Programování v jazyce C

8 Práce s pamětí

- Mapa paměti a správa paměti
- Alokace a dealokace bloků
- Dynamické proměnné
- Adresování paměti
- Znakové řetězce
- Knihovna `string`

Správa paměti prostředky ANSI C
Obecné skutečnosti

- Funkce a datové typy pro správu paměti jsou v knihovně **stdlib** → připojit hlavičkový soubor příkazem preprocesoru: `#include <stdlib.h>`

- Jazyk C nemá žádné prostředky pro automatický management paměti, žádný „garbage collector“ – **vše je v rukou programátora**.
- Přidělenou paměť musí program **uvolňovat** (OS uvolní paměť až při ukončení procesu), jinak může v průběhu výpočtu „dojít“ → tzn. jakmile program daný paměťový blok nepotřebuje, měl by ho „vrátit“.

Při práci s pamětí dochází k naprosté většině všech programátorských chyb v C/C++!

Správa paměti prostředky ANSI C
Obecné skutečnosti

- Paměťové operace probíhají prostřednictvím **ukazatelů**, tj. teoreticky je programátorovi k dispozici celý adresní prostor → **může docházet ke konfliktům s jinými procesy** (moderní OS tomu brání)

- **Překladač C nekontroluje platnost** (obsah) **ukazatelů** předávaných jako parametry funkcím pro správu paměti, tj. lze pracovat s pamětí, která patří jinému procesu nebo vůbec neexistuje (adresa nemusí být platná).

- K úspěšné práci s pamětí v C je **nezbytně nutné** chápat alespoň základní principy organizace paměti z hlediska CPU a OS
  → ovšem v různých OS a různé implementace standardní knihovny C mohou s pamětí zacházet různě...

Základní funkce pro práci s pamětí
Alokace a dealokace (uvolnění) bloku paměti

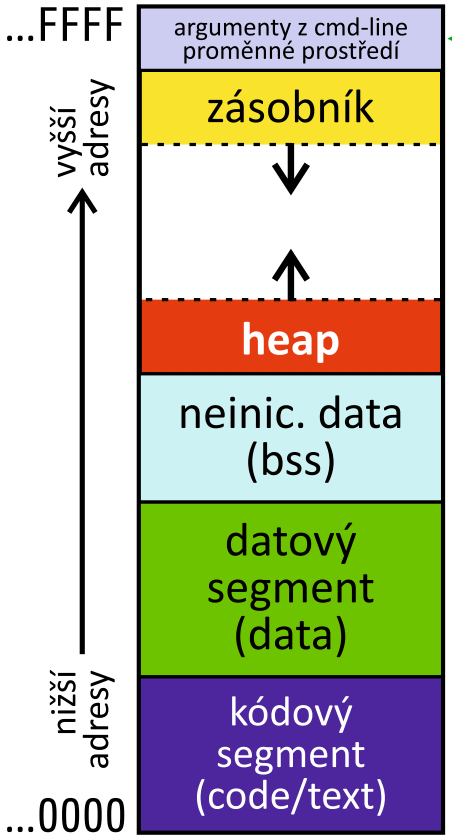
- **Alokace**, tj. získání bloku paměti pro potřeby programu, se provádí voláním funkce `void *malloc(size_t size);`
  dealokaci (uvolnění) paměťového bloku zajišťuje funkce `void free(void *ptr);`

```
#include <stdlib.h>

void main() {
  char *str = NULL;

  str = (char *) malloc(1024);
  ⋮

  if (str != NULL)
    free(str);
}
```

statická proměnná (ukazatel)

zde vzniká **dynamicky** pole – adresa jeho počátku je uložena do ukazatele `str`

Základní funkce pro práci s pamětí
Bližší pohled na alokaci – mapa paměti

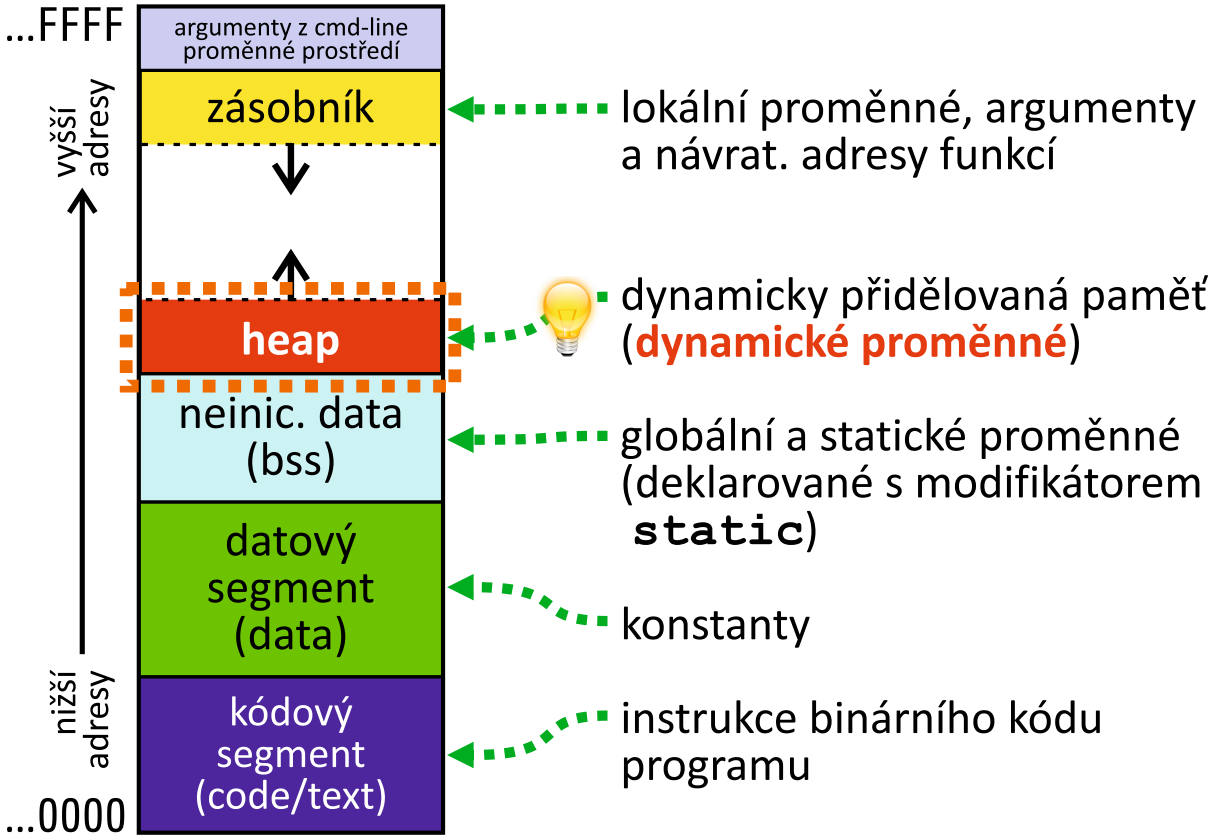


Flat (Linear) Memory Model:

- dnes prakticky všechny moderní OS používají tento způsob adresování (pokud ho „umí“ CPU);
- indexem do paměti (tj. obsahem ukazatele) je tzv. **logická adresa** – 32/64-bitové číslo, **není shodná s fyzickou adr!**;
- takovýchto ***segmentů*** může být mnoho (kolik dovolí fyzická velikost instalované RAM) → každý proces „vidí“, tj. může adresovat, **pouze ten svůj**.

Základní funkce pro práci s pamětí

Bližší pohled na alokaci – mapa paměti



Základní funkce pro práci s pamětí
Bližší pohled na alokaci – co se děje v paměti

```
char *str = NULL;
int i = 0;
⋮
str = malloc(1024);
⋮
for (i = 0; i < 1024; i++)
    str[i] = '\040';
⋮
free(str);
```

1

2

`malloc(·)` žádá 1024 byte paměti a vrací ukazatel na první prvek získaného bloku

`free(·)` „vrátí“ blok paměti, na který ukazuje **`str`**, zpět systému

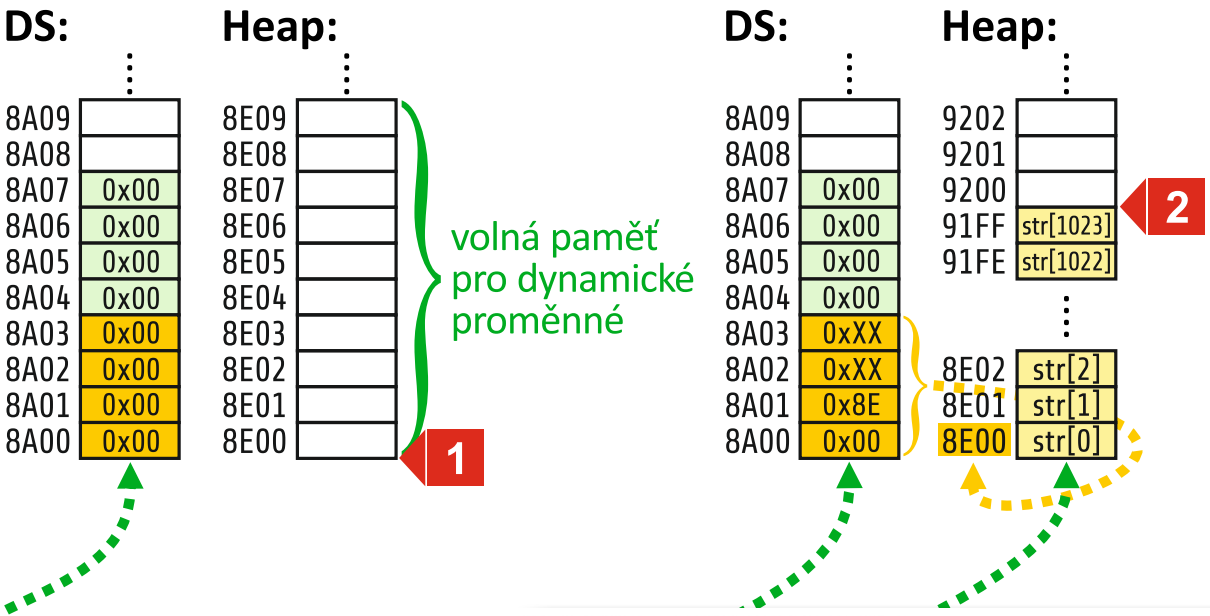
1 zde existují pouze statické proměnné **`int i`** a **`char *str`** – ukazatel na znak(ové pole),

2 tímto vznikla ***dynamická proměnná*** – pole 1024 znaků

- za voláním funkce **`free(·)`** už nesmí dojít k přístupu do dynamické proměnné, na kterou ukazoval ukazatel **`str`**, tj. **`*str = …`** nebo **`str[·] = …`**

Základní funkce pro práci s pamětí

Bližší pohled na alokaci – co se děje v paměti




```
char *str = NULL;
int i = 0;
⋮
str = malloc(1024);
```

The **RtlAllocateHeap** routine allocates a block of memory from a heap.

Syntax

Microsoft RtlAllocateHeap routine

C++

```
PVOID RtlAllocateHeap(
  _In_     PVOID  HeapHandle,
  _In_opt_ ULONG  Flags,
  _In_     SIZE_T Size
);
```

možný způsob imple-mentace v OS Windows

Alokace paměti
Deklarovaný pointer ≠ dynamická proměnná

```
typedef struct thenode {
   int key;
   thenode *left, *right;
} node;
⋮
node *n;
n->key = 0;
⋮
n = (node *) malloc(sizeof(node));
if (n) {
   n->key = 1;
   n->left = NULL;
   n->right = NULL;
}
else
   fprintf(stderr, "Out of memory!\n");
```

NELZE! – v tuto chvíli existuje pouze statický ukazatel **`n`**, nikoliv dyn. instance struktury!

test, zda se alokace podařila (v případě neúspěchu vrací **`malloc(·)`** hodnotu **`NULL`**)

Alokace paměti
Deklarovaný pointer ≠ dynamická proměnná

- Deklarace ukazatele na proměnnou typu **`struct node`** vede pouze k vytvoření statické proměnné typu ukazatel (32 či 64 bitů v datovém segmentu), nikoliv k vytvoření samotné struktury **`node`** v paměti.

```
⋮
node *n;
⋮
n = (node *) malloc(sizeof(node));
n->key = 1;
```

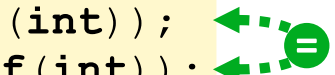
- Prostor v paměti pro strukturu **`node`** vytváří teprve alokace, tj. volání funkce **`malloc(·)`**, tzn. teprve nyní vznikla (dynamicky) proměnná, jejíž podoba je dána definicí struktury **`node`**, tj. vznikla její ***instance***.
- Ukazatel **`n`** je tedy tzv. ***referenční proměnnou*** dynamické proměnné typu **`struct node`**.

Alokace pole prvků v paměti
Funkce pro alokaci souvislého bloku

```
void *calloc(size_t count, size_t size);
```

- Alokuje **souvislou oblast** (tj. pole) o počtu **`count`** položek, přičemž velikost každé položky je **`size`** bytů a celou oblast vynuluje.
- Pokud není nutné oblast inicializovat (nulovat) a není vyžadovaná souvislost, lze nahradit voláním funkce **`malloc(·)`**:

```
int *arr;
arr = (int *) calloc(100, sizeof(int));
arr = (int *) malloc(100 * sizeof(int));
```



POZOR: Zde ↓ není zaručeno, že budou alokované **`int`**y ležet v paměti vedle sebe, tj. tvořit souvislé pole…

```
int *arr[256], i;

for (i = 0; i < 256; i++) {
  arr[i] = malloc(sizeof(int));
  ⋮
```

Realokace
Úprava velikosti alokovaného bloku paměti

```
void *realloc(void *ptr, size_t size);
```

- Funkce **`realloc(·)`** nastavuje velikost dynamické proměnné, na níž ukazuje **`ptr`** , na velikost **`size`** bytů – obsah paměti zůstane nedotčený, pokud je nová velikost větší než původní (přidaná oblast je nezinicializovaná).
- Pokud **`realloc(·)`** nezvládne přidat novou oblast za konec původní, alokuje zcela nový blok o požadované velikost a původní obsah tam nakopíruje.
- Vrací ukazatel na počátek upravené oblasti, v případě neúspěchu **`NULL`** (tehdy je obsah původní oblasti neporušen).
- Je-li nová velikost menší než původní, odřízne se konec dat.

`if`

`ptr == NULL`	▶ chová se jako **`malloc(·)`**
`ptr != NULL && size == 0`	▶ chová se jako **`free(·)`**

Realokace
Případová studie – ukládání příchozích vzorků

```
#define SAMPLE_INCR 100
int sample_lim = 0, sample_cnt = 0;
void *smp_blk = NULL;
double *samples = NULL;

int add_sample(double smp) {
  if (sample_cnt == sample_lim) {
    sample_lim += SAMPLE_INCR;
    smp_blk = realloc((void *) samples,
                      sample_lim * sizeof(double));
    if (!smp_blk) {
      fprintf(stderr, "Out of memory!\n");
      return 0;
    }
    else samples = smp_blk;
  }
  samples[sample_cnt++] = smp;
  return sample_cnt;
}
```

Dealokace paměti
Uvolnění alokovaného bloku (k dalšímu užití)

```
void free(void *ptr);
```

- Nemá návratovou hodnotu → nelze testovat úspěšnost!
- Každý alokovaný blok paměti (některou z funkcí **`malloc(·)`**, **`calloc(·)`** a **`realloc(·)`**) **musí být uvolněn**, jakmile ho již program nepotřebuje → to zabraňuje postupnému vyčerpání paměti.
- Bloky, které neuvolní program(átor) voláním **`free(·)`**, uvolní posléze (většina) OS při ukončení procesu (ale mezitím může paměť dojít).

Funkci **`free(·)`** je třeba předat **přesně ten ukazatel**, který vrátila alokační funkce – je-li jeho hodnota pozměněna např. prováděním ukazatelové aritmetiky, nedokáže **`free(·)`** paměť korektně uvolnit! (je to logické – tabulka alok. bloků)

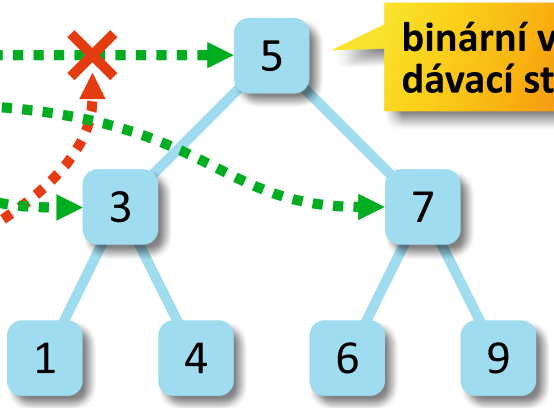
Dealokace paměti

Funkce `free(·)` není chytrá ani rekurzivní

```
node *root = …
     root->right
     root->left
```

NE

```
free(root);
```



↑ vede pouze ke **ztrátě reference** na celý strom, a tedy k nemožnosti ho kdykoliv v budoucnu uvolnit (či s ním jinak pracovat)!

```
void free_node(node *n) {
  if (n == NULL) return;

  if (n->left)
    free_node(n->left);
  if (n->right)
    free_node(n->right);
  free(n);
}
```

Dealokace paměti
Ladění programů s intenzivní správou paměti

```
unsigned int mem_blocks = 0;

⋮

void *getmem(size_t size) {
  void *ptr = malloc(size);
  if (ptr != NULL) mem_blocks++;

  return ptr;
}

void freemem(void **ptr) {
  if (*ptr != NULL) mem_blocks--;
  free(*ptr);
  *ptr = NULL;
}
```

globální proměnná s počtem alokovaných bloků paměti

tato proměnná musí mít v okamžiku skončení programu hodnotu 0…

z bezpečnostních důvodů bezodkladně vynulovat

- Hodnota **`NULL`** indikuje, že **dyn. proměnná** (již) **neexistuje!**

Alokace a dealokace paměti
Několik závěrečných poznámek

- Poté, co program(átor) paměť uvolní voláním **`free(·)`**, přestává dynamická proměnná fakticky existovat, tzn. **nelze k ní přistupovat!** (pokus o to vede obvykle k RTE)
- Ale **pozor**: statická referenční proměnná (ukazatel) samozřejmě existuje i nadále (jen ukazuje na místo, kde už „nic není“) → do ukazatele, který ukazoval na dyn. proměnnou je třeba okamžitě po jejím uvolnění přiřadit **`NULL`**, aby bylo jasné, že je **neplatný!**

```
⋮
free(ptr);
ptr = NULL;
⋮
```

- Pokus o uvolnění dyn. paměti, která nikdy nebyla alokovaná, vede buď k RTE nebo k „velmi podivnému chování“ programu (obtížně hledatelné chyby) → lepší verze:

! VŽDY OTESTOVAT PLATNOST POINTERU

```
if (ptr) free(ptr);
ptr = NULL;
```

Alokace a dealokace paměti
Několik závěrečných poznámek

- Je nutné aktivně bránit „vzniku“ pointerů, o kterých se neví, zda na něco ukazují nebo ne → pokud možno ne-deklarovat takto:

```
int *array;
⋮
```

o kus dál v kódu již není jasné, zda už dynamické pole vzniklo, či nikoliv…

ale raději **s okamžitou inicializací**:

```
int *arr1 = NULL;
int *arr2 = malloc(…);
⋮
```

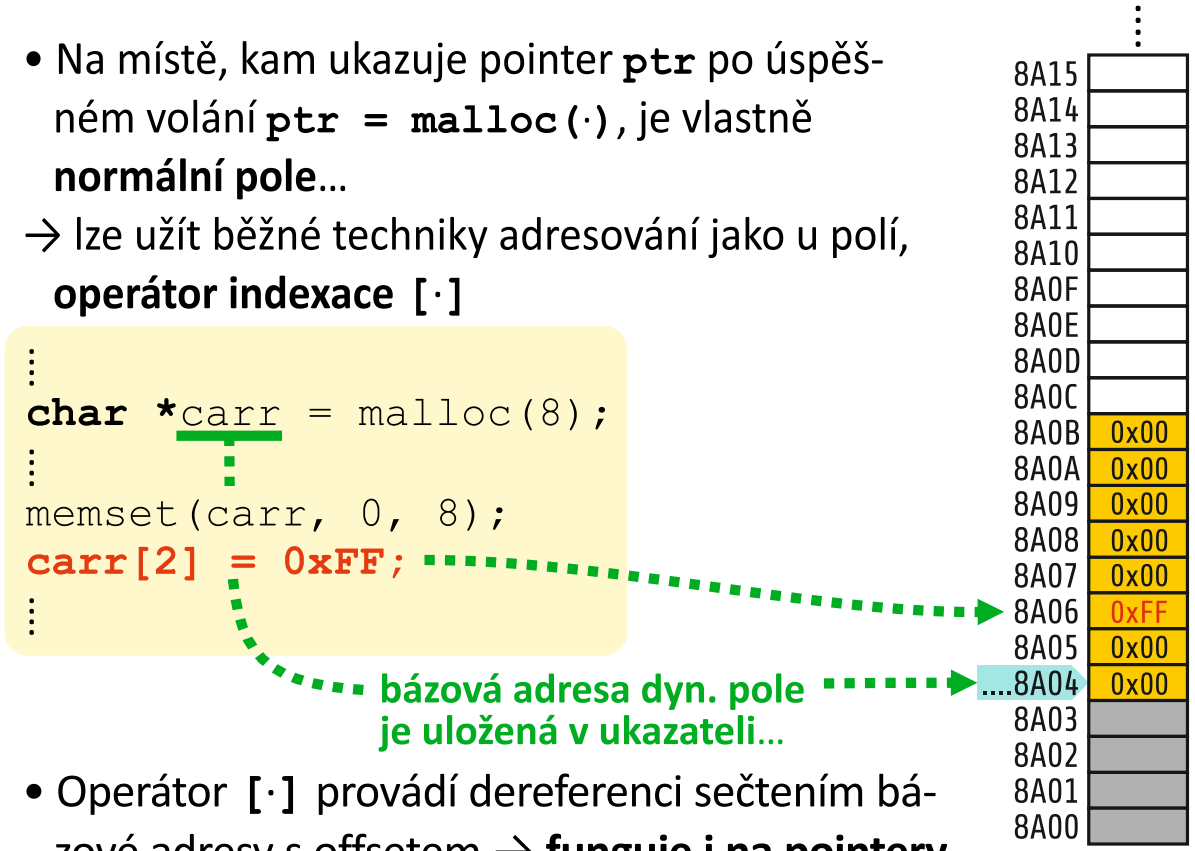
- Před „použitím“ je také rozumné pointer otestovat na přítomnost ne-**NULL** hodnoty.

Přístup k alokované paměti
Adresování paměti indexací ukazatele

- Na místě, kam ukazuje pointer **`ptr`** po úspěšném volání **`ptr = malloc(·)`**, je vlastně **normální pole**…

→ lze užít běžné techniky adresování jako u polí, **operátor indexace `[·]`**

```
⋮
char *carr = malloc(8);
⋮
memset(carr, 0, 8);
carr[2] = 0xFF;
⋮
```

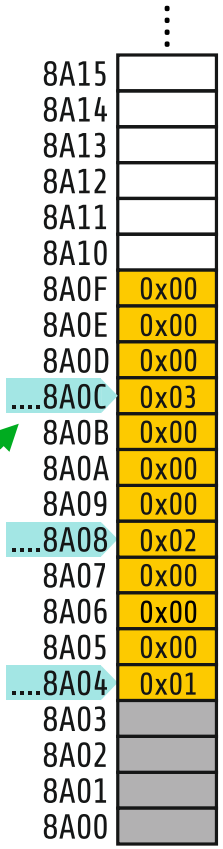


- Operátor **`[·]`** provádí dereferenci sečtením bázové adresy s offsetem → **funguje i na pointery**.

Přístup k alokované paměti
Ekvivalence adresování

- I staticky deklarované pole (viz dříve) je vnitřně realizováno jako paměťová oblast + ukazatel, ve kterém je uložena bázová adresa (ref. proměnná pole) → **pole i pointer lze adresovat (indexovat) úplně stejně**:

```
int *pa = NULL;
pa = (int *) malloc(3 * sizeof(int));
⋮
pa[0] ≡ *pa = 1;
pa[1] ≡ *(pa + 1) = 2;
pa[2] ≡ *(pa + 2) = 3;
```



- Uvedení správného bázového typu ukazatele je extrémně důležité → velikost bázového typu ovlivňuje přepočet indexu na offset.
  (+ 1 znamená + 1 int, nikoliv + 1 byte)

Přístup k alokované paměti
Pole (dyn. i stat.) jako argument funkce

- Referenční proměnná pole je vždy ukazatel na jeho počátek → předá-li se funkci pole, předá se vlastně bázová adresa – **nic jiného!** (nepředává se „objekt třídy pole“ jako v Javě)

```
double darr[10] = {0};
⋮
double max(double a[10]) {
   … /* vyhledání maxima v poli */
}
⋮
max(darr);
```

volání definované funkce `max(·)`

```
double max(double *a)
```

Nemůže fungovat – „uvnitř“ funkce nelze zjistit velikost předaného pole…

- Velikost pole je „uvnitř“ funkce nezjistitelná i tehdy, kdy je uvedena v prototypu (↑) – uvedení velikosti pole v prototypu nemá na mechanismus předávání argumentu žádný vliv… je to **vždy jen ukazatel s hodnotou bázové adresy**.

Přístup k alokované paměti
Pole (dyn. i stat.) jako argument funkce

```
double darr[10] = {0};
⋮
double max(double a[10]) {
   … = sizeof(a);
   … = sizeof(*a);
   …
}
```

velikost ukazatele na double (tj. 32/64 bitů)

velikost prvku pole, čili double (tj. 64 bitů)

- Jediný způsob, jak tento problém vyřešit, je **předávat fcím kromě referenční proměnné pole** (pole či ukazatel) také **další argument**, kterým je **velikost daného pole**:

```
double max(double a[], int alen) {
   …
}
```

(vypadá to neprakticky, ale má to řadu výhod, např. u technik typu Divide & Conquer*)

Přístup k alokované paměti
Pole (dyn. i stat.) jako argument funkce

```
#define SIZE 10

double max(double arr[], int size) {
  double *a_max = arr, *tmp;
  for (tmp = arr + 1; tmp < arr + size; tmp++)
    if (*tmp > *a_max) a_max = tmp;
  return *a_max;
}

void main() {
   double darr[SIZE];
   …
   printf("%lf\n", max(darr, SIZE));
}
```

*) **Výhodný trik**:

```
max(f + 3, 5);
max(&darr[3], 5);
```

- Díky nutnosti předávat bázovou adresu a velikost pole pak funkce může přirozeně pracovat jen s částí pole (bez úprav).

Pomocné funkce pro práci s pamětí
z knihovny funkcí pro znakové řetězce

- Knihovna pro práci se znakovými řetězci **string** nabízí několik pomocných funkcí usnadňujících zacházení s pamětí (protože alokovaný blok paměti je vlastně velmi „podobný“ řetězci znaků) → připojit hlavičkový soubor příkazem preprocesoru: `#include <string.h>`

- Tyto funkce dovolují zejména **nastavovat** (nulovat) **obsah** paměťového bloku, **porovnávat obsah** dvou bloků a **kopírovat obsah** jednoho bloku do druhého, tj. jedná se o operace, které lze snadno naprogramovat pomocí `for`-cyklu a práce s ukazateli.

- Smyslem ↑ je ulehčení programátorova života a efektivní implementace (řetězcové instrukce `REP MOVSB/W/D`).

Pomocné funkce pro práci s pamětí
Vynulování (nastavení hodnoty) bloku

```
void *memset(void *ptr, int c, size_t n);
```

- Funkce nastavuje všechny **byty** alokovaného bloku na hodnotu danou **`int c`** → **POZOR:** postupuje blokem po bytech a vkládá pouze nejméně významný byte z integeru (znak).

```
⋮
int *arr = (int *) malloc(SIZE * sizeof(int));
if (arr) memset(arr, 0, SIZE * sizeof(int));
⋮
```

- Tzn. pro nastavení na hodnotu 0 funguje výborně, pro hodnoty přes hranici 1 byte **NE**:

```
memset(arr, 0x01ABCDEF, SIZE);
for (i = 0; i < SIZE; i++)
  printf("0x%X\n", arr[i]);
```

```
0xEFEFEFEF
0xEFEFEFEF
0xEFEFEFEF
0xEFEFEFEF
0xEFEFEFEF
0x676F7250
0x206D6172
0x656C6946
0x78282073
0x5C293638
0x6D6D6F43
```

Pomocné funkce pro práci s pamětí
Vyhledávání znaku v bloku

```
void *memchr(const void *ptr, int c, size_t n);
```

- Funkce vyhledává znak (tj. 1 byte) v alokovaného bloku, na jehož bázi ukazuje netypový ukazatel **`ptr`**.
- Hledaný znak je předán v parametru **`int c`**, ale funkce prohledává blok po bytech (**`int`** je vnitřně přetypován na **`unsigned char`**).
- Velikost prohledávaného bloku paměti v bytech se předává v parametru **`n`**.
- Vrací ukazatel na první výskyt hledaného znaku nebo **`NULL`**.

```
char str[] = "Example string";
char *p;
if (p = (char *) memchr(str, 'p', strlen(str)))
  printf("'p' is at pos %d.\n", p - str + 1);
else
  printf("'p' not found.\n");
```

Pomocné funkce pro práci s pamětí
Porovnávání obsahu dvou bloků

```
int memcmp(const void *ptr1,
           const void *ptr2, size_t n);
```

- Funkce porovnává po bytech obsah bloků, na jejichž báze ukazují netypové ukazatele **`ptr1`** a **`ptr2`**.
- Počet porovnávaných bytů (počítáno od báze) se předává v parametru **`n`** (zda tento počet představuje stále ještě index uvnitř obou bloků, je na zodpovědnosti programátora).
- **Vrací** (poněkud nelogicky) **hodnotu 0, pokud jsou bloky shodné**.
- Vrací -1, pokud je první blok (**`ptr1`**) lexikograficky menší, tj. stál by ve slovníku dříve, než druhý blok (**`ptr2`**).
- Vrací +1, pokud je první blok (**`ptr1`**) lexikograficky větší, tj. stál by ve slovníku dále, než druhý blok (**`ptr2`**).

Pomocné funkce pro práci s pamětí
Kopírování obsahu bloku

```
void *memcpy(void *dest, const void *src,
             size_t n);
```

- Funkce kopíruje obsah celkem **`n`** bytů od bázové adresy dané netypovým ukazatelem **`src`** do bloku, jehož bázová adresa je daná netypovým ukazatelem **`dest`**.
- Co (jaký datový typ obsahu) se na daných adresách nachází, není podstatné – funkce zajišťuje rychlé binární kopírování (využívá řetězcové instrukce **`REP MOVSB/W/D`**).
- Paměťové bloky by se neměly překrývat (funkce tento stav nijak neošetřuje) → pokud se překrývají, nemusí výsledný stav odpovídat záměru programátora.
- Počet kopírovaných bytů je na zodpovědnosti programátora, funkce nedokáže zjistit (natož zabránit) překročení hranice alokovaného bloku.

Pomocné funkce pro práci s pamětí
Kopírování obsahu bloku – bezpečná varianta

```
void *memmove(void *dest, const void *src,
              size_t n);
```

- Chová se stejně jako **`memcpy(·)`**, ale paměťové bloky se v tomto případě **mohou překrývat**.
- Funkce používá pomocný buffer, aby zabránila případnému přepsání zdrojových dat v případě překryvu bloků.
- Vrací bázovou adresu cílového bloku.

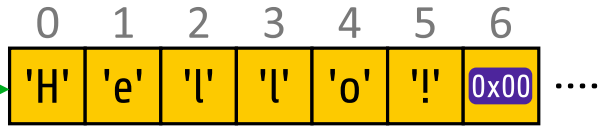
```
char str[] = "memmove can be very useful.....";
memmove(str + 20, str + 15, 11);
puts(str);
```

```
memmove can be very very useful.
```

Řetězce znaků (*strings*)
Práce s datovým typem pro uložení textu

- Jazyk ANSI C používá jako proměnné pro ukládání textu tzv. ***null-terminated strings*** – jednorozměrné pole znaků ukončené znakem s hodnotou 0 (`'\000'`).

```
char str[] = "Hello!";
```

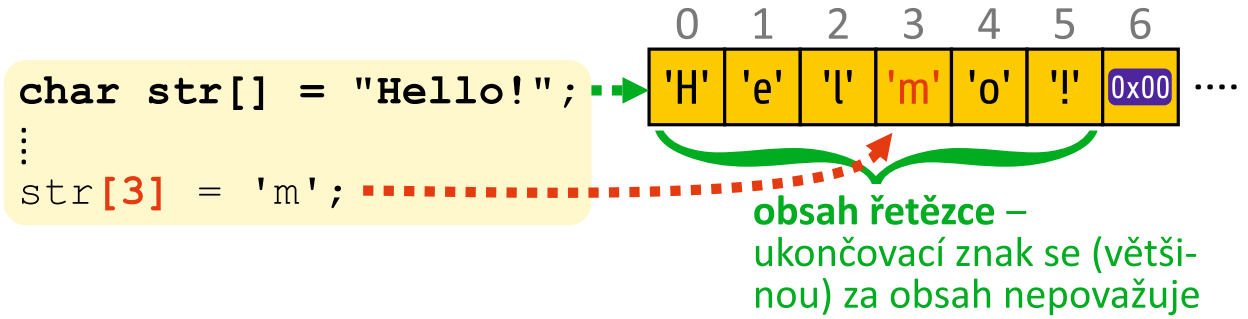


- Funkce a datové typy pro práci se znakovými řetězci jsou v knihovně **string** → připojit hlavičkový soubor příkazem preprocesoru: `#include <string.h>`
- V jazyce C **neexistuje** základní **datový typ znakový řetězec** → řetězec se vždy deklaruje jako pole znaků.

```
char str1[10];
char str2[10] = {0};
char str2[10] = "Hello!";
char str3[] = "Hello, world!";
```

ukončovací znak doplňuje překladač (u literálu)

Řetězce znaků (*strings*)
Uložení řetězců v paměti



- K jednotlivým znakům řetězce se přistupuje indexováním.
- **Prázdný řetězec** `char str[] = "";` obsahuje jeden znak, a to ukončovací znak `'\000'`.
- Prázdný řetězec s uvedenou velikostí `char str[10] = "";` má velikost 10, ale obsahuje jen jeden znak (ukončovací znak `'\000'` na pozici 0, zbytek je nezinicializovaný).
- **Neexistující řetězec** `char *str = NULL;` je něco zcela jiného → je to „latentní“ řetězec, který má referenční proměnnou, ale jeho obsah zatím neexistuje (nebyl vytvořen).

Deklarace řetězců znaků
Dynamická a různé statické deklarace

- Kromě rozdílné deklarace jsou staticky a dynamicky vytvořené řetězce prakticky rovnocenné (jsou to pole!).

```
char str_stat[10];
⋮
str_stat = "Hello!";
```

statická deklarace řetězce

NELZE: řetězcový literál se přiřazuje do ukazatele!

```
char *str_dyn = NULL;
str_dyn = (char *) malloc(10);
str_dyn = "Hello!";
strcpy(str_dyn, "Hello!");
```

dynamická deklarace

```
char *str = "Hello, world!";
⋮
str[7] = (char) 'K';
printf("%s\n", str);
```

řetězcová konstanta

syntakticky v pořádku, ale při spuštění → **RTE!**

Práce s řetězci znaků
Přístup k jednotlivým znakům řetězce

- **Řetězec = pole**, tj. jednotlivé znaky jsou prostě **`char`**y na indexech v příslušném poli (mechanismus stejný pro statické i dynamické řetězce).

```
char str[10];
int i;

for (i = 0; i < 10; i++) str[i] = '*';
printf("%s\n", str);
```

ČASTÁ CHYBA

POZOR: Když programátor nevloží na konec řetězce ukončovací znak, není to syntaktická chyba, ale **`printf(·)`** pak tiskne vše, dokud nenarazí v paměti na 0 (nebo nezpůsobí RTE/AVF).

```
⋮
for (i = 0; i < 9; i++) str[i] = '*';
str[9] = '\000';
printf("%s\n", str);
```

Práce s řetězci znaků
Pozor na rozdíl mezi znakem a řetězcem

- **`"X"`** je **řetězcová konstanta** (jednoznakový řetězec)
  – má velikost **2 chary** (**`'X'`** a **`'\000'`**).
- **`'X'`** je **znaková konstanta**
  – je typu **`int`**, má tedy velikost 32/64 bitů.

POZOR: Nezaměňovat řetězcovou a znakovou konstantu:

```
printf("Enter filename: ");
scanf("%s", filename);
f = fopen(filename, 'r');
```

CHYBA! Ale může to (kupodivu) fungovat

– záleží na **endianu CPU**, jak bude uložena v paměti znaková konstanta:

Funkce pro práci se znakovými řetězci
Přehled použitelných knihovních funkcí

- **Jazyk C nemá žádné syntaktické konstrukce** pro práci se znakovými řetězci! → veškeré manipulace s řetězci se provádí voláním funkcí z knihovny **`string`**, pokud není potřebná funkce k dispozici, musí si jí programátor napsat.

- **Podpora znaků národních abeced v ANSI C prakticky neexistuje** – pozdější normy jazyka C se to snaží napravit např. zavedením typu **`wchar_t`**, ale nepříliš úspěšně
  → je-li nutné pracovat s národními abecedami, je třeba **použít vhodnou externí knihovnu**/framework (utf8proc, MicroUTF-8, FOX Toolkit, GTK2, Qt, Boost, JUCE, apod.).

  bohužel jen pro C++

- Některé funkce z knihovny **`string`** mají tradičně dost nepochopitelná jména (viz dále).

Funkce pro práci se znakovými řetězci
Zjištění délky řetězce

```
size_t strlen(const char *str);
```

- Funkce zjišťuje délku řetězce (ukončovací znak `'\000'` se do délky nezapočítává).
- **Prázdný řetězec** má ukončovací znak na první pozici v poli (index 0) a jeho délka je tedy 0 (i když velikost je 1 byte).
- Funkce se řídí ukončovacím znakem → lze jí použít pouze na znakové řetězce, nikoliv např. na pole **`int`**ů, apod.

Není-li řetězec správně zformován, tj. např. chybí-li ukončovací znak, funkce buď způsobí RTE nebo vrátí počet znaků od báze řetězce k nejbližšímu byte s hodnotou 0 v paměti!

```
char str[STRLEN]; int i;
for (i = 0; i < STRLEN; i++) str[i] = '*';
printf("%d\n", strlen(str));
```

1F 00 00 00

zarazí se zde

Funkce pro práci se znakovými řetězci
Spojování řetězců

```
char *strcat(char *dest, const char *src);
```

- Funkce připojuje řetězec **`src`** na konec řetězce **`dest`**, vrací ukazatel na začátek spojeného řetězce.
- Znaky ze **`src`** se kopírují, dokud funkce nenarazí na ukončovací znak; ten se zkopíruje **také**.
- **Poměrně nebezpečná funkce:** Nijak netestuje (ani to nelze), zda se spojený řetězec „vejde" do oblasti paměti, na kterou ukazuje **`dest`**. Za dostatečnou velikost cílové oblasti zodpovídá programátor.

```
char *strncat(char *dest, const char *src,
              size_t n);
```

- **Bezpečná varianta**, kopíruje nejvýše **`n`** znaků z řetězce **`src`** a **přidá ukončovací znak** (tj. vlastně **`n + 1`**, není-li **`n`**-tým znakem právě ukončovací znak).

Funkce pro práci se znakovými řetězci
Kopírování řetězců (duplikace obsahu)

```
char *strcpy(char *dest, const char *src);
```

- Funkce kopíruje obsah řetězce **src** do řetězce **dest**; původní obsah řetězce **dest** je přepsán. Vrací ukazatel na **dest**.
- Kopíruje celý obsah **src** včetně ukončovacího znaku.
- **Také nebezpečná funkce:** Netestuje, zda je oblast **dest** dost velká pro řetězec **src** (zodpovídá programátor).

```
char *strncpy(char *dest, const char *src,
              size_t n);
```

- **Bezpečná varianta**, kopíruje do **dest** přesně **n** znaků; je-li v **src** méně než **n** znaků, doplní do počtu **n** znakem **'\000'**.
- Je-li znaků ≥ **n**, kopíruje se právě **n** → **ukončovací znak není ošetřen** (může být zkopírován, byl-li na **src[n - 1]**, jinak ho musí doplnit programátor).
- **Pokud se oblasti překrývají, chování fce není definované.**

Funkce pro práci se znakovými řetězci
Kopírování řetězců (použití)

- Pomocí **`strcpy(·)`** lze např. realizovat spojení řetězců:

```
char *strcat(char *dest, const char *src) {
  char *tmp = dest + strlen(dest);
  strcpy(tmp, src);
  return dest;
}
```

`tmp` teď ukazuje na ukončovací znak (tj. za konec) řetezce `dest`

POZOR: V jazyce ANSI C není možné řetězce přiřazovat!
Operátor '=' není přetížen (to C neumí) pro řetězce (není datový typ), tj. přiřazuje ukazatel → jiné chování:

```
char str1[80] = {0};
char str2[] = "Something very important";

str1 = str2;
strcpy(str1, str2);
```

`str1 = str2;` ztrácíme referenci na původní řetězec `str1`

Funkce pro práci se znakovými řetězci
Lexikografické porovnání řetězců

```
int strcmp(const char *s1, const char *s2);
```

- Funkce porovnává (lexikograficky) obsah řetězců **s1** a **s2**.
- **Vrací 0, pokud jsou shodné**.
- Vrací -1, pokud by **s1** stál ve slovníku před **s2** (tj. je menší).
- Vrací +1, pokud by **s1** stál ve slovníku za **s2** (tj. je větší).

```
char *strncmp(const char *s1, const char *s2,
              size_t n);
```

- Pracuje shodně, ale porovnává maximálně **n** znaků z obou řetězců (tj. delší řetězec ořízne).
- Pokud je **n** větší než délka delšího z řetězců, chová se úplně stejně jako **strcmp(·)** a řídí se ukončovacími znaky.

POZOR: V jazyce ANSI C není možné řetězce porovnávat operátorem '==‘! (stejné důvody i následky, viz předchozí)

Funkce pro práci se znakovými řetězci
Hledání znaku v řetězci

```
char *strchr(const char *str, int c);
```

- Funkce hledá **první výskyt** znaku **`c`** v řetězci **`str`**.
- Pokud znak **`c`** najde, vrací ukazatel na místo jeho prvního výskytu; pokud nenajde, vrací **`NULL`**.
- Ukončovací znak se považuje za součást řetězce, proto jeho hledání bude vždy úspěšné (vrátí ukazatel na něj, tj. za poslední platný znak prohledávaného řetězce).

```
char *strrchr(const char *str, int c);
```

- Hledá poslední výskyt znaku (resp. první výskyt zprava).

Funkce pro práci se znakovými řetězci
Hledání znaku v řetězci (použití)

- Pomocí **`strchr(·)`** lze např. implementovat funkci pro zjištění počtu výskytů znaku v řetězci:

```
int chrcnt(const char *str, int c) {
  int n = 0;

  while (str) {
    str = strchr(str, c);
    if (str) n++, str++;
  }

  return n;
}
```

parametr **`str`** se (lokálně) upravuje tak, aby ukazoval na tu část řetězce, která následuje za naposledy nalezeným výskytem znaku…

Funkce pro práci se znakovými řetězci
Hledání podřetězce v řetězci

```
char *strstr(const char *haystack,
             const char *needle);
```

- Funkce hledá **první výskyt** celého (pod)řetězce **needle** (kromě ukončovacího znaku) v řetězci **haystack**.
- Vrací ukazatel na začátek prvního výskytu; pokud se řetězec **needle** v řetězci **haystack** nenachází, vrací **NULL**.
- Je case-sensitive (zkoumá shodu ASCII hodnot znaků).

```
int main() {
  char str[] = "This is a simple string";
  char *pch = strstr(str, "simple");
  strncpy(pch, "sample", 6);
  printf("%s\n", str);
  return 0;
}
```

Funkce pro práci se znakovými řetězci
Filtrování znaků v řetězci

```
size_t strspn(const char *s, const char *set);
```

- Funkce hledá v řetězci **s** první výskyt takového znaku, který **není obsažen** v řetězci **set**, přičemž přeskakuje znaky, které v řetězci **set** jsou → vrací **délku nejdelšího počátečního úseku** řetězce **s**, který obsahuje pouze znaky ze **set**.
- Pokud se všechny znaky řetězce **s** nacházejí i v řetězci **set**, pak vrátí délku řetězce **s** (bez ukončovacího znaku).
- Parametr **set** představuje **množinu „povolených" znaků**, tj. jejich pořadí či případné opakování v řetězci **set** nehraje žádnou roli.

Funkce pro práci se znakovými řetězci
Filtrování znaků v řetězci

```
size_t strcspn(const char *s, const char *set);
char *strpbrk(const char *s, const char *set);
```

- Funkce **`strcspn(·)`** funguje opačně než **`strspn(·)`** a hledá v řetězci **`s`** první výskyt takového znaku, který **je obsažen** v řetězci **`set`**, přičemž přeskakuje znaky, které v řetězci **`set`** nejsou → vrací **délku nejdelšího počátečního úseku** řetězce **`s`**, který neobsahuje znaky ze **`set`**.

- Funkce **`strpbrk(·)`** pracuje stejně, ale vrací ukazatel na první nalezený znak z množiny **`set`** v řetězci **`s`**; pokud žádný takový nenajde, vrací **`NULL`**.
- Parametr **`set`** představuje **množinu „zakázaných" znaků**, tj. jejich pořadí či případné opakování v řetězci **`set`** nehraje žádnou roli.

Funkce pro práci se znakovými řetězci
Tokenizace (kriteriální rozdělení) řetězce

```
char *strtok(char *str, const char *delim);
```

- Funkce rozdělí řetězec **`str`** na řadu podřetězců oddělených kterýmkoliv znakem z množiny oddělovačů **`delim`**.
- Pokud se podaří najít v řetězci **`str`** oddělovací znak z množiny **`delim`**, vrací funkce ukazatel na podřetězec, který za ním bezprostředně následuje (tj. další token); pokud oddělovač nalezen není, vrací **`NULL`**.
- Aby bylo možné rozdělit vstupní řetězec na více podřetězců než jeden, musí být funkce **volána opakovaně** (viz dále).
- Při prvním volání se předá řetězec **`str`**, který se má rozdělit; při dalších voláních se jako první parametr předává **`NULL`**, čímž se funkci dává najevo, že má pokračovat v dělení řetězce dodaného při dřívějším (prvním) volání.

Funkce pro práci se znakovými řetězci
Tokenizace řetězce (použití)

```
#include <stdio.h>
#include <string.h>

int main() {
  char str[] = "- This, a sample string.";
  char *pch;

  pch = strtok(str, "␣,.-");
  while (pch != NULL) {
    printf("%s\n", pch);
    pch = strtok(NULL, "␣,.-");
  }

  return 0;
}
```

```
This
a
sample
string
```

Funkce pro práci se znakovými řetězci
Převod řetězce na číslo

```
double strtod(const char *str, char **ptr);
long strtol(const char *str, char **ptr,
            int base);
unsigned long strtoul(const char *str,
                      char **ptr, int base);
```

- Funkce převádějí řetězec **`str`** na číslo; ukazatel **`ptr`** nastaví funkce na začátek té části řetězce, kterou již nelze na číslo převést (např. jednotky za číselnou hodnotou, viz dále).
- Začíná-li řetězec **`str`** bílými znaky (ve smyslu **`isspace(·)`**), funkce je přeskočí.
- Parametr **`base`** udává očekávaný základ číselné soustavy převáděného čísla; pokud má hodnotu 0, pokusí se funkce sama určit soustavu z tvaru čísla (podle pravidel pro zápis číselných konstant).
- Tyto funkce poskytují lepší možnost řídit převod řetězce na číslo než např. **`sscanf(·)`** nebo **`atoi(·)`**/**`atof(·)`**.

Funkce pro práci se znakovými řetězci
Převod řetězce na číslo (použití)

```
double x;
long i;
char inp[] = "-12.59e-1 deg";
char *rest;

x = strtod(inp, &rest);
i = strtol("0xFA", NULL, 0);
i = strtol("FA", NULL, 16);
```

rest ukazuje na " deg"

x == -1.259

základ je třeba uvést

- Je-li **base** rovno 0, očekává se osmičkové, desítkové nebo šestnáctkové číslo (základ je odvozen od formátu konstanty).
- Je-li **base** mezi 2 a 36, musí se řetězec skládat z nenulové posloupnosti písmen a číslic, které reprezentují číslo při daném základu soustavy (‘a’ až ‘z’ nebo ‘A’ až ‘Z’ značí hodnoty 10 až 36).

Funkce pro práci se znakovými řetězci
Převod řetězce na číslo (deprecated!)

```
double atof(const char *str);
int atoi(const char *str);
long atol(const char *str);
```

- Funkce převádějí řetězec na číslo – **pokud řetězec na číslo nelze převést, není jejich chování definováno** (v lepším případě vrací 0 → nelze říct, zda v převáděný řetězec měl tvar `"0"` nebo se převod nezdařil).
- V ANSI C jsou tyto funkce jen kvůli zpětné kompatibilitě s K&R C překladači (staré UNIXy).
- Norma ANSI C doporučuje používat **`strtod(·)`**, **`strtol(·)`** a **`strtoul(·)`**, které umožňují převod lépe řídit (lze zjistit, že se převod nepovedl).
- V ANSI C jsou tyto funkce v knihovně **`stdlib`**, nikoliv v knihovně **`string`**.

Duff's Device – rychlé kopírování řetězců

Šílené propojení dvou jazykových konstrukcí

```
void duffcpy(char *to, char *from, int count) {
  int n = (count + 7) / 8;

  switch (count % 8) {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
    case 6:      *to++ = *from++;
    case 5:      *to++ = *from++;
    case 4:      *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
        } while (--n > 0);
  }
}
```

NEPOUŽÍVAT NORMÁLNĚ: EXTRÉMNÍ PROGRAMO-VÁNÍ!

větve `case` v konstrukci `switch` jsou vlastně jen návěští (s ord. hodnotou místo pojmenování), tzv. lze je použít pro skok „dovnitř” cyklu (**`break`** samozřejmě chybí záměrně).