Šablonové metaprogramování v C++

Miroslav Virius
KSI FJFI ČVUT

Šablonové (generické) metaprogramování

- Šablona v C++, genericita v jiných jazycích
- Výpočetní úplnost
- Problémy
- Příklad
- Porovnání s klasickým výpočtem

C++ vs. Java 5, C# 2

Podstatný rozdíl mezi C++ na jedné straně a Javou a C# na druhé straně:

- C++: Parametrizovaná množina objektových typů a volných funkcí, instance se definují v době překladu a představují různé typy či funkce
- C#, Java: existuje jen jeden typ či metoda, kontroly použití v době překladu, přetypování za běhu

Šablona v C++

- Většinou chápána jako *nástroj pro generování parametrizované množiny tříd nebo volných funkcí*
- Tak opravdu vznikla
- Návrh STL si vynutil rozšíření možností
  - Parciální a úplné specializace
  - Vnořené šablony
  - Přetěžování šablon volných funkcí
  - Atd.
- **Nástroj pro programování překladače**

První metaprogram

- Erwin Unruh, 1995:
- Program, který nelze přeložit, ale v chybových hlášeních vypíše prvních *n* prvočísel
- Metaprogramování jako směr
- Standardní knihovna C++ je založena mj. na některých metaprogramových konstrukcích

Parciální a úplná specializace

```
// Primární šablona
template<typename T> class vector {
    // Nějaká definice (nemusí být)
};

// Parciální specializace pro určitou
// skupinu typů
template<typename T> class vector<T*> {
    // Nějaká jiná definice
};
// Úplná specializace
template<> class vector<bool> { /*...*/ }
```

Nejjednodušší příklad

- Analogie makra assert() v době překladu

```
// Primární šablona není definována
template<bool b> class Assert;

// Specializace pro true
template<> class Assert<true> {};

// Užití:
Assert< (N > 0) > a;
```

Výpočetní úplnost

- Zobrazení typů a čísel na objektové typy

```
template <int N> struct int2typ {
    enum {vysledek = N};
};

template <class T> struct typ2typ {
    typedef T typ;
};

typ2typ<int>::typ x = int2typ<5>::vysledek;
// int x = 5;
```

Výpočetní úplnost: Podmínka

```
template<bool, class, class>
  class IfThenElse;

template<typename T1, typename T2>
struct IfThenElse<true, T1, T2> {
    typedef T1 Typ;
};

template<typename T1, typename T2>
struct IfThenElse<false, T1, T2> {
    typedef T2 Typ;
};

IfThenElse<n==2,int,double>::Typ x = 3;
```

Výpočetní úplnost: Přepínač

```
template<int N,class T1,class T2,class T3>
  struct Switch {
    typedef T3 typ;
};     // Alternativa default:

template<class T1, class T2, class T3>
  struct Switch<0, T1, T2, T3> {
    typedef T1 typ;
};     // Pro N == 0 atd.
```

Výpočetní úplnost: Cyklus

- Náhrada rekurzivními parciálními specializacemi
- Ukončení cyklu explicitní specializací
- Výsledek: v podstatě funkcionální programování (připomíná např. LISP)
- Rekurzivní generování typů
- Pokud typ neobsahuje než výčtové typy a **`typedef`**, nezpůsobí generování žádného kódu

Příklad cyklu: Výpočet faktoriálu

```
template<int n>    // Primární šablona
struct fakt        // pro n > 0
{
   enum{ vysledek=n*fakt<n-1>::vysledek };
};

template<>         // Specializace pro n == 0
struct fakt<0>     // ukončuje rekurzi
{
   enum{ vysledek=1 };
};

int x = fakt<5>::vysledek;
```

Shrnutí: Výpočetní úplnost

Aparát šablon umožňuje:

- používat celočíselné parametry šablon jako stavové proměnné,
- implementovat rozhodování prostřednictvím specializace šablon nebo pomocí podmíněného operátoru `? :`,
- implementovat cykly pomocí rekurzivních explicitních nebo parciálních specializací šablon,
- používat celočíselnou aritmetiku

Problémy

- Omezení na celočíselnou aritmetiku
- Překladač může omezit hloubku rekurze při generování instancí šablon na pouhých 17 (!)
- Obtížné ladění
- Překladače nevyhovující standardu…

Využití idejí ŠM pro optimalizaci: Výpočet skalárního součinu

- Jde o výpočet

```
a[0]*b[0] + a[1]*b[1] + ... + a[n-1]*b[n-1]          (*)
```

- Tradiční způsob:

```
template<typename T>
inline T SkSoucin(int n, T* a, T* b)
{
   T r = 0;
   for(int i = 0; i < n; ++i)
      r += a[i]*b[i];
   return r;
}
```

- Překladače optimalizují na velký počet opakování cyklu, při n == 2 nebo 3 je to kontraproduktivní. Rozepsat (*) je nejvýhodnější, ale …

Pomocí metaprogramu (1)

```
// Primární šablona
// pro výpočet skalárního součinu
template<int n, typename T>
struct SkalSoucin
{
    static T hodnota(T* a, T* b)
    {
     return *a * *b +
     SkalSoucin<n-1, T>::hodnota(a+1, b+1);
    }
};
```

Metaprogram (2)

```
// Parciální specializace pro n == 1
// ukončuje rekurzi
template<typename T>
struct SkalSoucin<1, T>
{
    static T hodnota(T* a, T* b)
    {
        return *a * *b ;
    }
};
```

Metaprogram (3)

```
// Neintuitivní zápis
C = SkalSoucin<3, double>::hodnota(a, b);

// lze obalit pomocnou funkcí
template<int n, typename T>
inline T skal_souc(T *a, T *b)
{
 return SkalSoucin<n, T>::hodnota(a, b);
}
```

Porovnání

```
int main()
{                                          // volatile
   volatile double A[3] = {1,2,3}; // zabraňuje
   volatile double B[3] = {4,5,6}; // optimalizacím
   double C;
   // Přípravné operace
   _timeb T1, T2;
   _ftime(&T1);                    // Začátek měření času
   for(int i = 0; i < N; ++i)
      C = SkSoucin(3, A, B);
   _ftime(&T2);                    // Konec měření času
   // ... Výpočet rozdílu časů a výstup výsledku
   // ... Totéž pro skal_souc<3>(A, B)
   // a pro rozepsaný skalární součin
}
```

Výsledky (v sekundách)

```
const int N=1000000000    // Počet násobených polí 10^9
```

	BCX	GNU	Intel	MSVC
Klasicky	72,3	83,5	42,0 (0,6)*	15,6
Metaprogram	10,8	52,3	2050 (0,35)*	4,1
Rozepsaný součin	3,0	4,1	2049 (0,16)*	5,3

* Bez modifikátoru **`volatile`**

SFINAE

- Substitution Failure Is Not An Error
  - Chyby při dosazení odvozených parametrů šablony funkce (nesmyslný typ apod.) znamenají vyloučení šablony ze seznamu kandidátů, nikoli ukončení překladu
  - Knihovní šablona enable_if<podm, typ = void>

SFINAE: Příklad

```
// č. 1
template <class T, T* param>
int f(int n)
{
        return int(*param)*n;
}

// č.2
template<class T, T param>
int f(int n)
{
        return n * param;
}

int main()
{
        // Zavolá se funkce č. 2
        int i2 = f<int, 1>(0);
}
```

SFINAE: Příklad

```
// Výběr funkce podle parity parametru šablony
template <int I>
void parita(char(*)[I % 2 == 0] = 0)
{
        cout << "parametr šablony I = " << I << " je sudý"
<< endl;
}

template <int I>
void parita(char(*)[I % 2 == 1] = 0)
{
        cout << "parametr šablony I = " << I << " je lichý"
<< endl;
}

int main()
{
        system("CHCP 1250 > NUL");
        parita<3>();
        parita<8>();
}
```

Podivná rekurze šablon

- Curiously recurring template pattern (CRTP)
  - Použijeme potomka jako parametr šablony předka
  - Umožňuje potomkovi upravit předka
  - Předek nesmí obsahovat instanci šablonového parametru, může ale obsahovat ukazatel nebo referenci