Ukazatele a dynamická alokace

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 05

BAB36PRGA – Programování v C

Přehled témat

- Část 1 – Ukazatele
  - Kódovací příklad – `hexdump`
  - Modifikátor `const` a ukazatele
  - Ukazatelová aritmetika
  - Ukazatel na funkci
  - Příklad - řazení

  S. G. Kochan: kapitola 10
- Část 2 – Dynamická alokace
  - Dynamická alokace paměti
  - Příklady

  S. G. Kochan: Příloha B
- Část 3 – Zadání 4. domácího úkolu (HW4)
- Část 4 – Shrnutí přednášky
- *Kódovací příklady*

Část I

Část 1 – Ukazatele

Obsah

Kódovací příklad – hexdump

Modifikátor const a ukazatele

Ukazatelová aritmetika

Ukazatel na funkci

Příklad - řazení

Kódovací příklad – hexdump – 1/4

- Implementujme program, který vytiskne vstup načtený ze `stdin` na `stdout` v **hex**a formátu.

Obdoba programu `hexdump`.

```
$ cat hw01-2.out
Desitkova soustava: 3759 -10000
Sestnactkova soustava: eaf ffffd8f0
Soucet: 3759 + -10000 = -6241
Rozdil: 3759 - -10000 = 13759
Soucin: 3759 * -10000 = -37590000
Podil: 3759 / -10000 = 0
Prumer: -3120.5
$ hexdump -C hw01-2.out
00000000  44 65 73 69 74 6b 6f 76  61 20 73 6f 75 73 74 61  |Desitkova sousta|
00000010  76 61 3a 20 33 37 35 39  20 2d 31 30 30 30 30 0a  |va: 3759 -10000.|
00000020  53 65 73 74 6e 61 63 74  6b 6f 76 61 20 73 6f 75  |Sestnactkova sou|
00000030  73 74 61 76 61 3a 20 65  61 66 20 66 66 66 66 64  |stava: eaf ffffd|
00000040  38 66 30 0a 53 6f 75 63  65 74 3a 20 33 37 35 39  |8f0.Soucet: 3759|
00000050  20 2b 20 2d 31 30 30 30  30 20 3d 20 2d 36 32 34  | + -10000 = -624|
00000060  31 0a 52 6f 7a 64 69 6c  3a 20 33 37 35 39 20 2d  |1.Rozdil: 3759 -|
00000070  20 2d 31 30 30 30 30 20  3d 20 31 33 37 35 39 0a  | -10000 = 13759.|
00000080  53 6f 75 63 69 6e 3a 20  33 37 35 39 20 2a 20 2d  |Soucin: 3759 * -|
00000090  31 30 30 30 30 20 3d 20  2d 33 37 35 39 30 30 30  |10000 = -3759000|
000000a0  30 0a 50 6f 64 69 6c 3a  20 33 37 35 39 20 2f 20  |0.Podil: 3759 / |
000000b0  2d 31 30 30 30 30 20 3d  20 30 0a 50 72 75 6d 65  |-10000 = 0.Prume|
000000c0  72 3a 20 2d 33 31 32 30  2e 35 0a                 |r: -3120.5.|
000000cb
```

- Program vypíše na `stdout` nejvýše **16 hodnot** na řádek, oddělených čárkou.

```
$ clang hexdump.c -o hexdump && ./hexdump < hw01-2.out
HEXDUMP:
44, 65, 73, 69, 74, 6b, 6f, 76, 61, 20, 73, 6f, 75, 73, 74, 61
76, 61, 3a, 20, 33, 37, 35, 39, 20, 2d, 31, 30, 30, 30, 30, 0a
53, 65, 73, 74, 6e, 61, 63, 74, 6b, 6f, 76, 61, 20, 73, 6f, 75
73, 74, 61, 76, 61, 3a, 20, 65, 61, 66, 20, 66, 66, 66, 66, 64
38, 66, 30, 0a, 53, 6f, 75, 63, 65, 74, 3a, 20, 33, 37, 35, 39
20, 2b, 20, 2d, 31, 30, 30, 30, 30, 20, 3d, 20, 2d, 36, 32, 34
31, 0a, 52, 6f, 7a, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2d
20, 2d, 31, 30, 30, 30, 30, 20, 3d, 20, 31, 33, 37, 35, 39, 0a
53, 6f, 75, 63, 69, 6e, 3a, 20, 33, 37, 35, 39, 20, 2a, 20, 2d
31, 30, 30, 30, 30, 20, 3d, 20, 2d, 33, 37, 35, 39, 30, 30, 30
30, 0a, 50, 6f, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2f, 20
2d, 31, 30, 30, 30, 30, 20, 3d, 20, 30, 0a, 50, 72, 75, 6d, 65
72, 3a, 20, 2d, 33, 31, 32, 30, 2e, 35, 0a
HEXDUMP END
```

- Na `stderr` vypíše na začátku `"HEXDUMP:\n"` a na konci `"HEXDUMP END\n"`.

Kódovací příklad – hexdump – 2/4

- Program napíšeme nejdříve kreativně, ale s počtem hodnot na řádek `MAX_WIDTH`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
4  #ifndef MAX_WIDTH
5  #define MAX_WIDTH 16
6  #endif
8  int main(void)
9  {
10    int ret = EXIT_SUCCESS;
11    int n = 0; // initialize
12    int c; // we do not need to init.
14    fprintf(stderr, "HEXDUMP:\n");
15    while ((c = getchar()) != EOF) {
16       if (n >= MAX_WIDTH) {
17          n = 0;
18          putchar('\n');
19       }
20       if (n > 0) {
21          printf(", ");
22       }
23       printf("%02x", c);
24       n += 1;
25    } //end while loop
26    if (n > 0) {
27       putchar('\n');
28    }
29    fprintf(stderr, "HEXDUMP END\n");
30    return ret;
31 }
```

Kódovací příklad – hexdump – 3/4

- Program upravíme redukcí počtu zanoření vyjmutím (dekompozice).

```
 8  void print_hex(int c, int *n);
10  int main(void)
11  {
12     int n = 0;
13     int c; // we do not need to init
15     fprintf(stderr, "HEXDUMP:\n");
16     while ((c = getchar()) != EOF) {
17        print_hex(c, &n);
18     } //end while loop
19     if (n > 0) {
20        putchar('\n'); // final EOL
21     }
22     fprintf(stderr, "HEXDUMP END\n");
23     return EXIT_SUCCESS; // always ok
24  }
```

```
15  void print_hex(int c, int *n)
16  {
17     if (*n >= MAX_WIDTH) {
18        *n = 0;
19        putchar('\n');
20     }
21     if (*n > 0) {
22        printf(", ");
23     }
24     printf("%02x", c);
25     *n += 1;
26  }
```

- Předávání ukazatele (adresy) proměnné `n` je nutné.
- Vyzkoušejte chování programu s předáváním hodnoty `n`.

```
void print_hex(int c, int n);
```

Kódovací příklad – hexdump – 4/4

- Program spustíme s přesměrováním `stdin` ze souboru, např. `hw01-2.out` a přesměrováním výstupu `stdout` do souboru `hex.out`.

```
$ clang hexdump-2.c -o hexdump && ./hexdump < hw01-2.out > hex.out
 HEXDUMP:
 HEXDUMP END
$ cat hex.out
44, 65, 73, 69, 74, 6b, 6f, 76, 61, 20, 73, 6f, 75, 73, 74, 61
76, 61, 3a, 20, 33, 37, 35, 39, 20, 2d, 31, 30, 30, 30, 30, 0a
53, 65, 73, 74, 6e, 61, 63, 74, 6b, 6f, 76, 61, 20, 73, 6f, 75
73, 74, 61, 76, 61, 3a, 20, 65, 61, 66, 20, 66, 66, 66, 66, 64
38, 66, 30, 0a, 53, 6f, 75, 63, 65, 74, 3a, 20, 33, 37, 35, 39
20, 2b, 20, 2d, 31, 30, 30, 30, 30, 20, 3d, 20, 2d, 36, 32, 34
31, 0a, 52, 6f, 7a, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2d
20, 2d, 31, 30, 30, 30, 30, 20, 3d, 20, 31, 33, 37, 35, 39, 0a
53, 6f, 75, 63, 69, 6e, 3a, 20, 33, 37, 35, 39, 20, 2a, 20, 2d
31, 30, 30, 30, 30, 20, 3d, 20, 2d, 33, 37, 35, 39, 30, 30, 30
30, 0a, 50, 6f, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2f, 20
2d, 31, 30, 30, 30, 20, 3d, 20, 30, 0a, 50, 72, 75, 6d, 65
72, 3a, 20, 2d, 33, 31, 32, 30, 2e, 35, 0a
```

- Obsah `stderr` není přesměrován, proto se vypíše na terminál.

- Při kompilaci definujeme počet hodnot na řádek `MAX_WIDTH=20` a program spustíme s přesměrováním `stderr` do souboru `hex.err`.

```
$ clang -DMAX_WIDTH=20 hexdump.c -o hexdump && ./hexdump <hw01-2.out 2> hex.err
44, 65, 73, 69, 74, 6b, 6f, 76, 61, 20, 73, 6f, 75, 73, 74, 61, 76, 61, 3a, 20
33, 37, 35, 39, 20, 2d, 31, 30, 30, 30, 30, 0a, 53, 65, 73, 74, 6e, 61, 63, 74
6b, 6f, 76, 61, 20, 73, 6f, 75, 73, 74, 61, 76, 61, 3a, 20, 65, 61, 66, 20, 66
66, 66, 66, 64, 38, 66, 30, 0a, 53, 6f, 75, 63, 65, 74, 3a, 20, 33, 37, 35, 39
20, 2b, 20, 2d, 31, 30, 30, 30, 30, 20, 3d, 20, 2d, 36, 32, 34, 31, 0a, 52, 6f
7a, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2d, 20, 2d, 31, 30, 30, 30, 30, 20
3d, 20, 31, 33, 37, 35, 39, 0a, 53, 6f, 75, 63, 69, 6e, 3a, 20, 33, 37, 35, 39
20, 2a, 20, 2d, 31, 30, 30, 30, 30, 20, 3d, 20, 2d, 33, 37, 35, 39, 30, 30, 30
30, 0a, 50, 6f, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2f, 20, 2d, 31, 30, 30
30, 30, 20, 3d, 20, 30, 0a, 50, 72, 75, 6d, 65, 72, 3a, 20, 2d, 33, 31, 32, 30
2e, 35, 0a
$ cat hex.err
HEXDUMP:
HEXDUMP END
```

- Obsah `stderr` je přesměrování, proto se nevypíše na terminál.

Vyzkoušejte program s přesměrování binárního souboru na stdin našeho `hexdump`.

Obsah

Kódovací příklad – hexdump

Modifikátor const a ukazatele

Ukazatelová aritmetika

Ukazatel na funkci

Příklad - řazení

Modifikátor typu const

- Uvedením klíčového slova **const** můžeme označit proměnnou jako konstantu.

Překladač nás kontroluje, zdali se snažíme hodnotu proměnné změnit.

- Definovat konstantu můžeme např.

```
const float pi = 3.14159265f;
```

- Symbolická konstanta

```
#define PI 3.14159265
```

- je pojmenování literálu, ve zdrojovém souboru je výkyt `PI` textově nahrazen literálem.

Připomínka

Ukazatele na konstantní proměnné a konstantní ukazatele

- Klíčové slovo **const** můžeme zapsat před jméno proměnné nebo před `*` (typ/).
- Dostáváme 3 možnosti jak definovat ukazatel s `const`.
  (a) `const int *ptr;` – ukazatel na konstantní proměnnou.
    - Nemůžeme použít pointer pro změnu hodnoty proměnné.

  (b) `int *const ptr;` – konstantní ukazatel (`const` před jménem proměnné a mezi `*`).
    - Pointer nemůžeme nastavit na jinou adresu než tu při inicializaci.

  (c) `const int *const ptr;` – konstantní ukazatel na konstantní hodnotu.
    - Kombinuje předchozí dva případy.

  `lec05/const_pointers.c`

  Další alternativy zápisu (a) a (c) jsou
  - `const int *` lze též zapsat jako `int const *`; `const` *je stále před* `*`.
  - `const int * const` lze též zapsat jako `int const * const`.

  `const` může být vlevo nebo vpravo od jména typu.

- Nebo komplexnější definice, např. `int ** const ptr;` – konstantní ukazatel na ukazatel na `int`.

Příklad – Ukazatel na konstantní proměnnou (hodnotu)

- Prostřednictvím ukazatele na konstantní proměnnou nemůžeme tuto proměnnou měnit.

```
1  int v = 10;
2  int v2 = 20;
4  const int *ptr = &v;  // ptr cannot be used to modify v
5  printf("*ptr: %d\n", *ptr);
7  *ptr = 11; /* IT IS NOT ALLOWED! */
9  v = 11;  /* We can modify the original variable */
10 printf("*ptr: %d\n", *ptr);
12 ptr = &v2; /* We can assign new address to ptr */
13 printf("*ptr: %d\n", *ptr);
```

lec05/const_pointers.c

Příklad – Konstantní ukazatel

- Hodnotu konstantního ukazatele nelze po inicializaci měnit.
- Zápis `int *const ptr;` můžeme číst zprava doleva:
  - `ptr` – proměnná, která je;
  - `*const` – konstantním ukazatelem;
  - `int` – na proměnnou typu `int`.

```
1  int v = 10;
2  int v2 = 20;
3  int *const ptr = &v;
4  printf("v: %d *ptr: %d\n", v, *ptr);

6  *ptr = 11; /* We can modify addressed value */
7  printf("v: %d\n", v);

9  ptr = &v2; /* IT IS NOT ALLOWED! */
```

lec05/const_pointers.c

Příklad – Konstantní ukazatel na konstantní proměnnou

- Hodnotu konstantního ukazatele na konstantního proměnnou nelze po inicializaci měnit a ani nelze prostřednictvím takového ukazatele měnit hodnotu adresované proměnné.
- Zápis `const int *const ptr;` čteme "zprava doleva":
  - `ptr` – proměnná, která je;
  - `*const` – konstantním ukazatelem;
  - `const int` – na proměnnou typu `const int`.

```
1  int v = 10;
2  int v2 = 20;
3  const int *const ptr = &v;
5  printf("v: %d *ptr: %d\n", v, *ptr);
7  ptr = &v2; /* IT IS NOT ALLOWED! */
8  *ptr = 11; /* IT IS NOT ALLOWED! */
```

lec05/const_pointers.c

Konstantní ukazatel (na konstantní hodnotu)

Příklad	Konstatní hodnota	Konstantní ukazatel	Popis *„Čtu zprava doleva.“*
`char *ptr`	Ne	Ne	„`ptr` je ukazatel (*) na hodnotu `char`.“
`const char *ptr`	**Ano**	Ne	„`ptr` je ukazatel na hodnotu `char` konstantní.“
`char const *ptr`	**Ano**	Ne	„`ptr` je ukazatel na konstantní hodnotu `char`.“
`char* const ptr`	Ne	**Ano**	„`ptr` je konstantní ukazatel na hodnotu `char`.“
`const char *const ptr`	**Ano**	**Ano**	„`ptr` je konstantní ukazatel na hodnotu `char` konstantní.“

- *Konstantní ukazatel je proměnná, jejíž hodnotu nemohu měnit. Ukazatel odkazuje na (stejné) paměťové místo, které mohu případně měnit.*
- *Konstantní hodnotu nemohu měnit. Tedy nemohu měnit obsah paměťového místa, na které odkazuje ukazatel (jehož adresa je uloženo v proměnné typu ukazatel).*

Definice typu – typedef

- Operátor `typedef` umožňuje definovat nový datový typ.
- Slouží k pojmenování typů, např. ukazatele, struktury a uniony.

Struktury a uniony viz přednáška 6.

- Například typ pro ukazatele na `double` a nové jméno pro `int`:

```
1 typedef double* double_p;
2 typedef int integer;
3 double_p x, y;
4 integer i, j;
```

- je totožné s použitím původních typů

```
1 double *x, *y;
2 int i, j;
```

- Zavedením typů operátorem `typedef`, např. v hlavičkovém souboru, umožňuje systematické používání nových jmen typů v celém programu. *Viz např.* `<inttypes.h>`.
- Výhoda zavedení nových typů je především u složitějších typů jako jsou ukazatele na funkce nebo struktury.

Obsah

Kódovací příklad – hexdump

Modifikátor const a ukazatele

Ukazatelová aritmetika

Ukazatel na funkci

Příklad - řazení

Ukazatelová (pointerová) aritmetika

- S ukazately (pointery) lze provádět aritmetické operace + a − (přičítat nebo odčítat celé číslo).
  - ukazatel = ukazatel stejného typu + (nebo −) a celé číslo (int).
  - Nebo lze používat zkrácený zápis např. `ukazatel += 1` a unární operátory např. `ukazatel++`.
- Aritmetické operace jsou užitečné pokud ukazatel odkazuje na více položek daného typu (souvislý blok paměti).
  - Např. pole položek příslušného typu;
  - Dynamicky alokovaný souvislý blok paměti.
- Přičtením hodnoty celého čísla k pointeru „posouváme“ hodnotu pointeru na další prvek, např.

```
int a[10];
int *p = a;
int i = *(p+2); //odkazuje na hodnotu 3.  prvku pole a
```

  - Podle typu ukazatele se hodnota adresy příslušně zvýší.
  - `(p+2)` je ekvivalentní adrese `p + 2*sizeof(int)`.
  - Příklad použití viz `lec05/pointers_and_array.c`

Příklad ukazatelové aritmetiky

- Ukazatel je proměnná jejíž hodnota je adresa v paměti.
- Uazatelová aritmetika zoohledňuje typ proměnné, její velikost v paměti.
- Přičtením hodnoty `1` k proměnné typu ukazatel je vypočtena adresa následujícího prvku, adresa je zvětšena o hodnotu odpovídající `sizeof()` příslušného typu.

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

```c
int getLengthPtr1(char *str)
{
    int ret = 0;
    while (str && (*str++) != '\0') {
        ret += 1;
    }
    return ret;
}
```

```c
int getLengthPtr2(char *str)
{
    int ret = 0;
    while (str && (*str++)) ret += 1;
    return ret;
}
```

- Textový řetězec můžeme interpretovat jako pole znaků `char[]` nebo ukazatel `char*`.

lec05/string_length.c

Obsah

Kódovací příklad – hexdump

Modifikátor const a ukazatele

Ukazatelová aritmetika

Ukazatel na funkci

Příklad - řazení

Ukazatel na funkci

- Implementace funkce je umístěna někde v paměti a podobně jako na proměnnou v paměti může ukazatel odkazovat na paměťové místo s definicí funkce.
- Můžeme definovat **ukazatel na funkci** a dynamicky volat funkci dle aktuální hodnoty ukazatele.
- Součástí volání funkce jsou předávané argumenty, které jsou též součástí typu ukazatele na funkci, resp. typy argumentů.
- Funkce (a volání funkce) je identifikátor funkce a (), tj.

```
typ_návratové_hodnoty funkce(argumenty funkce);
```

- Ukazatel na funkci definujeme jako

```
typ_návratové_hodnoty (*ukazatel)(argumenty funkce);
```

Příklad – Ukazatel na funkci 1/2

- Používáme dereferenční operátor `*` podobně jako u proměnných.

```
double do_nothing(int v); /* function prototype */
double (*function_p)(int v); /* pointer to function */
function_p = do_nothing; /* assign the pointer */
(*function_p)(10); /* call the function */
```

- Závorky `(*function_p)` „*pomáhají*“ číst definici ukazatele.

  Můžeme si představit, že závorky reprezentují jméno funkce. Definice proměnné ukazatel na funkci se tak v zásadě neliší od prototypu funkce.

- Podobně je volání funkce přes ukazatel na funkci identické běžnému volání funkce, kde místo jména funkce vystupuje jméno ukazatele na funkci.

Příklad – Ukazatel na funkci 2/2

- V případě funkce vracející ukazatel postupujeme identicky.

```
double* compute(int v);
double* (*function_p)(int v);
        ^^^^^^^^^^^^^---- substitute a function name
function_p = compute;
```

- Příklad použití ukazatele na funkci – `lec05/pointer_fnc.c`
- Ukazatele na funkce umožňují realizovat dynamickou vazbu volání funkce identifikované za běhu programu. *V objektově orientovaném programování je dynamická vazba klíčem k realizaci polymorfismu.*

Ukazatel na funkci se může hodit v implementaci HW4 povinné a bonusové zadání. Při vhodném návrhu programu je základní část společná, „jen" zaměníme funkci pro porovnávání dvou řetězců s využitím Hammnigovy nebo Levenštejnovy vzdálenosti. V případě obou funkcí může být vstup dva textové řetězce, případně včetně délky. Tedy můžeme jednoduše zaměnit ukazatel na funkci.

Příklad použití ukazatele na funkci

- Vhodným využitím ukazatele na funkci je zajištění přístupu k datům pro jinak naprosto identický algoritmus, jako je řazení (funkce `qsort` z `stdlib.h`). *Zejména pro pole hodnot složeného typu.*

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

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

void print(int n, int array[n]);
int compare(const void *pa, const void *pb);

int main(void)
{
    const int n = 10;
    int array[n];
    for (int i = 0; i < n; ++i) {
        array[i] = rand() % 100;
    }
    print(n, array);
    qsort(array, n, sizeof(array[0]), compare);
    print(n, array);
    return 0;
}

void print(int n, int array[n])
{
    for(int i = 0; i < n; ++i) {
        i > 0 ? printf(", ") : 0;
        printf("%d", array[i]);
    }
    n > 0 ? putchar('\n') : 0;
}

int compare(const void *pa, const void *pb)
{
    const int a = *(int*)pa;
    const int b = *(int*)pb;
    return (a < b) - (a > b);
}
```

lec05/demo-pointer_fnc.c

Obsah

Kódovací příklad – hexdump

Modifikátor const a ukazatele

Ukazatelová aritmetika

Ukazatel na funkci

Příklad - řazení

Kódovací příklad – Pole a ukazatel na funkci 1/4

- Implementujte program, který vytvoří pole náhodných kladných, celých čísel voláním funkce `rand()` z `stdlib.h`. *Funkce fill.*
- Hodnota celých čísel je omezena na `MAX_NUM`, např. nastavena na 20, `#define MAX_NUM 20`.
- Počet náhodných čísel `LEN` může být nastaven při kompilaci – `clang -DLEN=10 program.c`.
- Pole je vypsáno na `stdout`. *Funkce print.*
- Pole je uspořádáno funkcí `qsort()` ze `stdlib.h`. *Seznamte s funkcí, viz* `man qsort`.
- Uspořádané pole je vypsáno na `stdout`.
- Program je dále rozšířen o zpracování argumentu programu definujícího počet náhodných čísel s využitím funkce `atoi()`.

```c
#ifndef LEN
#define LEN 5
#endif
#define MAX_NUM 20
void fill_random(size_t l, int a[l]);
void print(const char *s, size_t l, int a[l]);
int main(void)
{
    int a[LEN]; // allocate the array
    fill_random(LEN, a); // fill the array
    print("Array random: ", LEN, a);
    // TODO call qsort
    print("Array sorted: ", LEN, a);
    return 0;
}
```

Kódovací příklad – Pole a ukazatel na funkci 2/4

```
void fill_random(size_t l, int a[l])
{
   for (size_t i = 0; i < l; ++i) {
      a[i] = rand() % MAX_NUM;
   }
}
void print(const char *s, size_t l, int a[l])
{
   if (s) {
      printf("%s", s);
   }
   for (size_t i = 0; i < l; ++i) {
      printf("%s%d", i > 0 ? " " : "", a[i]);
   }
   putchar('\n');
}
```

- Vizte `man qsort`.

```
void qsort(
  void *base, size_t nmemb, size_t size,
  int (*compar)(const void *, const void *)
);
```

  - `base` je ukazatel na první prvek;
  - `nmemb` je počet prvků;
  - `size` je velikost (každého) prvku;
  - `compar` je ukazatel na funkci porovnání.

```
int compare(const void *ai, const void *bi)
{
   const int *a = (const int*)ai;
   const int *b = (const int*)bi;
   //ascending
   return *a == *b ? 0 : (*a < *b ? -1 : 1);
}
```

Změňte pořadí na sestupné.

Kódovací příklad – Pole a ukazatel na funkci 3/4

- Název funkce použijte jako ukazatel na funkci.

```
int compare(const void *, const void *);
int main(void)
{
   int a[LEN]; // do not initialize
   fill_random(LEN, a);
   print("Array random: ", LEN, a);
   qsort(a, LEN, sizeof(int), compare);
   print("Array sorted: ", LEN, a);
   return 0;
}
```

- Kompilujte a spusťte program pouze pokud byla kompilace úspěšná použitím `shell` **logický and** operátor `&&`.

```
$ clang sort.c -o sort && ./sort
Array random: 13 17 18 15 12
Array sorted: 12 13 15 17 18
```

- Použijte argument kompilátoru `-DLEN=10` k definici velikosti pole 10.

```
$ clang -DLEN=10 sort.c -o sort && ./sort
Array random: 13 17 18 15 12 3 7 8 18 10
Array sorted: 3 7 8 10 12 13 15 17 18 18
```

Kódovací příklad – Pole a ukazatel na funkci 4/4

- Rozšiřte `main()` o předání argumentů.
- Definujte návratovou hodnotu při chybě.

```
1  enum { ERROR = 100 };
3  int main(int argc, char *argv[])
4  {
5     const size_t len = argc > 1 ?
6        atoi(argv[1]) : LEN;
7     if (len > 0) {
8        int a[len];
9        fill_random(len, a);
10       print("Array random: ", len, a);
11       qsort(a, len, sizeof(int), compare);
12       print("Array sorted: ", len, a);
13    }
14    return len > 0 ? EXIT_SUCCESS : ERROR;
15 }
```

- Použijeme **Variable Length Array** (**VLA**), které umožňuje definovat velikost pole za běhu.

```
$ clang sort-vla.c -o sort && ./sort
Array random: 13 17 18 15 12 3
Array sorted: 3 12 13 15 17 18
$ clang sort-vla.c -DLEN=7 -o sort && ./sort
Array random: 13 17 18 15 12 3 7
Array sorted: 3 7 12 13 15 17 18
$ clang sort-vla.c -o sort && ./sort 11
Array random: 13 17 18 15 12 3 7 8 18 10 19
Array sorted: 3 7 8 10 12 13 15 17 18 18 19
```

- Uvědomte si, že velikost pole `a` je omezena velikostí **zásobníku**, viz `ulimit -s`.

Část II

Část 2 – Dynamická alokace

Obsah

Dynamická alokace paměti

Příklady

Dynamická alokace paměti

- Přidělení bloku paměti velikosti `size` lze realizovat funkcí

```
void* malloc(size);
```

Z knihovny <stdlib.h>

  - Velikost alokované paměti je uložena ve správci paměti.
  - **Velikost není součástí ukazatele**.
  - Návratová hodnota je typu `void*` – přetypování nutné/vhodné.
  - Je plně na uživateli (programátorovi), jak bude s pamětí zacházet.
- Příklad alokace paměti pro 10 proměnných typu `int`.

```
1 int *int_array;
2 int_array = (int*)malloc(10 * sizeof(int));
```

- Operace s více hodnotami v paměťovém bloku je podobná poli.
  - Používáme pointerovou aritmetiku.
- **Uvolnění paměti**

```
void free(pointer);
```

  - Správce paměti uvolní paměť asociovanou k ukazateli.
  - Hodnotu ukazatele však nemění!

Stále obsahuje předešlou adresu, která však již není platná.

Obsah

Dynamická alokace paměti

Příklady

Příklad alokace dynamické paměti 1/3

- Alokace se nemusí nutně povést – testujeme návratovou hodnotu funkce `malloc()`.
- Pro vyplnění adresy alokované paměti předáváme proměnnou jako ukazatel na proměnnou typu ukazatel na `int`.

```
1  void* allocate_memory(int size, void **ptr)
2  {
3     // use **ptr to store value of newlly allocated
4     // memory in the pointer ptr (i.e., the address the
5     // pointer ptr is pointed).
8     // call library function malloc to allocate memory
9     *ptr = malloc(size);
11    if (*ptr == NULL) {
12       fprintf(stderr, "Error: allocation fail");
13       exit(-1); /* exit program if allocation fail */
14    }
15    return *ptr;
16 }
```

lec05/malloc_demo.c

Příklad alokace dynamické paměti 2/3

- Pro vyplnění hodnot pole alokovaného dynamicky nám postačuje předávat hodnotu adresy paměti pole.

```
1 void fill_array(int size, int* array)
2 {
3    for (int i = 0; i < size; ++i) {
4       *(array++) = random();
5    }
6 }
```

- Po uvolnění paměti odkazuje ukazatel stále na původní adresu, proto můžeme explicitně nulovat.

Předání ukazatele na ukazatele je nutné, jinak nemůžeme nulovat.

```
1 void deallocate_memory(void **ptr)
2 {
3    if (ptr != NULL && *ptr != NULL) {
4       free(*ptr);
5       *ptr = NULL;
6    }
7 }
```

lec05/malloc_demo.c

Příklad alokace dynamické paměti 3/3

```
1  int main(int argc, char *argv[])
2  {
3      int *int_array;
4      const int size = 4;
6      allocate_memory(sizeof(int) * size, (void**)&int_array);
7      fill_array(int_array, size);
8      int *cur = int_array;
9      for (int i = 0; i < size; ++i, cur++) {
10         printf("Array[%d] = %d\n", i, *cur);
11     }
12     deallocate_memory((void**)&int_array);
13     return 0;
14 }
```

lec05/malloc_demo.c

Příklad - Načítání textového řetězce 1/3

- Implementujete načtení libovolně dlouhého řádku ze `stdin`.
- Řádek je zakončen znakem nového řádku `'\n'`, který není součástí načteného vstupu.
- Reportujte chybové stavy `ERROR_IN = 100` a `ERROR_MEM = 101`.
- Po úspěšném načtení vstupu, reportujte velikost vstupu voláním funkce `strlen()` z `string.h`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifndef INIT_SIZE
#define INIT_SIZE 128
#endif
enum {
   ERROR_OK = EXIT_SUCCESS,
   ERROR_IN = 100,
   ERROR_MEM = 101,
};
char* read(int *error);
char* enlarge_string(size_t len, size_t *
    capacity, char *str);
```

```c
int main(int argc, char *argv[])
{
   int ret = EXIT_SUCCESS;
   char *str = read(&ret);
   if (str) {
      printf("Input string size %ld\n", strlen(str));
      printf("Input string \"%s\"\n", str);
      free(str);
   } else {
      fprintf(stderr, "ERROR: read return %d\n", ret);
   }
   return ret;
}
```

lec05/read.c

Příklad - Načítání textového řetězce 2/4

```
31 // local function only for calling from read()
32 static char* handle_str(char r, size_t l,
33       char *str, int *error);
34 char* read(int *error)
35 {
36    size_t capacity = INIT_SIZE;
37    size_t l = 0; // no.  of read chars
38    char* str = malloc(capacity + 1);
39    int r = '\0';
40    while (
41          str
42          && *error == ERROR_OK
43          && (r = getchar()) != EOF
44          && r != '\n'
45          ) {
46       if (l == capacity) { // enlarge if need
47          // new address of str can be set
48          str = enlarge_string(l, &capacity, str);
49       }
50       //Is it correct?  Can str be NULL?
51       str[l++] = r;
52    } // end while
53    str = handle_str(r, l, str, error);
54    return str;
55 }
```

```
57 char* handle_str(char r, size_t l, char *str, int *error)
58 {
59    if (str) {
60       if (r != '\n') { // end-of-line has not been read
61          *error = ERROR_IN; // report input error
62          free(str);
63          str = NULL;
64       } else {
65          str[l] = '\0'; // null terminating string
66       }
67    } else if (*error == ERROR_OK) { // str is NULL
68       *error = ERROR_MEM; // but error needs to be set
69    }
70    return str;
71 }
73 char* enlarge_string(size_t len, size_t *capacity, char *str)
74 {
75    char *t = realloc(str, *capacity * 2 + 1);
76    if (!t) {
77       free(str);
78       str = NULL; // indicate error
79    } else {
80       str = t;
81       *capacity *= 2;
82    }
83    return str;
84 }
```

lec05/read.c

Příklad - Načítání textového řetězce 3/4

- Příklad vstupu programu `clang read.c -o read`.
- Vstup soubor `read-in-1.txt`.

```
./read <read_in-1.txt; echo $?
Input string size 11
0
hexdump -C read_in-1.txt
00000000  49 20 6c 69 6b 65 20 70  72 67 21 0a              |I like prg!.|
0000000c
```

lec05/read_in-1.txt

- Vstup soubor `read-in-2.txt`.

```
./read <read_in-2.txt; echo $?
ERROR: read return 100
100
hexdump -C read_in-2.txt
00000000  49 20 6c 69 6b 65 20 70  72 67 21                 |I like prg!|
0000000b
```

lec05/read_in-2.txt

Příklad - Načítání textového řetězce 4/4

- Generování náhodného vstupu.

```
cat /dev/urandom | env LC_ALL=C tr -dc 'a-zA-Z0-9' | fold -w 10485760 | head -n 1
```

lec05/create_rand_string.sh

- Omezení paměti programu.

```
clang read.c -o read
./create_rand_string.sh >10MB.txt
du -h 10MB.txt
10M 10MB.txt
./read <10MB.txt
Input string size 10485760
```

```
ulimit -v 10240
./read <10MB.txt; echo $?
ERROR: read return 101
101
```

lec05/read.c

Část III

Část 3 – Zadání 4. domácího úkolu (HW4)

Zadání 4. domácího úkolu HW4

Téma: **Caesarova šifra** Povinné zadání: **3b**; Volitelné zadání: není; Bonusové zadání: **5b**

- **Motivace**: Získat zkušenosti s dynamickou alokací paměti. Implementovat výpočetní úlohu optimalizačního typu.
- **Cíl**: Osvojit si práci s dynamickou alokací paměti.
- **Zadání**: `https://cw.fel.cvut.cz/wiki/courses/bab36prga/hw/hw4`
  - Načtení dvou vstupních textů a tisk dekódované zprávy na výstup.
  - Zakódovaný text i (špatně) odposlechnutý text mají stejné délky.
  - Nalezení největší shody dekódovaného a odposlechnutého textu na základě hodnoty posunu v Caesarově šifře.
  - Optimalizace hodnoty Hammingovy vzdálenosti.

    `https://en.wikipedia.org/wiki/Hamming_distance`
  - Volitelné zadání rozšiřuje úlohu o uvažování chybějících znaků v odposlechnutém textu, což vede na využití Levenštejnovy vzdálenosti.

    `https://en.wikipedia.org/wiki/Levenshtein_distance`
- Termín odevzdání: 12.04.2025, 23:59:59 PDT.
- Bonusová úloha: 23.05.2025, 23:59:59 CEST.

Shrnutí přednášky

Diskutovaná témata

- Ukazatele a modifikátor `const`
- Dynamická alokace paměti
- Ukazatel na funkce

- Příště: Příklady dynamické alokace, práce se soubory, struktur a uniony.

Diskutovaná témata

- Ukazatele a modifikátor `const`
- Dynamická alokace paměti
- Ukazatel na funkce

- Příště: Příklady dynamické alokace, práce se soubory, struktur a uniony.

Část V

Kódovací příklady

Obsah

Kódovací příklad – Textové řetězce – toupper()

Kódovací příklad – Textové řetězce – strrev()

Kódovací příklad – Textové řetězce – strwc()

Kódovací příklad – Textové řetězce – strsplit()

Knihovna strings.h

NATO Abeceda

NATO Abeceda („jinak“)

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!
```

```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;
}
```

- 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'`).

Obsah

Kódovací příklad – Textové řetězce – toupper()

Kódovací příklad – Textové řetězce – strrev()

Kódovací příklad – Textové řetězce – strwc()

Kódovací příklad – Textové řetězce – strsplit()

Knihovna strings.h

NATO Abeceda

NATO Abeceda („jinak“)

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;
}
```

```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;
}
```

- 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.

- 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.

Obsah

Kódovací příklad – Textové řetězce – toupper()

Kódovací příklad – Textové řetězce – strrev()

Kódovací příklad – Textové řetězce – strwc()

Kódovací příklad – Textové řetězce – strsplit()

Knihovna strings.h

NATO Abeceda

NATO Abeceda („jinak“)

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>
5  #include <ctype.h>
7  int main(void)
8  {
9     int c, wc = 0;
10    bool inword = false;
12    while ((c = getchar()) != EOF) {
13       if (isalpha(c)) {
14          if (!inword) {
15             inword = true;
16             wc += 1;
17          }
18       } else {
19          inword = false;
20       }
21    }
22    printf("Input contains %d words.\n", wc);
23    return EXIT_SUCCESS;
24 }
```

- Řádky 14–17 můžeme nahradit následujícím řádkem.

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

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

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

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

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

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

```
ssize_t getline(char ** restrict linep, size_t * restrict 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!

Obsah

Kódovací příklad – Textové řetězce – toupper()

Kódovací příklad – Textové řetězce – strrev()

Kódovací příklad – Textové řetězce – strwc()

Kódovací příklad – Textové řetězce – strsplit()

Knihovna strings.h

NATO Abeceda

NATO Abeceda („jinak“)

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`.

Obsah

Kódovací příklad – Textové řetězce – toupper()

Kódovací příklad – Textové řetězce – strrev()

Kódovací příklad – Textové řetězce – strwc()

Kódovací příklad – Textové řetězce – strsplit()

Knihovna strings.h

NATO Abeceda

NATO Abeceda („jinak“)

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`.

```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.h

```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
```

strings.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;
}
```

demo-wc.c

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

Obsah

Kódovací příklad – Textové řetězce – toupper()

Kódovací příklad – Textové řetězce – strrev()

Kódovací příklad – Textové řetězce – strwc()

Kódovací příklad – Textové řetězce – strsplit()

Knihovna strings.h

NATO Abeceda

NATO Abeceda („jinak“)

Kódovací příklad – NATO Abeceda – 1/4

- Implementujme program, který převede vstupní text (ASCII, znaky A–Z a a–z) do NATO abecedy, ve které jsou písmena hláskována prostřednictvím následujících jmen.
  - **A**lpha, **B**ravo, **C**harlie, **D**elta, **E**cho, **F**oxtrot, **G**olf, **H**otel, **I**ndia, **J**uliett, **K**ilo, **L**ima, **M**ike, **N**ovember, **O**scar, **P**apa, **Q**uebec, **R**omeo, **S**ierra, **T**ango, **U**niform, **V**ictor, **W**hiskey, **X**-ray, **Y**ankee, **Z**ulu.
- V programu definujeme pole ukazatelů na textové literály s jednotlivými slovy.
- Programově otestujeme, že slova odpovídají počátečním písmenům A–Z.

- Očekávaný výstup pro vstup `in.txt`.

```
$ cat in.txt
I like PRG and programming in C.
$ clang nato-alphabet.c && ./a.out < in.txt 2>/dev/null
India Lima India Kilo Echo Papa Romeo Golf Alpha November Delta Papa
      Romeo Oscar Golf Romeo Alpha Mike Mike India November Golf
     India November Charlie
```

- Implementujeme testovací funkce.

```
1 static char *words[] = { // static to be "private"
2    "Alpha", "Bravo", "Charlie", "Delta", "Echo", "Foxtrot", "
     Golf", "Hotel", "India", "Juliett", "Kilo", "Lima", "Mike
     ", "November", "Oscar", "Papa", "Quebec", "Romeo", "
     Sierra", "Tango", "Uniform", "Victor", "Whiskey", "X-ray"
     , "Yankee", "Zulu", NULL
3 }; // it is an array of pointers to text literls
5 int count_words_array(char *words[]); // Using array
6 int count_words(char **words); // Pointer to pointers
7 bool check_alphabet_words(char *words[]);
```

Kódovací příklad – NATO Abeceda – 2/4

```c
// array is termined by NULL used for counting
static char *words[] = { "Alpha",  ..,  "Zulu", NULL };
// array-like variant
int count_words_array(char *words[])
{
   int n = 0;
   while(words[n] != NULL) {
      fprintf(stderr, "DEBUG: \"%s\"\n", words[n]);
      n += 1;
   }
   return n;
}
// pure pointer variant
int count_words(char **words)
{
   int n = 0;
   char **cur = words;
   while (*cur) {
      fprintf(stderr, "DEBUG: \"%s\"\n", *cur);
      cur++;
      n += 1;
   }
   return c;
}
```

```c
bool check_alphabet_words(char *words[])
{
   bool ret = true; // true is from #include <stdbool.h>
   char c = 'A'; // char is an integer ASCII code number
   char **cur = &words[0]; // there is always at least one item
   while (*cur) {
      fprintf(stderr, "DEBUG: check %s[0] for '%c'\n", *cur, c);
      if (c != *cur[0]) { // the first letter needs to match
         ret = false; // false is from #include <stdbool.h>
         break;
      } else {
         c += 1;
         cur += 1;
      }
   }
   return ret;
}
```

- Pole `words` je posloupnost prvků stejného typu (ukazatel na `char` – textový řetězec).
- Hodnota `&words[0]` je identická adresa jako hodnota `words`.

Kódovací příklad – NATO Abeceda – 3/4

- Můžeme použít const.

```c
static const char * const words[] = { "Alpha", ..., NULL };
int count_words_array(const char * const words[])
{
   int n = 0;
   while(words[n] != NULL) {
      n += 1;
   }
   return n;
}
int count_words(const char * const * const words)
{
   int n = 0;
   // cur je ukazatel na data typu konstantní
   // ukazatel na konstantní textový řetězec
   // (na konstantní ukazatel na konstantní hodnoty char).
   const char * const * cur = words; // cur chceme měnit
   while (*cur) {
      cur++; // cur není konstantní ukazatel
      n += 1;
   }
   return n;
}
```

```c
#include <stdio.h>
#include <stdbool.h>
static const char * const words[] = { "Alpha",.., NULL };
int count_words_array(const char * const words[]);
int count_words(const char * const * const words);
bool check_alphabet_words(const char * const words[]);
int main(void)
{
   int ret = EXIT_SUCCESS;
   fprintf(stderr, "DEBUG: size %lu\n", sizeof(words));
   int n = count_words_array(words);
   fprintf(stderr, "DEBUG: no. of words: %i\n", n);
   n = count_words(&words[0]);
   fprintf(stderr, "DEBUG: no. of words: %i\n", n);
   bool checked = check_alphabet_words(words);
   fprintf(stderr, "DEBUG: check_alphabet_words passed [%s]\n", checked ? "
    OK" : "FAIL");
   return ret;
}
```

Kódovací příklad – NATO Abeceda – 4/4

```
1  ...
2  static const char * const words[] = { "Alpha", ..., NULL };
3  ...
4  char my_toupper(char c);
6  int main(void)
7  {
8     ...
9     int c;
10    while ((c = getchar()) != EOF) {
11       c = my_toupper(c); // or toupper() from <ctype.h>
12       if (c >= 'A' && c <= 'Z') {
13          printf("%s ", words[c - 'A']); // always print space
14       }
15    }
16    ...
17 }
19 char my_toupper(char c) // or use toupper() from <ctype.h>
20 {
21    if (c >= 'a' && c <= 'z') {
22       c =  c - 'a' + 'A';
23    }
24    return c;
25 }
```

- Funkci `my_toupper()` můžeme nahradit použitím ternárního operátoru.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <stdbool.h>
6  static const char * const words[] = { "Alpha",.., NULL };
7  ...
8  int count_words_array(const char * const words[]);
9  bool check_alphabet_words(const char * const words[]);
11 int main(void)
12 {  // assert macro debug and development only, see -DNDEBUG
13    assert(count_words_array(words) == 'Z' - 'A' + 1);
14    assert(check_alphabet_words(words));
15    int c;
16    while ((c = getchar()) != EOF) {
17       c = (c >= 'a' && c <= 'z') ? c =  c - 'a' + 'A' : c;
18       if (c >= 'A' && c <= 'Z') {
19          printf("%s\n", words[c - 'A']);
20       }
21    }
22    return EXIT_SUCCESS;
23 }
```

- V rámci zpřehlednění můžeme překlad (řádky 15–21) dát do samostatné funkce `void translate(const char * const 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

NATO Abeceda

NATO Abeceda („jinak“)

Kódovací příklad – NATO Abeceda („*jinak*“) – 1/2

- Slova abecedy uložíme jako řetězec `alphabet` všech slov spojených bez mezery, do kterého budeme odkazovat na jednotlivá slova polem ukazatelů na textové řetězce (`words`).
  - Slov je `'Z' - 'A' + 1`, ale řetězec je posloupnost znaků zakončená `'\0'`.
  - První písmeno slova abecedy používáme k indexaci, např. `'C'harlie` je odkazované ukazatelem `words['C' - 'A']`. První znak slova tak můžeme v abecedě `alphabet` nahradit znakem `'\0'` získáme textové řetězce.

Bez prvního znaku!

```
1  //Ukazatel na textový literál.  Literál nemůžeme měnit!
2  //static char *alphabet = "AlphaBravoCharlie...";
3  static char alphabet[] =
4        "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
5        "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
6        "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
8  //pole ukazatelů na textové řetězce
9  static char *words['Z' - 'A' + 1] = { [0] = NULL };
11 int fill_words(char* str, char *words[]);
13 int main(void)
14 {
15    int ret = fill_words(alphabet, words);
16    if (!ret) {
17       for (char c = 'A'; c <= 'Z'; ++c) {
18          fprintf(stderr, "DEBUG: %02d. '%c' - %c%s\n",
                 c, c, c, words[c - 'A']);
19       } //                ↳ První písmeno slova abecedy.
20    }
21    return ret;
22 }
```

```
24 int fill_words(char* alphabet, char *words[])
25 {
26    int ret = EXIT_SUCCESS;
27    char *cur = alphabet; // kurzor do pole s písmeny abecedy
28    for (char c = 'A'; c <= 'Z'; ++c) {
29       assert(words[c - 'A'] == NULL); // nemá být nastaveno
30       cur = strchr(cur, c); // vyhledání řetězce začínající c
31       assert(cur); // písmeno c musí být v abecedě
32       *cur = '\0';
33       words[c - 'A'] = ++cur; // nastavení a posun kurzoru
34       assert(words[c - 'A']); //it should be set now
35    }
36    return ret; // pragmaticky vždy EXIT_SUCCESS nebo assert.
37 }
```

- V implementaci použijeme (makro) `assert()` k testování správné incializace datových struktur.

Makro slouží pro ladění, viz `man assert`.

Kódovací příklad – NATO Abeceda („*jinak*“) – 2/2

- Přidáme překlad znaků načítaných ze `stdin` a implementaci zpřehledníme.

```c
static char alphabet[] =
      "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
      "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
      "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
static char *words['Z' - 'A' + 1] = { [0] = NULL };

int fill_words(char* str, char *words[]);

int main(void)
{
   int ret = fill_words(alphabet, words);
   if (!ret) {
      for (char c = 'A'; c <= 'Z'; ++c) {
         fprintf(stderr, "DEBUG: %02d. '%c' - %c%s\n",
           c, c, c, words[c - 'A']);
      } //              ↳ První písmeno slova abecedy.
   }
   int c;
   while ((c = getchar()) != EOF) {
      c = (c >= 'a' && c <= 'z') ? c =  c - 'a' + 'A' : c;
      //   ↳ volání funkce toupper() bude přehlednější!
      if (c >= 'A' && c <= 'Z') {
         printf("%c%s ", c, words[c - 'A']);
      }
   }
   return ret;
}
```

```c
static char alphabet[] =
      "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
      "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
      "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
static char *words['Z' - 'A' + 1] = { [0] = NULL };

void fill_words(char* str, char *words[]);
void translate(char *words[]);

int main(void)
{
   fill_words(alphabet, words);
   translate(words);
   return EXIT_SUCCESS;
}

void translate(char *words[])
{
   int c;
   while ((c = getchar()) != EOF) {
      c = toupper(c); // funkce z #include<ctype.h>
      if (c >= 'A' && c <= 'Z') {
         printf("%c%s ", c, words[c - 'A']); // první znak!
      }
   }
}
```

- Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy `'A'-'Z'` a `'a'-'z'`.