Pole

Doposud jsme pracovali s tzv. primitivními proměnnými, které uchovávaly jednu hodnotu. V řadě případů (např. počítání průměru známek) se však hodí mít k dispozici jakousi sérii proměnných, která má jméno a jednotlivé proměnné jsou číslované (např.: znamka1, znamka2, znamka3, ...). Takový požadavek je v jazyce C (a dalších) řešen pomocí tzv. pole.

Pole je **datová struktura**, ve které jsou data (jednotlivé proměnné) uchována v řadě za sebou (je tedy lineární strukturou). Místo slova „proměnná" se však v souvislosti s polem hovoří o **prvcích**. Paměť pro pole je vyhrazena tak, jak je to zobrazeno na obrázku níže (jeden prvek pole za druhým). Místo číslování prvků se však v jazyce C používá indexování. Velmi zjednodušeně jde vlastně o číslování, které začíná nulou. Indexy se za název zapisují do hranatých závorek (např. znamky[0], znamky[1], ... nebo např. p[0], p[1], ...). Pro pole s N prvky má tedy první prvek pole index 0 a N-tý prvek má index N - 1.

Ukázka 10ti prvkového pole (N=10) s názvem „p" a jeho podoba v paměti.

Deklarace pole

Deklarace pole má následující strukturu:

```
<typ> <název>[<[N]>];
```

Deklarace se příliš neliší od deklarace proměnné. **`typ`** je základní typ buňky pole (pole je homogenní, není možné měnit typ u jednotlivých buněk, všechny buňky tedy mají stejný typ), **`název`** je identifikátor pole – název proměnné typu pole – a **`N`** určuje, kolik buněk bude pole mít. V konkrétním případě tedy pole s názvem „p" o deseti prvcích, kde každý prvek bude typu int bude vypadat takto:

```
int pole[10];
```

Inicializace pole

Pole můžeme inicializovat jako každou jinou proměnnou při jejím vzniku. U pole je to ale o něco složitější. Musíme totiž určit hodnoty jednotlivých prvků.

```
int pole1[5] = {10, 20, 30, 40, 50};

int pole2[] = {10, 20, 30, 40, 50};

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

int pole4[5] = {0};

int poleNevim[5];
```

Na první řádce kompletně inicializujeme pole s pěti prvky. Na druhé řádce je ukázáno, jak vytvoříme ekvivalentní pole (všechny prvky mají stejnou hodnotu) s polem pole1 a přitom není uveden počet prvků v hranatých závorkách. To jazyk C umožňuje a počet prvků si dopočítá sám z hodnot uvedených ve složených závorkách.

Co se ale stane s polem pole3? Má 5 prvků, ale uvedeny jsou jen 2 hodnoty. V případě, že pole má více prvků než je počet hodnot uvedených ve složených závorkách, budou zbylé prvky pole (pro které už nejsou uvedeny hodnoty) vynulovány. U pole3 tedy „vyplníme“ pouze první dva prvky a zbytek bude vynulován. Stejně tak pole4 bude kompletně vyplněné nulami.

V případě poleNevim nemají prvky definovanou hodnotu. Některé překladače jazyka C takové pole vynulují, jiné nedělají nic a prvky budou mít neurčitou hodnotu (záleží na tom, co bylo dříve uloženo v paměti, než byla vyhrazena pro naše pole).

Práce s polem

Indexace

Máme-li vytvořenou proměnnou typu pole, můžeme k jednotlivým buňkám přistupovat pomocí operátoru indexace, nebo-li indexem v hranatých závorkách. Nesmíme ovšem zapomínat na to, že první buňka má index nula.

Mějme např.: pole:

```
int pole[5] = {15, 25, 35, 45, 55};
```

a vyzkoušejme následujícíc výpisy:

```
cout << pole[0]; // první buňka pole, vypíše se 15

cout << pole[1]; // druhá buňka pole, vypíše se 25

cout << pole[5]; // chyba, saháme mimo pole!! (indexy jsou 0-4)
```

Na poslední řádce se pokoušíme číst buňku s indexem pět. To je samozřejmě chybně, protože pole má poslední buňku na indexu čtyři. Jazyk C nemá žádnou zabudovanou kontrolu mezí polí. V případě jako je výše, dojde k potenciálně nebezpečnému čtení paměti. To může způsobit pád programu nebo náhodně vrácená data. Jazyk C prostě mechanicky vezme data z paměti tam, kde by ležel šestý prvek, kdybychom pole deklarovali jako šestiprvkové (šestý prvek = index 5).

Kromě určení indexu prvku konstantou (viz např. konstanta 1 v: `cout << pole[1];`) lze index určit i pomocí proměnné nebo výrazu. Na následující ukázce je opět pole 10ti prvků typu int deklarované takto:

```
int p[10];
```

Proveďmě postupně následující příkazy:

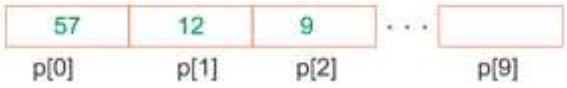
```
p[1] = 12;

int k = 2;
```

```
p[k] = 9;

p[k-2] = 45 + p[1];
```

Prostudujte výšeuvedené příkazy a ověřte, že hodoty prvních tří prvků budou dle obr. níže:



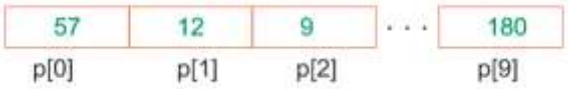
Určit index pole lze dokonce i hodnotou jiného prvku pole. V následujícím příkazu vidíme, že index je určen hodnotou prvku p[2]:

```
p[ p[2] ] = 180;
```

Dosadíme-li pomyslně do uvedeného příkazu za p[2] jeho hodnotu (v p[2] je 9) pak příkaz bude vypadat takto:

```
p[ 9 ] = 180;
```

A naše pole bude obsahovat:



Načtení do pole a výpis pole

Mezi základní operace s polem bezesporu patří načtení prvků do pole a výpis pole. Jeden z možných přístupů je na následující ukázce:

```
int i;
int p[10];
for(i = 0 ; i < 10 ; i++)
{
        cout << p[i];
}

for(i = 0 ; i < 10 ; i++)
{
        cout << "\t" << p[i];
}
cout << "\n";
```

Porovnávání polí

Pole nelze jednoduše porovnávat, jako to jde s jednoduchými číselnými typy. Od následujícího kódu nelze očekávat korektní chování, pokud nám jde o srovnání obsahu polí:

if (pole1 == pole2)

Chceme-li srovnat obsahy polí, nezbude nám nic jiného, než vytvořit cyklus a porovnat buňku po buňce. Máme-li stejně dlouhá pole, provedeme srovnání následovně:

```
int pole1[10], pole2[10];
// pokusně naplňte obě pole a) stejnými hodnotamy, b) různými
//a vyzkoušejte
int shoda = 1;
for (i = 0; i < 10; i++)
{
  if (pole1[i] != pole2[i]) // pole se liší
  {
    shoda = 0;
    break; // přerušíme cyklus a pokračujeme za ním
  }
}
if (shoda)
{
   printf("pole se rovnaji\n");
}
```

Součet a průměr

Další typickou úlohou prováděnou s polem je součet prvků, popř. průměr prvků pole.
V následujícím příkladu tedy máme proměnnou `soucet`, která má na začátku hodnotu 0 a následně do ní v cyklu přičítáme hodnoty jednotlivých prvků pole. Na konci vypisujeme průměr. Všimněte si přetypování při výpisu průměru (viz operátor / a celočíselné, reálné dělení).

```
    int i, soucet = 0;
    int p[6];
     //nacitame 6 hodnot z klávesnice a ukládáme je do pole
    for( i = 0 ; i < 6 ; i ++ )
    {
       cin >> p[i];
    }

    i = 0;
    for(  ; i < 6 ; i ++ )
    {
       soucet = soucet + p[i];
    }

    cout << "Prumer je " << (float)soucet / i << "\n";
```

Minimum – hlednání nejmenší hodnoty v poli
Nejjednodušší metodou hledání minima je postupně procházet celé pole prvek za prvkem a v nějaké pomocné proměnné si „pamatovat“ nejmenší hodnotu, kterou jsme zatím zaznamenali. Pokud při procházení prvků narazíme na nižší hodnotu, zapamatujeme si právě

ji a pokračujeme v procházení. Až dojdeme na konec pole, máme v pomocné proměnné uloženu nejmenší hodnotu, která se v poli vyskytuje.
Následující program představuje výše nastíněný popis. Důležitým detailem je počáteční nastavení proměnné `pamet`.
Ta se běžně nastavuje na hodnotu prvního prvku pole (zde `pole[0]`) a dále se pak polem prochází od druhého prvku. Tento postup je logický. Při postupném procházení polem je kontrolován nejprve první prvek, v danou chvíli je jediným prvkem, který jsme zatím „viděli“, tudíž jde o zatím nejmenší prvek, na který jsme v poli narazili.

```
int p[8];
int k;
for ( k = 0 ; k < 8 ; k = k + 1 )
{
    cin >> p[k];
}
cout << "\n\n";

int i;
int pamet = p[0];
for( i = 1 ; i < 8 ; i = i + 1 )//jdeme od druhého prvku
{
    if( p[i] < pamet)
    {
        pamet = p[i];
        poz = i;
    }
}
cout << "Minimum je " << pamet;
```

Další běžnou úlohou je kromě nalezení nejmenší hodnoty i vypsání indexu pole, kde se tato hodnota nachází. Upravme výše uvedený program tak, aby vypsal i index prvku, ve kterém je nejmenší hodnota. Pro tento účel doplníme do programu proměnnou `pozice`, ve které si budeme pamatovat index prvku s nejmenší hodnotou.

```
int p[8];

int k;
for ( k = 0 ; k < 8 ; k = k + 1 )
{
    cin >> p[k];
}
cout << "\n\n";

int i;
int pamet = p[0];
int pozice = 0; //pamatujeme si index
for( i = 0 ; i < 8 ; i = i + 1 )
```

```
{
    if( p[i] < pamet)
    {
        pamet = p[i];
        poz = i;
    }
}
cout << "Min. je " << pamet << " a je na indexu " << poz;
```

Pozor, pokud je např. nejmenší hodnota 1 a v poli je 2x (ve dvou různých prvcích), bude vypsán index prvního prvku obsahující hodnotu 1.

Nyní zkuste předělat výše uvedený program tak, aby místo minima, hledal maximum. Teprve pak se podívejte na další příklad, který řeší současně minimum i maximum.

```
int i;
int pole[8];

for ( i = 0 ; i < 8 ; i = i + 1)
{
    cin >> pole[i];
}
cout << "\n-----------------\n";

int max = pole[0], max_index = 0;
int min = pole[0], min_index = 0;

for( i = 0 ; i < 8 ; i = i + 1 )
{
    if( pole[i] > max )
    {
        max = pole[i]; max_index = i;
    }

    if( pole[i] < min )
    {
        min = pole[i]; min_index = i;
    }

}
cout << "maximum je "<<max<<" a index je "<<max_index;
cout << "\n minimum je "<<min<<" a index je "<<min_index;
```

Program není napsán záměrně efektivně, aby byl co nejsrozumitelnější. Zkuste jej upravit tak, aby se v cyklu „zbytečně“ nevyhodnocovaly vždy dvě podmínky.