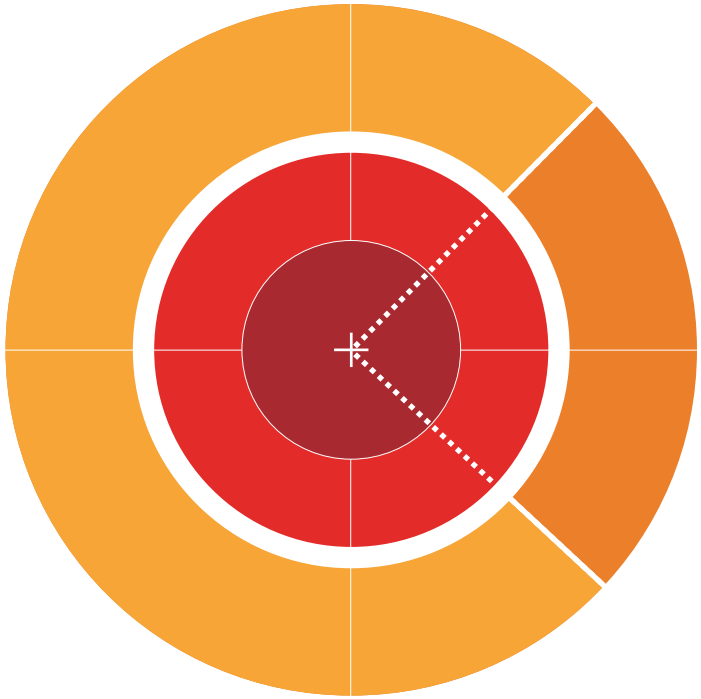
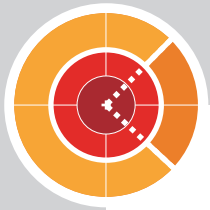






# Správa paměti

- funkce pro správu paměti jsou uloženy v knihovně **`stdlib`** (**St**an**d**ard **Lib**rary) => připojit pomocí příkazu preprocesoru **`#include <stdlib.h>`**
  (v ANSI C, v některých implementacích není třeba připojovat nebo jsou definovány jinde)
- **v C je paměť zcela v rukách programátora**
- přidělenou paměť musím **uvolnit** (snaží se provádět OS při ukončení programu, ale ne vždy to je možné) => jakmile program přidělenou paměť nepotřebuje, měl by ji "vrátit"
- díky operacím s ukazateli je programátorovi k dispozici celý adresový prostor => nebezpečí konfliktů s jinými procesy
- **překladač** C **nekontroluje** platnost (obsah) ukazatelů, tj. je možné ukládat data do paměti, která patří jinému procesu nebo vůbec neexistuje
- při práci s pamětí dochází k 90% všech chyb v C





# Základní operace s pamětí

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

1

2

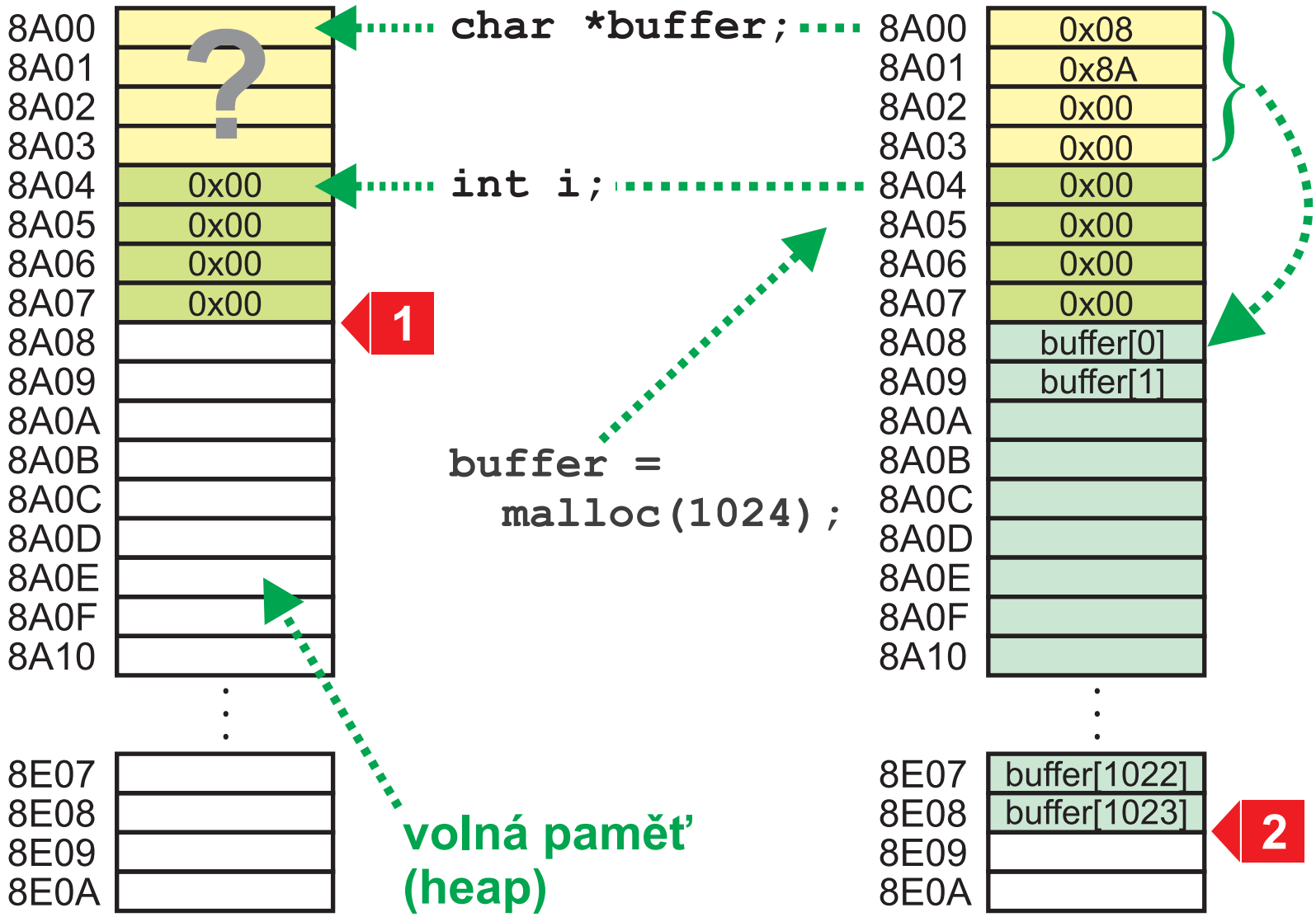
**`malloc(n)`** získá od OS **`n`** byte paměti a vrátí ukazatel na první prvek této oblasti

**`free(ptr)`** vrátí úsek paměti, na který ukazuje **`ptr`** zpět systému

**(1)** existuje statická proměnná **`buffer`** - ukazatel na znak
**(2)** vznikla ***dynamická proměnná*** - pole 1024 znaků

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

# Co se děje v paměti...







# Alokace paměti

```
void *malloc(size_t size);
```

**malloc()** vrací netypový ukazatel, takže je možné jej přetypovat

```
typedef struct s_node { int key;
  s_node *left, *right; } Node;
...
Node *node;
...
node = malloc(sizeof(Node));
if (node != NULL) {
   node->key = 5;
   node->left = NULL;
   node->right = NULL;
}
else fprintf(stderr, "Out of memory\n");
```

v případě neuspěchu vrací **malloc()** NULL



# Alokace paměti

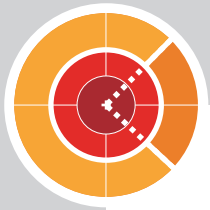
```
Node *p_node;
p_node->key = 5;
...
p_node = malloc(sizeof(Node));
p_node->key = 5;
...
```

**nelze** - v tuto chvíli existuje pouze ukazatel na Node, nikoliv struktura samotná!

teprve nyní vznikla tzv. ***instance*** struktury **`Node`** = **dynamická proměnná** jejíž struktura je dána definicí **`Node`** (ukazuje na ni, tj. zpřístupňuje ji ukazatel **`p_node`**)

- lze také alokovat souvislou oblast **`count`** prvků, každý má velikost **`size`** bytů (užívat **`sizeof()`**):

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



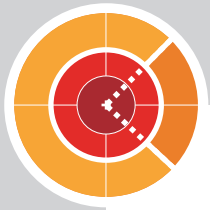


# Uvolňování (dealokace) paměti

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

- uvolňuje paměť alokovanou funkcemi **`malloc()`**, **`calloc()`** a **`realloc()`**
- parametr **`ptr`** je ukazatel vrácený alokační funkcí - **musí být** (až na případné přetypování) **stejný**, tj. nelze provádět ukazatelovou aritmetiku
- poté, co program paměť uvolní, přestává dynamická proměnná fakticky existovat => **nelze k ní přistupovat**
- nepotřebnou paměť je třeba v průběhu výpočtu uvolňovat, jinak dojde (důležité zejména v cyklech, rekurzi, atp.)
- neuvolněnou paměť se pokusí uvolnit OS po skončení programu (to se ale nemusí vždy povést)

Je vhodné si udělat vlastní alokační a dealokační funkce, které budou měnit hodnotu globální proměnné, představující množství alokované paměti. Na konci výpočtu tam musí být 0.







# Realokace paměti

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

- **`ptr`** je ukazatel na oblast paměti, přidělenou standardními alokačními funkcemi
- **`realloc()` změní velikost** této oblasti na **`size`**, aniž by porušila obsah (je-li to nutné, překopíruje se obsah do nové souvislé oblasti paměti)
- vrácen je ukazatel na (novou) oblast paměti
- nemůže-li být požadavek splněn, vrací fce NULL a obsah původní oblasti zůstává nedotčen
- je-li **`ptr == NULL`**, chová se stejně jako **`malloc()`**
- je-li **`ptr != NULL`** a **`size == 0`**,chová se jako **`free()`**
- **nový prostor se přidává za konec původních dat**
- je-li nová velikost menší než původní, odřízne se konec
- nový prostor není nijak zinicializovaný, je třeba předpokládat náhodná data



# Realokace paměti

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

int AddSample(double smp) {
    if (sample_cnt == sample_lim) {
        sample_lim += SAMPLE_INCR;
        samples = (double *) realloc(
            (char *) samples,
            sample_lim * sizeof(double));
        if (samples == NULL) {
            fprintf(stderr, "Out of memory.\n");
            return 1;
        }
    }
    samples[sample_cnt++] = smp;
    return sample_cnt;
}
```

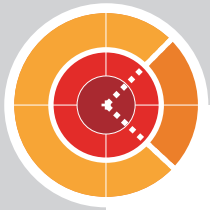




# Funkce pro ladění programů s intenzivní správou paměti

```
/* globální proměnná s počtem alokací */
unsigned int alloc_cnt = 0;

void *getmem(size_t size) {
    void *ptr;

    ptr = malloc(size);
    if (ptr != NULL) alloc_cnt++;

    return ptr;
}

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

z bezpečnostních důvodů bezodkladně vynulovat





## Paměť a pole - dynamická pole

- vnitřně je pole a pointer totéž, liší se pouze deklarací a mechanismem alokace

```
int a_var[20];
int *p_var;
```

**statické pole**, v okamžiku spuštění programu již existuje v paměti oblast 20-ti integerů, na jejíž 1. prvek ukazuje **`a_var`**, tj. **alokaci zajišťuje překladač**

**ukazatel na integer** - může (ale nemusí) ukazovat na jediný integer, ale také na 1. prvek souvislého úseku paměti, který se pak bude interpretovat, jako by byl naplněn integery; v okamžiku spuštění programu ale žádná oblast v paměti neexistuje, tj. **alokaci zajišťuje programátor**

- adresa libovolného prvku lze (v obou případech) vypočítat

**`&x[i] =`** *<bázová adresa>* **`x + i * sizeof(`***<typ>* **`x)`**



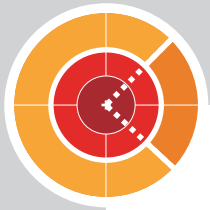
## Ekvivalence adresování

- pointer a pole lze adresovat (indexovat) úplně stejně

```
int *p_i;
p_i = (int *) malloc(4 * sizeof(int));
...
p_i[0] == *p_i;
p_i[1] == *(p_i + 1);
p_i[2] == *(p_i + 2);
...
```

- adresa (index) pole lze porovnávat s pointerem (zde test, zda **p_i** ukazuje "dovnitř" pole)

```
int i[4], *p_i;
...
if (p_i >= i && p_i < i + 4) {...}
```





## Optimalizace přístupu k prvkům pole

- zaměnitelností pointeru a pole lze obejít např. mapovací funkci pole a tak urychlit výpočet

```
double mat1[5][5], mat2[5][5];
double *pm1, *pm2;

for (pm1 = mat1, pm2 = mat2;
    pm1 < mat1 + 25; ) *pm2++ = *pm1++;
```

- kód je hůře čitelný, ale **rychlejší** než použití dvou smyček přes oba indexy matic
- navíc v tomto případě odpadá nutnost výpočtu mapovací funkce



## Pole uvnitř funkce

- pole se funkci předávají **vždy odkazem** (protože se jedná vlastně o ukazatel) - narozdíl od proměnných, které se předávají hodnotou

```
double darr[10];

double max(double a[10]) {
    ...
}
```

ekvivalentní s `double *a`

- **POZOR!** => uvnitř funkce **nelze zjistit velikost** pole (ani kdyby byla uvedena v prototypu jako v uvedeném příkladu)
- je nutné předávat **dalším parametrem velikost**, protože

`sizeof(a)` = velikost ukazatele na double (4)
`sizeof(*a)` = velikost prvku pole, tj. double (8)

# Pole uvnitř funkce (pokračování)

```
#define SIZE 10

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

int main() {
    double darr[SIZE];
    ...
    printf("%d\n", max(darr, SIZE));
}
```

možný trik -

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

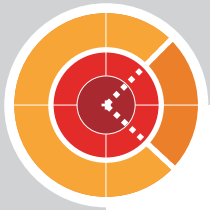




# Znakové řetězce (strings)

- jazyk C používá tzv. ***null-terminated strings*** - jednorozměrné pole znaků s ukončovacím znakem `'\000'`
- funkce pro práci s řetězci jsou v knihovně **`string`**, tj. je třeba připojit pomocí `#include <string.h>`
- za řetězcové konstanty doplňuje ukončovací znak překladač, v případě dynamických (za běhu vzniklých) řetězců to musí udělat programátor
- některé knihovní funkce ukončovací znak **doplňují**, některé ne - poněkud chaos, je třeba se přesvědčit v dokumentaci

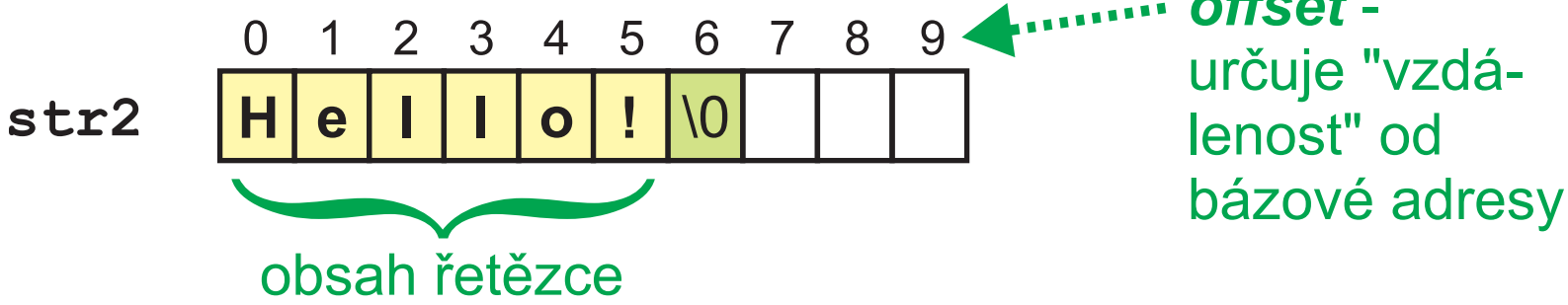
```
char str1[10];
char str2[10] = "Hello!";
char str3[] = "Hello, world!";
```







# Uložení řetězců v paměti




- **prázdný řetězec** neobsahuje žádný znak a je reprezentován ukazatelem na znak `'\000'`
- **POZOR!** - to je něco zcela jiného než nulový ukazatel (NULL), ten neukazuje na vůbec žádný znak
- při přenosu znaků do řetězce se obvykle **nedělá žádná kontrola** velikosti cílového prostoru => je věcí programátora, aby zajistil pro výsledný řetezec včetně ukončovacího znaku dost místa v paměti



# Statické a dynamické řetězce

- kromě rozdílné deklarace prakticky rovnocenné

```
char s_stat[10];

s_stat = "Hello!";
```

statická deklarace

**nelze** - přiřazení řetězce do ukazatele!

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

dynamická deklarace





# Přístup k jednotlivým znakům řetězce

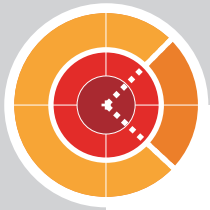
- stejný mechanismus, ať je řetězec statický nebo dynamický

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

for (i = 0; i < 9; i++) s_stat[i] = '*';
s_stat[9] = '\0';
```

přidat na konec ukončovací znak `'\0'`

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

for (i = 0; i < 10; i++)
    s_stat[i] = '*';
```

**POZOR! - častá chyba:** programátor nenechá místo na ukončovací znak







# Rozdíl mezi znakem a řetězcem

- `"x"` je **řetězcová konstanta** délky 2 znaky (`'x'` a `'\0'`)
- `'x'` je **znaková konstanta** typu **int** (**!!!**)

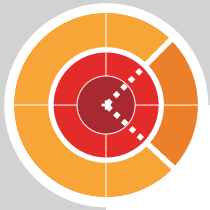
```
printf("Enter filename: ");
scanf("%s", s1);
f = fopen(filename, "r");
```

vyžaduje-li funkce jako parametr řetězec, je třeba předat **řetězec** a **ne znakovou konstantu**

**POZOR!**

**(i) nulový pointer:** `s1 = NULL;`

**(ii) prázdný (nulový) řetězec:** `s1[0] = '\0';`





## Funkce pro práci s řetězci

- **funkce** pro práci se znakovými řetězci **nejsou součástí** jazyka C, ale norma ANSI C předepisuje základní soubor těchto funkcí v knihovně **`string`**

## Zjištění délky řetězce - strlen()

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

- vrací počet znaků, které v řetězci **`s`** leží před ukončovacím znakem
- prázdný řetězec má ukončovací znak na první pozici a jeho délka je tedy 0
- v některých implementacích jazyka C se tato funkce nazývá **`lenstr()`**





## Spojení řetězců - strcat()

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

- připojí obsah řetězce **src** za poslední znak řetězce **dest**
- vrací ukazatel na **dest**
- znaky ze **src** (včetně nulového na konci) se zapisují přes původní nulový znak na konci **dest** a další znaky, které za ním v paměti následují
- znaky se kopírují, dokud funkce nenarazí v **src** na nulový znak
- předpokládá se, že **dest** je dost velký pro spojený řetězec
- **strncat()** omezuje počet připojených znaků ze src na nejvýše **n** (nenarazila-li funkce do té doby na nulový znak, přidá se na konec spojeného řetězce jako n + 1. znak)



## Porovnání řetězců - strcmp()

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

- porovnává lexikograficky obsah řetězců (oba musí být zakončené nulovým znakem)
- vrací 0, jsou-li řetězce shodné
- vrací **zápornou hodnotu**, je-li **`s1`** menší než **`s2`**
- vrací **kladnou hodnotu**, je-li **`s1`** větší než **`s2`**
- **`strncmp()`** pracuje shodně, ale porovnává maximálně **`n`** znaků





## Kopírování řetězců - strcpy()

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

- kopíruje obsah řetězce **src** do řetězce **dest**, přičemž přepíše původní obsah **dest**
- kopíruje **celý obsah** **src** včetně ukončovacího znaku i tehdy, když je **src** delší než **dest**
- vrací ukazatel na **dest**

- **strncpy()** kopíruje do **dest** přesně **n** znaků; je-li v **src** méně než **n** znaků, doplní se do počtu **n** znakem **'\000'**
- je-li znaků více než **n**, zkopíruje se právě **n** => **ukončovací znak není ošetřen** (může být zkopírován, byl-li **src** dlouhý právě **n** - 1 znaků, jinak ho musí doplnit programátor)
- **pokud se oblasti překrývají, není chování definované**





## Ukázka použití strcpy()

```
#include <string.h>

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

je věcí programátora, aby měl na výsledný řetězec dost místa





## Hledání znaku v řetězci - strchr() a strrchr()

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

- obě funkce hledají znak **c** v řetězci s zakončeném **'\000'**
- **strchr()** hledá **první** výskyt znaku - najde-li znak **c**, vrací ukazatel na jeho první výskyt; nenajde-li, vrací **NULL**
- funkce **strrchr()** hledá **poslední** výskyt znaku
- ukončovací znak se považuje za součást řetězce, takže jeho hledání bude vždy úspěšné a vrátí ukazatel za poslední platný znak řetězce (tj. na ukončovací znak)



## Ukázka použití strchr()

```
#include <string.h>

int chrcnt(char *s, int c) {
    int n = 0;

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

    return n;
}
```

parametr **s** se (lokálně) upravuje, aby ukazoval na část řetězce, která následuje za naposledy nalezeným znakem





# Filtrování znaků v řetězci - strspn()

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

- 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** **s**, který obsahuje pouze znaky z řetězce **set**
- pokud se každý znak z **s** vyskytuje v **set**, pak vrátí délku řetězce **s** (nepočítaje v to ukončovací znak)

- parametr **set** představuje množinu povolených znaků, tj. jakýsi filtr, přičemž jejich pořadí ani případné opakování **nemá** žádný význam

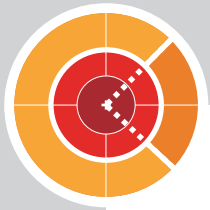




# Filtrování znaků v řetězci - strcspn() a strpbrk()

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

- **strcspn()** hledá v **s** první výskyt znaku, který **je obsažen** v **set** a přeskakuje znaky, které v **set** nejsou
- vrací délku nejdelšího počátečního úseku **s**, který neobsahuje znaky ze **set**
- **strpbrk()** pracuje stejně, ale vrací ukazatel na první nalezený znak, který se vyskytuje v **set** - nenajde-li žádný takový znak, vrací **NULL**







# Vyhledávání podřetězců - strstr() a strtok()

```
char *strstr(const char *str,
    const char *substr);
char *strtok(char *str, const char *set);
```

- **strstr()** zjišťuje **výskyt řetězce substr** v řetězci **str** a vrací ukazatel na začátek prvního výskytu - pokud se **substr** ve **str** nevyskytuje, vrací **NULL**

- **strtok()** **rozděluje str na atomy**, které jsou odděleny znaky z řetězce **set**
- pro každý atom se volá **strtok()** znovu (s případnou změnou hodnoty **set**)
- při prvním volání se předá jako parametr **str**, při dalších voláních se předává **NULL**, čímž se říká, že se má pokračovat ve zpracování původního řetězce od konce předchozího atomu (viz příklad)



# Ukázka použití strtok()

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

int main() {
    char line[LINE_LENGTH];
    char *word;
    while (TRUE) {
        fgets(line, LINE_LENGTH, stdin);
        if (strlen(line) <= 1) break;
        word = strtok(line, " .,?\"\n");
        while (word != NULL) {
            printf("<%s>\n", word);
            word = strtok(NULL, " .,?\"\n");
        }
    }
    return 0;
}
```

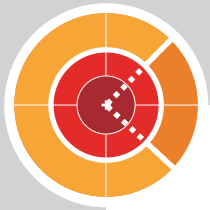




# Převod řetězce na číslo - strtod(), strtol() a strtoul()

```
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);
```

- převádějí řetězec na číslo, **str** ukazuje na řetězec, který se má převést, **ptr** nastaví funkce tak, že ukazuje na první znak **str** následující za převedenou částí řetězce
- začíná-li **str** prázdnými znaky (ve smyslu **isspace()**), pak se takové znaky přeskočí
- parametr **base** udává očekávaný základ číselné soustavy
- tyto funkce poskytují lepší možnost řídit převod řetězce na číslo než např. **sscanf()**





# Převod řetězce na číslo (pokračování)

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

x = strtod(inp, &rest);
```

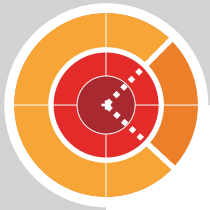
**rest** ukazuje na " **deg**"

**x == -1.259**

```
x = strtol("0xFA", NULL, 0);
x = strtol("FA", NULL, 16);
```

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 číslo skádat z nenulové posloupnosti písmen a číslic, které reprezentují číslo při daném základu ('a' až 'z' nebo 'A' až 'Z' značí hodnoty 10 až 36)





# Převod řetězce na číslo - atof(), atoi() a atol()

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

- převádějí řetězec na číslo - **nedokážou-li řetězec převést, není jejich chování definováno**
- v ANSI C jsou tyto funkce jen kvůli kompatibilitě se systémy typu klasického UNIXu
- ANSI C dává přednost **`strtod()`**, **`strtol()`**, **`strtoul()`**, které umožňují převod lépe řídit
- v ANSI C jsou definovány v knihovně **`stdlib`**, nikoli ve **`string`**