Chapter 8 Static Metaprogramming in C++[142]

8.1 What Is Static Metaprogramming?

In linguistics, a *metalanguage* is defined as follows [Dict]:

Metalanguage

> *"any language or symbolic system used to discuss, describe, or analyze another language or symbolic system"*

This definition characterizes the main idea of *metaprogramming*, which involves writing programs related by the meta-relationship, i.e. the relationship of "being about". A program that manipulates another program is clearly an instance of metaprogramming. In Section 7.4.7, we saw other instances of metaprogramming which do not necessarily involve program manipulation (e.g. before and after methods[143]); however, in this chapter we focus on *generative metaprograms*, i.e. programs manipulating and generating other programs.

Metaprogramming

Generative metaprograms

generative metaprogram = algorithm + program representation

Metaprograms can run in different contexts and at different times. In this chapter, however, we only focus on *static metaprograms*. These metaprograms run before the load time of the code they manipulate.[144]

Static metaprograms

The most common examples of systems involving metaprogramming are compilers and preprocessors. These systems manipulate representations of input programs (e.g. abstract syntax trees) in order to transform them into other languages (or the same language as the input language but with a modified program structure). In the usual case, the only people writing metaprograms for these systems are compiler developers who have the full access to their sources.

The idea of an *open compiler* is to make metaprogramming more accessible to a broader audience by providing well-defined, high-level interfaces for manipulating various internal program representations (see e.g. [LKRR92]). A transformation system is an example of an open compiler which provides an interface for writing transformations on abstract syntax trees (ASTs). We discussed them in Section 6.3. An open compiler may also provide access to the parser, runtime libraries, or any other of its parts. The Intentional Programming System is a good example of a widely-open programming environment. Unfortunately, currently no industrial-strength open compilers are commercially available (at least not for the main-stream object-oriented languages).[145]

Open compilers

8.2 Template Metaprogramming

A practicable approach to metaprogramming in C++ is *template metaprogramming*. Template metaprograms consist of class templates operating on numbers and/or types as data. Algorithms are expressed using template recursion as a looping construct and class template specialization as a conditional construct. Template recursion involves the direct or indirect use of a class template in the construction of its own member type or member constant.

Here is an example of a class template[146] which computes the factorial of a natural number:

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

//the following template specialization terminates the recursion
template<>
struct Factorial<0>
{
  enum { RET = 1 };
```

```
};
```

We can use this class template as follows:

```
void main()
{
  cout << Factorial<7>::RET << endl; //prints 5040
}
```

The important point about this program is that Factorial<7> is instantiated at compile time. During the instantiation, the compiler also determines the value of Factorial<7>::RET. Thus, the code generated for this main() program by the C++ compiler is the same as the code generated for the following main():

```
void main()
{
  cout << 5040 << endl; //prints 5040
}
```

We can regard Factorial<> as a *function* which is evaluated at compile time. This particular function takes one number as its parameter and returns another in its RET member (RET is an abbreviation for RETURN; we use this name to mimic the return statement in a programming language). It is important to note that we are dealing with a shift of intentionality here: The job of the compiler is to do type inference and type construction which involves computation. We use the fact that the compiler does computation and, by encoding data as types, we can actually use (or abuse) the compiler as a processor for *interpreting* metaprograms.

Metafunctions

Thus, we refer to functions such as Factorial<> as *metafunctions*. Factorial<> is a metafunction since, at compilation time, it computes constant data of a program which has not been generated yet.

A more obvious example of a metafunction would be a function which returns a type, especially a class type. Since a class type can represents computation (remember: it may contain methods), such a metafunction actually manipulates representations of computation.

The following metafunction takes a Boolean and two types as its parameters and returns a type:

```
template<bool cond, class ThenType, class ElseType>
struct IF
{
  typedef ThenType RET;
}

template<class ThenType, class ElseType>
struct IF<false, ThenType, ElseType>
{
  typedef ElseType RET;
};
```

As you have probably recognized, this function corresponds to an if statement: it has a condition parameter, a "then" parameter, and an "else" parameter. If the condition is true, it returns ThenType in RET. This is encoded in the base definition of the template. If the condition is false, it returns ElseType in RET. Thus, this metafunction can be viewed as a meta-control statement.

We can use IF<> to implement other useful metafunctions. For example, the following metafunction determines the larger of two numeric types:

```
#include <limits>
using namespace std;

template<class A, class B>
struct PROMOTE_NUMERIC_TYPE
{
   typedef IF<
      numeric_limits<A>::max_exponent10 < numeric_limits<B>::max_exponent10
```

```
    ||
    (numeric_limits<A>::max_exponent10==numeric_limits<B>::max_exponent10
       &&
     numeric_limits<A>::digits < numeric_limits<B>::digits),

    B,
    A>::RET RET;
};
```

This metafunction determines the larger numeric type as follows:

1. it returns the second type if its exponent is larger than the exponent of the first type or, in case the exponents are equal, it returns the second type if it has a larger number of significant digits;

2. otherwise, it returns the first type.

For example, the following expression evaluates to float:

```
PROMOTE_NUMERIC_TYPE<float, int>::RET
```

Metainformation

PROMOTE_NUMERIC_TYPE<> retrieves information about number types (i.e. *metainformation*) such as maximum exponent and number of significant digits from the numeric_limits<> templates provided by the standard C++ include file limits. Templates providing information about other types are referred to as *traits templates* [Mye95] (cf. traits classes in Section 6.4.2.4).

Traits templates

The standard traits template numeric_limits<> encodes many important properties of numeric types. The following is the base template (the base template is specialized for different numeric types):

```
template<class T>
   class numeric_limits
{
 public:
   static const bool has_denorm = false;
   static const bool has_denorm_loss = false;
   static const bool has_infinity = false;
   static const bool has_quiet_NaN = false;
   static const bool has_signaling_NaN = false;
   static const bool is_bounded = false;
   static const bool is_exact = false;
   static const bool is_iec559 = false;
   static const bool is_integer = false;
   static const bool is_modulo = false;
   static const bool is_signed = false;
   static const bool is_specialized = false;
   static const bool tinyness_before = false;
   static const bool traps = false;
   static const float_round_style round_style = round_toward_zero;
   static const int digits = 0;
   static const int digits10 = 0;
   static const int max_exponent = 0;
   static const int max_exponent10 = 0;
   static const int min_exponent = 0;
   static const int min_exponent10 = 0;
   static const int radix = 0;
   static T denorm_min() throw();
   static T epsilon() throw();
   static T infinity() throw();
   static T max() throw();
   static T min() throw();
   static T quiet_NaN() throw();
   static T round_error() throw();
   static T signaling_NaN() throw();
```

```
  };
```

As stated, this base template is specialized for different numeric types. Each specialization provides concrete values for the members, e.g.:

```
template<>
   class numeric_limits <float>
{
 public:
    //...
    static const bool is_specialized = true; // yes, we have metainformation information about float
    //...
   static const int digits = 24;
   //...
   static const int max_exponent10 = 128;
   //...
};
```

Given the above definition, the following expression evaluates to 24:

```
numeric_limits<float>::digits
```

In general, we have three ways to provide information about a type:

- define a traits template for the type,
- provide the information directly as members of the type, or
- define a configuration repository (we discuss this approach in Section 8.7).

The first approach is the only possible one if the type is a basic type or if the type is user defined and you do not have access to its source or cannot modify the source for some other reason. Moreover, the traits solution is often the preferred one since it avoids putting too much information into the type.

There are also situations however, where you find it more convenient to put the information directly into the user-defined type. For example, later in Section 10.3.1.3, we will use templates to describe the structure of a matrix type. Each of these templates represents some matrix parameter or parameter value. In order to be able to manipulate matrix structure descriptions, we need some means for testing whether two types were instantiated from the same template or not. We can accomplish this by marking the templates with IDs. Let us take a look at an example. Assume that we want to be able to specify the parameter temperature of some metafunction in Fahrenheit and in Celsius (i.e. centigrade). We could do so by wrapping the specified number in the type indicating the unit of temperature used, e.g. fahrenheit<212> or celsius<100>. Since we want to be able to distinguish between Fahrenheit and Celsius, each of these two templates will have an ID. We provide the IDs through a base class:

```
struct base_class_for_temperature_values
{
  enum {
    // IDs of values
    celsius_id,
    fahrenheit_id };
};
```

And here are the two templates:

```
//celsius
template<int Value>
struct celsius : public base_class_for_temperature_values
{ enum {
    id= celsius_id,
```

```
    value= Value };
};

//fahrenheit
template<int Value>
struct fahrenheit: public base_class_for_temperature_values
{ enum {
    id= fahrenheit_id,
    value= Value };
};
```

Please note that each template "publishes" both its ID and its parameter value as enum constants.[147] Now, let us assume that some metafunction SomeMetafunction<> expects a temperature specification in Fahrenheit or in Celsius, but internally it does its computations in Celsius. Thus, we need a conversion metafunction, which takes a temperature specification in Fahrenheit or in Celsius and returns a temperature value in Celsius:

```
template<class temperature>
struct ConvertToCelsius
{ enum {
    RET=
      temperature::id==temperature::fahrenheit_id ?
        (temperature::value-32)*5/9                :
         temperature::value };
};
```

Here is an example demonstrating how this metafunction works:

```
cout << ConvertToCelsius<fahrenheit<212> >::RET << endl;     // prints "100"
cout << ConvertToCelsius<celsius<100> >::RET << endl;        // prints "100"
```

Now, you can use the conversion function in SomeFunction<> as follows:

```
template<class temperature>
class SomeFunction
{
  enum {
    celsius = ConvertToCelsius<temperature>::RET,
    // do some other computations...
    //... to compute result
    };
  public:
    enum {
      RET = result };
};
```

The latter metafunction illustrates a number of points:

- You do some computation in the private section and return the result in the public section.
- The enum constant initialization is used as a kind of "assignment statement".
- You can split computation over a number of "assignment statements" or move them into separate metafunctions.

There are two more observations we can make based on the examples discussed so far:

- Metafunctions can take numbers and/or types as their arguments and return numbers or types.
- The equivalent of an assignment statement for types is a member typedef declaration (see the code for PROMOTE_NUMERIC_TYPE<> above).

8.3 Metaprograms and Code Generation

The previous examples of metafunctions demonstrated how to perform computations at compile time. But we can also arrange metafunctions into metaprograms that generate code.

We start with a simple example. Assume that we have the following two types:

```
struct Type1
{
  static void print()
  {
    cout << "Type1" << endl;
  }
};

struct Type2
{
  static void print()
  {
    cout << "Type2" << endl;
  }
};
```

Each of them defines the static inline method print(). Now, consider the following statement:

```
IF< (1<2), Type1, Type2>::RET::print();
```

Since the condition 1<2 is true, Type1 ends up in RET. Thus, the above statement compiles into machine code which is equivalent to the machine code obtained by compiling the following statements:

```
cout << "Type1" << endl;
```

The reason is that print() is declared as a static inline method and the compiler can optimize away any overhead associated with the method call. This was just a very simple example, but in general you can imagine a metafunction that takes some parameters and does arbitrary complex computation in order to select some type providing the right method:

```
SomeMetafunction</* takes some parameters here*/>::RET::executeSomeCode();
```

The code in executeSomeCode() can also use different metafunctions in order to select other methods it calls. In effect, we are able to combine code fragments at compile time based on the algorithms embodied in the metafunctions.

As a further example of static code generation using metafunctions, we show you how to do loop unrolling in C++. In order to do this, we will use a meta while loop available at [TM]. In order to use the loop we need a statement and a condition. Both will be modeled as structs. The statement has to provide the method execute() and the next statement in next. The code in execute() is actually the code to be executed by the loop:

```
template <int i>
struct aStatement
{ enum { n = i };
  static void execute()
  { cout << i << endl;
  }
  typedef aStatement<n+1> next;
};
```

The condition provides the constant evaluate, whose value is used to determine when to terminate the loop.

```
struct aCondition
{ template <class Statement>
  struct Test
  { enum {  evaluate = (Statement::n <= 10) };
```

```
  };
};
```

Now, you can write the following code:

```
WHILE<Condition,aStatement<1> >::EXEC();
```

The compiler expands this code into:

```
cout << 1 << endl;
cout << 2 << endl;
cout << 3 << endl;
cout << 4 << endl;
cout << 5 << endl;
cout << 6 << endl;
cout << 7 << endl;
cout << 8 << endl;
cout << 9 << endl;
cout << 10 << endl;
```

You can use this technique to implement optimizations by unrolling small loops, e.g. loops for vector assignment.

Another approach to code generation is to compose class templates. For example, assume that we have the following two class templates:

- List<>, which implements the abstract data type list and
- ListWithLengthCounter<>, which represents a wrapper on List<> implementing a counter for keeping track of the length of the list.

Now, based on a flag, we can decide whether to wrap List<> into ListWithLengthCounter<> or not:

```
typedef IF<flag==listWithCounter,
           ListWithLengthCounter<List<ElementType> >
           List<ElementType> >::RET list;
```

We will use this technique later in Section 8.7 in order to implement a simple list generator.

8.4 List Processing Using Template Metaprograms

Nested templates can be used to represent lists. We can represent many useful things using a list "metadata structure", e.g. a meta-case statement (which is shown later) or other control structures (see e.g. Section 10.3.1.6). In other words, template metaprogramming uses types as compile-time data structures. Lists are useful for doing all sorts of computations in general. Indeed, lists are the only data structure available in Lisp and this does not impose any limitation on its expressiveness.

In Lisp, you can write a list like this:

```
(1 2 3 9)
```

This list of four numbers can be constructed by calling the list constructor cons four times:

```
(cons 1 (cons 2 (cons 3 (cons 9 nil))))
```

where nil is an empty list. In the prefix notation, this list would look like this:

```
cons(1, cons(2, cons(3, cons(9, nil))))
```

We can do this with nested templates, too. We can write something like this:

```
Cons<1, Cons<2, Cons<3, Cons<9, End> > > >
```

Here is the code for End and Cons<>:

```
//tag marking the end of a list
const int endValue = ~(~0u >> 1); //initialize with the smallest int
```

```
struct End
{
  enum    { head = endValue };
  typedef End Tail;
};

template<int head_, class Tail_ = End>
struct Cons
{
  enum    { head = head_ };
  typedef Tail_ Tail;
};
```

Please note that End and Cons<> publish head and Tail as their members.

Now, assume that we need a metafunction for determining the length of a list. Here is the code:

```
template<class List>
struct Length
{
  // make a recursive call to Length and pass Tail of the list as the argument
  enum { RET= Length<List::Tail>::RET+1 };
};

// stop the recursion if we've got to End
template<>
struct Length<End>
{
  enum { RET= 0 };
};
```

We can use this function as follows:

```
typedef Cons<1,Cons<2,Cons<3> > > list1;
cout << Length<list1>::RET << endl; // prints " 3"
```

The following function tests if a list is empty:

```
template<class List>
struct IsEmpty
{
  enum { RET= 0 };
};

template<>
struct IsEmpty<End>
{
  enum { RET= 1 };
};
```

Last<> returns the last element of a list:

```
template<class List>
struct Last
{
  enum {
    RET=IsEmpty<List::Tail>::RET ?
      List::head                 :
      Last<List::Tail>::RET * 1 //multiply by 1 to make VC++ 5.0 happy
  };
};

template<>
struct Last<End>
{
```

```
  enum { RET= endValue };
};
```

Append1<> takes the list List and the number n and returns a new list which is computed by appending n at the end of List:

```
template<class List, int n>
struct Append1
{
  typedef  Cons<List::head, Append1<List::Tail, n>::RET> RET;
};

template<int n>
struct Append1<End, n>
{
  typedef Cons<n> RET;
};
```

Please note that the recursion in Append1<> is terminated using partial specialization, where List is fixed to be End whereas n remains variable.

A general Append<> for appending two lists is similar:

```
template<class List1, class List2>
struct Append
{
  typedef Cons<List1::head, Append<List1::Tail, List2>::RET > RET;
};

template<class List2>
struct Append<End, List2>
{
  typedef List2 RET;
};
```

The following test program tests our list metafunctions:

```
void main()
{
  typedef Cons<1,Cons<2,Cons<3> > > list1;

  cout << Length<list1>::RET << endl;     // prints 3
  cout << Last<list1>::RET << endl;       // prints 3

  typedef Append1<list1, 9>::RET list2;

  cout << Length<list2>::RET << endl;     // prints 4
  cout << Last<list2>::RET << endl;       // prints 9

  typedef Append<list1, list2>::RET list3;

  cout << Length<list3>::RET << endl;     // prints 7
  cout << Last<list3>::RET << endl;       // prints 9

}
```

You can easily imagine how to write other functions such as Reverse(List), CopyUpTo(X,List), RestPast(X,List), Replace(X,Y,List), etc. We can make our list functions more general by declaring the first parameter of Cons<> to be a type rather than an int and providing a type wrapper for numbers:

```
template<class head_, class Tail_ = End>
struct Cons
{
  typedef head_ head;
  typedef Tail_ Tail;
};
```

```
template <int n>
struct Int
{
  enum { value=n };
};
```

Given the above templates, we can build lists consisting of types and numbers, e.g.

```
Cons<Int<1>, Cons<SomeType, Cons<Int<3> > > >
```

Indeed, this technique allows us to implement a simple Lisp as a template metaprogram (see Section 8.12).

8.5 Workarounds for Doing Without Partial Specialization of Class Templates

As you saw in the previous section, partial specialization allows you to terminate the recursion of a template whose some parameters remain fixed during all iterations and other parameters vary from iteration to iteration. Unfortunately, currently only very few compilers support partial specialization. But here is the good news: We have always been able to find a workaround. This section gives you the necessary techniques for doing without partial specialization.

We will consider two cases: a recursive metafunction with numeric arguments and a recursive metafunction with type parameters.

We start with a recursive metafunction with numeric arguments. A classic example of a numeric function requiring partial specialization is raising a number to the m-th power. Here is the implementation of the power metafunction with partial specialization:

```
//power
template<int n, int m>
struct Power
{
  enum { RET = m>0 ? Power< n, (m>0) ? m-1:0 >::RET * n
                   : 1 };
};

// terminate recursion
template<int n>
struct Power<n, 0>
{
  enum { RET = 1 };
};
```

This function works as follows: It takes `n` and `m` as arguments and recursively computes the result by calling `Power<n, m-1>`. Thus, the first argument remains constant, while the second argument is "consumed". We know that the final call looks like this: `Power<n, 0>`. This is the classic case for partial specialization: One argument could be anything and the other one is fixed.

If our C++ compiler does not support partial specialization, we need another solution. The required technique is to map the final call to one that has all of its arguments fixed, e.g. `Power<1, 0>`. This can be easily done by using an extra conditional operator in the first argument to the call to `Power<n, m-1>`. Here is the code:

```
//power
template<int n, int m>
struct Power
{
  enum { RET = m>0 ? Power<(m>0) ? n:1, (m>0) ? m-1:0>::RET * n
                   : 1 };
};

template<>
```

```
struct Power<1,0>
{
  enum { RET = 1 };
};

cout << "Power<2,3>::RET = " << Power<2,3>::RET << endl; // prints " 8"
```

We can use the same trick in order to reimplement the metafunction Append1<> (see Section 8.4) without partial specialization. All we have to do is to map the recursion termination to a case where both arguments are fixed:

```
template<class List, int n>
struct Append1
{
  enum {
    new_head = IsEmpty<List>::RET  ?
      endValue                     :
      n
  };

  typedef
    IF<IsEmpty<List>::RET,
      Cons<n>,
      Cons<List::head, Append1<List::Tail, new_head>::RET >
    >::RET RET;
};

template<>
struct Append1<End, endValue>
{
  typedef End RET;
};
```

The same technique also works with type arguments. Here is the metafunction Append<> without partial specialization (remember that the second argument was constant during recursion):

```
template<class List1, class List2>
struct Append
{
  typedef
    IF<IsEmpty<List1>::RET,
      End,
      List2
    >::RET NewList2;

  typedef
    IF<IsEmpty<List1>::RET,
      List2,
      Cons<List1::head, Append<List1::Tail, NewList2>::RET >
    >::RET RET;
};

template<>
struct Append<End, End>
{
  typedef End RET;
};
```

You may ask why we worry about terminating recursion in the else branch of the second IF<> if in the terminal case we return List2 and not the value of the recursive call. The reason why we have to worry is that the types in both branches of a meta-if (i.e. IF<>) are actually built by the compiler. Thus, meta-if has a slightly different behavior than an if statement in a conventional programming language: we have always to keep in mind that both branches of a meta-if are "evaluated" (at least on VC++ 5.0).

We have used meta-if in order to eliminate partial specialization in Append<>. But if you take a look at the implementation of meta-if in Section 8.2, you will realize that meta-if uses partial specialization itself. Thus, we also need to provide an implementation of meta-if without partial specialization.

A meta-if involves returning the first of its two argument types if the condition is true and the second one if the condition is false. Let us first provide two metafunctions Select, where each of them takes two parameters and the first one returns the first parameter and the second one returns the second parameter. Additionally, we wrap the metafunctions into classes, so that we can return them as a result of some other metafunction. Here is the code for the two metafunctions:

```
struct SelectFirstType
{
  template<class A, class B>
  struct Select
  {
    typedef A RET;
  };
};

struct SelectSecondType
{
  template<class A, class B>
  struct Select
  {
    typedef B RET;
  };
};
```

Now, we implement another metafunction which takes a condition number as its parameter and returns SelectFirstType if the condition is other than 0 and SelectSecondType if the condition is 0:

```
template<int condition>
struct SelectSelector
{
  typedef SelectFirstType RET;
};

template<>
struct SelectSelector<0>
{
  typedef SelectSecondType RET;
};
```

Finally, we can implement our meta-if:

```
template<int condition, class A, class B>
class IF
{
    typedef SelectSelector<condition>::RET Selector;

  public:
    typedef Selector::Select<A, B>::RET RET;
};
```

Higher-order metafunctions

Please note that this implementation of meta-if does not use partial specialization. Another interesting thing to point out is that we returned metafunctions (i.e. SelectFirstType and SelectSecondType) as a result of some other metafunction (we could use the same technique to pass metafunctions as arguments to some other metafunctions). Thus, template metaprogramming supports *higher-order metafunctions*.

8.6 Control Structures

The structure of metaprograms can be improved by using basic metafunction which provide the functionality equivalent to the control structures used in programming languages such as if, switch, while, for, etc.

You have already seen the meta-if in Section 8.2. A *meta-switch* is used as follows:

```
SWITCH < 5
  , CASE < 1, TypeA
  , CASE < 2, TypeB
  , DEFAULT < TypeC > > >
  >::RET                          //returns TypeC
```

As you probably recognized, this switch uses the list processing techniques we discussed in Section 8.4. The implementation is given in Section 8.13.

You also saw the *meta-while loop* in Section 8.3. Other loops such as *meta-for* and *meta-do* also exist (see [TM]).

8.7 Configuration Repositories and Code Generation

In Section 6.4.2.4, we showed how to use a configuration repository class in order to encapsulate the horizontal parameters of a configuration and to propagate types up and down aggregation and inheritance hierarchies. Such configuration repositories can be combined with template metaprogramming into a very powerful static generation technique.

We will present this generation technique step by step, starting with a simple example of configurable list components based on a configuration repository.

We start with the implementation of the base component representing a list. The component implements methods such as head() for reading the head of the list, setHead() for setting the head, and setTail() for setting the tail of the list. The component takes a configuration repository class as its parameter and retrieves a number of types from it:

```
template<class Config_>
class List
{
  public:
    // publish Config so that others can access it
    typedef Config_ Config;

  private:
    // retrieve the element type
    typedef Config::ElementType ElementType;

    // retrieve my type (i.e. the final type of the list); this is necessary since
    // we will derive from List<> later and
    // we want to use the most derived type as the type of tail_;
    // thus, we actually pass a type from the subclass
    // to its superclass
    typedef Config::ReturnType ReturnType;

  public:
    List(const ElementType& h, ReturnType *t = 0) :
      head_(h), tail_(t)
    {}

    void setHead(const ElementType& h)
    { head_ = h; }

    const ElementType& head() const
```

```
    { return head_; }

    void setTail(ReturnType *t)
    { tail_ = t; }

    ReturnType *tail() const
    { return tail_; }

  private:
    ElementType head_;
    ReturnType *tail_;
};
```

Next, imagine that you need to keep track of the number of elements in the list. We can accomplish this using an inheritance-based wrapper (see Section 6.4.2.4):

```
template<class BaseList>
class ListWithLengthCounter : public BaseList
{
  public:
    typedef BaseList::Config Config;

  private:
    typedef Config::ElementType ElementType;
    typedef Config::ReturnType ReturnType;

    //get the type for the length counter
    typedef Config::LengthType LengthType;

  public:
    ListWithLengthCounter(const ElementType& h, ReturnType *t = 0) :
    BaseList(h,t), length_(computedLength())
    { }

    //redefine setTail() to keep track of the length
    void setTail(ReturnType *t)
    {
      BaseList::setTail(t);
      length_ = computedLength();
    }

    //an here is the length() method
    const LengthType& length() const
    { return length_; }

  private:
    LengthType computedLength()
    {
      return tail() ? tail()->length()+1
                    : 1; }

    LengthType length_;
};
```

Furthermore, we might be also interested in logging all calls to `head()`, `setHead()`, and `setTail()`. For this purpose, we provide yet another wrapper, which is implemented similarly to `ListWithLengthCounter<>`:

```
template<class BaseList>
class TracedList : public BaseList
{
  public:
    typedef BaseList::Config Config;
```

```
  private:
    typedef Config::ElementType ElementType;
    typedef Config::ReturnType ReturnType;

  public:
    TracedList(const ElementType& h, ReturnType *t = 0) :
    BaseList(h,t)
    { }

    void setHead(const ElementType& h)
    {  cout << "setHead(" << h << ")"<< endl;
       BaseList::setHead(h);
    }
    const ElementType& head() const
    {  cout << "head()"<< endl;
       return BaseList::head();
    }

    void setTail(ReturnType *t)
    {  cout << "setTail(t)"<< endl;
       BaseList::setTail(t);
    }
};
```

Given these three components, we can construct four different configurations:

- SimpleList: a simple list;
- ListWithCounter: a list with a counter;
- TracedList: a list with tracing;
- TracedListWithCounter: a list with a counter and with tracing.

Here is the code:

```
//SimpleList
struct ListConfig
{
  typedef int ElementType;
  typedef List<ListConfig> ReturnType;
};
typedef ListConfig::ReturnType SimpleList;

// ListWithCounter
struct CounterListConfig
{
  typedef int ElementType;
  typedef int LengthType;
  typedef ListWithLengthCounter<List<CounterListConfig> > ReturnType;
};
typedef CounterListConfig::ReturnType ListWithCounter;

// TracedList
struct TracedListConfig
{
  typedef int ElementType;
  typedef TracedList<List<TracedListConfig> > ReturnType;
};
typedef TracedListConfig::ReturnType TracedList;

// TracedListWithCounter
struct TracedCounterListConfig
{
```

```
  typedef int ElementType;
  typedef int LengthType;
  typedef TracedList<ListWithLengthCounter<List<TracedCounterListConfig> > > ReturnType;
};
typedef TracedCounterListConfig::ReturnType TracedListWithCounter;
```

The problem with this solution is that we have to provide an explicit configuration repository for each of these four combinations. This is not a big problem in our small example. However, if we have thousands of different combinations (e.g. the matrix components described in Chapter 10 can be configured in over 140 000 different ways), writing all these configuration repository classes becomes unpractical (remember that for larger configurations each of the configuration repository classes is much longer than in this example).

A better solution is to generate the required configuration repository using template metaprogramming. For this purpose, we define the metafunction LIST_GENERATOR<> which takes a number of parameters describing the list type we want to generate and returns the generated list type.

Here is the declaration of this metafunction:

```
template<class ElementType, int counterFlag, int tracingFlag,  class LengthType>
class LIST_GENERATOR;
```

The semantics of the parameters is as follows:

- ElementType: This is the element type of the list.
- counterFlag: This flag specifies whether the generated list should have a counter or not. Possible values are with_counter or no_counter.
- tracingFlag: This flag specifies whether the generated list should have a counter or not. Possible values are with_tracing or no_tracing.
- LengthType: This is the type of the length counter (if any).

We first need to provide the values for the flags:

```
enum {with_counter, no_counter};
enum {with_tracing, no_tracing};
```

And here is the code of the list generator metafunction:

```
template<
  class ElementType_ = int,
  int counterFlag = no_counter,
  int tracingFlag = no_tracing,
  class LengthType_ = int
>
class LIST_GENERATOR
{
  public:
    // provide the type of the generator as Generator
    typedef LIST_GENERATOR<ElementType_, counterFlag, tracingFlag, LengthType_> Generator;

  private:
    //define a simple list; please note that we pass Generator as parameter
    typedef List<Generator> list;

    // wrap into ListWithLengthCounter if specified
    typedef
```

```
      IF<counterFlag==with_counter,
        ListWithLengthCounter<list>,
        list
      >::RET list_with_counter_or_not;

    // wrap into TracedList if specified
    typedef
      IF<tracingFlag==with_tracing,
        TracedList<list_with_counter_or_not>,
        list_with_counter_or_not
      >::RET list_with_tracing_or_not;

  public:
    // return finished type
    typedef list_with_tracing_or_not RET;

    // provide Config; Config is retrieved by List<> from Generator and passed on to its wrappers
    struct Config
    {
      typedef ElementType_ ElementType;
      typedef LengthType_ LengthType;
      typedef RET ReturnType;
    };
};
```

Please note that we passed Generator to List<> rather than Config. This is not a problem since List<> can retrieve Config from Generator. Thus, we need to change two lines in List<>:

```
template<class Generator>
class List
{
  public:
    //publish Config so that others can access it
    typedef Generator::Config Config;
    // the rest of the code remains the same as above
    //...
};
```

Using the list generator, we can produce each of the four list types as follows:

```
typedef LIST_GENERATOR<>::RET SimpleList;
typedef LIST_GENERATOR<int, with_counter>::RET ListWithCounter;
typedef LIST_GENERATOR<int, no_counter, with_tracing>::RET TracedList;
typedef LIST_GENERATOR<int, with_counter, with_tracing>::RET TracedListWithCounter;
```

The list generator is an example of a very simple generator. In Section 10.3.1.5, we show you an advanced matrix generator, which performs parsing, computing default values for parameters, and assembling components.

8.8 Template Metaprograms and Expression Templates

The main idea behind the technique of expression templates [Vel95b] is the following: if you implement binary operators to be used in an expression, e.g. A+B+C, where A, B, and C are vectors, each operator does not return the resulting vector (or whatever the result is), but an object representing the expression itself. Thus, the operators are actually type constructors and if you implement them as overloaded template operators, they will derive the exact type of the expression at compile time. This type encodes the structure of the expression and you can pass it to a template metafunction which analyzes it and generates efficient code for it using the techniques presented in Section 8.3.

The arguments to the expressions could be abstract data types generated using a generator such as the one described in the previous section (i.e. LIST_GENERATOR<>). In this case, the configuration

repositories contain all the metainformation about the ADTs they belong to. Please note that you can access the configuration repository at compile time, e.g.:

```
TracedListWithCounter::Config
```

Furthermore, you can provide traits about the types of the expression objects. These traits could define the algebraic properties of the involved operators. The ADT metainformation and the operator metainformation can then be used by the code generating metafunction to perform a complex analysis of the whole expression type and to generate highly-optimized code.

Section 10.3.1.7 will demonstrate some of these techniques.

8.9 Relationship Between Template Metaprogramming and Other Metaprogramming Approaches

As we stated at the beginning of this chapter, generative metaprograms manipulate program representations. Compilers and transformation systems usually manipulate abstract syntax trees (ASTs). Lisp allows us to write metaprograms which manipulate language constructs, more precisely, lists representing functions. Smalltalk and CLOS allow us to (dynamically) manipulate classes. Template metaprograms can manipulate (e.g. compose) program representations such as template classes and methods.

An important question is whether metaprograms have to be written in a different language than the programs being manipulated. This is often the case with transformation systems which usually provide a specialized transformation API which is independent of the structures being transformed. In the case of Lisp, CLOS, and Smalltalk, both the metaprograms and the programs being manipulated are part of the same language in a reflective way, i.e. base programs can call metaprograms and metaprograms can call base programs. In the case of template metaprogramming, we use a “metaprogramming sublanguage” of C++, which can manipulate types (e.g. compose class templates, select functions for inlining), but it has no access to their metainformation other than what was explicitly encoded using techniques such as traits. Also, template metaprograms run before the programs they generate and the base code cannot call the metacode.

In this context, we can view C++ as a two-level language. Two-level languages contain static code (which is evaluated at compile-time) and dynamic code (which is compiled, and later executed at run-time). Multi-level languages [GJ97] can provide a simpler approach to writing program generators (see e.g., the Catacomb system [SG97]).

8.10 Limitations of Template Metaprogramming

Template metaprogramming has a number of limitations which are discussed in Section 10.3.1.8.

Table 116 (in Section 10.3.2) compares the applicability of template metaprogramming and of the Intentional Programming System for implementing algorithmic libraries.

8.11 Historical Notes

The first documented template metaprogram was written by Unruh [Unr94]. The program generated prime numbers as compiler warnings. To our knowledge, the first publication on template metaprogramming was the article by Veldhuizen [Vel95b], who pioneered many of the template metaprogramming techniques by applying them to numerical computing. In particular, he’s been developing the numerical array package Blitz++ [Vel97], which we already mentioned in Section 7.6.4.

Currently, there is a number of other libraries using template metaprogramming, e.g. Pooma [POOMA], A++/P++ [BDQ97], MTL [SL98], and the generative matrix package described in Chapter 10.

The contribution of Ulrich Eisenecker and the author was to provide explicit meta-control structures. A further contribution is the combination of configuration repositories and metafunctions into generators. The concept of a meta-if in C++ was first published in [Cza97], but only after developing the workarounds for partial specialization (see Section 8.5), we were able to fully develop our ideas (since we did not have a compiler supporting partial specialization).

8.12 Appendix: A Very Simple Lisp Implementation Using Template Metaprogramming

```
#include <iostream.h>
#include "IF.h"

/*
The siplest Lisp implementation requires the following
primitive expressions:

 *data types: numbers, symbols, and lists
 *primitive functions: list constructor (cons),
  list accessors (first, rest), type-testing predicates
  (numberp, symbolp, listp), equivalence test (eq),
  algebraic operations on numbers (plus, minus, times,
  divide, rem), numeric comparison (eqp, lessp, greaterp)
*/

//**************************
// Symbolic data of Lisp:
//     *Symbolic atoms: numbers and symbols
//     *Symbolic expressions (S-expressions): lists
//

// In Lisp, we have numbers, symbols, and lists.
// We also need primitive methods for checking
// the type of a Lisp type.

//LispObject defines the interface
//for all lisp objects.
//Some of the members do not make
//sence for some subclasses.
//Yet they are required
//in order for numbers, symbols,
//and lists to be treated
//polymorphicaly by the compiler.

struct LispObject
{
  enum {
    N_VAL=0,
    ID=-1,
    IS_NUMBER=0,
    IS_SYMBOL=0,
    IS_LIST=0
  };
  typedef LispObject S_VAL;
  typedef LispObject FIRST;
  typedef LispObject REST;
};
```

```
struct NumberType : public LispObject
{
  //the following enums should accessed by the type testing predicates
  //numberp, symbolp, and listp, only.
  enum {
    IS_NUMBER=1,
    IS_SYMBOL=0,
    IS_LIST=0
  };
};

struct SymbolType : public LispObject
{
  enum {
    IS_NUMBER=0,
    IS_SYMBOL=1,
    IS_LIST=0
  };
};

struct ListType : public LispObject
{
  enum {
    IS_NUMBER=0,
    IS_SYMBOL=0,
    IS_LIST=1
  };
};

//numbers
//Numbers are defined as "wrapped" classes.
//We need a wrapper for numbers in
//order to be able to pass number to class templates
//expecting classes. The problem is that C++
//does not allow us provide two versions of a class template
//one with class and other with number parameters.
//Unfortunately, we can only handle integral types at the moment.
template <int n>
struct Int : NumberType
{
  enum { N_VAL=n };
};

//symbols
//symbols are defined as "wrapped" classes
//We need a wrapper for classes in
//order to be able provide them with IDs.
//IDs allow us to test for equivalence.
template <class Value, int SymbolID>
struct Symbol : public SymbolType
{
  typedef Value S_VAL;
  enum { ID=SymbolID };
};

//lists
//Lists are represented by nesting the list contructor
//cons defined later

//but first we need an empty list
struct EmptyList : public ListType {};


//*****************************************
```

```
//Primitive function
//*****************************************

//List constructor cons(X,L)
//
//an example of a list:
//Cons<Int<1>,Cons<Int<2>,Cons<Symbol<SomeClass,1>,EmptyList>>>
//In Lisp syntax, this list would look like this:
//cons(1,cons(2,cons("SomeClass",()))) = (1,2,"SomeClass")
//
//In this implementation of cons, we do not define RET
//since this would unnecessary complicate our C++ Lisp syntax.
//The list type is the type of an intantiated cons.

template <class X, class List>
struct Cons : public ListType
{
  typedef X FIRST;
  typedef List REST;
};


//list accessors first and rest

//first(List)
// returns the first element of List
//
//first is achieved through:
//Cons<Int<1>,Cons<Int<2>,Cons<Symbol<SomeClass,1>,
//  EmptyList>>>::FIRST = Int<1>

template <class List>
struct First
{
  typedef List::FIRST RET;
};

//rest(List)
// retuns the List without the first element

//rest is achieved through:
//Cons<Int<1>,Cons<Int<2>,Cons<Symbol<SomeClass,1>,
//  EmptyList>>>::REST
//   = Cons<Int<2>,Cons<Symbol<SomeClass,1>,EmptyList>>

template <class List>
struct Rest
{
  typedef List::REST RET;
};


//type testing predicates numberp, symbolp, listp

//numberp(X)
//this predicate returns true (i.e. 1) if X is a number
//otherwise false (i.e. 0)

template <class X>
struct Numberp
{
  enum { RET=X::IS_NUMBER};
};

//symbolp(X)
```

```
//this predicate returns true (i.e. 1) if X is a symbol
//otherwise false (i.e. 0)

template <class X>
struct Symbolp
{
  enum { RET=X::IS_SYMBOL};
};

//listp(X)
//this predicate returns true (i.e. 1) if X is a list
//otherwise false (i.e. 0)

template <class X>
struct Listp
{
  enum { RET=X::IS_LIST};
};

//empty(List)
//return true List=EmptyList
//otherwise false
template <class List>
struct IsEmpty
{
  enum { RET=0 };
};

struct IsEmpty<EmptyList>
{
  enum { RET=1 };
};


//equivalence test

//eq(X1,X2)
//return true if X1 is equivalent to X2
//otherwise false

template <class X1, class X2> struct Eq;
template <class N1, class N2> struct EqNumbers;
template <class S1, class S2> struct EqSymbols;
template <class L1, class L2> struct EqLists;

template <class X1, class X2>
struct Eq
{
  enum {
    RET=
      Numberp<X1>::RET && Numberp<X2>::RET ?
        EqNumbers<X1,X2>::RET :
        Symbolp<X1>::RET && Symbolp<X2>::RET ?
          EqSymbols<X1,X2>::RET :
          Listp<X1>::RET && Listp<X2>::RET ?
            EqLists<X1,X2>::RET :
            //should be 0, but VC++ 4.0 does not take it
            1 ?
              0:
              0
  };
};

struct Eq<EmptyList, EmptyList>
{
```

```
  enum { RET=1 };
};

struct Eq<LispObject,LispObject>
{
  enum { RET=1 };
};

template <class N1, class N2> //private
struct EqNumbers
{
  enum { RET=N1::N_VAL==N2::N_VAL };
};

template <class S1, class S2> //private
struct EqSymbols
{
  enum { RET=S1::ID==S2::ID };
};

template <class L1, class L2> //private
struct EqLists
{
  enum {
    RET=
      IsEmpty<L1>::RET ?
        0 :
        IsEmpty<L2>::RET ?
          0 :
          Eq<First<L1>::RET,First<L2>::RET>::RET ?
            EqLists<Rest<L1>::RET,Rest<L2>::RET>::RET :
            0
  };
};

struct EqLists<EmptyList, EmptyList>
{
  enum { RET=1 };
};
//to avoid a compile error
struct EqLists<LispObject, LispObject>
{
  enum { RET=1 };
};


//arithmetic functions

//plus
template <class N1, class N2>
struct Plus
{
  enum { RET=N1::VAL+N2::VAL };
};

//minus, times, and divide are defined similarly

//lessp, greaterp
//...


//let expression corresponds to typedef or enum

//define expression corresponds to a class template
//note: a template can be passed to a template by defining it
```

```
//as member of a concrete class (corresponds to passing functions)

//****************** end of primitives ***************

//**********************************************
//convenience functions

//length(List)
template <class List>
struct Length
{
  enum { RET=Length<Rest<List>::RET>::RET+1 };
};

struct Length<EmptyList>
{
  enum { RET=0 };
};

//second(List)
template <class List>
struct Second
{
  typedef First<Rest<List>::RET>::RET RET;
};

//third(List)
template <class List>
struct Third
{
  typedef Second<Rest<List>::RET>::RET RET;
};

//n-th(n,List)
struct nil {};

template <int n, class List>
struct N_th
{
  typedef
    IF<n==1,
      EmptyList,
      Rest<List>::RET
    >::RET tail;

  typedef
    IF<n==1,
      First<List>::RET,
      N_th<n-1, tail>::RET
    >::RET RET;
};

template<>
struct N_th < 0, EmptyList >
{
  typedef nil RET;
};

//last(List)
template <class List>
struct Last
{
  typedef N_th<Length<List>::RET,List>::RET RET;
};
```

```
//append1(List,LispObject)
template<class List, class LispObject>
struct Append1
{
  typedef
    IF<IsEmpty<List>::RET,
      nil,
      LispObject
    >::RET new_object;

  typedef
    IF<IsEmpty<List>::RET,
      nil,
      Rest<List>::RET
    >::RET new_list;

  typedef
    IF<IsEmpty<List>::RET,
      Cons<LispObject, EmptyList>,
      Cons<First<List>::RET, Append1<new_list, new_object>::RET >
    >::RET RET;
};

template<>
struct Append1<nil, nil>
{
  typedef EmptyList RET;
};

//here should come some other functions:
//append(L1, L2), reverse(List), copy_up_to(X,L), rest_past(X,L), replace(X,Y,L), etc.



// just for testing:
class SomeClass1 {};
class SomeClass2 {};

typedef Symbol<SomeClass1,1> symb1;
typedef Symbol<SomeClass2,2> symb2;

void main()
{
  cout << "typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l1;" << endl;
  typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l1;
  cout << "typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l2;" << endl;
  typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l2;
  cout << "typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,Cons<Int<5>,EmptyList> > > > l3;" << endl;
  typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,Cons<Int<5>,EmptyList> > > > l3;
  cout << "typedef Cons<symb1,Cons<Int<2>,Cons<symb2,EmptyList> > > l4;" << endl;
  typedef Cons<symb1,Cons<Int<2>,Cons<symb2,EmptyList> > > l4;

  cout << "EqNumbers<Int<1>,Int<1> >::RET =" << endl;
  cout << EqNumbers<Int<1>,Int<1> >::RET << endl;

  cout << "EqSymbols<symb1,symb2 >::RET =" << endl;
  cout << EqSymbols<symb1,symb2 >::RET << endl;

  cout << "Eq<Int<1>,Int<1> >::RET =" << endl;
  cout << Eq<Int<1>,Int<1> >::RET << endl;

  cout << "Eq<symb1,symb2>::RET =" << endl;
  cout << Eq<symb1,symb2>::RET << endl;
```

```
  cout << "Third<l3>::RET::N_VAL =" << endl;
  cout << Third<l3>::RET::N_VAL << endl;

  cout << "N_th<4, l3>::RET::N_VAL =" << endl;
  cout << N_th<4, l3>::RET::N_VAL << endl;

  cout << "Last<l2>::RET::N_VAL =" << endl;
  cout << Last<l2>::RET::N_VAL << endl;

  cout << "Eq<l1,l3>::RET =" << endl;
  cout << Eq<l1,l3>::RET << endl;

  cout << "Length<l1>::RET =" << endl;
  cout << Length<l1>::RET << endl;

  cout << "typedef Append1<l1, Int<9> >::RET l5;" << endl;
  typedef Append1<l1, Int<9> >::RET l5;

  cout << "N_th<4, l5>::RET::N_VAL =" << endl;
  cout << N_th<4, l5>::RET::N_VAL << endl;

  cout << "Length<l5>::RET =" << endl;
  cout << Length<l5>::RET << endl;
}
```

8.13 Appendix: Meta-Switch Statement

```
//*****************************************************************
//Authors: Ulrich Eisenecker and Johannes Knaupp
//*****************************************************************

#include "../if/IF.H"

const int NilValue        = ~(~0u >> 1); //initialize with the smallest int
const int DefaultValue  = NilValue+1;

struct NilCase
{
  enum   { tag                = NilValue
         , foundCount         = 0
         , defaultCount       = 0
       };
  typedef NilCase RET;
  typedef NilCase DEFAULT_RET;
};

class MultipleCaseHits {};    // for error detection
class MultipleDefaults {};    // for error detection

template< int Tag, class Statement, class Next = NilCase >
struct CASE
{
  enum   { tag = Tag };
  typedef Statement statement;
  typedef Next next;
};

template< class Statement, class Next = NilCase >
struct DEFAULT
{
  enum   { tag = DefaultValue };
  typedef Statement statement;
  typedef Next next;
};

template <int Tag, class aCase, bool acceptMultipleCaseHits = false >
```

```
struct SWITCH
{
  typedef aCase::next    NextCase;

  enum  { tag         = aCase::tag
        , nextTag     = NextCase::tag
        , found       = (tag == Tag)
        , isDefault   = (tag == DefaultValue)
        };

  typedef IF< (nextTag != NilValue)
            , SWITCH< Tag, NextCase >
            , NilCase
            >::RET NextSwitch;

  enum  { foundCount   = found     + NextSwitch::foundCount
        , defaultCount = isDefault + NextSwitch::defaultCount
        };

  typedef IF< isDefault
            , aCase::statement
            , NextSwitch::DEFAULT_RET
            >::RET DEFAULT_RET;

  typedef IF< found
            , IF< ((foundCount == 1) || (acceptMultipleCaseHits == true))
                , aCase::statement
                , MultipleCaseHits
                >::RET
            , IF< (foundCount != 0)
                , NextSwitch::RET
                , DEFAULT_RET
                >::RET
            >::RET ProvisionalRET;

  typedef IF< (defaultCount <= 1)
            , ProvisionalRET
            , MultipleDefaults
            >::RET RET;
};
```

8.14 References

[BDQ97] F. Bassetti, K. Davis, and D. Quinlan. A Comparison of Performance-enhancing Strategies for Parallel Numerical Object-Oriented Frameworks. In *Proceedings of the first International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, Marina del Rey, California, Dec 1997, http://www.c3.lanl.gov/~dquinlan/

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95), ACM SIGPLAN Notices*, vol. 30, no. 10, 1995, pp. 285-299, http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

[Cza97] K. Czarnecki. Statische Konfiguration in C++. In *OBJEKTspektrum* 4/1997, pp. 86-91, see http://nero.prakinf.tu-ilmenau.de/~czarn

[Dict] *Webster's Encyclopedic Unabridged Dictionary of the English Language*. Portland House, New York, 1989

[GJ97] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. In *Lisp and Symbolic Computation*, vol. 10, no. 2, 1997, pp. 113-158

[LKRR92] J. Lamping, G. Kiczales, L. H. Rodriguez Jr., and E. Ruf. An Architecture for an Open Compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992, see http://www.parc.xerox.com/spl/groups/eca/pubs/complete.html

[Mye95] N. C. Myers. Traits: a new and useful template technique. In *C++ Report*, June 1995, see http://www.cantrip.org/traits.html

[POOMA] POOMA: Parallel Object-Oriented Methods and Applications. A framework for scientific computing applications on parallel computers. Available at http://www.acl.lanl.gov/pooma

[SG97] J. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997

[SL98] J. G. Siek and A. Lumsdaine. A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98)*, 1998, see http://www.lsc.nd.edu/

[TM] Meta-control structures for template metaprogramming available at http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm

[Unr94] E. Unruh. Prime number computation. ANSI X3J16-94-0075/SO WG21-462

[Vel95a] T. Veldhuizen. Using C++ template metaprograms. In *C++ Report*, vol. 7, no. 4, May 1995, pp. 36-43, see http://monet.uwaterloo.ca/blitz/

[Vel95b] T. Veldhuizen. Expression Templates. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31, see http://monet.uwaterloo.ca/blitz/

[Vel97] T. Veldhuizen. Scientific Computing: C++ versus Fortran. In *Dr. Dobb's Journal*, November 1997, pp. 34-41, see http://monet.uwaterloo.ca/blitz/

Part IV

PUTTING IT ALL TOGETHER: DEMRAL AND THE MATRIX CASE STUDY