Adam Kruszewski

Metaprogramowanie za pomocą szablonów w C++

Plan prezentacji

1. Co to jest metaprogramowanie?
2. Dlaczego metaprogramowanie?
3. Trochę o szablonach w C++
4. Proste przykłady
5. Przykład wykorzystania przy implementacji równań fizycznych
6. Kiedy warto stosować tę technikę

Co to jest metaprogramowanie?

Jest technika programistyczna polegająca na pisaniu programów, które tworzą lub modyfikują inne programy (lub siebie).

Przykład metaprogramu

```
#!/bin/bash
# metaprogram
echo '#!/bin/bash' >program
for ((I=1; I<=992; I++)); do
    echo "echo $I" >>program
done
chmod +x program
```

Cele metaprogramowania?

- pisanie ogólnego kodu, tak żeby tracić jak najmniej efektywności
- wykonywanie części obliczeń już podczas kompilacji
- prosty interfejs powstających programów
- kontrola typów

Jakie mogą być szablony?

Wszystkie podawane parametry szablonu muszą być znane w trakcie kompilacji.
Przykłady szablonów:

```
template <typename T>
class Vector{
...}

template <typename T, int N>
class Vector{
...}
```

Jakie mogą być szablony

Parametryzowane mogą być również funkcje oraz struktury.

Przykłady:

```
template <typename T>
T sum(T *tab, int numElements){
...}

template <class S, int N>
struct Wezel{
    S sasiedzi[N];
}
```

Jak przebiega kompilacja szablonów?

```
template <typename T, int N>
class Vector{
...};

Vector <int, 10> v;
Vector <int, 10> v1;
Vector <int, 11> v2;
```

Dla deklaracji v, v2 generowany jest kod oddzielnej klasy. Zatem v i v1 są tego samego typu, natomiast już v2 jest innego typu.

Kompilacja szablonów

Konrektyzacja – proces zastępowania parametrów szablonów przez podane typy
Specjalizacja – efekt konkretyzacji
Częściowa specjalizacja – efekt zastąpienia tylko części parametrów szablonu konretnymi wartościami

Na czym polega metaprogramowanie za pomocą szablonów w C++

- Używanie procesu instancjonowania szablonów jako mechanizmu obliczeniowego
- Używanie parametryzowanych typów i stałych do wyrażenia rekordów
- Używanie częściowej specjalizacji do implentacji warunków

Przykład 1

```
template <int N>
struct Factorial {
  enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0> {
  enum { value = 1 };
};
```

Przykład 2

```
template<int N>
class Pow3 {
  public:
    enum { result = 3 * Pow3<N-1>::result };
};
template<>
class Pow3<0> {
  public:
    enum { result = 1 };
};
```

Przykład 3

```
template <int N, int I=1>
class Sqrt {
  public:
    enum { result = (I*I<N) ? Sqrt<N,I+1>::result
                            : I };
};
template<int N>
class Sqrt<N,N> {
  public:
    enum { result = N };
};
```

Analiza przykładu

Krok 1:

*result = (1*1<4 ? Sqrt<4,2>::result : 1*

Krok 2:

*result = (1*1<4 ? (2*2<4) ? Sqrt<4,3>::result : 2* : 1

Krok 3:

*result = (1*1<4 ? (2*2<4) ? (3*3<4) ? Sqrt<4,4>::result : 3*
: 2 : 1

Krok 4:

*result = (1*1<4 ? (2*2<4) ? (3*3<4) ? 4 : 3* :2 : 1

Równania fizyczne

Gdy liczymy ręcznie nie piszemy jednostek, bo wiemy, że nie możemy dodać masy do długości.
Czego oczekujemy:

- Jeśli będziemy mieli masę m1 i m2, że będziemy mogli je dodać.
- Jeśli będziemy mieli masą m1 i długośc l1, że nie będziemy mogli ich dodać.
- Jeśli wymnożymy prędkość przez czas, że wyjdzie nam długość

Zwykle podejście

Jak rozwiązalibyśmy problem w naturalnym podejściu:

```
typedef int dimension[7]; // m  l  t  ...
dimension const mass      = {1, 0, 0, 0, 0, 0, 0};
dimension const length    = {0, 1, 0, 0, 0, 0, 0};
dimension const time      = {0, 0, 1, 0, 0, 0, 0};
...
dimension const force  = {1, 1, -2, 0, 0, 0, 0};
```

A może metaprogramowanie

Do zdefiniowania poszczególnych wielkości użyjemy metaprogramowanie i definicji już zawartych w bibliotece mpl.
Najpierw jak tego można używać?

```
template <int N> struct int_;

namespace mpl = boost::mpl; // namespace alias
static int const five = mpl::int_<5>::value;
```

Jak używać biblioteki mpl

Definicja masy wyglądała by następująco:

```
typedef mpl::vector<
   mpl::int_<1>, mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
 , mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
> mass;
```

Wady:

- nieczytelne
- trudne do weryfikacji

Użycie mpl cd.

```
typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length; // or position
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
typedef mpl::vector_c<int,0,0,0,1,0,0,0> charge;
typedef mpl::vector_c<int,0,0,0,0,1,0,0> temperature;
typedef mpl::vector_c<int,0,0,0,0,0,1,0> intensity;
typedef mpl::vector_c<int,0,0,0,0,0,0,1> angle;
```

Inne wielkości:

```
typedef mpl::vector_c<int,0,1,-1,0,0,0,0> velocity;     // l/t
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration; // l/(t2)
typedef mpl::vector_c<int,1,1,-1,0,0,0,0> momentum;     // ml/t
typedef mpl::vector_c<int,1,1,-2,0,0,0,0> force;        // ml/(t2)
```

Jak stworzyć konkretne wartości?

```
template <class T, class Dimensions>
struct quantity
{
    explicit quantity(T x)
      : m_value(x)
    {}

    T value() const { return m_value; }
 private:
    T m_value;
};
```

Wreszcie...

```
quantity<float,length> l( 1.0f );
quantity<float,length> l1( 2.0f );
quantity<float,mass> m( 2.0f );

l = m; // takie wyrażenie nie zostanie skompilowane
l1 = l; //to się skompiluje, bo l i l1 są tego samego typu
l = l1 + l;//nie skompiluje się, bo nie mamy jeszcze operatora +
```

Operator + i - dla wielkości fizycznych

```
template <class T, class D>
quantity<T,D>
operator+(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() + y.value());
}

template <class T, class D>
quantity<T,D>
operator-(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() - y.value());
}
```

Co teraz możemy

```
quantity<float,length> len1( 1.0f );
quantity<float,length> len2( 2.0f );

len1 = len1 + len2;   // OK
```

ale

```
len1 = len2 + quantity<float,mass>( 3.7f ); // error
```

Mnożenie wielkości fizycznych

```
#include <boost/mpl/transform.hpp>

template <class T, class D1, class D2>
quantity<
    T
  , typename mpl::transform<D1,D2,mpl::plus>::type
>
```

To nie zadziała, bo transform trzeci parametr musi być typem, a *mpl:plus* jest szablonem klasy.

Mnożenie wielkości fizycznych

```
struct plus_f
{    template <class T1, class T2>
   struct apply
   {   typedef typename mpl::plus<T1,T2>::type type;    };
};

template <class T, class D1, class D2>
quantity< T  ,
 typename mpl::transform<D1,D2,plus_f>::type>
operator*(quantity<T,D1> x, quantity<T,D2> y)
{
   typedef typename mpl::transform<D1,D2,plus_f>::type dim;
   return quantity<T,dim>( x.value() * y.value() );
}
```

Użycie już stworzonych wartości

Teraz możemy używać wartości i wykonywać na nich działania.

```
quantity<float,mass> m(5.0f);
quantity<float,acceleration> a(9.8f);
std::cout << "force = " << (m * a).value();
```

Ale powstanie problem przy:

```
quantity<float,force> f = m * a;
```

Częściowe rozwiązanie problemu

```
template <class T, class Dimensions>
struct quantity
{
   // converting constructor
   template <class OtherDimensions>
   quantity(quantity<T,OtherDimensions> const& rhs)
     : m_value(rhs.value())
   { }
```

Ale teraz legalne byłoby wyrażenie:

```
quantity<float,mass> bogus = m * a;
```

Rozwiązanie problemu

```
template <class OtherDimensions>
quantity(quantity<T,OtherDimensions> const& rhs)
  : m_value(rhs.value())
{
    BOOST_STATIC_ASSERT((
      mpl::equal<Dimensions,OtherDimensions>::type::value
    ));
}
```

Dzielenie wartości fizycznych

```
template <class D1, class D2>
struct divide_dimensions
  : mpl::transform<D1,D2,mpl::minus<_1,_2> > // forwarding again
{};

template <class T, class D1, class D2>
quantity<T, typename divide_dimensions<D1,D2>::type>
operator/(quantity<T,D1> x, quantity<T,D2> y)
{
   return quantity<T, typename divide_dimensions<D1,D2>::type>(
      x.value() / y.value());
}
```

Funkcje wyższego rzędu

Będziemy chcieli teraz zdefiniować *twice(f):*

twice(f) := f(f(x))

```
template <class F, class X>
struct twice
{
    typedef typename F::template apply<X>::type once;    // f(x)
    typedef typename F::template apply<once>::type type; // f(f(x))
};
```

Funkcje wyższego rzędu

Użycie twice:

```
struct add_pointer_f
{
    template <class T>
    struct apply : boost::add_pointer<T> {};
};

BOOST_STATIC_ASSERT((
    boost::is_same<
        twice<add_pointer_f, int>::type
      , int**
    >::value
));
```

Funkcje wyższego rzędu cd.

Ale chcielibyśmy móc używać twice w wygodniejszy sposób:

```
template <class X>
struct two_pointers
    : twice<boost::add_pointer<_1>, X>
{};
```

Definicja add_pointer z biblioteki mpl

```
template <class T>
struct add_pointer
{    typedef T* type;};
```

Lambda abstrakcja

```
template <class F, class X>
struct twice
   : apply1<
         typename mpl::lambda<F>::type
       , typename apply1<
             typename mpl::lambda<F>::type
           , X
         >::type
     >
{};
```

Kiedy używać metaprogramowania za pomocą szablonów?

- mechanizm makr jest niewystarczający
- gdy używamy fukcji rekurencyjnych o znanej liczbie obrotów
- gdy używamy pętli, które mogą być rozwinięte w czasie kompilacji
- gdy używamy stałych, które zależą od innych stałych zawartych w programie

Kiedy go nie używać?

- kiedy makro może zrobić to samo
- kiedy zależy na małym rozmiarze pliku wykonywalnego
- kiedy program już długo się kompiluje
- kiedy mocno zbliża się termin

Bibliografia

http://en.wikipedia.org/wiki/Template_metaprogramming
http://www.mywikinet.com/mpl/paper/html/
-opis biblioteki boost
http://www.informit.com/articles/article.asp?p=30667&rl=1
http://osl.iu.edu/~tveldhui/papers/pepm99/
http://www.boost.org/libs/mpl/doc/