

# 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)

Modifikátor `const` a ukazatele

Dynamická alokace paměti

## 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.

*Přípomínka*

## 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 – 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

## 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

## Konstantní ukazatel (na konstantní hodnotu)

Příklad	Konstantní hodnota	Konstantní ukazatel	Popis „Čtu zprava doleva.“
<code>char *ptr</code>	Ne	Ne	„ <code>ptr</code> je ukazatel ( <code>*</code> ) 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).

Modifikátor const a ukazatele Dynamická alokace paměti

## 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. typu 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);`

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 11 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

## 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.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 12 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

## Příklad – Ukazatel na funkci 2/2

- V případě funkce vracějící ukazatel postupujeme identicky.
 

```
double* compute(int v);
double* (*function_p)(int v);
function_p = compute;
```

----- substitute a function name
- Příklad použití ukazatele na funkci – `lec05/pointer_fnc.c`
- Ukazatele na funkce umožňují realizovat dynamickou vazbu volání funkce identifikované za běhu programu.
 

*V objektové orientovaném programování je dynamická vazba klíčem k realizaci polymorfiamu.*

*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.*

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 13 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

## 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>
3 void print(int n, int array[n]);
4 int compare(const void *pa, const void *pb);
5 int main(void)
6 {
7     const int n = 10;
8     int array[n];
9     for (int i = 0; i < n; ++i) {
10        array[i] = rand() % 100;
11    }
12    print(n, array);
13    qsort(array, n, sizeof(array[0]), compare);
14    print(n, array);
15    return 0;
16 }
```

```
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 }
28 {
29     const int a = *(int*)pa;
30     const int b = *(int*)pb;
31     return (a < b) - (a > b);
32 }
```

lec05/demo-pointer\_fnc.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 14 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

## 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.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 15 / 48

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*` – přetytování 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.*

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 17 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

## 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     // call library function malloc to allocate memory
7     *ptr = malloc(size);
8     if (*ptr == NULL) {
9         fprintf(stderr, "Error: allocation fail");
10        exit(-1); /* exit program if allocation fail */
11    }
12    return *ptr;
13 }
```

lec05/malloc\_demo.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 18 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

## 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

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 19 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

## Příklad alokace dynamické paměti 3/3

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

lec05/malloc\_demo.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 20 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

### Příklad - Načítání textového řetězce 1/3

- Implementujete načtení libovolné dlouhé řádky ze stdin.
- Řádek je zakončen znakem nového řádky '\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 #ifndef INIT_SIZE
5 #define INIT_SIZE 128
6 #endif
7 enum {
8     ERROR_OK = EXIT_SUCCESS,
9     ERROR_IN = 100,
10    ERROR_MEM = 101,
11 };
12 char* read(int *error);
13 char* enlarge_string(size_t len, size_t *capacity, char *str);
14
15 int main(int argc, char *argv[])
16 {
17     int ret = EXIT_SUCCESS;
18     char *str = read(&ret);
19     if (str) {
20         printf("Input string size %ld\n", strlen(str));
21         printf("Input string \"%s\"\n", str);
22         free(str);
23     } else {
24         fprintf(stderr, "ERROR: read return %d\n", ret);
25     }
26     return ret;
27 }
28
29 int main(int argc, char *argv[])
30 {
31     int ret = EXIT_SUCCESS;
32     char *str = read(&ret);
33     if (str) {
34         printf("Input string size %ld\n", strlen(str));
35         printf("Input string \"%s\"\n", str);
36         free(str);
37     } else {
38         fprintf(stderr, "ERROR: read return %d\n", ret);
39     }
40     return ret;
41 }
42
43 char* read(int *error)
44 {
45     size_t capacity = INIT_SIZE;
46     size_t l = 0; // no. of read chars
47     char* str = malloc(capacity + 1);
48     int r = '\0';
49     while (
50         str
51         && *error == ERROR_OK
52         && (r = getchar()) != EOF
53         && r != '\n'
54     ) {
55         if (l == capacity) { // enlarge if need
56             // new address of str can be set
57             str = enlarge_string(l, &capacity, str);
58         }
59         //Is it correct? Can str be NULL?
60         str[l++] = r;
61     } // end while
62     str = handle_str(r, l, str, error);
63     return str;
64 }
65
66 char* handle_str(char r, size_t l, char *str, int *error)
67 {
68     if (r != '\n') { // end-of-line has not been read
69         *error = ERROR_IN; // report input error
70         free(str);
71         str = NULL;
72     } else {
73         str[l] = '\0'; // null terminating string
74     }
75     if (*error == ERROR_OK) { // str is NULL
76         *error = ERROR_MEM; // but error needs to be set
77     }
78     return str;
79 }
80
81 char* enlarge_string(size_t len, size_t *capacity, char *str)
82 {
83     char *t = realloc(str, *capacity * 2 + 1);
84     if (!t) {
85         *error = ERROR_MEM;
86         return NULL; // indicate error
87     }
88     str = t;
89     *capacity *= 2;
90     return str;
91 }

```

lec05/read.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 21 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

### 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 }
56
57 char* handle_str(char r, size_t l, char *str, int *error)
58 {
59     if (r != '\n') { // end-of-line has not been read
60         *error = ERROR_IN; // report input error
61         free(str);
62         str = NULL;
63     } else {
64         str[l] = '\0'; // null terminating string
65     }
66     if (*error == ERROR_OK) { // str is NULL
67         *error = ERROR_MEM; // but error needs to be set
68     }
69     return str;
70 }
71
72 char* enlarge_string(size_t len, size_t *capacity, char *str)
73 {
74     char *t = realloc(str, *capacity * 2 + 1);
75     if (!t) {
76         *error = ERROR_MEM;
77         return NULL; // indicate error
78     }
79     str = t;
80     *capacity *= 2;
81     return str;
82 }

```

lec05/read.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 22 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

### 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

```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 23 / 48

Modifikátor const a ukazatele Dynamická alokace paměti

### 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

```

- 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/create\_rand\_string.sh

lec05/read.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 24 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## Část II

### Část 2 – Paměťové třídy, model výpočtu

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 25 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

### Paměť počítače s uloženým programem v operační paměti

- Posloupnost instrukcí je čtena z operační paměti.
- Flexibilita ve tvorbě posloupnosti.
- Architektura počítače se společnou pamětí pro data a program.

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

Program lze libovolně měnit.

John von Neumann (1903–1957)

PAMĚŤ

...	...
JNE	DO
MOV	AX,1
CMP	BX,1
JMP	END
...	...
x10	xb3
x0f	xac
x01	xfe
xed	xca
...	...

PC ⇒

Program  
posloupnost  
instrukcí

Data  
hodnoty  
proměnných

- 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.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 27 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

### 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 (podmíněně 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.

PAMĚŤ

↑ ↓

ŘADIČ

↔

ALU

↔

VSTUP

↔

VÝSTUP

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 28 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

### Základní rozdělení paměti

- Přidělenou paměť programu můžeme kategorizovat na 5 částí.
- Zásobník** – lokální proměnné, argumenty funkcí, návratová hodnota funkce.
- Halda** – dynamická paměť (malloc(), free()).
- Statická** – globální nebo „lokální“ statické proměnné.
- Literály** – hodnoty zapsané ve zdrojovém kódu programu, např. textové řetězce.
- Program** – strojové instrukce.

Spravováno automaticky.

Spravuje programátor.

Inicializováno při startu.

Inicializováno při startu.

Inicializováno při startu.

Args & Env

Stack

Halda

Heap

Static Data

Literals

Instructions

Argumenty příkazové řádky a proměnné prostředí

Zásobník

Dynamická paměť (halda)

Statická (globální) data

Literály

Program (instrukce)

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 29 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

### Rozsah platnosti (scope) lokální proměnné

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

```

1 int a = 1; // globální proměnná
2 void function(void)
3 {
4     // zde a ještě reprezentuje globální proměnnou
5     int a = 10; // lokální proměnná, zastihuje 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    b = 10; // b není platnou proměnnou
15 }

```

- 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)

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 31 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## 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
4
5 #include <stdio.h>
6 #include "extern_var.h"
7 static int module_variable;
8 void function(int p)
9 {
10     fprintf(stdout, "function: p %d global
    variable %d\n", p, global_variable);
11 }
12                                     lec05/extern_var.c
    
```

```

1 #include <stdio.h>
2 #include "extern_var.h"
3 int main(int argc, char *argv[])
4 {
5     global_variable ++ 1;
6     function(1);
7     global_variable ++ 1;
8     function(1);
9     return 0;
10                                     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-24a8d1.o:(.data+0x0): first
defined here
clang: error: linker command failed with exit
code 1 (use -v to see invocation)
    
```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 32 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## 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)**.

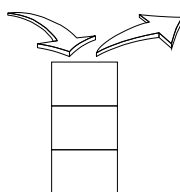
Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 33 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## Zásobník

- Úseky paměti přidělované lokálním proměnným a parametrům funkce tvoří tzv. **zásobník (stack)**.
- Úseky se přidávají a odebírají.
  - Vždy se odebere naposledy přidávaný úsek.
- Na zásobník se ukládá „volání funkce“.
  - Argumenty (parametry) jsou de facto lokální proměnné.
- Ze zásobníku se alokují proměnné parametrů funkce.

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



*LIFO – last in, first out.*

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

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 34 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## Příklad rekurzivního volání funkce

- Vyzkoušejte si program pro omezenou velikost zásobníku.

```

1 #include <stdio.h>
2 void printValue(int v)
3 {
4     printf("value: %i\n", v);
5     printValue(v + 1);
6 }
7 int main(void)
8 {
9     printValue(1);
10 }
11                                     lec05/demo-stack_overflow.c
    
```

```

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
    
```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 35 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## 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 umísťovat 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>

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 36 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## 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 přerušením toku programu), např.
 

```

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

```

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

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 37 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## 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 alokovan při startu programu a nikdy není uvolněn.
  - Automatická** alokace – probíhá automaticky v případě lokálních proměnných (nebo argumentů funkce); paměťový prostor je alokovan 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 knihovnými 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)

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 39 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## 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 stejně 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**.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 40 / 48

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

## 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"
3 static int module_variable;
4 int global_variable;
5 void function(int p);
6 int main(void)
7 {
8     int local;
9     function(1);
10    function(1);
11    return 0;
12 }
    
```
- 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.c` nebudeme linkovat s jiným program s vlastní (jinou) funkcí `main()`.*

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 41 / 48

### 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;

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

Diskutovaná témata

## Shrnutí přednášky

### Kódovací příklad – NATO Abeceda – 1/4

- Implementujeme 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, Foxtro, 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     "Foxtro", "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 literals
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[]);
```

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

Diskutovaná témata

- Ukazatele a modifikátor `const`
- Dynamická alokace paměti
- Ukazatel na funkce
- Paměťové třídy
- Volání funkcí

■ Přístě: Struktury a union, přesnost výpočtu a vnitřní reprezentace číselných typů.

### Kódovací příklad – NATO Abeceda – 2/4

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

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

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

**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 rádcích 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.

Kódovací příklad – NATO Abeceda Kódovací příklad – NATO Abeceda („jinak“) Kódovací příklad – Rotace textového řetězce

## Část V Appendix

### Kódovací příklad – NATO Abeceda – 3/4

- Můžeme použít `const`.
 

```
1 static const char * const words[] = { "Alpha", ... , NULL };
2 int count_words_array(const char * const words[])
3 {
4     int n = 0;
5     while(words[n] != NULL) {
6         n++;
7     }
8     return n;
9 }
10 int main(void)
11 {
12     int ret = EXIT_SUCCESS;
13     int n = count_words_array(words);
14     fprintf(stderr, "DEBUG: size %i\n", sizeof(words));
15     fprintf(stderr, "DEBUG: no. of words: %i\n", n);
16     n = count_words(&words[0]);
17     fprintf(stderr, "DEBUG: no. of words: %i\n", n);
18     bool checked = check_alphabet_words(words);
19     fprintf(stderr, "DEBUG: check_alphabet_words passed [%s]\n",
20         checked ? "OK" : "FAIL");
21     return ret;
22 }
```

## Kódovací příklad – NATO Abeceda – 4/4

```
1 ...
2 static const char * const words[] = { "Alpha", ..., NULL };
3 ...
4 char my_toupper(char c);
5 int main(void)
6 {
7     ...
8     int c;
9     while ((c = getchar()) != EOF) {
10        c = my_toupper(c); // or toupper() from <ctype.h>
11        if (c >= 'A' && c <= 'Z') {
12            printf("%s ", words[c - 'A']); // always print space
13        }
14        ...
15    }
16    char my_toupper(char c) // or use toupper() from <ctype.h>
17    {
18        if (c >= 'a' && c <= 'z') {
19            c = c - 'a' + 'A';
20        }
21        return c;
22    }
23 }
24 #include <stdio.h>
25 #include <stdlib.h>
26 #include <assert.h>
27 #include <stdbool.h>
28 static const char * const words[] = { "Alpha", ..., NULL };
29 ...
30 int count_words_array(const char * const words[]);
31 bool check_alphabet_words(const char * const words[]);
32 int main(void)
33 { // assert macro debug and development only, see -ENDEBUG
34     assert(count_words_array(words) == 'Z' - 'A' + 1);
35     assert(check_alphabet_words(words));
36     int c;
37     while ((c = getchar()) != EOF) {
38         c = (c >= 'a' && c <= 'z') ? c - 'a' + 'A' : c;
39         if (c >= 'A' && c <= 'Z') {
40             printf("%s\n", words[c - 'A']);
41         }
42     }
43     return EXIT_SUCCESS;
44 }
```

- Funkci `my_toupper()` můžeme nahradit použitím ternárního operátoru.
- V rámci zpráhlednění můžeme překlád (řádky 15–21) dát do samostatné funkce `void translate(const char * const words[])`.

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

- Slova abecedy uložíme jako jednotlivá slova polem ukazatelů na textové řetězce (`words`).
- Slovo je 'Z' - 'A' + 1, ale řetězec je posoupněn znaků zakončená '\0'.
- První písmeno slova abecedy používáme k indexaci, např. 'C' harle je odkazované ukazatelem 'l' 'C' - 'A'. První znak slova tak můžeme v abecedě `alphabet` nahradit znakem '\0' ziskáme 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     "JuliettKiloLimaMikeNovemberOscarPapayaQuebecRomeo"
6     "SierraTangoUniformVictorWhiskey-rayYankeeZulu";
7 //pole ukazatelů na textové řetězce
8 static char *words["Z" - 'A' + 1] = { 0 } = NULL };
9 int fill_words(char* str, char *words[]);
10 int main(void)
11 {
12     int ret = fill_words(alphabet, words);
13     if ((ret) <
14         for (char c = 'A'; c <= 'Z'; ++c) {
15             fprintf(stderr, "DEBUG: %02d. '%c' - %c\n",
16                 c, c, words[c - 'A']);
17         }
18     } //
19     }
20     return ret;
21 }
22 }
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         return ret; // pragmatically vždy EXIT_SUCCESS nebo assert.
36     }
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
```

- 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 zpráhledníme.

```
1 static char alphabet[] =
2     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
3     "JuliettKiloLimaMikeNovemberOscarPapayaQuebecRomeo"
4     "SierraTangoUniformVictorWhiskey-rayYankeeZulu";
5 static char *words["Z" - 'A' + 1] = { 0 } = NULL };
6 int fill_words(char* str, char *words[]);
7 int main(void)
8 {
9     int ret = fill_words(alphabet, words);
10    if ((ret) <
11        for (char c = 'A'; c <= 'Z'; ++c) {
12            fprintf(stderr, "DEBUG: %02d. '%c' - %c\n",
13                c, c, words[c - 'A']);
14        }
15    } //
16    }
17    int c;
18    while ((c = getchar()) != EOF) {
19        c = (c >= 'a' && c <= 'z') ? c - 'a' + 'A' : c;
20        // i volání funkce toupper() bude zpřáhlednější!
21        if (c >= 'A' && c <= 'Z') {
22            printf("%c%c ", c, words[c - 'A']);
23        }
24    }
25    return ret;
26 }
27 }
28 }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
```

- 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 ukončujeme voláním `exit(129)`;

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <limits.h> // for INT_MAX
5 #define INIT_LEN 8
6 #define INIT_LEN 8
7 #endif
8 int main(void)
9 {
10     enum { ERROR_OK = EXIT_SUCCESS, ERROR_IN = 100, ERROR_MEM = 129 };
11     void* my_realloc(void *ptr, size_t size, const char *file, const int line);
12     void* my_realloc(void *ptr, size_t size, const char *file, const int line)
13     {
14         void* ret = realloc(ptr, size);
15         if ((ret) <
16             fprintf(stderr, "ERROR: Cannot realloc %lu bytes -- called at %s:%d\n", size, file, line);
17             free(ptr);
18             exit(ERROR_MEM);
19         }
20         return ret;
21     }
22     free(l1); // free(ptr) - If ptr is NULL no action occurs.
23     free(l2); // See man free.
24     return ret;
25 }
```

- Volání `realloc()` alokuje nebo přelokuje 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

```
1 char* read_line(void); // read a line from stdin, terminated by '\n' return as null-terminated string
2 char* shift(int offset, const char* src, size_t n, char *dst); // src and dst are strings at least n long (+1 for '\0')
3 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
4 int print_offset(const char *s, size_t n, int offset);
5 int main(void)
6 {
7     int ret = ERROR_OK;
8     char *l1 = read_line();
9     char *l2 = read_line();
10    size_t n1, n2;
11    if ((n1 && n2 && (n1 == strlen(l1)) == (n2 == strlen(l2))) &&
12        fprintf(stderr, "DEBUG: l1[%lu]: %s\n", n1, l1);
13        fprintf(stderr, "DEBUG: l2[%lu]: %s\n", n2, l2);
14        int offset = get_offset(l1, n1, l2, n2);
15        fprintf(stdout, "Matching offset %d\n", offset);
16        offset >= 0 && print_offset(l2, n2, offset); // call print_offset only if offset >= 0
17    } else {
18        fprintf(stderr, "ERROR: Wrong input!\n");
19        ret = ERROR_IN;
20    }
21    free(l1); // free(ptr) - If ptr is NULL no action occurs.
22    free(l2); // See man free.
23    return ret;
24 }
```

- Funkci předáváme soubor a číslo řádku, kde funkci `my_realloc()` voláme, pro indikaci, kde došlo k chybě.

## Kódovací příklad – Rotace textového řetězce – 3/4

```
1 char* read_line(void)
2 {
3     size_t capacity = INIT_LEN;
4     char *str = my_realloc(NULL, sizeof(char) * (INIT_LEN + 1),
5         __FILE__, __LINE__); //+1 for '\0'
6     size_t len = 0;
7     int c;
8     while ((c = getchar()) != EOF && c != '\n') {
9         if ((len == capacity) &&
10            str = my_realloc(str, sizeof(char) * (capacity + 1),
11                __FILE__, __LINE__); //+1 for '\0'
12            str[len++] = c;
13        }
14        if ((len > 0) &&
15            str[len] = '\0';
16        } else {
17            free(str);
18        }
19        return str;
20    }
21 }
22 }
23 }
24 }
25 }
26 }
27 }
28 }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
```

- `read_line()` vrací `NULL` pouze pokud je načten prázdný řádek.
- Chyba alokace dynamické paměti ukončí program voláním `exit()` v naší funkci `my_realloc()`.
- Posuneme 2. řádek (s) a testujeme jestli je identický s 1. řádkem.
- Funkce `strncmp()` porovná 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 ukončíme voláním `free()`.

```
1 int print_offset(const char *s, size_t n, int offset)
2 {
3     int ret = 1;
4     char *str = my_realloc(NULL, sizeof(char) * (n + 1),
5         __FILE__, __LINE__); // +1 for '\0'
6     shift(offset, s, n, str);
7     fprintf(stderr, "DEBUG: shift: %s\n", str);
8     free(str);
9     return ret;
10 }
```

```
1 char *l1 = read_line();
2 char *l2 = read_line();
3 size_t n1, n2;
4 if ((n1 && n2 && (n1 == strlen(l1)) == (n2 == strlen(l2))) &&
5     fprintf(stderr, "DEBUG: l1[%lu]: %s\n", n1, l1);
6     fprintf(stderr, "DEBUG: l2[%lu]: %s\n", n2, l2);
7     int offset = get_offset(l1, n1, l2, n2);
8     fprintf(stdout, "Matching offset %d\n", offset);
9     offset >= 0 && print_offset(l2, n2, offset);
10 } else {
11     fprintf(stderr, "ERROR: Wrong input!\n");
12     ret = ERROR_IN;
13 }
```

```
1 Lorem ipsum dolor sit amet.
2 sit amet.Lorem ipsum dolor
3
4 $ clang -g shift.c -o shift && ./shift <input.txt; echo $?
5 DEBUG: l1[07]: "Lorem ipsum dolor sit amet."
6 DEBUG: l2[27]: "sit amet.Lorem ipsum dolor "
7 Matching offset 9
8 DEBUG: shift: "Lorem ipsum dolor sit amet."
9
```

```
1 $ valgrind ./shift < input.txt
2 ...
3 ==80708== Invalid write of size 1
4 ==80708==   at 0x202240: shift (shift.c:84)
5 ==80708==   by 0x202092: get_offset (shift.c:95)
6 ==80708==   by 0x201DF2: main (shift.c:36)
```