Metaprogramming in C++

Johannes Koskinen

March 9, 2004

Contents

1 Introduction **2**
1.1 Terminology 2

2 Metaprogramming in C++ **3**
2.1 Support for metaprogramming 3
2.1.1 Reflection 3
2.1.2 Two-Level Languages 3
2.2 Example 1, Factorial 3

3 Template Metaprogramming **5**
3.1 Template Metafunctions 6
3.2 Metainformation and Traits 6
3.3 Nested Templates 6
3.4 Code Generation 7
3.5 Composing Templates 7
3.6 Expression Templates 7
3.7 Recursive Code Expansion 7

4 Control Structures **9**
4.1 Conditions 9
4.2 Loops 9
4.2.1 While-Loop WHILE<> 11

5 Problems with Template Metaprogramming **13**

1 Introduction

This paper is mainly based on Czarnecki's book on generative programming [2].

There is an increasing demand for systems that can be easily configured for a specific deployment environment. In addition to configuration, the produced systems may need to be able to adjust themselves (e.g. modifying their caching policies depending on the current workload) at runtime. Such systems are called *adaptive*[1]. Adaptivity is not limited to runtime. We can also have libraries that automatically adapt (e.g. by selecting algorithms) the code they contribute to a system being compiled.

1.1 Terminology

- *Metaprogramming* is writing programs that represent and manipulate other programs (e.g. compilers, program generators, interpreters) or themselves (reflection).
- *Metaprograms* represent and manipulate other programs or themselves.
- *Reflection* is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*.
- *Introspection* is the ability of a program to observe and therefore reason about its own state.
- *Intercession* is the ability of a program to modify its own execution state or alter its own interpretation or meaning.
- *Metaobjects* represent methods, execution stacks, the processor, and nearly all elements of the language and its execution environment.
- *Metalevel architecture* is an architecture where a metalevel provides information about selected system properties and makes the software self-aware while a base level includes the application logic.
- *Partial evaluation* is a technique to optimize the system when running some code multiple times on sets of input data in which one part is varying and another part is constant. In that case, it is useful to pre-evaluate the code with respect to the constant part of the data.

[1] *Adaptable* systems can be adapted to a praticular deployment environment, whereas *adaptive* systems adapt themselves to a deployment environment.

2 Metaprogramming in C++

The need for representing metainformation at compile time became apparent during the development of the C++ Standard library. The problems were solved by the traits template idiom (see 3.2). The first article on template metaprogramming [7] was published in 1995. The IF<> template was the first control structure in a generic form [3] and the remaining control structures were published in [4].

2.1 Support for metaprogramming

The metaprogramming support in C++ is not a designed feature but more or less abuse of C++ compiler. The code of metaprograms is quite peculiar and obscure. There are also limitations for complexity of the metaprograms set by the compilers (see section 5).

2.1.1 Reflection

Different languages provide different levels of reflection. While Smalltalk allows you to directly modify classes or methods by modifying their metaobjects at runtime, Java provides a much lower lever of reflection, mainly suitable for *JavaBeans* component model and the *Remote Method Invocation* mechanism. C++ provides even less support for reflection than Java, the most important feature being *runtime type-identification (RTTI)*. There are also reflective extensions available to C++ (e.g. IBM System Object Model) [5].

2.1.2 Two-Level Languages

An important concept to static metaprogramming is a concept of *two-level languages*. Two-level languages contain static code, which is evaluated at compile time, and dynamic code, which is compiled and later executed at runtime. ISO/ANSI C++ contains a template mechanism for defining parameterized classes and functions. This includes type and integral template parameters and partial ad full template specialization.

Templates together with other C++ features constitute a Turing-complete[2], compile-time sublanguage of C++ (and so C++ is a two-level language). Because the sublanguage is Turing complete, there are no theoretical limits to what you can implement with it. In practice, there might be technical limitations, such as compiler limits.

The main compile-time conditional construct is template specialization: The compiler has to select a matching template out of several alternatives. The compile-time looping construct is template recursion (e.g. a member of class templates used in its own definition.). The compiler has to expand such patterns recursively.

2.2 Example 1, Factorial

One of the most common examples of using static code (i.e., code executed at compile time) is computing the factorial

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

of a natural number. A conventional C++ factorial function is shown in Figure 1.

[2] A *Turing-complete* language is a language with at least a conditional and a looping construct. All Turing-complete languages are equally powerful, that is, they are equivalent to a Turing machine.

```
int factorial(int n)
{
  return (n==0)? 1: n*factorial(n-1);
}
```

Figure 1: Recursive factorial function

The corresponding static code for computing the factorial at compile time is given in Figure 2. *RET* is used as an abbreviation for a return statement of a conventional function. When the compiler tries to instantiate the structure template Factorial<7> (where n=7). This involves initializing the enumerator *RET* with Factorial<6>::RET*7 and the compiler has to instantiate Factorial<6>::RET (and so on). The template specialization matches for n=0 (i.e. Factorial<0>).

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

// Note: template specialization
template<>
struct Factorial<0>
{ enum{RET=1};
};

// Usage:
cout << "Factorial(7)= " << Factoral<7>::RET << endl;
```

Figure 2: Static code for computing factorial

3 Template Metaprogramming

The ability to perform computations at compile time is not very exciting, but the possibility of using static code to manipulate dynamic code is more interesting. This is a form of static metaprogramming, whish is referred to as *template metaprogramming*.

Template metaprogramming can be divided into several categories (see Figure 3):

- Writing metafunctions for computing types and numbers
- Representing metainformation using member traits, traits classes, traits templates, and nested templates
- Using static control structures (see section 4.1).
- Using metafunctions to generate code
- Developing embedded domain-specific languages using expression templates

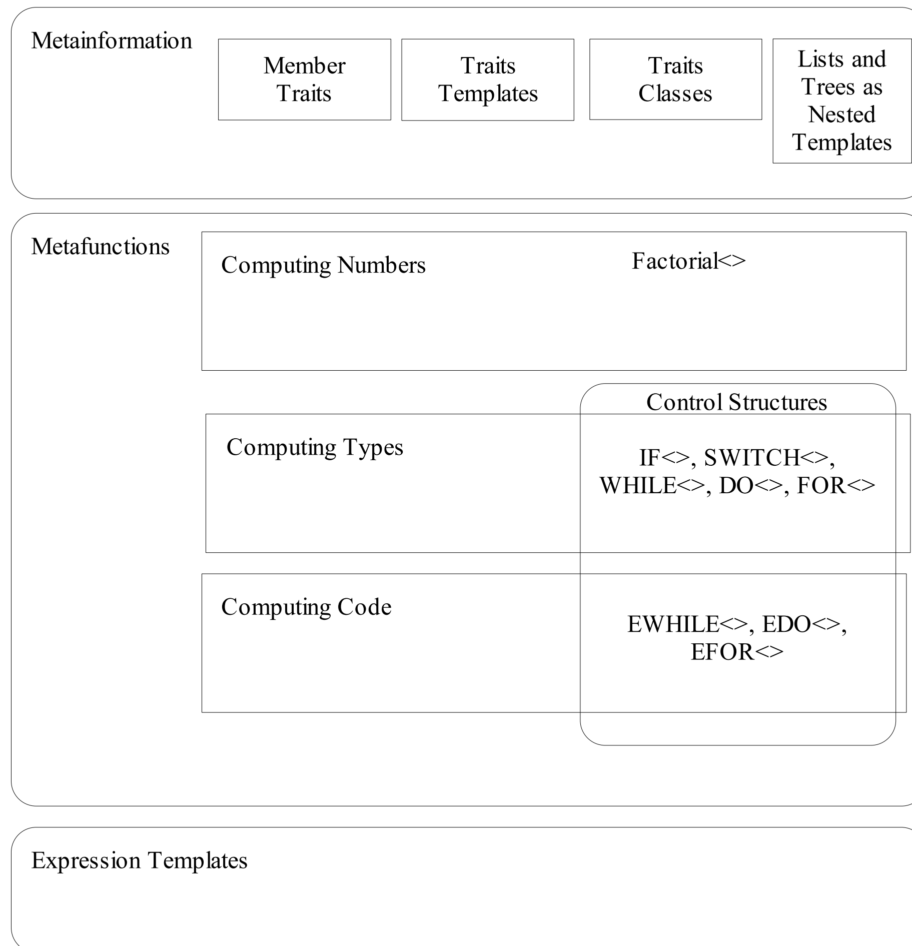

Figure 3: Map of template metaprogramming

3.1 Template Metafunctions

Template metafunctions are class templates (such as `Factorial<>` (section 2.2) or `IF<>` (section 4.1)) that operate on elements of dynamic code (i.e. they are at the metalevel of the dynamic code). The metafunctions can call other metafunctions (like functions can call other functions).

Numeric metafunctions can operate directly (in C++) only on integral numbers because floating-point numbers cannot be used as template parameters or compile-time constant initializers. However, the floating-point arithmetic can be implemented on top of integral arithmetic, if needed.

3.2 Metainformation and Traits

In template metaprograms, static code operates on dynamic code. In order to manipulate elements of the dynamic code, knowledge of some characteristics (e.g. value range covered by a numeric type and its precision) of the elements is needed. The characteristics are also referred to as *traits*. The traits can be encoded in static code as integral constats or types.

There are three ways to associate traits with the type they describe:

- *Member traits:* Define each trait as a member type or constant of the type it describes
- *Traits classes:* Encapsulate several traits in a seprate class
- *Traits templates:* Define a class template to hold the traits of a family types (e.g. `numeric_limits<>` in ANSI C++)

3.3 Nested Templates

In addition to basic data structuring mechanism, more complex compile-time data structures (such as lists or trees) are also needed. They can easily represented using nested templates. For example, the Lisp code `(cons 1 (cons 2 (cons 3 (cons 9 nil))))` can be simulated at the compile-time level in C++ using nested templates `Cons<1, Cons<2, Cons<3, Cons<9,End> > > >`.

3.4 Code Generation

Template metaprograms can be used to compose code fragments, to compose templates, and to unroll loops at compile time. This is useful for generating highly optimized and compact code for a given deployment context.

Template metaprogramming allows much more control in selecting code than preprocessor directives (*#ifdef*). The major difference is that static metaprogramming can interpret static C++ data embedded in the controlled C++ program and compute new static data, which can be used in the program.

For example, code fragments can be composed at compile time based on parameters given:

```
IF<(MAX_NO<AlgorithmVariantA::MAX_ALLOWED),
  AlgorithmVariantA,
  AlgorithmVariantB>::RET::execute();
```

3.5 Composing Templates

The composing templates can be used to generate concrete types or class hierarchies based on a number of abstract flags at compile time.

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

3.6 Expression Templates

Expression templates [6] is a programming technique that allows one to generate custom code for C++ expressions involving function and operator calls. With expression templates it is possible to implement

- Compile-time domain-specific checks on the structure of expressions, which the C++ type system cannot express otherwise (e.g. “an expression cannot contain more than five plus operators.”)
- Compile-time optimization transformations and custom code generation for expressions.

3.7 Recursive Code Expansion

Template programming can be used to expand code recursively (e.g. loop unrolling or to generate test code). For example, code for raising m to the power of n can be optimized, if n is known at compile time (see Figure 4). Even if C++ compilers will perform loop unrolling, template metaprogramming gives you control over which loop to unroll.

Normal static loops (see 4.2) also have code generation counterparts. These loops (with prefix **E**) can be used to call static metafunctions with different parameters. For example, calling PrintStat with parameters 1. . . 39 is done as follows:

```
EFOR<1,Less,40,+1,PrintStat>::exec();
```

```
template<int n>
inline int power(const int& m)
{ return power<n-1>(m)*m;}

template<>
inline int power<1>(const int& m)
{ return m;}

template<>
inline int power<0>(const int& m)
{ return 1;}
//
cout << power<3>(m)<<endl;
```

Figure 4: Raising a number to the power of n, where n is known at compile time

4 Control Structures

The control flow in template metaprograms is defined using recursion (recursive templates), selective matching (template specializations), and the conditional operator ?:. The C++ selection statements (*if* and *switch*) can be simulated using metafunctions.

4.1 Conditions

Factorial<> in 2.2 can be regarded as a function, which is evaluated at compile time and whose return value is in its RET member. Arguments and return values of such functions can also be types instead of integer numbers. The metafunction IF<> in Figure 5 takes a Boolean and two types and returns a type. If `condition` is `false`, it returns `Then` in RET, otherwise it returns `Else` in RET. Metafunction IF<> uses *partial template specialization* (only condition for `condition==false` is specialized). There is also version of IF<> with full template specialization for compilers that do not support partial template specialization. The code for that is left out in order to save space.

```
template<bool condition, class Then, class Else>
struct IF
{ typedef Then RET;
};
// NOTE: specialization
template<class Then, class Else>
struct IF<false,Then,Else>
{ typedef Else RET;
};

// Usage:
IF<(1+2>4), short, int>::RET i;
// the type of i is int
```

Figure 5: Implementation of IF<>

Moreover, a switch-statement (SWITCH<>) can be done using static part of C++. It takes an integral expression and a list of case statements, where each case is a pair of an integral tag an a type. The metafunction selects the first case matching the expression value (i.e. the first case whose tag is equal to the expression value or to some designated default tag), and returns the type contained in the case. The code is introduced in the book[2] and the usage of SWITCH<> is shown in Figure 6.

4.2 Loops

The basic looping mechanism in template metaprogramming is template recursion. However, there are cases in which a simple recursive definition of an algorithm is very inefficient. An example of a problem with inefficient but simple recursive formulation is computing *Fibonacci* numbers. Fibonacci numbers are defined as follows: $fib_{n+1} = fib_n + fib_{n-1}$, where $n > 0$ and $fib_0 = 0$ and $fib_1 = 1$. The problem with a

```
struct A {static void execute() {cout << "A" << endl;}};
struct B {static void execute() {cout << "B" << endl;}};
struct D {static void execute() {
                          cout << "Default" << endl;}};

// prints "Default"
SWITCH<(3+3),
    CASE<1,A,
    CASE<2,B,
    CASE<DEFAULT,D> > >
>::RET::execute();
```

Figure 6: Example of using SWITCH<>

recursive implementation is multiple computation of the same values (for example, $fib_3 = fib_2 + fib_1$,where $fib_2 = fib_1 + fib_0$). An iterative implementation has linear time complexity.

```
template<int n>
struct Fib
{ enum { RET=Fib<n-1>::RET + Fib<n-2>::RET };
};
template <>
struct Fib<0>
{ enum { RET=0 };
};

template<>
struct Fib<1>
{ enum {RET=1};
};

// Usage:
cout << Fib<8>::RET << endl;
```

Figure 7: Static implementation of the recursive version of Fib<>

The recursive definitions need not be inefficient when implemented statically. For example, the static translation of the recursive definition of Fibonacci numbers does not suffer from the same inefficiencies as its dynamic counterpart. The static code instantiates Fib<> only on the first use with a given argument, and subsequent uses with the same argument do not require new instantiations.

Static equivalents to the standard runtime looping statements (*while*, *do*, and *for*) can also be implemented. They allow easy translation from dynamic code with looping statements into static code.

4.2.1 While-Loop WHILE<>

A conventional while-loop takes a condition and executes a user-defined body in each iteration as long as the condition is true. A static loop will also take a condition as parameter and it takes a code (template metafunction) to execute in each iteration. Because static code does not allow side effects, each iteration will have to receive initial state and return new state explicitly. The state variables for WHILE<> can be stored to an *environment* type, which keeps the state of the static computation. The metafunction takes an environment as a parameter and returns a new one. The continuation condition can be done as a metafunction, which takes an environment and returns the information on whether the loop should keep going on.

While the use of WHILE<> is a quite complex because the requirement of three different templates, the situation should be improved by merging the environment into the statement. Also, with help of IF<> the implementation can be considerably simplified.

```
namespace intimate
{ template<class Statement>
  struct STOP
  { typedef Statement RET;
  };
};
template<class Condition, class Statement>
struct WHILE
{ typedef typename
       IF<Condition::template Code<Statement::RET,
          WHILE<Condition,typename Statement::Next>,
          intimate::Stop<Statement>
         >::RET::RET RET;
};
```

Figure 8: Static implementation of WHILE<>

```
template<int i_, int x_, int y>
struct FibStat
{ enum { i = i_,
         x = x_
       };
  typedef FibStat<i+1, x+y, x> Next;
};
//continue condition for Fib<>
template<int n>
struct FibCond
{ template<class Statement>
  struct Code
  { enum { RET = Statement::i < n };
  };
};
//Fib<>
template<int n>
struct Fib
{ enum { RET = WHILE<FibCond<n>,FibStat<1,1,0>
                    >::RET::x };
};
```

Figure 9: Iterative version of Fib<>

5 Problems with Template Metaprogramming

Unfortunately, the template metaprogramming has a number of problems.

- *Debugging:* Debugging template metaprograms is very difficult as there is no debugger for the C++ compilation process and the length of typenames in template metaprogramming can easily reach thousands of characters due to deep template nesting.
- *Error reporting:* There is no way for a template metaprogram to output a string during compilation.
- *Readability of the code:* The readability of template metacode is not very high. Template metaprogramming is not a result of careful language design but an accident.
- *Compilation times:* Template metaprograms may extend compilation times by orders of magnitude. Template metacode is interpreted rather than compiled.
- *Compiler limits:* Complex computations quickly lead to very complex types in template metaprogramming. The complexity and size limits of different compilers vary, but the limits can be quickly reached.
- *Limitations of expression templates:* There are no variables through which information can be passed between expressions at compile time in C++. Also, the lack of "typeof" features in C++ is a problem. The result type of the expression has to be manually computed.
- *Protability:* Template metaprogramming is based on many advanced C++ language features, which might not be supported by all compilers (even though they are in ISO C++ standard).

The complexity of template metaprograms is more or less limited by the compiler limits, compilation times, and debugging problems. *Intentional Programming*[1] does not suffer from these problems, but requires special programming environment instead of stardard C++ compiler.

References

[1] Aitken, W. et al., *Transformation in Intentional Programming*, Proc. 5th Int'l. Conf. on Software Reuse, IEEE Comput. Soc. Press, Los Alamitos, CA, 1998, pp. 114-123.

[2] Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[3] Czarnecki, K. *Statische Konfiguration in C++*. In OBJEKTspektrum 4/1997, pp. 86-91.

[4] Eisenecker, U. *Template-Metaprogrammierung: Kontrollstrukturen*. In OBJEKT-spektrum, No. 4, July/August 1999, pp. 81-86.

[5] Forman, I. R. and Danforth, S. H. *Putting metaclasses to Work: A New Dimension in Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1998.

[6] Veldhuizen, T. *Expression Templates*. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31.

[7] Veldhuizen, T. *Using C++ template metaprograms*. In C++ Report, vol. 7, no. 4, May 1995, pp. 36-43.