ŠABLONOVÉ METAPROGRAMOVÁNÍ V C++

Miroslav Virius

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

Abstrakt

Šablonové metaprogramování představuje specifický přístup k programování, založený především na využití mechanizmu parciální specializace šablon v C++. Metaprogram je výpočet (programová konstrukce), který proběhne v době překladu, nikoli v době běhu přeloženého programu. V tomto příspěvku si ukážeme základní principy šablonového metaprogramování a oblasti jeho použití.

1. ŠABLONY A METAPROGRAMOVÁNÍ (HISTORICKÉ OHLÉDNUTÍ)

Většina programátorů v C++ – včetně nejzkušenějších – se na šablony dívá především jako na nástroj pro vytváření kontejnerů se silnou typovou kontrolou a pro programování algoritmů pro práci s nimi.

Takto šablony opravdu vznikly. Jejich první implementace nabízely omezenou množinu služeb – v podstatě jen deklaraci a definici šablony a možnost explicitní specializace pro určité hodnoty parametrů. Řada problémů přitom zůstávala otevřených a různé implementace je řešily po svém.

V průběhu standardizace jazyka C++ a především jeho knihoven se však ukázalo, že má-li standardní knihovna splňovat požadavky kladené na kvalitní software, je třeba aparát šablon specifikovat pečlivěji a doplnit do něj řadu nástrojů. Tak se součástí jazyka stala pravidla o vnořených šablonách, o dvojím čtení šablon v průběhu překladu, o parciálních specializacích šablon, o přetěžování funkčních šablon atd.

Brzy se ukázalo, že tím vznikl nástroj pro programování překladače, který je výpočetně úplný v oboru celočíselných datových typů jazyka C++. Už v r. 1995 publikoval E. Unruh první metaprogram. Z dnešního pohledu je netypický tím, že ho nelze přeložit, ale překladač v chybových hlášeních vypíše všechna prvočísla menší nebo rovná zadané hodnotě [1].

Od té doby se metaprogramování začalo používat v některých oblastech optimalizace programu, návrhu programové struktury založeném na běžných návrhových vzorech atd.

2. PŘÍKLAD: ASERCE VYHODNOCOVANÁ V DOBĚ PŘEKLADU

Následující příklad pochází z [4] a ukazuje jeden z nejjednodušších metaprogramů. Podívejme se na následující konstrukci, která funguje podobně jako makro `assert()` známé z jazyka C, avšak na rozdíl od tohoto makra se vyhodnocuje již v době překladu:

```
template<bool b> class Assert;      // Deklarace primární šablony
template<> class Assert<true> {};   // Parciální specializace
```

Zde nejprve deklarujeme primární šablonu `Assert<>` s jedním hodnotovým parametrem typu `bool`. Tuto šablonu však nikde nedefinujeme.

Pak deklarujeme a definujeme její parciální specializaci pro hodnotu parametru rovnou `true`. Výsledkem je, že máme šablonu, která není definována pro hodnotu `false`.

Nyní ji můžeme použít:

```
Assert<(N<10)> a;                   // Použití
```

`N` musí být konstanta, jejíž hodnotu překladač zná. Pokud tato konstanta splňuje uvedenou podmínku (`N<10`), vytvoří se bezvýznamná a nikde jinde v programu nepoužitá instance `a`

třídy `Assert<true>`, která bude při optimalizaci odstraněna. Pokud ale tato podmínka splněna není, pokusí se překladač vytvořit instanci třídy `Assert<false>`, a protože odpovídající specializace neexistuje, ohlásí chybu.

Všimněte si, že v tomto příkladu je stejně důležité to, co jsme definovali, jako to, co jsme nedefinovali.

3. VÝPOČETNÍ ÚPLNOST

V úvodu jsme si řekli, že šablony tvoří nástroj výpočetně úplný v oboru celočíselných datových typů. V tomto oddílu si to ukážeme, tj. předvedeme, že pomocí šablon lze

- pracovat s celočíselnými hodnotami,
- naprogramovat rozhodování,
- naprogramovat cyklus.

3.1 Práce s celými čísly

Šablony objektových typů mohou mít vedle typových parametrů také hodnotové parametry celočíselných typů. Podmínkou je, že hodnotu skutečného parametru musí překladač znát (tedy např. umět vypočítat z jiných celočíselných hodnot, které již zná). To znamená, že hodnotové parametry mohou v metaprogramech fungovat jako celočíselné proměnné. Je ale třeba zajistit možnost, jak tyto výsledek metavýpočtu zveřejnit – jak ho dát k dispozici pro další metavýpočty nebo pro výsledný přeložený program. K tomu poslouží jednoduchá šablona třídy obsahující pouze deklaraci veřejně přístupného nepojmenovaného výčtového typu:

```
template<int n> struct Int2Typ
{
   enum{ vysledek = n };
};
```

Tato šablona umožňuje např. deklarovat proměnnou

```
int n = Int2Typ<32>::vysledek;                  // n má hodnotu 32
```

Poznamenejme, že tělo této šablonové třídy je prázdné, takže její použití nezpůsobí generování žádného kódu v přeloženém programu – překladač sice oklikou, ale bez problémů zjistí, že chceme proměnnou `n` inicializovat hodnotou 32, a to také udělá.

Přínos této konstrukce sice zatím není zřejmý, ukáže se však ve spojitosti s dalšími nástroji.

Podobně můžeme „zveřejnit“ datový typ. K tomu použijeme šablonu třídy obsahující pouze veřejně přístupnou deklaraci `typedef` zavádějící nové jméno pro typ předaný jako parametr:

```
template<typename T> struct Typ2Typ
{
   typedef T Typ;
};
```

Nyní můžeme deklaraci proměnné `n` přepsat do tvaru

```
Typ2Typ<int>::Typ n = Int2Typ<32>::vysledek;  // n je typu int
```

Tato deklarace znamená sice stále totéž co

```
int n = 32;
```

nicméně přináší pružnost, kterou deklarace v C++ dosud neměly: Jak počáteční hodnota, tak typ proměnné mohou být výsledkem metavýpočtu.

3.2 Rozhodování

Při sestavování metaprogramů se musíme umět rozhodnout mezi dvěma hodnotami nebo mezi dvěma typy. Protože šablona `Int2Typ<>` zobrazuje celá čísla na datové typy, znamená to, že stačí ukázat, že umíme zajistit rozhodování mezi dvěma datovými typy.

V případě čísel můžeme použít operátor `?:` ve výrazu, který představuje výsledek:

```
template<int n1, int n2, bool podm> struct VyberZeDvou
{
   enum{ vysledek = (podm? n1 : n2)};
};
```

Skutečné parametry ovšem mohou být výsledkem metavýpočtů, a protože jde o parametry šablony, vyhodnotí se oba. To může vést ve složitějších případech k vytváření velkého počtu zbytečných instancí šablon. Proto je výhodnější použít např. následující konstrukci založenou na parciálních specializacích šablon:

```
template<bool, typename, typename>    // Primární šablona opět
class IfThenElse;                     // není definována

template<typename T1, typename T2>    // Parciální specializace pro true
struct IfThenElse<true, T1, T2>       // vrací jeden typ
{
   typedef T1 Typ;
};

template<typename T1, typename T2>   // Parciální specializace pro false
struct IfThenElse<false, T1, T2>     // vrací druhý typ
{
   typedef T2 Typ;
};
```

Proměnná deklarovaná příkazem

```
IfThenElse<(k < 100), int, double>::Typ x = 3.1;
```

bude typu `int`, pokud bude konstanta `k` menší než 100, a typu `double` v opačném případě.

Primární šablonu jsme nedefinovali, neboť parciální specializace pro hodnoty `false` a `true` prvního parametru pokrývají všechny případy, které mohou nastat.

Podobným způsobem můžeme vytvořit i analogii příkazu `switch`. Vytvoříme šablonu, jež bude mít první parametr typu, podle něhož chceme rozhodovat. Primární šablona bude odpovídat alternativě `default`, parciální specializace pak alternativám pro vybrané hodnoty typu prvního parametru. Například analogie příkazu `switch` pro výběr datového typu s alternativami pro hodnoty 1, 5 a pro všechny ostatní může vypadat takto:

```
template<int s, typename T1,
         typename T2, typename T3>    // Primární šablona vyjadřuje
struct Switch                         // alternativu default
{                                     // a vrací typ T3
      typedef T3 Typ;
};

template<typename T1, typename T2,
         typename T3>                // Parciální specializace pro 1
struct Switch<1, T1, T2, T3>         // vrací typ T1
{
   typedef T1 Typ;
};

template<typename T1, typename T2,
         typename T3>                // Parciální specializace pro 5
struct Switch<5, T1, T2, T3>         // vrací typ T2
{
   typedef T2 Typ;
};
```

Bude-li mít první parametr šablony `Switch<>` hodnotu 1 nebo 5, použijí se parciální specializace, jinak se použije primární šablona.

3.3 Cykly

Způsob metaprogramování cyklů je – alespoň na první pohled – poněkud méně zřejmý než v případě rozhodování. Může nám však pomoci analogie s funkcionálně orientovanými programovacími jazyky, jako je Lisp: Tyto jazyky také neobsahují příkazy cyklu, opakování se programuje oklikou pomocí rekurze. Přitom stačí mít na paměti, že:

- V šabloně objektového typu s celočíselným parametrem se můžeme odvolat na instanci téže šablony s jinou hodnotou parametru. Tak donutíme překladač rekurzivně vytvořit posloupnost různých instancí téže šablony.
- K zastavení rekurze použijeme parciální (nebo podle okolností explicitní) specializaci této šablony pro koncovou hodnotu.

Ukážeme si metaprogram, který vypočte faktoriál zadané nezáporné konstanty n, tedy $n!$. Připomeňme si, že $n!$ můžeme rekurzivně definovat vztahy

- $n! = n \cdot (n-1)!$ pro přirozená čísla větší než 0,
- $0! = 1$.

I když *program* založený na této definici faktoriálu bývá uváděn jako příklad nevhodného použití rekurze, při *metaprogramování* pomocí šablon v C++ jinou možnost nemáme.

Nejprve deklarujeme primární šablonu, která obstará rekurzivní výpočet:

```
template<int N>    // Primární šablona obstarávající rekurzivní vpočet
struct Faktorial
{
      enum {vysledek = N*Faktorial<N-1>::vysledek};
};
```

Tato šablona definuje hodnotu `Faktorial<N>::vysledek` pomocí hodnoty instance s parametrem o 1 menším. Rekurzi ukončíme pomocí explicitní specializace této šablony pro hodnotu 0:

```
template<>         // Explicitní specializace ukončující rekurzi
struct Faktorial<0>
{
      enum {vysledek = 1};
};
```

Pokud bychom při ladění potřebovali ověřit, že je šablona `Faktorial<>` použita vždy pouze pro nezáporné hodnoty parametru, můžeme využít šablonu `Assert<>`, s níž jsme se setkali v úvodu, a upravit deklaraci primární šablony `Faktorial<>` následujícím způsobem:

```
struct Faktorial
{
      Assert< (N >= 0) > a;        // Test správnosti použití
      enum {vysledek = N*Faktorial<N-1>::vysledek};
};
```

Explicitní specializaci této šablony pro hodnotu 0 samozřejmě není třeba upravovat.

4. PŘÍKLAD: VĚTŠÍ ZE DVOU KONSTANT

Jednoduchým příkladem metaprogramu je šablona, která najde větší ze dvou konstant. Při řešení můžeme použít operátor `?:`, lze však také využít aparátu, který jsme si dosud vybudovali.

Ukážeme si druhou možnost. Náš postup bude následující:

- Konstanty, předané jako parametry šablony, „zabalíme“ do typů vytvořených pomocí šablony `Int2Typ<>`.
- Typ obsahující větší z nich vybereme pomocí šablony `IfThenElse<>`.
- Získáme hodnotu z tohoto typu.

Nebudeme zde opakovat deklarace šablon `Int2Typ<>` a `IfThenElse<>`, ty jsou stejné jako v oddílu 3.1 a 3.2. Deklarace šablony pro výpočet maxima pak může mít tvar

```
template<int M, int N>
struct Max
{
   enum {vysledek =
      IfThenElse<(M>N), Int2Typ<M>, Int2Typ<N> >::Typ::vysledek};
};
```

I když to možná vypadá nepřehledně, použití je jednoduché: Jsou-li `m` a `n` dvě konstanty, jejichž hodnoty jsou známy v době překladu, můžeme napsat `Max<m,n>::vysledek` a získat tak konstantu (opět známou v době překladu), jejíž hodnota je rovna větší z `m` a `n`.

5. SLOŽITĚJŠÍ PŘÍKLAD: ODSTRANĚNÍ CYKLU Z PROGRAMU

Cykly jsou překladačem zpravidla optimalizovány pro velký počet opakování. To může být v případě cyklů s malým počtem opakování i kontraproduktivní, vzhledem k režii cyklu to může vést k nižší rychlosti programu než v případě, že bychom cyklus „rozepsali“. Rozepsáním cyklu bychom ale ztratili možnost naprogramovat tento úsek programu jako funkci použitelnou pro různé počty opakování (nehledě k tomu, že program ty tím značně ztratil na přehlednosti). Zde může pomoci šablonové metaprogramování.

Položme si nyní za úkol napsat program, který bude vrátí hodnotu největšího prvku v poli číselného typu. Poznamenejme, že jde o trochu jinou úlohu, než jaké jsme dosud řešili – prvky pole nejsou konstanty. Budeme tedy muset využít funkce a metody.

Klasický program pro nalezení největšího prvku v daném poli může vypadat takto:

```
template<typename T>
inline T Maximum(int N, T* a) // Předáváme délku pole a počet prvků
{
  T pom = a[0];
  for(int i = 0; i < N; i++)
  if(a[i] > pom)pom = a[i];
    return pom;
}
```

Metaprogramová konstrukce musí být založena na rekurzivní formulaci řešení. Ta využívá skutečnosti, že největší prvek v celém poli najdeme, když najdeme největší prvek v daném poli počínaje druhým prvkem, porovnáme ho s prvním prvkem a vrátíme větší z nich. Podívejme se nejdříve na programovou (nikoli metaprogramovou) realizaci funkce založené na této formulaci:

```
template<typename T>
T maximum(int N, T* a)
{
      if(N == 1) return *a;
      else return max(*a, Maximum(N-1, a+1));
}
```

V příkazu `return` využíváme standardní šablonovou funkci `max()` z hlavičkového souboru `<algorithm>`.

Nyní přejdeme k metaprogramu. Rozhodování, zda je `N == 1`, za nás vyřeší mechanizmus specializace šablon. Primární šablona bude obsahovat rekurzivní výpočet, specializace pro hodnotu `N == 1` ukončí rekurzi.

```
template<int N, class T> struct Maximum
{                                  // Primární šablona
      static T Hodnota(T* a)
      {
```

```
            return max(*a, Maximum<N-1,T>::Hodnota(a+1));
      }
};

template<class T>
struct Maximum<1, T>                // Parciální specializace
{                                   // ukončuje rekurzi
   static inline T Hodnota(T* a)
   {
      return *a;
   }
};
```

Je-li `A` pole deklarované příkazem

```
int A[] = {1,2,3};
```

můžeme najít jeho největší prvek zápisem

```
Maximum<3,int>::Hodnota(A)
```

To je poněkud neobvyklé, a proto můžeme volání metody `Maximum<N, T>::Hodnota()` „zabalit“ do pomocné funkce,

```
template<int N, typename T>
inline T nejvetsi(T *a)
{
  return Maximum<N,T>::Hodnota(a);
}
```

Nevýhodou popsaného řešení je, že velikost pole musíme znát již v době překladu.

6. OMEZENÍ

Podívejme se nyní na omezení, se kterými musíme při používání tohoto nástroje počítat.

- Možnosti využití šablonového metaprogramování snižuje omezení na celočíselnou aritmetiku, dané tím, že standard u šablon dovoluje hodnotové parametry pouze celočíselných typů, nikoli např. typu `double`.
- Standard [5] jazyka C++ dovoluje implementacím, aby omezily hloubku rekurzivního vytváření instancí; doporučené, nikoli však vyžadované minimum je pouhých 17 úrovní rekurze.
- Mnohé překladače dosud šablony neimplementují v plném rozsahu.
- Odstraňování syntaktických, ale především sémantických chyb v metaprogramech je podstatně složitější než „obyčejných programech“.

Dnešní překladače se standardu pouze blíží, i když rozdíly jsou již většinou velice malé. Bohužel se však zpravidla týkají právě implementace šablon. Můžeme si ale dovolit předpokládat, že v průběhu několika dalších let odlišnosti implementací hlavních komerčních i nekomerčních překladačů C++ od standardu zmizí.

Dovolená hloubka rekurze při vytváření instancí šablon závisí mimo jiné i na paměti, kterou má překladač k dispozici. I v tomto případě lze očekávat pozitivní vývoj.

Omezení na parametry celočíselných typů zřejmě souvisí s tradičním pojetím celočíselných konstant v C++. Očekává se ale, že bude v některé z příštích úprav standardu odstraněno, neboť jde o zásah do jazyka, který nepředstavuje žádný závažný problém.

Drobné technické problémy může při metaprogramování působit skutečnost, že různé instance téže šablony, které se liší typem nebo hodnotou parametrů, představují navzájem různé třídy, které nemají žádná zvláštní práva, pokud jde o přístup ke složkám. Proto zpravidla používáme šablony tříd deklarované pomocí klíčového slova `struct`, nikoli pomocí `class`.

7. K ČEMU TO?

Zbývá položit si základní otázku: Co nám může šablonové metaprogarmování přinést? Ukazuje se, že mnohé.

- Není příliš těžké vytvořit seznam datových typů, který bude existovat pouze v době návrhu programu. Z tohoto seznamu lze vybírat (lze ho v podstatě indexovat) a typy z tohoto seznamu lze používat v dalších konstrukcích.
- Seznam typů lze uspořádat (setřídit) v době překladu tak, že nejodvozenější typu v něm budou na počátku.
- Tyto a podobné konstrukce umožňují vytvořit např. kvalitní šablonovou knihovnu podporující využívání vybraných návrhových vzorů (Singleton, Příkaz, Abstraktní továrna atd.). Podrobnější informace o jedné realizaci takové knihovny lze najít v [4].
- Programy využívající odstranění cyklů, založené na metaprogramování, mohou být výrazně rychlejší. Výsledná hodnota výrazně závisí na použitém překladači, pokusy ukazují, že rozdíl v rychlosti funkce s metaprogramově odstraněným cyklem a „klasické“ funkce se mohou pohybovat v rozmezí 15–80 %. Podrobnější údaje o jednom z testů lze najít v [6].

Poděkování

Tento příspěvek vznikl v rámci práce na fyzikálním experimentu COMPASS v CERN podporované z grantu MŠMT Kontakt ME492.

ODKAZY

[1] http://www.erwin-unruh.de/primorg.html

[2] T. Veldhuizen: *Template Metaprograms*. C++ Report Vol. 7 (1995), No. 4, str. 36–43; http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html

[3] D. Vandevoorde, N. Josuttis: *C++ Templates*. Addison-Wesley Publishing Comp. 2003. ISBN 0-201-73484-2.

[4] A. Alexandrescu: *Modern C++ Design*. Addison-Wesley Publishing Comp., New York 2001. ISBN 0-201-70431-5.

[5] *International Standard ISO/IEC 14882:1998. Programming Languages — C++*.

[6] Virius, M.: *Fascinující svět šablon*. Ve sborníku *Objekty 2004*, ed. D. Ježek a V. Merunka, ČZU Praha 2004, ISBN 80-248-0672-X, str. 305.