Šablona třídy a deklarace friend

V tomto příspěvku si povšimneme záležitostí, které prošly tak bouřlivým vývojem, že při tvorbě standardu byly několikrát měněny a nakonec došlo i na změnu jejich sémantiky, takže chyb při programování je dost a dost. Snad se teď alespoň některým vyhnete...

Kde se vzala složitost C++

Někde na počátku byl jazyk C. Jazyk to byl dobrý, používá se dodnes. Byl ale poznamenán dobou svého vzniku, kdy objektově orientované programování teprve čekalo na své objevení. Právě o ně se "céčko" rozhodl rozšířit Bjarne Stroustrup - vznikl jazyk C++. Ale ani to zanedlouho nestačilo a došla řada i na generické programování. A tak C++ rostlo, rostlo, až svým tvůrcům trochu přerostlo přes hlavu (o programátorech, kteří tento jazyk používají, ani nemluvě). Dokladem může být jeho mezinárodní standard ISO/IEC 14882:1998 - více než 700 stran plných složitých a někdy téměř nepochopitelných pravidel mluví samo za sebe. Není se co divit, že i tvůrci překladačů mají co dělat, aby správně pochopili a implementovali požadované vlastnosti.

Proč taková složitost? Odpověď není tak těžká. Pokud mají vedle sebe existovat různá (ale opravdu různá) programovací paradigmata, je třeba ošetřit jejich "styčné plochy". Vezměme třeba objektově orientované programování a generické programování. První představuje znovupoužitelnost na úrovni binárních komponent, kdežto druhé znovupoužitelnost na úrovni zdrojového kódu, tedy něco zcela odlišného. Jsou-li pak obě součástmi jednoho programovacího jazyka, je kvůli konzistenci nutné zavést spoustu pravidel regulujících vzájemný vztah obou paradigmat. A k tomu C++ ještě navíc podporuje procedurální (strukturované) programování, šablonové metaprogramování a kdoví co ještě - asi už je jasné, kde se vzalo těch 700 stran a proč je to tak složité.

V této souvislosti je vhodné zmínit jazyky Java a C#. Zde šli autoři jinou cestou. Prostě zvolili programovací paradigma, které je zrovna (komerčně) nejúspěšnější, a ta ostatní buď vypustili úplně (jako se to stalo šablonám), nebo omezili do patřičných mezí (třeba unsafe kód v C#). Proto jsou tyto jazyky "průhlednější" než C++. (Je to samozřejmě pohled trochu zjednodušený; víc se dozvíte např. v [4].)

Ale pojďme zpět k C++ a proberme si jednu z jeho "lahůdek" - střet hned tří paradigmat: generického programování (šablona třídy, funkce), objektově orientovaného programování (deklarace friend) a procedurálního programování (obyčejná funkce). Tato část byla v průběhu standardizace několikrát změněna, přičemž nešlo jen o přidání nové syntaxe, ale i o změnu významu některých dříve používaných konstrukcí. Sluší se proto upozornit ty, kdo používají volně dostupnou pracovní verzi standardu z roku 1996, že tato část je ve finální verzi změněna.

Něco je zase jinak...

Výklad začneme trochu netradičně častou chybou, která je důsledkem změny sémantiky deklarace friend. Tuto chybu nejdete i v autorově "typické" implementaci vektoru v článku o šablonách výrazů [3]. Přitom ještě před několika lety by bylo vše v pořádku. Za toto "nedopatření, ke kterému dochází nejvýše jednou za 10 let..." se autor čtenářům omlouvá.

Nejdříve si připomeňme onu zmiňovanou typickou implementaci vektoru:

```
template <class T>
class Vektor
{
public:
  // ...
friend
  Vektor<T> operator +(
    const Vektor<T> & v1,
    const Vektor<T> & v2);
private:
  T * data_;
  int size_;  // velikost
};
```

```
// šablona globálního operátoru +
template <class T>
Vektor<T> operator +(
   const Vektor<T> & v1,
   const Vektor<T> & v2)
{
  Vektor<T> pom(v1.size_);
  for (int i = 0; i < pom.size_; ++i)
  {
    pom.data_[i] =
       v1.data_[i] + v2.data_[i];
  }
  return pom;
}
```

Konstruktor, destruktor a ostatní metody jistě dokážete doplnit sami. Nezapomeňte na kopírovací konstruktor, ten zde hraje důležitou roli - provádí tzv. hlubokou kopii (deep copy), která je nutná pro správnou funkci např. operátoru sčítání (příkaz return pom;).

Tento kód je syntakticky v pořádku - to znamená, že jej lze bez potíží přeložit. Problém nastane, když někde dál budeme chtít spřátelený operátor sčítání použít:

```
Vektor<double> a;
Vektor<double> b;
// ...
a + b;    // !
```

Ve starších překladačích (např. Borland C++ 5.0) to projde bez povšimnutí a vše funguje. Nové překladače, které se řídí standardem ISO jazyka C++ (např. Borland C++ Builder 4.0 nebo 5.0), kód sice přeloží, ale nedokáží ho slinkovat. Objeví se chybové hlášení o tom, že linkovací program nemůže nalézt operátor

```
Vektor<double> operator+(
   const Vektor<double> &,
   const Vektor<double> &);
```

Na první pohled to vypadá jako chyba překladače. Co je tedy jinak? Abychom na to dokázali odpovědět, musíme nejdříve podniknout výzkumnou výpravu do oněch 700 stran standardu C++. Zajímat nás bude význam deklarace friend, a to v souvislosti se šablonami.

Cesta do hlubin standardu C++

Zde jsou základní pravidla (standard [1], sekce 14.5.3.1):

Přítelem třídy nebo šablonové třídy může být šablona třídy či funkce, specializace šablony třídy či funkce nebo obyčejná (nešablonová) funkce či třída. Pro deklaraci spřátelené funkce, která není šablonovou deklarací, platí:

[1] jestliže jméno spřátelené funkce je kvalifikované nebo nekvalifikované id-šablony, pak deklarace odpovídá dané specializaci šablonové funkce, jinak

[2] jestliže jméno spřátelené funkce je kvalifikované a příslušná nešablonová funkce je nalezena ve specifikované třídě nebo prostoru jmen, pak deklarace odpovídá této funkci, jinak

[3] jestliže jméno spřátelené funkce je kvalifikované a ve specifikované třídě nebo prostoru jmen je nalezena odpovídající specializace šablonové funkce, pak deklarace odpovídá této specializaci, jinak

[4] musí jít o nekvalifikované jméno, které deklaruje (nebo re-deklaruje) obyčejnou (nešablonovou) funkci.

Abychom se v takovéhle hrůze mohli vyznat, nejdříve si trochu objasněme terminologii.

* Vysvětlení pojmu specializace šablony naleznete v článku [2].

* Jména mohou být kvalifikovaná nebo nekvalifikovaná. Operátor kvalifikace je :: ("čtyřtečka"). Zjednodušeně řečeno, pokud je součástí jména operátor ::, jde o kvalifikované jméno. Příklady:

```
neco       // nekvalifikované jméno
::neco     // všechna ostatní jsou kvalifikovaná
MojeTrida::neco
ProstorJmen::neco
ProstorJmen::MojeTrida::neco
```

* id-šablony je jméno šablony následované seznamem šablonových argumentů v lomených závorkách. Pokud mají některé šablonové parametry deklarovány implicitní argumenty, je možné je vynechat (podle analogických pravidel jako u implicitních argumentů funkcí); nesmíme však vynechat lomené závorky - i když jsou prázdné. Příklady:

```
neco<T>  // nekvalifikované id-šablony
::MojeTrida<double, 3>
ProstorJmen::neco<>
```

A teď prakticky...

Nyní si projděte uvedená pravidla ještě jednou a pokuste se vyřešit následující úkol. Máme několik deklarací funkcí označených (1) až (5) a šablonovou třídu Trida, ve které jsou deklarace spřátelených tříd a funkcí. Pokuste se vysvětlit, co jednotlivé deklarace znamenají, a v případě deklarací spřátelených funkcí určit, která z deklarací (1) až (4) je jimi myšlena, případně podle kterého z pravidel [1] až [4] bylo rozhodnuto.

```
template <class T> class Trida;

class MojeTrida;
template <class U> class MojeSablTrida;

template <class T> void f(Trida<T> &);  // (1)
void f(Trida<int> &);  // (2)
template <class T> void g(Trida<T> &);  // (3)
template <class T> void h(T); // (4)
void h();  // (5)

template <class T> class Trida
{
  // spřátelené třídy
  friend class MojeTrida;   // (a)
  template <class U>
    friend class MojeSablTrida;  // (b)
  friend class Trida<int>;  // (c)

  // spřátelene funkce f
  friend void f<T>(Trida<T> &);  // (d)
  // friend void ::f<T>(Trida<T> &);  // (e)
  friend void ::f(Trida<T> &);  // (f)
  friend void f(Trida<T> &);  // (g)

  // spřátelene funkce g
  friend void g<T>(Trida<T> &);  // (h)
  // friend void ::g<T>(Trida<T> &);  // (i)
  friend void ::g(Trida<T> &);  // (j)
  friend void g(Trida<T> &);  // (k)

  // ten zbytek
  friend void h();   // (l)
  template <class U> friend void h(U); // (m)
};
```

Řešení:

(a) Deklarujeme spřátelenou třídu MojeTrida pro všechny specializace šablonové třídy Trida.

(b) Deklarujeme spřátelenou šablonovou třídu MojeSablTrida. Všechny specializace šablonové třídy MojeSablTrida budou spřátelené s jakoukoliv specializací šablonové třídy Trida.

(c) Každá specializace šablonové třídy Trida bude mít spřátelenou třídu Trida<int>.

(d) Deklarace spřátelené funkce; použijeme tedy výše uvedená pravidla. Jedná se o nekvalifikované id-šablony, použije se proto pravidlo [1] a deklarace (1). Znamená to tedy jisté provázání šablonových argumentů: funkce f<T> je spřátelená s třídou Trida<T>.

(e) Stejně jako (c) s jediným rozdílem - jde o kvalifikované id-šablony.

(f) Zde je situace složitější. Musíme rozlišit dva případy. Jestliže T je int, pak podle pravidla [2] je použita nešablonová funkce (2). Pokud T není int, pak podle pravidla [3] jde o funkci (1). (Vzhledem ke zkušenostem s běžnými překladači berte toto zatím jako sci-fi.)

(g) Nekvalifikované jméno; podle pravidla [4] je to nešablonová funkce f. Jde o obyčejnou (nešablonovou) funkci, jejíž parametry přímo závisí na šablonových parametrech třídy Trida. To znamená, že deklarace (2) pokrývá pouze jednu možnost, kdy T je int. Kdyby T byl jakýkoliv jiný typ, museli bychom

dodefinovat příslušnou nešablonovou funkci.

(h) Analogicky jako v bodu (c); pravidlo [1] určí deklaraci (3).

(i) Analogicky jako v bodu (d); pravidlo [1] určí deklaraci (3).

(j) Kvalifikované jméno; není nalezena žádná vhodná nešablonová funkce, takže podle pravidla [3] se jedná o deklaraci (3), tj. použije se specializace šablonové funkce.

(k) Nekvalifikované jméno; podle pravidla [4] je to nešablonová funkce g (dokonce její deklarace). Taková funkce tu však není definována. Přesto nemusí jít o chybu - stejně jako v případě (g) deklarace friend říká, že pokud taková funkce existuje, pak je spřátelená - nic víc. Stejně jako v případě (f) je zde závislost parametrů funkce na šablonových parametrech třídy Trida.

(l) Spřátelenou funkcí je obyčejná řadová funkce; pravidlo [4], deklarace (5). Jelikož parametry této funkce vůbec nesouvisí se šablonovými parametry třídy Trida, je tato funkce spřátelená s jakoukoliv specializací třídy Trida.

(m) Deklarujeme spřátelenou šablonovou funkci h; deklarace (4). Zde se pravidla [1] až [4] nepoužívají, neboť jde o šablonovou deklaraci. Všechny specializace šablonové funkce h jsou spřátelené s jakoukoliv specializací šablonové třídy Trida.

Co je ještě dobré vědět

Deklarace spřátelené funkce, která není zároveň šablonovou deklarací a ve které je jméno funkce nekvalifikované id-šablony, musí odkazovat na šablonu funkce v nejbližším nadřízeném prostoru jmen.

```
template <class T> void f(T);
void g(int);  // alias ::g

namespace N
{
  template <class T> void h(T);

  class A
  {
    // ...
    friend void f<>(int);  // nelze
    friend void h<>(int);  // OK, N::h
    friend void g(int); // OK,
    // deklarace funkce N::g, nikoli ::g
  };
}
```

Všimněte si také, že deklarace spřátelené funkce g (ve skutečnosti N::g) nemá s globální funkcí g nic společného.

Jako přítele můžeme deklarovat šablonu funkce - případ (m), nebo třídy - případ (b). Je zde jeden rozdíl: příslušnou funkci smíme definovat uvnitř třídy (tedy na místě deklarace friend), kdežto spřátelenou třídu ne.

```
class A
{
  template <class T> friend void f(T) { /*...*/ }  // OK
  template <class T> friend class B;  // pouze deklarace
};
```

Šablona funkce či třídy nemůže být přítelem lokální třídy, tj. třídy definované uvnitř nějaké funkce.

Pokud deklarace spřátelené funkce odkazuje na specializaci šablonové funkce, nesmí se v deklaraci parametrů vyskytnout implicitní argumenty ani nesmí být použito klíčové slovo inline.

Jak to tedy bylo

Nyní se vrátíme k našemu problému z úvodu článku. Ve světle nových poznatků můžeme říci, že deklarace friend deklaruje řadovou funkci, nikoli specializaci šablony; viz pravidlo [4]. To znamená, že překladač nepoužije námi nabízenou šablonu operátoru + k vytvoření specializace a očekává, že si potřebnou operátorovou funkci napíšeme sami. To jsme však neudělali, a proto ji linkovací program nenašel. Museli bychom dodefinovat řadovou nešablonovou funkci

```
Vektor<double> operator+(
   const Vektor<double> &,
   const Vektor<double> &);
```

Ale co když použijeme Vektor<int>? Nastane zase ta samá chyba. Museli bychom ještě napsat operátorovou funkci

```
Vektor<int> operator+(
   const Vektor<int> &,
   const Vektor<int> &);
```

A tak dále pro všechny možné typy... Je jistě jasné, že takhle jsme to nechtěli. Musíme proto provést několik úprav. Pro snazší orientaci budou tyto úpravy v dalším textu zvýrazněny.

Řešení 1

Provedeme propojení šablonových argumentů použitím id-šablony v deklaraci friend.

```
// předběžné deklarace
template <class T> class Vektor;

template <class T>
Vektor<T> operator +(
   const Vektor<T> & v1,
   const Vektor<T> & v2);

// upravená třída Vektor
template <class T>
class Vektor
{
public:
  // ...
friend
  Vektor<T> operator +<T>(
    const Vektor<T> & v1,
    const Vektor<T> & v2);
private:
  T * data_;
  int size_;  // velikost
};
```

Šablona operátoru + zůstane beze změny. Rozdíl je tedy v tom, že při deklaraci spřátelené funkce uvedeme explicitně šablonové argumenty, tj. místo operator+ napíšeme operator+<T>. Tím se prováže specializace třídy Vektor se specializací operátoru +. Přitom je nutné poskytnout překladači předem informace o tom, že máme šablonu operátoru +. To jsou ony předběžné deklarace. Nyní vše funguje tak, jak jsme chtěli.

Řešení 2

Předchozí řešení předpokládá, že příslušný operátor + bude pouze specializace šablony. Pokud bychom pro nějaký typ měli k dispozici odpovídající nešablonovou operátorovou funkci, máme smůlu. Ale ne tak docela - můžeme na to jít trochu jinak:

```
// předběžné deklarace
template <class T> class Vektor;

// šablona operátoru +
template <class T>
Vektor<T> operator +(
   const Vektor<T> & v1,
   const Vektor<T> & v2);

// nešablonový operátor +
Vektor<int> operator +(
  const Vektor<int> & v1,
  const Vektor<int> & v2);

// upravená třída Vektor
template <class T>
class Vektor
{
public:
  // ...
```

```
friend
  Vektor<T> ::operator +(
    const Vektor<T> & v1,
    const Vektor<T> & v2);
private:
  T * data_;
  int size_;  // velikost
};
```

Finta spočívá v použití kvalifikovaného jména (a nepoužití id-šablony). Nyní do hry vstupují pravidla [2] a [3]. Pro třídu Vektor<int> je podle pravidla [2] nalezena nešablonová operátorová funkce. Pro všechny ostatní specializace třídy Vektor se použije šablona operátoru + podle pravidla [3].

Ještě poznamenejme, že potřeba použití nešablonové funkce namísto šablony bývá často spojena s optimalizací. Šablona je sama o sobě dost obecná záležitost a pro některé specifické typy může být lepší naprogramovat daný úkol jinak. Je sice možné použít explicitní specializaci, ale někdy snad může být dobré použít nešablonovou funkci. Jazyk C++ nenutí používat to či ono, ale pokud jedno z toho použijeme, máme jistotu, že si s tím dokáže poradit (alespoň teoreticky).

Závěr

Pokud nepoužijeme šablony, je přátelství v C++ poměrně snadnou záležitostí. Jakmile do hry vstoupí šablony, je třeba dávat velký pozor na to, co jednotlivé deklarace přátel znamenají. Jelikož došlo ke změně významu některých konstrukcí, mohou se objevit problémy při kompilaci (a linkování) starších programů.

Typickým příkladem je výše zmíněná implementace třídy a spřáteleného operátoru. Vzhledem ke stavu dnešních překladačů je lépe použít "Řešení 1", tj. provázat šablonové argumenty.

A nakonec ještě malá poznámka k té původní "typické" implementaci. I když tam připíšeme nešablonovou definici potřebného operátoru, mají s tím překladače Borland C++ Builder problémy. Verze 4 ohlásí chybu již při generování (!) kódu a pokročilejší verze 5 občas skončí se záhadnou chybou při linkování. Naproti tomu Microsoft Visual C++ 6.0 s tím problémy nemá.

Ukázky kódu naleznete na Chip CD 10/01 v rubrice Chip Plus. Některé věci však bohužel ještě nefungují tak, jak by podle standardu měly...

Jaroslav Franěk

Literatura

[1] Standard C++: International standard ISO/IEC 14882, 1998-09-01
[2] M. Virius: Šablony po šesti letech, Chip 12/00
[3] J. Franěk: Šablony výrazů, Chip 4/01
[4] M. Virius: Kafe, mříž a dva plusy, Chip 7/01 a 8/01