Debugging C++ Template Metaprograms

Zoltán Porkoláb
Eötvös Loránd University,
Dept. of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
gsd@elte.hu

József Mihalicza
Eötvös Loránd University,
Dept. of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
pocok@inf.elte.hu

Ádám Sipos
Eötvös Loránd University,
Dept. of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
shp@elte.hu

Abstract

Template metaprogramming is an emerging new direction in C++ programming for executing algorithms in compilation time. Despite all of its already proven benefits and numerous successful applications, it is yet to be accepted in industrial projects. One reason is the lack of professional software tools supporting the development of template metaprograms. A strong analogue exists between traditional runtime programs and compile-time metaprograms. This connection presents the possibility for creating development tools similar to those already used when writing runtime programs. This paper introduces *Templight*, a debugging framework that reveals the steps executed by the compiler during the compilation of C++ programs with templates. Templight's features include following the instantiation chain, setting breakpoints, and inspecting metaprogram information. This framework aims to take a step forward to help template metaprogramming become more accepted in the software industry.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classification – C++; D.2.5 [*Testing and Debugging*]: Tracing

General Terms Languages

Keywords C++, template metaprogramming, debugging

1. Introduction

Programming is a human activity to understand a problem, make design decisions, and express our intentions for the computer. In most cases the last step is writing code in a certain programming language. The compiler then tries to interpret the source code through lexical, syntactic, and semantic analysis. In case the source code is syntactically correct, the compiler takes further steps to generate runnable code.

However, in numerous cases the code accepted by the compiler will not work as we had expected, and intended. The causes vary from simple typos – that (unfortunately) do not affect the syntax – to serious design problems. There are various methods to decrease the possibility of writing software diverging from its specification. Strongly-typed programming languages, good programming conventions, and many other techniques can help avoid such frustrating situations. Nevertheless in many cases we have *made* some error(s), and we have to fix it. For this we have to recognise that the bug exists, isolate its nature, and find the place of the error to apply the fix. This procedure is called debugging.

Debuggers are software tools to help the debugging process. The main objective of a debugger is to help us understand the hidden sequence of events that led to the error. In most cases this means following the program's control flow, retrieving information on memory locations, and showing the execution context. Debuggers also offer advanced functionality to improve efficiency of the debugging process. These include stopping the execution on a certain *breakpoint*, continuing the running step by step, step into, step out, or step over functions, etc. Still, debugging can be one of the most difficult and frustrating tasks for a programmer.

The more complex the language environment we work in, the more sophisticated the debugging toolset we need. In object-oriented languages like C++, a good debugger has to explore the attributes of objects, handle virtual function calls, explore overloading situations. Debuggers available for object-oriented languages successfully tackle these challenges.

Templates are also key language elements for the C++ programming language [16]. They are essential for capturing commonalities of abstractions without performance penalties in runtime [18]. *Generic programming* [14], a recently emerged programming paradigm for writing highly reusable components – usually libraries – is implemented in C++ with heavy use of templates [20]. The Standard Template Library – the most notable example – is now an unavoidable part of most professional C++ programs [8]. Another important application field of templates is the *C++ template metaprogramming*. Template metaprogramming is a technique in which template definitions are used to control the compilation process so that during the process of compiling a (meta)program is executed [21]. The output of this process is still checked by the compiler and run as an ordinary program.

Template metaprograming are proved to be a Turing-complete sublanguage of C++ [4]. We write metaprograms for various purposes. *Expression templates* [22] replace runtime computations with compile-time activities to enhance runtime performance. Static interface checking increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [10, 12]. Subtype relationships could be controlled with metaprograms too [5, 23].

While generic programming quickly became an essential part of C++ programs, template metaprogramming was able to break out from the academic world only recently. Despite the growing number of positive examples, developers are still wary of using template

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010...$5.00.

metaprogramming in strict, time-restricted software projects. One reason beside the necessity of highly trained C++ professionals is the lack of supporting software tools for template metaprogramming.

In this article we describe *Templight*, a debugging framework for C++ template metaprograms. The framework reads a C++ source and modifies it to emit well-formed messages during compilation. Information gained this way under the "running" of a metaprogram is used to reconstruct the compilation process. A front-end tool enables the user to set breakpoints, gives a step-by-step reconstruction of the instantiation flow of templates, and provides various other features.

In Section 2 we give an overview of C++ template metaprogramming compared to runtime programming. The ontology of metaprogram errors is presented in Section 3. In Section 4 we discuss the possibility of debugging template metaprograms. Our debugging framework is described in detail in Section 5 with demonstrative examples. Limitations and future directions are discussed in Section 6.

2. C++ Template metaprograms

It is hard to draw the boundary between an ordinary program using templates and a template metaprogram. We will use the notion *template metaprogram* for the collection of templates, their instantiations, and specialisations, whose purpose is to carry out operations in compile-time. Their expected behaviour might be emitting messages or generating special constructs for the runtime execution. Henceforth we will call a *runtime program* any kind of runnable code, including those which are the results of template metaprograms.

There is yet another kind of (pre)compile-time programming: preprocessor metaprogramming ([27]). Later we show that our debug method works properly together with preprocessor metaprogramming techniques.

Executing programs in either way means executing pre-defined actions on certain entities. It is useful to compare those actions and entities between runtime and metaprograms. The following table describes the parallels between metaprograms', and runtime programs' 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 initialisation enum definition type inference	initialisation *(but no assignment!)*

Table 1. Comparison of runtime and metaprograms

There is a clear relationship between a number of entities. C++ template metaprogram actions are defined in the form of template definitions and are "executed" when the compiler instantiates them. Templates can refer to other templates, therefore their instantiation can instruct the compiler to execute other instantiations. This way we get an instantiation chain very similar to a call stack of a runtime program. Recursive instantiations are not only possible but regular in template metaprograms to model loops.

```
template <int N>
class Factorial
{
public:
    enum { value = N*Factorial<N-1>::value };
};
template<>
class Factorial<1>
{
public:
    enum { value = 1 };
};
int main ()
{
    const int r = Factorial<5>::value;
}
```

Conditional statements (and stopping recursion) are solved via specialisations. Templates can be overloaded and the compiler has to choose the narrowest applicable template to instantiate. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters for other metaprograms [4].

```
// accumulate(n,f) := f(0) + f(1) + ... + f(n)
template <int n, template<int> class F>
struct Accumulate
{
    enum { RET=Accumulate<n-1,F>::RET+F<n>::RET };
};
struct Accumulate<0,F>
{
    enum { RET = F<0>::RET };
};
template <int n>
struct Square
{
    enum { RET = n*n };
};
int main()
{
    const int r = Accumulate<3,Square>::RET;
}
```

Data is expressed in runtime programs as constant values or literals. In metaprograms we use static const and enumeration values to store quantitative information. Results of computations under the execution of a metaprogram are stored either in new constants or enumerations. Furthermore, execution of metaprograms can cause new types be created. Types hold information that can influence the further run of the metaprogram.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favourite implementation form of expression templates [22]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [26].

However, there is a fundamental difference between runtime programs and C++ template metaprograms: once a certain entity (constant, enumeration value, type) has been defined, it will be immutable. There is no way to change its value or meaning. Therefore no such thing as a metaprogram assignment exists. In this sense metaprograms are similar to pure functional programming languages, where *referential transparency* is obtained. That is the reason why we use recursion and specialisation to implement loops:

we are not able to change the value of any loop variable. Immutability – as in functional languages – has a positive effect too: unwanted side effects do not occur.

3. Ontology of template metaprogram errors

The first examples of C++ template metaprograms were written by Erwin Unruh and circulated among members of the ANSI/ISO C++ standardisation committee [19]. These programs did not have to be run, their purpose was to generate warnings or error messages when being compiled. The most famous example computed the prime numbers, and produced the following output as error messages:

```
conversion from enum to D<2> requested in Prime
conversion from enum to D<3> requested in Prime
conversion from enum to D<5> requested in Prime
conversion from enum to D<7> requested in Prime
conversion from enum to D<11> requested in Prime
conversion from enum to D<13> requested in Prime
conversion from enum to D<17> requested in Prime
conversion from enum to D<19> requested in Prime
```

This is an erroneous C++ program in the sense of ”traditional” programming, since the compiler emits error messages, but as a template metaprogram it does exactly the right thing. These error messages were intentional and produced the expected output. In fact, the lack of error messages would denote an error in the prime generator template metaprogram. However, generating other messages than primes would be considered erroneous too.

This example points out the difference of the notions *correct* and *erroneous* behaviour between traditional runtime programs and template metaprograms. The C++ International Standard defines the notion *well-formed* and *ill-formed* programs [3]. A program is well-formed if it was constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule. It is ill-formed, if it is not well-formed. In case any ddiagnosable requirements are broken, the compiler ”shall issue a diagnostic message”. However, no such requirements for other forms of ill-formed programs exist. Moreover, ”whenever this International Standard places a requirement on the execution of a program ... and the data encountered during execution do not meet that requirement, the behavior of the program is *undefined* and ... places no requirements at all on the behavior of the program”. Even well-formed programs may be a cause of problems if they exceed resource limits.

The following code demonstrates an *ill-formed* program, with *diagnostic message*, since variable `i` is undefined. This leads to a compile-time error. The program does not start to run.

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

The next example compiles but fails to stop running.

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

This example demonstrates an *ill-formed* program without *diagnostic message*. It does compile, but 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.

Let us further examine the `Factorial` metaprogram described in Section 2, and let us suppose that the template specialisation `Factorial<1>` has a syntactic error: a semicolon is missing at the end of the class definition.

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

This is an *ill-formed* template metaprogram, with *diagnostic message*. The metaprogram has not been run: no template instantiation happened. Another *ill-formed* template metaprogram with *diagnostic message* is shown in the next example. However, it starts to ”run”, i.e. the compiler starts to instantiate the `Factorial` classes, but the metaprogram *aborts* (in compilation time) [15].

```
template <int N>
class Factorial
{
public:
    enum { value = N*Factorial<N-1>::value };
};
template<>
class Factorial<1>
{
// public: missing
    enum { value = 1 };
};
int main ()
{
    const int f = Fibonacci<4>::value;
    const int r = Factorial<5>::value;
}
```

As the full specialisation for `Factorial<1>` is written in form of a `class`, the default visibility rule for a class is `private`. Thus `enum { value=1 }` is a private member, so we receive a compile-time error when the compiler tries to acquire the value of `Factorial<1>::value`, while `Factorial<2>` is being instantiated. The template metaprogram is *ill-formed*, with no *diagnostic message*. However, if this code fragment is part of a bigger program which has a well-formed `Fibonacci` template, the `Fibonacci` part of the metaprogram would be successfully executed before abortion.

Let us now erase the full specialisation.

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

// specialisation for N==1 is missing

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

As the `Factorial` template has no explicit specialisation, the `Factorial<N-1>` expression will trigger the instantiations of `Factorial<1>` followed by `Factorial<0>`, `Factorial<-1>` etc. We have written a compile-time infinite recursion. This is an *ill-formed* template metaprogram with *no diagnostic message*.

In fact different compilers might react differently to this code. g++ 3.4 halts the compilation process after the 17 levels of implicit instantiations is reached, as defined by the C++ standard. The compiler for MSVC 6 runs until its resources are exhausted (reached `Factorial<-1308>` in our test). MSVC 7.1 halted the compilation reaching a certain recursion depth. The error message received was *fatal error C1202: recursive type or function dependency context too complex.*

Another possible source of error in template metaprogramming world is the exhaustion of the compiler's resources. Let us suppose that we want to calculate `Factorial<125>::value` somewhere in the program, and that we are using the original flawless `Factorial` metaprogram. In the case of a standard-compliant compiler it will still result in a compile-time error, as the compiler will exceed the permitted 17 levels of implicit instantiation. However, some compilers, like g++ can be parameterised to accept deeper instantiation levels. In this case the compiler continues the instantiation risking that the resources will be exhausted. In that unfortunate situation the compiler will crash.

4. Metaprogram debugging

There are certain techniques to reduce the possibility of writing malfunctioning template metaprograms.

4.1 Static assert

The tool described is a technique used in the case of runtime programs, called *assertation*. An `assert` is a function (usually implemented as a macro) that requires a logic expression as parameter. If the condition is false, the program is terminated, and an informative message is printed to the screen. If an invariant is not met at a certain point, the program stops immediately, before any other (also incorrect) command could modify the variables in a way that renders the invariant true again. Thus we avoid overlooking a logic error in the program.

On the other hand, a *static assert* is one of the possible ways to handle a metaprogram situation, when a compile-time requirement is not met by a type or a value. A *static assert* is capable of halting the compilation of a program at the point of the error's detection, thus we can avoid an incorrect program to come into being. At the same time, we aspire to create a static assert that contains some sensible error message, thus it is easier for the programmer to find the bug.

The simplest way to do this is by using a macro as defined in [28, 7], whose simplified version is as follows:

```
template <bool> struct STATIC_ASSERT_FAILURE;
template<> struct STATIC_ASSERT_FAILURE<true>{};
template<int x> struct static_assert_test{};

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

If the expression `B` is true, the existing specialisation of `STATIC_ASSERTION_FAILURE` is used as the `sizeof`'s argument. Otherwise the missing specialisation for `false` causes a compile-time error. In the `error` argument a typename has to be provided that passes messages for the programmer:

```
struct SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_LONG {};
```

```
STATIC_ASSERT(sizeof(int)==sizeof(long),
             SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_LONG)
```

Static assert is a widely used technique for C++ programs using templates [2, 25].

4.2 Concept checking

One of the most specific characteristics of C++ templates is that we are not able to explicitly specify the requirements for a type used as a template argument. In other words: C++ templates are not constrained. This is an intentional design decision of the language [17]. However, there are situations when this feature might cause unwanted behaviour of template metaprograms.

The possibilities for specifying requirements for template arguments are researched extensively, and the results are implemented in libraries like `boost::concept` [25], or Alexandrescu's `Loki` [2]. A template introspection library was proposed in [24].

Major efforts are underway to extend the C++ language with the explicit language support for constrained generics. Concepts (a type system for templates) can reduce the possibility of misusing templates and enhance better modularity. Early discussions on concept checking can be found in [10] and [12]. Currently there are two proposals for constrained generics for C++. The proposal of Stroustrup and Dos Reis [18] draws motivation from generic programming. Another proposal originated from Siek et. al. [13] is based on earlier comparative studies on generic language features of various programming languages [6]. An experimental compiler derivate – ConceptGCC – already supports the latest proposal.

Both proposals are targeting key weaknesses of the C++ template facilities. Current template instantiation techniques combine information from both definition and instantiation context. This reduces the possible separation of template definitions from its user context. Therefore C++ template definitions are harder (if not impossible) to modularise and check deeply compared to other languages like ADA [9].

4.3 Objectives for a metaprogram debugger

As we have seen in Section 3, compiler diagnostics are often useless when aiming to detect the place and cause of the unwanted behaviour of C++ template metaprograms. In these situations we need to debug our code with different methods. These methods have to include the following major components:

- We have to follow the ”control flow” of the program. In case of template metaprograms, this is equivalent to following the chain of the template instantiations. We should give special attention to recognising recursive instantiations.
- We need to inform the programmer of the ”variables'” values (i.e. the arguments and static const values of templates being instantiated). The debugger has to track the values of variables, and these values have to be made available to the programmer.
- To increase the efficiency of debugging the user should be able to set breakpoints – meaning suspending the compilation process when templates with certain arguments are instantiated
- The user should be able to control the debugging process to step into, step over, or step out of instantiation events.

Tools we use in everyday programming for debugging are not available in the well–known way, when dealing with metaprograms. We have no command for printing to the screen (in fact we have practically no commands at all), we have no framework to manage running code. On the other hand, we still have some options. Having a set of good debugging tools in the runtime world and a strong analogue between the runtime and compile-time realm we can attempt to implement a template metaprogram debugging framework.

A common property of debugging tools is that they analyse a specific execution of our program. In most cases this execution does not depend on the usage of the tool[1]. In such cases it does not matter whether we are using the tool on the running program or on a previously generated trace of its runtime steps.

One of the possible solutions is the modification of the C++ compiler to serve the information required for debugging purposes. With the strong help of compiler vendors, this approach has many benefits. However, to identify the exact specification for such a major step earlier experiences are needed. Such results could be obtained from more portable, lightweight solutions with the co-operation of standard C++ language elements and external tools.

Applying the analogue we can say that if we had a trace file that contains all template instantiations, the appropriate template parameters, the inner `typedef` definitions, timestamps for a profiler etc. step by step along the compilation process then we could apply the tools to analyse the behaviour of our template metaprograms.

All we need is a utility that generates a detailed trace file about the compilation process.

5. Implementation

Most compilers generate additional information for debuggers and profilers. Obviously the simplest way for providing information for our debugging framework would be the implementation of another compiler setting that makes the framework generate the desired trace file. However, an immediate and more portable solution is to use external tools cooperating with standard C++ language elements. An appropriate compiler support could be an ideal long-term solution.

Without the modification of the compiler the only way of obtaining any information during compilation is generating informative warning messages that contain the details we are looking for [1]. Therefore the task is the *instrumentation* of the source, i.e. its transformation into a functionally equivalent modified form that triggers the compiler to emit talkative warning messages. The concept of such instrumentation is a usual idea in the field of debuggers, profilers and program slicers [11]. Everytime the compiler instantiates a template, defines an inner type etc. the inserted code fragments generate detailed information on the actual template-related event. Then we have to gather the desired information from the corresponding warning messages in the compilation output and form a trace file. The front-end system uses this trace file to implement its various debugging features.

5.1 Overview

The input of the process is a C++ source file and the output is a trace file, a list of events like *instantiation of template X began*, *instantiation of template X ended*, *typedef definition found* etc. The procedure begins with the execution of the preprocessor with exactly the same options as if we were to compile the program. As a result we acquire a single file, containing all #included template definitions and the original code fragment we are debugging. The preprocessor decorates its output with #line directives to mark where the different sections of the file come from. This information is essential for a precise jump to the original source file positions as we step through the compilation while debugging in an IDE for example. To simplify the process we handle the mapping of the locations in the single processed file to the original source files in a separate thread. Simple filter scripts move the location information from #line directives into a separate mapping file and delete #line directives.

At this point we have a single preprocessed C++ source file, that we transform into a C++ token sequence. To make our framework as portable and self-containing as possible we apply the `boost::wave` C++ parser. Note that even though `boost::wave` supports preprocessing, we still use the original preprocessor tool of the compilation environment to eliminate the chance of bugs occurring due to different tools being used. Our aim is to insert warning-generating code fragments at the instrumentation points. As `wave` does no semantic analysis we can only recognise these places by searching for specific token patterns. We go through the token sequence and look for patterns like *template keyword* + *arbitrary tokens* + *class or struct keyword* + *arbitrary tokens* + { to identify template definitions. The end of a template class or function is only a } token that can appear in quite many contexts, so we should track all { and } tokens in order to correctly determine where the template contexts actually end. This pattern matching step is called annotating, its output is an XML file containing annotation entries in a hierarchical structure following the scope.

The instrumentation takes this annotation and the single source and inserts the warning-generating code fragments for each annotation at its corresponding location in the source thus producing a source that emits warnings at each annotation point during its compilation. The next step is the execution of the compiler to have these warning messages generated. The inserted code fragments are intentionally designed to generate warnings that contain enough information about the context and details of the actual event. Since the compiler may produce output independently of our instrumentation, it is important for debugger warnings to have a distinct format that differentiates them. This is the step where we ask the compiler for valuable information from its internals. Here the result is simply the build output as a text file. The warning translator takes the build output, looks for the warnings with the aforementioned special format and generates an event sequence with all the details. The result is an XML file that lists the events that occurred during the compilation in chronological order. The annotations and the events can be paired. Each event signals that the compiler went through the corresponding annotation point. We can say events are actual occurrences of the annotation points in the compilation process.

5.2 Preprocessing

The C++ preprocessor is executed in order to instrument all templates that appear in the given compilation unit. It is important to enable the #line directives since they serve the mapping between the original source file positions and the positions in the single preprocessed file. By using the same preprocessor as in the normal compilation process all possible preprocessor tricks done in the source are processed exactly the same way, and there are no compatibility issues even if the code contains preprocessor metaprogramming or any other compiler-dependent technique.

5.3 #line filtering

Simple filters are executed that eliminate the #line directives from the preprocessed source. At the same time they generate a map file that later can be used to retrieve the original positions with the position adjuster.

5.4 Annotating

The annotator parses the C++ source (using `boost::wave`) and looks for special token sequences like

- beginning of scope
- end of scope

[1] this immediately leads to a limitation described in Section 6

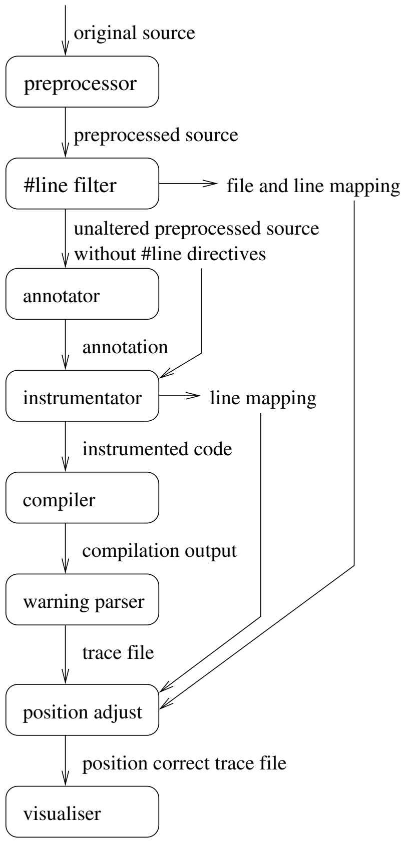

Figure 1. Architecture of debugging framework

- beginning of template class definition (optionally a specialisation)
- beginning of template function definition (optionally a specialisation)
- (inner) typedef definition

We use the following grammar:

```
openBrace  = wave::T_LEFTBRACE ;
closeBrace = wave::T_RIGHTBRACE ;

templateClassBegin = wave::T_TEMPLATE >>
    *(anychar_p - (
        wave::T_SEMICOLON |
        wave::T_LEFTBRACE |
        wave::T_CLASS     |
        wave::T_STRUCT
    )) >>
    +(
        (wave::T_CLASS | wave::T_STRUCT) >>
        *(anychar_p - (
            wave::T_CLASS     |
            wave::T_STRUCT    |
            wave::T_SEMICOLON |
            wave::T_LEFTBRACE
        ))
    ) ;
templateClassDeclaration =
    templateClassBegin >> wave::T_SEMICOLON ;
templateClassDefinition =
    templateClassBegin >> wave::T_LEFTBRACE ;
```

```
templateFunctionBegin =
    wave::T_TEMPLATE >>
    +(
        *(anychar_p - (
            wave::T_SEMICOLON |
            wave::T_LEFTBRACE |
            wave::T_RIGHTPAREN
        )) >>
        wave::T_RIGHTPAREN
    );
templateFunctionDeclaration =
    templateFunctionBegin >> wave::T_SEMICOLON ;
templateFunctionDefinition =
    templateFunctionBegin >> wave::T_LEFTBRACE ;

typedefDeclaration =
    wave::T_TYPEDEF >>
    *(anychar_p - (
        wave::T_SEMICOLON  |
        wave::T_IDENTIFIER
    )) >>
    *(wave::T_IDENTIFIER >>
        +(anychar_p - (
            wave::T_SEMICOLON  |
            wave::T_IDENTIFIER
        ))
    ) >>
    wave::T_IDENTIFIER >> wave::T_SEMICOLON ;
```

The grammar is implemented by `boost::spirit` rules that operate on the token sequence of the wave output. The result is an XML file that describes the structure of the source file. Consider the following simple example:

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

template<>
struct Factorial<1>
{
    enum { value = 1 };
};

int main ()
{
    return Factorial<5>::value;
}
```

The resulting XML file is the following:

```
<?xml version="1.0" standalone="yes"?>
<!-- [annotation] generated by Templight -->
<FileAnnotation
    beginpos = "test2.cpp.preprocessed.cpp|1|1"
    endpos   = "|1|1">
    <TemplateClassAnnotation
        beginpos = "test2.cpp|1|1"
        endpos   = "test2.cpp|5|2"
        afteropenbracepos = "test2.cpp|3|2"
        beforeclosebracepos   = "test2.cpp|5|1">
        <ScopeAnnotation
            beginpos = "test2.cpp|4|7"
            endpos   = "test2.cpp|4|40"/>
    </TemplateClassAnnotation>
    <TemplateClassAnnotation
        beginpos = "test2.cpp|7|1"
        endpos   = "test2.cpp|11|2"
        afteropenbracepos = "test2.cpp|9|2"
        beforeclosebracepos   = "test2.cpp|11|1">
        <ScopeAnnotation
            beginpos = "test2.cpp|10|7"
            endpos   = "test2.cpp|10|20"/>
    </TemplateClassAnnotation>
    <ScopeAnnotation
        beginpos = "test2.cpp|14|1"
        endpos   = "test2.cpp|16|2"/>
</FileAnnotation>
```

The outmost `FileAnnotation` element represents the source file, while the two `TemplateClassAnnotation` elements correspond to the general `Factorial` template class definition and its specialisation, respectively. The `beginpos` and `endpos` attributes show the borders of the definition while `afteropenbracepos` and `beforeclosebracepos` contain the locations where the instrumented code fragments should be inserted. Note the presence of the `ScopeAnnotation` elements following all '{' ... '}' pairs regardless of their meaning.

5.5 Instrumenting

The instrumentator takes the annotation file and for each element that marks a debug point it inserts a corresponding code fragment that generates a compilation warning containing the desired information. In the current implementation we use a `double` to `unsigned` conversion. This construct is proved to emit a warning when compiled by current C++ compilers. However, it is possible to insert multiple warning generator fragments to cover a wider range of compilers. At the warning parsing step (in 5.7) the unnecessary warnings are dropped.

The other output of this step is the line mapping that enables the position adjuster to retrieve the original positions from the line numbers referring to the instrumented code in the warnings. Inserted code fragments were defined to be semantically neutral, i.e. they must not affect the functionality of the original template metaprogram. The result of the previous example's instrumentation is the following:

```
/* ---------------- begin inserted ----------- */
namespace Templight {
    template<class C, int C::*>
    struct ReportTemplateBegin {
        static const unsigned Value = -1.0;
    };
    template<class C, int C::*>
    struct ReportTemplateEnd {
        static const unsigned Value = -1.0;
    };
    template<class C, int C::*, class Type>
    struct ReportTypedef {
        typedef Type Result;
        static const unsigned Value = -1.0;
    };
}
/* ----------------- end inserted ------------ */
template<int i>
struct Factorial
{
/* ---------------- begin inserted ----------- */
struct _TEMPLIGHT_0s { int a; };
enum { _TEMPLIGHT_0 =
    Templight::ReportTemplateBegin<
        _TEMPLIGHT_0s, &_TEMPLIGHT_0s::a
    >::Value
};
/* ----------------- end inserted ------------ */
    enum { value = Factorial<i-1>::value };
/* ---------------- begin inserted ----------- */
struct _TEMPLIGHT_1s { int a; };
enum { _TEMPLIGHT_1 =
    Templight::ReportTemplateEnd<
        _TEMPLIGHT_1s, &_TEMPLIGHT_1s::a
    >::Value
};
/* ----------------- end inserted ------------ */
};

template<>
struct Factorial<1>
{
/* ---------------- begin inserted ----------- */
struct _TEMPLIGHT_2s { int a; };
enum { _TEMPLIGHT_2 =
    Templight::ReportTemplateBegin<
        _TEMPLIGHT_2s, &_TEMPLIGHT_2s::a
    >::Value
};
/* ----------------- end inserted ------------ */
    enum { value = 1 };
/* ---------------- begin inserted ----------- */
struct _TEMPLIGHT_3s { int a; };
enum { _TEMPLIGHT_3 =
    Templight::ReportTemplateEnd<
        _TEMPLIGHT_3s, &_TEMPLIGHT_3s::a
    >::Value
};
/* ----------------- end inserted ------------ */
};

int main ()
{
    return Factorial<5>::value;
}
```

There is a fixed part instrumented to the beginning of the file containing the following helper types: `ReportTemplateBegin`, `ReportTemplateEnd`, `ReportTypedef`.

The occurrence of these helper types in a warning message obviously signals that the compiler met an instrumented code fragment. Each instrumentation introduces a new inner class and passes an address of its static variable to the helper templates to enforce new instantiations each time. Having this parameter omitted we would get the warning only at the first – the only – instance of the helper template. Unfortunately this trick is not always enough to get the desired warnings for each instance (see Section 6).

typedef instrumentation Some compilers display the `typedef` identifier rather than the actual type in the warning messages which makes diagnostics more difficult, as described in [1]. To overcome this problem the instrumentation slightly modifies the type of the `typedef` and inserts a forwarding template in order to have the actual type displayed. This forwarding type is `ReportTypedef`. The following example shows a `typedef` instrumentation. The original source is the following:

```
template<int N>
struct Test
{
    typedef Int<N> IntType;
};
```

And the corresponding instrumentation:

```
template<int N>
struct Test
{
    // { ... ReportTemplateBegin part ... }

struct _TEMPLIGHT_2s { int a; };
typedef typename
    Templight::ReportTypedef<
        _TEMPLIGHT_2s, &_TEMPLIGHT_2s::a,
        Int<N>
    >::Result IntType;
};

    // { ... ReportTemplateEnd part ... }
};
```

5.6 Compiling

The C++ compiler is executed on the instrumented source file to get the talkative error messages. If the original source is ill-formed, the compiler may emit compilation errors and warnings independent of our framework. These messages are preserved since they are also converted to events by the warning parser. The compilation may stop on serious errors caused by the original source. In this case the trace contains all the events up to that point. This way it is still possible to debug the compilation and follow the control up to the point our compilation aborted. This situation is similar to the crash of an ordinary program, where one can debug the behaviour to the point of the problematic instruction. Sometimes the running program even survives several illegal instructions, exactly the same way as the compiler does not stop at the first error. A fragment of the previous example's compilation output:

```
test2.cpp.patched.cpp(1) : warning C4244:
  'initializing' : conversion from 'double' to
  'const unsigned int', possible loss of data
      test2.cpp.patched.cpp(9) : see reference to
      class template instantiation
      'Templight::ReportTemplateBegin<C,__formal>'
      being compiled
      with
      [
          C=Factorial<4>::_TEMPLIGHT_0s,
          __formal=pointer-to-member(0x0)
      ]
      test2.cpp.patched.cpp(10) : see reference to
      class template instantiation
      'Factorial<i>' being compiled
      with
      [
          i=4
      ]
```

Here we can see the artificially generated warning message that comes from our `Templight::ReportTemplateBegin` template class showing that it is not a warning of the original compilation, but an instrumented one. In order to catch the original context an inner class is passed to this template class. As our previous compiler output shows it is recognisable that the warning is originated from the `Factorial<4>` template instance. In this example the compiler also helped us by reporting the context, but this backtrace is not always present.

5.7 Parsing warnings

The warning parser takes the compilation output and looks for our special warning messages and collects the encoded information as well as the position and instantiation history of the event. The result is an XML file containing a sequence of events, where all events correspond to an element in the annotation file, set up with the actual properties of activation. The resulting event entry of the warning is the following:

```
<TemplateBegin>
    <Position position=
        "test2.cpp.patched.cpp|9|1"/>
    <Context context="Factorial<4>"/>
    <History>
        <TemplateContext instance="Templight::
                ReportTemplateBegin<C,__formal>">
            <Parameter name="C" value=
                "Factorial<4>::_TEMPLIGHT_0s"/>
            <Parameter name="__formal"
                value="pointer-to-member(0x0)"/>
        </TemplateContext>
        <TemplateContext
            instance="Factorial<i>">
            <Parameter name="i" value="4"/>
        </TemplateContext>
    </History>
</TemplateBegin>
```

The name of the tag describes that this event is the beginning of a template instantiation, while the child elements contain additional details. The Position tag shows the location of the annotation point in the source, the Context tag shows the actual template context, and the History tag lists the template instantiation backtrace that was present in the warning.

5.8 Adjusting positions, visualisation

The position adjuster corrects the file positions in the trace file using the previously saved line mappings of the #line filter and instrumentator steps. Here we have a full trace of all the events that happened during the compilation with exact source file positions. This is an ideal input for post-processing tools.

A debugger frontend maintains breakpoint positions set by the user. Traversing the trace the frontend stops at each breakpoint and allows the user to examine the *instantiation stack* of templates including template names and arguments at each level. Then she can advance to the next breakpoint or take only one step. In the second case she can *step into* the next template instantiation inside the actual one, *step over*, i.e. skip nested instantiations, or *step out* skipping all instantiations until reaching the context where the actual template was instantiated. Thus the programmer is able to focus on desired portions of the compilation process and can reduce debugging effort.

A profiler frontend may display the template instantiations sorted by their compile time. The compile times can mean only local times, where the times of nested instantiations are subtracted, or full times meaning the full time spent inside the given instantiations.

5.9 Remarks

The framework was intentionally designed to be as portable as possible, for this end we tried to use portable and standard-compliant tools. All components except for the #line filter scripts are written in C++ using the `STL`, `boost` and `Xerces` libraries. The only compiler-dependent step in our framework is the warning parser.

6. Limitations and future works

No intervention Note that while in traditional debugging the program is actually running when being debugged, in our case we only traverse the previously generated trace file. As a result we are unable to modify the values of the variables at a given point and see what happens if the variable had that value at that point. A real interactive template metaprogram debugger can only be implemented by strong intentional support of compilers. On the other hand we can see the whole process and can arbitrarily go back and forth on the timeline, which is impossible in immediate debugging.

Compiler support The process detailed above works only if the compiler gives enough information when it meets the instrumented erroneous code. Unfortunately not all compilers fulfil this criterion today. The table below summarises our experiences with some compilers:

compiler	result
g++ 3.3.5	ok
g++ 4.1.0	ok
MSVC 7.1	ok
MSVC 8.0	ok
Intel 9.0	no instantiation backtrace
Comeau 4.3.3	no instantiation backtrace
Metrowerks CodeWarrior 9.0	no instantiation backtrace
Borland 5.6	no warning message at all
Borland 5.8	no instantiation backtrace, but the warning message is printed for each instantiation

Table 2. Our experiences with different compilers

It is a frequent case when a warning is emitted, but there is no information about its context. Comeau 4.3.3., Intel 9.0, and Metrowerks CodeWarrior 9.0 print the `double` to `unsigned` warning only once in the `ReportTemplateBegin` helper template without referring to the instantiation environment.

The example shown in subsection 5.5 produces the following output using the Comeau C++ online evaluation compiler:

```
"ComeauTest.c", line 5: error:
    expression must have integral or enum type
          static const unsigned Value = -1.0;
                                         ^
"ComeauTest.c", line 9: error:
    expression must have integral or enum type
          static const unsigned Value = -1.0;
                                         ^
"ComeauTest.c", line 14: error:
    expression must have integral or enum type
          static const unsigned Value = -1.0;
                                         ^
3 errors detected in
    the compilation of "ComeauTest.c".
```

Similar output is produced by the Intel 9.0 compiler. The messages are only emitted once for each helper template regardless of the number of their actual instantiations.

The most surprising find was that the Borland 5.6 compiler does not print any warnings to our instrumented statement even with all warnings enabled. A later version of this compiler (version 5.8) prints the desired messages, but similarly to many others it does not generate any context information. In contrast to the others this compiler prints the same warning for each instantiation.

Since we do not have semantic information we fall back on using mere syntactic patterns. Unfortunately without semantic information there are ambiguous cases where it is impossible to determine the exact role of the tokens. This simply comes from the environment-dependent nature of the language and from the heavily overloaded symbols. The following line for example can have totally different semantics depending on its environment:

```
enum { a = b < c > :: d };
```

If the preceding line is

```
enum { b = 1, c = 2, d = 3 };
```

then the `<` and `>` tokens are relational operators, and `::` stands for 'global scope', while having the following part instead of the previous line

```
template<int>
struct b {
    enum { d = 3 };
};
enum { c = 2 };
```

the < and > tokens become template parameter list parentheses and `::` the dependent name operator. This renders recognising `enum` definitions more difficult.

Future works include the implementation of new IDE extensions providing better functionality for the debugger frontend. This includes the placing and removing of breakpoints, following the compilation process, and examination of the instantiation stack.

Another direction is to create a real, interactive debugger, that can suspend the compilation process at breakpoints, collect information, and is even able to modify the compilation process. As far as we know this can be done only by the modification of the compiler. However, the most adequate modifications and the protocol describing how the compiler and the outer tools interact are currently open.

7. Related work

Template metaprogramming was first investigated in Veldhuizen's articles [21]. Static interface checking was introduced by McNamara [10] and Siek [12]. Compile-time assertion appeared in Alexandrescu's work [2]. Vandevoorde and Josuttis introduced the concept of a *tracer*, which is a specially designed class that emits runtime messages when its operations are called [20]. When this type is passed to a template as an argument, the messages show in what order and how often the operations of that argument class are called. The authors also defined the notion of an *archetype* for a class whose sole purpose is checking that the template does not set up undesired requirements on its parameters. In their book on `boost` [1] Abrahams and Gurtovoy devoted a whole section to diagnostics, where the authors showed methods for generating textual output in the form of warning messages. They implemented the compile-time equivalent of the aforementioned runtime tracer (`mpl::print`, see [26]).

Even though in the preprocessing phase the `__TIME__` predefined macro can be used to determine the exact system time, it is impossible to get such information during the actual compilation step. Therefore profiling information can only be generated by the modification of the compiler. Using the Templight framework we implemented a light-weight template profiler for the g++ compiler. We modified the compiler to output timestamps when emitting the aforementioned warning messages. These timestamps are collected by the warning parser and then inserted to the trace file.

8. Conclusion

C++ template metaprogramming is a new, evolving programming technique. Even though it extends traditional runtime programming with numerous advantages, the lack of development tools retains its applicability in time-constrained, large-scale industrial projects.

In this paper we listed the analogues between the realms of runtime and compile-time programming. We set up an ontology of template metaprogram errors and defined the requirements for a template metaprogram debugger. We described our prototype framework that fulfils these requirements and supplies a portable solution for debugging template metaprograms in practice. We showed the details of the framework's operation step by step through examples.

Our framework is also an experimental tool towards discovering the requirements of intentional compiler support for debugging and profiling template metaprograms in the future.

References

[1] David Abrahams, Aleksey Gurtovoy: C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Boston, 2004.

[2] Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)

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

[4] Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)

[5] Ulrich W. Eisenecker, Frank Blinn and Krzysztof Czarnecki: A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.

[6] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock: A Comparative Study of Language Support for Generic Programming. Proceedings of the 18th ACM SIGPLAN OOPSLA 2003, pp. 115-134.

[7] Björn Karlsson: Beyond the C++ Standard Library, A Introduction to Boost. Addison-Wesley, 2005.

[8] David R. Musser and Alexander A. Stepanov: Algorithm-oriented Generic Libraries. Software-practice and experience, 27(7) July 1994, pp. 623-642.

[9] David R. Musser and Alexander A. Stepanov: The Ada Generic Library: Linear List Processing Packages. Springer Verlag, New York, 1989.

[10] Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.

[11] Krisztián Pócza, Mihály Biczó, Zoltán Porkoláb: Cross-language Program Slicing in the .NET Framework. Journal of .NET Technologies Vol. 3., Number 1-3, 2005 pp. 141-150.

[12] Jeremy Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.

[13] Jeremy Siek and Andrew Lumsdaine: Essential Language Support for Generic Programming. Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, NY, USA, pp 73-84.

[14] Jeremy Siek: A Language for Generic Programming. PhD thesis, Indiana University, August 2005.

[15] Ádám Sipos: Effective Metaprogramming. M.Sc. Thesis. Budapest, 2006.

[16] Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)

[17] Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)

[18] Gabriel Dos Reis, Bjarne Stroustrup: Specifying C++ concepts. Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006: pp. 295-308.

[19] Erwin Unruh: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.

[20] David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)

[21] Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[22] Todd Veldhuizen: Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[23] István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830 (2003), pp. 209 - 227.

[24] István Zólyomi, Zoltán Porkoláb: Towards a template introspection library. LNCS Vol.3286 pp.266-282 2004.

[25] Boost Concept checking.
`http://www.boost.org/libs/concept_check/concept_check.htm`

[26] Boost Metaprogramming library.
`http://www.boost.org/libs/mpl/doc/index.html`

[27] Boost Preprocessor library.
`http://www.boost.org/libs/preprocessor/doc/index.html`

[28] Boost Static assertion.
`http://www.boost.org/regression-logs/cs-win32_metacomm/doc/html/boost_staticassert.html`