Dude, Meta.

So, I usually pretend to have some pretense of general applicability for these techniques. Today, I'm abandoning that. The only way I can possibly think of motivating the first half of today's lecture is to say something about the computational power of templates. So, let's start with some

Terminology

I already addressed this some, but let's formalize some notions. (Some of this I'm making up/generalizing from current practice. Others are very real terms. I'm putting what I think are my coinages in quotes.) The "**metavalues**" of metaprogramming are class-struct-or-union types and their templatized forms. Also, we include anything that can be interpreted as an integer of some size: integral types, enums, function pointers, member function pointers, etc. A **metafunction** is a template structure or class (or sometimes just a class) that takes some number of template parameters and "returns" some number of types, templates, or values (usually 1) through enumeration or through typedefs, or clever rebinding.

```
template <class T>
struct add_const
{
      typedef const T type;
};

template <int A, int B>
struct product
{
      enum {value = A * B};
};

struct give_me_product
{
      template <int A, int B>
      struct apply
      {
            typedef product<A,B> type;
      };
};
```

We also discussed recursion, which is achieved mostly through template specialization:

```
template <unsigned int X>
struct factorial
{
      enum { value = X * factorial<X-1>::value};
};

template <>
struct factorial<0>
{
```

```
        enum { value = 1};
};
```

Prime Numbers!

Ok, so those are the main concepts. And so we're going to jump right into my favorite metaprogram, which spits out prime numbers as error messages. So let's begin:

```
template <int p, int i>
struct is_prime
{
        enum { value = (p ==2 ) || (p%i) && is_prime<(i>2?p:0),i-1>::value};
};

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

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

// Set up a wrapper class.
template <int i>
struct D
{
        D(void*) {}
};

template <int i>
struct prime_printer
{
        prime_printer<i-1> a;
        void f()
        {
                D<i> d = is_prime<i,i-1>::value ? 1 : 0;
                a.f();
        }
};

int main()
{
        prime_printer<42> a; // or some suitable number
        a.f();
}
```

The main trick is the initialization of d from 1 or 0. The only number that can be implicitly recast as a pointer is 0, which leads to the behavior we want.

Dimensional Analysis

This section is wholly copied from <http://www.boost.org/libs/mpl/doc/tutorial/dimensional-analysis.html>

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.

Representing Dimensions

An international standard called *Système International d'Unites* (SI), 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};
```

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;
```

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;
```

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;
```

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 );
```

Addition

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
```

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:

$$(x^a)(x^b) == x^{(a+b)}$$

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 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
));
```

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 exterted by gravity on a 5 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:

```
mpl::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 `mpl::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.