Ukazatele, dynamická alokace, práce s textem

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 05

B0B36PRP – Procedurální programování

Přehled témat

- Část 1 – Ukazatele a dynamická alokace

  Modifikátor `const` a ukazatele

  Dynamická alokace paměti

  S. G. Kochan: kapitoly 8 a 11

- Část 2 – Práce s textem

  Práce s textem

- Část 3 – Zadání 5. domácího úkolu (HW05)

Část I

Část 1 – Ukazatele a dynamická alokace

Obsah

Modifikátor const a ukazatele

Dynamická alokace paměti

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).*

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 HW05 povinné a volitelné 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;
}
```

```c
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

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

Modifikátor const a ukazatele

Dynamická alokace paměti

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`. *Dynamicky alokované pole.*

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

- Operace s více hodnotami v paměťovém bloku je podobná (identická) jako práce s polem.
  - 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.

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 je hodnota proměnné stále původní adresa, 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

- Implementujte 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 70 21 0a              |I like prp!.|
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 70 21                 |I like prp!|
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 II

Část 2 – Práce s textem

Obsah

Práce s textem

Kódovací příklad – Rotace textového řetězce – 1/4

- Implementujme program, který načte ze `stdin` dva textové řetězce (dva řádky zakončené `'\n'`) a pokusí se najít rotaci (posunutí – `offset`) druhého řádku tak, aby odpovídal prvnímu řádku.
- Oba řádky (řetězce) předpokládáme, že jsou stejně dlouhé.
- Chybu dynamické alokace program indikuje návratovou hodnotou `129`, chybu vstupu hodnotou `100`, jinak vrací `EXIT_SUCCESS`.
- Délka řetězců je až do maximálního hodnoty `size_t`, posunutí pouze do `INT_MAX`.
- V případě neúspěšné dynamické alokace program ukončujeme voláním `exit(129)`;

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h> // for INT_MAX

#ifndef INIT_LEN
#define INIT_LEN 8
#endif

enum { ERROR_OK = EXIT_SUCCESS, ERROR_IN = 100, ERROR_MEM = 129 };

void* my_realloc(void *ptr, size_t size, const char *file, const int line);
```

```c
void* my_realloc(void *ptr, size_t size, const char *file, const int line)
{
   void* ret = realloc(ptr, size);
   if (!ret) {
      fprintf(stderr, "ERROR: Cannot realloc %lu bytes -- called at %s:%i\n
     ", size, file, line);
      free(ptr); // can we call free(NULL)?
      exit(ERROR_MEM);
   }
   return ret;
}
```

- Volání `realloc()` alokuje nebo přealokuje paměť.
- Funkci předáváme soubor a číslo řádku, kde funkci `my_realloc()` voláme, pro indikaci, kde došlo k chybě.

Kódovací příklad – Rotace textového řetězce – 2/4

```
14 char* read_line(void); // read a line from stdin, terminated by '\n' return as null-terminated string
16 char* shift(int offset, const char* src, size_t n, char *dst); // src and dst are strings at least n long (+1 for '\0')
18 int get_offset(const char *s1, size_t n1, const char *s2, size_t n2); // offset - max INT_MAX; strings - up to can size_t
20 int print_offset(const char *s, size_t n, int offset);
22 int main(void)
23 {
24    int ret = ERROR_OK;
25    char *l1 = read_line();
26    char *l2 = read_line();
27    size_t n1, n2;
29    if (l1 && l2 && (n1 = strlen(l1)) == (n2 = strlen(l2)) ) {
30       fprintf(stderr, "DEBUG: l1[%lu]: \"%s\"\n", n1, l1);
31       fprintf(stderr, "DEBUG: l2[%lu]: \"%s\"\n", n2, l2);
32       int offset = get_offset(l1, n1, l2, n2);
33       fprintf(stdout, "Matching offset %d\n", offset);
34       offset >= 0 && print_offset(l2, n2, offset); // call print_offset only if offset >= 0
35    } else {
36       fprintf(stderr, "ERROR: Wrong input!\n");
37       ret = ERROR_IN;
38    }
39    free(l1); // free(ptr) - If ptr is NULL no action occurs.
40    free(l2); // See man free.
41    return ret;
42 }
```

Kódovací příklad – Rotace textového řetězce – 3/4

```
59 char* read_line(void)
60 {
61    size_t capacity = INIT_LEN;
62    char *str = my_realloc(NULL, sizeof(char) * (INIT_LEN + 1),
       __FILE__, __LINE__); //+1 for '\0'
63    size_t len = 0;
64    int c;
65    while ((c = getchar()) != EOF && c != '\n') {
66       if (len == capacity) {
67          capacity *= 2;
68          str = my_realloc(str, sizeof(char) * (capacity + 1),
       __FILE__, __LINE__); //+1 for '\0'
69       }
70       str[len++] = c;
71    }
72    if (len > 0) {
73       str[len] = '\0';
74    } else {
75       free(str);
76       str = NULL;
77    }
78    return str;
79 }
```

- `read_line()` vrací `NULL` pouze pokud je načten prázdný řadek.
- Chyba alokace dynamické paměti ukončí program voláním `exit()` v naší funkci `my_realloc()`.

```
81 char* shift(int offset, const char* src, size_t n, char *dst)
82 {
83    for (size_t i = 0; i < n; ++i) { // n type is size_t !!!
84       dst[i] = src[(offset + i) % n];
85    }
86    return dst;
87 }
89 int get_offset(const char *s1, size_t n1, const char *s2, size_t n2)
90 { // we already checked that s1 && s2 && n1 == n2
91    int ret = -1;
92    int max_shift = INT_MAX < n2 ? INT_MAX : n2; // limits.h
93    char *s = my_realloc(NULL, sizeof(char) * (n2 + 1), __FILE__, __LINE__);
       // +1 for '\0'
94    for (int i = 0; i < max_shift; ++i) {
95       s = shift(i, s2, n2, s); // shift s2 to s and return s
96       if (strcmp(s1, s) == 0) { //strings matched
97          ret = i; // perfect match, exit the loop
98          break;
99       }
100   }
101   free(s); // s is dynamically allocated, releaase the memory
102   return ret;
103 }
```

- Posuneme 2. řádek (`s`) a testujeme jestli je identický s 1. řádkem.
- Funkce `strcmp()` porovnává řetězce lexikograficky, proto vrací `int`.

Kódovací příklad – Rotace textového řetězce – 4/4

- K vytištění posunutého řetězce v samostatné funkci `print_offset()` alokujeme dynamickou paměť, kterou před ukončení funkce opět uvolníme.

```
105 int print_offset(const char *s, size_t n, int offset)
106 {
107     int ret = 1;
108     char *str = my_realloc(NULL, sizeof(char) * (n + 1), __FILE__,
          __LINE__); // +1 for '\0'
109     shift(offset, s, n, str);
110     fprintf(stderr, "DEBUG: shift: \"%s\"\n", str);
111     free(str);
112     return ret;
113 }
```

```
105     char *l1 = read_line();
106     char *l2 = read_line();
107     size_t n1, n2;
109     if (l1 && l2 && (n1 = strlen(l1)) == (n2 = strlen(l2)) ) {
110         fprintf(stderr, "DEBUG: l1[%lu]: \"%s\"\n", n1, l1);
111         fprintf(stderr, "DEBUG: l2[%lu]: \"%s\"\n", n2, l2);
112         int offset = get_offset(l1, n1, l2, n2);
113         fprintf(stdout, "Matching offset %d\n", offset);
114         offset >= 0 && print_offset(l2, n2, offset);
115     } else {
116         fprintf(stderr, "ERROR: Wrong input!\n");
117         ret = ERROR_IN;
118     }
```

- Program otestujeme pro ukázkový vstup.

```
Lorem ipsum dolor sit amet.
sit amet.Lorem ipsum dolor
```

```
$ clang -g shift.c -o shift && ./shift <input.txt; echo $?
DEBUG: l1[27]: "Lorem ipsum dolor sit amet."
DEBUG: l2[27]: "sit amet.Lorem ipsum dolor "
Matching offset 9
DEBUG: shift: "Lorem ipsum dolor sit amet."
0
```

- Vyzkoušejte si chování programu v kombinaci s `valgrind` pro detekci chybného přístupu k paměti, např. chybná alokace paměti pro posunutý řetězec.

```
83     for (size_t i = 0; i < n; ++i) {
84       dst[i] = src[(offset + i) % n];
85     }
```

```
$ valgrind ./shift < input.txt
...
==80708== Invalid write of size 1
==80708==    at 0x202240: shift (shift.c:84)
==80708==    by 0x202092: get_offset (shift.c:95)
==80708==    by 0x201DF2: main (shift.c:36)
```

Část III

Část 3 – Zadání 5. domácího úkolu (HW05)

Zadání 5. domácího úkolu HW05

Téma: **Caesarova šifra**

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

- **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/b0b36prp/hw/hw05`
  - 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: 22.11.2025, 23:59:59 PST.

Shrnutí přednášky

Diskutovaná témata

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

- Příště: Struktury a uniony.

Diskutovaná témata

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

- Příště: Struktury a uniony.

Část V

Appendix

Obsah

Kódovací příklad – NATO Abeceda

Kódovací příklad – 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 PRP and programming in C.
$ clang nato-alphabet.c && ./a.out < in.txt 2>/dev/null
India Lima India Kilo Echo Papa Romeo Papa 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

```
1  // array is termined by NULL used for counting
2  static char *words[] = { "Alpha",  ..,  "Zulu", NULL };
4  // array-like variant
5  int count_words_array(char *words[])
6  {
7     int n = 0;
8     while(words[n] != NULL) {
9        fprintf(stderr, "DEBUG: \"%s\"\n", words[n]);
10       n += 1;
11    }
12    return n;
13 }
15 // pure pointer variant
16 int count_words(char **words)
17 {
18    int n = 0;
19    char **cur = words;
20    while (*cur) {
21       fprintf(stderr, "DEBUG: \"%s\"\n", *cur);
22       cur++;
23       n += 1;
24    }
25    return c;
26 }
```

```
26 bool check_alphabet_words(char *words[])
27 {
28    bool ret = true; // true is from #include <stdbool.h>
29    char c = 'A'; // char is an integer ASCII code number
30    char **cur = &words[0]; // there is always at least one item
31    while (*cur) {
32       fprintf(stderr, "DEBUG: check %s[0] for '%c'\n", *cur, c);
33       if (c != *cur[0]) { // the first letter needs to match
34          ret = false; // false is from #include <stdbool.h>
35          break;
36       } else {
37          c += 1;
38          cur += 1;
39       }
40    }
41    return ret;
42 }
```

- 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 – NATO Abeceda

Kódovací příklad – 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!

```c
//Ukazatel na textový literál.  Literál nemůžeme měnit!
//static char *alphabet = "AlphaBravoCharlie...";
static char alphabet[] =
      "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
      "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
      "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
//pole ukazatelů na textové řetězce
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.
   }
   return ret;
}
```

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

- 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'`.