Práce s pamětí a předávání parametrů

Úvod do programování 1

Motivace

Zatím jsme se setkali s následujícími „problémy“:

- Proměnná existuje / je dostupná jen v bloku, kde vznikla
- Pole existuje v bloku, kde bylo vytvořeno

```
int *funkce(int a, int b) {
    int c = a + b;
    int pole[] = { a, b };
    return &c; // NEJDE!!!
    return &a; // NEJDE!!!
    return pole; // NEJDE!!!
}
```

- Pole musí být vytvořeno o velikosti zadané v době kompilace

```
int *vytvor_pole(int velikost){
    int pole[velikost]; // NEJDE!!!
    return pole; // NEJDE!!!
}
```

Obecný ukazatel

- Ukazatel bez uvedení typu, se kterým bude moci pracovat
```
void *pamet;
```
- Nelze s ním provádět aritmetické operace ani dereferenci
```
int pole[3];
pamet = pole;
pamet++; // NEJDE!!!
int cislo = 5;
void *ukazatel = &cislo;
*ukazatel = 7; // NEJDE!!!
```
- Po přetypování je již možné provádět vše
```
((int *)pamet)++;
*((int*)ukazatel) = 7;
```

Dynamické alokování paměti

- Provádíme například pomocí funkce `malloc`
- Jediný parametr udává velikost paměti, kterou chceme alokovat (v bytech)
- Výsledek volání funkce je ukazatel na začátek přidělené paměti (`void *`)
- Nebo `NULL` v případě neúspěchu alokace

```
int *pole;
pole = malloc(40); // alokace 40 bytu
pole = malloc(10 * sizeof(int)); // spravne
pole = (int *)malloc(10 * sizeof(int)); // jeste lepsi
...
pole[0] = 10;
pole[9] = 42;
```

Příklad použití

- Funkce vracející pole druhých mocnin čísel ze vstupního pole:

```
int *powers(int *input, int count) {
    int i;
    int *output = (int *)malloc(sizeof(int) * count);
    // int output[count]; -> toto nefunguje

    for (i = 0; i < count; i++) {
        output[i] = input[i] * input[i];
    }
    return output;
}
```

Další možnosti alokace

Funkce `void *calloc(size_t items, size_t size)`

- alokuje paměť pro `items` prvků o velikosti `size`
- tedy `items*size` bytů
- navíc tuto paměť vyplní nulami

Funkce `void *realloc(void *old, size_t size)`

- zajišťuje změnu velikosti dříve alokovaného bloku paměti `old`
- na velikost `size`
- data původního bloku paměti budou zkopírována

Všechny zmíněné funkce najdete v knihovně `stdlib.h`

Uvolnění paměti

- Pokud alokovanou paměť (pomocí `malloc`, `calloc`, `realloc`) nepotřebujeme, měli bychom ji uvolnit
- Jinak bude nedostupná až do konce programu
- Toto může způsobit u reálného softwaru značné problémy
- K uvolnění paměti se používá funkce `void free(void *block)`
- Příklad:

```
int *cisla = (int *)calloc(velikost, sizeof(int));
// tady pole cisla k necemu použijeme
...
// a kdyz uz neni potreba
free(cisla);
```

Příklad práce s pamětí

```
const int b_length = 100; /* block length */
int *data, *h;
int b_count = 1; /* block count */
…
data = (int *)malloc(b_length * sizeof(int));
if (data == NULL) return 1; /* chyba */
…
b_count++;
h = (int *) realloc(data, b_count * b_length * sizeof(int));
if (h == NULL) return 1; /* chyba */
else data = h;
…
free(data);
```

Parametry a návratová hodnota

- Běžně funkce vrací jeden výsledek pomocí své návratové hodnoty

```
int soucet(int a, int b) {
    return a + b;
}
```

- Změny parametrů v těle funkce se neprojeví v místě volání funkce

```
int nsd(int a, int b){
    while (b){
        int r = a % b; a = b; b = r;
    }
    return a;
}
...
int prvni = 84, druhe = 120;
printf("%i", nsd(prvni, druhe));
```

Pole jako parametr funkce

- Trochu jinak se chová jako parametr funkce pole

```
void na2(int cisla[], int pocet){
    for (int i = 0; i < pocet; i++)
    cisla[i] *= cisla[i];
}

...

int data[] = { 1, 2, 3, 4, 5 };
na2(data, 5);
for (int i = 0; i < 5; i++)
    printf("%i, ", data[i]);
```

- Proč?

Parametr předávaný odkazem

- Stejného principu můžeme použít i u jiných typů parametrů

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
...
int prvni = 84, druhe = 120;
swap(&prvni, &druhe);
```

- Lze využít pro funkce, které mění parametry
- Nebo mají více návratových hodnot
- Další výhodou může být nevytváření kopie dat (tj. ušetření paměti; především u velkých struktur)

Předání pole odkazem

- Pokud bychom chtěli změnit uvnitř funkce samotný ukazatel:

```
int vytvor_pole(int pocet, int hodnota, int **vysledek){
    *vysledek = (int *)malloc(počet * sizeof(int));
    if (*vysledek)
        for (int i = 0; i < pocet; i++)
            (*vysledek)[i] = hodnota;
    else { printf("Chyba!"); return 1; }
    return 0;
}
...
int *jednicky;
vytvor_pole(10, 1, &jednicky);
for (int i = 0; i < 10; i++)
    printf("%i, ", jednicky[i]);
```

Předávání hodnotou vs. odkazem

Při předávání hodnotou:

- Parametr je jen vstupní
- Typ parametru odpovídá datům
- Při volání předáváme jakoukoli hodnotu daného typu
- Běžný způsob předání parametru

```
void moje_fce(int data){ ... }
...
int cislo;
moje_fce(cislo);
moje_fce(3 * cislo);
```

Při předávání odkazem:

- Parametr může být i výstupem
- Typ parametru je ukazatel na typ dat
- Při volání předáváme adresu dat
- Používáme pro předávání výstupních parametrů a dále u polí (nejde jinak) a velkých struktur

```
void moje_fce(int *data){ ... }
...
int cislo;
moje_fce(&cislo);
```