Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 13.06. 2002
Url: http://www.builder.cz/art/cpp/cpp_arraycp.html

Řízené kopírování prvků v poli v C++

V minulém článku jsme se seznámili se způsobem řízení kopírování velkých objektů. Uvedli jsme si jednu velkou nevýhodu celé techniky. Obecně ji nelze nijak parametrizovat. Velice často ale nastává případ, že objekt je veliký, protože jeho atributem je veliké pole. Taková byla i situace v ukázkovém příkladě v minulém článku . Takový speciální případ parametrizovat lze. Nejprve ale všem, kteří můj minulý článek "Kopírování velkých objektů v C++" nečetli, doporučuji jej přečíst. V tomto článku budu na předchozí článek navazovat.

Představme si velké pole. Podle návodu uvedeného v minulém článku jej rozdělíme na dvě části. Na část "obsahovou" a "přístupovou". Přístupovou část můžeme nazývat také "prezentační". Při kopírování pole vytvoříme pouze plytkou kopii. Pokud budeme z plytké kopie pouze číst, není důvod vytvářet hlubokou kopii. Hluboká kopie se vytvoří automaticky až při zápisu do prvku pole. Pro jednoduchost budeme při zápisu do jednoho prvku pole v případě potřeby vždy kopírovat celé pole, nejen daný prvek. O výhodách a nevýhodách si povíme v závěru.

Mohli by jsme si takhle vytvořit pole prvků nějakého typu. Poté jiné pole prvků jiného typu. Je zřejmé, že všechny třídy si budou podobné. Budou se lišit pouze typem uložených prvků. Jak asi mnohé napadne, vhodným řešením je šablona. Šablona se napíše jednou a podle potřeby se budou vytvářet její instance - třídy.

Nejprve vytvoříme šablonu obsahových tříd. Nazveme ji `__Atribut`. Šablona přístupových tříd se bude jmenovat `Atribut`. To proto, že instance šablony bude typ, jehož objekty budou atributy jiných tříd. Například v minulém článku jsme měli třídu s atributem pole typu `int`. `Atribut` možná není nejvhodnější název, ale mne žádný jiný nenapadl. Šablona `__Atribut` bude podle rad z minulého článku obsahovat čítač referencí, samotná data (pole) a délku pole. Dále metody, které manipulují s čítačem referencí, metody vracející ukazatel na první prvek v poli, konstruktory a destruktor.

```
template<class Type> class __Atribut
{
private:
  // Počet referencí a délka pole
  unsigned int ReferenceCount, Lenght;
  // Zapouzdřená data. Data jsou "velká".
  Type *Data;
public:
  // Konstruktory
  __Atribut():ReferenceCount(1),Lenght(0), Data(NULL) {}
```

```
__Atribut(unsigned int n):ReferenceCount(1),Lenght(n),
    Data(new Type[n]) {}
__Atribut(const __Atribut<Type> &original)
  :ReferenceCount(1),Lenght(original.Lenght)
{
  Data = new Type[Lenght];
  std::copy(Data,&Data[Lenght],original.Data);
  original.incrementReferenceCount();
}
// Destruktor
~__Atribut()
{
  delete[] Data;
}
// Operátor =
__Atribut<Type> &operator=(const __Atribut<Type> &original)
{
  if (Data != NULL)
  {
    delete[] Data;
  }
  Lenght = original.getLenght();
  Data = new Type[Lenght];
  std::copy(original.Data,&original.Data[Lenght],Data);
  return *this;
}
// Metoda getReferenceCount vrátí počet referencí
inline unsigned int getReferenceCount() const {
    return ReferenceCount;
   }
// Metoda decrementReferenceCount() sníží počet referencí
      a vrátí nový počet referencí
inline unsigned int decrementReferenceCount() {
   return --ReferenceCount;
   }
// Metoda incrementReferenceCount() zvýší počet referencí
      a vrátí nový počet referencí
inline unsigned int incrementReferenceCount() {
    return ++ReferenceCount;
   }
// Metoda getLenght() vrátí délku pole
inline unsigned int getLenght() const { return Lenght; }
```

```
  // Metoda getData vrátí ukazatel na data
  inline const Type *getData() const { return Data; }
  inline Type *getData() { return Data; }
};
```

Parametrem šablony je typ prvků, které budou v poli uloženy.

Přistupme k implementaci šablony přístupových tříd.

```
template<class Type> class Atribut
{
private:
  // Object je skutečné pole s čítačem referencí.
  __Atribut<Type> *Object;
  // Soukromé metody pro řízení kopírování
  void copy();
  void free();
public:
  // Konstruktory
  Atribut():Object(new __Atribut<Type>){}
  Atribut(unsigned int n):Object(new __Atribut<Type>(n)){}
  Atribut(const Atribut<Type> &original);
  // Destruktor
  ~Atribut();

  // Operátor =
  Atribut<Type> &operator=(const Atribut<Type> &original);

  //Operátory [] a metoda at pro přístup k prvkům
  inline const Type &operator[](unsigned int index) const;
  Type &operator[](unsigned int index);
  Type at(unsigned int index) const ;

  // Relační operátory
  bool operator==(const Atribut<Type> &atribut) const  ;
  bool operator!=(const Atribut<Type> &atribut) const  ;

  // Metoda vracející počet ukazatelů na objekt, na který
  // se odkazuje tento ukazatel. Vždy alespoň 1
  inline unsigned int getReferenceCount() const   ;
```

```
  // Metoda getPointer slouží k získání ukazatele na
  // první prvek pole.
  inline const Type *getPointer() const { return Object->getData();
  inline Type *getPointer() { copy(); return Object->getData();

  // Vytvoří nové pole. Slouží k změně rozměru pole.
  // Staré prvky v poli smaže. Nevytváří kopii!
  void newArray(unsigned int size);
};
```

Šablona třídy je v principu stejná jako přístupová třída z minulého článku. Obdobným způsobem zvyšuje a snižuje počet referencí. Soukromá metoda `copy` je zavolána vždy, je-li potřeba vytvořit kopii. Tedy jestliže je prováděna operace, která by změnila hodnotu nějakého prvku v poli. Naopak metoda `free` je zavolána v případě, že přístupový objekt "zahazuje" referenci na "obsahový" objekt.

Metoda `at` slouží k získání prvku v poli. Metoda `at` vrací samotný prvek, nikoliv referenci na něj. Proto by měla být použita jen na levé straně výrazu. Oproti operátoru `[]` neprovádí hlubokou kopii. Při použití operátoru `[]` (vrací referenci) je nutné provést hlubokou kopii. Nevíme totiž, jestli bude použit na pravé, či levé straně výrazu. Pod pojmy "na pravé straně výrazu" a "na levé straně výrazu" mám na mysli to, jestli je ve výrazu něco napravo nebo nalevo od operátoru =. Například `a = b;` je `a` na levé straně výrazu a `b` na pravé straně výrazu.

Kompletní implementace šablony třídy `Atribut` a `__Atribut` jsou v souboru atribut.h. Šablona je opět deklarovaná v prostoru jmen `www_builder_cz`.

Nyní si vytvoříme jednoduchý příklad použití šablony `Atribut`. Bude se jednat o matici, která bude mít řízené kopírování svých prvků. Bude kopírován pouze řádek, ve kterém došlo ke změně. Matice může mít mnoho prvků. Kopírovat se ale bude jen několik řádků, podle toho jaké prvky budeme měnit. Při velmi velkých rozměrech matice to bude výhodné. Ještě než se podíváme na zdrojový text třídy `Matice`, musíme si ujasnit, že šablona `Atribut` (resp. `__Atribut`) má jako svůj parametr typ , který má k dispozici bezparametrický konstruktor, kopírovací konstruktor a operátor =.

Kromě matice vytvoříme funkce `vypis` a `fce`. Funkce `vypis` vypíše prvky matice. Je dobré si všimnout, že matice předávaná této funkci jako parametr nebude vlastně nikdy kopírována. Při předávání parametru dojde k vytvoření pouze mělké kopie. Druhá funkce `fce` je ukázková a asi nemá žádné využití. Vynuluje zadaný řádek a ve vzniklé matici sečte zadaný sloupec.

```
#include <iostream>
#include "atribut.h"
using namespace std;

class Matice
{
```

```
private:
  www_builder_cz::Atribut<www_builder_cz::Atribut<int> > Data;
  unsigned int Radky, Sloupce;
public:
  Matice(unsigned int r, unsigned int s):Data(r),Radky(r),Sloupce(s)
  {
    for(unsigned int p = 0; p < r; p++)
    {
      Data[p].newArray(s);
    }
  }
  Matice(const Matice &druha):Data(druha.Data),Radky(druha.Radky),
     Sloupce(druha.Sloupce){}
  Matice &operator=(const Matice &druha)
  {
    Data = druha.Data;
    Sloupce = druha.Sloupce;
    Radky = druha.Radky;
    return *this;
  }
  inline unsigned int dejSloupec() const { return Sloupce; }
  inline unsigned int dejRadek() const { return Radky; }
  inline void nastavPrvek(unsigned int r, unsigned int s, int prvek)
     { Data[r][s] = prvek; }
  inline const int dejPrvek(unsigned int r, unsigned int s)
     const { return Data.at(r).at(s); }
  int sectiSloupec(unsigned int index);
};

int Matice::sectiSloupec(unsigned int index)
{
  int navrat = 0;
  for(unsigned int p = 0; p < Radky; p++)
  {
    navrat += Data.at(p).at(index);
  }
  return navrat;
}

int fce(Matice matice, int radek, int sloupec)
{
  for(unsigned int p = 0; p < matice.dejSloupec(); p++)
```

```
  {
    matice.nastavPrvek(radek,p,0);
  }
  return matice.sectiSloupec(sloupec);
}

void vypis(Matice m)
{
  for(unsigned int r = 0; r < m.dejRadek(); r++)
  {
    for(unsigned int s = 0; s < m.dejSloupec(); s++)
    {
      cout << m.dejPrvek(r,s) << "\t";
    }
    cout << endl;
  }
  cout << endl;
}

int main()
{
  Matice m1(5,8),m2(m1);
  m1.nastavPrvek(1,1,1);
  cout << m1.dejPrvek(1,1) << endl;
  m2.nastavPrvek(1,1,2);
  cout << m1.dejPrvek(1,1) << endl;
  cout << m2.dejPrvek(1,1) << endl;
  for(unsigned int r = 0; r < m1.dejRadek(); r++)
  {
    for(unsigned int s = 0; s < m1.dejSloupec(); s++)
    {
      m1.nastavPrvek(r,s,r+s);
      m2.nastavPrvek(r,s,(s+1)*(r+1));
    }
  }
  vypis(m1);
  vypis(m2);
  cout << m1.sectiSloupec(1) << " " << m2.sectiSloupec(1) << endl;
  for(unsigned int r = 0; r < m1.dejRadek(); r++)
  {
    cout << fce(m1,r,1) << " " << fce(m2,r,1) << endl;
  }
```

```
  return 0;
}
```

V této myšlence by se dalo pokračovat dále. Šlo by například vytvořit pole, které při zapsání libovolného prvku nekopíruje všechny prvky, ale pouze prvky, které se mění. Vždy ale musíme zvážit, jestli řízení kopírování nemá příliš velkou režii, která se nevyplatí. Jedná-li se o pole "velkých" objektů, může se to vyplatit. V našem případě, kdy jsou prvky v poli čísla `int` by se to nejspíše nevyplatilo.

V příštím díle náš seriál o programování v C++ ukončíme. Povíme si o klíčových slovech `mutable` a `static`. Jedná se o poměrně základní věc z C++. Já na ně ale v začátku seriálu nějak zapomněl.