Chapter 17

Metaprograms

Metaprogramming consists of "programming a program." In other words, we lay out code that the programming system executes to generate new code that implements the functionality we really want. Usually the term *metaprogramming* implies a reflexive attribute: The metaprogramming component is part of the program for which it generates a bit of code/program.

Why would metaprogramming be desirable? As with most other programming techniques, the goal is to achieve more functionality with less effort, where effort can be measured as code size, maintenance cost, and so forth. What characterizes metaprogramming is that some user-defined computation happens at translation time. The underlying motivation is often performance (things computed at translation time can frequently be optimized away) or interface simplicity (a metaprogram is generally shorter than what it expands to) or both.

Metaprogramming often relies on the concepts of traits and type functions as developed in Chapter 15. We therefore recommend getting familiar with that chapter prior to delving into this one.

17.1 A First Example of a Metaprogram

In 1994 during a meeting of the C++ standardization committee, Erwin Unruh discovered that templates can be used to compute something at compile time. He wrote a program that produced prime numbers. The intriguing part of this exercise, however, was that the production of the prime numbers was performed by the compiler during the compilation process and not at run time. Specifically, the compiler produced a sequence of error messages with all prime numbers from two up to a certain configurable value. Although this program wasn't strictly portable (error messages aren't standardized), the program did show that the template instantiation mechanism is a primitive recursive language that can perform nontrivial computations at compile time. This sort of compile-time computation that occurs through template instantiation is commonly called *template metaprogramming*.

As an introduction to the details of metaprogramming we start with a simple exercise (we will show Erwin's prime number program later on page 318). The following program shows how to compute at compile time the power of three for a given value:

```
// meta/pow3.hpp

#ifndef POW3_HPP
#define POW3_HPP

// primary template to compute 3 to the Nth
template<int N>
class Pow3 {
  public:
    enum { result = 3 * Pow3<N-1>::result };
};

// full specialization to end the recursion
template<>
class Pow3<0> {
  public:
    enum { result = 1 };
};

#endif // POW3_HPP
```

The driving force behind template metaprogramming is recursive template instantiation.[1] In our program to compute 3^N, recursive template instantiation is driven by the following two rules:

1. $3^N = 3 * 3^{N-1}$
2. $3^0 = 1$

The first template implements the general recursive rule:

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

When instantiated over a positive integer N, the template Pow3<> needs to compute the value for its enumeration value result. This value is simply twice the corresponding value in the same template instantiated over N-1.

The second template is a specialization that ends the recursion. It establishes the result of Pow3<0>:

[1] We saw an example of a recursive template in Section 12.4 on page 200. It could be considered a simple case of metaprogramming.

```
template<>
class Pow3<0> {
  public:
    enum { result = 1 };
};
```

Let's study the details of what happens when we use this template to compute 3^7 by instantiating `Pow3<7>`:

```
// meta/pow3.cpp

#include <iostream>
#include "pow3.hpp"

int main()
{
    std::cout << "Pow3<7>::result = " << Pow3<7>::result
              << '\n';
}
```

First, the compiler instantiates `Pow3<7>`. Its `result` is

```
3 * Pow3<6>::result
```

Thus, this requires the instantiation of the same template for `6`. Similarly, the result of `Pow3<6>` instantiates `Pow3<5>`, `Pow3<4>`, and so forth. The recursion stops when `Pow3<>` is instantiated over zero which yields one as its `result`.

The `Pow3<>` template (including its specialization) is called a *template metaprogram*. It describes a bit of computation that is evaluated at translation time as part of the template instantiation process. It is relatively simple and may not look very useful at first, but there are situations when such a tool comes in very handy.

17.2 Enumeration Values versus Static Constants

In old C++ compilers, enumeration values were the only available possibility to have "true constants" (so-called *constant-expressions*) inside class declarations. However, this has changed during the standardization of C++, which introduced the concept of in-class static constant initializers. A brief example illustrates the construct:

```
struct TrueConstants {
    enum { Three = 3 };
    static int const Four = 4;
};
```

In this example, `Four` is a "true constant"—just as is `Three`.

With this, our `Pow3` metaprogram may also look as follows:

```
// meta/pow3b.hpp

#ifndef POW3_HPP
#define POW3_HPP

// primary template to compute 3 to the Nth
template<int N>
class Pow3 {
  public:
    static int const result = 3 * Pow3<N-1>::result;
};

// full specialization to end the recursion
template<>
class Pow3<0> {
  public:
    static int const result = 1;
};

#endif // POW3_HPP
```

The only difference is the use of static constant members instead of enumeration values. However, there is a drawback with this version: Static constant members are lvalues. So, if you have a declaration such as

```
void foo(int const&);
```

and you pass it the result of a metaprogram

```
foo(Pow3<7>::result);
```

a compiler must pass the address of `Pow3<7>::result`, which forces the compiler to instantiate and allocate the definition for the static member. As a result, the computation is no longer limited to a pure "compile-time" effect.

Enumeration values aren't lvalues (that is, they don't have an address). So, when you pass them "by reference," no static memory is used. It's almost exactly as if you passed the computed value as a literal. These considerations motivate us to use enumeration values in all metaprograms throughout this book.

17.3 A Second Example: Computing the Square Root

Lets look at a slightly more complicated example: a metaprogram that computes the square root of a given value N. The metaprogram looks as follows (explanation of the technique follows):

```
// meta/sqrt1.hpp

#ifndef SQRT_HPP
#define SQRT_HPP

// primary template to compute sqrt(N)
template <int N, int LO=1, int HI=N>
class Sqrt {
  public:
    // compute the midpoint, rounded up
    enum { mid = (LO+HI+1)/2 };

    // search a not too large value in a halved interval
    enum { result = (N<mid*mid) ? Sqrt<N,LO,mid-1>::result
                                : Sqrt<N,mid,HI>::result };
};

// partial specialization for the case when LO equals HI
template<int N, int M>
class Sqrt<N,M,M> {
  public:
    enum { result = M };
};

#endif // SQRT_HPP
```

The first template is the general recursive computation that is invoked with the template parameter N (the value for which to compute the square root) and two other optional parameters. The latter represent the minimum and maximum values the result can have. If the template is called with only one argument, we know that the square root is at least one and at most the value itself.

Our recursion then proceeds using a binary search technique (often called *method of bisection* in this context). Inside the template, we compute whether result is in the first or the second half of the range between LO and HI. This case differentiation is done using the conditional operator ?:. If $\texttt{mid}^2$ is greater than N, we continue the search in the first half. If $\texttt{mid}^2$ is less than or equal to N, we use the same template for the second half again.

The specialization that ends the recursive process is invoked when LO and HI have the same value M, which is our final result.

Again, let's look at the details of a simple program that uses this metaprogram:

```
// meta/sqrt1.cpp

#include <iostream>
#include "sqrt1.hpp"

int main()
{
    std::cout << "Sqrt<16>::result = " << Sqrt<16>::result
              << '\n';
    std::cout << "Sqrt<25>::result = " << Sqrt<25>::result
              << '\n';
    std::cout << "Sqrt<42>::result = " << Sqrt<42>::result
              << '\n';
    std::cout << "Sqrt<1>::result =  " << Sqrt<1>::result
              << '\n';
}
```

The expression

```
Sqrt<16>::result
```

is expanded to

```
Sqrt<16,1,16>::result
```

Inside the template, the metaprogram computes `Sqrt<16,1,16>::result` as follows:

```
mid = (1+16+1)/2
    = 9
result = (16<9*9) ? Sqrt<16,1,8>::result
                  : Sqrt<16,9,16>::result
       = (16<81) ? Sqrt<16,1,8>::result
                 : Sqrt<16,9,16>::result
       = Sqrt<16,1,8>::result
```

Thus, the result is computed as `Sqrt<16,1,8>::result`, which is expanded as follows:

```
mid = (1+8+1)/2
    = 5
result = (16<5*5) ? Sqrt<16,1,4>::result
                  : Sqrt<16,5,8>::result
       = (16<25) ? Sqrt<16,1,4>::result
                 : Sqrt<16,5,8>::result
       = Sqrt<16,1,4>::result
```

And similarly `Sqrt<16,1,4>::result` is decomposed as follows:

```
mid = (1+4+1)/2
    = 3
result = (16<3*3) ? Sqrt<16,1,2>::result
                  : Sqrt<16,3,4>::result
       = (16<9) ? Sqrt<16,1,2>::result
                : Sqrt<16,3,4>::result
       = Sqrt<16,3,4>::result
```

Finally, `Sqrt<16,3,4>::result` results in the following:

```
mid = (3+4+1)/2
    = 4
result = (16<4*4) ? Sqrt<16,3,3>::result
                  : Sqrt<16,4,4>::result
       = (16<16) ? Sqrt<16,3,3>::result
                 : Sqrt<16,4,4>::result
       = Sqrt<16,4,4>::result
```

and `Sqrt<16,4,4>::result` ends the recursive process because it matches the explicit specialization that catches equal high and low bounds. The final result is therefore as follows:

```
result = 4
```

Tracking All Instantiations

In the preceding example, we followed the significant instantiations that compute the square root of 16. However, when a compiler evaluates the expression

```
(16<=8*8) ? Sqrt<16,1,8>::result
          : Sqrt<16,9,16>::result
```

it not only instantiates the templates in the positive branch, but also those in the negative branch (`Sqrt<16,9,16>`). Furthermore, because the code attempts to access a member of the resulting class type using the `::` operator, all the members inside that class type are also instantiated. This means that the full instantiation of `Sqrt<16,9,16>` results in the full instantiation of `Sqrt<16,9,12>` and `Sqrt<16,13,16>`. When the whole process is examined in detail, we find that dozens of instantiations end up being generated. The total number is almost twice the value of `N`.

This is unfortunate because template instantiation is a fairly expensive process for most compilers, particularly with respect to memory consumption. Fortunately, there are techniques to reduce this explosion in the number of instantiations. We use specializations to select the result of computation instead of using the condition operator `?:`. To illustrate this, we rewrite our `Sqrt` metaprogram as follows:

```
// meta/sqrt2.hpp

#include "ifthenelse.hpp"

// primary template for main recursive step
template<int N, int LO=1, int HI=N>
class Sqrt {
  public:
    // compute the midpoint, rounded up
    enum { mid = (LO+HI+1)/2 };

    // search a not too large value in a halved interval
    typedef typename IfThenElse<(N<mid*mid),
                                Sqrt<N,LO,mid-1>,
                                Sqrt<N,mid,HI> >::ResultT
            SubT;
    enum { result = SubT::result };
};

// partial specialization for end of recursion criterion
template<int N, int S>
class Sqrt<N, S, S> {
  public:
    enum { result = S };
};
```

The key change here is the use of the `IfThenElse` template, which was introduced in Section 15.2.4 on page 272:

```
// meta/ifthenelse.hpp

#ifndef IFTHENELSE_HPP
#define IFTHENELSE_HPP

// primary template: yield second or third argument depending on first argument
template<bool C, typename Ta, typename Tb>
class IfThenElse;

// partial specialization: true yields second argument
template<typename Ta, typename Tb>
class IfThenElse<true, Ta, Tb> {
  public:
    typedef Ta ResultT;
};

// partial specialization: false yields third argument
template<typename Ta, typename Tb>
class IfThenElse<false, Ta, Tb> {
  public:
    typedef Tb ResultT;
};

#endif // IFTHENELSE_HPP
```

Remember, the `IfThenElse` template is a device that selects between two types based on a given Boolean constant. If the constant is true, the first type is `typedef`ed to `ResultT`; otherwise, `ResultT` stands for the second type. At this point it is important to remember that defining a typedef for a class template instance does not cause a C++ compiler to instantiate the body of that instance. Therefore, when we write

```
typedef typename IfThenElse<(N<mid*mid),
                            Sqrt<N,LO,mid-1>,
                            Sqrt<N,mid,HI> >::ResultT
        SubT;
```

neither `Sqrt<N,LO,mid-1>` nor `Sqrt<N,mid,HI>` is fully instantiated. Whichever of these two types ends up being a synonym for `SubT` is fully instantiated when looking up `SubT::result`. In contrast to our first approach, this strategy leads to a number of instantiations that is proportional to log_2(`N`): a very significant reduction in the cost of metaprogramming when `N` gets moderately large.

17.4 Using Induction Variables

You may argue that the way the metaprogram is written in the previous example looks rather complicated. And you may wonder whether you have learned something *you* can use whenever you have a problem to solve by a metaprogram. So, let's look for a more "naive" and maybe "more iterative" implementation of a metaprogram that computes the square root.

A "naive iterative algorithm" can be formulated as follows: To compute the square root of a given value `N`, we write a loop in which a variable `I` iterates from one to `N` until its square is equal to or greater than `N`. This value `I` is our square root of `N`. If we formulate this problem in ordinary C++, it looks as follows:

```
int I;
for (I=1; I*I<N; ++I) {
    ;
}
// I now contains the square root of N
```

However, as a metaprogram we have to formulate this loop in a recursive way, and we need an end criterion to end the recursion. As a result, an implementation of this loop as a metaprogram looks as follows:

```
// meta/sqrt3.hpp

#ifndef SQRT_HPP
#define SQRT_HPP

// primary template to compute sqrt(N) via iteration
template <int N, int I=1>
class Sqrt {
  public:
    enum { result = (I*I<N) ? Sqrt<N,I+1>::result
                            : I };
};

// partial specialization to end the iteration
template<int N>
class Sqrt<N,N> {
  public:
    enum { result = N };
};

#endif // SQRT_HPP
```

We loop by "iterating" I over Sqrt<N,I>. As long as I*I<N yields true, we use the result of the next iteration Sqrt<N,I+1>::result as result. Otherwise I is our result.

For example, if we evaluate Sqrt<16> this gets expanded to Sqrt<16,1>. Thus, we start an iteration with one as a value of the so-called *induction variable* I. Now, as long as I^2 (that is I*I) is less than N, we use the next iteration value by computing Sqrt<N,I+1>::result. When I^2 is equal to or greater than N we know that I is the result.

You may wonder why we need a template specialization to end the recursion because the first template always, sooner or later, finds I as the result, which seems to end the recursion. Again, this is the effect of the instantiation of both branches of operator ?:, which was discussed in the previous section. Thus, the compiler computes the result of Sqrt<4> by instantiating as follows:

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

Although we find the result in step 2, the compiler instantiates until we find a step that ends the recursion with a specialization. Without the specialization, the compiler would continue to instantiate until internal compiler limits are reached.

Again, the application of the `IfThenElse` template solves the problem:

```
// meta/sqrt4.hpp

#ifndef SQRT_HPP
#define SQRT_HPP

#include "ifthenelse.hpp"

// template to yield template argument as result
template<int N>
class Value {
  public:
    enum { result = N };
};

// template to compute sqrt(N) via iteration
template <int N, int I=1>
class Sqrt {
  public:
        // instantiate next step or result type as branch
        typedef typename IfThenElse<(I*I<N),
                                    Sqrt<N,I+1>,
                                    Value<I>
                                   >::ResultT
                SubT;

        // use the result of branch type
        enum { result = SubT::result };
};

#endif // SQRT_HPP
```

Instead of the end criterion we use a `Value<>` template that returns the value of the template argument as `result`.

Again, using `IfThenElse<>` leads to a number of instantiations that is proportional to log_2(`N`) instead of `N`. This is a very significant reduction in the cost of metaprogramming. And for compilers with template instantiation limits, this means that you can evaluate the square root of much larger values. If your compiler supports up to 64 nested instantiations, for example, you can process the square root of up to 4096 (instead of up to 64).

The output of the “iterative” `Sqrt` templates is as follows:

```
Sqrt<16>::result = 4
Sqrt<25>::result = 5
Sqrt<42>::result = 7
Sqrt<1>::result =  1
```

Note that this implementation produces the integer square root rounded up for simplicity (the square root of 42 is produced as 7 instead of 6).

17.5 Computational Completeness

The `Pow3<>` and `Sqrt<>` examples show that a template metaprogram can contain:

- State variables: the template parameters
- Loop constructs: through recursion
- Path selection: by using conditional expressions or specializations
- Integer arithmetic

If there are no limits to the amount of recursive instantiations and the amount of state variables that are allowed, it can be shown that this is sufficient to compute anything that is computable. However, it may not be convenient to do so using templates. Furthermore, template instantiation typically requires substantial compiler resources, and extensive recursive instantiation quickly slows down a compiler or even exhausts the resources available. The C++ standard recommends but does not mandate that 17 levels of recursive instantiations be allowed as a minimum. Intensive template metaprogramming easily exhausts such a limit.

Hence, in practice, template metaprograms should be used sparingly. The are a few situations, however, when they are irreplaceable as a tool to implement convenient templates. In particular, they can sometimes be hidden in the innards of more conventional templates to squeeze more performance out of critical algorithm implementations.

17.6 Recursive Instantiation versus Recursive Template Arguments

Consider the following recursive template:

```
template<typename T, typename U>
struct Doublify {};

template<int N>
struct Trouble {
    typedef Doublify<typename Trouble<N-1>::LongType,
                     typename Trouble<N-1>::LongType> LongType;
};

template<>
struct Trouble<0> {
    typedef double LongType;
};

Trouble<10>::LongType ouch;
```

The use of `Trouble<10>::LongType` not only triggers the recursive instantiation of `Trouble<9>`, `Trouble<8>`, ..., `Trouble<0>`, but it also instantiates `Doublify` over increasingly complex types. Indeed, Table 17.1 illustrates how quickly it grows.

As can be seen from Table 17.1, the complexity of the type description of the expression `Trouble<N>::LongType` grows exponentially with `N`. In general, such a situation stresses a C++ compiler even more than recursive instantiations that do not involve recursive template arguments. One of the problems here is that a compiler keeps a representation of the mangled name for the type. This mangled name encodes the exact template specialization in some way, and early C++ implementations used an encoding that is roughly proportional to the length of the template-id. These compilers then used well over 10,000 characters for `Trouble<10>::LongType`.

Newer C++ implementations take into account the fact that nested template-ids are fairly common in modern C++ programs and use clever compression techniques to reduce considerably the growth

Typedef Name	**Underlying Type**
`Trouble<0>::LongType`	`double`
`Trouble<1>::LongType`	`Doublify<double,double>`
`Trouble<2>::LongType`	`Doublify<Doublify<double,double>, Doublify<double,double> >`
`Trouble<3>::LongType`	`Doublify<Doublify<Doublify<double,double>, Doublify<double,double> >, <Doublify<double,double>, Doublify<double,double> > >`

Table 17.1. Growth of `Trouble<N>::LongType`

in name encoding (for example, a few hundred characters for `Trouble<10>::LongType`). Still, all other things being equal, it is probably preferable to organize recursive instantiation in such a way that template arguments need not also be nested recursively.

17.7 Using Metaprograms to Unroll Loops

One of the first practical applications of metaprogramming was the unrolling of loops for numeric computations, which is shown here as a complete example.

Numeric applications often have to process *n*-dimensional arrays or mathematical vectors. One typical operation is the computation of the so-called *dot product*. The dot product of two mathematical vectors `a` and `b` is the sum of all products of corresponding elements in both vectors. For example, if each vectors has three elements, the result is

```
a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
```

A mathematical library typically provides a function to compute such a dot product. Consider the following straightforward implementation:

```
// meta/loop1.hpp

#ifndef LOOP1_HPP
#define LOOP1_HPP

template <typename T>
inline T dot_product (int dim, T* a, T* b)
{
    T result = 0;
    for (int i=0; i<dim; ++i) {
        result += a[i]*b[i];
    }
    return result;
}

#endif // LOOP1_HPP
```

When we call this function as follows

```
// meta/loop1.cpp

#include <iostream>
#include "loop1.hpp"

int main()
{
    int a[3] = { 1, 2, 3};
    int b[3] = { 5, 6, 7};

    std::cout << "dot_product(3,a,b) = " << dot_product(3,a,b)
              << '\n';
    std::cout << "dot_product(3,a,a) = " << dot_product(3,a,a)
              << '\n';
}
```

we get the following result:

```
dot_product(3,a,b) = 38
dot_product(3,a,a) = 14
```

This is correct, but it takes too long for serious high-performance applications. Even declaring the function inline is often not sufficient to attain optimal performance.

The problem is that compilers usually optimize loops for many iterations, which is counterproductive in this case. Simply expanding the loop to

```
a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
```

would be a lot better.

Of course, this performance doesn't matter if we compute only some dot products from time to time. But, if we use this library component to perform millions of dot product computations, the differences become significant.

Of course, we could write the computation directly instead of calling `dot_product()`, or we could provide special functions for dot product computations with only a few dimensions, but this is tedious. Template metaprogramming solves this issue for us: We "program" to unroll the loops. Here is the metaprogram:

```
// meta/loop2.hpp

#ifndef LOOP2_HPP
#define LOOP2_HPP

// primary template
template <int DIM, typename T>
class DotProduct {
  public:
    static T result (T* a, T* b) {
        return *a * *b  +  DotProduct<DIM-1,T>::result(a+1,b+1);
    }
};

// partial specialization as end criteria
template <typename T>
class DotProduct<1,T> {
  public:
    static T result (T* a, T* b) {
        return *a * *b;
    }
};

// convenience function
template <int DIM, typename T>
inline T dot_product (T* a, T* b)
{
    return DotProduct<DIM,T>::result(a,b);
}

#endif // LOOP2_HPP
```

Now, by changing your application program only slightly, you can get the same result:

```
// meta/loop2.cpp

#include <iostream>
#include "loop2.hpp"
```

```
int main()
{
    int a[3] = { 1, 2, 3};
    int b[3] = { 5, 6, 7};

    std::cout << "dot_product<3>(a,b) = " << dot_product<3>(a,b)
              << '\n';
    std::cout << "dot_product<3>(a,a) = " << dot_product<3>(a,a)
              << '\n';
}
```

Instead of writing

```
dot_product(3,a,b)
```

we write

```
dot_product<3>(a,b)
```

This expression instantiates a convenience function template that translates the call into

```
DotProduct<3,int>::result(a,b)
```

And this is the start of the metaprogram.

Inside the metaprogram the `result` is the product of the first elements of `a` and `b` plus the `result` of the dot product of the remaining dimensions of the vectors starting with their next elements:

```
template <int DIM, typename T>
class DotProduct {
  public:
    static T result (T* a, T* b) {
        return *a * *b + DotProduct<DIM-1,T>::result(a+1,b+1);
    }
};
```

The end criterion is the case of a one-dimensional vector:

```
template <typename T>
class DotProduct<1,T> {
  public:
    static T result (T* a, T* b) {
        return *a * *b;
    }
};
```

Thus, for

```
dot_product<3>(a,b)
```

the instantiation process computes the following:

```
DotProduct<3,int>::result(a,b)
= *a * *b  + DotProduct<2,int>::result(a+1,b+1)
= *a * *b  + *(a+1) * *(b+1)  + DotProduct<1,int>::result(a+2,b+2)
= *a * *b  + *(a+1) * *(b+1)  + *(a+2) * *(b+2)
```

Note that this way of programming requires that the number of dimensions is known at compile time, which is often (but not always) the case.

Libraries, such as Blitz++ (see [*Blitz++*]), the MTL library (see [*MTL*]), and POOMA (see [*POOMA*]), use these kinds of metaprograms to provide fast routines for numeric linear algebra. Such metaprograms often do a better job than optimizers because they can integrate higher-level knowledge into the computations.[2] The industrial-strength implementation of such libraries involves many more details than the template-related issues we present here. Indeed, reckless unrolling does not always lead to optimal running times. However, these additional engineering considerations fall outside the scope of our text.

17.8 Afternotes

As mentioned earlier, the earliest documented example of a metaprogram was by Erwin Unruh, then representing Siemens on the C++ standardization committee. He noted the computational completeness of the template instantiation process and demonstrated his point by developing the first metaprogram. He used the Metaware compiler and coaxed it into issuing error messages that would contain successive prime numbers. Here is the code that was circulated at a C++ committee meeting in 1994 (modified so that it now compiles on standard conforming compilers)[3]:

```
// meta/unruh.cpp

// prime number computation by Erwin Unruh

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

[2] In some situations metaprograms significantly outperform their Fortran counterparts, even though Fortran optimizers are usually highly tuned for these sorts of applications.

[3] Thanks to Erwin Unruh for providing the code for this book. You can find the original example at [*Unruh-PrimeOrig*].

```
template<>
class is_prime<0,0> {
  public:
    enum {prim=1};
};

template<>
class is_prime<0,1> {
  public:
    enum {prim=1};
};

template <int i>
class D {
  public:
    D(void*);
};

template <int i>
class Prime_print {          // primary template for loop to print prime numbers
  public:
    Prime_print<i-1> a;
    enum { prim = is_prime<i,i-1>::prim
         };
    void f() {
        D<i> d = prim ? 1 : 0;
        a.f();
    }
};

template<>
class Prime_print<1> {       // full specialization to end the loop
  public:
    enum {prim=0};
    void f() {
        D<1> d = prim ? 1 : 0;
    };
};
```

```
#ifndef LAST
#define LAST 18
#endif

int main()
{
    Prime_print<LAST> a;
    a.f();
}
```

If you compile this program, the compiler will print error messages when in `Prime_print::f()` the initialization of `d` fails. This happens when the initial value is `1` because there is only a constructor for `void*`, and only `0` has a valid conversion to `void*`. For example, on one compiler we get (among other messages) the following errors:

```
unruh.cpp:36: conversion from ‘int’ to non-scalar type ‘D<17>’ requested
unruh.cpp:36: conversion from ‘int’ to non-scalar type ‘D<13>’ requested
unruh.cpp:36: conversion from ‘int’ to non-scalar type ‘D<11>’ requested
unruh.cpp:36: conversion from ‘int’ to non-scalar type ‘D<7>’ requested
unruh.cpp:36: conversion from ‘int’ to non-scalar type ‘D<5>’ requested
unruh.cpp:36: conversion from ‘int’ to non-scalar type ‘D<3>’ requested
unruh.cpp:36: conversion from ‘int’ to non-scalar type ‘D<2>’ requested
```

The concept of C++ template metaprogramming as a serious programming tool was first made popular (and somewhat formalized) by Todd Veldhuizen in his paper *Using C++ Template Metaprograms* (see [*VeldhuizenMeta95*]). Todd's work on Blitz++ (a numeric array library for C++, see [*Blitz++*]) also introduced many refinements and extensions to the metaprogramming (and to expression template techniques, introduced in the next chapter).