3

A Deeper Look at Metafunctions

With the foundation laid so far, we're ready to explore one of the most basic uses for template metaprogramming techniques: adding static type checking to traditionally unchecked operations. We'll look at a practical example from science and engineering that can find applications in almost any numerical code. Along the way you'll learn some important new concepts and get a taste of metaprogramming at a high level using the MPL.

3.1 Dimensional Analysis

The first rule of doing physical calculations on paper is that the numbers being manipulated don't stand alone: most quantities have attached *dimensions,* to be ignored at our peril. As computations become more complex, keeping track of dimensions is what keeps us from inadvertently assigning a mass to what should be a length or adding acceleration to velocity—it establishes a type system for numbers.

Manual checking of types is tedious, and as a result, it's also error-prone. When human beings become bored, their attention wanders and they tend to make mistakes. Doesn't type checking seem like the sort of job a computer might be good at, though? If we could establish a framework of C++ types for dimensions and quantities, we might be able to catch errors in formulae before they cause serious problems in the real world.

Preventing quantities with different dimensions from interoperating isn't hard; we could simply represent dimensions as classes that only work with dimensions of the same type. What makes this problem interesting is that different dimensions *can* be combined, via multiplication or division, to produce arbitrarily complex new dimensions. For example, take Newton's law, which relates force to mass and acceleration:

$$F = ma$$

Since mass and acceleration have different dimensions, the dimensions of force must somehow capture their combination. In fact, the dimensions of acceleration are already just such a composite, a change in velocity over time:

$$dv/dt$$

Since velocity is just change in distance (l) over time (t), the fundamental dimensions of acceleration are:

$$(l/t)/t = l/t^2$$

And indeed, acceleration is commonly measured in "meters per second squared." It follows that the dimensions of force must be:

$$ml/t^2$$

and force is commonly measured in kg(m/s^2), or "kilogram-meters per second squared." When multiplying quantities of mass and acceleration, we multiply their dimensions as well and carry the result along, which helps us to ensure that the result is meaningful. The formal name for this bookkeeping is **dimensional analysis**, and our next task will be to implement its rules in the C++ type system. John Barton and Lee Nackman were the first to show how to do this in their seminal book, *Scientific and Engineering C++* [BN94]. We will recast their approach here in metaprogramming terms.

3.1.1 Representing Dimensions

An international standard called *Système International d'Unites* breaks every quantity down into a combination of the dimensions *mass, length* or *position, time, charge, temperature, intensity,* and *angle.* To be reasonably general, our system would have to be able to represent seven or more fundamental dimensions. It also needs the ability to represent composite dimensions that, like *force,* are built through multiplication or division of the fundamental ones.

In general, a composite dimension is the product of powers of fundamental dimensions.[1] If we were going to represent these powers for manipulation at runtime, we could use an array of seven `int`s, with each position in the array holding the power of a different fundamental dimension:

```
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};
...
```

In that representation, force would be:

```
dimension const force  = {1, 1, -2, 0, 0, 0, 0};
```

1. Divisors just contribute negative exponents, since $1/x = x^{-1}$.

that is, mlt^{-2}. However, if we want to get dimensions into the type system, these arrays won't do the trick: they're all the same type! Instead, we need types that *themselves* represent sequences of numbers, so that two masses have the same type and a mass is a different type from a length.

Fortunately, the MPL provides us with a collection of **type sequences**. For example, we can build a sequence of the built-in signed integral types this way:

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

typedef boost::mpl::vector<
     signed char, short, int, long> signed_types;
```

How can we use a type sequence to represent numbers? Just as numerical metafunctions pass and return wrapper *types* having a nested `::value`, so numerical sequences are really sequences of wrapper types (another example of polymorphism). To make this sort of thing easier, MPL supplies the `int_<N>` class template, which presents its integral argument as a nested `::value`:

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

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

Namespace Aliases

`namespace` *alias* = *namespace-name*`;`

declares *alias* to be a synonym for *namespace-name*. Many examples in this book will use `mpl::` to indicate `boost::mpl::`, but will omit the alias that makes it legal C++.

In fact, the library contains a whole suite of integral constant wrappers such as `long_` and `bool_`, each one wrapping a different type of integral constant within a class template.

Now we can build our fundamental dimensions:

```
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;

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

Whew! That's going to get tiring pretty quickly. Worse, it's hard to read and verify: The essential information, the powers of each fundamental dimension, is buried in repetitive syntactic "noise." Accordingly, MPL supplies **integral sequence wrappers** that allow us to write:

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

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;
```

Even though they have different types, you can think of these `mpl::vector_c` specializations as being equivalent to the more verbose versions above that use `mpl::vector`.

If we want, we can also define a few composite dimensions:

```
// base dimension:        m l  t ...
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/(t^2)
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/(t^2)
```

And, incidentally, the dimensions of scalars (like pi) can be described as:

```
typedef mpl::vector_c<int,0,0,0,0,0,0,0> scalar;
```

3.1.2 Representing Quantities

The types listed above are still pure metadata; to typecheck real computations we'll need to somehow bind them to our runtime data. A simple numeric value wrapper, parameterized on the number type `T` and on its dimensions, fits the bill:

```
template <class T, class Dimensions>
struct quantity
{
```

```
    explicit quantity(T x)
       : m_value(x)
    {}

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

Now we have a way to represent numbers associated with dimensions. For instance, we can say:

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

Note that `Dimensions` doesn't appear anywhere in the definition of `quantity` outside the template parameter list; its *only* role is to ensure that `l` and `m` have different types. Because they do, we cannot make the mistake of assigning a length to a mass:

```
m = l;    // compile time type error
```

3.1.3 Implementing Addition and Subtraction

We can now easily write the rules for addition and subtraction, since the dimensions of the arguments must always match.

```
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());
}
```

These operators enable us to write code like:

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

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

but prevent us from trying to add incompatible dimensions:

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

3.1.4 Implementing Multiplication

Multiplication is a bit more complicated than addition and subtraction. So far, the dimensions of the arguments and results have all been identical, but when multiplying, the result will usually have different dimensions from either of the arguments. For multiplication, the relation:

$$\left(x^a\right)\left(x^b\right) = x^{(a+b)}$$

implies that the exponents of the result dimensions should be the sum of corresponding exponents from the argument dimensions. Division is similar, except that the sum is replaced by a **difference**.

To combine corresponding elements from two sequences, we'll use MPL's `transform` algorithm. `transform` is a metafunction that iterates through two input sequences in parallel, passing an element from each sequence to an arbitrary binary metafunction, and placing the result in an output sequence.

```
template <class Sequence1, class Sequence2, class BinaryOperation>
struct transform;  // returns a Sequence
```

The signature above should look familiar if you're acquainted with the STL `transform` algorithm that accepts two *runtime* sequences as inputs:

```
template <
    class InputIterator1, class InputIterator2
  , class OutputIterator, class BinaryOperation
>
void transform(
    InputIterator1 start1, InputIterator2 finish1
  , InputIterator2 start2
  , OutputIterator result, BinaryOperation func);
```

Now we just need to pass a `BinaryOperation` that adds or subtracts in order to multiply or divide dimensions with `mpl::transform`. If you look through the MPL reference manual, you'll come across `plus` and `minus` metafunctions that do just what you'd expect:

```
#include <boost/static_assert.hpp>
#include <boost/mpl/plus.hpp>
#include <boost/mpl/int.hpp>
namespace mpl = boost::mpl;

BOOST_STATIC_ASSERT((
    mpl::plus<
        mpl::int_<2>
      , mpl::int_<3>
    >::type::value == 5
));
```

BOOST_STATIC_ASSERT

is a macro that causes a compilation error if its argument is false. The double parentheses are required because the C++ preprocessor can't parse templates: it would otherwise be fooled by the comma into treating the condition as two separate macro arguments. Unlike its runtime analogue `assert(...)`, `BOOST_STATIC_ASSERT` can also be used at class scope, allowing us to put assertions in our metafunctions. See Chapter 8 for an in-depth discussion.

At this point it might seem as though we have a solution, but we're not quite there yet. A naive attempt to apply the `transform` algorithm in the implementation of `operator*` yields a compiler error:

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

template <class T, class D1, class D2>
quantity<
    T
  , typename mpl::transform<D1,D2,mpl::plus>::type
>
operator*(quantity<T,D1> x, quantity<T,D2> y) { ... }
```

It fails because the protocol says that metafunction arguments must be types, and `plus` is not a type, but a class template. Somehow we need to make metafunctions like `plus` fit the metadata mold.

One natural way to introduce polymorphism between metafunctions and metadata is to employ the wrapper idiom that gave us polymorphism between types and integral constants. Instead of a nested integral constant, we can use a class template nested within a **metafunction class**:

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

Definition

A **metafunction class** is a class with a publicly accessible nested metafunction called `apply`.

Whereas a metafunction is a template but not a type, a metafunction class wraps that template within an ordinary non-templated class, which *is* a type. Since metafunctions operate on and return types, a metafunction class can be passed as an argument to, or returned from, another metafunction.

Finally, we have a `BinaryOperation` type that we can pass to `transform` without causing a compilation error:

```
template <class T, class D1, class D2>
quantity<
    T
  , typename mpl::transform<D1,D2,plus_f>::type  // new dimensions
>
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() );
}
```

Now, if we want to compute the force exerted by gravity on a five kilogram laptop computer, that's just the acceleration due to gravity (9.8 m/sec^2) times the mass of the laptop:

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

Our `operator*` multiplies the runtime values (resulting in 6.0f), and our metaprogram code uses `transform` to sum the meta-sequences of fundamental dimension exponents, so that the result type contains a representation of a new list of exponents, something like:

```
vector_c<int,1,1,-2,0,0,0,0>
```

However, if we try to write:

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

we'll run into a little problem. Although the result of `m * a` does indeed represent a force with exponents of mass, length, and time 1, 1, and -2 respectively, the type returned by `transform` isn't a specialization of `vector_c`. Instead, `transform` works generically on the elements of its inputs and builds a new sequence with the appropriate elements: a type with many of the same sequence properties as `vector_c<int,1,1,-2,0,0,0,0>`, but with a different C++ type altogether. If you want to see the type's full name, you can try to compile the example yourself and look at the error message, but the exact details aren't important. The point is that `force` names a different type, so the assignment above will fail.

In order to resolve the problem, we can add an implicit conversion from the multiplication's result type to `quantity<float,force>`. Since we can't predict the exact types of the dimensions involved in any computation, this conversion will have to be templated, something like:

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

Unfortunately, such a general conversion undermines our whole purpose, allowing nonsense such as:

```
// Should yield a force, not a mass!
quantity<float,mass> bogus = m * a;
```

We can correct that problem using another MPL algorithm, `equal`, which tests that two sequences have the same elements:

```
template <class OtherDimensions>
quantity(quantity<T,OtherDimensions> const& rhs)
  : m_value(rhs.value())
```

```
{
    BOOST_STATIC_ASSERT((
       mpl::equal<Dimensions,OtherDimensions>::type::value
    ));
}
```

Now, if the dimensions of the two quantities fail to match, the assertion will cause a compilation error.

3.1.5 Implementing Division

Division is similar to multiplication, but instead of adding exponents, we must subtract them. Rather than writing out a near duplicate of `plus_f`, we can use the following trick to make `minus_f` much simpler:

```
struct minus_f
{
    template <class T1, class T2>
    struct apply
      : mpl::minus<T1,T2> {};
};
```

Here `minus_f::apply` uses inheritance to expose the nested `type` of its base class, `mpl::minus`, so we don't have to write:

```
typedef typename ...::type type
```

We don't have to write `typename` here (in fact, it would be illegal), because the compiler knows that dependent names in `apply`'s initializer list must be base classes.[2] This powerful simplification is known as **metafunction forwarding**; we'll apply it often as the book goes on.[3]

Syntactic tricks notwithstanding, writing trivial classes to wrap existing metafunctions is going to get boring pretty quickly. Even though the definition of `minus_f` was far less verbose than that of `plus_f`, it's still an awful lot to type. Fortunately, MPL gives us a *much* simpler way to pass metafunctions around. Instead of building a whole metafunction class, we can invoke `transform` this way:

2. In case you're wondering, the same approach could have been applied to `plus_f`, but since it's a little subtle, we introduced the straightforward but verbose formulation first.
3. Users of EDG-based compilers should consult Appendix C for a caveat about metafunction forwarding. You can tell whether you have an EDG compiler by checking the preprocessor symbol `__EDG_VERSION__`, which is defined by all EDG-based compilers.

```
typename mpl::transform<D1,D2, mpl::minus<_1,_2> >::type
```

Those funny looking arguments (_1 and _2) are known as **placeholders**, and they signify that when the `transform`'s `BinaryOperation` is invoked, its first and second arguments will be passed on to `minus` in the positions indicated by _1 and _2, respectively. The whole type `mpl::minus<_1,_2>` is known as a **placeholder expression**.

> **Note**
>
> MPL's placeholders are in the `mpl::placeholders` namespace and defined in `boost/mpl/placeholders.hpp`. In this book we will usually assume that you have written:
>
> ```
> #include<boost/mpl/placeholders.hpp>
> using namespace mpl::placeholders;
> ```
>
> so that they can be accessed without qualification.

Here's our division operator written using placeholder expressions:

```
template <class T, class D1, class D2>
quantity<
    T
  , typename mpl::transform<D1,D2,mpl::minus<_1,_2> >::type
>
operator/(quantity<T,D1> x, quantity<T,D2> y)
{
   typedef typename
     mpl::transform<D1,D2,mpl::minus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() / y.value() );
}
```

This code is considerably simpler. We can simplify it even further by factoring the code that calculates the new dimensions into its own metafunction:

```
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());
}
```

Now we can verify our "force-on-a-laptop" computation by reversing it, as follows:

```
quantity<float,mass> m2 = f/a;
float rounding_error = std::abs((m2 - m).value());
```

If we got everything right, `rounding_error` should be very close to zero. These are boring calculations, but they're just the sort of thing that could ruin a whole program (or worse) if you got them wrong. If we had written `a/f` instead of `f/a`, there would have been a compilation error, preventing a mistake from propagating throughout our program.

3.2 Higher-Order Metafunctions

In the previous section we used two different forms—metafunction classes and placeholder expressions—to pass and return metafunctions just like any other metadata. Bundling metafunctions into "first class metadata" allows `transform` to perform an infinite variety of different operations: in our case, multiplication and division of dimensions. Though the idea of using functions to manipulate other functions may seem simple, its great power and flexibility [Hudak89] has earned it a fancy title: **higher-order functional programming**. A function that operates on another function is known as a **higher-order function**. It follows that `transform` is a higher-order metafunction: a metafunction that operates on another metafunction.

Now that we've seen the power of higher-order metafunctions at work, it would be good to be able to create new ones. In order to explore the basic mechanisms, let's try a simple example. Our task is to write a metafunction called `twice`, which—given a unary metafunction f and arbitrary metadata x—computes:

$$twice(f, x) := f(f(x)) .$$

This might seem like a trivial example, and in fact it is. You won't find much use for `twice` in real code. We hope you'll bear with us anyway: Because it doesn't do much more than accept and invoke a metafunction, `twice` captures all the essential elements of "higher-orderness" without any distracting details.

If f is a metafunction class, the definition of `twice` is straightforward:

```
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))
};
```

Or, applying metafunction forwarding:

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

> **C++ Language Note**
>
> The C++ standard requires the `template` keyword when we use a **dependent name** that refers to a member template. `F::apply` may or may not name a template, *depending* on the particular `F` that is passed. See Appendix B for more information about `template`.

Given the need to sprinkle our code with the `template` keyword, it would be nice to reduce the syntactic burden of invoking metafunction classes. As usual, the solution is to factor the pattern into a metafunction:

```
template <class UnaryMetaFunctionClass, class Arg>
struct apply1
  : UnaryMetaFunctionClass::template apply<Arg>
{};
```

Now `twice` is just:

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

To see `twice` at work, we can apply it to a little metafunction class built around the `add_pointer` metafunction:

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

Now we can use `twice` with `add_pointer_f` to build pointers-to-pointers:

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

3.3 Handling Placeholders

Our implementation of `twice` already works with metafunction classes. Ideally, we would like it to work with placeholder expressions, too, much the same as `mpl::transform` allows us to pass either form. For example, we would like to be able to write:

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

But when we look at the implementation of `boost::add_pointer`, it becomes clear that the current definition of `twice` can't work that way.

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

To be invokable by `twice`, `boost::add_pointer<_1>` would have to be a metafunction class, along the lines of `add_pointer_f`. Instead, it's just a nullary metafunction returning the almost senseless type `_1*`. Any attempt to use `two_pointers` will fail when `apply1` reaches for a nested `::apply` metafunction in `boost::add_pointer<_1>` and finds that it doesn't exist.

We've determined that we don't get the behavior we want automatically, so what next? Since `mpl::transform` can do this sort of thing, there ought to be a way for us to do it too—and so there is.

3.3.1 The `lambda` Metafunction

We can *generate* a metafunction class from `boost::add_pointer<_1>`, using MPL's `lambda` metafunction:

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

BOOST_STATIC_ASSERT((
    boost::is_same<
        typename two_pointers<int>::type
      , int**
    >::value
));
```

We'll refer to metafunction classes like `add_pointer_f` and placeholder expressions like `boost::add_pointer<_1>` as **lambda expressions**. The term, meaning "unnamed function object," was introduced in the 1930s by the logician Alonzo Church as part of a fundamental theory of computation he called the *lambda-calculus*.[4] MPL uses the somewhat obscure word `lambda` because of its well-established precedent in functional programming languages.

Although its primary purpose is to turn placeholder expressions into metafunction classes, `mpl::lambda` can accept any lambda expression, even if it's already a metafunction class. In that case, `lambda` returns its argument unchanged. MPL algorithms like `transform` call `lambda` internally, before invoking the resulting metafunction class, so that they work equally well with either kind of lambda expression. We can apply the same strategy to `twice`:

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

4. See http://en.wikipedia.org/wiki/Lambda_calculus for an in-depth treatment, including a reference to Church's paper proving that the equivalence of lambda expressions is in general not decidable.

Now we can use `twice` with metafunction classes *and* placeholder expressions:

```
int* x;

twice<add_pointer_f, int>::type          p = &x;
twice<boost::add_pointer<_1>, int>::type q = &x;
```

3.3.2 The `apply` Metafunction

Invoking the result of `lambda` is such a common pattern that MPL provides an `apply` metafunction to do just that. Using `mpl::apply`, our flexible version of `twice` becomes:

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

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

You can think of `mpl::apply` as being just like the `apply1` template that we wrote, with two additional features:

1. While `apply1` operates only on metafunction classes, the first argument to `mpl::apply` can be any lambda expression (including those built with placeholders).
2. While `apply1` accepts only one additional argument to which the metafunction class will be applied, `mpl::apply` can invoke its first argument on any number from zero to five additional arguments.[5] For example:

```
// binary lambda expression applied to 2 additional arguments
mpl::apply<
    mpl::plus<_1,_2>
  , mpl::int_<6>
  , mpl::int_<7>
>::type::value // == 13
```

Guideline

When writing a metafunction that invokes one of its arguments, use `mpl::apply` so that it works with lambda expressions.

5. See the Configuration Macros section of the MPL reference manual for a description of how to change the maximum number of arguments handled by `mpl::apply`.

3.4 More Lambda Capabilities

Lambda expressions provide much more than just the ability to pass a metafunction as an argument. The two capabilities described next combine to make lambda expressions an invaluable part of almost every metaprogramming task.

3.4.1 Partial Metafunction Application

Consider the lambda expression `mpl::plus<_1,_1>`. A single argument is directed to both of `plus`'s parameters, thereby adding a number to itself. Thus, a *binary* metafunction, `plus`, is used to build a *unary* lambda expression. In other words, we've created a whole new computation! We're not done yet, though: By supplying a non-placeholder as one of the arguments, we can build a unary lambda expression that adds a fixed value, say 42, to its argument:

```
mpl::plus<_1, mpl::int_<42> >
```

The process of binding argument values to a subset of a function's parameters is known in the world of functional programming as **partial function application**.

3.4.2 Metafunction Composition

Lambda expressions can also be used to assemble more interesting computations from simple metafunctions. For example, the following expression, which multiplies the sum of two numbers by their difference, is a **composition** of the three metafunctions `multiplies`, `plus`, and `minus`:

```
mpl::multiplies<mpl::plus<_1,_2>, mpl::minus<_1,_2> >
```

When evaluating a lambda expression, MPL checks to see if any of its arguments are themselves lambda expressions, and evaluates each one that it finds. The results of these inner evaluations are substituted into the outer expression before it is evaluated.

3.5 Lambda Details

Now that you have an idea of the semantics of MPL's `lambda` facility, let's formalize our understanding and look at things a little more deeply.

3.5.1 Placeholders

The definition of "placeholder" may surprise you:

Definition

A **placeholder** is a metafunction class of the form `mpl::arg<X>`.

3.5.1.1 Implementation

The convenient names `_1`, `_2`,... `_5` are actually `typedef`s for specializations of `mpl::arg` that simply select the *N*th argument for any *N*.[6] The implementation of placeholders looks something like this:

```
namespace boost { namespace mpl { namespace placeholders {

template <int N> struct arg; // forward declarations
struct void_;

template <>
struct arg<1>
{
    template <
      class A1, class A2 = void_, ... class Am = void_>
    struct apply
    {
        typedef A1 type; // return the first argument
    };
};
typedef arg<1> _1;

template <>
struct arg<2>
{
    template <
      class A1, class A2, class A3 = void_, ...class Am = void_
    >
    struct apply
    {
        typedef A2 type; // return the second argument
    };
};
```

6. MPL provides five placeholders by default. See the Configuration Macros section of the MPL reference manual for a description of how to change the number of placeholders provided.

```
typedef arg<2> _2;

more specializations and typedefs...

}}}
```

Remember that invoking a metafunction class is the same as invoking its nested `apply` metafunction. When a placeholder in a lambda expression is evaluated, it is invoked on the expression's actual arguments, returning just one of them. The results are then substituted back into the lambda expression and the evaluation process continues.

3.5.1.2 The Unnamed Placeholder

There's one special placeholder, known as the **unnamed placeholder**, that we haven't yet defined:

```
namespace boost { namespace mpl { namespace placeholders {

typedef arg<-1> _; // the unnamed placeholder

}}}
```

The details of its implementation aren't important; all you really need to know about the unnamed placeholder is that it gets special treatment. When a lambda expression is being transformed into a metafunction class by `mpl::lambda`,

> the *n*th appearance of the unnamed placeholder *in a given template specialization* is replaced with *_n*.

So, for example, every row of Table 3.1 contains two equivalent lambda expressions.

Table 3.1 Unnamed Placeholder Semantics

`mpl::plus<_,_>`	`mpl::plus<_1,_2>`

`boost::is_same<`
`  _`
`  , boost::add_pointer<_>`
`>`
	`boost::is_same<`
	`  _1`
	`  , boost::add_pointer<_1>`
	`>`

`mpl::multiplies<`
`  mpl::plus<_,_>`
`  , mpl::minus<_,_>`
`>`
	`mpl::multiplies<`
	`  mpl::plus<_1,_2>`
	`  , mpl::minus<_1,_2>`
	`>`

Especially when used in simple lambda expressions, the unnamed placeholder often eliminates just enough syntactic "noise" to significantly improve readability.

3.5.2 Placeholder Expression Definition

Now that you know just what *placeholder* means, we can define **placeholder expression**:

> **Definition**
> A placeholder expression is either:
>
> - a placeholder
>
> *or*
>
> - a template specialization with at least one argument that is a placeholder expression.

In other words, a placeholder expression always involves a placeholder.

3.5.3 Lambda and Non-Metafunction Templates

There is just one detail of placeholder expressions that we haven't discussed yet. MPL uses a special rule to make it easier to integrate ordinary templates into metaprograms: After all of the placeholders have been replaced with actual arguments, if the resulting template specialization *X* doesn't have a nested `::type`, the result of lambda is just *X* itself.

For example, `mpl::apply<std::vector<_>, T>` is always just `std::vector<T>`. If it weren't for this behavior, we would have to build trivial metafunctions to create ordinary template specializations in lambda expressions:

```
// trivial std::vector generator
template<class U>
struct make_vector { typedef std::vector<U> type; };
typedef mpl::apply<make_vector<_>, T>::type vector_of_t;
```

Instead, we can simply write:

```
typedef mpl::apply<std::vector<_>, T>::type vector_of_t;
```

3.5.4 The Importance of Being Lazy

Recall the definition of `always_int` from the previous chapter:

```
struct always_int
{
    typedef int type;
};
```

Nullary metafunctions might not seem very important at first, since something like `add_pointer<int>` could be replaced by `int*` in any lambda expression where it appears. Not all nullary metafunctions are that simple, though:

```
struct add_pointer_f
{
    template <class T>
    struct apply : boost::add_pointer<T> {};
};
typedef mpl::vector<int, char*, double&> seq;
typedef mpl::transform<seq, boost::add_pointer<_> > calc_ptr_seq;
```

Note that `calc_ptr_seq` is a nullary metafunction, since it has `transform`'s nested `::type`. A C++ template is not instantiated until we actually "look inside it," though. Just naming `calc_ptr_seq` does not cause it to be evaluated, since we haven't accessed its `::type` yet.

Metafunctions can be invoked *lazily,* rather than immediately upon supplying all of their arguments. We can use **lazy evaluation** to improve compilation time when a metafunction result is only going to be used conditionally. We can sometimes also avoid contorting program structure by *naming* an invalid computation without actually performing it. That's what we've done with `calc_ptr_seq` above, since you can't legally form `double&*`. Laziness and all of its virtues will be a recurring theme throughout this book.

3.6 Details

By now you should have a fairly complete view of the fundamental concepts and language of both template metaprogramming in general and of the Boost Metaprogramming Library. This section reviews the highlights.

Metafunction forwarding. The technique of using public derivation to supply the nested `type` of a metafunction by accessing the one provided by its base class.

Metafunction class. The most basic way to formulate a compile-time function so that it can be treated as polymorphic metadata; that is, as a type. A metafunction class is a class with a nested metafunction called `apply`.

MPL. Most of this book's examples will use the Boost Metaprogramming Library. Like the Boost type traits headers, MPL headers follow a simple convention:

```
#include <boost/mpl/component-name.hpp>
```

If the component's name ends in an underscore, however, the corresponding MPL header name does not include the trailing underscore. For example, `mpl::bool_` can be found in `<boost/mpl/bool.hpp>`. Where the library deviates from this convention, we'll be sure to point it out to you.

Higher-order function. A function that operates on or returns a function. Making metafunctions polymorphic with other metadata is a key ingredient in higher-order metaprogramming.

Lambda expression. Simply put, a lambda expression is callable metadata. Without some form of callable metadata, higher-order metafunctions would be impossible. Lambda expressions have two basic forms: *metafunction classes* and *placeholder expressions.*

Placeholder expression. A kind of lambda expression that, through the use of placeholders, enables in-place *partial metafunction application* and *metafunction composition.* As you will see throughout this book, these features give us the truly amazing ability to build up almost any kind of complex type computation from more primitive metafunctions, right at its point of use:

```
// find the position of a type x in some_sequence such that:
//         x is convertible to 'int'
//      && x is not 'char'
//      && x is not a floating type
typedef mpl::find_if<
      some_sequence
    , mpl::and_<
          boost::is_convertible<_1,int>
        , mpl::not_<boost::is_same<_1,char> >
        , mpl::not_<boost::is_float<_1> >
      >
    >::type iter;
```

Placeholder expressions make good on the promise of algorithm reuse without forcing us to write new metafunction classes. The corresponding capability is often sorely missed in the runtime world of the STL, since it is often much easier to write a loop by hand than it is to use standard algorithms, despite their correctness and efficiency advantages.

The `lambda` metafunction. A metafunction that transforms a lambda expression into a corresponding metafunction class. For detailed information on `lambda` and the lambda evaluation process, please see the MPL reference manual.

The `apply` metafunction. A metafunction that invokes its first argument, which must be a lambda expression, on its remaining arguments. In general, to invoke a lambda expression, you should always pass it to `mpl::apply` along with the arguments you want to apply it to in lieu of using `lambda` and invoking the result "manually."

Lazy evaluation. A strategy of delaying evaluation until a result is required, thereby avoiding any unnecessary computation and any associated unnecessary errors. Metafunctions are only invoked when we access their nested `::type`s, so we can supply all of their arguments without performing any computation and delay evaluation to the last possible moment.

3.7 Exercises

3-0. Use `BOOST_STATIC_ASSERT` to add error checking to the `binary` template presented in section 1.4.1, so that `binary<N>::value` causes a compilation error if `N` contains digits other than `0` or `1`.

3-1. Turn `vector_c<int,1,2,3>` into a type sequence with elements (2,3,4) using `transform`.

3-2. Turn `vector_c<int,1,2,3>` into a type sequence with elements (1,4,9) using `transform`.

3-3. Turn `T` into `T****` by using `twice` twice.

3-4. Turn `T` into `T****` using `twice` on itself.

3-5. There's still a problem with the dimensional analysis code in section 3.1. Hint: What happens when you do:

```
f = f + m * a;
```

Repair this example using techniques shown in this chapter.

3-6. Build a lambda expression that has functionality equivalent to `twice`. Hint: `mpl::apply` is a metafunction!

3-7*. What do you think would be the semantics of the following constructs:

```
typedef mpl::lambda<mpl::lambda<_1> >::type t1;
typedef mpl::apply<_1,mpl::plus<_1,_2> >::type t2;
typedef mpl::apply<_1,std::vector<int> >::type t3;
```

```
typedef mpl::apply<_1,std::vector<_1> >::type t4;
typedef mpl::apply<mpl::lambda<_1>,std::vector<int> >::type  t5;
typedef mpl::apply<mpl::lambda<_1>,std::vector<_1> >::type t6;
typedef mpl::apply<mpl::lambda<_1>,mpl::plus<_1,_2> >::type t7;
typedef mpl::apply<_1,mpl::lambda< mpl::plus<_1,_2> > >::type t8;
```

Show the steps used to arrive at your answers and write tests verifying your assumptions. Did the library behavior match your reasoning? If not, analyze the failed tests to discover the actual expression semantics. Explain why your assumptions were different, what behavior you find more coherent, and why.

3-8*. Our dimensional analysis framework dealt with dimensions, but it entirely ignored the issue of *units*. A length can be represented in inches, feet, or meters. A force can be represented in newtons or in kg m/sec^2. Add the ability to specify units and test your code. Try to make your interface as syntactically friendly as possible for the user.