacích.

Příklad sčítání dvou čísel

```
#include <stdio.h>
int main(void)
{
    double a = 1e+10;
    double b = 1e-10;
    printf("a   : %24.12lf\n", a);
    printf("b   : %24.12lf\n", b);
    printf("a+b: %24.12lf\n", a + b);
    return 0;
}
clang sum.c && ./a.out
a   : 10000000000.000000000000
b   :                0.000000000100
a+b: 10000000000.000000000000
```

lec06/sum.c



Necelá čísla a přesnost výpočtu 2/2

Příklad dělení dvou čísel

```
#include <stdio.h>
int main(void)
{
    const int number = 100;
    double dV = 0.0;
    float fV = 0.0f;
    for (int i = 0; i < number; ++i) {
        dV += 1.0 / 10.0;
        fV += 1.0 / 10.0;
    }
    printf("double value: %lf ", dV);
    printf(" float value: %lf ", fV);
    return 0;
}
```

```
clang division.c && ./a.out
```

```
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 vyhradzujeme omezený paměťový prostor.

Příklad – zápis čísla $\frac{1}{3}$ v dekadické soustavě

- $= 33333333 \dots 3333$
- $= 0,3\bar{3}$
- $\approx 0,33333333333333333333$
- $\approx 0,333$

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

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

- 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.



- Maximální zobrazitelné kladné číslo $0,999z^{99}$.



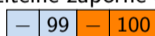
- Minimální zobrazitelné kladné číslo $0,100z^{-99}$.



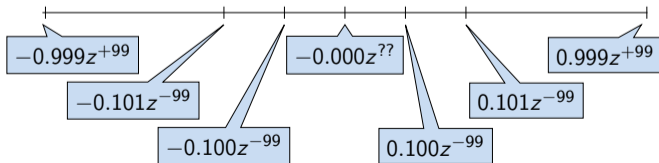
- Příklad $x = 77,5 = 0,775 \cdot z^{+02}$.



- Maximální zobrazitelné záporné číslo $-0,100z^{-99}$.

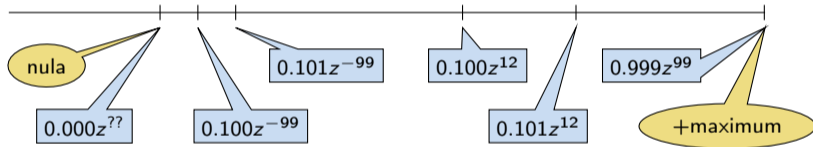


- Minimální zobrazitelné záporné číslo $-0,999z^{+99}$.

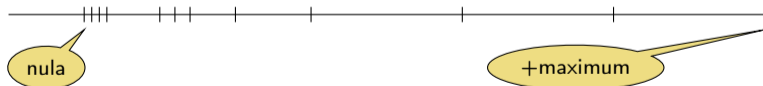


Model reprezentace reálných čísel a vzdálenost mezi aproximacemi

- Rozsah hodnot pro konkrétní exponent je dán velikostí mantisy.
- Absolutní vzdálenost dvou aproximací tak záleží na exponentu.
 - Mezi hodnotou 0 a 1,0 je využít 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

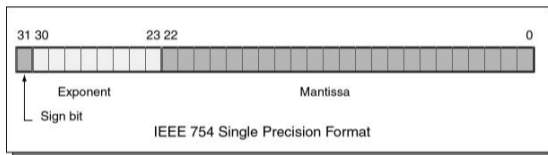
- Reálné číslo x se zobrazuje ve tvaru

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

Základ 2.

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. $\text{bias} = 2^{eb-1} - 1$, kde eb je počet bitů exponentu.



Příklad reprezentace float hodnot dle IEEE 754

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	-1	2^8	1.0029296875
Encoded as:	1	135	24576
Binary:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

You entered

Value actually stored in float: +1

Error due to conversion: -1

Binary Representation

Hexadecimal Representation

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	-1	2^8	1.0028905868530273
Encoded as:	1	135	24248
Binary:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/>

You entered

Value actually stored in float: +1

Error due to conversion: -1

Binary Representation

Hexadecimal Representation

- Chyba reprezentace -256.75 vs -256.74.
- **Infinity** (0x7f800000), **-Infinity** (0xff800000), a **NaN** (0xff7fffff).

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>



Obsah

Základní číselné typy a jejich reprezentace v počítači

Typové konverze

Matematické funkce



Přiřazovací operátor a příkaz

- Slouží pro nastavení hodnoty proměnné.

Uložení číselné hodnoty do paměti, kterou proměnná reprezentuje.

- Tvar přiřazovacího operátoru.

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

Výraz je literál, proměnná, volání funkce, ...

- Zkrácený zápis

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

- 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** 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
```

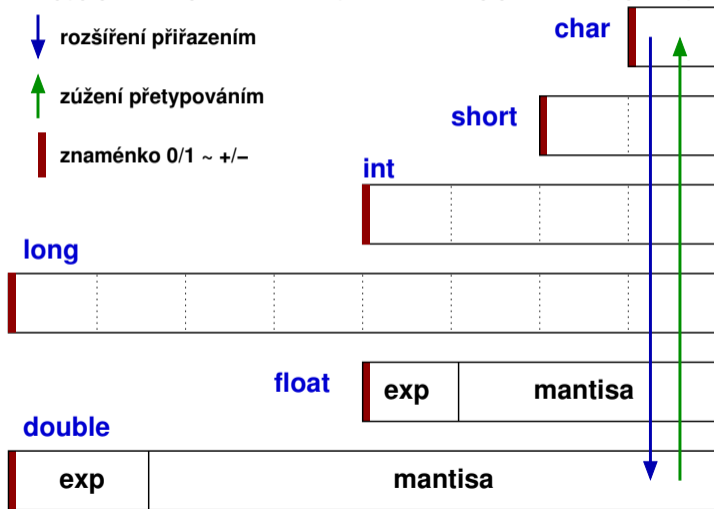
```
1 long l = 5000000000L;
2 int i = (int)l;
4 // i je 705032704
5 // (oříznuté 4 bajty)
```

lec06/demo-type_conversion.c



Konverze primitivních číselných typů

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



Obsah

Základní číselné typy a jejich reprezentace v počítači

Typové konverze

Matematické funkce



Matematické funkce

- `<math.h>` – základní funkce pro práci s „reálnými“ čísly.

- Výpočet odmocniny nečelého 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 III

Část 3 – Zadání 5. domácího úkolu (HW5)



Zadání 5. domácího úkolu HW5

Téma: Hledání textu v souborech

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

- **Motivace:** Dekomponovat výpočetní úlohu na dílčí výpočetní kroky.
- **Cíl:** Osvojit si práci se soubory.
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/bab36prga/hw/hw5>
 - Zpracování vstupu po řádcích a detekce textového řetězce ve vstupním souboru.
 - **Volitelné zadání** rozšiřuje úlohu o zpracování tří základních kvantifikátorů regulárních výrazů (pouze pro předcházející znak).
 - Znak pro kvantifikátory: $?$, $*$, $+$.
- **Termín odevzdání:** **13.04.2024, 23:59:59 PDT.**



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.



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 V

Appendix



Obsah

Kódovací příklad – Textové řetězce – `toupper()`

Kódovací příklad – Textové řetězce – `strrev()`

Kódovací příklad – Textové řetězce – `strwc()`

Kódovací příklad – Textové řetězce – `strsplit()`

Knihovna `strings.h`

„String objekt“



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", filename, line);
11        exit(-1);
12    }
13    return ret;
14 }
                                     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 prg!"; // 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 přepíšeme 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 prg!";
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 prg!
3 I LIKE PRG!

```

- 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
11    while (cur && *cur) {
12        *d = *cur++;
13        if (*d >= 'a' && *d <= 'z') {
14            *d = *d - 'a' + 'A';
15        }
16        d += 1;
17    }
18    *d = '\0'; // ret je vždy nejméně 1 byte dlouhý.
19    return ret;
20 }

```



Obsah

Kódovací příklad – Textové řetězce – `toupper()`

Kódovací příklad – Textové řetězce – `strrev()`

Kódovací příklad – Textové řetězce – `strwc()`

Kódovací příklad – Textové řetězce – `strsplit()`

Knihovna `strings.h`

„String objekt“



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 int main(void)
5 {
6     char *str = "I like prg!";
7     size_t j, n = strlen(str);
8     printf("%s\n", str);
9     for (size_t i = 0, j = n-1; i < n/2; ++i, --j) {
10         char t = str[i];
11         str[i] = str[j];
12         str[j] = t;
13     }
14     printf("%s\n", str);
15     return EXIT_SUCCESS;
16 }

```

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

```

- Program však skončí chybou! Zapisujeme do paměti literálů!

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

- Nahrazením ukazatele na literál polem, program funguje.

```

1 $ clang -g strrev.c && ./a.out
2 I like prg!
3 !prg 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 prg!";
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__,
24                          __LINE__);
25     const char *cur = str + n; // ukazatelová aritmetika
26     char *dst = ret;
27     while (str && cur != str) { // kontrola str!
28         *dst = *--cur;
29         dst += 1;
30     }
31     *dst = '\0'; //ret je vždy nejméně 1 byte dlouhý.
32     return ret;
33 }

```

- 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.
- Nicméně volíme přehlednost, menší počet řádků a jediný `return` ve funkci.



Obsah

Kódovací příklad – Textové řetězce – `toupper()`

Kódovací příklad – Textové řetězce – `strrev()`

Kódovací příklad – Textové řetězce – `strwc()`

Kódovací příklad – Textové řetězce – `strsplit()`

Knihovna `strings.h`

„String objekt“



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 prg!
3
4 $ clang -g wc.c && ./a.out < in.txt
5 Input contains 3 words.

```

- Po počátečním odladění implementujme 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 linecapp, 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
18     long stored in %lu bytes.\n", line, l, cap);
19     printf("Input contains %d words.\n", wc);
20     free(line); // proměnná je alokovaná 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 prg!"
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 prg!"
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!



Obsah

Kódovací příklad – Textové řetězce – `toupper()`

Kódovací příklad – Textové řetězce – `strrev()`

Kódovací příklad – Textové řetězce – `strwc()`

Kódovací příklad – Textové řetězce – `strsplit()`

Knihovna `strings.h`

„String objekt“



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 PRG
10     especially!";
11     char *s1, *s2;
12     char *delim = "and";
13     char *s = strstr(str, delim);
14
15     s1 = s2 = NULL;
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 PRG especially!"
7 s1: "I like programming "
8 s2: "and PRG especially!"

```



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 bool strsplit(const char *str, const char *delim, char
7             **s1, char **s2);
8 int main(void)
9 {
10     const char *str = "I like programming and PRG
11     especially!";
12     char *delim = "and";
13     char *s1, *s2;
14     strsplit(str, delim, &s1, &s2);
15     printf("String: \"%s\"\n", str);
16     printf("s1: \"%s\"\n", s1);
17     printf("s2: \"%s\"\n", s2);
18     free(s1); // it is ok to call free(NULL);
19     free(s2);
20     return EXIT_SUCCESS;
21 }

```

- 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     }
10
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 PRG especially!"
3 s1: "I like programming "
4 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`.



Obsah

Kódovací příklad – Textové řetězce – `toupper()`

Kódovací příklad – Textové řetězce – `strrev()`

Kódovací příklad – Textové řetězce – `strwc()`

Kódovací příklad – Textové řetězce – `strsplit()`

Knihovna `strings.h`

„String objekt“



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, char **s2);
11 #endif
                                     strings.h

```

```

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 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 #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 bytes.\n", line, l, cap);
13    printf("Input contains %d words.\n", wc);
14    free(line);
15    return EXIT_SUCCESS;
16 }
                                     demo-wc.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 prg!"
6 " that is 12 long stored in 16 bytes.
7 Input contains 3 words.

```



Obsah

Kódovací příklad – Textové řetězce – `toupper()`

Kódovací příklad – Textové řetězce – `strrev()`

Kódovací příklad – Textové řetězce – `strwc()`

Kódovací příklad – Textové řetězce – `strsplit()`

Knihovna `strings.h`

„String objekt“



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;
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 = 0, .getLength = &strlen };
20
21     string_create(&string, "I like PRG!");
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);
39     s->str = myMalloc((s->len + 1) * sizeof(char),
40                     __FILE__, __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 = 0;
50     }
51 }

```

```

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

```

