Precompilation: an alternative approach to provide native Generic Programming support in C++

Daniel F. Gutson
GSG Argentina
daniel.gutson@motorola.com

Román L. Alarcón
GSG Argentina
roman.alarcon@motorola.com

Abstract

In C++, Generative Programming (GP) techniques are being used to generate highly customized and optimized products automatically manufactured at compile-time; to provide these functionalities increasing compiling power is required.

This work presents an improved compilation model for C++ by adding the 'precompilation' phase, leading beyond the Template Meta Programming technique to produce constants and conditional code.

Procedural, object-oriented and all the remaining language features become available to produce constants, instances, and compile-time checks, opening, at the same time, a new way for metadata types treatment. In addition to that, when compiling for embedded platforms, some calculi may be moved from resource-critical run time to compile time, taking advantage of the processing power of the host platform.

A tool named PRECOMP C++ is also presented in this work as a precompilation-enabled C++ extension that supports GP in standard C++ execution during compile time, providing the ability to run metaprograms that operate with more complex data types and features than those supported in Template Meta Programming, such as floating point, pointers arithmetic, inclusion polymorphism, and dynamic memory.

Keywords: *C++*, *C++ Templates*, *Code Generation*, *Generative Programming*, *Metacompiler*, *Template Metaprogramming*.

1. Introduction

One of the current trends in Computer Science aims at facilitating the development of applications using compile-time code generation techniques. Through these techniques it is possible to build software components that can be customized before building the final application [4].

The use of templates (*Generic Programming*) in C++ provides a way to perform static computations and generate code at compile-time. From this fact, and almost accidentally, the *template metaprogramming technique* arose. The template metaprogramming technique uses the compiler as a transformational function that interprets a template and generates other programs that are later compiled as normal C++ code. With this in mind, templates can be seen as *metaprograms* (programs that generate other programs), because they are "executed" by the compiler and generate code that constitutes a new program, according to the parameters used during the template instantiation. By using this technique it is possible to perform partial evaluations such as loop unrolling [10], which is useful to create optimized applications.

As a simple example of a template metaprogram, let's suppose that we want to calculate the factorial of a number at compile-

time and use its output value as the input of some function. We can write the metaprogram as follows:

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

template<> struct Factorial<0>
{
  enum {value = 1};
};
```

This is a simple C++ template structure containing an enumeration which has only one value. This value is computed by recursively "invoking" at compile time the Factorial metaprogram as the template is interpreted. The first template represents the inductive step in the recursion, and the last one acts as the base case. To invoke this metaprogram we can write something like this:

```
int main()
{
  ...
  Array a(Factorial<6>::value);
  ...
}
```

supposing that there is a class Array having a constructor requiring an integer value to know the length of the array to be constructed.

Metaprograms are taken as a base for *Generative Programming* (*GP*), a programming paradigm that allows the modeling of families of highly customized and optimized software systems, by means of the use of software entities able to build those families automatically under demand. Active Libraries (*AL*) use generative programming, and put together normal code and metacode (metaprograms). ALs can generate components and algorithms and can also specialize or optimize code. In addition, they can interact with other libraries to produce concrete components and to adapt them to a particular system [4].

In this paper we show an alternative way to provide native GP support in C++ by introducing metaprogram execution in a *pre-compilation* phase. This new phase is separated from the rest of the normal C++ compiling process, and allows the simulation of metaprogram calling by the introduction of *tags* representing metaprograms added to the user code. The new phase of pre-compilation along the rest of the phases of the C++ compiling process (preprocessing, compiling, optimizing and linking) is referred here as the *macrocompiling process*. In macrocompilation a program containing metacode goes through the following phases:

1. Preprocessing (the normal C++ preprocessor behavior);
2. Precompilation, which performs parsing of tags, code generation and injection of the generated code in the final code to be compiled;
3. Compilation (normal C++ compiling-optimizing-linking behavior).

Metaprograms inserted in the code can be considered as part of an AL that generates customized and optimized code according to the target system. Once the final code is generated by the AL, the C++ compiler receives the output and treats it as a normal C++ program, as if it was written "by hand". With this technique we can run many tasks at compile time, avoiding them once the program is executing. Thus, the target platform can be focused only on those tasks that are really important.

A brief description of some techniques and tools that use GP is offered. Also, we describe a simple tool called PRECOMP C++ that implements the proposed pre-compilation phase combining tag parsing and transformation with template metaprogramming. There are some case studies and a qualitative comparison between our approach and that selected by other tools. Also, we describe the further work regarding this alternative concept.

Notation note: we'll refer "precomp-C++" as the concept described in this work, whose compilation model includes the precompilation phase, whereas we will refer PRECOMP C++ as the tool for implementing the concept.

2. Current implementations

There are many ways to implement GP that are different in complexity and ability to generate code at compile time. The most trivial one is that related to the computation of a single value from an expression involving only constant values. For example, in the expression (2+3)*5 the compiler generates the constant value 25 before using it to assign to a variable or to pass as argument of some function. Second, we have the C++ preprocessor. This preprocessor allows the programmer to control the flow of the preprocessing activity by means of directives such as **#if** and **#elif**, thus allowing to format the code to be compiled and to tailor it regarding the target platform. Macro expansion provides another simple way to inject code into an application by replacing each macro calling by the text written in the #define used to define the macro. However, macros have fewer uses in C++, and it is suggested not to use them unless it is necessary. Because only the expanded form of a macro is seen by the compiler, it's difficult for the compiler to report errors before the expansion is performed [8].

As stated in the previous section, another way to implement GP is by means of the Template Metaprogramming technique (TMP). Templates were designed to provide generic programming, but accidentally it was discovered that they allowed writing code generators and executing static computations. Using templates facilitates the injection and inlining of code when the template is instantiated, thus allowing (in some cases) optimizations such as forcing the use of the stack instead of using dynamic memory [10]. However, the construction of template metaprograms is not easy and the resulting syntax gets hard to read [10]. The programmer needs a solid background to develop useful and interesting metaprograms.

Through his work, Daveed Vandevoorde proposed an extended C++ language implementing metacode [9]. In a program using metacode there are some functions that can be evaluated at compile time. Also, there exist some mechanisms of code injection (in the scope of a class or a namespace) and a *standard library* of metacode can be used. As a limitation, none of the functions involved in the metacode (meta-functions) can be ***virtual*** or invoke other non meta-functions.

There are other tools such as Open C++ that implement a mixing between the C++ language and the *Metaobject Protocol*. A *metaobject* is any entity that exhibits aspects of an object (the object's type, its interface, its methods and attributes, and so on), and a metaobject protocol is a generalized way to handle a group of metaobjects as a whole. With Open C++ a programmer can develop different translations of the source code, define new syntax and new object behavior [3]. With these elements at hand, it is possible for a programmer to develop a mechanism to introduce a phase of precompilation. In this work, Open C++ was considered to implement the concepts; after the analysis a simpler alternative was selected to demonstrate the feasibility of using existing, well known off-the-shelf standard-compliant C++ compilers.

Another work about the need to improve the way C++ language is compiled is, for example, proposed in [11], where it's possible to remove type analysis from the compiler by introducing a separate *type system library* that is treated by the compiler as source code. The library is used along with the user code as input to the phase of lexical and syntax analysis of the compiler. The compiler's front-end inserts calls to this library when translating the source code. This shows that the idea of separating (or adding) some activities from (or

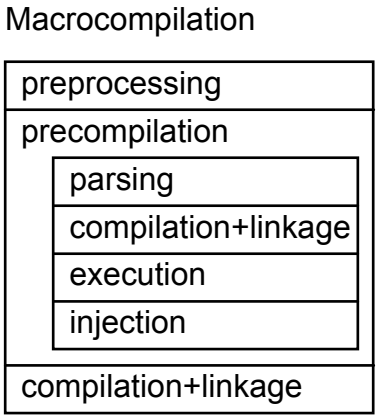
to) the compiling process is an important and useful approach.

There also exists a *metacompiler*, called FOG (Flexible Object Generator) [12], that offers an alternative to the C++ preprocessor. FOG extends C++ doing the C++ preprocessor redundant (that is, it is no necessary to use it) by providing a C++ dialect in which it is possible to write metaprograms. In this dialect, all of the macros are replaced by meta-variables and meta-functions, and the preprocessor statements are replaced by meta-statements.

3. Precomp-C++ implementation

The main goal of the compilation model described in this paper (referred as *macrocompilation*) is to provide the ability of writing metacode in standard C++ language, including the ability of invoking user and standard libraries. A secondary goal is to use the host system (where the macrocompilation occurs) to execute the metacode. Lastly, a third goal of this work is to keep the syntax standard-C++ compliant.

In order to achieve these goals, the macrocompilation herein described includes a phase named *precompilation*, after the preprocessing and before the compilation.



The basic idea of the precompilation phase is to extract the metaprograms contained in *tags*, compile them using a standard C++ compiler, execute them, and inject the output back in the original code. The effect is seen as replacing the tags by their evaluation results.

To do so, the precompilation phase requires the following components:

- a simple parser
- a standard C++ compiler
- a system caller for executing the code
- a results injector

3.1 Syntax Specialization

In order to identify the code to extract and execute (and which results will be injected back into the original code) we propose to identify a specific tag which looks exactly like a standard template-function call, which will be referred as the *PRECOMP-TAG*. The C++ declaration of the PRECOMP-TAG is:

```
template <class T> T PRECOMP(T expression);
```

We define the '*expression associated to a PRECOMP-TAG*' to the PRECOMP-TAG function parameter. Similarly, we define the '*type associated to a PRECOMP-TAG*' to its template parameter. We also define the '*metaprogram M of the program P*' to the set of statements contained in the PRECOMP-TAG calls present in program P. We will use the following notation:

$$M = \text{metaprogram}(P)$$

According to the third goal mentioned early in this section, M shall be standard C++ compliant.

3.2 Semantic

Given a program P, the semantic that the precompilation phase carries out is to compile and execute the metaprogram(P), using a standard C++ compiler and the host system respectively; finally, the precompilation phase

replaces each PRECOMP-TAG by the actual execution result of its associated statement.

The semantic is achieved by the components mentioned above, which have the following roles:

- The role of the precompilation parser is to obtain the metaprogram, by identifying the PRECOMP-TAG calls, extracting both their associated expressions and their associated types.
- The role of the standard C++ compiler (within the precompilation context) is to compile the metaprogram extracted by the parser, and generate an executable (named '*temporal-executable*').
- The role of the system caller is to execute the temporal-executable, using the host system
- The role of the results injector is to get the results generated by the execution of the metaprogram, and replace with them each PRECOMP-TAG which they were obtained from, generating a new standard C++ code (named '*transformed-code*')

Parsing errors detected by the precompilation parser, compilation errors detected while executing the standard C++ compiler, and execution errors (or thrown exceptions) occurred during execution of the temporal-executable will be called *macrocompilation errors*. This set of situations act as an early fault barrier that allows safe checking before run time.

During the precompilation phase, the following actions may take place:

- calculate constants
- interact with I/O streams (such as reading and parsing external files)
- check conditions and throw exceptions (which will be seen as macro compilation errors)
- invoke external APIs

It's important to note that *any* of these actions can be coded in standard C++ code, optionally involving Template Meta Programming code.

Since the metacode is *actually* executed in the host system, the following features are available:

1. STL and stdlib
2. Dynamic memory
3. Global instances and singletons
4. Polymorphism (parametric and inclusion)
5. Exception handling, RTTI
6. Template Meta Programming
7. User libraries
8. Standard input/output

Additionally, the PRECOMP execution limits are the same as the host system limits, including performance and resources.

3.3 The PRECOMP C++ prototype

We developed an implementation tool named *PRECOMP C++* to prove the macrocompilation concept. The PRECOMP C++ generates instances of data during precompilation-time, which will become constants in run-time. Additionally, the PRECOMP C++ catches any exceptions thrown by the metaprogram, reporting them as macrocompilation errors.

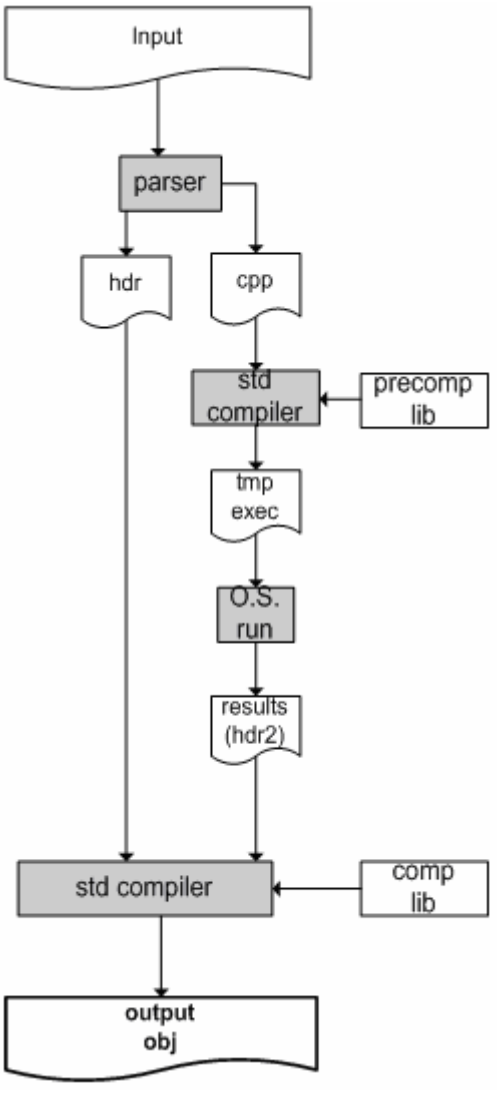
The PRECOMP C++ implements the macrocompilation model by using the following components:

- a **script** batch file to invoke the phases in a sequenced manner, providing the macrocompilation front-end
- a handwritten **parser** as the precompilation parser (written in C++)
- three commercial **compilers** (Intel® C++ Compiler, Microsoft® Visual C++ Toolkit 2003, and Comeau C/C++™ Compiler) to alternately play the role of the standard compiler (being the host and target

systems the same)

- a C++ library to generate results with the metaprogram (referred as '**precomp-time library**'), acting as the precompilation-time part of the results injector
- a C++ library with inlined functions to embed the results (referred as '**compile-time library**'), acting as the compile time part of the results injector.

Usage:



According to the syntax specialization described before, the user encloses the expressions to be evaluated during precompilation time by providing the PRECOMP-TAGs, in the following syntax:

`PRECOMP<`*`type`*`>(`*`expr`*`)`

where *expr* is the expression associated to the PRECOMP-TAG, and can be any C++-valid expression (including a function call), and *type* is the return type of *expr*.

The whole compilation process is led by the **script**, which invokes the **parser** and the standard C++ **compiler**. The **parser** recognizes all the PRECOMP-TAGs, and generates a temporary C++ file containing the metaprogram of the Input, and a temporary C++ header file.

Next, the **script** invokes the host standard C++ compiler for compiling the generated temporary code together with the **precomp-time library** and any user-defined library, to generate a temporal executable.

Then, the **script** runs the temporal executable, which generates a C++ header containing the results of the PRECOMP-TAGs associated expressions. This C++ header file with the generated C++ file will constitute the transformed code.

Finally, the **script** invokes the (target) standard C++ **compiler** again with the transformed code plus the **compile-time library** for compiling the final object file, which embeds the results generated during precompilation-time.

When cross-compiling for a different platform, the first compiler is the compiler for the host platform, whereas the second is the cross-compiler.

During the whole macrocompilation process, the following events may occur:

1. a parsing error, from the **parser**
2. a compiler error when compiling the intermediate files
3. a run-time error when executing the temporal executable (such as an exception caught)

4. a compiler error when compiling the transformed code

4. Case Studies

In order to evaluate the abilities of the precompilation phase, the following cases were analyzed using the PRECOMP C++ tool:

a. calculus of constants
b. embedding external raw data files
c. user-defined literals
d. compile-time memory allocation
e. others: Finite State Machines (FSMs) and Graphic User Interfaces (GUIs).

These examples are used to illustrate the concept that, by itself, it is not limited to the presented cases. The tool was built as an instrument to put the concept into practice and can be easily extended.

A) Calculus of constants

Constant values calculus requires both the source code to perform the calculation, memory for temporary variables, and the time for the process itself. In tight-equipped systems (such as many embedded systems), this process is not possible to be performed in run-time. Two examples are exposed here: the calculus of PI, and the calculus of a number raised to a power:

```
const double PI = PRECOMP<double>(atn(1)*4);
const char str[PRECOMP<int>(pow(3,4))];
```

In the first example, the constant value of PI is obtained by invoking the ATN function and the * operator, as the PRECOMP expression.

In the second example, the result of the expression 3^4 is used as the dimension of the 'str' array. Note that ANSI/ISO C++ forbids the declaration of a static array dimensioned by a non-const expression [7].

B) Embedding external raw data files

When data is stored externally to the C++ source file, the application may include the data statically (during compile time), or dynamically (by loading it during run time from an external file).

The usual way of embedding data statically into a source file is through constant arrays initialized with braces, such as

```
const int c[]={ 0x123, 0x321,  0x333, 0x222};
```

This involves importing the external data and performing a C++-like syntax conversion, either manually or with a tool.

The following two examples show how PRECOMP can be used to embed external raw data:

Example 1: the text file. Let's assume that a command-line console application has to show a help text when it receives the wrong number of arguments. Such text is maintained externally in a free-text file named 'help.txt'. The PRECOMP C++ environment has the ability of storing dynamically loaded precompilation-time data as static constant compile-time data. The example can be solved by loading the text file into a PRECOMP_STRING (which is a specialization of the STL's *string*):

```
#include <string>
#include <fstream>
#include "precomp_types.h"
using namespace std;

  PRECOMP_STRING loadFile(const char* file)
  {
    ifstream f;
    string line, ret;
    f.open(file);
    if(f.bad()) throw "File not found!";
    while (getline(f,line))
      ret += line;
    return ret;
  }

  void showHelp()
  {
    const PRECOMP_STRING help =
       PRECOMP<PRECOMP_STRING>(
          loadFile("help.txt"));
```

```
  cout << help;
}
```

Three important observations can be made in this example: the first is that the 'loadFile' function would be executed during run-time in a standard C++ file; however, such execution takes place in precompilation time due to the PRECOMP<>() statement. The second is that if the "help.txt" file is not found, an exception is thrown during precompilation time, which will be seen as a MACROCOMPILER ERROR. The PRECOMP C++ tool catches all the exception types (instead of showing up during run-time), and has specific behavior for some of them; in the case of a const char*, the PRECOMP C++ shows the text and stops the execution of the temporal generated application. Lastly, the PRECOMP C++ provides class wrappers for some of the standard containers (i.e. string, vector, map) in order to encapsulate the constant storage of dynamically-allocated content. Additionally, the PRECOMP C++ defines an interface for any data type that contains pointers, so any user-defined class can be properly used for precompilation time processing. Specifically, any *type* provided in the PRECOMP-TAG that is a model of the *PRECOMP template concept* can be used in the precompilation process.

The *PRECOMP template concept* implies:

- The casting-to-unsigned integral operator
- The casting-to-void pointer operator

In fact, the PRECOMP_STRING wrapper just provides those operators by returning the size of the string, and the address of the first character, respectively.

Example 2: embedding an image stored as a graphic format file. Another usage not detailed in this publication can be loading a graphic file (e.g. a GIF file) from a file during precompilation time. In such case, both file access and format consistency can be checked before run time.

C) User defined literals

There are several generic-length integer C++ library implementations. However, there is a performance bottleneck to enter literals since they have to be parsed. For example, the number

```
12,345,678,901,234,567,890
```

does not fit in a 32-bits integer. Many big-integer libraries, such as GMP [6], use strings to assign values. The latter number should be assigned in GMP with the following statement:

```
e=mpz_set_str(R, "12345678901234567890", 0);
```

This statement parses the string, determines the base, and assigns it to the big-integer R. The error code is returned to e. However, the whole process could take place during precompilation-time, including the validation, where a parsing error (i.e. the string contains an invalid character) would be reported as a *MACROCOMPILER ERROR*. Additionally, new literal prefixes can be parsed, such as '0b' for binary integer literals. *Example:*

```
int literal(const char* lit)
{
   /* parse lit */
}
const int binInt =
      PRECOMP<int>(literal("0b1001"));
```

The *literal* function parses a string and determines its base from the prefix. It may perform a validation (for example, in the case of a binary base, all digits shall be either 0 or 1). In the case of an invalid digit, an exception is thrown.

D) Compile time memory allocation

Memory arrangement and organization must take place during design-time for some

embedded systems, when there is no memory management unit or operating system. This involves a static pre-planned data organization, and therefore no memory fragmentation.

One technique of implementing this organization is using a structure and placing all the address-fixed data as fields of the structure, to finally place a unique instance of the structure in a known position. This requires the planning and maintenance of the structure during design and coding.

The PRECOMP solution is to 'allocate' the data space during precompilation time, and track all the allocations to become fixed-located during compile time. Moreover, if the PRECOMP allocation exceeds the (future) available run-time memory, an exception can be thrown alerting that it'll not fit in the memory.

As an example, let's suppose that there are 4 elements that need fixed-allocation, a runtime RAM size of 64 bytes, and that the data shall start at address 0x010. The 4 elements are: an array of 4 16-bit integers, an array of 20 characters, one 32-bit float, and a 12-bytes length structure:

```
int element1[4];
char element2[20];
float element3;
struct Type element4;
```

According to the technique described above, the four elements should be enclosed in a structure.

The proposed way of performing this in PRECOMP C++, could be defining a 'precomp_alloc' function and a global precomp instance tracking the latest pointer:

```
size_t lastPos = 0;
char* const initPos = (char*)0x10;
const size_t RAM_size = 64;
template <class T>
void* precomp_alloc(size_t size = 1)
{
   void* const ret = initPos + lastPos;
   lastPos += size*sizeof(T);
      if(ret > initPos+RAM_size)
         throw bad_alloc();
   return ret;
}

//define the 'At' macro:
#define At(instance, type, address) \
     type& instance = \
     *reinterpret_cast<type*>(address)

At(element1, int,
  PRECOMP<int*>(precomp_alloc<int>(4)));

At(element2, char,
  PRECOMP<char*>(precomp_alloc<char>(20)));

At(element3, float,
  PRECOMP<float*>(precomp_alloc<float>()));

At(element4, Type,
  PRECOMP<Type*>(precomp_alloc<Type>()));
```

The 'At' macro is a tool to place a variable in a given address, and it's defined here just to clarify the code.

A better C++-like syntax could be reached by overloading the 'new' operator, but such case is not exposed here due to sizing reasons.

E) others: FSMs, GUIs

This last case study is just mentioned but not deeply analyzed here, in order to consider the precompilation phase to instantiate Finite States Machines (FSMs), and Graphic User Interfaces (GUIs) from external editors.

An FSM can be described by the State Transition Table, which contains the information that given a stimulus, what transition function shall be invoked and the next state to transition to.

This information could be described in an external data file (i.e. generated from a tool) and then it can be read and parsed during precompilation in order to generate instantiation information for state-classes (as described in Gamma [5]).

Similarly, information regarding GUI controls can be provided in a separate file, which can be read during precompilation in order to instantiate GUI classes, instead of generating code with an external tool.

5. Qualitative Comparison

The TMP technique can be used to calculate values during compile-time; however, resources are limited to the capabilities of the compiler and only the Functional-programming paradigm is allowed. On the other hand, the preprocessor cannot use pointers, or dynamic memory or execute I/O stream operations. The preprocessor does not respect scope; therefore macros can accidentally and sometimes silently replace code. In practice, preprocessor metaprogramming is far simpler and more portable than template metaprogramming [2].

It turns out that the current ways of implementing GP have some weaknesses. A comparative table summarizes some of the features that are or aren't present in the other techniques mentioned before:

Tools / Features	Preprocessor	TMP	**PRECOMP C++**
Allows Reflection	NO	NO	NO(*)
Allows Debugging	NO	NO	YES
Allows Compile time assertions	YES	YES	YES
Readability of code?	Sometimes	Only in trivial cases	YES
Easy to use	YES	Only in trivial cases	YES
Is C++ syntactic compliant?	YES	YES	YES
Static / dynamic language symmetry	NO	NO	YES
Can emit friendly diagnostic messages	YES	NO	YES
Can use I/O streams	NO	NO	YES
Can use Pointers	NO	NO	YES
Can use Dynamic Memory	NO	NO	YES
Can compute non integral expressions	NO	NO	YES

(*) planned for future evolutions. Refer to the Future Work section.

In a two level language it is important to achieve *symmetry* between its static and dynamic aspects [9], that is, execute tasks at compile time or execution time without noting any difference. By taking Veldhuizen's view of C++ as a two-level language, this work homogenizes the dynamic and static levels into a seamless syntactical and functional unification. Another important thing this work provides is the notion of *compile time assertions*, which are assertions that are evaluated during the compilation process. This kind of assertions is useful when it is necessary to perform static checking to prevent errors during the execution of the application [1]. Also, the use of non-integral types is conflicting in both preprocessor and TMP, but is simple in precomp-C++. For example, given

```
#define PI 3.14
```

the preprocessor statement,

```
#if PI > 3
```

becomes a preprocessor error. Similarly, given a template metaprogram to calculate the cosine, templates cannot accept floating points as non-type template parameters; for example these statements are invalid:

```
Cos<1.25>::value
Cos<getValue()>::value
```

because the instantiation of templates with non integral values or with unknown values at compile time (as in the case of te getValue function call) is not allowed. In contrast, these features are available in precomp-C++: non-integral values and function calls can be used within each *PRECOMP-TAGs* and all of the

functions invoked exist in standard libraries such as *stdlib*, whereas in TMP we need to create these functions (as templates) to use them in template instantiation.

7. Future Work

This paper currently presents a mechanism to generate constant data during compilation time. Two evolutions will be addressed:

- types generation
- statements and flow control generation

A C++ template structure can be thought of a function that receives data types as parameters, and returns a data type. The 'types generation' evolution will address the ability of generating types as regular C++ templates do, as well as generating type in an object-oriented manner, that is, objects whose methods receive data types as parameters and return data types, to be evaluated in precompilation time. Some reflection features will be present, by both enhancing the *typeid* operator (imperative-like), and by pattern matching (functional-like).

Similarly, the statements and flow control generation evolution will provide the ability to consider statements as precompilation-time objects, and the ability to define functions that accept statement-objects as parameters, and return (transformed) statement-objects, as well as objects whose methods accept statement-objects as parameters and return statement-objects. Reflection will be available for statement-objects as well, following the analogy of a *statementid* operator returning a *statement_info* class.

8. Conclusions

This work exposes the benefits of adding the precompilation phase, over the current compilation model. Comparison between current techniques (such as TMP) and the precompilation-enabled C++ is provided, including constants calculus, early checking and data importing during precompilation time.

While TMP requires re-writing all the numeric libraries in a functional style (with limitations on precision and compiler abilities), PRECOMP C++ just invokes them as any regular C++ program does.

Finally, an implementation is provided in order to show that current compilers have all they need to implement the precompilation phase, since no new syntax is required, but just the ability to execute a generated binary using the system.

Both embedded systems and system programming can be benefited from the proposed enhanced compilation model.

9. References

[1] **Alexandrescu, Andrei,** "*Modern C++ Design: Generic Programming and Design Patterns Applied*". Addison Wesley, Reading, Massachusetts, 2001.

[2] **Boost libraries,** http://www.boost.org, Known Problems of the C/C++ Preprocessor.

[3] **Chiba, Shigeru**, "*A Metaobject Protocol for {C++}*", ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), SIGPLAN Notices 30(10), Austin, Texas, USA, pp 285-299, 1995.

[4] **Czarnecki, Krzysztof**, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen, "*Generative programming and active libraries (extended abstract)*". In *Generic Programming. Proceedings* (M. Jazayeri, D. Musser, and R. Loos, eds.), pp. 25-39. Volume 1766 of Lecture Notes in Computer Science. Springer-Verlag. 2000.

[5] **Gamma, Erich**, R. Halm, R. Johnson, J.Vlissides, "*Design Patterns: elements of reusable object-oriented software*", Addison Wesley, 1995.

[6] **GNU MP Bignum Library**, http://www.swox.com/gmp/

[7] **INCITS ISO IEC 14882-1998** International Standard, Programming Languages - C++, 8.3.4 § 1

[8] **Stroustrup, Bjarne**, "*The C++ programming language Special Edition*", Addison Wesley Publishing Co., Reading, Mass., 2000, pp. 160-161.

[9] **Vandevoorde, Daveed**, "*Reflective Metaprogramming in C++*", N1471/03-0054, JTC1.22.32 Programming Language C++ Evolution Working Group, ISO/IEC IS 14882:2003(E), 2003.

[10] **Veldhuizen, Todd**, "*C++ templates as partial evaluation*", ACMSIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'98), ACM Press, San Antonio, TX, USA, 1999.

[11] **Veldhuizen, Todd**, "*Five Compilation Models for C++ Templates*", First Workshop on {C++} Template Programming, Erfurt, Germany, 2000.

[12] **E. D. Willink,** V. B. Muchnick**.** "*An Object-Oriented Preprocessor Fit for C++*", IEEE Proc. on Software, 147(2), 2000.