Szablony

dla co nieco zaawansowanych programistów C++

Zbigniew Koza

Instytut Fizyki Teoretycznej
Uniwersytet Wrocławski

Wrocław, 10 października 2017

Spis treści

Spis treści

Literatura

- D. Vandervoode, N. M. Josuttis,
  C++ Szablony. Vademecum profesjonalisty,
  Helion 2003
  - Źródła: ftp.helion.pl
- A gentle introduction to Template Metaprogramming with C++

Rodzaje szablonów

- Szablony funkcji

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

Rodzaje szablonów

- Szablony funkcji

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

- Szablony klas

```
template <typename T>
class vector
{
    size_t size;
    size_t capacity;
    T*     p;
public:
    vector(size_t n = 0);
    ...
};
```

Używanie szablonów

- Parametry szablonów funkcji mogą podlegać dedukcji lub być podawane jawnie

```
max(1, 2);                  // dedukcja: T = int
max(3.14, 4.5);             // dedukcja: T = double
max<double>(1, 3.14);
max<double>(1, 2);
max<>(1,3);                 // max jest szablonem; dedukcja T
```

- Parametry szablonów klas muszą być podane jawnie

```
vector<double> v(100);
```

Konkretyzacja szablonów

- Tworzenie klas/funkcji z szablonów to ich konkretyzacja

```
vector<int> v; // <-- możliwa konkretyzacja klasy vector<int>
```

- Szablony skonkretyzowane innymi typami reprezentują różne klasy C++:

```
array<int, 10> *v;
array<int, 11> *w;
// w = v; /* błąd! */
```

Kompilacja szablonów

- Treść szablonów musi być znana podczas kompilacji – szablony umieszczamy w plikach nagłówkowych
  - Kompilatory nie obsługują słowa kluczowego `export`
- Szablon funkcji pozbawionej atrybutu `inline` jest traktowany jak zwykła funkcja
- Kompilacja przeprowadzana jest w dwóch fazach
  - Ogólne sprawdzenie składni (wstępna diagnostyka błędów)
  - Konkretyzacje dla określonych parametrów
- Konkretyzacji podlegają tylko te metody, które tego wymagają
  - Wiele błędów ujawnia się dopiero podczas konkretyzacji
  - Kompilator nie może rozstrzygnąć, czy błąd jest w szablonie, czy w sposobie jego konkretyzacji (użycie niepasującego parametru)

Parametry szablonów

- Parametry szablonów (funkcji lub klas) mogą być typami lub wyrażeniami „całkowitymi” znanymi w czasie kompilacji

```
template <typename T, int N>
class array
{
    T tab[N];
...
};
...
array<int, 10> v;
array<int, 'a'> w; // !????
```

Domyślne parametry szablonów

- Często używane w szablonach klas

```
template <typename T, int N = 128>
class array
{
    T tab[N];
...
};

array<int> v; // N = 128
array<int> w; // N = 128
```

- ~~W szablonach funkcji~~

Ważne!

Szablony funkcji nie mogą mieć parametrów domyślnych

Przeciążanie szablonów funkcji

- Szablony funkcji (ale nie klas!) mogą być przeciążane

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

template <typename T>
inline T const& max (const* T const& a, const* T const& b)
{
    return max(*a, *b);
}
```

- Ruguły rządzące polomorfizmem nazw są skomplikowane (szablony/specjalizacje szablonów/funkcje/funkcje składowe/ dziedziczenie/przestrzenie nazw/typy argumentów)

Specjalizacja częściowa

- Specjalizacja częściowa: tylko dla szablonów klas

```
template <typename T>
vector
{
...
};

// specjalizacja częściowa dla wskaźników
template <typename T>
vector<T*>
{
...
};
```

- Cel: optymalizacja dla niektórych klas typów
- Ułatwia unikanie pompowania kodu (*code bloat*)

Specjalizacja pełna

- Dostępna dla szablonów funkcji i klas

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// specjalizacja dla const char*
template <>
inline const char* const& max
  (const char* const& a, const char* const& b)
{
    return strcmp(a,b) == 1 ? b : a;
}
```

Specjalizacja pełna

- Dostępna dla szablonów funkcji i klas

```
template <typename T>
struct X
{
   void info() { std::cout << "X" << "\n"; }
};

template <>
struct X<int>
{
   int z;
   void info_int() { std::cout << "int jest piękny\n"; }
};
```

Specjalizacja pełna

- Dostępna dla szablonów funkcji i klas

```
template <typename T>
struct X
{
   void info() { std::cout << "X" << "\n"; }
};

template <>
struct X<int>
{
   int z;
   void info_int() { std::cout << "int jest piękny\n"; }
};
```

Ważne!

Specjalizacje klas z klasą bazową łączy tylko nazwa

Podsumowanie wstępu

	Szablony funkcji	Szablony klas
Przeciążanie	Tak	Nie
Specjalizacje pełne	Tak	Tak
Specjalizacje częściowe	Nie	Tak
Argumenty domyślne	Nie	Tak
Dedukcja parametrów	Tak	Nie

Część 2

Składnia

```
template <typename T>
class Y {
...
};

template <typename T>
class X {
    X x()
    {
        Y<T>::y * z; // definicja czy wyrażenie?
    }
};
```

- Za względu na możliwość zdefiniowania specjalizacji `Y`, bez przeprowadzenia specjalizacji nie można rozstrzygnąć, czy `Y::y` oznacza typ, czy obiekt.
- Kompilator zakłada, że chodzi o obiekt.
- Użycie `Y.y` jako typu wymaga specjalnej składni

Słowo kluczowe typename

- Słowa kluczowego typename należy używać tam, gdzie nazwa uzależniona od parametru szablonu reprezentuje typ

```
template <typename T>
class X
{
    typename T::iterator * it; // deklaracja wskaźnika
};
```

- Użycie typename umożliwia odróżnianie deklaracji wskaźnika od mnożenia przed próbą konkretyzacji szablonu (bez znajomości T)

```
    T::n * b;  //  mnożenie przez składową statyczną T::n
```

Konstrukcje .template, ::template i ->template

- Konstrukcje .template, ::template i ->template informują kompilator, że następujący po nich identyfikator reprezentuje szablon

```
template <int N>
void print(std::bitset<N> const& bs)
{
  std::cout << bs.template to_string<char> ();
}
```

- Używane są tylko wewnątrz szablonów
- Dzięki nim kompilator w pierwszej fazie kompilacji szablonu nie myli < i > z operatorami relacyjnymi

Dziedziczenie szablonu z szablonu

```
template <typename T>
struct X {
    T size;
};

template <typename T>
class Y : public X<T>
{
   Y() {
       size * x; //  skąd wiemy, że klasa  bazowa ma składową size?
       this->size * x; // OK
       X<T>::size * x; // OK
   }
};
```

- Konieczna nazwa kwalifikowana (`X<T>::` lub `this->`)

Podsumowanie

- Kompilacja szablonów odbywa się w dwóch etapach: weryfikacja składni i konkretyzacja
- Możliwość definiowania specjalizacji szablonów znacznie ogranicza możliwość weryfikacji składni
- Jeżeli wewnątrz drugiego szablonu używamy szablonu sparametryzowanego parametrem pierwszego szablonu, kompilator będzie miał kłopot z identyfikacją znaczenia identyfikatorów (typ czy zmienna?)
- Podobny kłopot powstaje, gdy w szablonie klasie pochodnej chcemy odwołać się do składowych szablonu klasy podstawowej

Zapamiętaj!

```
typename X::typ
.template,  ::template,  ->template
this->składowa, KLASA<...>::składowa
```

Metaprogramowanie

- Metaprogramowanie w C++ polega na użyciu szablonów oraz reguł gramatyki języka w celu wymuszenia na kompilatorze wygenerowania kodu źródłowego, skompilowania go i włączenia do programu.
- Metaprogramowanie może służyć do generowania
  - Stałych
  - Kodu (klas, funkcji)
- Zastosowanie praktyczne: rozwijanie pętli

Przykład 1: zliczanie bitów w bajcie

```
template <unsigned char byte>
struct BITS_SET
{
enum {
     B0 = (byte & 0x01) ? 1:0,
     B1 = (byte & 0x02) ? 1:0,
     B2 = (byte & 0x04) ? 1:0,
     B3 = (byte & 0x08) ? 1:0,
     B4 = (byte & 0x10) ? 1:0,
     B5 = (byte & 0x20) ? 1:0,
     B6 = (byte & 0x40) ? 1:0,
     B7 = (byte & 0x80) ? 1:0
    };
 enum{RESULT = B0 + B1 + B2 + B3 + B4 + B5 + B6 + B7};
};
...
std::cout << BITS_SET<15>::RESULT;
```

Przykład 2: silnia

```
template <int N>
class SILNIA{
  public:
      enum {WYNIK = N * SILNIA<N-1>::WYNIK};
};

template<>
class SILNIA <1>
{
  public:
      enum {WYNIK = 1};
};
...
std::cout << SILNIA<8>::WYNIK;
```

Przykład 3: rozwijanie pętli

```
template <typename T, int N>
struct SKALARNY
{
  T static OBLICZ(const T* x, const T* y)
  {
    return *x * *y + SKALARNY<T, N-1>::OBLICZ(x + 1, y + 1);
  }
};

template <typename T>
struct SKALARNY<T, 1>
{
  T static OBLICZ(const T* x, const T* y)
  {
    return *x * *y;
  }
};
```

Przykład 3: rozwijanie pętli (c.d.)

- Funkcja pomocnicza:

```
template <int N, typename T>
inline
T skalarny(const T* x, const T* y)
{
    return SKALARNY<T, N>::OBLICZ(x,y);
}
```

- Użycie w programie:

```
int x[] = {0, 1, 2};
int y[] = {1, 2, 3};
...
std::cout << skalarny<3>(x,y) << "\n";
```

lub:

```
std::cout << SKALARNY<int, 3>::OBLICZ(x, y> << "\n";
```

Przykład 3: rozwijanie pętli (c.d.)

- Rozwiązanie „klasyczne”:

```
inline
int fskalarny(const int *x, const int *y, int n)
{
    int wynik;
    for (int i = 0; i < n; i++)
       wynik += x[i] * y[i];
    return wynik;
}
```

- Użycie w programie:

```
int x[] = {0, 1, 2};
int y[] = {1, 2, 3};
...
std::cout << fskalarny(x, y, 3) << "\n";
```

Przykład 3: rozwijanie pętli (c.d.)

- Rozwiązanie „klasyczne”:

```
std::cout << fskalarny(x, y, 3) << "\n";
```

- Rozwiązanie za pomocą metaprogramowania:

```
std::cout << skalarny<3>(x, y) << "\n";
```

- Wniosek: minimalna różnica w składni

Przykład 3: rozwijanie pętli (c.d.)

- Czas wykonania na moim komputerze pętli złożonej z `N = 1000000000` wywołań funkcji dla iloczynu skalarnego wektora `int`-ów o długości `M`

M	Metaprogram	Funkcja inline z pętlą
2	0.48	2.0
3	0.47	2.6
4	0.61	2.7
5	0.78	6.8
6	1.18	7.0

Przykład 3: rozwijanie pętli (c.d.)

- Uwaga: zaprezentowany tu sposób rozwijania pętli wymaga kompilacji w trybie Release (włączona optymalizacja)

Literatura

- D. Vandervoode, N. M. Josuttis,
  C++ Szablony. Vademecum profesjonalisty,
  Helion 2003
- Źródła: ftp.helion.pl