Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 301–308.

On the correctness of template metaprograms*

Ádám Sipos, István Zólyomi, Zoltán Porkoláb

Dept. of Programming Languages and Compilers,
Fac. of Informatics, Eötvös Loránd University, Budapest
e-mail:{shp,scamel,gsd}@elte.hu

Abstract

C++ template mateprogramming (TMP) is a recently emerged programming paradigm that assists the creation of efficient code and flexible libraries. On the other hand, TMP is not yet widely used, due to the lack of coding standards and methodologies applicable when writing metaprograms.

In this paper we present methods for writing efficient and reliable metaprograms. We define the correctness of metaprograms and the possible types of failures to meet the specification. We describe the methods for creating metaprograms meeting their specifications. One method is checking for expected properties of types and constants (*concept checking*). Another tool is *static assert*, a construct capable of halting the compilation of an erroneous program.

Keywords: C++, metaprogram, correctness

MSC: 68N19 Other programming techniques

1. Templates, metaprograms

1.1. Templates

Templates are an important part of the expressive power of the C++ language, by enabling data structures and algorithms be parametrized by types. The mechanism behind a list containing integer numbers, or strings is essentially the same, it is only the *type* of the contained objects that differs. With templates we can express this abstraction, thus this *generic* language construct aids code reuse, and the introduction of higher abstraction levels. Let us consider the following code:

```
template <class T>              int main()
class list                      {
```

*Supported by the Hungarian Ministry of Education under Grant FKFP0018/2002.

```
{                                       .
public:                                 .
    list();                             .
    void insert(const T& x);            list<int> li;  // instantiation
    T first();                          li.insert(1928);
    void sort();                    }
    .
    .
    .
};
```

This *list* template has one type parameter, called *T*, referring to the future type whose objects the list will contain. In order to use a list with some specific type, an *instantiation* is needed. This process can be invoked either implicitly by the compiler when the new list is needed, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments, and this new code is compiled.

The template mechanism of C++ is unique, as it enables the definition of *partial* and *full specializations*. Let us suppose that for some type (in our example `int`) we would like to create a more efficient type-specific implementation of the *list* template. We may define the following specialization:

```
template<>
class list<int>
{
public:
    list();
    void insert(const int& x);
    int first();
    void sort();
    .
    .
    .
    // a completely different implementation
    // may appear here
};
```

The specialization and the original template only share the *name*. A specialization does not need to provide the same functionality, interface, or implementation as the original.

Another unique property of C++ templates is that not only types but also integers, floating point numbers, pointers-to-functions and others may be template parameters.

1.2. Metaprograms

In case the compiler deduces that in a certain expression a concrete instance of a template is needed, an implicit instantiation is carried out. Let us consider the following codes demonstrating programs computing the factorial of some integer number by invoking a recursion:

```
// compile-time recursion
template <int N>
struct Factorial
{
    enum { value = N *
      Factorial <N-1>::value };
};
template<>
struct Factorial<1>
{
    enum { value = 1 };
};
int main()
{
    int r=Factorial<5>::value;
}
```

```
// runtime recursion
int Factorial(int N)
{
    if (N==1) return 1;
    return N*Factorial(N-1);
}

int main()
{
    int r=Factorial(5);
}
```

As the expression `Factorial<5>::value` must be evaluated in order to initialize `r` with a value, the `Factorial` template is instantiated with the argument `5`. Thus in the template the parameter `N` is substituted with `5` resulting in the expression `4 * Factorial<4>::value`. Note that `Factorial<5>`'s instantiation cannot be finished until `Factorial<4>` is instantiated, etc. This chain is called an *instantiation chain*. When `Factorial<1>::value` is accessed, instead of the original template, the full specialization is chosen by the compiler so the chain is stopped, and the instantiation of all types can be finished.

This is a *template metaprogram*, a program run in compile-time, calculating the factorial of 5. Since this operation happens in compile-time instead of runtime, this metaprogram may significantly slow the compilation process. On the other hand, this operation calculating a number's factorial results in a `O(n)` complexity in runtime. By replacing the calculation to compile-time, it will cause only a `O(1)` complexity in runtime. An important application of metaprograms is transferring calculations to compile-time, thus speeding up the execution of the program. Among other important applications of metaprograms are the implementation of *concept checking* [4] (testing for certain properties of types in compile-time), data structures containing types in compile-time (e.g. *typelist* [3]), *active libraries* [6], and others. By enabling the compile-time code adaptation, TMP is a style within the *generative programming* paradigm [5]. Template metaprogramming is *Turing-complete* [10], in theory its expressive power is equivalent to that of a Turing machine (and thus most programming languages).

As seen before in the `Factorial` example, a strong analogue exists between compile-time and runtime entities:

Metaprogram	Runtime program
(template) class	subprogram (function, procedure)
static const and enum class members	data (constant, literal)
symbolic names (typenames, typedefs)	variable
recursive templates, typelist	abstract data structures
static const initialization enum definition type inference	initialization *(but no assignment!)*

Table 1: Comparison of runtime and metaprograms

In the following with the help of this analogue we analyze the possible causes of errors in metaprograms.

2. Error categorization

Metaprograms take action in compile-time utilizing the language's type system. In case a metaprogram describes an erroneous construct, the whole compilation may fail. On the other hand the notion "erroneous" is ambiguous in the TMP world. In the following we discuss different scenarios involving metaprogram errors.

Let us suppose we would like to print the numbers from 0 to 4. Let us consider the following code:

```
#include <iostream>
int main ()
{
    for (i=0; i!=4; ++i)
        std::cout << i << std::endl;
}
```

This is an *ill-formed* program, with a *diagnostic message*. Since variable `i` is undefined, the compilation of this program will fail. Thus the program does not start to run. On the other hand, the algorithm itself is correct, and if the variable was defined, the program would implement the functionality we had intended.

In the following an *ill-formed* program, with *no diagnostic message* is presented:

```
#include <iostream>
int main ()
```

```
{
    for (int i=0; ; ++i)
        std::cout << i << std::endl;
}
```

Even though this code does compile, it implements an endless `for` loop. The cause of the error is the missing loop condition, and this is the bug we will need to find using some debugging method.

Now we define these error types in the metaprogramming realm. Let us suppose that the `Factorial` metaprogram described in Section 1.2 is implemented incorrectly, as `Factorial<1>` has a syntactic error, a semicolon is missing at the end of the definition.

```
template <int N>
class Factorial
{
public:
    enum { value = N*Factorial<N-1>::value };
};
template<>
class Factorial<1>
{
public:
    enum { value = 1 };
} // ; missing
```

This metaprogram is in many ways similar to our first program: this is an *ill-formed* template metaprogram, with *diagnostic message*. The metaprogram has not been run: no template instantiation happened.

Now let us suppose that we have forgotten to write the full specialization to the `Factorial` template.

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

// specialization for N==1 is missing

int main()
{
    const int r = Factorial<5>::value;
}
```

As the `Factorial` template has no explicit specialization, the `Factorial<N-1>` expression will trigger the instantiations of `Factorial<1>`, then `Factorial<0>`,

`Factorial<-1>` etc. We have written a compile-time infinite recursion. This is an *ill-formed* template metaprogram with *no diagnostic message*.

A third type of metaprogram errors is the *exhaustion of compiler resources*. This may happen when we invoke the –otherwise correctly implemented– `Factorial` template metaprogram with some large number: `Factorial<125>::value`. The C++ language standard guarantees only 17 implicit instantiations of the same template, going beyond this number will result in a compile-time error. However, many compilers can be parameterized to accept deeper instantiation levels. Thus the compiler will attempt to finish the compilation, and will obviously fail when using up all of its resources.

3. Writing correct metaprograms

We have seen that metaprograms are error-prone due to the complex syntax, and the new programming approach needed to create them. With the increasing number and complexity of metaprograms used in both the academic world and the IT industry, programmers need to rely on error preventing and debugging methods in order to create correct metaprograms.

Error prevention has both theoretical and practical methods. General theoretical methods for proving program correctness have been researched for decades. A common property of these *formal verification* methods is describing the abstract program in a formal language and creating a proof for the program's correctness [1, 8, 7].

In the following we describe two practical methods for preventing metaprogram errors.

3.1. Concept checking

There is no language-level mechanism in C++ to specify properties expected from template arguments. In case of complex template metaprograms the lack of sufficient support for checking argument properties may easily lead to errors. The research area *concept checking* aims to develop special language constructs that enable us to describe properties of template arguments, and other compile-time entities like constants, types.

The research results are implemented in libraries like boost [4], and Loki [3].

The language's designers have recognized the growing importance of template metaprograms and concept checking in general. Thus the next C++ standard (due in 2009-2010) will introduce the language construct *concept*.

3.2. Static assert

A runtime *assert* expects a logical expression, and if the expression's value is the program will be terminated with the error message *assertion failed*. Asserts are used to check invariants, pre-, and postconditions.

The assert's compile-time equivalent is a *static assert*. These are language constructs that when given a logical expression with the value false, will halt the compilation, thus preventing an erroneous program coming into being. Another important feature of a good static assert implementation is the capability of printing an appropriate error message informing the programmer about the error. In [3] the following solution is proposed, utilizing template partial specialization:

```
template <bool, class msg> struct STATIC_ASSERTION_FAILURE;
template <class msg> struct STATIC_ASSERTION_FAILURE<true,T>{};

template<int x> struct static_assert_test{};

#define STATIC_ASSERT( B , error)  \
   typedef static_assert_test< \
      sizeof(STATIC_ASSERTION_FAILURE< (bool)(B),error >)> \
   static_assert_typedef_;
```

In case the assertion fails, "STATIC_ASSERTION_FAILURE" and the name of the second macro parameter will both be included in the compiler's error message.

```
struct SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE {};

STATIC_ASSERT(sizeof(int)==sizeof(double),
  SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE)
```

This empty struct's name can be used for further information passing.

4. Conclusion

In this paper we have introduced template metaprogramming (TMP), a generative programming style used for compile-time code manipulation. We have presented the definition of the correctness of metaprograms, and the types of errors that might arise during the writing and executing of metaprograms. We have also presented two approaches for debugging metaprograms, one using a modification of the *g++* compiler, and another utilizing a standard C++ language construct.

References

[1] Abrial, J.-R., The B-Book: Assigning Programs to Meanings, *Cambridge University Press*, (1996).

[2] ANSI/ISO C++ Committee, Programming Languages – C++, ISO/IEC 14882:1998(E), American National Standards Institute, (1998).

[3] Alexandrescu, A., Modern C++ Design: Generic Programming and Design Patterns Applied, *Addison-Wesley* (2001).

[4] Boost Concept checking, `http://www.boost.org/libs/concept/_check.htm`

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

[6] Czarnecki, K., Eisenecker, U. W., Glück, R., Vandevoorde, D., Veldhuizen, T. L., Generative Programming and Active Libraries, *Springer-Verlag*, (2000).

[7] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, *Communications of the ACM*, 12:576.580, (1969).

[8] Valerie, N., Daniel, M., Viliam, S., Tree Automata in the Mathematical Theory, SAMI 2007 Proceedings, 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics, Poprad, ISBN 978-963-7154-56-0 (2007), 447–456.

[9] Sipos, Á., Effective Metaprogramming, M.Sc. Thesis, Budapest, (2006).

[10] Veldhuizen, T., C++ Templates are Turing Complete.

[11] Veldhuizen, T., Using C++ Template Metaprograms, *C++ Report* vol. 7 no. 4, (1995), 36–43.