NEEFEKTIVNÍ KONSTRUKCE V C++

Miroslav Virius
Fakulta jaderná a fyzikálně inženýrská ČVUT v Praze,
Trojanova 13, 120 00 Praha 2
virius@km1.fjfi.cvut.cz

ABSTRAKT:
V tomto příspěvku se zaměřuji na některé z konstrukcí v C++, které mohou způsobit snížení efektivity programu. Nejde o chybné konstrukce – výsledný program dělá přesně to, co od něj programátor očekává, a nejde také o použitý chybných nebo neefektivních algoritmů. Přesto tyto konstrukce mohou způsobit, že výsledný program běží pomaleji nebo s většími nároky na paměť, než je nezbytné.

KLÍČOVÁ SLOVA:
C++, efektivita, STL, konstruktor, přetěžování operátorů

ÚVOD

Programovací jazyk C byl navržen jako objektově orientované rozšíření jazyka C. Po celou dobu vývoje, od prvního návrhu na počátku osmdesátých let minulého století až po standardizaci v roce 1998, byla jedním z hlavních kritérií efektivita. To často vede k představě, že program musí být efektivní už jen proto, že byl napsán v C++.
Na druhé straně se lze stále sekat s představou, že objektově orientovaný program je neefektivní, *protože objekty prostě jsou neefektivní.*
Ani jedna z těchto představ není pravdivá. Přesto ovšem obsahuje jazyk C++ některé konstrukce, které mohou způsobit snížení efektivity programu; o nich si povíme v první části tohoto příspěvku. Ve druhé části se podíváme na některé mýty a pověry, které se týkají efektivity programů napsaných v C++.

NEEFEKTIVNÍ KONSTRUKCE

V určitých případech může (musí) překladač jazyka C++ automaticky zavolat některé metody objektových typů. To se týká konstruktorů, destruktorů a konverzních funkcí.

Předávání parametrů objektových typů hodnotou

Podívejme se na následující příklad funkce, která tiskne text zobrazený v okně:

```
void VytiskniObsah(Okno vokno)
{
   for (int i = 0; i < pocetRadku; i++)
   {
    // Zde se vytiskne obsah jednoho řádku textu z okna
   }
}
```

Předávání parametrů hodnotou je zpravidla vždy pomalejší než předávání odkazem, neboť znamená vytvoření lokální kopie parametru. V případě objektových typů ovšem přistupují ještě další dvě operace:
1. K vytvoření lokální kopie se použije kopírovací konstruktor, a

2. při ukončení funkce se pro tuto lokální kopii zavolá destruktor.
Jestliže tyto funkce obsahují netriviální operace, může jejich volání způsobit zpomalení běhu programu.

Vytvoření pomocné instance
Najde-li překladač jazyka C++ ve výrazu operand, pro který není daný operátor definován, pokusí se konvertovat ho na vhodný typ. K tomu může použít konverzní konstruktor (tj. konstruktor, který lze volat s jedním parametrem) nebo konverzní metodu `operator typ()`.
To ovšem znamená, že se vytvoří pomocná instance, a pro tu se později musí zavolat destruktor.
Nadbytečnému volání konstruktoru a destruktoru lze ovšem často zabránit vhodným přetížením odpovídajícího operátoru.
Podívejme se na příklad. V programu je deklarována třída `cplx` představující komplexní čísla:

```
class cplx
{
   public:
   cplx(double _re = 0, double _im  = 0)
        : re(_re), im(_im){}
   ~cplx(){}
   friend cplx operator+(cplx a, cplx b);
private:
   double re, im;
};

cplx operator+(cplx a, cplx b)
{
   return cplx(a.re + b.re, a.im + b.im);
}
```

Jsou-li `a` a `b` dvě instance takto definované třídy `cplx`, můžeme napsat

```
b = a + 1.0;
```

a program provede to, co očekáváme: sečte reálné a komplexní číslo. Přitom bude postupovat takto:

1. Nejprve zavolá konstruktor třídy `cplx` a vytvoří pomocnou instanci – komplexní číslo s hodnotou 1 + 0i. (Konstruktor třídy `cplx` lze volat s jedním parametrem, může tedy sloužit jako nástroj pro implicitní konverze.)
2. Pak použije přetížený operátor sčítání pro komplexní čísla.
3. Nakonec zavolá destruktor pomocné instance.

První a třetí krok je zbytečný. Jestliže přetížíme operátor sčítání také po dvojici operandů obsahující reálné a komplexní číslo, např. takto,

```
inline cplx operator+(double a, cplx b)
{
     return cplx(a + b.re, b.im);
}
```

```
inline cplx operator+(cplx a, double b)
{
     return b + a;
}
```

tyto kroky odpadnou.

Vytvoření pomocné instance může způsobit použití pomalejšího algoritmu
V některých situacích může vytvoření pomocné instance dokonce způsobit, že program použije méně efektivní algoritmus. Podívejme se na příklad. Vezmeme třídu `matice`, která je deklarována takto:

```
class matice
{
public:
     matice(double d = 0);         // Konverzní konstruktor
     ~matice();                    // Destruktor
     double* operator[](int i);    // Operátor indexování
     matice nasob(matice &m);      // Násobení matic
     enum {N = 2};                 // Velikost matice
private:
     double* pole;                 // Pole velikosti N*N
     void nuluj();                 // Nulování pole
     void alokuj();                // Alokace paměti
};
```

Konstruktor této třídy vytvoří `d`-násobek jednotkové matice (matici, které má na diagonále hodnotu `d` a mimo diagonálu nuly). Lze ho volat s jedním parametrem, takže může sloužit i jako nástroj pro implicitní konverzi z typu `double` na typ `matice`.
Metoda `nasob()` implementuje obvyklý algoritmus násobení dvou matic,

```
matice matice::nasob(matice& m)
{
     matice pom;
     for(int i = 0; i < N; i++)
     for(int j = 0; j < N; j++)
     for(int k = 0; k < N; k++)
          pom[i][j] += (*this)[i][k]*m[k][j];
     return pom;
}
```

Operátor `*`, deklarovaný jako volná funkce, se na tuto metodu odvolává:

```
matice operator*(matice a, matice b)
{
     return a.nasob(b);
}
```

I tentokrát můžeme napsat

```
t = m * 4;
```

Přestože třída `matice` neobsahuje operátor pro násobení matice a čísla, dostaneme správný výsledek – to známe již z předchozího oddílu. Překladač totiž druhý operand, číslo `2`, pomocí konverzního konstruktoru převede na typ `matice`, tj. vytvoří pomocnou instanci tohoto typu, a získané matice pak vynásobí. Nakonec zavolá destruktor pomocné instance.
Ovšem násobení matice velikosti `N` × `N` číslem je operace, jež vyžaduje N^2 operací (každý prvek matice vynásobíme daným číslem), zatímco násobení dvou matic je operace, jež vyžaduje N^3 operací. *Zde tedy vede vytvoření pomocné instance k použití o řád pomalejšího algoritmu.*
Řešením je samozřejmě definovat přetížený operátor násobení, jež bude mít jeden operand typu `matice` a druhý typu `double`. Vedle úspory operací při násobení odpadne také volání konstruktoru a destruktoru pomocné instance.

Poznámka
Deklarace operátoru indexování ve třídě `matice` může na první pohled vypadat podivně, je ale správná. Tento operátor umožňuje zacházet s jednorozměrným polem délky `N*N` jako s dvourozměrným polem s `N` řádky a `N` sloupci. Má tvar

```
double* matice::operator[](int i)
{
     return this->pole + N*i; // posun o i řádků
}
```

a vrátí ukazatel na prvek `pole`, který představuje začátek i-tého řádku matice. Protože tento operátor vrací ukazatel, můžeme na jeho výsledek aplikovat další (tentokrát standardní) operátor indexování. To znamená, že je-li `m` instance třídy `matice`, můžeme napsat `m[i][j]` a dostaneme opravdu odpovídající prvek matice `m`.

Optimalizace návratové hodnoty
Podívejme se na následující dvě varianty operátoru sčítání pro komplexní čísla:

```
cplx operator+(cplx a, cplx b)        // 1
{
     return cplx(a.re + b.re, a.im + b.im);
}

cplx operator+(cplx a, cplx b)        // b
{
     cplx pom(a.re + b.re, a.im + b.im);
     return pom;
}
```

Je mezi nimi nějaký rozdíl? V obou případech se vytvoř pomocná proměnná a ta bude vrácena (pomocí kopírovacího konstruktoru bude překopírována zpět).
Ve skutečnosti je zde ještě jeden rozdíl. Jsou-li `a`, `b` a `c` instance třídy `cplx` a napíšeme-li

```
c = a + b;
```

může ve druhém případě překladač optimalizovat program tak, že se výsledek zkonstruuje přímo na místě proměnné `c`, tedy výsledku. To znamená, že odpadne konstrukce a destrukce lokální pomocné proměnné, konstrukce a destrukce vrácené proměnné (lokální v těle volající funkce) a její přiřazování.
V prvním případě to není možné: Překladač nesmí při optimalizaci odstranit pojmenovanou instanci.

Ukládání dat do třídy `vector<>` a jiných kontejnerů

Třída `vector<>` je součástí standardní šablonové knihovny jazyka C++. Programátorům nabízí „nafukovací" pole, tedy pole, jehož velikost se v případě potřeby po přidání nového prvku pomocí metody `push_back()` automaticky zvětšuje.
Podívejme se na příklad použití:

```
vector<int> data;
// ...
for(int i = 0; i < N; i++)
     data.push_back(i);
```

Co je na tom špatně? Pokud jde o smysl programu, je vše v pořádku; program ale nebude příliš efektivní.
Implicitní konstruktor vytvoří kontejner s kapacitou 0 prvků. Při vložení prvního prvku se kapacita zvýší na 1 prvek. Pokaždé, když bude třeba vložit prvek, pro který již není místo, kapacita vektoru se zvětší – zpravidla se zdvojnásobí.
Zvětšení kapacity ovšem znamená, že se

1. alokuje nová paměť,
2. obsah původní paměti se do ní překopíruje,
3. původní paměť se uvolní,
4. do volného místa se vloží nový prvek.

Jestliže známe alespoň přibližně počet prvků, které budeme do kontejneru vkládat, lze předejít zbytečným alokacím tak, že konstruktoru zadáme požadovanou kapacitu, tedy počet vkládaných prvků:

```
vector<int> data(N);
```

Třída vector<> a paměť

Třída `vector<>` sice obsahuje metody `erase()` a `clear()`, jež umožňují odstranit data z kontejneru, ale ty nezpůsobí uvolnění paměti. Jinými slovy, jakmile si instance alokuje paměť, už se jí dobrovolně nevzdá. V určitých situacích ale nechceme plýtvat pamětí a přitom předem neznáme potřebný počet údajů.
Pak se nabízí následující možnost:

1. Pomocí kopírovacího konstruktoru třídy `vector<>` vytvoříme pomocnou proměnnou – kopii naší instance.
2. Pomocí metody `swap()` prohodíme alokovanou paměť těchto dvou instancí.

Například takto:

```
vector<int>(data).swap(data);
```

Kopírovací konstruktor alokuje pouze tolik paměti, kolik je nezbytně třeba. Tím, že vytvoříme nepojmenovanou proměnnou, způsobíme, že tato pomocná proměnná v zápětí zanikne.

POVĚRY

Vedle představy o zaručené efektivitě programu napsaného v C++ se lze setkat s představami, že některé konstrukce jsou nesmírně neefektivní. Podívejme se nejčastější z nich.

Je přetěžování operátorů neefektivní?

Použití přetíženého operátoru přeloží překladač jako volání odpovídajících operátorových funkcí. To znamená, že zápis pomocí přetížených operátorů vede ke stejně efektivnímu programu jako použití odpovídajících „obyčejných" funkci, pouze zdrojový kód je (při rozumném používání těchto operátorů) podstatně přehlednější.

Je použití přetěžování funkcí neefektivní?

Přetěžování funkcí bývá občas chápáno jako forma polymorfizmu, a to vede k představě, že je neefektivní.

O tom, která z přetížených funkcí se má zavolat, ovšem rozhoduje překladač; v přeloženém programu jsou už volání vyřešena a přeložený program s přetíženými funkcemi se v zásadě neliší od programu, v němž se přetěžování nepoužívá. (Lze si představit, že překladač nahradí stejná jména jmény navzájem různými, tedy jednoznačnými.)

Na efektivitu výsledného programu tedy nemá přetěžování žádný vliv.

Je používání šablon neefektivní?

Také šablony zpracovává v C++ překladač (na rozdíl od C# a některých jiných jazyků). To znamená, že existují pouze ve zdrojovém kódu; přeložený program obsahuje pouze jejich instance, tedy třídy, metody nebo funkce, které překladač na základě šablon vytvořil. (Z tohoto hlediska se šablony neliší od maker preprocesoru generujících zdrojový kód.)

Používání šablon tedy v principu zatěžuje překladač, nikoli běžící program. Je ale třeba mít na paměti, že některá rafinovaná použití šablon – např. šablonové metaprogramování – mohou snížit srozumitelnost programu a programátor pak snáze vytvoří některou z neefektivních konstrukcí uvedených výše.

Je použití implicitních hodnot parametrů neefektivní?

Také o dosazení implicitních hodnot parametrů rozhoduje překladač; v přeloženém programu jsou již všechny parametry dosazeny. To znamená, že podobně jako použití přetížených funkcí, i použití implicitních hodnot parametrů zatěžuje překladač, nikoli přeložený program.

Jsou virtuální metody neefektivní?

Mechanizmus volání virtuálních metod v C++ je všeobecně znám:

1. V instanci polymorfního typu je (zpravidla na počátku instance) uložen ukazatel na tabulku virtuálních metod.
2. Pomocí tohoto ukazatele program nalezme tabulku virtuálních metod daného typu.
3. V ní vyhledá adresu volané funkce a tu zavolá.

To sice vypadá složitě, ale ve skutečnosti to na PC znamená pouze jednu instrukci navíc. Například volání nevirtuální metody

```
t -> g();                    // nevirtuální metoda
```

může být přeloženo do asembleru takto,

```
mov  ecx,dword ptr [t]       // Předání ukazatele this
call cplx::g (419A64h)       // Vlastní volání
```

zatímco volání virtuální metody

```
t -> f();                       // virtuální metoda
```

se přeloží takto:

```
mov  eax,dword ptr [esp+4]   // Zjištění ukazatele na TVM
lea  ecx,[esp+4]             // Předání this
call dword ptr [eax+8]       // Vlastní volání
```

(Uvedený kód jsem získal pomocí disasembleru Visual Studia 2003 od Microsoftu.)
Zde přibyla jediná instrukce v asembleru. I když instrukce pro vlastní volání nejsou totožné, je zřejmé, že rozdíl bude ve naprosto převážné většině případů zanedbatelný.

ZÁVĚR

Programovací jazyk C++ opravdu může vést k velice efektivním programům. Je ale třeba znát situace, které mohou způsobit problémy.

LITERATURA

1. M. Virius: Pasti a propasti jazyka C++. 2. vydání, Computer Press, Brno 2004. ISBN 80-251-0509-1

ABSTRACT

The C++ programming language is usually considered to be very efficient; nevertheless, there are C++ constructs that may produce inefficient programs. This article shows some of them. First is discussed the effect of implicit conversions of class types, that brings the overhead of additional constructor and destructor call. In some cases, it may even lead to the use of worse algorithms. Next we discuss the return value optimization. Last case, discussed in this article, is the use of the vector template – the effect of improper initialization and the reduction of the used memory.
Some superstitions concerning the C++ efficiency – the operator and function overloading, the use of virtual methods and the use of C++ templates – are also discussed.