

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

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

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 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	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 (jejíž 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);
//----- 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 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 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 #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     else 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     else {
89         *capacity *= 2;
90     }
91     return t;
92 }
93
94 char* handle_str(char r, size_t l, char *str, int *error)
95 {
96     if (r != '\n') { // end-of-line has not been read
97         *error = ERROR_IN; // report input error
98         free(str);
99         str = NULL;
100    } else {
101        str[l] = '\0'; // null terminating string
102    }
103    else if (*error == ERROR_OK) { // str is NULL
104        *error = ERROR_MEM; // but error needs to be set
105    }
106    return str;
107 }
108
109 char* enlarge_string(size_t len, size_t *capacity, char *str)
110 {
111     char *t = realloc(str, *capacity * 2 + 1);
112     if (!t) {
113         *error = ERROR_MEM;
114         return NULL; // indicate error
115     }
116     else {
117         *capacity *= 2;
118     }
119     return t;
120 }
121
122 char* handle_str(char r, size_t l, char *str, int *error)
123 {
124     if (r != '\n') { // end-of-line has not been read
125         *error = ERROR_IN; // report input error
126         free(str);
127         str = NULL;
128     } else {
129         str[l] = '\0'; // null terminating string
130     }
131     else if (*error == ERROR_OK) { // str is NULL
132         *error = ERROR_MEM; // but error needs to be set
133     }
134     return str;
135 }
136
137 char* enlarge_string(size_t len, size_t *capacity, char *str)
138 {
139     char *t = realloc(str, *capacity * 2 + 1);
140     if (!t) {
141         *error = ERROR_MEM;
142         return NULL; // indicate error
143     }
144     else {
145         *capacity *= 2;
146     }
147     return t;
148 }
149
150 char* handle_str(char r, size_t l, char *str, int *error)
151 {
152     if (r != '\n') { // end-of-line has not been read
153         *error = ERROR_IN; // report input error
154         free(str);
155         str = NULL;
156     } else {
157         str[l] = '\0'; // null terminating string
158     }
159     else if (*error == ERROR_OK) { // str is NULL
160         *error = ERROR_MEM; // but error needs to be set
161     }
162     return str;
163 }
164
165 char* enlarge_string(size_t len, size_t *capacity, char *str)
166 {
167     char *t = realloc(str, *capacity * 2 + 1);
168     if (!t) {
169         *error = ERROR_MEM;
170         return NULL; // indicate error
171     }
172     else {
173         *capacity *= 2;
174     }
175     return t;
176 }
177
178 char* handle_str(char r, size_t l, char *str, int *error)
179 {
180     if (r != '\n') { // end-of-line has not been read
181         *error = ERROR_IN; // report input error
182         free(str);
183         str = NULL;
184     } else {
185         str[l] = '\0'; // null terminating string
186     }
187     else if (*error == ERROR_OK) { // str is NULL
188         *error = ERROR_MEM; // but error needs to be set
189     }
190     return str;
191 }
192
193 char* enlarge_string(size_t len, size_t *capacity, char *str)
194 {
195     char *t = realloc(str, *capacity * 2 + 1);
196     if (!t) {
197         *error = ERROR_MEM;
198         return NULL; // indicate error
199     }
200     else {
201         *capacity *= 2;
202     }
203     return t;
204 }
205
206 char* handle_str(char r, size_t l, char *str, int *error)
207 {
208     if (r != '\n') { // end-of-line has not been read
209         *error = ERROR_IN; // report input error
210         free(str);
211         str = NULL;
212     } else {
213         str[l] = '\0'; // null terminating string
214     }
215     else if (*error == ERROR_OK) { // str is NULL
216         *error = ERROR_MEM; // but error needs to be set
217     }
218     return str;
219 }
220
221 char* enlarge_string(size_t len, size_t *capacity, char *str)
222 {
223     char *t = realloc(str, *capacity * 2 + 1);
224     if (!t) {
225         *error = ERROR_MEM;
226         return NULL; // indicate error
227     }
228     else {
229         *capacity *= 2;
230     }
231     return t;
232 }
233
234 char* handle_str(char r, size_t l, char *str, int *error)
235 {
236     if (r != '\n') { // end-of-line has not been read
237         *error = ERROR_IN; // report input error
238         free(str);
239         str = NULL;
240     } else {
241         str[l] = '\0'; // null terminating string
242     }
243     else if (*error == ERROR_OK) { // str is NULL
244         *error = ERROR_MEM; // but error needs to be set
245     }
246     return str;
247 }
248
249 char* enlarge_string(size_t len, size_t *capacity, char *str)
250 {
251     char *t = realloc(str, *capacity * 2 + 1);
252     if (!t) {
253         *error = ERROR_MEM;
254         return NULL; // indicate error
255     }
256     else {
257         *capacity *= 2;
258     }
259     return t;
260 }
261
262 char* handle_str(char r, size_t l, char *str, int *error)
263 {
264     if (r != '\n') { // end-of-line has not been read
265         *error = ERROR_IN; // report input error
266         free(str);
267         str = NULL;
268     } else {
269         str[l] = '\0'; // null terminating string
270     }
271     else if (*error == ERROR_OK) { // str is NULL
272         *error = ERROR_MEM; // but error needs to be set
273     }
274     return str;
275 }
276
277 char* enlarge_string(size_t len, size_t *capacity, char *str)
278 {
279     char *t = realloc(str, *capacity * 2 + 1);
280     if (!t) {
281         *error = ERROR_MEM;
282         return NULL; // indicate error
283     }
284     else {
285         *capacity *= 2;
286     }
287     return t;
288 }
289
290 char* handle_str(char r, size_t l, char *str, int *error)
291 {
292     if (r != '\n') { // end-of-line has not been read
293         *error = ERROR_IN; // report input error
294         free(str);
295         str = NULL;
296     } else {
297         str[l] = '\0'; // null terminating string
298     }
299     else if (*error == ERROR_OK) { // str is NULL
300         *error = ERROR_MEM; // but error needs to be set
301     }
302     return str;
303 }
304
305 char* enlarge_string(size_t len, size_t *capacity, char *str)
306 {
307     char *t = realloc(str, *capacity * 2 + 1);
308     if (!t) {
309         *error = ERROR_MEM;
310         return NULL; // indicate error
311     }
312     else {
313         *capacity *= 2;
314     }
315     return t;
316 }
317
318 char* handle_str(char r, size_t l, char *str, int *error)
319 {
320     if (r != '\n') { // end-of-line has not been read
321         *error = ERROR_IN; // report input error
322         free(str);
323         str = NULL;
324     } else {
325         str[l] = '\0'; // null terminating string
326     }
327     else if (*error == ERROR_OK) { // str is NULL
328         *error = ERROR_MEM; // but error needs to be set
329     }
330     return str;
331 }
332
333 char* enlarge_string(size_t len, size_t *capacity, char *str)
334 {
335     char *t = realloc(str, *capacity * 2 + 1);
336     if (!t) {
337         *error = ERROR_MEM;
338         return NULL; // indicate error
339     }
340     else {
341         *capacity *= 2;
342     }
343     return t;
344 }
345
346 char* handle_str(char r, size_t l, char *str, int *error)
347 {
348     if (r != '\n') { // end-of-line has not been read
349         *error = ERROR_IN; // report input error
350         free(str);
351         str = NULL;
352     } else {
353         str[l] = '\0'; // null terminating string
354     }
355     else if (*error == ERROR_OK) { // str is NULL
356         *error = ERROR_MEM; // but error needs to be set
357     }
358     return str;
359 }
360
361 char* enlarge_string(size_t len, size_t *capacity, char *str)
362 {
363     char *t = realloc(str, *capacity * 2 + 1);
364     if (!t) {
365         *error = ERROR_MEM;
366         return NULL; // indicate error
367     }
368     else {
369         *capacity *= 2;
370     }
371     return t;
372 }
373
374 char* handle_str(char r, size_t l, char *str, int *error)
375 {
376     if (r != '\n') { // end-of-line has not been read
377         *error = ERROR_IN; // report input error
378         free(str);
379         str = NULL;
380     } else {
381         str[l] = '\0'; // null terminating string
382     }
383     else if (*error == ERROR_OK) { // str is NULL
384         *error = ERROR_MEM; // but error needs to be set
385     }
386     return str;
387 }
388
389 char* enlarge_string(size_t len, size_t *capacity, char *str)
390 {
391     char *t = realloc(str, *capacity * 2 + 1);
392     if (!t) {
393         *error = ERROR_MEM;
394         return NULL; // indicate error
395     }
396     else {
397         *capacity *= 2;
398     }
399     return t;
400 }
401
402 char* handle_str(char r, size_t l, char *str, int *error)
403 {
404     if (r != '\n') { // end-of-line has not been read
405         *error = ERROR_IN; // report input error
406         free(str);
407         str = NULL;
408     } else {
409         str[l] = '\0'; // null terminating string
410     }
411     else if (*error == ERROR_OK) { // str is NULL
412         *error = ERROR_MEM; // but error needs to be set
413     }
414     return str;
415 }
416
417 char* enlarge_string(size_t len, size_t *capacity, char *str)
418 {
419     char *t = realloc(str, *capacity * 2 + 1);
420     if (!t) {
421         *error = ERROR_MEM;
422         return NULL; // indicate error
423     }
424     else {
425         *capacity *= 2;
426     }
427     return t;
428 }
429
430 char* handle_str(char r, size_t l, char *str, int *error)
431 {
432     if (r != '\n') { // end-of-line has not been read
433         *error = ERROR_IN; // report input error
434         free(str);
435         str = NULL;
436     } else {
437         str[l] = '\0'; // null terminating string
438     }
439     else if (*error == ERROR_OK) { // str is NULL
440         *error = ERROR_MEM; // but error needs to be set
441     }
442     return str;
443 }
444
445 char* enlarge_string(size_t len, size_t *capacity, char *str)
446 {
447     char *t = realloc(str, *capacity * 2 + 1);
448     if (!t) {
449         *error = ERROR_MEM;
450         return NULL; // indicate error
451     }
452     else {
453         *capacity *= 2;
454     }
455     return t;
456 }
457
458 char* handle_str(char r, size_t l, char *str, int *error)
459 {
460     if (r != '\n') { // end-of-line has not been read
461         *error = ERROR_IN; // report input error
462         free(str);
463         str = NULL;
464     } else {
465         str[l] = '\0'; // null terminating string
466     }
467     else if (*error == ERROR_OK) { // str is NULL
468         *error = ERROR_MEM; // but error needs to be set
469     }
470     return str;
471 }
472
473 char* enlarge_string(size_t len, size_t *capacity, char *str)
474 {
475     char *t = realloc(str, *capacity * 2 + 1);
476     if (!t) {
477         *error = ERROR_MEM;
478         return NULL; // indicate error
479     }
480     else {
481         *capacity *= 2;
482     }
483     return t;
484 }
485
486 char* handle_str(char r, size_t l, char *str, int *error)
487 {
488     if (r != '\n') { // end-of-line has not been read
489         *error = ERROR_IN; // report input error
490         free(str);
491         str = NULL;
492     } else {
493         str[l] = '\0'; // null terminating string
494     }
495     else if (*error == ERROR_OK) { // str is NULL
496         *error = ERROR_MEM; // but error needs to be set
497     }
498     return str;
499 }
500
501 char* enlarge_string(size_t len, size_t *capacity, char *str)
502 {
503     char *t = realloc(str, *capacity * 2 + 1);
504     if (!t) {
505         *error = ERROR_MEM;
506         return NULL; // indicate error
507     }
508     else {
509         *capacity *= 2;
510     }
511     return t;
512 }
513
514 char* handle_str(char r, size_t l, char *str, int *error)
515 {
516     if (r != '\n') { // end-of-line has not been read
517         *error = ERROR_IN; // report input error
518         free(str);
519         str = NULL;
520     } else {
521         str[l] = '\0'; // null terminating string
522     }
523     else if (*error == ERROR_OK) { // str is NULL
524         *error = ERROR_MEM; // but error needs to be set
525     }
526     return str;
527 }
528
529 char* enlarge_string(size_t len, size_t *capacity, char *str)
530 {
531     char *t = realloc(str, *capacity * 2 + 1);
532     if (!t) {
533         *error = ERROR_MEM;
534         return NULL; // indicate error
535     }
536     else {
537         *capacity *= 2;
538     }
539     return t;
540 }
541
542 char* handle_str(char r, size_t l, char *str, int *error)
543 {
544     if (r != '\n') { // end-of-line has not been read
545         *error = ERROR_IN; // report input error
546         free(str);
547         str = NULL;
548     } else {
549         str[l] = '\0'; // null terminating string
550     }
551     else if (*error == ERROR_OK) { // str is NULL
552         *error = ERROR_MEM; // but error needs to be set
553     }
554     return str;
555 }
556
557 char* enlarge_string(size_t len, size_t *capacity, char *str)
558 {
559     char *t = realloc(str, *capacity * 2 + 1);
560     if (!t) {
561         *error = ERROR_MEM;
562         return NULL; // indicate error
563     }
564     else {
565         *capacity *= 2;
566     }
567     return t;
568 }
569
570 char* handle_str(char r, size_t l, char *str, int *error)
571 {
572     if (r != '\n') { // end-of-line has not been read
573         *error = ERROR_IN; // report input error
574         free(str);
575         str = NULL;
576     } else {
577         str[l] = '\0'; // null terminating string
578     }
579     else if (*error == ERROR_OK) { // str is NULL
580         *error = ERROR_MEM; // but error needs to be set
581     }
582     return str;
583 }
584
585 char* enlarge_string(size_t len, size_t *capacity, char *str)
586 {
587     char *t = realloc(str, *capacity * 2 + 1);
588     if (!t) {
589         *error = ERROR_MEM;
590         return NULL; // indicate error
591     }
592     else {
593         *capacity *= 2;
594     }
595     return t;
596 }
597
598 char* handle_str(char r, size_t l, char *str, int *error)
599 {
600     if (r != '\n') { // end-of-line has not been read
601         *error = ERROR_IN; // report input error
602         free(str);
603         str = NULL;
604     } else {
605         str[l] = '\0'; // null terminating string
606     }
607     else if (*error == ERROR_OK) { // str is NULL
608         *error = ERROR_MEM; // but error needs to be set
609     }
610     return str;
611 }
612
613 char* enlarge_string(size_t len, size_t *capacity, char *str)
614 {
615     char *t = realloc(str, *capacity * 2 + 1);
616     if (!t) {
617         *error = ERROR_MEM;
618         return NULL; // indicate error
619     }
620     else {
621         *capacity *= 2;
622     }
623     return t;
624 }
625
626 char* handle_str(char r, size_t l, char *str, int *error)
627 {
628     if (r != '\n') { // end-of-line has not been read
629         *error = ERROR_IN; // report input error
630         free(str);
631         str = NULL;
632     } else {
633         str[l] = '\0'; // null terminating string
634     }
635     else if (*error == ERROR_OK) { // str is NULL
636         *error = ERROR_MEM; // but error needs to be set
637     }
638     return str;
639 }
640
641 char* enlarge_string(size_t len, size_t *capacity, char *str)
642 {
643     char *t = realloc(str, *capacity * 2 + 1);
644     if (!t) {
645         *error = ERROR_MEM;
646         return NULL; // indicate error
647     }
648     else {
649         *capacity *= 2;
650     }
651     return t;
652 }
653
654 char* handle_str(char r, size_t l, char *str, int *error)
655 {
656     if (r != '\n') { // end-of-line has not been read
657         *error = ERROR_IN; // report input error
658         free(str);
659         str = NULL;
660     } else {
661         str[l] = '\0'; // null terminating string
662     }
663     else if (*error == ERROR_OK) { // str is NULL
664         *error = ERROR_MEM; // but error needs to be set
665     }
666     return str;
667 }
668
669 char* enlarge_string(size_t len, size_t *capacity, char *str)
670 {
671     char *t = realloc(str, *capacity * 2 + 1);
672     if (!t) {
673         *error = ERROR_MEM;
674         return NULL; // indicate error
675     }
676     else {
677         *capacity *= 2;
678     }
679     return t;
680 }
681
682 char* handle_str(char r, size_t l, char *str, int *error)
683 {
684     if (r != '\n') { // end-of-line has not been read
685         *error = ERROR_IN; // report input error
686         free(str);
687         str = NULL;
688     } else {
689         str[l] = '\0'; // null terminating string
690     }
691     else if (*error == ERROR_OK) { // str is NULL
692         *error = ERROR_MEM; // but error needs to be set
693     }
694     return str;
695 }
696
697 char* enlarge_string(size_t len, size_t *capacity, char *str)
698 {
699     char *t = realloc(str, *capacity * 2 + 1);
700     if (!t) {
701         *error = ERROR_MEM;
702         return NULL; // indicate error
703     }
704     else {
705         *capacity *= 2;
706     }
707     return t;
708 }
709
710 char* handle_str(char r, size_t l, char *str, int *error)
711 {
712     if (r != '\n') { // end-of-line has not been read
713         *error = ERROR_IN; // report input error
714         free(str);
715         str = NULL;
716     } else {
717         str[l] = '\0'; // null terminating string
718     }
719     else if (*error == ERROR_OK) { // str is NULL
720         *error = ERROR_MEM; // but error needs to be set
721     }
722     return str;
723 }
724
725 char* enlarge_string(size_t len, size_t *capacity, char *str)
726 {
727     char *t = realloc(str, *capacity * 2 + 1);
728     if (!t) {
729         *error = ERROR_MEM;
730         return NULL; // indicate error
731     }
732     else {
733         *capacity *= 2;
734     }
735     return t;
736 }
737
738 char* handle_str(char r, size_t l, char *str, int *error)
739 {
740     if (r != '\n') { // end-of-line has not been read
741         *error = ERROR_IN; // report input error
742         free(str);
743         str = NULL;
744     } else {
745         str[l] = '\0'; // null terminating string
746     }
747     else if (*error == ERROR_OK) { // str is NULL
748         *error = ERROR_MEM; // but error needs to be set
749     }
750     return str;
751 }
752
753 char* enlarge_string(size_t len, size_t *capacity, char *str)
754 {
755     char *t = realloc(str, *capacity * 2 + 1);
756     if (!t) {
757         *error = ERROR_MEM;
758         return NULL; // indicate error
759     }
760     else {
761         *capacity *= 2;
762     }
763     return t;
764 }
765
766 char* handle_str(char r, size_t l, char *str, int *error)
767 {
768     if (r != '\n') { // end-of-line has not been read
769         *error = ERROR_IN; // report input error
770         free(str);
771         str = NULL;
772     } else {
773         str[l] = '\0'; // null terminating string
774     }
775     else if (*error == ERROR_OK) { // str is NULL
776         *error = ERROR_MEM; // but error needs to be set
777     }
778     return str;
779 }
780
781 char* enlarge_string(size_t len, size_t *capacity, char *str)
782 {
783     char *t = realloc(str, *capacity * 2 + 1);
784     if (!t) {
785         *error = ERROR_MEM;
786         return NULL; // indicate error
787     }
788     else {
789         *capacity *= 2;
790     }
791     return t;
792 }
793
794 char* handle_str(char r, size_t l, char *str, int *error)
795 {
796     if (r != '\n') { // end-of-line has not been read
797         *error = ERROR_IN; // report input error
798         free(str);
799         str = NULL;
800     } else {
801         str[l] = '\0'; // null terminating string
802     }
803     else if (*error == ERROR_OK) { // str is NULL
804         *error = ERROR_MEM; // but error needs to be set
805     }
806     return str;
807 }
808
809 char* enlarge_string(size_t len, size_t *capacity, char *str)
810 {
811     char *t = realloc(str, *capacity * 2 + 1);
812     if (!t) {
813         *error = ERROR_MEM;
814         return NULL; // indicate error
815     }
816     else {
817         *capacity *= 2;
818     }
819     return t;
820 }
821
822 char* handle_str(char r, size_t l, char *str, int *error)
823 {
824     if (r != '\n') { // end-of-line has not been read
825         *error = ERROR_IN; // report input error
826         free(str);
827         str = NULL;
828     } else {
829         str[l] = '\0'; // null terminating string
830     }
831     else if (*error == ERROR_OK) { // str is NULL
832         *error = ERROR_MEM; // but error needs to be set
833     }
834     return str;
835 }
836
837 char* enlarge_string(size_t len, size_t *capacity, char *str)
838 {
839     char *t = realloc(str, *capacity * 2 + 1);
840     if (!t) {
841         *error = ERROR_MEM;
842         return NULL; // indicate error
843     }
844     else {
845         *capacity *= 2;
846     }
847     return t;
848 }
849
850 char* handle_str(char r, size_t l, char *str, int *error)
851 {
852     if (r != '\n') { // end-of-line has not been read
853         *error = ERROR_IN; // report input error
854         free(str);
855         str = NULL;
856     } else {
857         str[l] = '\0'; // null terminating string
858     }
859     else if (*error == ERROR_OK) { // str is NULL
860         *error = ERROR_MEM; // but error needs to be set
861     }
862     return str;
863 }
864
865 char* enlarge_string(size_t len, size_t *capacity, char *str)
866 {
867     char *t = realloc(str, *capacity * 2 + 1);
868     if (!t) {
869         *error = ERROR_MEM;
870         return NULL; // indicate error
871     }
872     else {
873         *capacity *= 2;
874     }
875     return t;
876 }
877
878 char* handle_str(char r, size_t l, char *str, int *error)
879 {
880     if (r != '\n') { // end-of-line has not been read
881         *error = ERROR_IN; // report input error
882         free(str);
883         str = NULL;
884     } else {
885         str[l] = '\0'; // null terminating string
886     }
887     else if (*error == ERROR_OK) { // str is NULL
888         *error = ERROR_MEM; // but error needs to be set
889     }
890     return str;
891 }
892
893 char* enlarge_string(size_t len, size_t *capacity, char *str)
894 {
895     char *t = realloc(str, *capacity * 2 + 1);
896     if (!t) {
897         *error = ERROR_MEM;
898         return NULL; // indicate error
899     }
900     else {
901         *capacity *= 2;
902     }
903     return t;
904 }
905
906 char* handle_str(char r, size_t l, char *str, int *error)
907 {
908     if (r != '\n') { // end-of-line has not been read
909         *error = ERROR_IN; // report input error
910         free(str);
911         str = NULL;
912     } else {
913         str[l] = '\0'; // null terminating string
914     }
915     else if (*error == ERROR_OK) { // str is NULL
916         *error = ERROR_MEM; // but error needs to be set
917     }
918     return str;
919 }
920
921 char* enlarge_string(size_t len, size_t *capacity, char *str)
922 {
923     char *t = realloc(str, *capacity * 2 + 1);
924     if (!t) {
925         *error = ERROR_MEM;
926         return NULL; // indicate error
927     }
928     else {
929         *capacity *= 2;
930     }
931     return t;
932 }
933
934 char* handle_str(char r, size_t l, char *str, int *error)
935 {
936     if (r != '\n') { // end-of-line has not been read
937         *error = ERROR_IN; // report input error
938         free(str);
939         str = NULL;
940     } else {
941         str[l] = '\0'; // null terminating string
942     }
943     else if (*error == ERROR_OK) { // str is NULL
944         *error = ERROR_MEM; // but error needs to be set
945     }
946     return str;
947 }
948
949 char* enlarge_string(size_t len, size_t *capacity, char *str)
950 {
951     char *t = realloc(str, *capacity * 2 + 1);
952     if (!t) {
953         *error = ERROR_MEM;
954         return NULL; // indicate error
955     }
956     else {
957         *capacity *= 2;
958     }
959     return t;
960 }
961
962 char* handle_str(char r, size_t l, char *str, int *error)
963 {
964     if (r != '\n') { // end-of-line has not been read
965         *error = ERROR_IN; // report input error
966         free(str);
967         str = NULL;
968     } else {
969         str[l] = '\0'; // null terminating string
970     }
971     else if (*error == ERROR_OK) { // str is NULL
972         *error = ERROR_MEM; // but error needs to be set
973     }
974     return str;
975 }
976
977 char* enlarge_string(size_t len, size_t *capacity, char *str)
978 {
979     char *t = realloc(str, *capacity * 2 + 1);
980     if (!t) {
981         *error = ERROR_MEM;
982         return NULL; // indicate error
983     }
984     else {
985         *capacity *= 2;
986     }
987     return t;
988 }
989
990 char* handle_str(char r, size_t l, char *str, int *error)
991 {
992     if (r != '\n') { // end-of-line has not been read
993         *error = ERROR_IN; // report input error
994         free(str);
995         str = NULL;
996     } else {
997         str[l] = '\0'; // null terminating string
998     }
999     else if (*error == ERROR_OK) { // str is NULL
1000        *error = ERROR_MEM; // but error needs to be set
1001    }
1002    return str;
1003 }
1004
1005 char* enlarge_string(size_t len, size_t *capacity, char *str)
1006 {
1007     char *t = realloc(str, *capacity * 2 + 1);
1008     if (!t) {
1009         *error = ERROR_MEM;
1010         return NULL; // indicate error
1011     }
1012     else {
1013         *capacity *= 2;
1014     }
1015     return t;
1016 }
1017
1018 char* handle_str(char r, size_t l, char *str, int *error)
1019 {
1020     if (r != '\n') { // end-of-line has not been read
1021         *error = ERROR_IN; // report input error
1022         free(str);
1023         str = NULL;
1024     } else {
1025         str[l] = '\0'; // null terminating string
1026     }
1027     else if (*error == ERROR_OK) { // str is NULL
1028         *error = ERROR_MEM; // but error needs to be set
1029     }
1030     return str;
1031 }
1032
1033 char* enlarge_string(size_t len, size_t *capacity, char *str)
1034 {
1035     char *t = realloc(str, *capacity * 2 + 1);
1036     if (!t) {
1037         *error = ERROR_MEM;
1038         return NULL; // indicate error
1039     }
1040     else {
1041         *capacity *= 2;
1042     }
1043     return t;
1044 }
1045
1046 char* handle_str(char r, size_t l, char *str, int *error)
1047 {
1048     if (r != '\n') { // end-of-line has not been read
1049         *error = ERROR_IN; // report input error
1050         free(str);
1051         str = NULL;
1052     } else {
1053         str[l] = '\0'; // null terminating string
1054     }
1055     else if (*error == ERROR_OK) { // str is NULL
1056         *error = ERROR_MEM; // but error needs to be set
1057     }
1058     return str;
1059 }
1060
1061 char* enlarge_string(size_t len, size_t *capacity, char *str)
1062 {
1063     char *t = realloc(str, *capacity * 2 + 1);
1064     if (!t) {
1065         *error = ERROR_MEM;
1066         return NULL; // indicate error
1067     }
1068     else {
1069         *capacity *= 2;
1070     }
1071     return t;
1072 }
1073
1074 char* handle_str(char r, size_t l, char *str, int *error)
1075 {
1076     if (r != '\n') { // end-of-line has not been read
1077         *error = ERROR_IN; // report input error
1078         free(str);
1079         str = NULL;
1080     } else {
1081         str[l] = '\0'; // null terminating string
1082     }
1083     else if (*error == ERROR_OK) { // str is NULL
1084         *error = ERROR_MEM; // but error needs to be set
1085     }
1086     return str;
1087 }
1088
1089 char* enlarge_string(size_t len, size_t *capacity, char *str)
1090 {
1091     char *t = realloc(str, *capacity * 2 + 1);
1092     if (!t) {
1093         *error = ERROR_MEM;
1094         return NULL; // indicate error
1095     }
1096     else {
1097         *capacity *= 2;
1098     }
1099     return t;
1100 }
1101
1102 char* handle_str(char r, size_t l, char *str, int *error)
1103 {
1104     if (r != '\n') { // end-of-line has not been read
1105         *error = ERROR_IN; // report input error
1106         free(str);
1107         str = NULL;
1108     } else {
1109         str[l] = '\0'; // null terminating string
1110     }
1111     else if (*error == ERROR_OK) { // str is NULL
1112         *error = ERROR_MEM; // but error needs to be set
1113     }
1114     return str;
1115 }
1116
1117 char* enlarge_string(size_t len, size_t *capacity, char *str)
1118 {
1119     char *t = realloc(str, *capacity * 2 + 1);
1120     if (!t) {
1121         *error = ERROR_MEM;
1122         return NULL; // indicate error
1123     }
1124     else {
1125         *capacity *= 2;
1126     }
1127     return t;
1128 }
1129
1130 char* handle_str(char r, size_t l, char *str, int *error)
1131 {
1132     if (r != '\n') { // end-of-line has not been read
1133         *error = ERROR_IN; // report input error
1134         free(str);
1135         str = NULL;
1136     } else {
1137         str[l] = '\0'; // null terminating string
1138     }
1139     else if (*error == ERROR_OK) { // str is NULL
1140         *error = ERROR_MEM; // but error needs to be set
1141     }
1142     return str;
1143 }
1144
```

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
2   variable with initialization is a
3   definition */
4 int global_variable = 10;
5 void function(int p);      lec05/extern_var.h
6
7 #include <stdio.h>
8 #include "extern_var.h"
9 static int module_variable;
10 void function(int p)
11 {
12     fprintf(stdout, "function: p %d global
13     variable %d\n", p, global_variable);
14 }
15                                     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 }
11                                     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)`.

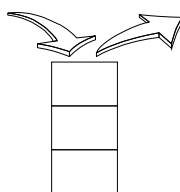
Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 33 / 48

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
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.
- 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;
        } else {
            break;
        }
    }
}

```

```

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

```

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

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`
- Zdrojový soubor `vardec.c`

```

1 extern int global_variable;
2                                     lec05/vardec.h

```

```

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 }
13                                     lec05/vardec.c

```

```

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 }

```

■ 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.
 - Vícelslovná 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

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 III

Část 3 – Zadání 5. domácího úkolu (HW5)

Diskutovaná témata

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`.

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 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 count_words(const char * const * const words)
11 {
12     int n = 0;
13     // cur je ukazatel na data typu konstantní
14     // ukazatel na konstantní textový řetězec
15     // (na konstantní ukazatel na konstantní hodnoty char).
16     const char * const * cur = words; // cur chceme měnit
17     while (*cur) {
18         cur++; // cur není konstantní ukazatel
19         n++;
20     }
21     return n;
22 }
```

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

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("%c ", 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 }

```

```

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

```

- 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 posoupněm znaků zakončená '\0'.
- První písmeno slova abecedy používáme k indexaci, např. 'C'harlie je odkazované ukazatelem 'D'elta - 'A'. První znak slova tak můžeme v abecedě `alphabet` nahradit znakem '\0' získáme textové řetězce.

```

1 //Ukazatel na textový literál. Literál nemůžeme měnit!
2 //static char *alphabet = "AlphaBravoCharlie...";
3 static char alphabet[] =
4     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
5     "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
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     return ret;
20 }

```

```

1 int fill_words(char* alphabet, char *words[])
2 {
3     int ret = EXIT_SUCCESS;
4     char *cur = alphabet; // kurzor do pole s písmeny abecedy
5     for (char c = 'A'; c <= 'Z'; ++c) {
6         assert(words[c - 'A'] == NULL); // nemá být nastaveno
7         cur = strchr(cur, c); // vyhledání řetězce začínající c
8         assert(cur); // písmeno c musí být v abecedě
9         *cur = '\0';
10        words[c - 'A'] = *cur; // nastavení a posun kurzoru
11        assert(words[c - 'A']); // it should be set now
12        return ret; // pragmatically vždy EXIT_SUCCESS nebo assert.
13    }
14 }

```

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

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

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

```

1 static char alphabet[] =
2     "AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
3     "JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
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    int c;
17    while ((c = getchar()) != EOF) {
18        c = (c >= 'a' && c <= 'z') ? c - 'a' + 'A' : c;
19        // i volání funkce toupper() bude zpřáhlednějí!
20        if (c >= 'A' && c <= 'Z') {
21            printf("%c\n", words[c - 'A']);
22        }
23    }
24    return ret;
25 }

```

```

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

```

- 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 enum { ERROR_OK = EXIT_SUCCESS, ERROR_IN = 100, ERROR_MEM = 129 };
9 void* my_realloc(void *ptr, size_t size, const char *file, const int line);
10 void* ret = realloc(ptr, size);
11 if ((ret) {
12     fprintf(stderr, "ERROR: Cannot realloc %lu bytes -- called at %s:%d", size, file, line);
13     free(ptr);
14     exit(ERROR_MEM);
15 }
16 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
15 char* shift(int offset, const char* src, size_t n, char *dst); // src and dst are strings at least n long (+1 for '\0')
16 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
17 int print_offset(const char *s, size_t n, int offset);
18 int main(void)
19 {
20     int ret = ERROR_OK;
21     char *l1 = read_line();
22     char *l2 = read_line();
23     size_t n1, n2;
24     if ((l1 && l2 && (n1 = strlen(l1)) == (n2 = strlen(l2))) {
25         fprintf(stderr, "DEBUG: 11[%lu]: \"%s\n\"", n1, l1);
26         fprintf(stderr, "DEBUG: 12[%lu]: \"%s\n\"", n2, l2);
27         int offset = get_offset(l1, n1, l2, n2);
28         fprintf(stdout, "Matching offset %d\n", offset);
29         offset >= 0 && print_offset(l2, n2, offset); // call print_offset only if offset >= 0
30     } else {
31         fprintf(stderr, "ERROR: Wrong input!\n");
32         ret = ERROR_IN;
33     }
34     free(l1); // free(ptr) - If ptr is NULL no action occurs.
35     free(l2); // See man free.
36     return ret;
37 }

```

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            capacity *= 2;
11            str = my_realloc(str, sizeof(char) * (capacity + 1),
12                __FILE__, __LINE__); //+1 for '\0'
13        }
14        str[len++] = c;
15    }
16    if ((len > 0) {
17        str[len] = '\0';
18    } else {
19        free(str);
20    }
21    return str;
22 }

```

- `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 int get_offset(const char *s1, size_t n1, const char *s2, size_t n2)
89 { // we already checked that s1 && s2 && n1 == n2
90     int ret = -1;
91     int max_shift = INT_MAX < n2 ? INT_MAX : n2; // limits n
92     char *s = my_realloc(NULL, sizeof(char) * (n2 + 1), __FILE__,
93         __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, release 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 ukončíme funkcí `free()`.

```

100 int print_offset(const char *s, size_t n, int offset)
101 {
102     int ret = 1;
103     char *str = my_realloc(NULL, sizeof(char) * (n + 1),
104         __FILE__, __LINE__); //+1 for '\0'
105     shift(offset, s, n, str);
106     fprintf(stderr, "DEBUG: shift: \"%s\n", str);
107     free(str);
108     return ret;
109 }

```

- Program otestujeme pro ukázkový vstup.

```

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: 11[27]: "Lorem ipsum dolor sit amet."
6 DEBUG: 12[27]: "sit amet.Lorem ipsum dolor "
7 Matching offset 9
8 DEBUG: shift: "Lorem ipsum dolor sit amet."
9

```

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

```

103 for (size_t i = 0; i < n; ++i) {
104     dst[i] = src[(offset + i) % n];
105 }
106 $ valgrind ./shift < input.txt
107 ...
108 ==80708== Invalid write of size 1
109 ==80708== at 0x202240: shift (shift.c:84)
110 ==80708== by 0x202092: get_offset (shift.c:95)
111 ==80708== by 0x201DF2: main (shift.c:36)

```