Template Meta Programming

Jonas R. Glesaaen
glesaaen@th.physik.uni-frankfurt.de

May 27th 2015

Literature

[1] Boost c++ library.
`http://www.boost.org`.

[2] C++ reference.
`http://cppreference.com`.

[3] D. Abrahams and A. Gurtovoy.
C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.
Pearson Education, 2004.

[4] A. Alexandrescu.
Modern C++ design.
Addison-Wesley, 2001.

What is Template Meta Programming?

Programming using the template interface of C++ so that certain common computations can be carried out at compile time.

The "language" is functional in nature, no mutable data.

Advantages:

- Reduce code duplication
- Increase readability
- Move error checks to compile time
- More sophisticated type checking and lookup

Looks familiar?

Type traits

```
template <class Itt>
void iterator_swap (Itt first, Itt second)
{
  typedef typename std::iterator_traits<Itt>::value_type ↩
      → iterator_deref_type;

  iterator_deref_type temp_value = *first;

  *first = *second;
  *second = temp_value;
}
```

Looks familiar?

enable_if (C++14)

```
template <
  class Itt,
  typename = std::enable_if_t<
    !std::is_same<
      typename std::iterator_traits<Itt>::value_type,
      void
     >::value
  >
>
void iterator_function (Itt first, Itt second)
{
  // ...
}
```

Recap: Template Specialisation

Heavily used in TMP to signal return paths and branch points for control structures.

iterator_traits

```
template <class Itt>
struct iterator_traits
{
  typedef typename Itt::value_type value_type;
};

template <class Type*>
struct iterator_traits
{
  typedef Type value_type;
};
```

When do you need `typename`?

`typename` is used to tell the compiler that what is coming up is a type. Used when you have a **dependent** name.

`typename` keyword

```
template <class Type>
typename traits_func<Type>::value_type //...
```

Exactly what `traits_func<Type>::value_type` is cannot be known at point of definition because of possible template specialisation. `typename` fixes that issue.

`::value_type` is said to be a dependent type.

When do you need `template`?

If the template class itself is a template, or has a template function, we need to tell the compiler.

`template` keyword

```
template <class Type, unsigned N>
void foo(int x)
{
  Type::function<N>(x);
};
```

which is interpreted as

```
(Type::function < N) > x;
```

When do you need `template`?

If the template class itself is a template, or has a template function, we need to tell the compiler.

`template` keyword

```
template <class Type, unsigned N>
void foo(int x)
{
  Type::template function<N>(x);
};
```

`template` is required when a **dependent** name access a template via `.`, `->` or `::`.

The Canonical Example

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

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

int main(int, char**)
{
  std::cout << Factorial<10>::value << std::endl;
}
```

The Canonical Example

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

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

int main(int, char**)
{
  std::cout << Factorial<10>::value << std::endl;
}
```

Runtime constant

Vocabulary

Metadata

A constant "value" accessible by calling `::value`

Metafunction

A function which takes its arguments as template arguments, and the result is stored in `::type`

```
some_metafunction<Arg1, Arg2>::type
```

Metafunction class

A function object that itself can be treated as a type. Function call accessed by a nested metafunction named `apply`

```
struct some_metafunction
{
  template <class Arg1, class Arg2>
  struct apply
  {
    // ...
  };
};
```

Example: Multiplication

```
template <int N>
struct integer
{
  constexpr static int value = N;
  typedef integer type;
};

template <class Arg1, class Arg2>
struct multiply
{
  typedef integer< Arg1::value * Arg2::value > type;
};

int main(int, argc**)
{
  typedef integer<5> five;
  typedef integer<-9> m_nine;

  std::cout << multiply<five,m_nine>::type::value
    << std::endl;
}
```

Example: Multiplication

```
template <int N>
struct integer
{
  constexpr static int value = N;
  typedef integer type;
};

template <class Arg1, class Arg2>
struct multiply
  : integer< Arg1::value * Arg2::value >
{};

int main(int, argc**)
{
  typedef integer<5> five;
  typedef integer<-9> m_nine;

  std::cout << multiply<five,m_nine>::type::value
    << std::endl;

  std::cout << multiply<five,m_nine>::value
    << std::endl;
}
```

Metafunction forwarding

Higher Order Metafunctions

As TMP inherently is a functional programming language, it is best at doing those kind of computations, computations with functions.

Let us implement the `nest` function so that:

```
nest(f,x,5) = f(f(f(f(f(x)))))
```

Assume that the `integer` and `multiply` still are defined as previous.

Higher Order Metafunctions

```
template <class F, class X, unsigned N>
struct nest
  : nest<F, typename F::template apply<X>::type, N-1>
{};

template <class F, class X>
struct nest <F,X,0>
  : X
{};

struct squared_f
{
  template <class Arg>
  struct apply
    : multiply<Arg,Arg>
  {};
};

int main(int, char**)
{
  typedef integer<5> five;
  nest<squared_f,five,3>::type::value; // ((5^2)^2)^2
}
```

Higher Order Metafunctions

```
template <class F, class X, unsigned N>
struct nest
  : nest<F, typename F::template apply<X>::type, N-1>
{};

template <class F, class X>
struct nest <F,X,0>
  : X
{};

struct squared_f
{
  template <class Arg>
  struct apply
    : multiply<Arg,Arg>
  {};
};

int main(int, char**)
{
  typedef integer<5> five;
  nest<squared_f,five,3>::type::value; // ((5^2)^2)^2
}
```

Template specialisation

Metafunction class

Higher Order Metafunctions

```
template <class F, class X, unsigned N>
struct nest
  : nest<F, typename F::template apply<X>::type, N-1>
{};

template <class F, class X>
struct nest <F,X,0>
  : X
{};

struct squared_f
{
  template <class Arg>
  struct apply
    : multiply<Arg,Arg>
  {};
};

int main(int, char**)
{
  typedef integer<5> five;
  nest<squared_f,five,3>::type::value; // ((5^2)^2)^2
}
```

The MPL boost library

Collection of useful types and definitions to simplify TMP

- Metadata wrappers:
  - bool_<b>, int_<N>, long_<N>, ...
- Arithmetic functions and logic operators:
  - plus<Arg1,Arg2>, times<Arg1,Arg2>, ...
  - less<Arg1,Arg2>, equal_to<Arg1,Arg2>, ...
  - and_<Arg>, or_<Arg>, nor_<Arg>

The MPL boost library

Collection of useful types and definitions to simplify TMP

- Metadata wrappers:
  - bool_<b>, int_<N>, long_<N>, ...
- Arithmetic functions and logic operators:
  - plus<Arg1,Arg2>, times<Arg1,Arg2>, ...
  - less<Arg1,Arg2>, equal_to<Arg1,Arg2>, ...
  - and_<Arg>, or_<Arg>, nor_<Arg>
- Lambda functions and placeholders

The MPL boost library

Collection of useful types and definitions to simplify TMP

- Metadata wrappers:
  - bool_<b>, int_<N>, long_<N>, ...
- Arithmetic functions and logic operators:
  - plus<Arg1,Arg2>, times<Arg1,Arg2>, ...
  - less<Arg1,Arg2>, equal_to<Arg1,Arg2>, ...
  - and_<Arg>, or_<Arg>, nor_<Arg>
- Lambda functions and placeholders
- Type selection
  - if_<Pred,Func1,Func2>, eval_if<Pred,Func1,Func2>

The MPL boost library

Collection of useful types and definitions to simplify TMP

- Metadata wrappers:
  - bool_<b>, int_<N>, long_<N>, ...
- Arithmetic functions and logic operators:
  - plus<Arg1,Arg2>, times<Arg1,Arg2>, ...
  - less<Arg1,Arg2>, equal_to<Arg1,Arg2>, ...
  - and_<Arg>, or_<Arg>, nor_<Arg>
- Lambda functions and placeholders
- Type selection
  - if_<Pred,Func1,Func2>, eval_if<Pred,Func1,Func2>
- Containers and iterators
  - vector<Arg1,Arg2,...,ArgN>, set<Arg1,Arg2,...,ArgN>, ...
  - next<It>, prior<It>, advance<It,N>, ...
- STL like algorithm library
  - transform<Seq,Fun>, copy_if<Seq,Pred>, ...

Lambda functions and placeholders

First!

We will assume that we have the following header on all our code to reduce the examples:

```
namespace mpl = boost::mpl;
using namespace mpl::placeholders;
```

If not, we would have to write the following everywhere we wanted an MPL placeholder:

```
boost::mpl::placeholders::_1,
boost::mpl::placeholders::_2,
boost::mpl::placeholders::_3, ...
```

which gets tedious...

Lambda functions and placeholders

Lambda functions are a signature part of any functional programming language and also go very well with STL like algorithms.

From our example earlier with `square_f<Arg>`, that function in itself seems a bit redundant as it can easily be written as `multiply<Arg,Arg>` with the same argument. But we run into two problems:

- The `multiply` function is a metafunction, while the `nest` function takes a metafunction class (a functor).
- We have no way of reducing `multiply`'s argument list to only take one argument

MPL's placeholders solve this!

Lambda functions and placeholders

With MPL lambda functions

```
template <class F, class X, unsigned N>
struct nest
  : nest<F, typename F::template apply<X>::type, N-1>
{};

template <class F, class X>
struct nest <F,X,0>
  : X
{};

int main(int, char**)
{
  typedef integer<5> five;
  nest<
    mpl::lambda< multiply<_1,_1> >::type,five,3
  >::type::value;
}
```

Lambda functions and placeholders

With mpl::apply and placeholders

```
template <class F, class X, unsigned N>
struct nest
  : nest<F, typename mpl::apply<F,X>::type, N-1>
{};

template <class F, class X>
struct nest <F,X,0>
  : X
{};

int main(int, char**)
{
  typedef integer<5> five;
  nest<multiply<_1,_1>,five,3>::type::value;
}
```

Control structures

Previously: Used template specialisation to switch between implementations

Simple template specialisation

```
template <class Type, bool FastImpl>
struct algorithm
{
  void operator() (const Type &)
  {
    // faster algorithm
  }
};

template <class Type>
struct algorithm<Type,false>
{
  void operator() (const Type &)
  {
    // safer algorithm
  }
};
```

Specialised for `FastImpl = false`

Control structures

With TMP we can do more sophisticated checks and switches

One more level of indirection

```
struct fast_algorithm
{
  template <class Itt1, class Itt2>
  static void execute(Itt1, Itt2);
};

struct safe_algorithm
{
  template <class Itt1, class Itt2>
  static void execute(Itt1, Itt2);
};
```

Control structures

With TMP we can do more sophisticated checks and switches

Choosing an implementation

```
struct algorithm
{
  template <class Itt1, class Itt2>
  static void execute(Itt1 i1, Itt2 i2)
  {
    mpl::if_<
      typename mpl::and_<
        is_random_access<Itt1>,
        is_random_access<Itt2>
      >::type,
      fast_algorithm,
      safe_algorithm
    >::type::execute(i1,i2);
  }
};
```

Containers and iterators

boost provides a complete STL like container and algorithm library.

Different containers have different access concepts

Forward sequence
begin<S>, end<S>, size<S>, front<S>
push_front<S,x>, pop_front<S>
insert<S,it,x>, erase<S,it>, clear<s>
Bidirectional sequence
..., back<S>, push_back<S,x>, pop_back<S>
Random access sequence
..., at<S,n>

All functions return new sequences because we have no mutable objects.

Containers and iterators: Short example

mpl::transform and mpl::vector

```
typedef mpl::vector<
  integer<3>, integer<7>, integer<-1> > my_vector;

typedef mpl::transform<
  my_vector,
  multiply<_1,_1>
>::type square_vector;

typedef mpl::begin<square_vector>::type begin;
typedef mpl::next<begin>::type next;

mpl::is_same<
  mpl::deref<next>::type,
  integer<49>
>::value;
```

$$\{\ 3,\ 7,\ -1\ \}$$

Containers and iterators: Short example

mpl::transform and mpl::vector

```
typedef mpl::vector<
  integer<3>, integer<7>, integer<-1> > my_vector;

typedef mpl::transform<
  my_vector,
  multiply<_1,_1>
>::type square_vector;

typedef mpl::begin<square_vector>::type begin;
typedef mpl::next<begin>::type next;

mpl::is_same<
  mpl::deref<next>::type,
  integer<49>
>::value;
```

$$\{\ 9,\ 49,\ 1\ \}$$

Containers and iterators: Short example

mpl::transform and mpl::vector

```
typedef mpl::vector<
  integer<3>, integer<7>, integer<-1> > my_vector;

typedef mpl::transform<
  my_vector,
  multiply<_1,_1>
>::type square_vector;

typedef mpl::begin<square_vector>::type begin;
typedef mpl::next<begin>::type next;

mpl::is_same<
  mpl::deref<next>::type,
  integer<49>
>::value;
```

$$\{\ 9,\ 49,\ 1\ \}$$

Containers and iterators: Short example

mpl::transform and mpl::vector

```
typedef mpl::vector<
  integer<3>, integer<7>, integer<-1> > my_vector;

typedef mpl::transform<
  my_vector,
  multiply<_1,_1>
>::type square_vector;

typedef mpl::begin<square_vector>::type begin;
typedef mpl::next<begin>::type next;

mpl::is_same<
  mpl::deref<next>::type,
  integer<49>
>::value;
```

$$\left\{ \; 9, \; 49, \; 1 \; \right\}$$

Containers and iterators: Short example

mpl::transform and mpl::vector

```
typedef mpl::vector<
  integer<3>, integer<7>, integer<-1> > my_vector;

typedef mpl::transform<
  my_vector,
  multiply<_1,_1>
>::type square_vector;

typedef mpl::begin<square_vector>::type begin;
typedef mpl::next<begin>::type next;

mpl::is_same<
  mpl::deref<next>::type,
  integer<49>
>::value;
```

true

Where to go from here?

Try it for yourself!

Where to go from here?

Try it for yourself!

- Try to write simple programs
  - Calculate an arithmetic sum
  - Sum up the elements of a vector
  - Implement your own for-loop
  - ...
- Study the literature
- Familiarise yourself with the boost MPL library
- See if you can make use of type switching in your own programs
- See if you can catch potential errors in your own programs

Summary

- We have seen how we can use the C++ template system to write metaprograms that look like normal programs.
- Metadata are types that contain their value in a public `::value` type.
- Metafunctions are called by their public `::type` type

```
some_metafunction<Arg1,Arg2,...,ArgN>::type
```

- Language facilitates a functional programming style with functions that manipulate other functions
- boost's MPL library implement a lot of useful metafuntions and types