

Pole a ukazatele

Pole nebo řetězec – ukazatel na jeho nultý prvek.

```
int a[3] = {1, 2, 3};           // a je ukazatel na nultý prvek
int *pta;                       // proměnná obsahující ukazatele (na int)
char s[] = "FIT";               // řetězec (přepisovatelný)
char *t = "ABC";                // ukazatel na nultý znak konst. řetězce
char *ptc;                      // proměnná obsahující ukazatele na znaky
char **pts;                     // proměnná obsahující ukazatele na řetězce

printf("%c%d\n", s[0], a[0]);   // nulté prvky řetězce a pole...
printf("%c%d\n", *s, *a);       // ...totéž

pta = a;                        // do pta uloží adresu nultého prvku pole

printf("%p\n%p\n", a, pta);     // ukazatele na stejné místo

pts = &ptc;                     // do pts uloží adresu ukazatele ptc
*pts = t;                       // do ptc uloží adresu písmene 'A'
printf("%c%c\n", *ptc, **pts);  // "AA"

ptc = s;                        // adresa písmene 'F'
*s = 'G';                       // přepíše 'F' na 'G'
printf("%s\n", *pts);           // "GIT"
```

Pole a ukazatele

Co vypíše tento program?

```
int main (void) {  
    char s[] = "STR";  
    char *ptc;  
    char **pts;  
  
    pts = &ptc;  
    ptc = s;  
    **pts = '\\0';  
    printf("\\\"%s\\\"\\n", ptc);  
}
```

Pole jako argument funkce

Napište proceduru

```
void map (T (*f)(T), int n, T a[] ),
```

která v poli `a` velikosti `n` nahradí každý prvek `x` prvkem `(*f)(x)`.

Přitom `(*f) : T → T` je funkce (tedy `f` je ukazatel na funkci)

a `T` je obecný skalární typ, například `typedef int T;`

Doplňte také její volání v hlavním programu, které všechny prvky pole umocní na druhou.

Pole jako argument funkce

```
void map (int (*f)(int), int n, int a[]) {  
    /* Doplňte */  
    return;  
}  
  
int square (int n) { return n*n; }  
  
int main (void) {  
    int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
    int i;  
    for (i=0; i<10; i++) printf(" %4d", arr[i]);  
    putchar('\n');  
  
    map( ..... ); /* Doplňte volání map */  
  
    for (i=0; i<10; i++) printf(" %4d", arr[i]);  
    putchar('\n');  
    return 0;  
}
```

Pole jako argument funkce

V programu máme deklarace

```
int arr[100000];  
int N = 100000;
```

a definované nějaké tři funkce (za `return` jsou *čisté* výrazy, tedy výrazy, které *nemění* globální stav)

```
int f1 (int k) { return ... ; }  
int f2 (int k) { return ... ; }  
int f3 (int k) { return ... ; }
```

V hlavní funkci `main` jsou tři po sobě jdoucí volání funkce `map`:

```
map(f1, N, arr);  
map(f2, N, arr);  
map(f3, N, arr);
```

Zapište tyto příkazy efektivněji, jediným voláním funkce `map`, tj. jediným průchodem polem `arr`.

K tomu si můžete definovat další pomocnou funkci `int g (int k)`.

Paměťové třídy dat

Data, s nimiž pracuje program, rozlišujeme

- podle času, kdy mají přidělenou paměť,
- a podle umístění v paměti.

Statická data

existují ve statické části paměti (společně s kódem) po celou dobu běhu programu. Paměť jim alokuje kompilátor.

V C jsou to

- proměnné deklarované na nejvyšší úrovni (vně funkcí),
- lokální proměnné ve funkcích deklarované s třídou **static**.

Zásobníková data

Alokují se v *systemovém zásobníku* při zavolání funkce nebo při vstupu do bloku, v němž jsou definována. Při návratu z funkce nebo při opuštění bloku jsou ze zásobníku odstraněna.

V C mezi zásobníková data patří

- formální parametry funkcí,
- data definovaná ve funkcích a ve vnořených blocích, pokud nemají explicitně specifikovanou jinou paměťovou třídu.

Dynamická data na hromadě

Paměť pro dynamická data si program sám alokuje i uvolňuje explicitním voláním systémových funkcí pro práci s tzv. *hromadou*.

Alokace paměti:

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

Uvolnění paměti:

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

Každý alokovaný úsek paměti musí program vždy uvolnit, jakmile ho přestane potřebovat. Jinak hrozí *únik paměti*, kdy se program rozrůstá nad přijatelné meze a může zahltit operační systém.

Dynamicky alokované vektory

Vektor je reprezentován svou dimenzí (tj. počtem souřadnic) `dimen` a polem souřadnic (položek) `coords[]`.

```
typedef struct {  
    unsigned int dimen;    // počet položek  
    double coords[];      // položky  
} Vector;
```

Všimněme si, že pole `coords` nemá specifikovanou velikost. Ta je známa až při vytváření vektoru v dynamické paměti.

Při alokaci paměti pro n -rozměrný vektor (pro vektor s n položkami) zavoláme funkci `malloc` s argumentem vyjadřujícím velikost paměti v bytech. Jaká je tato velikost?

```
pv = malloc(sizeof(Vector) + n*sizeof(double))
```

Dynamicky alokované vektory

Napište funkce `constr_vec` a `destr_vec` pro konstrukci a destrukci vektoru dané dimenze.

Konstruktor alokuje paměť a destruktory tuto paměť opět uvolní.

```
Vector *constr_vec (unsigned int dim);  
void destr_vec (Vector *v);
```

Napište funkci `add_vec`, která sečte dva vektory a součet vloží do třetího vektoru.

```
int add_vec (Vector *v1, Vector *v2, Vector *v3)
```

Funkce vrátí kladný chybový kód, když některý z ukazatelů je `NULL` nebo když vektory mají nekompatibilní dimenze. Jinak vrátí nulu.

```

typedef struct {
    unsigned dimen;    // počet položek
    double coords[];  // položky
} Vector;

Vector *constr_vec (unsigned dim) {    /* DOPLŇTE */    }

void destr_vec (Vector *v) {    /* DOPLŇTE */    }

int add_vec (Vector *v1, Vector *v2, Vector *v3) {
    if (v1 == NULL || v2 == NULL || v3 == NULL) return 9;
    if (v1->dimen != v2->dimen || v2->dimen != v3->dimen) return 1;
    /* ... DOPLŇTE sčítání */
}

int main (void) {
    Vector *u, *v, *w;
    /* DOPLŇTE konstrukci, součet a destrukci vektorů */
    return 0;
}

```

Argumenty příkazového řádku

Program `range` vypisuje souvislou část textového souboru od daného řádku po daný řádek.

Jméno souboru i čísla řádků jsou argumenty příkazového řádku:

```
./range -f m -t n -s filename
```

- za volbami `-f` (from) a `-t` (to) musí bezprostředně následovat celé číslo
- volba `-s` (source) musí být bezprostředně následovaná jménem souboru
- každá volba je nepovinná, žádná se však nesmí opakovat
- na pořadí voleb nezáleží
- chybějící `-f` se v konfiguraci doplní číslem 1
- chybějící `-t` se v konfiguraci doplní číslem `INT_MAX`
- chybějící `-s` se v konfiguraci doplní prázdným řetězcem

Argumenty příkazového řádku

Příkazový řádek:

```
./range -f m -t n -s filename
```

Struktura pro konfiguraci:

```
typedef struct {  
    char filename[50];  
    int fromline;  
    int toline;  
} Config;
```

```
Config config = {  
    .filename = "",  
    .fromline = 1,  
    .toline = INT_MAX  
};
```

Argumenty příkazového řádku

Napište funkci

```
int parsecmdline (int argc, char **argv, Config *conf)
```

která nastaví konfigurační strukturu `Config config`.

Jako svůj výsledek

- vrátí nulu, je-li příkazový řádek syntakticky správně,
- jinak vrátí kladné číslo chyby.

Argumenty příkazového řádku

```
typedef struct {    // Typ konfiguračního záznamu
    char filename[50];
    int fromline;
    int toline;
} Config;

// Konfigurační záznam s implicitními hodnotami
Config config = {.filename = "", .fromline = 1, .toline = INT_MAX};

// Analýza příkazového řádku a naplnění záznamu *conf
int parsecmdline (int argc, char **argv, Config *conf) {
    /* DOPLŇTE kód pro [-f číslo] [-t číslo] [-s jméno] */
}

int main (int argc, char **argv) {
    int exitcode = parsecmdline(argc, argv, &config);
    // if (exitcode != 0) ... ošetření chyby

    // Kontrolní výpis konfigurace
    printf("filename: \"%s\"\nfrom line:%3d\nto line:  %3d\n",
        config.filename, config.fromline, config.toline);

    return 0;
}
```

Argumenty příkazového řádku

Poznámka: Implementace funkce `parsecmdline` je cvičná. Pro syntaktickou analýzu příkazového řádku ve složitějších programech je výhodnější použít funkce ze skupiny funkcí `getopt` z knihovny `unistd`.