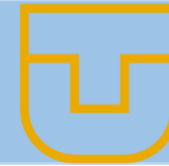


Cvičenie č. 6

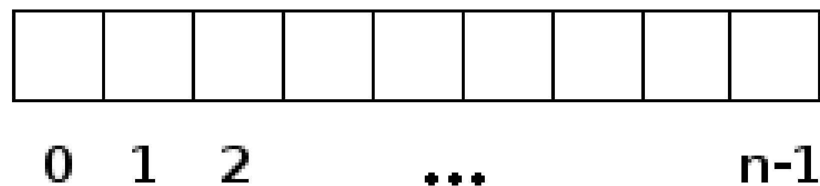
Náplň:

- Jednorozmerné (1R) polia
- Triedenie/usporadúvanie prvkov 1R poľa
- Vyhľadávanie v 1R poliach
- Príkaz **for()**

Ing. Ondrej Kováč, PhD.

Reprezentácia údajov v poliach

- Pole je dátová štruktúra, v ktorej sú dáta v pamäti uchovávané za sebou.
- Pole je v pamäti uchované nasledovne:



- **Pole v jazyku C indexujeme od nuly.**
- Deklarácia poľa vyzerá nasledovne:

```
<typ> <názov>[<veľkosť>];
int pole[10];
```

Inicializácia poľa

- Pri deklarácií poľa dochádza k rezervovaniu miesta v pamäti RAM. Nedochádza však k odstráneniu údajov, ktoré tam zanechali iné programy. Preto je pole vhodné inicializovať.
- Ak chceme aby prvky poľa boli definované už pri deklarácii môžeme to spraviť takto:

```
int pole1[5] = {1, 2, 3, 4, 5};
```

Ak inicializujeme nie je potrebé zadávať veľkosť poľa pretože tá je známa z inicializácie

```
int pole2[] = {1, 2, 3, 4, 5};
```

Pole môžeme inicializovať aj čiastočne

```
int pole3[5] = {1, 2};
int pole4[5] = {0};
```

Inicializáciu je možné vykonať len raz a to pri deklarácii poľa neskôr hodnotu prvkov možno meniť len ich prepísaním!

Indexovanie prvkov poľa

K jednotlivým bunkám poľa pristupujeme pomocou ich indexácie.

int **pole[5] = {1, 2, 3, 4, 5};**

int a = pole[0]; // v a bude 1. bunka poľa
int b = pole[1]; // v b bude 2. bunka poľa
int c = pole[5]; // **POZOR!! Pole má veľkosť 5 prvkov. Tatko sa ale snažíme pristúpiť k šiestemu prvku.**

Jazyk C nemá kontrolu rozsahu poľa!

```
#include <stdio.h>
int main()
{
	int i;
	int pole[5] = {1,2,3,4};

	for(i=0;i<6;i++)
		printf("%d ",pole[i]);
		printf("\n");
}
```

Program bol spustený 5x za sebou

```
ok@zap:P5$ ./a.out
 1   2   3   4   0   32764
ok@zap:P5$ ./a.out
 1   2   3   4   0   32766
ok@zap:P5$ ./a.out
 1   2   3   4   0   32764
ok@zap:P5$ ./a.out
 1   2   3   4   0   32766
```

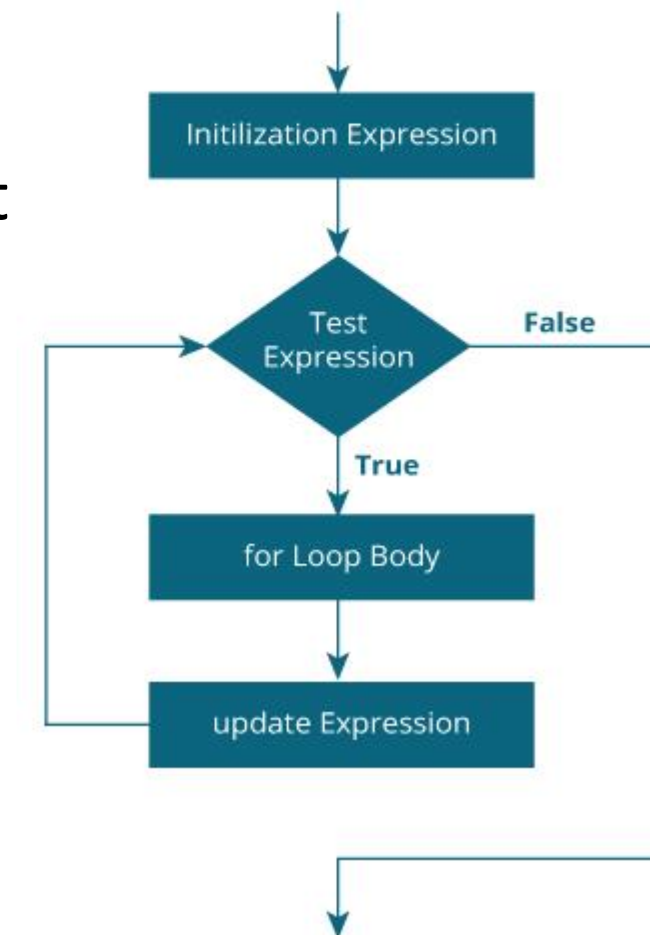
Náhodná hodnota z RAM pamäte

Nápomocným nástrojom môže byť **cppcheck**. Tento nástroj vykonáva statickú kontrolu kódu.

Príkaz for()

Tento príkaz slučky používame hlavne vtedy ak vieme počet iterácií.

for(<inicializácia>;<podmienka ukoncenia> ; <inkrementácia/dekrementácia>)



Inkrementácia

```
for(i=0;i<6;i++)
for(i=-10;i<=0;i++)
```

Dekrementácia

```
for(i=10;i>0;i--)
```

Zistenie veľkosti poľa - **sizeof()**

Veľkosť poľa v jazyku C môžeme zistiť pomocou operátora **sizeof()**. Návratová hodnota sizeof() predstavuje počet bajtov premennej, ktorej veľkosť je zisťovaná. Preto pre zistenie počtu prvkov poľa, je potrebné túto veľkosť deliť veľkosťou takého údajového typu, ako je samotné pole.

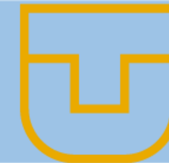
```
int pole[] = { 1, 5, 2, 3, 6, 8, 9, 7, 10, 18, 16, 14, 13, 15, 12, 11, 4, 17};

int size = sizeof(pole)/sizeof(int);
```

Dynamická alokácia veľkosti poľa

Zatiaľ sme vždy veľkosť poľa vedeli. Vedeli sme to preto lebo sme ju zadávali pri deklarácii poľa. Teda alokovali sme pamäť **staticky**.
Niekedy programátor nevie koľko pamäte bude potrebovať a preto je potrebné aby sa alokovala **dynamicky** za behu programu.

Predstavme si prípad keď program od používateľa najprv zistí koľko hodnôt zamýšľa zadať a až následne sa v alokuje potrebná pamäť. Potom používateľ zadá požadovaný počet hodnôt.




```
#include<stdio.h>
int main()
{
        int pocet;
        printf("Zadaj pocet prvkov pola \n>> ");
        scanf("%d", &pocet);                          // Nacitanie poctu prvkov
        int pole[pocet];                              // dynamicky alokovane
        for(int i=0; i<pocet; i++)
        {
                printf("Zadaj %d. prvok poľa \n>> ",i);
                scanf("%d",&pole[i]);                 //Nacitanie i-teho prvku
        }
        for(int i=0; i<pocet; i++)
        {
               printf("%d ",pole[i]);
        }
        printf("\n");
return 0;
}
```

Ak som lenivý vypisovať ručne → Presmerovanie vstupu

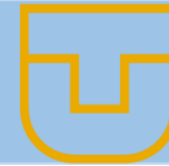
```
ok    zd@zap:P5$ gcc dynamicka_alokacia.c -Werror -Wall -o DYN
ok    zd@zap:P5$ cat vstup.txt
10
1 5 4 3 7 6 2 8 10 9
ok    zd@zap:P5$ ./DYN < vstup.txt
Zadaj pocet prvkov pola
>> Zadaj 0. prvok poľa
>> Zadaj 1. prvok poľa
>> Zadaj 2. prvok poľa
>> Zadaj 3. prvok poľa
>> Zadaj 4. prvok poľa
>> Zadaj 5. prvok poľa
>> Zadaj 6. prvok poľa
>> Zadaj 7. prvok poľa
>> Zadaj 8. prvok poľa
>> Zadaj 9. prvok poľa
>> 1 5 4 3 7 6 2 8 10 9
```

Triedenie prvkov poľa

V praxi sa častokrát stretávame s požiadavkou na usporiadanú reprezentáciu dát. Veľakrát sú dáta uložené neusporiadané a k ich usporiadaniu dochádza až neskôr. Triediacich algoritmov je viac než 15 a dnes si ukážeme najbežnejšie triediace algoritmy.

Bublinkové triedenie – Bubble sort

Rýchle triedenie - QuickSort



Algoritmus **Bubble sort**

Pracuje opakovaným prechodom cez zoznam, ktorý má byť utriedený porovnávajúc vždy dva prvky. Ak prvky nie sú v správnom poradí, zamení ich. Porovnávanie prvkov v zozname trvá, pokiaľ sú potrebné výmeny, teda pokiaľ nie je zoznam usporiadaný. Algoritmus dostal názov vďaka tomu, že menšie prvky sa „prebublinkujú“ na začiatok zoznamu.

```
void bsort(int numbers[], const int size)
{
    for(int j = size-1; j > 0; j--)
        for(int i = 0; i < j; i++)
        {
            if(numbers[i+1] < numbers[i])
            {
                int tmp = numbers[i+1];
                numbers[i+1] = numbers[i];
                numbers[i] = tmp;
            }
        }
}
```

Algoritmus **QuickSort – qsort()**

```
void qsort (void* triedene_pole, int <velkost_pola>, int <velkkost_prvku_pola>,int  funkcia_porovnania);

#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i;
   int pole[] = {1, 4, 2, 5, 8, 7, 3, 6};
   int pocet=sizeof(pole)/sizeof(int);
   qsort(pole, pocet, sizeof(int), porovnaj);
   for(i=0; i<pocet; i++)
      printf("%d ", pole[i]);
}
```

Funkcia pre porovnanie by mala vyzerať takto

```
int porovnaj(const void* var1, const void* var2)
{
    return *(int*)var1 - *(int*)var2;
}
```

Vyhľadávanie v poliach

Operácia vyhľadávania patrí k najčastejším operáciám, ktoré budete nad údajmi v poli robiť. Napr. aj najdôležitejšia operácia pri práci s *SQL* databázami je práve operácia *selekcie* (vyhľadávania).
V čom spočíva problém - máme pole, ktoré obsahuje obrovské množstvo údajov a našou úlohou je nájsť požadovaný údaj čo najrýchlejšie (a teda - čo najefektívnejšie).
Aj pri vyhľadávaní existuje viacero spôsobov dosiahnutia výsledku. Najjednoduchším spôsobom je **lineárne vyhľadávanie**.

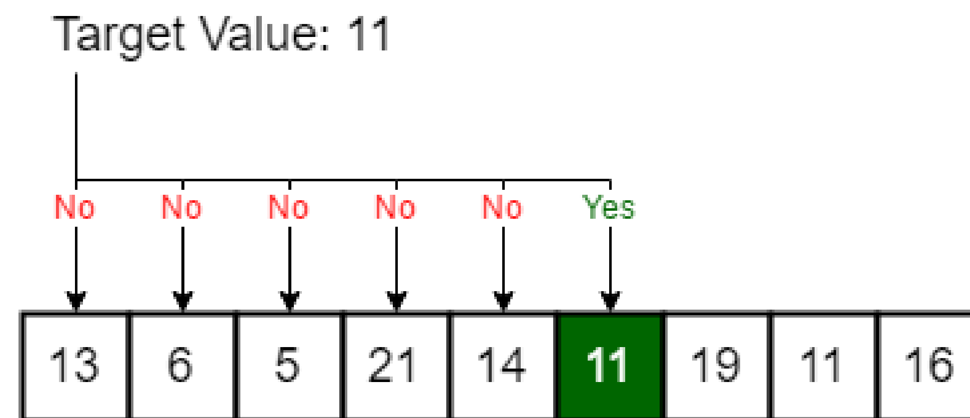
Lineárne vyhľadávanie

Z algoritmického pohľadu je tento spôsob najjednoduchší, pretože je veľmi intuitývny. Spočíva v postupnom prehľadávaní celého poľa. Toto prehľadávanie končí ak sa dosiahne koniec poľa alebo ak sa nájde vyhľadávaný prvok.
Na vyhľadanie prvku potrebujeme minimálne jedno testovanie *(to je v prípade, že hľadaný prvok je prvým prvkom poľa)*

O(1)

V najhoršom prípade je potrebné vykonať toľko testovaní koľko je prvkov poľa.

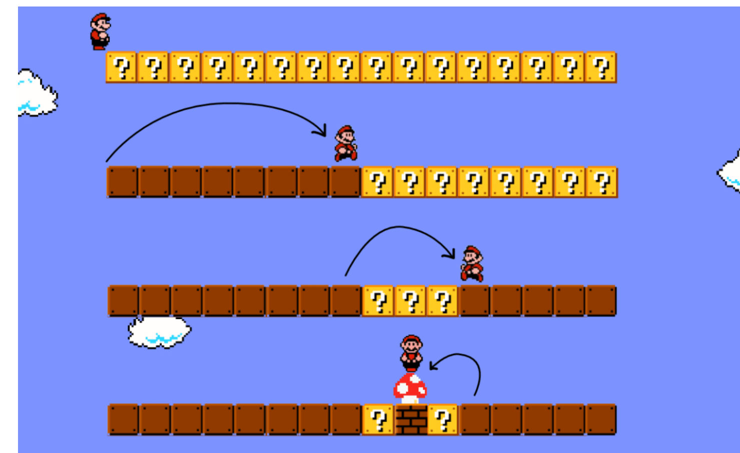
O(N)



Vyhľadávanie v poliach

Binárne vyhľadávanie

V porovnaní s lineárnym vyhľadávaním ide o sofistikovanejší spôsob vyhľadávania. Princíp spočíva v delení intervalu (celého poľa) na subintervaly. Pritom sa hľadaný prvok porovnáva s prostredným prvkom daného subintervalu. Ak nie sú zhodné vyhodnotí sa, či je hľadaný prvok väčší alebo menší a vyhľadávanie sa zúži na oblasť vľavo alebo vpravo od prostredného prvku. Takto sa vždy eliminuje polovica prvkov poľa.



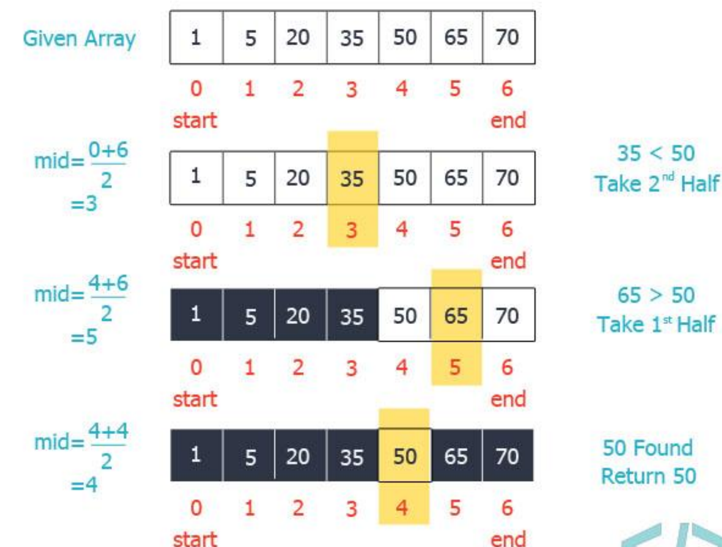
Na vyhľadanie prvku potrebujeme minimálne jedno testovanie *(to je v prípade, že hľadaný prvok je prostredným prvkom poľa)*

O(1)

a v najhoršom prípade je potrebné vykonať nasledovný počet testovaní.

O($\log_2(N)$)

Binárne vyhľadávanie funguje len na zoradených poliach!



Samostatná práca

- Pracujte na úlohách ku cvičeniu
- V prípade otázok zodvihnite ruku, prídem
- Kto veci v úlohách ovláda môže odísť