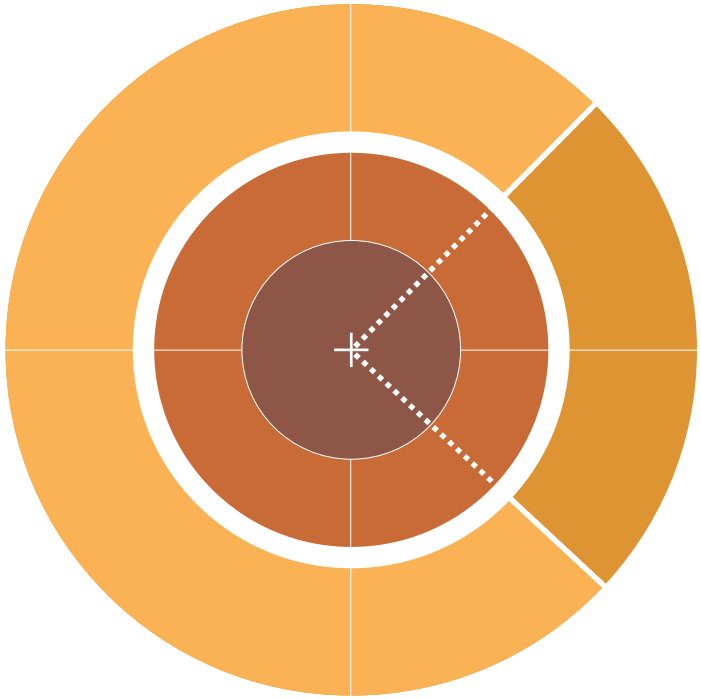
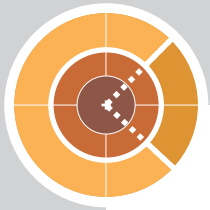
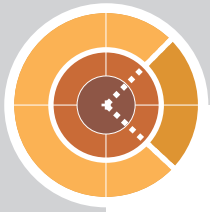




Standardní rozšíření jazyka

- funkce těsně spojené s jazykem C, zvyšují míru přenositelnosti
- řeší problémy vznikající v důsledku odlišného chování jednotlivých OS nebo procesorových architektur
- umožňují komunikaci s OS (identifikace terminálu, příp. uživatele, atp.)
- "vyšší", tj. neelementární funkce, např. vyhledávání metodou bisekce nebo řazení algoritmem QuickSort
- umístěné v různých knihovnách



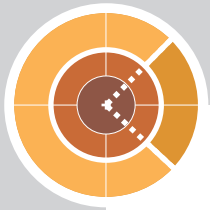
Offset složky struktury - offsetof()

```
#define offsetof(str, item) ...
```

- makro je definované v knihovně **stddef**, tj. připojit pomocí `#include <stddef.h>`
- vrací **offset** složky **item** (adresu od počátku) struktury **str**

```
#include <stddef.h>
#include <stdlib.h>
...
struct s { char buf[10]; int a; } x;
size_t offset;

offset = offsetof(struct s, a);
...
```

offset má hodnotu 10 (0 - 9 zabírá **buf**)



Zpracování chyb - errno, strerror() a perror()

```
extern int errno;
```

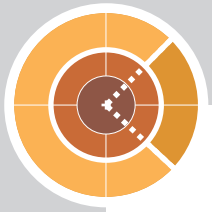
- externí proměnná definovaná v **errno.h**, je-li nulová, znamená to, že při poslední knihovní operaci chyba nenastala
- je-li **nenulová**, došlo k chybě, její kód je v **errno** uložen

```
void perror(const char *msg);
```

- definovaná ve **stdio**; tiskne na **stderr** posloupnost: **msg**, dvojtečka, mezera, zpráva o chybě, jejíž kód je v **errno** a znak nového řádku

```
char strerror(int errcode);
```

- definovaná v **string**; vrací ukazatel na řetězec chybového hlášení, jehož obsah je daný implementací podle **errcode**
- řetězec je statický, tj. příští volání ho přepíše

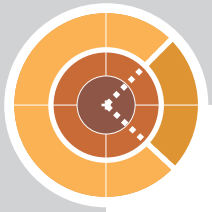


Zpracování chyb (příklad)

```
#include <math.h>
#include <errno.h>
...
errno = 0;
x = sqrt(y);
if (errno) {
   perror("Chyba při odmocňování");
}
...
errno = 0;
x = sqrt(y);
if (errno) {
   printf("Nastala chyba %s\n",
      strerror(errno));
}
...
```

vynulování příznaku chyby

nenulové, tj. nastala chyba

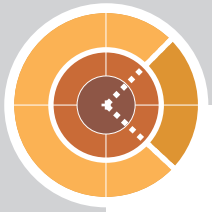


Proměnné prostředí - getenv()

```
#include <stdlib.h>
char *getenv(const char *varname);
```

- parametrem funkce je řetězec, který se interpretuje jako jméno proměnné z operačního prostředí (OS, konzole)
- funkce vrací ukazatel na řetězec, který představuje hodnotu zvolené proměnné prostředí
- pokud proměnná prostředí **varname** neexistuje nebo je prázdná, vrací funkce **NULL**
- vrácený řetězec **nesmí** programátor modifikovat; je statický, další volání ho může přepsat

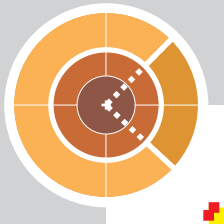
```
char *path;
path = getenv("PATH");
if (path != NULL)
   printf("PATH = %s\n", path);
```



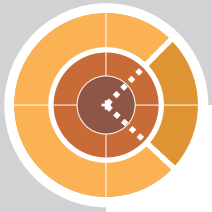
Prohledávání bisekcí - bsearch()

```
#include <stdlib.h>
void *bsearch(const void *key,
    const void *base, size_t count,
    size_t size, int (*compar) (
        const void *k, const void *v));
```

- prohledává pole **count** prvků, na jehož první prvek ukazuje parametr **base**; velikost každého prvku je **size** znaků
- parametr **compar** ukazuje na **porovnávací funkci**, která má dva parametry, netypové pointery na klíč a prvek pole
- porovnávací funkce vrací zápornou, nulovou nebo kladnou hodnotu podle toho, zda je prvek pole menší, stejný nebo větší než klíč
- prvky pole musí být na začátku seřazeny vzestupně
- funkce **bsearch()** vrací ukazatel na prvek pole, který je roven klíči **key** nebo **NULL** v případě neúspěchu



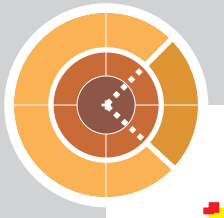

```c
#include <stdlib.h>
#define COUNT 100

struct elem { int key; int data; } Table[COUNT];

int key_compare(const void *k, const void *v) {
    int v1 = ((struct elem *) k)->key;
    int v2 = ((struct elem *) v)->key;
    return (v1 < v2) ? -1 : (v1 > v2) ? 1 : 0;
}

int fetch(int key) {
    struct elem *result, key_elem;
    key_elem.key = key;
    result = (struct elem *) bsearch(
        (void *) key_elem, (void *) Table,
        (size_t) COUNT, sizeof(struct elem),
        key_compare);
    return (result == NULL) ? 0 : result->data;
}
```



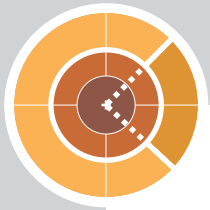
Řazení dat algoritmem QuickSort - qsort()

```
#include <stdlib.h>
void qsort(const void *base, size_t count,
    size_t size, int (*compar) (
        const void *elem1,
        const void *elem2));
```

- řadí pole **count** prvků, na jehož první prvek ukazuje **base**
- velikost každého prvku je **size** znaků
- **compar** je pointer na porovnávací funkci, která má jako parametry dva pointery na prvky pole - vrací **-1**, **0** nebo **1**, pokud je první prvek **menší**, **roven** nebo **větší** než druhý
- po proběhnutí funkce budou prvky seřazené vzestupně




```c
#include <stdlib.h>
#define COUNT 100

struct elem { int key; int data; } Table[COUNT];

int key_compare(const void *k, const void *v) {
    int v1 = ((struct elem *) k)->key;
    int v2 = ((struct elem *) v)->key;
    return (v1 < v2) ? -1 : (v1 > v2) ? 1 : 0;
}

void sort_table() {
    qsort((void *) Table,
        (size_t) COUNT,
        sizeof(struct elem),
        key_compare);
}
```



Funkce s proměnným počtem parametrů

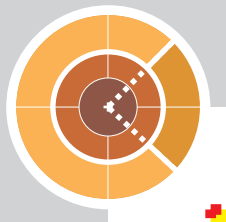
- knihovna **stdarg** (tj. **#include <stdarg.h>**) poskytuje datové typy, makra a funkce pro definici funkcí s proměnným počtem parametrů
- funkce s proměnným počtem parametrů se deklaruje

```
void vargfn(char *types, ...) { ...
```

3 tečky značí, že parametrů je neurčité množství

- blok zpracování parametrů začíná voláním funkce **va_start()** a končí voláním **va_end()**
- jednotlivé parametry se získají pomocí funkce **va_arg()**

```
typ va_arg(va_list ap, typ);
```




```c
#include <stdarg.h>
#include <stdio.h>

void printargs(char *argtype, ...) {
    va_list arguments;
    int arg_int;
    char *arg_str, thisarg;
    va_start(arguments, argtype);
    while ((thisarg = *argtype++) != '\0') {
        switch (thisarg) {
            case 'i':
                arg_int = va_arg(arguments, int);
                printf("%d\n", arg_int);
                break;
            case 's':
                arg_str = va_arg(arguments, char *);
                printf("%s\n", arg_str);
        }
    }
    va_end(arguments);
}
```

ukazatel na poslední "pevný" p-metr

```c
printargs("isis",
    5, "abc", 7, "def");
```