Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 27.05. 2002
Url: http://www.builder.cz/art/cpp/cpp_copybigobj.html

Kopírování velkých objektů v C++

Dnes si ukážeme jak mít plně pod kontrolou kopírovaní instancí. Představme si situaci, kdy máme veliký objekt. Objekt je veliký například proto, že obsahuje veliké pole. Můžeme se dostat do situace, kdy je nutné vytvořit jeho kopii. Například jej musíme předat funkci, nebo metodě jako parametr. Kopírovat objekt je neefektivní. Ale předávat funkci, či metodě pouze ukazatel, nebo referenci není vždy možné. V těle funkce nebo metody můžeme chtít provádět s objektem nekonstantní operace, tedy operace které změní vnitřní stav objektu. Kdyby byla funkci nebo metodě předána jako parametr reference (nebo ukazatel), došlo by ke změně stavu i u originálu. Což může být nežádoucí.

Ideální by bylo, kdyby se objekt kopíroval pouze v případě, kdy je to potřeba, ne hned při předávání objektu jako parametru. A úplně nejideálnější by bylo, kdyby se objekt kopíroval jen v případě, kdy je to potřeba, a jen ta část objektu, kterou je potřeba zkopírovat. Jak na to?

Pro přesnost musím jen podotknout, že ke kopii vlastně dojde vždy. Řeší se zde, jestli má být kopie hluboká, nebo plytká. Pojmy hluboká a plytká kopie objektu jsme si vysvětlili ve článku Kopírovací konstruktor v C++. Je-li objekt velký (Například obsahuje velké pole), vytváření hluboké kopie je velmi náročné na čas (a také na paměť). Naopak plytká kopie je někdy nepoužitelná. Vytvoříme tedy třídu, jejíž objekty budou vždy kopírovány jako plytká kopie, ale v případě potřeby dojde dodatečně k vytvoření hluboké kopie. Budeme mít vlastně vytvořený objekt, který se bude kopírovat vždy jako plytká kopie. Až v případě potřeby se automaticky dodatečně vytvoří hluboká kopie.

Obvykle se volí postup, kdy se objekt vlastně rozdělí na dvě části. Na část "obsahovou" a "přístupovou". Je tedy nutné vytvořit dvě třídy. Třída "obsahových" objektů má jako své atributy data, jejichž kopírování chceme mít pod kontrolou. Instance této třídy obsahují "velká" data. Programátor by neměl mít na tyto objekty žádné reference, nebo ukazatele. Pracovat s nimi by měl pouze pomocí "přístupových" objektů. Obsahový objekt v sobě musí mít zapouzdřen čítač referencí. Musí vědět, kolik "přístupových" objektů se na něj odkazuje. "Obsahový" objekt se bude kopírovat pouze v nejnutnějším případě. Naproti tomu třída "přístupových" objektů bude jako svůj atribut nutně obsahovat ukazatel na jeden "obsahový" objekt. Přístupový objekt bude možné libovolně kopírovat, protože je malý. Programátor bude pracovat s "přístupovým" objektem.
Mějme úplně obyčejnou třídu, jejíž instance budou pravděpodobně zabírat velkou část paměti. Chceme ji předělat tak, aby jsme ušetřili zbytečné kopírování takové instance. Obecně lze doporučit postup:

- 1) Vyjmeme z naší třídy všechny atributy, které dělají její instanci velikou. Dáme je do jiné třídy, kterou můžeme například nazvat stejně, jenom s dvěmi podtržítky na začátku. Dejme tomu, že jsme měli třídu `Třída`, nyní máme třídy `Třída` a `__Třída`. `Třída` je třída přístupových objektů, `__Třída` je třída obsahových objektů. O existenci třídy "`__Třída`" a o jejích instancích nemusí programátor používající přístupovou třídu vůbec vědět.

- 2) Třídě `__Třída` přidáme soukromý atribut udávající počet existujících referencí na instanci. Nazvěme si jej například `ReferenceCount`. Bude typu `unsigned int`. Dále přidáme veřejné metody, které zvýší, sníží, vrátí počet referencí. Pojmenujme si je například `incrementReferenceCount, decrementReferenceCount, getReferenceCount`.
- 3) Třídě `__Třída` vytvoříme kopírovací konstruktor a operátor = tak, aby vytvářeli hlubokou kopii. Dále by měl být k dispozici pochopitelně destruktor, který uvolní paměť a také nějaké jiné konstruktory. Ve všech konstruktorech nastavíme výchozí počet referencí na 1.
- 4) Třídě `Třída` přidáme ukazatel na instanci typu `__Třída`. Nazvěme jej například `Objekt`. Tedy `Třída` má atribut `__Třída *Objekt`. Měl by být soukromý.
- 5) Třídě `Třída` přidáme metodu, která odregistruje objekt. Nejprve sníží počet referencí na objekt, na který se odkazuje ukazatel `Objekt`. Provede to pomocí metody `decrementReferenceCount`. Je-li po zavolání metody `decrementReferenceCount` počet odkazů na objekt 0, potom jej zničí destruktorem. Metodu nazveme například `free`. Neměla by být veřejná. Měla by být soukromá, nebo chráněná.
- 6) Třídě `Třída` přidáme metodu, která dodatečně provede kopírování instance `Objekt` do hloubky. Metodu můžeme nazvat například `copy`. Je-li počet referencí 1, kopie není potřeba a metoda se ukončí. V opačném případě zavoláme metodu `free` (uvolnění starého objektu) a poté vytvoříme hlubokou kopii objektu `Objekt` například pomocí kopírovacího konstruktoru třídy `__Třída`.
- 5) Třídě `Třída` vytvoříme kopírovací konstruktor a operátor =, které vytvoří jen plytkou kopii instance `Objekt`. Navíc zavolají objektu `Objekt` metodu `incrementReferenceCount`.
- 6) Třídě `Třída` vytvoříme destruktor, ve kterém zavoláme metodu `free`.
- 7) Všechny metody (kromě konstruktorů, destruktoru, operátoru =, metod `copy` a `free`) ze třídy `Třída` "přesuneme" do třídy `__Třída`.
- 8) Pro všechny metody, které jsme v bodě 7 přesunuli vytvoříme ve třídě `Třída` metody, které "přesměrují" volání na objekt `Objekt`. Jedná-li se navíc o metodu, která mění vnitřní stav objektu, zavoláme na jejím začátku metodu `copy`.
  Ukázka "přesměrování" metody, která nemění vnitřní stav objektu:

```
návratová_hodnota Trida::metoda(parametry)
{
      return Objekt->metoda(parametry);
}
```

  Ukázka "přesměrování" metody, která mění vnitřní stav objektu:

```
návratová_hodnota Trida::metoda(parametry)
{
      copy();
      return Objekt->metoda(parametry);
}
```

Uveďme si velice jednoduchý příklad. Vytvoříme velice jednoduchý příklad třídy, která bude zapouzdřovat velmi rozsáhlé pole.

```
#include<algorithm>
#include<numeric>

class Trida
```

```
{
  private:
     int *Pole;
  public:
     Trida() : Pole(new int[10000]) {}
     Trida(const Trida &original);
     ~Trida() { delete[] Pole; }
     Trida &operator=(const Trida &original);
     void nastavPrvek(int index,int hodnota){Pole[index]=hodnota;}
     int dejPrvek(int index) { return Pole[index]; }
     int secti() { return std::accumulate(Pole,&Pole[10000],0); }
};

Trida::Trida(const Trida &original):Pole(new int[10000])
{
   std::copy((int *)original.Pole,&original.Pole[10000],Pole);
}

Trida &Trida::operator=(const Trida &original)
{
   std::copy((int *)original.Pole,&original.Pole[10000],Pole);
   return *this;
}
```

Nyní si představme hodně zvláštní funkci, která jako svůj parametr bude mít objekt naší třídy a příznak. Je-li příznak nulový, funkce vrátí součet prvků v poli objektu naší třídy. Je-li příznak nenulový, přičte k prvním deseti prvkům příznak a vrátí součet prvků v poli.

```
int fce(Trida objekt, int priznak)
{
     if (priznak != 0)
     {
         for(int p = 0; p < 10; p++)
         {
            objekt.nastavPrvek(p,objekt.dejPrvek(p) + priznak);
         }
     }
     return objekt.secti();
}
```

Je zřejmé, že v této funkci nemůže být parametr `objekt` třídy `Třída` předáván referencí, nebo ukazatelem. Došlo by ke změně prvních deseti prvků i u originálního objektu. Na druhou stranu je zbytečné vytvářet na zásobníku kopii objektu v případě, že parametr `příznak` bude 0. Řešením je potlačit kopírování objektu. Dodržme postup, který jsem uvedl v osmi bodech a vytvořme nové dvě třídy. Výsledek bude vypadat takto:

```
#include<algorithm>
#include<numeric>

class __Trida
{
  private:
    int *Pole;
    unsigned int ReferenceCount;
  public:
    __Trida() : Pole(new int[10000]),ReferenceCount(1) {}
    __Trida(const __Trida &original):Pole(new int[10000]),ReferenceCount(1)
    {
        std::copy((int *)original.Pole,&original.Pole[10000],Pole);
    }
    ~__Trida() { delete[] Pole; }
    __Trida &operator=(const __Trida &original)
    {
        std::copy((int *)original.Pole,&original.Pole[10000],Pole);
        return *this;
    }
    inline void incrementReferenceCount() { ReferenceCount++; }
    inline void decrementReferenceCount() { ReferenceCount--; }
    inline int getReferenceCount() const { return ReferenceCount; }
    void nastavPrvek(int index, int hodnota) { Pole[index] = hodnota; }
    int dejPrvek(int index) { return Pole[index]; }
    int secti() { return std::accumulate(Pole,&Pole[10000],0); }
};

class Trida
{
  private:
     __Trida *Objekt;
     void free();
     void copy();
  public:
     Trida() : Objekt(new __Trida) {}
     Trida(const Trida &original):Objekt(original.Objekt)
     {
        Objekt->incrementReferenceCount();
     }
     ~Trida() { free(); }
```

```
    Trida &operator=(const Trida &original)
    {
       free();
       Objekt = original.Objekt;
       Objekt->incrementReferenceCount();
       return *this;
    }

    void nastavPrvek(int index, int hodnota)
    {
       copy();
       Objekt->nastavPrvek(index,hodnota);
    }
    int dejPrvek(int index) { return Objekt->dejPrvek(index); }
    int secti() { return Objekt->secti(); }
};

void Trida::free()
{
  Objekt->decrementReferenceCount();
  if (Objekt->getReferenceCount() == 0)
  {
     delete Objekt;
  }
}

void Trida::copy()
{
  if (Objekt->getReferenceCount() == 1)
  { // Kopie není potřeba
     return;
  }
  free();
  __Trida *temp = new __Trida(*Objekt);
  Objekt = temp;
}
```

Nyní je zřejmé, že k hluboké kopii, tedy ke kopírování velkého pole dojde pouze v těle metody `copy`. Ve zmiňované funkci `fce` nedojde ke kopírování, jestliže je parametr `příznak` roven 0.

V druhém příkladě bude programátor s třídou `Třída` zacházet normálně jako-by zacházel se třídou `Třída` v prvním příkladě.

V mnoha knihovnách se tato "technika řízeného kopírování" používá a programátor používající danou knihovnu o tom možná ani neví. Jen někde v

dokumentaci může být třeba napsáno, že objekty nějaké třídy používají řízené kopírování, kopírování až při potřebě nebo něco v tom smyslu. Mezi nevýhody patří zejména fakt, že se musí už při tvorbě třídy brát v úvahu fakt, že kopírování bude řízené. Je třeba vytvářet pomocné třídy a tím se zvyšuje množství zdrojového textu. Nelze vytvořit nějaký obecný program, který by dokázal předloženou třídu "přetransformovat" podle zmiňovaných osmi bodů na požadovaný výsledek. Stejně tak není možné vytvořit obecnou šablonu, které by jsme předložili "normální" třídu jako parametr a ona by nám vytvořila typ, který by měl řízené kopírování. Prostě vše musí udělat programátor ručně.

Velmi často se ale vyskytují ve třídách atributy, které jsou jednorozměrná pole (i v ukázkovém příkladu v tomto článku). Taková pole mohou být velice objemná. Tím se kopírování objektů ,obsahujících taková pole, stává velmi náročné. Zde by se dala vytvořit šablona, jejíž parametr by byl typ prvku v poli. Pole by se kopírovalo pouze v případě, že by se do pole zapisovalo. Pro čtení z pole není nutné provádět kopii. Takovou šablonu vytvořím a dám k dispozici ve svém příštím článku. Pomocí šablony, kterou předložím lze například pole rozdělit na několik částí. V případě zápisu do pole se provede kopie pouze té části, které se zápis týká. V příkladu, kde si ukážeme použití šablony vytvoříme matici, která bude při zápisu kopírovat pouze řádky, do kterých bude vepisováno.

Všem, které tento článek zaujal doporučuji přečíst si článek následující. Bude bezprostředně navazovat na tento článek.