Template Metaprogramming

Krzysztof Jusiak

Sep, 2011

Metaprogramming ?

- Metaprogramming - writing of computer programs that write or manipulate other programs (or themselves) as their data
- Template metaprogramming - metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled.

References

- C++ Template Metaprogramming by David Abrahams and Aleksey Gurtovoy
- Modern C++ Design by Andrei Alexandrescu
- Beyond the C++ Standard Library by Bjorn Karlsson
- C++ Coding Standards: 101 Rules, Guidelines, and Best Practices by Herb Sutter and Andrei Alexandrescu

Why metaprogramming ?

Pros:

- generic algorithms (libraries)
- execution speed (no virtual)
- type safety

Cons:

- compilation times (C++11 for the rescue)
- compilation errors (stlfilt / concepts)

Typename and template keyword ?

- use typename in case of type depends on parameter within template
- use template after

$$::$$

$$\rightarrow$$

$$.$$

  if expression before operator depends on template parameter and after operator name is an template

Typename and template keyword ?

```
template<typename T>
class A1 {
    typedef T::type type; //ERROR
};

template<typename T>
class A2 {
    typedef typename T::type type; //OK
};
```

Typename and template keyword ?

```
struct A {
    template<typename T> static void func() { }
};

template<typename T> class B1 {
    B1() {
        T::func<int>(); //ERROR
    }
};
template<typename T> class B2 {
    B2() {
        T::template func<int>(); //OK
    }
};
```

class vs typename ?

'class' and 'typename' means exactly the same:

```
template<class T> class Ex1 { };
template<typename T> class Ex2 { };
template<typename T1, typename T2 = typename T1::value_type> class Ex3 { };
template<template<class> class T1> class Ex4 { };
```

A point of instantiation

A point of instantiation (POI) is created when a code construct refers to a template specialization in such a way that the definition of the corresponding template needs to be instantiated to create that specialization. The POI is a point in the source where the substituted template could be inserted:

```
A<int>(); // POI

template class A<int>; //explicit POI
```

Generic programing techniques ?

- Concept checking - BOOST_CONCEPT_CHECK
- Algorithms - typename InputIterator
- Traits - boost::type_traits
- Tag dispatching - concept-based overloading
- Arbitrary Overloading - boost::enable_if
- Adaptors - std::stack (adopts container to provide stack)

Simple example - binary

```cpp
template<unsigned N>
struct binary {
    static const unsigned value = binary<N/10>::value * 2 + N%10;
};

template<>
struct binary<0> {
    static const unsigned value = 0;
};

std::clog << binary<100>::value;
// (((0) * 2 + 1) * 2 + 0) * 2 + 0 = 4
```

Metaprogramming idioms

- SFINAE Substitution Failure Is Not An Error (boost::enable_if, boost::type_traits)
- EBCO Empty Base Class Optimization
- CRTP Curiously Recurring Template Pattern
- PBTD Policy-Based Template Design (Loki)
- ET Expression Templates (boost::proto, boost::spirit)
- MPL Meta Programming Language (boost::mpl)

Substitution Failure Is Not An Error

```
template<typename T> class is_class {
    template<typename C>
    static yes test(int C::*);

    template<typename C>
    static no test(...);

public:
    static const bool value =
        sizeof(test<T>(0) == sizeof(yes));
};
```

Curiously Recurring Template Pattern

```
template<typename Derived>
class Base
{
public:
    void func() {
        static_cast<Derived*>(this)->impl();
    }
};

struct A { void impl() { } };

struct B { void impl() { } };
```

Policy-Based Template Design

```
template<
    typename Policy1 = DefaultPolicy1,
    typename Policy2 = DefaultPolicy2,
    typename Policy3 = DefaultPolicy3
>
class ClassWithPolicies { };

ClassWithPolicies<> l_classWithPolicies;
ClassWithPolicies<MyPolicy1> l_classWithPolicies;
ClassWithPolicies<MyPolicy1, MyPolicy2> ↩
    l_classWithPolicies;
```

Boost metaprogramming libraries (boost.org) ?

- Call Traits
- Concept Check
- Enable If
- Function Types
- Fusion
- MPL
- Parameter
- Proto
- Result Of
- Spirit
- Static Assert
- Tuple
- Type Traits
- Variant

Metaprogramming with c++98

A lot of preprocessor magic:

```
#define GENERATE_VECTOR(z, n, void_type)
BOOST_PP_COMMA_IF(n) typename T##n = void_type

#define GENERATE_VECTOR_TYPES(z, n, not_used)
BOOST_PP_COMMA_IF(n) typedef T##n type##n;

template
<BOOST_PP_REPEAT(10, GENRATE_VECTOR, none)>
struct Vector {
    BOOST_PP_REPEAT(10, GENRATE_VECTOR_TYPES, ~)
};
```

Metaprogramming improvements in C++11 / C++0x

New features:

```
variadic templates:
    template<typename... T> class Vector {
        typedef sizeof...(T) size;
    };

template aliases:
    template<typename, typename=int> class A { };
    template<typename T>
    using AliasForA = A<T, int>;

extern templates:
    extern template class A<void>;
```

Metaprogramming improvements - concepts

Not included in C++11 !

```
template<typename T>
Concept Simple {
    typedef T::type type;
    void func();
};

template<Simple T> class A1 { };
//or
template<typename T> requires<T, Simple>
class A2 { };
```

Testing with metaprogramming

```cpp
namespace Detail {
    class TDependecies { typedef F f; };

    template
    <
        typename T,
        typename Dependecies = TDependecies
    >
    class Yac { };
} // namespace Detail

template<typename T> struct Yac {
    typedef Detail::Yac<T> type;
};
```

Debugging with metaprogramming

Use warnings:

```
template<int Value = 0, typename T = void>
struct Print {
    unsigned : 80; //only in gcc
};

template<typename T> class A1
    : Print<__LINE__, T> // type size exceeded
{ };

template<typename T> class A2
    : boost::mpl::print<T> // sign comparison
{ };
```

Boost libraries

Almost of all boost libraries take advantage of metaprogramming.
Must know libraries (part of C++11):

- Static Assert
- Type Traits
- Enable If

Static Assert

Assert during compilation time:

```
template<bool> class assert;
template<> class assert<true> { };
```

Boost Static Assert

- BOOST_STATIC_ASSERT
- BOOST_STATIC_ASSERT_MSG
- BOOST_MPL_ASSERT
- BOOST_MPL_ASSERT_MSG

```
BOOST_STATIC_ASSERT((sizeof(T) == 1));
BOOST_MPL_ASSERT_MSG((sizeof(T) == 1), GivenTypeShouldHave1Byte, T);
```

Type Traits

A traits class provides a way of associating information with a compile-time entity

```
template <typename T>
struct is_void
{ static const bool value = false; };

template <>
struct is_void<void>
{ static const bool value = true; };
```

Type Traits

boost::remove_reference:

```
template<typename T>
struct remove_reference { typedef T type; }

template<>
struct remove_reference<T&> { typedef T type; }
```

Type Traits

boost::type_traits:

- add_const, add_cv, add_pointer, ...
- has_less, has_equal_to, ...
- is_const, is_same, is_base_of, ...
- remove_const, remove_cv, ...

Type Traits

boost::is_base_of:

```
class Base { };
class Derived : public Base { };

is_base_of<Base, Derived>::type  - true_type
is_base_of<Base, Derived>::value - true
is_base_of<Base, Base>::value    - true
```

Boost Enable If

If an invalid argument or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set instead of causing an compilation error

```
int func(int) {
    return 0;
}

template<typename T> typename T::type func(T) {
    return 0;
}
```

Boost Enable If

```
template<typename, typename Enable = void>
struct A { } {
    typedef int type;
}

template<typename T>
struct A<T, typename enable_if< is_base_of<Base, T>::type> > {
    typedef double type;
};
```

Boost Enable If

```
template<typename Seq, typename F>
void forEach(F,
 typename enable_if< empty<Seq> >::type* = 0)
{ }

template<typename Seq, typename F>
void forEach(F p_f,
 typename disable_if< empty<Seq> >::type* = 0)
{
    typedef typename front<Seq>::type front;
    p_f.template operator()<front>();
    forEach<typename pop_front<Seq>::type>(p_f);
}
```

Meta programming language

Concepts - Sequences:

- Forward Sequence (begin, end, size, empty, front) - vector, map
- Random Access Sequence
- Bidirectional Sequence
- Extensible Sequence
- Front Extensible Sequence
- Back Extensible Sequence
- Associative Sequence
- Extensible Associative Sequence
- Integral Sequence Wrapper
- Variadic Sequence

Meta programming language

Classes:

- vector - boost::mpl::vector¡int, double¿
- vector_c - boost::mpl::vector_c¡int, 3, 5¿
- set - boost::mpl::set¡int, double¿
- map - boost::mpl::map¡ boost::mpl::pair¡int, double¿ ¿

Meta programming language

Views:

- Transform View

```
typedef vector<int,long,char,char[50],double>
    types;
typedef max_element<
        transform_view< types, size_of<_> >
    >::type iter;

BOOST_MPL_ASSERT_RELATION( deref<iter>::type
    ::value, ==, 50 );
```

Meta programming language

Others views:

- empty_sequence
- filter_view
- iterator_range
- joint_view
- single_view
- transform_view
- zip_view

Meta programming language

Metafunctions:

- at, at_c
- begin, end, front, back
- insert, push_back, pop_back, pop_front, erase
- size, empty

Meta programming language

Metafunctions:

```
typedef range_c<long,10,50> range;

BOOST_MPL_ASSERT_RELATION(
    (at_c< range,0 >::type::value), ==, 10
);

typedef list0<> empty_list;

BOOST_MPL_ASSERT_RELATION(
    size<list>::value, ==, 0
);
```

Meta programming language

Iterators (next, advance, prior, defer, distance)

```
typedef vector_c<int,1> v;
typedef begin<v>::type first;
typedef end<v>::type last;

BOOST_MPL_ASSERT((
    is_same< next<first>::type, last >
));
```

Meta programming language

Algorithms:

- fold, reverse_fold
- find, find_if, contains, count, equal
- copy, copy_if, sort, unique, transform
- for_each (runtime)

Meta programming language

Algorithms:

```
typedef vector<long, float, int> types;

typedef fold
<
    types,
    int_<0>,
    if_< is_float<_2>, next<_1>, _1 >
>::type number_of_floats;

BOOST_MPL_ASSERT_RELATION(
    number_of_floats::value, ==, 1
);
```

Meta programming language

Type selection:

```
typedef if_<true_,char,long>::type t1;
typedef if_<false_,char,long>::type t2;

BOOST_MPL_ASSERT(( is_same<t1, char> ));
BOOST_MPL_ASSERT(( is_same<t2, long> ));
```

Meta programming language

Data types:

```
typedef int_<8> eight;
BOOST_MPL_ASSERT((
    is_same< eight::value_type, int > ));
BOOST_MPL_ASSERT((
    is_same< eight::type, eight > ));
BOOST_MPL_ASSERT((
    is_same< next< eight >::type, int_<9> > ));
BOOST_MPL_ASSERT((
    is_same< prior< eight >::type, int_<7> > ));
BOOST_MPL_ASSERT_RELATION(
    (eight::value), ==, 8 );
```

Exercise

```
class Base { };
class T1 : Base { };
class T2 { };
class T3 : Base { };

TEST(Exercise, Basic) {
    //given
    typedef boost::mpl::vector<T1, T2, T3> types;
    std::stringstream l_stream;
    //when
    forEach<types>(Print(l_stream));
    //then
    EXPECT_EQ("BTB", l_stream.str());
}
```

Exercise - Solution

```
#include <sstream>
#include <boost/type_traits/is_base_of.hpp>
#include <boost/mpl/vector.hpp>
#include <boost/mpl/empty.hpp>
#include <boost/mpl/front.hpp>
#include <boost/mpl/pop_front.hpp>
#include <boost/utility/enable_if.hpp>
```

Exercise - Solution

```
template<typename Seq, typename F>
void forEach(F,
 typename enable_if< empty<Seq> >::type* = 0)
{ }

template<typename Seq, typename F>
void forEach(F p_f,
 typename disable_if< empty<Seq> >::type* = 0)
{
    typedef typename front<Seq>::type front;
    p_f(front());
    forEach<typename pop_front<Seq>::type>(p_f);
}
```

Exercise - Solution

```
struct Print {
Print(std::stringstream& p_stream)
    : m_stream(p_stream) { }

template<typename T> void operator()(T, typename
 enable_if< is_base_of<Base, T> >::type* = 0) {
    m_stream << "B";
}
template<typename T> void operator()(T, typename
 disable_if< is_base_of<Base, T> >::type* = 0) {
    m_stream << "T";
}
std::stringstream& m_stream;
};
```

Questions

?