Pole

Kdybychom v jazyce C++chtěli načíst větší počet čísel nebo znaků a všechny bylo by nutné všechny tyto hodnoty nadále uchovávat v paměti počítače, tak by bylo potřeba v paměti počítače alokovat stejný počet proměnný jako počet uchovávaných hodnot. Tzn. existovalo by např. 100 různých proměnných a tím pádem 100 identifikátorů. Není nutné tedy zdůrazňovat, že výsledný program by byl velice nepřehledný.

Existuje řešení tohoto problému, které se nazývá pole. Pole je v podstatě v paměti počítače seznam hodnot stejného datového typu, které mají jednotný identifikátor. Pole deklarujeme velice podobně jako proměnou.

```
datovy_typ INDETIFIKATOR [počet_prvku]={prvek0,
prvek1,...,prvek n};
```

Jak je vidět bývá dobrým programátorským zvykem psát identifikátory pole velkými písmeny, abychom v programu od sebe jednoduše odlišili pole a jednotlivé proměnné.
V hranatých závorkách následuje nějaká hodnota nebo identifikátor proměnné, která nám udává kolik prvků má pole obsahovat. Poslední část obsahuje v složených závorkách výčet konkrétních prvků. Je nutné uvést, alespoň jednu z těchto částí. V případě neuvedení jedné části ji nelze vynechat zcela musíme uvést pak jen prázdné závorky.

Nyní nastává nám otázka, jak se dostaneme k jednotlivým prvkům v poli? Představíme-li si pole jako tabulku o jednom řádku, tak jednotlivé prvky mají svůj index - pořadové číslo. Prvnímu prvku odpovídá nula a poslednímu pak počet prvků mínus 1. Tohoto indexu využíváme k práci s polem. Do pole přistupujeme pomocí cyklu, nejčastěji se používá cyklus for. Nyní si ukážeme jednoduchý příklad, který nám vypíše obsah pole na monitor.

```
# include <iostream>

using namespace std;
int HODNOTY[]={1, 3, 5, 7, 2, 5, 1};

int main () {


   for (int i = 0; i < sizeof(HODNOTY) / sizeof(int); i++)
   {
    cout<<HODONOTY[i]<< endl;
   }
        cin.get();
        return 0;
   }
```

Uvedený program nám vypíše na řádky pod sebe obsah pole „HODNOTY“. Cyklus v každém kroku vypíše hodnotu, která je uvedena v poli pod daným indexem. Možná Vás zaskočilo klíčové slovo sizeof. Jelikož mám v poli prvky pouze vyjmenovány, tak program nemůže vědět kolik jich vlastně je. Sizeof je operátor jazyka C/C++, který při jeho aplikaci na nějaký objekt vrací jeho velikost v paměti počítače v bytech. První část sizeof(HODNOTY) vrací velikost pole HODNOTY a druhá část vrací velikost datového typu int. Výsledek dělení těchto dvou hodnot je počet prvků v poli.

Zápis hodnot do pole probíhá opět pomocí procházení for cyklu. Pro zápis používáme standardní objekt, který nám představuje vstup z klávesnice tedy cin. V následujícím příkladu si do pole uložíme několik celých čísel a ty pak vypíšeme.

```
# include <iostream>

using namespace std;
     int HODNOTY[10]={};
     int i, pom;

     int main () {
          for (i = 0; i < 10; i++)
          {
               cin >> pom;
               HODNOTY[i] = pom;
          }

          for (i = 0; i < 10); i++)
          {
               cout << "index = " << i<< endl;
               cout << "hodnota = " << HODNOTY[i]<< endl;
          }


               cin.get();
               cin.get();
          return 0;
     }
```

Již umíme pole vytvořit, naplnit ho hodnotami a vypsat je. Jakou výčet základních operací s polem si však musíme ukázat, alespoň jeden algoritmus jak pole seřadit. Použijeme velice jednoduchou metodu, která se nazývá „Buble sort“. Nejprve si vysvětlíme na jednoduchém příkladu řady čísel, jak metoda funguje.

13 5 3 8 0 1

V uvedené řadě porovnáváme vždy první prvek s druhým prvkem a pokud první prvek bude větší než druhý a tak je prohodíme.

5 13 3 8 0 1

Nyní budeme porovnávat druhý a třetí prvek a opět v našem případě bude následovat prochození.

5 3 13 8 0 1

Dále bude číselná řada vypadat takto:

5 3 8 13 0 1
5 3 8 0 13 1
5 3 8 0 1 13

Nyní budeme začínat opět od prvního prvku řady.

3 5 8 0 1 13

Po porovnání druhé a třetího prvku nedojde k prohození hodnot.

3 5 8 0 1 13

Dál pokračujeme následovně:

3 5 0 8 1 13
3 5 0 1 8 13
3 5 0 1 8 13

Opět začínáme od začátku:

3 5 0 1 8 13
3 0 5 1 8 13
3 0 1 5 8 13
3 0 1 5 8 13
3 0 1 5 8 13

Znova porovnáváme od začátku řady:

0 3 1 5 8 13
0 1 3 5 8 13
……
0 1 3 5 8 13

Konec algoritmu bude zajištěn tak, že když při porovnávání od začátku řady neprohodíme žádné pořadí dvou prvku tak je řada seřazena. Nyní přistoupíme přímo k ukázce již zmíněného programu.

```
#include <iostream>
using namespace std;
int main(){

int A[] = {8, 4, 2, 100, 0, 9, 7, 6, 3, 1};
int i, j, m, tmp;

for (i = 0; i < sizeof(A) / sizeof(int); i++) {
  m = i;
  for (j = i + 1; j < sizeof(A) / sizeof(int); j++)
  /*pouziti dvou cyklu for a ted dvou indexu
  zajistime vzdy nacetni dvou sousedicich hodnot
   hodnota pod indexem j bude mit vyd index o 1 vetsi*/
    if (A[m] > A[j]) m = j;
  /* v A[m] je minimum, prohoï A[m] a A[i] */
  tmp = A[i];
```

```
    A[i] = A[m];
    A[m] = tmp;
  }

  for (int i = 0;i < sizeof(A) / sizeof(int); i++)
    cout << A[i]<<" ";

      cin.get();
      cin.get();
      return 0;
  }
```

Řetězce string

Deklarace jsou v hlavičkovém souboru <string>
Řetězce nahrazují char * používané pro posloupnosti znaků v C
Výhody:

* nemají omezenou délku (v principu)
* jsou pro ně definovány užitečné operace:

```
string s1 = "Hello";
string s2 = "world";
string s3 = s1 + ", " + s2 + "!\n";

s1 += '\n'; // s3 obsahuje "Hello, world!\n"
char posledni = s2[s2.size()-1] // operátor [] přetížen i pro string!
int pos = s3.find("ll") // Vrati pozici zacatku podretezce ll v s3, tj. 2
int pos = s3.find_first_of(",ow")
  // Vrati pozici prvniho z uvedenych znaku, tj. 4; prvni je 'o' na pozici 4
  // Funkce find_first_not_of naopak najde první znak neobsazeny v danem retezci
  // U obou techto funkci muze byt parametrem znak, pole znaku i string;
  // pokud se retezec nebo znak nenajde, vrati cislo vetsi nez delka retezce
s3 = s1.substr(0,4); // 4 znaky počínaje nultým, tj. "Hell"
s1.replace(0,5,"Ahoj"); // Nahradit lze i podřetězcem jiné délky"
```

* lze je porovnávat s textem v uvozovkách a konvertovat na "staré" řetězce:

```
string s;
if (s == "ano") {
// ...
}
printf("%s\n", s.c_str()); // c_str() konvertuje string na char*
```

Vstupy a výstupy

Deklarace vstupních a výstupních proudů jsou v hlavičkovém souboru iostream
Výstup

Standardním výstupním proudem je cout, standardním proudem pro chybový výstup (nebo obecně oznámení určená uživateli, např. výzvu k zadávání dat) je cerr.

Je k dispozici přetížený operátor << (přetížený operátor je totéž jako přetížená funkce, jen jeho argumenty se nezapisují do závorek, ale infixově). Operátor vrací výsledek výstupní operace (v podstatě ukazatel do výstupního proudu), takže je možné vypsat více hodnot jedním příkazem:

```
cout << "Pocet bodu: " << n;
```

Tištěná data lze formátovat podobně jako v příkazu printf. Definice potřebných manipulátorů jsou v hlavičkovém souboru <iomanip>.

```
#include <iostream>
#include <iomanip>
using namespace std;
//...
double angle=0.1234569;
cout << angle << endl;
// vytiskne se 0.123457
cout << setprecision(4) << angle << endl;
// vzhledem k použití manipulátoru setprecision
// se vytiskne 0.1235
cout << angle << endl;
// vytiskne se opět 0.1235 !
cout << setprecision(8) << angle << endl;
// vytiskne se 0.1234569
cout << setw(10) << angle
// zarovná šířku na 10 znaků (tisk tabulek)
```

Většina manipulátorů působí až do další změny (manipulátor setw však platí jen pro následující položku)!

Vstupy

Standardním vstupním proudem je cin

Je k dispozici přetížený operátor >> (analogicky jako << pro výstup]

```
int i; double d;
cin >> i >> d;
```

Posloupnost znaků bez oddělovačů lze načítat přímo do proměnné typu string:

```
string str;
cout << "Napis krestni jmeno\n";
cin >> str;
```

řetězce typu string stejnojmennou funkcí z hlavičkového souboru string:

```
#include <string>
string str;
getline(cin, str)
```

Číst a psát znak po znaku můžeme pomocí funkcí get a put:

Program pro kopírování vstupu na výstup:

```
int main()
{
char c;
while (cin.get(c) ) cout.put(c);
}
```

Lze definovat vlastní verze operátorů << a >> pro tisk/načítání objektů, které potřebujeme zpracovávat v programu!