Šablonové metaprogramování v C++

Dřinu nechte překladači!

Šablonové metaprogramování (template metaprogramming) je velice zajímavá technika, vhodná především pro tvorbu knihoven a optimalizaci kódu. Její počátky se datují přibližně do roku 1994 a od té doby se postupně vyvinula v použitelný a nadějný prostředek, který, alespoň pokročilým, určitě stojí za seznámení...

Rozvíjející se standard C++ umožnil něco, o čem se dříve nikomu ani nesnilo: přidat do jazyka C++ kvalitativně novou vrstvu – metaprogramování. Prostředkem jsou šablony (templates), které zmiňovaný standard značným způsobem rozšířil. Pomocí šablon lze emulovat základní konstrukce programovacích jazyků, jako větvení (if-then-else, switch) nebo smyčky (for, do-while, while). Ukážeme si, jak na to.

Metatypy

Proměnnými v šablonovém metaprogramování jsou typy a celá čísla (přesněji řečeno všechno, co může být argumentem šablony). Vzhledem k tomu, že metaprogram je prováděn během kompilace, musí být čísla z pohledu klasického C++ konstanty. I v šablonovém metaprogramování lze rozlišit proměnné, konstanty a literály. Začneme s jednoduchými příklady:

```
const int N = 1;
template <int M> class A {};

void funkce()
{
 A<N> a; // zde přiřadíme M = N
}
```

Zde N je metakonstanta, M je metaproměnná, 1 je celočíselný metaliterál (předpony meta mají naznačit vztah k metaprogramu). Samozřejmě, N je zároveň konstanta a 1 je celočíselný literál. A nyní pro změnu typy:

```
typedef int X;
template <class T> class B {};

void funkce()
{
 B<X> b; // zde přiřadíme T = X
}
```

Jde o analogii předchozího příkladu: X je (typová) metakonstanta, T je (typová) metaproměnná, int je (typový) metaliterál. I bez složitých definic by mělo být jasné, co je co.

Metaoperace

Ukažme si teď, jak pomocí šablon zapsat některé jednoduché metaoperace. Začneme u násobení:

```
template <int M, int N> struct MetaMUL
{
 static cont int RETURN = M*N;
};
```

Použít to můžeme třeba takto:

```
template <int M, int N> class Matice // matice M x N
{
 // ... uvedeno jen to podstatné
 static const int PocetPrvku()
 {
   return MetaMUL<M, N>::RETURN;
 }
};
```

Metoda PocetPrvku() vrátí počet prvků matice, tj. M*N. Můžete namítnout, že to je zbytečně složité a místo metaoperace MetaMUL by zde šlo použít klasickou operaci násobení. Jednoduché

smysluplné příklady metaprogramování se však těžko hledají a musíme se zatím spokojit s těmi méně smysluplnými.

Ostatní operace s celými čísly lze definovat analogicky. Uvedeme ještě jeden příklad – porovnání:

```
template <int M, int N> struct MetaEQ
{
 static const bool RETURN = (M == N);
};
```

Porovnání je operace, kterou můžeme implementovat i pro typy. V tomto případě musíme ale použít částečnou specializaci šablon:

```
template <class T1, class T2> struct MetaEQTYPE
{
 static const bool RETURN = false;
}

template <class T> struct MetaEQTYPE<T, T>
{
 static const bool RETURN = true;
}
```

Smysluplný příklad použití se sem bohužel nevejde, tak alespoň to nejjednodušší, co lze napsat:

```
MetaEQTYPE<int, double>::RETURN; // výraz 1
MetaEQTYPE<int, int>::RETURN; // výraz 2
```

Pokud překladač narazí na výraz 1, použije nespecializovanou šablonu, a tudíž hodnotu false. Pro výraz 2 se použije specializovaná šablona a hodnota true.

Metafunkce

Od metaoperací je jen krůček k metafunkcím. Ukažme si využití rekurze v hodnotovém parametru šablony na klasickém příkladu výpočtu faktoriálu. Faktoriál nezáporného celého čísla n je definován rekurzivně takto:

```
0! = 1
n! = n.(n-1)!
```

A nyní v šablonách:

```
template <int N> struct MetaFactorial
{
 static const int F = N*MetaFactorial<N-1>::F;
};

// explicitní specializace, ukončení rekurze
template <> struct MetaFactorial<0>
{
 static const int F = 1;
};
```

Co na to překladač? Pokud narazí na výraz MetaFactorial<N>::F, pokusí se vyčíslit jeho hodnotu takto:

```
if (N == 0)
{
 použij specializovanou třídu MetaFactorial<0>
   MetaFactorial<0>::F = 1
}
else   // N > 0
{
 použij nespecializovanou třídu MetaFactorial<N>
    je třeba určit MetaFactorial<N-1>::F,
    a proto použij rekurzivně tento postup pro N-1
 proveď násobení N*cFaktorial<N-1>::F
}
```

Pokud v programu napíšeme cFaktorial<4>::F, překladač to nahradí v době překladu konstantou 24 – výpočet faktoriálu tedy není záležitostí běhu programu, ale překladače! Výpočet faktoriálu byl opět tím nejjednodušším příkladem. Můžeme ovšem vyčíslovat i mnohem

komplikovanější funkce, dokonce i odmocniny:

```
template <int SIZE, int LOW = 1, int HIGH = SIZE>
struct MetaRoot;
{
 static const int mean = (LOW+HIGH)/2;
 static const bool down = (mean*mean >= SIZE);
 static const int root =
   MetaRoot<SIZE, (down ? LOW : mean+1),
            (down ? mean : HIGH)>::root;
};

// částečná specializace
template <int SIZE, int MID>
struct MetaRoot<SIZE, MID, MID>
{
 static const int root = MID;
};
```

Výraz sRoot<N>::root je ekvivalentem výrazu std::ceil(std::sqrt(N)), tj. je to horní celá část odmocniny z N; lze jej použít na místě, kde je očekávána konstanta, např. při deklaraci statického pole:

```
const int N = 10;
int tabulka[sRoot<N>::root];
```

Ještě poznámka: Tento příklad, ačkoliv je zcela v souladu se standardem jazyka C++, v mém překladači (C++ Builder 4.0) nefunguje. Problém je v deklaraci první šablony. Překladač si nedokáže poradit s implicitním argumentem u šablonového parametru HIGH (tj. int HIGH = SIZE). Ohlásí vnitřní chybu a skončí. Doufejme, že v novějších překladačích už to bude lepší.

Meta-if

Větvení patří mezi základní programové konstrukce a v metaprogramování má také své místo. Vytvoříme primární šablonu třídy

```
template <bool B> struct MetaIF {};
```

a explicitní specializaci pro argumenty true a false:

```
template <> struct MetaIF<true>
{
 static void Neco();
};

template <> struct MetaIF<false> {};
```

Toto je implementace neúplného meta-if (kde chybí sekce else). Příklad použití:

```
MetaIF<(sizeof(int) > 2)>::Neco();
```

Zapišme schematicky, co chceme od překladače

```
if (sizeof(int) > 2)
{
 použij třídu MetaIF<true>
 zavolej metodu MetaIF<true>::Neco()
}
else
{
 použij třídu MetaIF<false>
 metoda MetaIF<false>::Neco() není definovaná
   ohlaš chybu
}
```

Pokud bychom chtěli emulovat úplné meta-if, stačí dodefinovat metodu MetaIF<false>::Neco(). Poznamenejme ještě, že pro použití MetaIF<B> je nutné, aby B byl výraz vyčíslitelný v době překladu a převoditelný na typ bool.

Meta-switch

Na podobném principu funguje i emulace konstrukce switch. Vytvoříme primární šablonu třídy MetaCASE. Ta pak bude sloužit jako sekce default v příkazu switch.

```
template <int I> struct MetaCASE
{
 static int NecoDelej() {return 0;}
};
```
Nyní pro každou hodnotu, která by se vyskytla v návěští case, vytvoříme explicitní specializaci.
```
template <> struct MetaCASE<1>
{
 static int NecoDelej() {return 1;}
};

template <> struct MetaCASE<2>
{
 static int NecoDelej() {return 2;}
};
```
Úloha překladače:
při výskytu MetaCASE<I> hledej vhodnou specializaci takto:
```
switch (I)
{
 case 1 :
 {
   použij specializovanou třídu MetaCASE<1>
   zavolej metodu MetaCASE<1>::NecoDelej()
   break
 }
 case 2 :
 {
   použij specializovanou třídu MetaCASE<2>
   zavolej metodu MetaCASE<2>::NecoDelej()
   break
 }
 default :
 {
   použij nespecializovanou třídu MetaCASE<I>
   zavolej metodu MetaCASE<I>::NecoDelej()
   break
 }
}
```
Příklad použití:
```
std::cout << MetaCASE<1>::NecoDelej() << ” ”
          << MetaCASE<3>::NecoDelej();
```
Na obrazovce se objeví výpis ”1 0” (samozřejmě ani toto není zrovna smysluplný příklad...).
Zkusme si ještě ukázat, jak můžeme šablonami nahradit některá makra pro podmíněný překlad.
Makra:
```
#if defined(__PROSTREDI_16_BITU) // int je 2B
// kód pro 16bitové prostředí
#elif defined(__PROSTREDI_32_BITU) // int je 4B
// kód pro 32bitové prostředí
#else
 #error Neni definovano prostredi
#endif
```
Totéž pomocí šablon:
```
template <int VELIKOSTINT> struct MetaProstredi {};

template <> struct MetaProstredi<2>
{
 static void KodZavislyNaVelikostiInt()
 {
 // kód pro 16bitové prostředí
 }
```

```
};

template <> struct MetaProstredi<4>
{
 static void KodZavislyNaVelikostiInt()
 {
 // kód pro 32bitové prostředí
 }
};

// nahradí makra v předchozím příkladu:
MetaProstredi<sizeof(int)>::KodZavislyNaVelikostiInt();
```

Metasmyčky

Emulace smyček je založena na rekurzi v hodnotovém parametru šablony. Počet průchodů smyčkou musí být znám v době překladu. Klasicky by to mohlo vypadat takto:

```
void funkce(int i)
{
 std::cout << i; // nějaká akce
}

const int N = 10;
for (int i = 0 ; i <= N ; i++) funkce(i);
A nyní pomocí šablon:
template <int I> struct MetaLOOP
{
 static void loop()
 {
   MetaLOOP<I-1>::loop(); // rekurze
   std::cout << I;    // nějaká akce
 }
};

// explicitní specializace, ukončí rekurzi
template <> struct MetaLOOP<0>
{
 static void loop()
 {
   std::cout << 0; // nějaká akce
 }
};
```

Ukažme si, co se stane, když překladač narazí na konstrukci MetaLOOP<N>::loop(), kde N je konstanta známá v době překladu:

```
if (N == 0)
{
 použij specializovanou třídu MetaLOOP<0>
 zavolej metodu MetaLOOP<0>::loop()
   (naše metoda vypíše na obrazovku hodnotu 0)
}
else   // N > 0
{
 použij nespecializovanou třídu MetaLOOP<N>
 zavolej metodu MetaLOOP<N>::loop()
 v   metodě MetaLOOP<N>::loop()
   je třeba zavolat metodu MetaLOOP<N-1>::loop(),
   proto použij rekurzivně tento postup pro N-1
 potom proveď další instrukce v metodě
   MetaLOOP<N>::loop()
   (naše metoda vypíše na obrazovku hodnotu N)
}
```

To v praxi znamená, že příkaz MetaLOOP<2>::loop() vypíše na obrazovku řetězec ”012”. Nejdříve se zavolá MetaLOOP<2>::loop(), ta zavolá MetaLOOP<1>::loop() a ta zavolá MetaLOOP<0>::loop(). Zde dojde k výpisu hodnoty 0. Vracíme se do funkce MetaLOOP<1>::loop(), kde dojde k výpisu hodnoty 1. Nakonec se vrátíme do MetaLOOP<2>::loop(), kde se vypíše hodnota 2. Pokud bychom chtěli obrátit sled hodnot, stačí zaměnit pořadí rekurze a ”akce” v metodě MetaLOOP<I>::loop():

```
template <int I> struct MetaLOOP
{
 static void loop()
 {
   std::cout << i;   // nějaká akce
   MetaLOOP<I-1>::funkce();        // rekurze
 }
};
```

Nyní by příkaz MetaLOOP<2>::loop() vypsal na obrazovku řetězec ”210”. A nyní se dostáváme k nevídaným možnostem optimalizace: Pokud budou metody deklarovány jako vložené, tj. deklarovány s použitím klíčového slova inline nebo definovány uvnitř třídy (jako zde), nebude překladač volat funkce, ale přímo tam vloží kód. Navíc je to uděláno rekurzivně, takže překladač v případě příkazu MetaLOOP<2>::loop() vygeneruje následující kód (pro původní vzestupnou verzi):

```
std::cout << 0;
std::cout << 1;
std::cout << 2;
```

Ani zmínka o nějaké smyčce či volání funkce! V angličtině se to označuje jako loop transformation, tedy transformace smyčky do jednosměrné posloupnosti příkazů. Tímto způsobem je možné donutit překladač optimalizovat na rychlost: odstraní se smyčka (testování výrazu, skok na začátek nebo na konec smyčky) i volání funkcí (uložení a odstranění parametrů v zásobníku, skok do funkce a návrat z funkce).

Podmínkou ovšem je, aby počet opakování smyčky byl znám v době překladu a byl malý (viz závěrečné poznámky). Proto tento přístup nelze použít vždy. Velice vhodný je pro tzv. ”malé” vektory a matice. O co jde? Mějme za úkol naprogramovat knihovnu pro práci s vektory (samozřejmě pomocí šablon). Vytvoříme šablonu třídy cVektorA tak, jak nás to učí učebnice C++:

```
template <class TYP> class cVektorA
{
public:
 cVektorA(int velikost)
   : Velikost(velikost), Data(new TYP [velikost]) {}
 ~cVektorA() {delete [] Data;}
 // ... uvedeny jsou jen důležité věci
 TYP operator *(cVektorA<TYP> & v);   // skalární součin
private:
 TYP * Data;
 int Velikost;
};

// přetížený operátor *, skalární součin
template <class TYP>
TYP cVektorA<TYP>::operator *(cVektorA<TYP> & v)
{
 // problém 1: nelze provést pro odlišné velikosti
 if (Velikost != v.Velikost)
 {
   throw std::logic_error(”* nelze provést pro vektory”
                                        ”odlišné velikosti”);
 }
 // problém 2: neefektivní pro ”male Velikosti”
 TYP suma = 0;
 for (int i = 0 ; i < Velikost ; i++)
 {
   suma += Data[i] * v.Data[i];
 }
```

```
 return suma;
}
```
V definici přetíženého operátoru násobení jsou naznačeny některé problémy. Naším cílem je co nejrychlejší kód, takže pro malé vektory to zkusíme jinak – s použitím šablonového metaprogramování. Nová třída cVektor bude parametrizována také velikostí vektoru.

```
template <class TYP, int VELIKOST> class cVektor
{
public:
 cVektor() : Data(new TYP [VELIKOST]) {}
 ~cVektor() {delete [] Data;}
 // ... uvedeny jsou jen důležité věci
 TYP operator *(cVektor<TYP, VELIKOST> & v);
private:
 TYP * Data;
};

 // pomocné třídy pro implementaci skalárního součinu
template <class TYP, int VELIKOST> struct MetaDOT
{
 static inline TYP apply(const TYP * a, const TYP * b)
 {
   return (*a) * (*b) +
    MetaDOT<TYP, VELIKOST-1>::apply(a+1, b+1);
 }
};

// částečně specializovaná třída (ukončení rekurze)
template <class TYP> struct MetaDOT<TYP, 1>
{
 static inline TYP apply(const TYP * a, const TYP * b)
 {
   return (*a) * (*b);
 }
};

// a nyní implementace operátoru skalárního součinu
template <class TYP, int VELIKOST> inline TYP
cVektor<TYP, VELIKOST>::operator *(cVektor<TYP, VELIKOST> & v)
{
 // není problém 1, velikosti jsou stejné
 // velice efektivní pro male VELIKOSTi -> není problém 2
 return MetaDOT<TYP, VELIKOST>::apply(Data, v.Data);
}
```
Jak vidíte, je skalární součin dvou vektorů překladačem přetransformován do tvaru

```
Data[0]*v.Data[0] + Data[1]*v.Data[1] + ... +
Data[VELIKOST-1]*v.Data[VELIKOST-1]
```
což je téměř optimální tvar. Stejně tak můžeme optimalizovat sčítání, odčítání, inicializaci, kopírování atd. Na Chip CD (viz infotipy) naleznete mj. zdrojový soubor metaloop_02.cpp, ve kterém je doplněn jednoduchý test rychlosti. Pro VELIKOST==2 je metakód asi čtyřicetkrát (!) rychlejší než původní implementace pomocí smyček. Pro rostoucí VELIKOST tento poměr klesá.

Několik poznámek na konec

Metakód je možno zapsat různými způsoby. V předchozím jsme všechny operace implementovali pomocí šablon tříd (struct, class) a jejich statických atributů a metod. Někteří autoři občas používají i šablony funkcí nebo místo statických atributů používají výčtové typy (enum).

Použití funkcí není zrovna nejchytřejší. Statická metoda třídy poskytuje v podstatě totéž co funkce. Navíc do třídy je možno vložit mnoho dalších důležitých informací, např. o typu vracené hodnoty, které je možné v metakódu využít.

Výčtové typy představují alternativu statických atributů. Ale pozor: Při použití výčtových typů i statických atributů jsem narazil na závažné problémy v některých překladačích (Borland C++

Builder 4.0); rozhodně se vždy vyplatí pár pokusů předem.

Další omezení se mohou vyskytnout v souvislosti se smyčkami. Nejde totiž o nic jiného než o rekurzivní vytváření instancí šablon. Norma doporučuje tvůrcům překladačů podporovat hloubku rekurze minimálně 17. Různé překladače se ovšem mohou lišit. Borland C++ Builder 4.0 dovolí v některých případech dosáhnout hloubky rekurze větší než 1000 (!), jindy zase nejvíce 86 – prostě pokaždé je to jinak (viz infotipy – ukázky programů na Chip CD). Proto znovu připomínám, že je třeba experimentovat.

Závěr

Hlavní význam šablonového metaprogramování spočívá v optimalizaci výsledného kódu z hlediska rychlosti. Z překladače se tak vlastně stává interpret, který podle našich instrukcí (metaprogramu) nejdříve vygeneruje téměř optimální kód a ten pak přeloží. Metaprogram není nic jiného než chytře vytvořené deklarace šablon tříd a jejich specializací. Rychlost výsledného kódu je pak srovnatelná s rychlostí po ruční optimalizaci.

Současné překladače ovšem zatím mají s některými konstrukcemi problémy. Není se co divit – většina z nich ještě plně neimplementuje šablony tak, jak to vyžaduje standard, a nápis na krabici ”Full ANSI/ISO template implementation” často vyjadřuje spíše přání než skutečnost.

Knihovny napsané pomocí šablonového metaprogramování lze nalézt na internetu. Mezi nejlepší patří Blitz++ a Matrix Template Library, které představují implementaci lineární algebry a některých numerických algoritmů. Tam také hledejte další, praktičtější příklady konstrukcí, které jsme si zde představili.

Optimalizace pomocí šablon samozřejmě nekončí u malých vektorů. Optimalizovat lze také operace prováděné s velkými vektory, o tom však snad někdy příště.

Jaroslav Franěk

infotipy

Standard C++:
International standard ISO/IEC 14882, Programming languages – C++. 1998-09-01.
Blitz++:
http://oonumerics.org/blitz/
MTL:
http://www.lsc.nd.edu/research/mtl/
Předchozí článek:
Šablony po šesti letech, Chip 12/00, str. 192.
Ukázky programů:
Chip CD 1/01, rubrika Chip Plus, Metaprogramy.