

# Ukazatele, paměťové třídy, volání funkcí

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 05

**BAB36PRGA – Programování v C**

# Přehled témat

- Část 1 – Ukazatele a dynamická alokace

  - Modifikátor `const` a ukazatele

  - Dynamická alokace paměti

*S. G. Kochan: kapitoly 8 a 11*

- Část 2 – Paměťové třídy a volání funkcí

  - Výpočetní prostředky a běh programu

  - Rozsah platnosti proměnných

  - Paměťové třídy

*S. G. Kochan: kapitola 8 a 11*

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

# Část I

## Část 1 – Ukazatele a dynamická alokace

# 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 konstantní 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.“
<code>char *ptr</code>	Ne	Ne	„ <code>ptr</code> je ukazatel (*) na hodnotu <code>char</code> .“
<code>const char *ptr</code>	Ano	Ne	„ <code>ptr</code> je ukazatel na hodnotu <code>char</code> konstantní.“
<code>char const *ptr</code>	Ano	Ne	„ <code>ptr</code> je ukazatel na konstantní hodnotu <code>char</code> .“
<code>char* const ptr</code>	Ne	Ano	„ <code>ptr</code> je konstantní ukazatel na hodnotu <code>char</code> .“
<code>const char *const ptr</code>	Ano	Ano	„ <code>ptr</code> je konstantní ukazatel na hodnotu <code>char</code> konstantní.“

- *Konstantní ukazatel je proměnná, jejíž hodnotu nemohu měnit. Ukazatel odkazuje na (stejně) 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 HW4 povinné a bonusové zadání. Při vhodném návrhu programu je základní část společná, „jen“ zaměníme funkci pro porovnávání dvou řetězců s využitím Hammingovy 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 *));
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
4 void print(int n, int array[n]);
5 int compare(const void *pa, const void *pb);
7 int main(void)
8 {
9     const int n = 10;
10    int array[n];
11    for (int i = 0; i < n; ++i) {
12        array[i] = rand() % 100;
13    }
14    print(n, array);
15    qsort(array, n, sizeof(array[0]), compare);
16    print(n, array);
17    return 0;
18 }
```

```
20 void print(int n, int array[n])
21 {
22     for(int i = 0; i < n; ++i) {
23         i > 0 ? printf(", ") : 0;
24         printf("%d", array[i]);
25     }
26     n > 0 ? putchar('\n') : 0;
27 }
29 int compare(const void *pa, const void *pb)
30 {
31     const int a = *(int*)pa;
32     const int b = *(int*)pb;
33     return (a < b) - (a > b);
34 }
```

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.

## Dynamická alokace paměti

- Přídělení 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.*

## 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 newly allocated
4     // memory in the pointer ptr (i.e., the address the
5     // pointer ptr is pointed).
6
7     // call library function malloc to allocate memory
8     *ptr = malloc(size);
9
10    if (*ptr == NULL) {
11        fprintf(stderr, "Error: allocation fail");
12        exit(-1); /* exit program if allocation fail */
13    }
14    return *ptr;
15 }
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;
5
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`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #ifndef INIT_SIZE
6 #define INIT_SIZE 128
7 #endif
8
9 enum {
10     ERROR_OK = EXIT_SUCCESS,
11     ERROR_IN = 100,
12     ERROR_MEM = 101,
13 };
14
15 char* read(int *error);
16 char* enlarge_string(size_t len, size_t *capacity, char
    *str);
```

```
18 int main(int argc, char *argv[])
19 {
20     int ret = EXIT_SUCCESS;
21     char *str = read(&ret);
22     if (str) {
23         printf("Input string size %ld\n", strlen(str));
24         printf("Input string \"%s\"\n", str);
25         free(str);
26     } else {
27         fprintf(stderr, "ERROR: read return %d\n", ret);
28     }
29     return ret;
30 }
```

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 }
72
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
```

```
0000000c
```

```
|I like prg!|
```

```
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
```

```
0000000b
```

```
|I like prg!|
```

```
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 – Paměťové třídy, model výpočtu

## Paměť počítače s uloženým programem v operační paměti

- Posloupnost instrukcí je čtena z operační paměti.
- Flexibilita ve tvorbě posloupnosti.

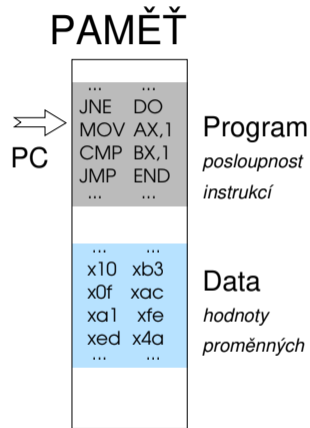
*Program lze libovolně měnit.*

- Architektura počítače se společnou pamětí pro data a program.

- von Neumannova architektura počítače

*John von Neumann (1903–1957)*

- sdílí program i data ve stejné paměti.
  - Adresa aktuálně prováděné instrukce je uložena v tzv. čítači instrukcí (Program Counter **PC**).



- Mimoto architektura se sdílenou pamětí umožňuje, aby hodnota ukazatele odkazovala nejen na data, ale také například na část paměti, kde je uložen program (funkce).

*Princip ukazatele na funkci.*



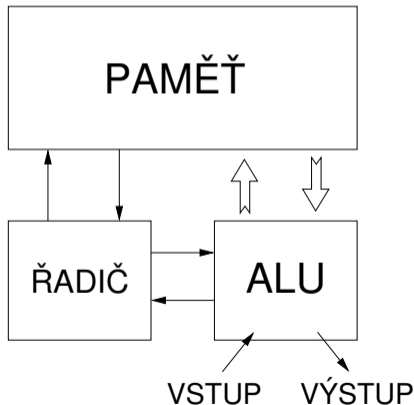
## von Neumannova architektura

*V drtivě většině případů je program posloupnost instrukcí zpracovávající jednu nebo dvě hodnoty (uložené na nějakém paměťovém místě) jako vstup a generování nějaké výstupní hodnoty, kterou ukládá někam do paměti nebo modifikuje hodnotu PC (podmíněné řízení běhu programu).*

- ALU - Aritmeticko logická jednotka (Arithmetic Logic Unit)

*Základní matematické a logické instrukce*

- PC obsahuje adresu kódu – při volání funkce tak jeho hodnotu můžeme uložit (na zásobník) a následně použít pro návrat na původní místo volání.



## Základní rozdělení paměti

- Přídělenou paměť programu můžeme kategorizovat na 5 částí.

- **Zásobník** – lokální proměnné, argumenty funkcí, návratová hodnota funkce.

*Spravováno **automaticky**.*

- **Halda** – **dynamická** paměť (`malloc()`, `free()`).

*Spravuje **programátor**.*

- **Statická** – globální nebo „lokální“ `static` proměnné.

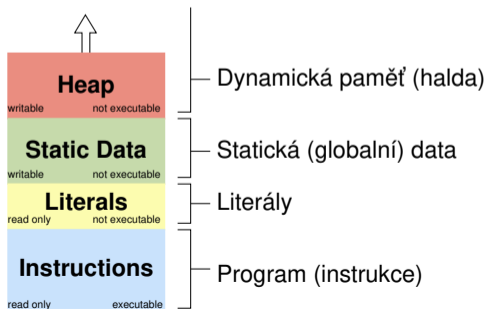
*Inicializováno **při startu**.*

- **Literály** – hodnoty zapsané ve zdrojovém kódu programu, např. textové řetězce.

*Inicializováno **při startu**.*

- **Program** – strojové instrukce.

*Inicializováno **při startu**.*



## Rozsah platnosti (scope) lokální proměnné

- Lokální proměnné mají rozsah platnosti pouze uvnitř bloku a funkce.

```
1 int a = 1; // globální proměnná
2
3 void function(void)
4 { // zde a ještě reprezentuje globální proměnnou
5     int a = 10; // lokální proměnná, zastiňuje globální a
6     if (a == 10) {
7         int a = 1; // nová lokální proměnná a; přístup
8                 // na původní lokální a je zastíněn
9         int b = 20; // lokální proměnná s platností pouze
10                // uvnitř bloku
11         a += b + 10; // proměnná a má hodnotu 31
12     } // konec bloku
13     // zde má a hodnotu 10, je to lokální proměnná z řádku 5
14
15     b = 10; // b není platnou proměnnou
16 }
```

- Globální proměnné mají rozsah platnosti „kdekoliv“ v programu.
  - Zastíněný přístup lze řešit modifikátorem `extern` (v novém bloku).

[http://www.tutorialspoint.com/cprogramming/c\\_scope\\_rules.htm](http://www.tutorialspoint.com/cprogramming/c_scope_rules.htm)

## Definice vs. deklarace proměnné – extern

- **Definice** proměnné je přidělení paměťového místa proměnné (dle typu). **Může být pouze jedna!**
- **Deklarace** “oznamuje”, že je proměnná někde definována.

```

1 // extern int global_variable = 10; /* extern
  variable with initialization is a
  definition */
2 int global_variable = 10;
3 void function(int p);          lec05/extern_var.h

```

```

1 #include <stdio.h>
3 #include "extern_var.h"
5 static int module_variable;
7 void function(int p)
8 {
9     fprintf(stdout, "function: p %d global
  variable %d\n", p, global_variable);
10 }
    lec05/extern_var.c

```

```

1 #include <stdio.h>
3 #include "extern_var.h"
5 int main(int argc, char *argv[])
6 {
7     global_variable += 1;
8     function(1);
9     function(1);
10    global_variable += 1;
11    function(1);
12    return 0;
13 }
    lec05/extern-main.c

```

- Vícenásobná definice končí chybou.

```

clang extern_var.c extern-main.c
/tmp/extern-main-619051.o:(.data+0x0): multiple
  definition of 'global_variable'
/tmp/extern_var-24da84.o:(.data+0x0): first
  defined here
clang: error: linker command failed with exit
  code 1 (use -v to see invocation)

```

## Přidělování paměti proměnným

- Přidělením paměti proměnné rozumíme určení paměťového místa pro uložení hodnoty proměnné (příslušného typu) v paměti počítače.
- Lokálním proměnným a parametrům funkce se paměť přiděluje při volání funkce.
  - Paměť zůstane přidělena jen do návratu z funkce.
  - Paměť se automaticky alokuje z rezervovaného místa – **zásobník (stack)**.  
*Při návratu funkce se přidělené paměťové místo uvolní pro další použití.*
  - Výjimku tvoří lokální proměnné s modifikátorem **static**.
    - Z hlediska platnosti rozsahu mají charakter lokálních proměnných.
    - Jejich hodnota je však zachována i po skončení funkce / bloku.
    - Jsou umístěny ve statické části paměti.
- Dynamické přidělování paměti
  - Alokace paměti se provádí funkcí **malloc()**.  
*Nebo její alternativou podle použité knihovny pro správu paměti (např. s garbage collectorem – boehm-gc).*
  - Paměť se alokuje z rezervovaného místa – **halda (heap)**.

# Zásobník

- Úseky paměti přidělované lokálním proměnným a parametrům funkce tvoří tzv. **zásobník** (stack).
- Úseky se přidávají a odebírají.
  - Vždy se odebere naposledy přidaný úsek.

*LIFO – last in, first out.*

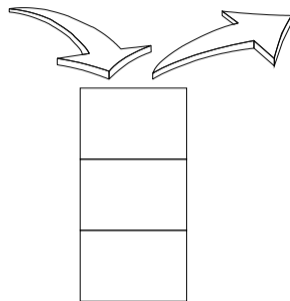
- Na zásobník se ukládá „volání funkce“.

*Na zásobník se také ukládá návratová hodnota funkce a také hodnota „program counter“ původně prováděné instrukce, před voláním funkce.*

- Ze zásobníku se alokují proměnné parametrů funkce.

*Argumenty (parametry) jsou de facto lokální proměnné.*

**Opakovaným rekurzivním voláním funkce můžeme zaplnit velikost přiděleného zásobníku a program skončí chybou.**



## Příklad rekurzivního volání funkce

- Vyzkoušejte si program pro omezenou velikost zásobníku.

```
1  #include <stdio.h>
3  void printValue(int v)
4  {
5      printf("value: %i\n", v);
6      printValue(v + 1);
7  }
9  int main(void)
10 {
11     printValue(1);
12 }
```

```
clang demo-stack_overflow.c
ulimit -s 10000; ./a.out | tail -n 3
value: 319816
value: 319817
Segmentation fault
ulimit -s 1000; ./a.out | tail -n 3
value: 31730
value: 31731
Segmentation fault
```

lec05/demo-stack\_overflow.c

## Návratová hodnota funkce a kódovací styl `return` 1/2

- Předání hodnoty volání funkce je předepsáno voláním `return`.

```
int doSomethingUseful() {  
    int ret = -1;  
    ...  
    return ret;  
}
```

- Jak často umisťovat volání `return` ve funkci?

```
int doSomething() {  
    if (  
        !cond1  
        && cond2  
        && cond3  
    ) {  
        ... do some long code ...  
    }  
    return 0;  
}
```

```
int doSomething() {  
    if (cond1) {  
        return 0;  
    }  
    if (!cond2) {  
        return 0;  
    }  
    if (!cond3) {  
        return 0;  
    }  
    ... some long code ....  
    return 0;  
}
```

<http://llvm.org/docs/CodingStandards.html>



## Návratová hodnota funkce a kódovací styl `return` 2/2

- Volání `return` na začátku funkce může být přehlednější.

*Podle hodnoty podmínky je volání funkce ukončeno.*

- Kódovací konvence může také předepisovat použití nejvýše jednoho volání `return`.

*Má výhodu v jednoznačné identifikaci místa volání, můžeme pak například jednoduše přidat další zpracování výstupní hodnoty funkce.*

- Dále není doporučováno bezprostředně používat `else` za voláním `return` (nebo jiným přerušení toku programu), např.

```
case 10:
    if (...) {
        ...
        return 1;
    } else {
        if (cond) {
            ...
            return -1;
        } else {
            break;
        }
    }
}
```

```
case 10:
    if (...) {
        ...
        return 1;
    } else {
        if (cond) {
            ...
            return -1;
        }
    }
break;
```

# Proměnné

- Proměnné představují vymezenou oblast paměti a v C je můžeme rozdělit podle způsobu alokace.
  - **Statická** alokace – provede se při definici **statické** nebo globální proměnné; paměťový prostor je alokován při startu programu a nikdy není uvolněn.
  - **Automatická** alokace – probíhá automaticky v případě lokálních proměnných (nebo argumentů funkce); paměťový prostor je alokován na **zásobníku** a paměť proměnné je automaticky uvolněna s koncem platnosti proměnné.

*Např. po ukončení bloku funkce.*
  - **Dynamická** alokace – není podporována přímo jazykem C, ale je přístupná knihovními funkcemi.

*Např. `malloc()` a `free()` z knihovny `<stdlib.h>` nebo `<malloc.h>`*

[http://gribblelab.org/CBootcamp/7\\_Memory\\_Stack\\_vs\\_Heap.html](http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html)

## Proměnné – paměťová třída

- Specifikátory paměťové třídy (Storage Class Specifiers – SCS).
  - **auto** (lokální) – Definuje proměnnou jako dočasnou (automatickou). Lze použít pro lokální proměnné definované uvnitř funkce. Jedná se o implicitní nastavení, platnost proměnné je omezena na blok. Proměnná je v **zásobníku**.
  - **register** – Doporučuje překladači umístit proměnnou do registru procesoru (rychlost přístupu). Překladač může, ale nemusí vyhovět. Jinak stejné jako **auto**.

*Zpravidla řešíme překladem s optimalizacemi.*
  - **static**
    - Uvnitř bloku `{...}` – definujeme proměnnou jako statickou, která si **ponechává hodnotu i při opuštění bloku**. Existuje po celou dobu chodu programu. Je uložena v **datové oblasti**.
    - Vně bloku – kde je implicitně proměnná uložena v **datové oblasti** (statická) omezuje její viditelnost na modul.
  - **extern** – rozšiřuje viditelnost statických proměnných z modulu na celý program. Globální proměnné s **extern** jsou definované v **datové oblasti**.

## Příklad definice proměnných

### ■ Hlavičkový soubor `vardec.h`

```
1 extern int global_variable;
```

`lec05/vardec.h`

### ■ Zdrojový soubor `vardec.c`

```
1 #include <stdio.h>
2 #include "vardec.h"
4 static int module_variable;
5 int global_variable;
7 void function(int p);
9 int main(void)
10 {
11     int local;
12     function(1);
13     function(1);
14     function(1);
15     return 0;
16 }
```

```
18 void function(int p)
19 {
20     int lv = 0; /* local variable */
21     static int lsv = 0; /* local static variable */
22     lv += 1;
23     lsv += 1;
24     printf("func: p%d, lv %d, lsv %d\n", p, lv, lsv);
25 }
```

`lec05/vardec.c`

### ■ Výstup

```
1 func: p 1, lv 1, slv 1
2 func: p 1, lv 1, slv 2
3 func: p 1, lv 1, slv 3
```

*Uvedený příklad demonstruje různé definice proměnných. V případě proměnné `global_variable` je její definice v modulu s funkcí `main()` diskutabilní. Modul `vardec.o` nebudeme linkovat s jiným program s vlastní (jinou) funkcí `main()`.*

## Definice proměnných a operátor přiřazení

- Proměnné definujeme uvedením typu a jména proměnné.
  - Jména proměnných volíme malá písmena.
  - Víceslovná jména zapisujeme s podtržítkem `_` nebo volíme tzv. *camelCase*.

<https://en.wikipedia.org/wiki/CamelCase>

- Proměnné definujeme na samostatném řádku.

```
1 int n;  
2 int number_of_items;
```

- Příkaz přiřazení se skládá z operátoru přiřazení `=` a ;
  - Levá strana přiřazení musí být **l-value – location-value, left-value** – musí reprezentovat paměťové místo pro uložení výsledku.
  - Přiřazení je výraz a můžeme jej tak použít všude, kde je dovolen výraz příslušného typu.

```
1 /* int c, i, j; */  
2 i = j = 10;  
3 if ((c = 5) == 5) {  
4     fprintf(stdout, "c is 5 \n");  
5 } else {  
6     fprintf(stdout, "c is not 5\n");  
7 }
```

lec05/assign.c

## 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).
    - Znaky pro kvantifikátory: `?`, `*`, `+`.
- **Termín odevzdání:** **13.04.2024, 23:59:59 PDT.**

# Shrnutí přednášky



## Diskutovaná témata

- Ukazatele a modifikátor `const`
- Dynamická alokace paměti
- Ukazatel na funkce
- Paměťové třídy
- Volání funkcí
  
- Příště: Struktury a union, přesnost výpočtu a vnitřní reprezentace číselných typů.

# Část V

## Appendix

## 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.
  - 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.
- 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", "
3     Foxtrot", "Golf", "Hotel", "India", "Juliett", "
4     Kilo", "Lima", "Mike", "November", "Oscar", "Papa",
5     "Quebec", "Romeo", "Sierra", "Tango", "Uniform", "
6     Victor", "Whiskey", "X-ray", "Yankee", "Zulu", NULL
7 }; // it is an array of pointers to text literls
8 int count_words_array(char *words[]); // Using array
9 int count_words(char **words); // Pointer to pointers
10 bool check_alphabet_words(char *words[]);
```

## Kódovací příklad – NATO Abeceda – 2/4

```

1 // array is terminated by NULL used for counting
2 static char *words[] = { "Alpha", .., "Zulu", NULL };
3
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 }
14
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 n;
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`.

```

1 static const char * const words[] = { "Alpha", ..., NULL };
2
3 int count_words_array(const char * const words[])
4 {
5     int n = 0;
6     while(words[n] != NULL) {
7         n += 1;
8     }
9     return n;
10 }
11
12 int count_words(const char * const * const words)
13 {
14     int n = 0;
15     // cur je ukazatel na data typu konstantní
16     // ukazatel na konstantní textový řetězec
17     // (na konstantní ukazatel na konstantní hodnoty char).
18     const char * const * cur = words; // cur chceme měnit
19     while (*cur) {
20         cur++; // cur není konstantní ukazatel
21         n += 1;
22     }
23     return n;
24 }

```

```

26 #include <stdio.h>
27 #include <stdbool.h>
28
29 static const char * const words[] = { "Alpha", ..., NULL };
30
31 int count_words_array(const char * const words[]);
32 int count_words(const char * const * const words);
33
34 bool check_alphabet_words(const char * const words[]);
35
36 int main(void)
37 {
38     int ret = EXIT_SUCCESS;
39     fprintf(stderr, "DEBUG: size %lu\n", sizeof(words));
40
41     int n = count_words_array(words);
42     fprintf(stderr, "DEBUG: no. of words: %i\n", n);
43
44     n = count_words(&words[0]);
45     fprintf(stderr, "DEBUG: no. of words: %i\n", n);
46
47     bool checked = check_alphabet_words(words);
48     fprintf(stderr, "DEBUG: check_alphabet_words passed [%s]\n",
49             checked ? "OK" : "FAIL");
50     return ret;
51 }

```

## 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 - '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[])`.

## 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`).
  - Slovo 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";
7
8 //pole ukazatelů na textové řetězce
9 static char *words['Z' - 'A' + 1] = { [0] = NULL };
10
11 int fill_words(char* str, char *words[]);
12
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",
19                 c, c, c, words[c - 'A']);
20         } //      ↳ První písmeno slova abecedy.
21     }
22     return ret;
23 }

```

```

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 inicializace datových struktur.

*Makro slouží pro ladění, viz man assert.*

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

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

```

1 static char alphabet[] =
2     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
3     "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
4     "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
5 static char *words['Z' - 'A' + 1] = { [0] = NULL };
6
7 int fill_words(char* str, char *words[]);
8
9 int main(void)
10 {
11     int ret = fill_words(alphabet, words);
12     if (!ret) {
13         for (char c = 'A'; c <= 'Z'; ++c) {
14             fprintf(stderr, "DEBUG: %02d. '%c' - '%c%s\n",
15                 c, c, c, words[c - 'A']);
16         } //           ↴ První písmeno slova abecedy.
17     }
18     int c;
19     while ((c = getchar()) != EOF) {
20         c = (c >= 'a' && c <= 'z') ? c = c - 'a' + 'A' : c;
21         //     ↴ volání funkce toupper() bude přehlednější!
22         if (c >= 'A' && c <= 'Z') {
23             printf("%c%s ", c, words[c - 'A']);
24         }
25     }
26     return ret;
27 }

```

```

1 static char alphabet[] =
2     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
3     "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
4     "SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
5 static char *words['Z' - 'A' + 1] = { [0] = NULL };
6
7 void fill_words(char* str, char *words[]);
8 void translate(char *words[]);
9
10 int main(void)
11 {
12     fill_words(alphabet, words);
13     translate(words);
14     return EXIT_SUCCESS;
15 }
16
17 void translate(char *words[])
18 {
19     int c;
20     while ((c = getchar()) != EOF) {
21         c = toupper(c); // funkce z #include<ctype.h>
22         if (c >= 'A' && c <= 'Z') {
23             printf("%c%s ", c, words[c - 'A']); // první znak!
24         }
25     }
26 }
27 }

```

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



## Kódovací příklad – Rotace textového řetězce – 1/4

- Implementujeme 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 ukončíme voláním `exit(129)`;

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <limits.h> // for INT_MAX
5
6 #ifndef INIT_LEN
7 #define INIT_LEN 8
8 #endif
9
10 enum { ERROR_OK = EXIT_SUCCESS, ERROR_IN = 100, ERROR_MEM = 129 };
11
12 void* my_realloc(void *ptr, size_t size,
13                const char *file, const int line);

```

```

48 void* my_realloc(void *ptr, size_t size,
49                const char *file, const int line)
50 {
51     void* ret = realloc(ptr, size);
52     if (!ret) {
53         fprintf(stderr, "ERROR: Cannot realloc %lu bytes -- called
54                 at %s:%i\n", size, file, line);
55         free(ptr);
56         exit(ERROR_MEM);
57     }
58     return ret;
59 }

```

- Volání `realloc()` alokuje nebo převalokuje 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)
63         , __FILE__, __LINE__); //+1 for '\0'
64     size_t len = 0;
65     int c;
66     while ((c = getchar()) != EOF && c != '\n') {
67         if (len == capacity) {
68             capacity *= 2;
69             str = my_realloc(str, sizeof(char) * (capacity + 1),
70                 __FILE__, __LINE__); //+1 for '\0'
71         }
72         str[len++] = c;
73     }
74     if (len > 0) {
75         str[len] = '\0';
76     } else {
77         free(str);
78         str = NULL;
79     }
80     return str;
81 }

```

- `read_line()` vrací `NULL` pouze pokud je načten prázdný řádek.
- 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 }
88
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__,
94         __LINE__); // +1 for '\0'
95     for (int i = 0; i < max_shift; ++i) {
96         s = shift(i, s2, n2, s); // shift s2 to s and return s
97         if (strcmp(s1, s) == 0) { //strings matched
98             ret = i; // perfect match, exit the loop
99             break;
100         }
101     }
102     free(s); // s is dynamically allocated, release the memory
103     return ret;
104 }

```

- 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),
109                          __FILE__, __LINE__); // +1 for '\0'
110     shift(offset, s, n, str);
111     fprintf(stderr, "DEBUG: shift: \"%s\"\n", str);
112     free(str);
113     return ret;

```

```

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)

```