Y36PJC Programování v jazyce C/C++

Pole, ukazatele, reference

Ladislav Vagner

Dnešní přednáška

- Staticky alokovaná pole.
- Vícerozměrná pole.
- Časté chyby při práci s poli.
- Ukazatel:
  - práce s ukazateli,
  - ukazatelová aritmetika,
  - vztah polí a ukazatelů.
- Reference.
- Vztah referencí a ukazatelů.

Minulá přednáška

- Funkce v C/C++:
  - parametry,
  - přetěžování,
  - implicitní hodnoty parametrů,
  - inline funkce.
- Preprocesor:
  - makra,
  - podmíněný překlad.

Pole

- Odvozený datový typ:
  - prostor pro **`n`** prvků stejného typu,
  - přístup k jednotlivým prvkům pomocí indexu.
- Typ prvku pole – libovolný, kromě:
  - referencí,
  - funkcí (ale mohou být ukazatele na funkci).
- Prvkem pole může být opět pole (vícerozměrná pole).
- Alokace pole:
  - statická, pokud je velikost známa v době překladu,
  - dynamická.
- Jak se liší C/C++ a Java v polích?

Jednorozměrná pole

Příklad deklarace:

```
int sample[100];
```

- Deklaruje pole o 100 prvcích typu **`int`**.
- Velikost pole musí být konstanta známá v době kompilace (dynamicky alokovaná pole - příště).
- Indexy prvků: 0 – 99 včetně.
- C/C++ nekontroluje rozsahy při indexaci.
- Zdroj možných, těžko odhalitelných chyb (přepisy paměti, pády programu, nedeterministické).
- Vždy si buďte jisti, že pracujete pouze s indexy v deklarovaných mezích !!!

Jednorozměrná pole

```
int        test[100];
const int SIZE_MAX = 200;
int        numbers[SIZE_MAX];
int        childs[2*SIZE_MAX + 80];
int        COUNT_MAX = 200;
int        other[COUNT_MAX]; // !! není konst
```

- Pole si nepamatuje svoji velikost, neexistuje možnost **`pole . length`** jako v Javě.
- Pro staticky alokovaná pole lze použít operátor **`sizeof`** (ale ne pro dynamicky alokovaná pole):

```
prvku = sizeof (test) / sizeof (test[0])
```

Jednorozměrná pole – časté chyby

```
void foo ( void )
 {
   int i, n;
   cin >> n;
   int pole[n];                    // !!
   for ( i = 0; i <= n; i ++ ) // !!
    cin >> pole[i];

   cout << "pozpatku";
   for ( i = n; i >= 0; i -- ) // !! znovu
    cout << pole[i];
   int kopie[n];                   // !! znovu
   kopie = pole;                   // !!
 }
```

Jednorozměrná pole – časté chyby

- Staticky alokovaná pole musí mít délku známou v době překladu:
  - nekonstantní délka není přenositelná,
  - přístup k poli nekonstantní délky může být pomalejší o 1 instrukci/přístup,
  - lokálně alokovaná pole nekonstantní délky jsou omezená velikostí zásobníku (např. Windows – std. limit je 1 MB).
- Indexem pole nelze překročit jeho velikost, a to ani pro čtení (zde 2x zpřístupněn prvek na indexu `n`).
- Staticky alokovaná pole mezi sebou nelze přiřazovat, musí se kopírovat prvek po prvku.

Jednorozměrná pole – časté chyby

```
void foo ( void )
 {
   int arr [100], cnt, i;

   cout << "Zadej pocet" << endl;
   cin >> cnt;
   for ( i = 0; i < cnt; i ++ )
    cin >> arr[i];
   …
 }
```

Jednorozměrná pole – časté chyby

- Alokujete-li pole podle minulého příkladu, dopouštíte se hrubé chyby:
  - program se chová správně pokud uživatel zadá číslo <= 100,
  - pro větší vstup se program chová, jako by pole mělo dostatečnou velikost,
  - program tedy přepíše paměť mimo alokované pole (chyba výpočtu, pád programu),
  - chyba typu buffer-overflow, častý cíl hackerských útoků.
- Záludnost chyby – v běžném testování se neprojeví, pouze pro extrémní velikosti vstupu.
- Obtížně se hledá a ladí.

Jednorozměrná pole – časté chyby

Odstranění buffer-overflow:

- Zvětšíme limit na 1000:
  - ale co vstup větší než 1000?
- Požádáme uživatele, aby nezadával více než 100:
  - co když nás hacker neposlechne?
- Zkontrolujeme rozsah, více než 100 omezíme shora:
  - ok, program nespadne, jen nebude fungovat pro všechny možné vstupy.
- Alokujeme pole dynamicky, s velikostí podle potřeby:
  - ok.

Jednorozměrná pole – časté chyby

```
void foo ( void )
 {
   const int MAX = 100;
   int arr [MAX], cnt, i;

   do {
     cout << "Zadej pocet" << endl;
     cin >> cnt;
   } while ( cnt < 1 || cnt > MAX );

   for ( i = 0; i < cnt; i ++ )   // ok
    cin >> arr[i];
   …
 }
```

Vícerozměrná pole

Příklad deklarace:

```
double matrix[4][5];
```

- Pole, kde prvkem je opět pole.
- **`matrix`** je pole o 4 prvcích, kde prvkem tohoto pole je 5-ti prvkové pole desetinných čísel.
- Všechny indexy od 0.
- Pole obdélníkové, neobdélníkové pole pouze dynamicky.
- Opět pozor na přepisy.

Vícerozměrná pole

Deklarace jinak:

```
typedef double TROW    [5];
typedef TROW   TMATRIX [4];

TMATRIX matrix1;
TROW    matrix2[4];
double  matrix3[4][5];
```

- **TROW** představuje vlastní datový typ – pole 5-ti desetinných čísel.
- **TMATRIX** je datový typ 2D pole 4x5 desetinných čísel.
- **matrix1**, **matrix2** a **matrix3** – proměnné stejné velikosti a struktury.

Vícerozměrná pole – časté chyby

```
double  matrix1[4,5];  // !! ,
double  matrix2[4][5];

matrix2[1,2] …  // <=> matrix2 [2]
matrix2[1][6] … // <=> matrix2 [2][1]
```

- Indexy nelze oddělovat čárkou (operátor zapomenutí).
- Překročení velikosti v jedné dimenzi může znamenat práci s úplně jiným prvkem téhož pole.
- Chyby s překročením dimenze se hledají ještě hůře než v 1D polích.

Pole – inicializace

- Staticky alokovaná pole:
  - lokální – nejsou inicializovaná,
  - globální – vyplněna nulami.
- Lze inicializovat při deklaraci:

```
int a1[5]    = {1, 2, 3, 4, 5};
int a2[]     = {9, 8, 7};              // bude a2[3]
int a3[10]   = {5, 4, 3};              // zbytek 0

int m1[2][3] = { {1, 2, 3}, {4, 5, 6}};
int m2[2][3] = { 0, 1, 2, 3, 4, 5 }; // po radcich
int m3[2][3] = { { 7 }, { 6 } };      // zbytek 0
```

Ukazatele

- Ukazatel - abstrakce adresy ve vyšším programovacím jazyce.
- Proměnná obsahuje adresu, na které se nachází zpřístupňovaná data:
  - proměnná,
  - objekt,
  - funkce,
  - jiný ukazatel.
- Operace s ukazatelem:
  - dereference – zpřístupnění místa, kam ukazatel směřuje,
  - reference – získání adresy paměťového místa (pořízení ukazatele).

Ukazatele

Deklarace proměnné typu ukazatel:

```
int * dataPtr;
```

- Vyhradí v paměti prostor pro adresu:
  - typ. 4B pro 32-bit prostředí,
  - typ. 8B pro 64-bit prostředí.
- Proměnná **`dataPtr`** je ukazatel, který může ukazovat na hodnotu typu **`int`**.
- Deklarací není nastavena adresa – **`dataPtr`** zatím ukazuje "někam do paměti".

Ukazatele – operace reference

```
int    iv, * iPtr;
double dv, * dPtr;
iPtr  = & iv;
dPtr  = & dv;
```

- Je–li **`X`** typu **`T`**, pak **`&X`** získá adresu **`X`**, adresa je ve formě ukazatele **`T*`**.
- Adresa může být uložena do proměnné - ukazatele stejného typu.
- Neshoda typů – lze přetypovat, ale neuvážené použití vede k problémům.

```
dPtr  = & iv;              // !!
dPtr  = (double *) & iv; // syntaxe ok, ale …
```

Ukazatele – operace dereference

```
int    iv, * iPtr = &iv;
double dv, * dPtr = &dv;
*iPtr = 10;   // iv = 10
*dPtr = 20.0; // dv = 20.0
```

- Je–li **`X`** ukazatel typu **`T*`**, pak **`*X`** zpřístupní místo v paměti typu **`T`**, kam ukazuje **`X`**.
- Obsahuje-li ukazatel neplatnou nebo nesmyslnou adresu – chyba programu (spadne, nesmyslný výsledek).

Ukazatele – časté chyby

Co je v kódu špatně ?

```
int main ( int argc, char * argv[] )
 {
   int deposit;

   cout << "Na Vasem konte je:"
        << deposit << " Kc" << endl;
   return ( 0 );
 }
```

Ukazatele – časté chyby

- Neinicializované proměnné způsobí zpravidla nesmyslný výsledek výpočtu.
- Neinicializované ukazatele vedou většinou k pádu programu.
- Zpřístupnění neinicializovaného ukazatele = náhodné čtení/zapsání někam do paměti.
- Program buď spadne (alespoň o chybě víme), v horším případě poskytne špatný výsledek, v nejhorším padá nebo špatně počítá jen občas.
- Neinicializované ukazatele se velmi špatně hledají a ladí.
- Vždy si buďte jisti, že každý ukazatel před použitím inicializujete.

Ukazatele – časté chyby

- Neinicializovaný ukazatel nikdy neobsahuje adresu volné paměti, kam se vejde tolik, kolik zamýšlíme.
- Jak inicializovat ukazatel:
  - přímo zadat adresu – typicky ne (pouze někdy pro specifické OS např. MS-DOS, pokud přesně víte, co děláte),
  - adresou jiné proměnné téhož typu v paměti,
  - z jiného inicializovaného ukazatele,
  - dynamickou alokací paměti.
- Nejprve inicializovat, teprve potom lze ukazatel dereferencovat.

Ukazatele a `const`

- Ukazatel může být konstantní (adresu nelze měnit).
- Ukazatel může být na konstantu (místo, kam odkazuje, nelze měnit).

```
int                x, y;
const int          z = 100;
int              * p0;        // bez omezeni
const int        * p1;        // na konstantu
int * const        p2 = &x;   // konstantni
const int * const p3 = &z;   // oboji
p0 = &z;      // chyba
p1 = &z;      // ok
p1 = &x;      // ok
cout << *p1; // ok
*p1 = 10;     // chyba
p2 = &y;      // chyba
p0 = p1;      // chyba, ale  p1 = p0; by bylo ok
```

Ukazatele - `void` a `NULL`

- Netypový ukazatel – **`void *`**:
  - nelze dereferencovat přímo, lze přetypovat na jiný ukazatel (víme-li, co děláme),
  - použití – pro uložení generických adres (např. při volání funkce OS pro zápis do souboru adresa bloku v paměti),
  - na **`void *`** se automaticky mohou konvertovat libovolné jiné typové ukazatele (opačně ne).
- Hodnota **`NULL`** – neplatná adresa.
- **`NULL`** odpovídá adrese 0.
- Dereference **`NULL`** ukazatele vede typicky k pádu programu (adresu 0 využívá OS).
- Používán pro signalizaci neplatného ukazatele.

Ukazatele a pole

- Staticky alokované pole má deklarací přiřazenu paměť.
- Staticky alokované pole zná svoji velikost, lze ji zjistit pomocí operátoru **`sizeof`**.
- Staticky alokované pole se umí implicitně převést na konstantní ukazatel na svůj počátek.
- Proto nelze pole navzájem přiřazovat.
- Práce s C/C++ polem = práce s ukazatelem.
- Pokud **`x`** je ukazatel nebo pole, platí:

```
x[i] = *(x+i) = *(i + x) = i[x];
```

- Pokud **`T`** je datový typ, pak:

```
T  arr[konst];
T * ptr = arr;  // ok, arr se
                // prevede na T * const
```

Ukazatele a pole

```
 int   z[20], *x;

 z[4]  =  1;  // z[4]  = 10;
*(z+10)=  2;  // z[10] = 2;

 x     = z;   // ok, z zde je  'int * const'
*x     = 10;  // z[0] = 10
*(x+1) = 20;  // z[1] = 20
 x[4]  = 30;  // z[4] = 30

 cout << sizeof ( z ); // 20*sizeof(int)
 cout << sizeof ( x ); // sizeof (int*)
```

Ukazatele – aritmetika

- K ukazatelům lze přičítat (odčítat) celočíselné hodnoty.
- Zvětšení ukazatele o **`n`** = ukazatel se tím posune na adresu o **`n*sizeof(typ)`** bajtů vyšší.
- Efektivně se takto lze pohybovat v poli hodnot stejného typu.
- Ukazatele lze odečítat – výsledkem je počet prvků mezi pozicemi v paměti, kam ukazatele směřují.
- Ukazatele lze porovnávat relačními operátory.

Ukazatele – aritmetika

```
 int * x;
 int   z[20];

 x     = z;  // ok
*x     = 10; // z[0] = 10
*(x+1) = 20; // z[1] = 20
 x[4]  = 30; // z[4] = 30
 x    += 2;
 x[0]  = 40; // z[2] = 40
*(x+3) = 50; // z[5] = 50
 x[-1] = 60; // z[1] = 60
 x[18] = 70; // chyba, mimo meze
 x--;
 x[18] = 10; // z[19] = 10;
```

Reference

- Datový typ, umožňuje vytvořit synonymum (jiné pojmenování, alias) již existující proměnné.
- Deklarace **`T & name =`** … **`.`**
- Součástí deklarace musí být inicializace proměnnou stejného typu.
- Reference uchovává po celou dobu své existence odkaz na zadanou proměnnou, odkaz nelze změnit.

```
int   x, array[100];
int   & r1 = x, & r2 = array[10];

r1 = 20;   // x = 20
r2 = r1;   // array[10] = x
```

Reference a ukazatele

- Nabízí podobnou funkcionalitu – zpřístupnění jiné proměnné.
- Ukazatel je obecnější:
  - lze měnit, kam odkazuje,
  - ukazatelová aritmetika,
  - má paměťovou reprezentaci (lze udělat ukazatel na ukazatel).
- Reference:
  - `T &` se trochu podobá `register T * const`,
  - ale reference se sama dereferencuje, nepíše se `*`.
- Použití referencí – hlavně parametry funkcí.
- Jsou reference v Javě referencemi či ukazateli?

Reference – použití

```
void normalize ( int & numerator,
                 int & denominator )
 {
   int cd = gcd ( numerator, denominator );
   numerator   /= cd;
   deniminator /= cd;
 }

int a = 20, b = 10, c = 5, d = 25;

normalize ( a, b );
normalize ( c, d );
```

Reference a ukazatele

```
void normalize ( int * numerator,
                 int * denominator )
 {
   int cd = gcd( *numerator, *denominator );
  *numerator   /= cd;
  *deniminator /= cd;
 }

int a = 20, b = 10, c = 5, d = 25;

normalize ( &a, &b );
normalize ( &c, &d );
```

Reference a `const`

- Zpřístupnění jiné proměnné pouze pro čtení.
- Obdoba ukazatelů na konstantu.

```
int         a   = 100;
const int   b   = 200;
int       & ir  = a;
const int & cir = b;
const int & x   = a;  // ok
int       & y   = b;  // chyba

ir   = 10;            // ok
cir  = 20;            // chyba
x    = 30;            // chyba
cout << ir << " " << cir << endl; // ok
```

Dotazy...

Děkuji za pozornost.