University of Nevada
Reno

CHIMP
The C/C++ Hybrid Imperative Meta-Programmer

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science

by

John L. Kenyon

Dr. Frederick C. Harris, Jr., Thesis Advisor

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

JOHN L. KENYON

entitled

Chimp
The C/C++ Hybrid Imperative
Meta-Programmer

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C Harris, Jr., Ph.D., Advisor

Sergiu Dascalu, Ph.D., Committee Member

Karen Schlauch, Ph.D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

May, 2008

Abstract

The C and C++ languages have been unchanged in over two decades, and during this time many shortcomings of the languages have become clear. Specifically, neither language supports reflection, and the meta-programming capabilities are very limited. Both of these problems can be solved by adding a pre-processing step, which can analyze and modify the code before the actual compiler translates it into object code. This means we can use metaprogramming and reflection to simplify some C/C++ tasks without having to change the languages at all. In order to investigate this, we have created CHIMP, a meta-programming tool that demonstrates the concept and can be used to explore the full capacity of the proposed programming technique.

Acknowledgments

As of the publication of this document, I have been a graduate student working on an MS longer than I was an undergrad. That means it is high time I got this out of the way. So here it is, my great idea.

I would like to thank my advisor, Dr. Frederick C. Harris, Jr. for all of his help keeping me on track during these last 7 years of my academic life, encouraging me to go beyond the track of a typical student, and tolerating my various crack-pot ideas (see "John's Cool Things", Fall 2006). I would also like to thank my other committee members, Dr. Sergiu Dascalu and Dr. Karen Schlauch, for taking time out of their busy schedules.

I would like to thank my parents, James Kenyon and Marty Baring, for their encouragement and understanding during my extended stay as a grad student, and acceptance of my various distractions and side projects (I promise I will give back your garage eventually).

I would also like to thank all of my friends and collegues from the ECSL lab, for their input and criticisms (contructive and otherwise), with a special thanks to David Carr for setting me on the right design path.

I would also like to thank my friend Dan Farmer, for being my chief editor and proof reader; without him this document would be nearly illegible.

I would also like to acknowledge the GNU foundation for GCC-XML, the Python Software Foundation and Guido Van Rossum for providing the excellent Python programming language, and the Pocoo project which developed the Jinja template engine. Thanks to all of these components, my thesis became a breeze.

January 2008

Contents

Abstract **i**

Acknowledgements **ii**

Table of Contents **iii**

List of Figures **v**

1 Introduction **1**

2 Literature and Concept Review **3**
2.1 Meta-Programming 3
2.1.1 Basic Idea of Meta-Programming 3
2.1.2 Template Meta-Programming 4
2.1.3 Template meta-programming is a bad idea 4
2.2 Reflection and Inspection 6
2.2.1 Inspective C++ 6
2.2.2 OpenC++ 7
2.2.3 Messing with C++ and departing from GCC are bad ideas . . 8
2.3 CodeSmith Tools 9

3 Imperative Meta-Programming and CHIMP **10**
3.1 Imperative Meta-Programming 10
3.1.1 The Application Phase 11
3.1.2 The Analysis Phase 13
3.2 Implementation of CHIMP 16

4 Results **21**
4.1 CHIMP Usage Guide 21
4.1.1 Command line usage 21
4.1.2 MetaProgramming Tags 22
4.2 Applications of CHIMP 24
4.2.1 XML 24
4.3 Limited Runtime Reflection 31

4.4 Role Based Meta-Programming 36
4.5 Failings of CHIMP 37
4.5.1 Gotchas: debugging 38
4.5.2 Gotchas: vanishing prototypes 38
4.5.3 Gotchas: vanishing statements 40

5 Conclusions and Future Work **41**
5.1 Conclusions 41
5.1.1 Gray Areas 41
5.2 Future Work 42
5.2.1 CHIMP Cleanup 42
5.2.2 CHIMP Variations 43
5.2.3 Inspired ideas 43

A Chimp Source Listing **46**
A.1 Chimp.py 46
A.2 Analysis.py 48
A.3 Template.py 52

B Applications of Chimp **54**
B.1 LUA 54
B.1.1 toLua 54
B.1.2 fromLua 56

Bibliography **60**

List of Figures

2.1 C++ Template meta-programming for Fibonacci 5

3.1 Added stage in the IMP compile process 11
3.2 HTML with PHP 12
3.3 Simple C++ loop 12
3.4 C++ Metacode to unroll a loop 13
3.5 C++ code generated by MetaCode Loop Unroller 13
3.6 C++ Code for DumpToScreen 14
3.7 PseudoCode for DumpToScreen 15
3.8 C++ MetaCode for DumpToScreen, written for CHIMP 16
3.9 DumpToScreen generated from MetaCode 17
3.10 Invalid C++ 18
3.11 Sample C++ for GCC-XML 19
3.12 Sample GCC-XML Output 19

4.1 Building a CHIMP project with Make 21
4.2 Building a CHIMP project with SCons 22
4.3 Jinja For loop 23
4.4 Jinja If statement 23
4.5 Jinja assignment 23
4.6 Jinja Macros 23
4.7 Jinja Filters 24
4.8 Sample CHIMP Makefile 24
4.9 xml example 01.mcpp 26
4.10 toxml.cmf 27
4.11 xml example 02.mcpp 28
4.12 toxml.cmf part1 29
4.13 toxml.cmf part2 30
4.14 Incompatible returns 31
4.15 Type Conversion with SStream 32
4.16 Simple reflect code 32
4.17 Reflect part1 33

4.18 Reflect part2 34
4.19 Reflect part3 35
4.20 Primary key role 37
4.21 Bad Metacode... 39
4.22 Becomes bad C++ code 39
4.23 Better Metacode... 39
4.24 Erroneous code vanishes 39
4.25 Correct code is generated 40

5.1 Role Based Programming Pseudo Example 45

Chapter 1

Introduction

One of the goals of a programmer is to produce as much functionality as possible in the limited time frame of the working day. For this reason, there is a perpetual arms race to develop new programming languages, development environments, and platforms, each designed to make the task of programming easier, faster, more reliable, and less repetitive. However, for various practical and political reasons, C and C++ remain very popular, despite their age and lack of modern features. Various attempts have been made to update the language, such as buiding new languages based on C/C++, extending the language through use of non-standard compilers, or using the existing templating system in C++. This thesis will offer an alternative approach dubbed "Imperative Meta-Programming."

Imperative Meta-Programming is a two step technique for meta-programming. First we inspect the contents of the target source code to get information about variables, types, and most importantly members and methods of classes and structs. Then we feed this information as a parameter into a template engine (Like PHP or ASP, not like the C++ template system!). The template engine will allow us to automatically generate large amounts of code based on the structure and content of existing structures, and it will allow us to do so without having to change the C/C++ compilers and languages definitions.

The remainder of this document will be as follows. Chapter 2 will review the basic ideas of meta-programming and reflection, and review the literature and related work. Chapter 3 will cover the technique of Imperative Meta-Programming

and the implementation of "CHIMP," the initial proof of concept imperative metaprogrammer. Chapter 4 will cover some actual applications of the CHIMP program to C++ metacode, including a short guide to using it, and a quick overview of some pitfalls of using it too. Chapter 5 will review the gains provided by imperative metaprogramming as a programming technique and will discuss future work. Appendix A will include the complete source code for CHIMP. Appendix B will include additional applications of CHIMP.

Chapter 2

Literature and Concept Review

2.1 Meta-Programming

2.1.1 Basic Idea of Meta-Programming

Typical programming is the act of writing code which will be translated into an executable program. Meta-Programming is the "art" of writing code that will generate *new code*, which will in turn be translated into an executable program. Meta-Programming is an old idea, and is already used in C through the PreProcessor [14] and in C++ through template meta-programming [6]. Programs like Yacc [13] and Lex [8] are also meta-programmers, as they take an input language and generate C code as an output.

The motivation behind meta-programming is to relieve the programmer of having to write repetitive or complex code, where a simple description in a meta-language can be expanded to a large and complex block of code in a target language. A simple example of this comes from the basic usage of C++ templates. Because of C++'s strict typing, if one writes a container class, it will only be able to contain the type that it was written for. If you want a container for another data type, you would effectively need to copy and paste the whole container class, but then change every instance of the data type. This is a pointless waste of programmer's time, and is simplified greatly by the C++ template system [14].

2.1.2 Template Meta-Programming

Template Meta-Programming has become a bit of a hot topic lately. Attempts to do a background search in the field of C++ meta-programming were rather difficult because any search for C++ meta-programming inevitably found "Template meta-programming" which is irrelevant to this project. However, one cannot address meta-programming in C++ without giving a nod to template meta-programming, so I will give a brief demo of it here.

Let's say one wants to calculate Fibonacci numbers at compile time. We can see an example in Figure 2.1, where we use template meta-programming in C++ to generate Fibonacci numbers at compile time, by using the C++ Template system. This is a reasonably approachable example, and it demonstrates the declarative (e.g., Prolog) nature of C++ Templates.

2.1.3 Template meta-programming is a bad idea

However, Template meta-programming is a bad idea for several reasons: it is hard to write, hard to read and hard to understand, and does not allow the full range of meta-programming techniques.

The first issue with Template meta-programming stems from the fact that it is exceptionally hard. Most programmers learn to program with imperative languages, like C or Python. However, template meta-programming is predominantly a declarative programming model, which is counter-intuitive to most programmers. This is evident by the general obscurity of Prolog in our modern language market. This declarative approach leaves many programmers guessing as to how it will actually behave. With sufficient training and experience, it can become obvious and intuitive. However, most students in the field of Computer Science are given an emphasis on imperative, object-oriented, and functional programming. Declarative programming tends to get an "honorary mention" in a progamming lanuages course.

The C++ template system was simply not designed for this kind of use, and as such the seam between template code and regular code is very messy. While the

```
 1      #include <iostream>
 2      using namespace std;
 3
 4      template <int N>
 5      struct fibonacci
 6      {
 7          enum {val = fibonacci<N-1>::val + fibonacci<N-2>::val};
 8      };
 9
10      template <>
11      struct <0>fibonacci
12      {
13          enum {val = 1};
14      };
15
16      template <>
17      struct <1>fibonacci
18      {
19          enum {val = 1};
20      };
21
22
23      int main()
24      {
25          cout << "Fibonacci number 5 is : "
26                  << fibonacci<5>::val << endl;
27          cout << "Fibonacci number 6 is : "
28                  << fibonacci<6>::val << endl;
29          cout << "Fibonacci number 7 is : "
30                  << fibonacci<7>::val << endl;
31          return 0;
32      }
```

Figure 2.1: C++ Template meta-programming for Fibonacci

declarative nature of C++ templates does work well for its purpose, this is overshadowed by the fact that one must use a lot of C++ features in a really strange way in order to trick the compiler into doing what it is that you really want. In the Fibonacci example above you can see that we must use enumerations to hold intermediate values, which is ironic since the enumeraton was designed for holding constant values instead of variables. Also, one notices that intermediate rules are

nested inside of structs. If we could apply the template system to a function and still get the Fibonacci result that might make more sense, but the only way to propagate the values is through a "constant" value defined within a struct.

As a final note, template meta-programming does not directly support reflection, which will be explained in the following section. Some attempts have been made to add reflection support to C++ template meta-programming. Most of these will be discussed in the following section. Many of them have the flaw that they require the user to explicitly list class members a second time, in a preprocessor macro. This kind of redundancy in the class definition is undesirable, since it offers an opportunity for inconsistent definitions which will cause errors to occur. Ideally, there should be exactly one definition of each thing.

2.2 Reflection and Inspection

Reflection is the common name for an object's awareness of its own members and methods. This is a powerful feature, since it offers the programmer a lot of options for dynamic programming techniques. A prime example of reflection in action is the pickle module in Python. In the Python programming language, any object can be serialized to a string or a file, regardless of its contents or inheritance tree. This is done without requiring the user to intervene and define serialization operations. Instead, Python looks at the contents of the objects and does a recursive traversal of all members, dumping each to a string or stream.

Most modern languages offer this capability, such as Java's java.lang.reflect package, Python's "dir" function. However, C++ does not offer any such mechanism for getting this information at run time or at compile time. Several projects and groups have attempted to add this feature.

2.2.1 Inspective C++

Inspective C++ is a C++ compiler, based on the GCC source, which seeks to add compile time reflection to the C++ language [12]. It does this by exposing reflective

information to the C++ template system. This approach offers several advantages and disadvantages over Imperative Meta-Programming.

Advantages of Inspective C++

1. Does not require extensive modification of the C++ language.
2. Does not require a two step compile process.

Disadvantages of Inspective C++

1. Uses C++ Template meta-programming techniques, which are complicated.
2. Requires maintenance of a separate branch of the GCC project.
3. Requires users to learn a new method of using C++ Templates.

2.2.2 OpenC++

OpenC++ is a project that aims to add reflection and meta-programming to C++ as well. It does so in an interesting manner by adding metaclass objects to C++. Metaclasses generate C++ syntax tree objects from snippets of code, which can then be added to the final C++ code. These classes are evaluated before the actual C++ compiler, and modifies the source tree. This is a useful idea, which is very similar to several early thought experiments that led to the Imperative Meta-Programming idea. While this approach is exceptionally powerful, it can also be very difficult and confusing. This means the programmer is interacting with the source indirectly, and through an unusual interface.

Advantages of Open C++

1. Allows more complex modifications to code, and is location independent with respect to the source code.
2. Does not require a two step compile process.

Disadvantages of Open C++

1. Changes the C++ language.

2. Modification to the program at the parse tree level can be hard to understand.

3. Requires maintenance of a separate compiler.

2.2.3 Messing with C++ and departing from GCC are bad ideas

Departing from C++ is a bad idea

C++ has now been around for over 25 years[14], and as such has 25 years worth of code already written in it. Attempts to modify the C++ language would inevitably break some of that code, which is a deterrent. Additionally, there are many developers who already know C++, and modifying the language would break their understanding of it.

This means that any attempt to start a new language is a huge gamble and requires a lot of resources to complete and maintain, and after all that it may never become popular. But subtle modifications to a language are also inadvisable unless the language is still under active development.

Departing from or modifying GCC is a bad idea

An unfortunately large number of Open Source projects fail shortly after they are started. One of the primary causes of this is that developers eventually leave the project with no new members to replace them. The practical ramifications of this are that one should be humble when starting a project, a project that can be developed in a matter of weeks and then be considered "done" has a much higher probability of success than one that will require a massive team of developers dedicated to the project for years, and then require maintenance for the lifetime of the language's usage. Additionally, when a project has a dependency on external libraries, it is wise to choose only projects that have a long history of success.

Given that developing and maintaining a compiler is a very complex task, it is best to leave that to groups like GCC and Microsoft. One can say with confidence that GCC will not be going anywhere anytime soon, but the prospects for many of the smaller startup languages are not so sure.

From these two points we can conclude that odds of success can be maximized by using a known language and compiler pair *as is*.

2.3 CodeSmith Tools

Recently a third party package for Visual Studio was released, called CodeSmith. CodeSmith is the nearest program we know of to the Imperative Meta-Programming method presented in this document. CodeSmith is described as a software development tool for code generation [1]. CodeSmith uses an ASP.NET style template system to automatically generate C# or Visual Basic .Net code, based on the contents of SQL Databases. This allows developers who are interfacing with databases, especially web developers, to rapidly get all of the basic operations up and running very quickly.

The idea of using a templating engine designed for generating webpages is one of the major components of the IMP concept, and so credit must be given to CodeSmith for doing this first. However, their input data set seems to be entirely centered around your SQLServer databases, providing no inspective information about the actual classes. This makes sense, since C# already provides reflection through the *System.Reflection* module, and so there is no need to add reflection to the language.

Chapter 3

Imperative Meta-Programming and CHIMP

3.1 Imperative Meta-Programming

The Imperative Meta-Programming (IMP) technique uses a powerful preprocessor for the target language, in this case: C++. The technique starts with a MetaCode file, which is primarily composed of the target language, with a few Metacode elements added in. These metacode elements will be processed and evaluated or removed, resulting in a new file completely composed of the target language. From here the compilation process can procede normally. Figure 3.1 shows this two stage process. For the remainder of the document, we will use C++ as the target language.

The IMP preprocessor has two stages: analyze and application, and as such modifies the compilation process as seen in Figure 3.1. The analysis phase collects information about the code, while ignoring all metacode. In this phase, the analyzer collects any relevant information, such as all functions, classes, structs, and global variables, as well as their respective members, methods and parameters. The information gathered from the analysis phase is then provided to the metacode engine during the application phase. The application phase then executes the metacode engine, generating new C++ code based on the information provided. I will explain each phase in detail, first the application phase (since it can be explained as a stand alone component), then the analysis phase.

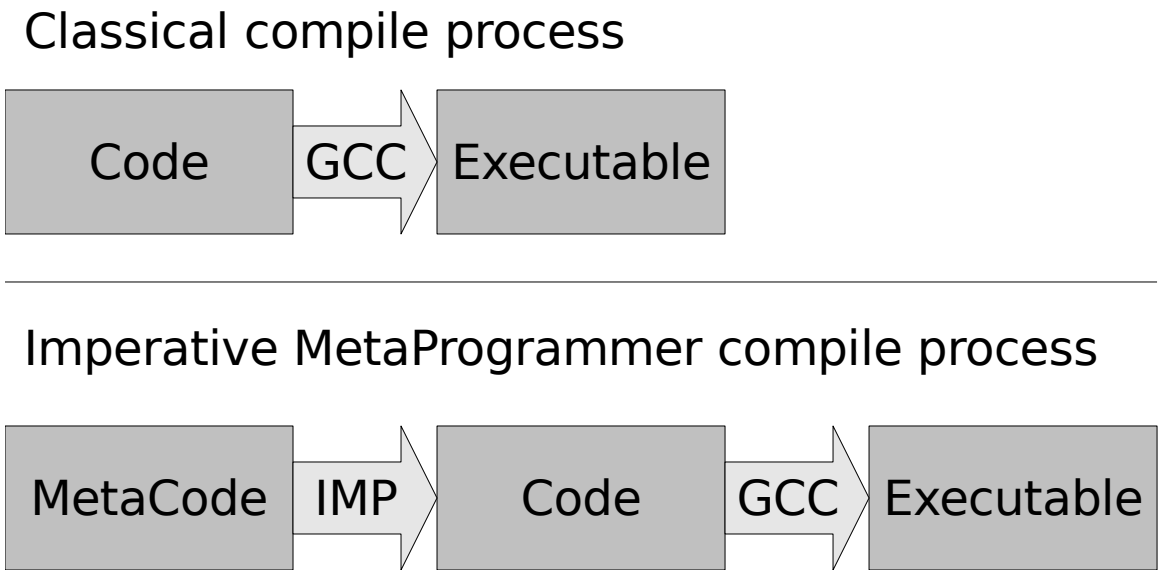

Figure 3.1: Added stage in the IMP compile process

3.1.1 The Application Phase

The application phase is very simple and can be used without the code analysis. For this one only needs a template engine, much like the ones used for web development: PHP, ASP, ERB, etc. Just like web development, we are going to have a lot of static content (HTML in web development, C++ in IMP), with small bits of dynamic content that is generated by the template language. Figure 3.2 shows these similarities by comparing PHP code embeded in HTML, along with some equivalent IMP code embedded in C++.

These two side by side examples should clarify the basic idea, we can embed logic directly in C++ code, without interfering with C++ syntax. This allows us to perform some very complex actions on the code, however it can be somewhat tricky. The programmer must now be very cautious, since the code generated must be valid C++ code. HTML is much more forgiving of small mistakes than GCC is.

A slightly more complex and practical example would be to perform loop unrolling. Loop unrolling is an optimization that is sometimes done by compilers [3]. It is often performed on very large bounded loops with very simple actions, where

```
1 <html>
2 <head><title></title></head>
3 <body>
4 <?php print "Hello web!"; ?>
5 </body>
6 </html>
---------------------------------
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    cout << "{% print "Hello C++" %}";
7    cout << endl;
8  };
```

Figure 3.2: HTML with PHP

the overhead of the looping operation becomes costly compared to the actual computations done. In the following examples I will show a typical C++ loop, then I will show the metacode to explicitly unroll that loop, and show the resulting unrolled loop.

Observe Figure 3.3. It is a simple summation of all members in a simple array, using a standard C++ for loop. In Figure 3.4, we will use an Imperative Meta-Programmer to write a loop that will be explicitly unrolled. In this case I am using CHIMP, which will be explained in more detail in the following section.

```
1      int i;
2      float array[10] = {1.0f,2.0f,3.0f,4.0f, \
3          5.0f,6.0f,7.0f,8.0f,9.0f,10.f};
4      float sum = 0.0f;
5      for(i = 0; i < 10; i++) {
6          sum += array[i];
```

Figure 3.3: Simple C++ loop

Finally, in Figure 3.5, we have the output generated by the metacode generated in Figure 3.4. This may seem a bit excessive, but loop unrolling is a serious problem for the compiler to manage. It is not a standard part of the C/C++ languages, and

```
1       int i;
2       float array[10] = {1.0f,2.0f,3.0f,4.0f,
3           5.0f,6.0f,7.0f,8.0f,9.0f,10.f};
4       float sum = 0.0f;
5       {% for x in range(10) -%}
6       sum += array[{@ x @}];
7       {% endfor %}
```

Figure 3.4: C++ Metacode to unroll a loop

also is a very delicate operation. In order for a loop to be eligible to be unrolled by a compiler, it must be able to verify that the index and sentinel values can not possibly be changed during the loop, which can be very tricky. In this case we can explicitly see the results once we are ready, but as programmers we were still able to use a loop construct rather than type the code manually.

```
1       int i;
2       float array[10] = {1.0f,2.0f,3.0f,4.0f, \
3           5.0f,6.0f,7.0f,8.0f,9.0f,10.f};
4       float sum = 0.0f;
5       sum += array[0];
6       sum += array[1];
7       sum += array[2];
8       sum += array[3];
9       sum += array[4];
10      sum += array[5];
11      sum += array[6];
12      sum += array[7];
13      sum += array[8];
14      sum += array[9];
```

Figure 3.5: C++ code generated by MetaCode Loop Unroller

3.1.2 The Analysis Phase

While the ability to generate code procedurally does help in some cases, we are still limited by the naive nature of the application phase. We can augment it by giving it more information about the code itself. Lets look at some simple repetitive tasks that are sensitive to existing code. Lets look at a simple function for debugging,

"dumpToScreen." We will create a struct and then write a C++ function that dumps its entire contents to stdout.

The DumpToScreen function example in Figure 3.6 is a perfect example of mindless repetitive code. It is boring for programmers to write, it is often written for every important struct/class in a program, and it is sensitive to changes in the original struct. If one were now to add a new member to ExampleStruct, like `int woot;`, the programmer would now need to go down to the DumpToScreen function for this Struct, and add the line `cout << " woot :  " << ptr->woot << endl;`. This is a tedious process, and it is an error prone one. If the struct were much larger, with many nested types, and there were many functions to dump the object to file in XML, or in YAML, or in JSON or in raw text, then one would need to go through all of these functions and fix them. This is also problematic from a human resources problem, because it may be too complex a job to trust to an intern, but its so mundane that the senior software engineer might not want to deal with it.

```
 1      struct ExampleStruct{
 2          int a;
 3          float b;
 4          long double c;
 5      };
 6
 7      void DumpToScreen(ExampleStruct *ptr)
 8      {
 9          cout << "ExampleStruct :" << endl;
10          cout << "    a : " << ptr->a << endl;
11          cout << "    b : " << ptr->b << endl;
12          cout << "    c : " << ptr->c << endl;
13          cout << endl;
14      }
```

Figure 3.6: C++ Code for DumpToScreen

What we really want to do, is be able to generate the DumpToScreen function dynamically, based on the contents of the ExampleStruct definition. A pseudocode example would look something like Figure 3.7. In that case, we assume we can iterate over the members of the target struct.

```
 1      struct ExampleStruct{
 2          int a;
 3          float b;
 4          long double c;
 5      };
 6
 7      void DumpToScreen(ExampleStruct *ptr)
 8      {
 9          PRINT name of ExampleStruct
10          FOR each member of ExampleStruct
11              PRINT current member name : current member value
12      }
```

Figure 3.7: PseudoCode for DumpToScreen

C++ does not support any form of reflection, and so cannot tell what members and methods an object might have. However, if we have a C++ parser that can look at the code first, and then provide this information to our template engine, then our Imperative Meta-Programmer would be able to generate code equivalent to that in Figure 3.7.

In Figures 3.8 and 3.9 we address this problem with a complete C++ program. Figure 3.8 is written in CHIMP MetaCode, and takes advantage of information gathered in the analysis phase to generate code based on the contents of ExampleStruct. This code is sensitive to the struct, one could add as many members or change the types of any members, and the source code would be updated appropriately after the next compile. This is very powerful because the DumpToScreen function is now immune to human negligence, and will match the struct definition instead of the users code.

In particular, we have removed the need for redundant information about ExampleStruct. When logic like this is hard-coded it effectively amounts to a second copy of the ExampleStruct definition. However, redundant definitions of data objects lend themselves to falling out of sync, which almost always causes problems. This problem is circumvented by removing the redundant information that would have been stored in the hardcoded copy of the DumpToScreen function.

```
 1      #include <iostream>
 2      using namespace std;
 3
 4      {% macro MetaDumpToScreen obj -%}
 5      void DumpToScreen( {@ obj.name @} *ptr)
 6      {
 7          cout << "{@obj.name@}" << endl;
 8          {% for name in obj.members -%}
 9          cout << "    {@ name @} : " << ptr->{@ name @} << endl;
10          {% endfor %}
11
12      }
13      {% endmacro -%}
14
15      struct ExampleStruct{
16          int a;
17          float b;
18          long double c;
19      };
20
21      void DumpToScreen(ExampleStruct *ptr);
22      {@ MetaDumpToScreen(ast.structs['ExampleStruct']) @}
23
24      int main()
25      {
26          ExampleStruct e;
27          DumpToScreen(&e);
28          return 0;
29      }
```

Figure 3.8: C++ MetaCode for DumpToScreen, written for CHIMP

Further applications will be shown in Chapter 4.

3.2 Implementation of CHIMP

CHIMP was developed as a proof of concept for Imperative Meta-Programming. It is a combination of the Jinja Template engine [11], the GCC-XML frontend to GCC [7] and the Element Tree library for XML [5], glued together with Python [5]. Jinja is a template engine that was developed for the Pocoo project, and is maintained by Armin Ronacher. Jinja is very similar to the Django template language [11]; however,

```
 1      #include <iostream>
 2      using namespace std;
 3
 4      struct ExampleStruct{
 5          int a;
 6          float b;
 7          long double c;
 8      };
 9
10      void DumpToScreen(ExampleStruct *ptr);
11      void DumpToScreen( ExampleStruct *ptr)
12      {
13          cout << "ExampleStruct" << endl;
14          cout << "    a : " << ptr->a << endl;
15          cout << "    c : " << ptr->c << endl;
16          cout << "    b : " << ptr->b << endl;
17
18      }
19
20      int main()
21      {
22          ExampleStruct e;
23          DumpToScreen(&e);
24          return 0;
25      }
```

Figure 3.9: DumpToScreen generated from MetaCode

it is more flexible and easier to modify, as was necessary for CHIMP. GCC-XML is is a C/C++ parser which outputs all information about a program in an XML format, so it can easily be parsed [7].

The first step is the analysis phase, which is mostly handled by GCC-XML. GCC-XML will parse a C++ file and read all information about the top level objects into an XML file. These object are: types, structs, classes, members, methods, functions, parameters, and global variables. Source code is not included in the XML output, but is parsed, and as such the entire file must be valid C++ code. This introduces a bit of a complication, since the metacode is not valid C++, and so will cause GCC-XML to fail. This can be addressed by removing all MetaCode in advance as we see in

Figure 3.10.

```
 1      struct foo {
 2          int a;
 3      };
 4
 5      // This line is not valid C++ with the metacode in place
 6      {% dosomething(ast.structs['foo']) -%}
 7
 8      int main(){
 9          return 0;
10      }
```

Figure 3.10: Invalid C++

The code in Figure 3.10 cannot be parsed, since `{% dosomething(...) %}` is not valid C++. If we could remove all metacode statements, then the code would be left with only valid C++ code. This can be done by making a copy of the metacode file, and feeding it into a modified version of the Jinja interpreter. This modified interpreter simply ignores all Jinja directives and blocks, instead of printing them. This way, only the plain C++ content is printed.

This step can have some complicated side effects. Since this "stripped" version of the code is mostly used to find data type definitions, we don't care about blocks of source code disappearing (that source code still exists in the original file). However, we may remove some function prototypes, which would have been generated by CHIMP. This can cause some problems, and will be discussed in Chapter 4.

Now that the MetaCode has been stripped from the code file, we have only valid C/C++ code, like that in Figure 3.11. Now we feed this into the GCC-XML program, which will parse the C/C++ code for us. The result is a fairly large (and mostly flat) XML file. Figure 3.11 shows both a small block of code and Figure 3.12 shows part of the XML that is generated by GCC-XML. Looking at this XML we can see that all different types are represented clearly, and everything in the file has an ID number. We use this ID number to cross-reference classes and their members. Types are also listed with an ID number, and must be cross-referenced the same way.

```
1       class MyClassType {
2       public:
3           MyClassType();
4           void action();
5       private:
6           int a;
7           float b;
8           char *c;
9       };
```

Figure 3.11: Sample C++ for GCC-XML

```
1       <Class id="_3" name="MyClassType" context="_1"
2           mangled="11MyClassType" location="f0:1" file="f0"
3           line="1" artificial="1" size="96" align="32"
4           members="_114 _115 _116 _117 _118 _119 " bases=""/>
5       ...
6       <Field id="_114" name="a" type="_126" offset="0"
7           context="_3" access="private"
8           mangled="_ZN11MyClassType1aE"
9           location="f0:6" file="f0" line="6"/>
10      <Field id="_115" name="b" type="_125" offset="32"
11          context="_3" access="private"
12          mangled="_ZN11MyClassType1bE"
13          location="f0:7" file="f0" line="7"/>
14      <Field id="_116" name="c" type="_136" offset="64"
15          context="_3" access="private"
16          mangled="_ZN11MyClassType1cE" location="f0:8" file="f0"
17          line="8"/>
18      <Method id="_119" name="action" returns="_131" context="_3"
19          access="public" mangled="_ZN11MyClassType6actionEv"
20          location="f0:4" file="f0" line="4" extern="1"/>
```

Figure 3.12: Sample GCC-XML Output

All of the information in this can easily be collected with a typical XML parser. For CHIMP the ElementTree package was used, because it has a nice implementation of XPath. XPath is a command language which allows the user to traverse and search an XML tree like a directory tree [2]. So a search might look like `/root/foo`, which means "find the node called foo which is a child of the top level node called root." Alternatively, one could search for `//foo` and the XPath processor would search the

XML document for any tags of type foo.

So the analyzer has to perform a two stage process. First it gathers all items in the file so they can be looked up later when we are finding members types. The XPath command for this is simply `/*`. Second, it gathers the four important top level things in C++: global variables, functions, classes, and structs. Unions were ignored for this iteration of the project. These four are found by the following XPath commands: `//Variable`, `//Function`, `//Class`, and `//Struct`. Once collected, we can look at the "members" attribute of the Class and Struct objects, and look all members up based on the lookup table we created previously. Finally, we look up the types of each member, variable or function return value. All this information is put into a Python object which has a more user friendly interface, and is made accessible to the Jinja template engine.

This brings us to the template engine. There are many template engines available for Python, such as Kid [15] or Cheetah [10]. Cheetah was strongly considered, since it claims that in addition to being used as a web template engine, it is also being used to generate "C++ game code [10]." A version of CHIMP that uses Cheetah may end up being future work. After a bit of research, the Jinja template engine was selected.

Jinja is a template engine written in Python for the Pocoo project [11]. However, Jinja is nicely decoupled from the project, and so lent itself to use in CHIMP. Jinja started as a clone of the Django template engine, and so has similar default delimiters and syntax. Another perk of the Jinja project is that it is very easy to override the default intermediate code generation step, by simply inheriting from the PythonTranslator object, and specifying your own new class in its place at run time. Thus it is easy to write a program to strip Jinja commands from the file without having to modify any of the Jinja code base. The ability to write the strip function is pivotal to its use in the IMP paradigm.

Examples of Jinja will be covered in much more detail in Chapter 4.

Chapter 4

Results

4.1 CHIMP Usage Guide

4.1.1 Command line usage

Usage of CHIMP is fairly straightforward. The last command line parameter is used as the input file name. Output goes to standard out unless specified with an optional -o flag. If the GCC-XML stage requires extra include flags, then these can be specified with the -I flag.

```
CHIMP [-o <output file>] [-I "<GCC Flags for GCC-XML>"] <input file>
```

Make files

CHIMP was designed to be easy to use within the scope of a normal project build. As such it can easily be invoked from within GNU/Make as shown in Figure 4.1 or other make systems such as SCons [4], as seen in Figure 4.2.

```
main: main.o
    g++ -o main main.o

main.o: main.cpp
    g++ -c main.cpp

main.cpp: main.mcpp
    chimp -o main.cpp main.mcpp
```

Figure 4.1: Building a CHIMP project with Make

```
chimp_bld = Builder(action = 'chimp -o $TARGET $SOURCE',
                    suffix = '.cpp', src_suffix = '.mcpp')
env = Environment(BUILDERS = {'Chimp':chimp_bld})
env.Chimp('main.mcpp')
env.Program('main.cpp')
```

Figure 4.2: Building a CHIMP project with SCons

4.1.2 MetaProgramming Tags

As previously mentioned, the Jinja template engine is used to do the meta-programming operations. The meta-programming tags must be delimited by using tags like `{% %}` for statements and `{@ @}` for looking up values and simple execution of macros. These tags can be customized within the Jinja engine, so it was tempting to use ASP/ERB style tags with `<% %>` and `<%= %>`, however `<% %>` are valid C++ tags, and are aliases for `{ }` [14]. For this reason, I kept the default. The variable lookup was switched to the `{@ @}` tags because its possible that `{{ }}` could be used in valid C++. It is quite possible that there are better tags that could be used, however it is imperative that they not conflict with the C++ language in any way.

Within the `{% %}` tags, we have a very Python-esque language. Unlike Python, we must have an ending delimiter to all of our control strucures, so a *for* tag must have an *endfor* tag as well. The format for the *for* loop can be found in Figure 4.3. *If* statments also behave like a normal Python *if* statement, except that they require an *endif* to explicitly end the *if* block. An example *if* statement can be found in Figure 4.4. Variable assignment is a little bit different than it is in Python, as it requires a *set* directive before it, as seen in Figure 4.5. [11]

Normal function calls are missing from Jinja, and instead we have macros and filters. Macros are the more useful of the two, as they are used for creating blocks of content. So if one wants to auto-generate a C++ function, that will most likely be generated by a Jinja macro. The code to create and then invoke a macro is shown in Figure 4.6. Filters on the other hand operate on data that is still in the Python/Jinja context. They are invoked with the `|` (pipe) operator, and behave like the pipe in

bash. Filters are more complicated, in both implementation and usage. The value on the left is fed into the filter function on the right as parameter. Filters are pure Python functions that are embedded in the Jinja engine. As such, it is hard to add new filters. Because of this, and because they are not as vital to CHIMP, only two filters will be explained. In the example in Figure 4.7 we can see the length filter being used. The length filter takes a list as a parameter and returns the length of that list. The other filter that is useful is "role_filter," which is part of CHIMP, and is explained in Section 4.4.

```
{% for <var> in <iterable expression> %}
   [loop body]
{% endfor %}
```

Figure 4.3: Jinja For loop

```
{% if <logical expression> %}
   [body]
{% elif <logical expression> %}
   [body]
{% else %}
   [body]
{% endif %}
```

Figure 4.4: Jinja If statement

```
{% set variable = 5 %}
{% set <varname> = <expression %}
```

Figure 4.5: Jinja assignment

```
{% macro my_macro param %}
  [Macro body]
{% endmacro %}

{@ my_macro(value) @}
```

Figure 4.6: Jinja Macros

```
{% set lst = [1,2,3] %}
{@ lst | length @}
```

Figure 4.7: Jinja Filters

4.2 Applications of CHIMP

4.2.1 XML

One of the more tedious tasks in programming is serializing and deserializing complex objects. Most modern languages support very clever serialization techniques, such as Python's Pickling library, as well as C#'s SerializeXml module. Unfortunately, C++ has no such library, so programmers must constantly re-write the tedious code to read and write XML based on the structure of their classes. In the following two examples, CHIMP will be used to automate this process for simple classes.

For this section the libxml++ library, created by the Gnome group [9], is used. It provides a convenient object-oriented interface to parsing and creating XML files, and provides XPath capabilities, which will be used to parse the files. This example is a limited proof of concept, and as such will only handle primitive types (int, float, etc.), STL strings, and member objects. For simplicity it will NOT handle arrays, pointers, STL lists/vectors, or STL maps. These types may be addressed in a later release.

As a last note before we get to the example, when we compile programs with libxml++, we need to specify an include path, with the -I flag. In order to get CHIMP to use these flags during the GCC-XML phase, we need to use CHIMP's -I flag as well. An example makefile can be seen in Figure 4.8.

```
 INC=-I/usr/include/libxml++-2.6/ -I/usr/include/glibmm-2.4/ \
     -I/usr/include/glib-2.0/ -I/usr/lib/glib-2.0/include/  \
     -I/usr/lib/glibmm-2.4/include/ -I/usr/lib/libxml++-2.6/include/
...
CHIMP -I '$(INC)' -o $@ $<
```

Figure 4.8: Sample CHIMP Makefile

toXML

Writing the code to convert a class to an XML file is fairly simple, but it is very tedious to do so manually. Ironically, for simply outputting XML, incorporating a third party library like `libxml++` can actually complicate things and increase the work load. So the goal is to make a few macros that can handle all of the library operations to create the XML document and write it to a file. These macros will be placed in a separate file, called "*toxml.cmf*", short for "CHIMP Macro File." The main file will be called XML_example01.mcpp, short for "Meta C Plus Plus." Please refer to Figure 4.9 for a full listing of the file. Now, a separate file called *toxml.cmf* will be created, which will include the source code and calls to `add_child`. Please see Figure 4.10 for the code for this. When run, the function simple::toXML and complicated::toXML are both implemented in proper C++ code.

fromXML

Reading data from an XML file is a lot harder than writing it to an XML file. In this case, the value of using an XML library really shines, since the effort of writing a parser for context free grammars is a lot of work. However, despite the help of libxml++, the process of getting data out of an XML document and into a live C++ class is still a very mundane process that we can avoid if we use a CHIMP macro. Like before, the code shall be separated into a metacode file and a source code file with a few metacode directives. Please see Figures 4.11, 4.12 and 4.13.

```
 1      #include <iostream>
 2      #include <fstream>
 3      #include <sstream>
 4      #include <string>
 5      using namespace std;
 6      #include <libxml++/libxml++.h>
 7
 8      {% include 'toxml.cmf' %}
 9
10      class simple
11      {
12      public:
13          simple() : a(42),b(13),
14              c("This is simple"),d(10.0f/6.0f)
15          {}
16          int a;
17          int b;
18          string c;
19          float d;
20          {@ proto_toXML() -@}
21      };
22
23      class complicated {
24      public:
25          complicated() :
26            component(),name("My name"),something(1)
27          {}
28          string name;
29          int something;
30          simple component;
31          {@ proto_toXML() -@}
32      };
33
34      //---------------------- Autogen
35      {@ make_toXML('simple') @}
36      {@ make_toXML('complicated') @}
37
38      int main()
39      {
40          complicated thing;
41
42          {% print "thing.toXML(\"output.xml\");" %}
43          return 0;
44      }
```

Figure 4.9: xml example 01.mcpp

```
 1 //---------------------- MetaCode
 2
 3 {% macro proto_toXML -%}
 4 public:
 5     int toXML(string fname);
 6     int _toXML(xmlpp::Element *lroot);
 7 {% endmacro -%}
 8
 9 {% macro make_toXML objname %}
10 {% set obj = ast.classes[objname] %}
11 int {@ obj.name @}::toXML(string fname)
12 {
13     xmlpp::Document doc;
14     xmlpp::Element *enode;
15     enode = doc.create_root_node("{@obj.name@}");
16     _toXML(enode);
17     doc.write_to_file(fname,"ISO-8859-1");
18 }
19
20 int {@ obj.name @}::_toXML(xmlpp::Element *lroot)
21 {
22     stringstream conv;
23     xmlpp::Element *enode;
24     xmlpp::TextNode *tnode;
25     {% for name,member in obj.members.items() -%}
26         // {@ member.tag @}
27         {% if not member.compound -%}
28             // Add atomic type
29             conv.str("");
30             conv << {@ name @};
31             enode = lroot->add_child("{@ name @}");
32             tnode = enode->add_child_text("quiet");
33             tnode->set_content(conv.str());
34
35         {% elif member.compound -%}
36             // Add compound type
37             enode = lroot->add_child("{@ name @}");
38             {@ name @}._toXML(enode);
39         {% endif -%}
40     {% endfor %}
41
42 }
43 {% endmacro %}
```

Figure 4.10: toxml.cmf

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <sstream>
 4 using namespace std;
 5 #include <libxml++/libxml++.h>
 6 {% include 'fromxml.cmf' %}
 7 //--------------------- Code
 8 class simple
 9 {
10 public:
11     simple(): a(42),
12       c("This is simple"),d(10.0f/6.0f) {}
13     int a;
14     string c;
15     float d;
16     {@ proto_fromXML() -@}
17     {@ friend_simpleDump('simple') @}
18 };
19
20 class complicated {
21 public:
22     complicated()
23         : component(),name("My name"),something(1)
24     {}
25     string name;
26     int something;
27     simple component;
28     {@ proto_fromXML() -@}
29     {@ friend_simpleDump('complicated') @}
30 };
31 //---------------------- Autogen
32 {@ make_fromXML('simple') @}
33 {@ make_fromXML('complicated') @}
34 {@ make_simpleDump('simple') @}
35 {@ make_simpleDump('complicated') @}
36
37 int main()
38 {
39     {% block main %}
40     complicated thing;
41     thing.fromXML("input.xml");
42     cout << thing << endl;
43     {% endblock %}
44     return 0;
45 }
```

Figure 4.11: xml example 02.mcpp

```
 1 //---------------------- MetaCode
 2
 3 {% macro proto_fromXML %}
 4 public:
 5     int fromXML(string fname);
 6     int _fromXML(xmlpp::Element *lroot);
 7 {% endmacro %}
 8
 9 {% macro make_fromXML objname %}
10 {% set obj = ast.classes[objname] %}
11 int {@ obj.name @}::fromXML(string fname)
12 {
13     xmlpp::DomParser parser;
14     xmlpp::Document *doc;
15     xmlpp::Element *root;
16     parser.parse_file(fname);
17     if(!parser){
18         return 0;
19     }
20
21     doc = parser.get_document();
22     root = doc->get_root_node();
23     _fromXML(root);
24 }
```

Figure 4.12: toxml.cmf part1

```
 1 int {@ obj.name @}::_fromXML(xmlpp::Element *lroot)
 2 {
 3     xmlpp::Element *enode;
 4     xmlpp::Node *node;
 5     string temp;
 6     {% for name,member in obj.members.items() -%}
 7         {% if not member.compound -%}
 8             {% if member.type ==
 9 "basic_string<char,std::char_traits<char>,std::allocator<char> >"
10             -%}
11             node = lroot->find("{@ name @}")[0];
12             enode = dynamic_cast<xmlpp::Element *>(node);
13             if(enode)
14             {
15                 temp = enode->get_child_text()->get_content();
16                 {@ name @} = temp;
17             }
18             {% else -%}
19             stringstream s_{@name@};
20             node = lroot->find("{@ name @}")[0];
21             enode = dynamic_cast<xmlpp::Element *>(node);
22             if(enode)
23             {
24                 temp = enode->get_child_text()->get_content();
25                 s_{@name@}.str(temp);
26                 s_{@name@} >> {@ name @};
27             }
28             {% endif -%}
29         {% elif member.compound -%}
30             node = lroot->find("{@ name @}")[0];
31             enode = dynamic_cast<xmlpp::Element *>(node);
32             if(enode)
33             {
34                 {@ name @}._fromXML(enode);
35             }
36         {% endif -%}
37     {% endfor %}
38 }
39 {% endmacro %}
```

Figure 4.13: toxml.cmf part2

4.3 Limited Runtime Reflection

Up until this point reflection has been demonstrated as a preprocessing feature, but is not accessible at run time. It is possible, however, to expose some of this reflection at runtime. This is very tricky to do, especially since C++ is a strictly typed language, and so functions must have a matching return type and return value. As an example, if we have a function like the one in Figure 4.14, it will fail to compile. The line that attempts to return "a" is fine, because the types match. However, the line that attempts to return "b" fails because the value of "b" is a string, but the function needs to return an int.

```
1  int function(name){
2      if(name == "a") // a is of type int
3          return a;
4      else if (name == "b") // b is of type string,
5                            // which is incompatible
6          return b;
7  }
```

Figure 4.14: Incompatible returns

We can resolve this problem by converting types to a common intermediate type. The C++ standard library offers a solution in the form of the *stringstream*, and its overloaded operators: `>>` `<<`. There is an example of this in Figure 4.15. Now, any types that allow use to write to or from a *stringstream* with these two operators will work (except for STL *string*s and *char ** strings, which have silly behavior for the `>>` operator, and need a special case). These operators are already overloaded for almost all primitive types, and can be overloaded by the user for new classes and types as is appropriate. One could even use imperative meta-programming techniques explained previously to overload these two operators for arbitrary classes.

Two methods must be implemented, a *get* and *set* function. The core logic for the *get* function is shown in Figure 4.16, and a full example with both functions can

```
1  stringstream ss;
2  type1 a;
3  type2 b;
4  ss << a;
5  ss >> b;
```

Figure 4.15: Type Conversion with SStream

be found in Figure 4.17, Figure 4.18 and Figure 4.19.

```
1 template <class T>
2 T _ref_get(string mname)
3 {
4     T ret;
5     stringstream ss;
6     {% for name,member in obj.members.items() -%}
7     if(mname == "{@ name @}"){
8         ss << {@ name @};
9         ss >> ret;
10        return ret;
11    }
12    {% endfor -%}
13    return ret;
14 }
```

Figure 4.16: Simple reflect code

```
 1 {% macro proto_reflective objname %}
 2 {% set obj = ast.classes[objname] %}
 3     static const char * const _ref_members[];
 4     template <class T>
 5     T _ref_get(string mname)
 6     {
 7         T ret;
 8         stringstream ss;
 9         {% for name,member in obj.members.items() -%}
10         if(mname == "{@ name @}"){
11             {% if member.type ==
12 'basic_string<char,std::char_traits<char>,std::allocator<char> >'
13                                                                 %}
14             ret = {@ name @};
15             {% else %}
16             ss << {@ name @};
17             ss >> ret;
18             {% endif %}
19             return ret;
20         }
21         {% endfor -%}
22         return ret;
23     }
24
25     template <class T>
26     void _ref_set(string mname,T val)
27     {
28         stringstream ss;
29         {% for name,member in obj.members.items() -%}
30         if(mname == "{@ name @}"){
31             {% if member.type ==
32 'basic_string<char,std::char_traits<char>,std::allocator<char> >'
33                                                                 -%}
34             {@ name @} = val;
35             {% else -%}
36             ss << val;
37             ss >> {@ name @};
38             {% endif -%}
39         }
40         {% endfor -%}
41     }
42 {% endmacro %}
```

Figure 4.17: Reflect part1

```
1 {% macro make_reflective objname %}
2 {% set obj = ast.classes[objname] %}
3 const char * const {@ obj.name @}::_ref_members[] = {
4     {% for name in obj.members %}
5     "{@ name @}",
6     {% endfor %}
7     NULL
8 };
9 {% endmacro %}
```

Figure 4.18: Reflect part2

```
 1 class simple
 2 {
 3 public:
 4     simple():a(42),b(13),c("This is simple"),d(10.0f/6.0f)
 5     {}
 6     int a;
 7     int b;
 8     string c;
 9     float d;
10     {@ proto_reflective('simple') @}
11 };
12
13 {@ make_reflective('simple') @}
14
15 int main()
16 {
17     simple thing;
18     cout << "Starting up" << endl;
19     {% block foo %}
20
21     cout << thing._ref_get<string>("a") << endl;
22     thing._ref_set<string>("a","33");
23     cout << thing._ref_get<string>("a") << endl;
24
25     cout << "----------------------------" << endl;
26     char * const *ptr;
27     for(ptr = (char * const *)simple::_ref_members;
28                                           *ptr != NULL; ptr++)
29     {
30         cout << *ptr << " : " << thing._ref_get<string>(*ptr)
31                                                   << endl;
32     }
33
34     {% endblock %}
35
36     return 0;
37 }
```

Figure 4.19: Reflect part3

4.4 Role Based Meta-Programming

Now that we are a bit more familiar with the prospect of meta-programming, we need to think about organization. Let's say we have one hundred functions, and we would like to do a meta-programming operation involving about twenty of them, but these twenty are randomly scattered throughout the list. It would be nice if we could label some of these functions with some meta-label that we could use later.

Let's take a hypothetical situation, we have a C++ class with a few members and we have a MySQL database. We want to create a table based on the definition of this class. For simple things like make an int called *number* and a string called *name*, this is easy. However, what if we want to make an int, which is also a primary key? How do we specify in the class definition which variable should be the primary key?

With CHIMP, we can take advantage of the GCC-XML feature that allows us to set meta-attributes on variables and functions (although not classes) through the use of the `__attribute` tag. For the purposes of CHIMP, we have dubbed this attribute to be a "role." Lets look at a simple example in Figure 4.20.

Of critical importance are the preprocessor lines that say

```
#define role(x) __attribute((gccxml(#x)))
```

and

```
#define primary_key role(primary_key)
```

The first line simply provides us with a clean interface to the GCC `__attribute` operator. With this in place, we can define new roles into the system by calling `#define rolename role(rolename)`. Now *rolename* is a valid modifier for variables, functions and class members. The parameter passed to "role" is the value that appears in the role variable associated with that member inside of the role list. We can filter lists of objects to only contain those which have a specified role with the "`role_filter`" function which is part of CHIMP.

```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4
 5 #define role(x) __attribute((gccxml(#x)))
 6 #define primary_key role(primary_key)
 7 #define secondary_key role(secondary_key)
 8
 9 class foo
10 {
11 public:
12     foo() :id(0),data(0),name("John Doe") {}
13     primary_key int id;
14     int data;
15     secondary_key string name;
16 };
17
18
19 int main()
20 {
21     {% for member in ast.classes['foo'].members.values()
22                               | role_filter('primary_key') %}
23         cout << "{@ member.name @}" << " is the PRIMARY KEY"
24                                                       << endl;
25     {% endfor %}
26
27     {% block anonymous %}
28     {% set keymember = ast.classes['foo'].members.values()
29                               | role_filter('primary_key')[0] %}
30     cout << "The KEY is " << "{@ keymember.name @}" << endl;
31     {% endblock %}
32     return 0;
33 }
```

Figure 4.20: Primary key role

4.5 Failings of CHIMP

While CHIMP is very useful for solving some problems, its complex execution flow causes some problems to arise.

4.5.1 Gotchas: debugging

CHIMP is, at present, only a proof of concept, and so is very light on the debugging capabilities. This means that many errors will pass silently through the Jinja template engine, resulting in incomplete code. Additionally, any C++ errors that occur on the stripped input file will result in an empty output file. There are two temporary file generated by CHIMP which can be found in the `/tmp/CHIMP/` directory: "raw.code" and "info.xml." These two files are used as intermediate by CHIMP, but can be insightful while debugging problems. The "raw.code" file is the input file, after it has been stripped of CHIMP tags. Info.xml is the full listing of classes, functions and other C++ objects, generated by GCC-XML.

4.5.2 Gotchas: vanishing prototypes

If we use CHIMP to autogenerate a function, we need to be aware of when the prototype will appear. The problem is that before the metaprogrammer has run, all metaprogramming tags will be stripped instead of run. This means your function will never be created. However, during the next stage, GCC-XML needs to parse a full and valid C++ document. This means that you may attempt to call an undefined function, which is an error in C++. This is more clear with an example, seen in Figures 4.21 through 4.25. Figure 4.21 presents an example of a function whose definition will only exist after the metacode has been evaluated. In Figure 4.22 is the same code example during the analysis phase, where the code has been stripped of all metacode. This code is not valid C++ because all functions must be declared before they have been defined. There are several ways around this, most obviously is to place a function prototype for the function outside of the metacode. However, this approach does not solve all possible problems of this nature, so instead we can address it with the code in Figure 4.23, which has both the function definition and the function call enclosed in metacode blocks. This code will be stripped into the code in Figure 4.24, which is valid because it has neither the function call nor the function definition. This is parsable C++, and will ultimately produce the code in

Figure 4.25, which is valid and complete code.

```
1  // Attempt 1 Metacode:
2  {@ make_function_foo() @}
3  int main()
4  {
5      foo();
6  }
```

Figure 4.21: Bad Metacode...

```
1  // Attempt 1 after the file is stripped
2  int main()
3  {
4      foo(); // Oops! the prototype for foo is gone!
5             // We cannot process this file!
6  }
```

Figure 4.22: Becomes bad C++ code

```
1  // Attempt 2 Metacode:
2  {@ make_function_foo() @}
3  int main()
4  {
5     {% block anonymous_1 %}
6      foo();
7     {% endblock %}
8  }
```

Figure 4.23: Better Metacode...

```
1  // Attempt 2 after the file is stripped
2  int main()
3  {
4      // Ah ha! Now both the prototype of foo
5      // and the call to it are gone...
6  }
```

Figure 4.24: Erroneous code vanishes

```
1  // Attempt 2 final stage
2  int foo() {... }
3  int main()
4  {
5      foo(); // Now after the meta-processor...
6             // foo has been defined above
7             // and appears in the body of main.
8  }
```

Figure 4.25: Correct code is generated

4.5.3 Gotchas: vanishing statements

There is another common problem in programming with CHIMP that arises during the analysis phase.

```
    cout << {@ someVar @} << endl;
```

After the metacode stripping phase this line becomes:

```
    cout <<  << endl;  // No good! Not valid!
```

This problem is just like the one above. We need to remove this entire statement during the preprocessing phase and have it come back for the application phase. There are two similar options for this. One option is to wrap the whole `cout` statement in a Jinja block, just as we did before, so that the whole thing will disappear until after we can fill in `{@ someVar @}` with its final value.

```
    {% block somerandomname %}
    cout << {@ someVar @} << endl;
    {% endblock %}
```

Alternatively, we could use the `{% print '' %}` tag to accomplish the same thing. This tag will vanish until the final meta-programming phase. The only practical difference is that `{% print '' %}` can be done cleanly in one line, while the `{% block %}` method is bulky for a single line, but clean for many lines.

```
    {% print 'cout << {@ someVar @} << endl;' %}
```

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The Imperative Meta-Programming Technique can be used to revitalize and extend a language without having to modify that languages definition or compiler at all. It can substantially reduce the amount of tedium in coding common tasks and patterns. The mechanism to develop with an Imperative MetaProgrammer is much more intuitive for users than the declarative nature of C++ Template Meta-Programming.

5.1.1 Gray Areas

The major drawback to using CHIMP is that it is somewhat complicated to use. It must be used in an organized manner or it can make code more difficult to understand, which puts additional weight on the developer to be organized. Of course, being organized is already expected of the developer so this is not a huge issue. Another aspect of its complexity comes in developing for it, while programming in CHIMP one is encouraged to solve all possible aspects of a problem, rather than the one specific case that you are currently faced with. This is nice in the long run because it saves the developer from having to write more code later, but it means that an immediate problem may be more complicated to solve with CHIMP than to write a simple duct-tape fix.

Another criticism would be that CHIMP metacode is very complicated and messy, one has to be aware of the constant context switching between template directives

and actual C++ code. While it is true that CHIMP code is messy, once it is written correctly, it should be quarantined into a library where it can be accessed cleanly via includes and macros. Many of the examples listed in this document contain more CHIMP code in the actual C++ file than would be usual in a real world scenario. It is more likely that there would be about 5 lines of CHIMP metacode per class definition in the actual code, and a few files of dedicated CHIMP macros.

5.2 Future Work

5.2.1 CHIMP Cleanup

While functional, the current iteration of CHIMP is not very user-friendly. There are several things that must be cleaned up before CHIMP is ready for mainstream distribution.

Organization

At the moment, there is no standard library for CHIMP, nor is there any place where one could keep commonly-used CHIMP macros. It would be nice to have a CHIMP standard library, kept in a system-wide location. Additionally, this system wide location should allow for third party CHIMP libraries to be installed. It would be convenient for users if CHIMP could also store libraries in a special folder within their home directory, saving them from having to rely on their systems administrator to install CHIMP libraries. All this would be much like Perl and the CPAN.

Install-ability

Our personal preference for package maintenance is the APT system used by Debian and Ubuntu. We would very much like to have CHIMP offered up through such means, either in the multiverse or in our own personal repository. In order to do this, we will also need to get an up to date version of Jinja into the repository as well, as the current Ubuntu repository only has Jinja 0.9, which is not compatible with the Jinja 1.2 release used in CHIMP.

5.2.2 CHIMP Variations

Alternative Template Tools

For CHIMP we used Jinja because it was easy to implement the strip operation, and was easy to pick up usage. However, the Jinja/Django template language is fairly limited, and forces users into the MVC model. As such, one cannot have large blocks of logical code mixed into the template. To this end, it would be worthwhile to consider other template languages. Alternatively, one could attempt to use the Ruby language and its ERB template language. This would allow for much more diversity in what you could do in meta code, such as writing to external files. It also could be that a whole new template engine should be written specifically for CHIMP.

CHIMP for Java

The code layout of Java actually lends itself much more to the IMP technique. Specifically, in C++ you need two macros to add a function to the class: one for the header file and one for the implementation file, which violates the redundancy principle that was mentioned in Section 3.1.2. Java would only need one macro, as the definition and the implementation are in one place.

Java already has a built-in reflection mechanism, which allows for many of the operations that we have shown so far. However, Java does not have any meta-programming or reflective tool for adding and implementing new functions. If a proper Java parser could be found to substitute for GCCXML then it might be interesting to see if a CHIMP for Java could be useful.

5.2.3 Inspired ideas

Role Based Programming

One realization that we came to while working with CHIMP was that the merging of logic and data, as is typical in object-oriented programming, might not be such a great idea after all. As an alternative, we would like to try to develop a simple proof of concept language which disjoints logic from data, much in the way CHIMP does.

In Figure 5.1 one can see a hypothetical example. In theory, the programmer would define **data** objects which are primarily or completely data members, each of which would have a formal name and several roles or characteristics. Then the programmer would define **logic** objects, which would define methods. Now a final class would be created by merging various logic and data objects together. Once merged, the methods from the logic objects would be made aware of any objects in their scope.

```
 1 DATA person_data {
 2     roles(serializable,key) string name;
 3     roles(serializable) string address;
 4     roles(serializable) string number;
 5 }
 6
 7 DATA transaction_data {
 8     roles(serializable,key) long int transaction id;
 9     roles(serializable) string cusomter_name;
10     roles(serializable) list(string) cart;
11 }
12
13 LOGICAL xml {
14     function toXML[obj](){
15         ret = ‘‘<xml_obj>’’;
16         for x in obj.role(serializable){
17             ret += x.toXML();
18         }
19         ret += ‘‘</xml_obj>’’;
20         return ret;
21     }
22 }
23
24 LOGICAL database {
25     function lookupInDatabase[obj](database){
26         key = obj.role(key);
27         ret = database.search(key);
28         return ret;
29     }
30     function doSomethingElse[obj](database){
31         ...
32     }
33 }
34
35 OBJECT employee = DATA(person_data)
36                 merged_with LOGIC(xml,database)
37  OBJECT transaction = DATA(transaction_data)
38                 merged_with LOGIC(xml,database)
```

Figure 5.1: Role Based Programming Pseudo Example

Appendix A

Chimp Source Listing

A.1 Chimp.py

```
#!/usr/bin/env python

import analysis
import template

import os
import sys
import getopt

if __name__=='__main__':
    dest = sys.stdout
    target = ''
    includes = ''

    cmdline_args = sys.argv[1:]
    opts,args = getopt.getopt(cmdline_args,"o:I:")

    for o,a in opts:
        if o == '-o':
            dest = file(a,"w")
        if o == '-I':
            includes = a

    if args:
        target = args[0]
        assert target
        assert os.path.isfile(target)

    try:
        os.mkdir('/tmp/chimp/')
    except:
        pass

    raw_cpp = '/tmp/chimp/raw.code'
    info_xml = '/tmp/chimp/info.xml'
```

```
os.system('rm -f %s'%raw_cpp)

template.stripFile(target,raw_cpp)

ast = analysis.analyzeFile(raw_cpp,info_xml,includes)

data = template.applyTemplate(target,ast)

dest.write(data)
```

A.2 Analysis.py

```
#!/usr/bin/env python

import os
import sys
import xml.etree.ElementTree as ET
import re

###############################################################################

global lookuptable
lookuptable = {}

global known_roles
known_roles = {}

###############################################################################

def digUpType(id):
    node = lookuptable[id]
    if node.tag in ['FundamentalType']:
        return node.attrib['name']
    elif node.tag in ['ArrayType']:
        tname = digUpType(node.attrib['type'])
        tname = tname + "[]"
        return tname
    elif node.tag in ['PointerType']:
        tname = digUpType(node.attrib['type'])
        tname = tname+"*"
        return tname
    else:
        #print "Warning, unknown xml tag type :",node.tag
        return "<mystery type>"

###############################################################################

class Obj:
    def __init__(self,node):
        self.node = node
        if not node.attrib.has_key('name'):
            node.attrib['name'] = node.attrib['mangled']
        self.name = node.attrib['name']
        self.tag = node.tag

        #Set defaults
        self.members = {}
        self.methods = {}
        self.compounds = {}
        self.classes = self.compounds
        self.structs = self.compounds
        self.type = ''
```

```
self.caste = ''
self.parameters = []
self.roles = []
self.arguements = []
self.compound = False

# Get members
if node.attrib.has_key('members'):
    for x in node.attrib['members'].split():
        target = lookuptable[x]
        if target.attrib and target.attrib.has_key('name'):
            subname = target.attrib['name']
            subobj = Obj(target)
            if subobj.tag in ['Class','Struct']:
                self.compounds[subname] = subobj
            elif subobj.tag in ['Field','Variable']:
                self.members[subname] = subobj
            elif subobj.tag in ['Function','Method']:
               self.methods[subname] = subobj

# Set type
try:
    if self.tag in ['Field','Variable']:
        if not node.attrib.has_key('type'):
            self.type = node.attrib['mangled']
        else:
            typeid = node.attrib['type']
            typenode = lookuptable[typeid]
            while typenode.attrib.has_key('type') or
                    typenode.attrib.has_key('returns'):
                typeid = typenode.attrib.get('type',
                            typenode.attrib.get('returns',None))
                typenode = lookuptable[typeid]
            if typenode.attrib.has_key('name'):
                self.type = typenode.attrib['name']
            elif typenode.attrib.has_key('mangled'):
                self.type = typenode.attrib['mangled']
            else:
                raise Exception('No type')
    elif self.tag in ['Function','Method']:
        typeid = node.attrib['returns']
        typenode = lookuptable[typeid]
        self.type = typenode.attrib['name']
    elif self.tag in ['Class','Struct']:
        if node.attrib.has_key('name'):
            self.type = node.attrib['name']
        else:
            self.type = node.attrib['mangled']
except:
    self.type = '<UNKNOWN>'

if self.tag in ['Function','Method']:
    lst = self.node.findall('Argument')
```

```
            for parm in lst:
                tname = ''
                pname = ''
                tname = digUpType(parm.attrib['type'])
                if parm.attrib.has_key('name'):
                    pname = parm.attrib['name']
                elif parm.attrib.has_key('mangled'):
                    pname = parm.attrib['mangled']
                else:
                    pname = 'anonymous'
                typle = (tname,pname, tname + pname)
                self.arguements.append(typle)
            pass

        # Set caste
        if self.tag in ['Field','Variable']:
            self.caste = 'variable'
        elif self.tag in ['Function','Method']:
            self.caste = 'function'
        elif self.tag in ['Class','Struct']:
            self.caste = 'compound'

        # Look for GCCXML attributes
        if node.attrib.has_key('attributes'):
            gxroles = node.attrib['attributes'].split()
            for r in gxroles:
               m = re.match(r"gccxml\((.*)\)",r)
               if m:
                   role = m.group(1)
                   if role:
                       self.roles.append(role)
                       if not known_roles.has_key(role):
                           known_roles[role] = []
                       known_roles[role].append(self)

        # Set compound standing
        if node.tag in ['Class','Struct']:
            self.compound = True
        elif node.tag in ['Field','Variable']:
            typenode = lookuptable[self.node.attrib['type']]
            if typenode.tag in ['Class','Struct']:
                self.compound = True

################################################################################

class Content:
    def __init__(self):
        self.compound = {}
        self.classes = self.compound
        self.structs = self.compound
        self.variables = {}
        self.functions = {}
        self.roles = {}
```

```
    def addObj(self,obj):
        name = obj.name
        if obj.tag in ['Class','Struct']:
            self.compound[name] = obj
        elif obj.tag in ['Function','Method']:
            self.functions[name] = obj
        elif obj.tag in ['Variable','Field']:
            self.variables[name] = obj

################################################################################

def analyzeFile(fname,oname='tmp/processed.xml',cflags = ''):
    cmdstring = 'gccxml --gccxml-cxxflags "%s"  %s -fxml=%s 2>&1'
        %(cflags,fname,oname)
    #print cmdstring
    res = os.popen(cmdstring)
    result = res.read()
    if len(result):
        raise Exception('Compile failed!\n'+result)

    tree = ET.parse(oname)
    structs = tree.findall('//Struct')
    classes = tree.findall('//Class')
    base_functions = tree.findall('//Function')
    variables = tree.findall('//Variable')

    # There are a lot of __builtin functions that gccxml puts there.
    # I don't care about them
    functions = filter(lambda x :
                not x.attrib['name'].startswith('__builtin'),base_functions)

    # Top
    toplevel = structs + classes + variables + functions

    for x in tree.findall('/*'):
        lookuptable[x.attrib['id']] = x

    content = Content()

    for node in toplevel:
        obj = Obj(node)
        content.addObj(obj)
    content.roles.update(known_roles)


    return content
```

A.3 Template.py

```
#!/usr/bin/env python
import os
import jinja
import sys
from pprint import pprint
from jinja.translators.python import PythonTranslator,Template



class JohnsTranslator(PythonTranslator):
#    def __init__(self,*args,**kwargs):
#        PythonTranslator.__init__(self,*args,**kwargs)

    def process(environment, node,source=None):
        """
        The only public method. Creates a translator instance,
        translates the code and returns it in form of an
        `Template` instance.
        """
        translator = JohnsTranslator(environment, node,source)
        filename = node.filename or '<template>'
        source = translator.translate()
        return Template(
            environment,
                compile(source, filename, 'exec'))
    process = staticmethod(process)

    def handle_for_loop(self,node):
        return ""

    def handle_if_condition(self,node):
        return ""

    def handle_cycle(self,node):
        return ""

    def handle_print(self,node):
        return ""

    def handle_macro(self,node):
        return ""

    def handle_call(self,node):
        return ""

    def handle_set(self,node):
        return ""

    def handle_filter(self,node):
        return ""
```

```
    def handle_block(self,node):
        return ""

    def handle_include(self,node):
        return ""

def stripFile(targetname,destname):
    assert os.path.isfile(targetname)

    env = jinja.Environment('{%', '%}', '{@', '@}', '{#', '#}',
            trim_blocks=True,loader=jinja.FileSystemLoader("./"))
    tree = env.parse(file(targetname).read())
    rv = JohnsTranslator.process(env, tree)
    data = rv.render()
    fout = file(destname,'w')
    fout.write(data)
    fout.close()

def do_role_filter(key):
    def searchkey(mem):
        try:
            if key in mem.roles:
                return True
        except:
            return False
        return False
    def wrapped(env,context,value):
        if type(value) is list:
            ret = filter(searchkey,value)
            return ret
        else:
            return value
    return wrapped


def applyTemplate(target,ast):
    env = jinja.Environment('{%', '%}', '{@', '@}', '{#', '#}',
            trim_blocks=True,loader=jinja.FileSystemLoader("./"))
    env.filters['role_filter'] = do_role_filter

    tmpl = env.get_template(target)
    ret = tmpl.render(ast=ast)
    ret = ret + "\n"
    return ret
```

Appendix B

Applications of Chimp

B.1 LUA

B.1.1 toLua

toLua MetaCode

```
#include <lua5.1/lua.h>
#include <lua5.1/lauxlib.h>
#include <lua5.1/lualib.h>

{% macro proto_toLua -%}
public:
    int toLua(string fname);
    int _toLua(stringstream &s,string indent = "");
{% endmacro -%}

{% macro make_toLua objname %}
{% set obj = ast.classes[objname] %}
int {@ obj.name @}::toLua(string fname)
{
    stringstream s;
    fstream fout;
    s << "-- To Lua" << endl;
    s << "{@obj.name@} = " ;
    _toLua(s);
    s << endl << endl;

    fout.open(fname.c_str(),fstream::out);
    cout << s.str() << endl;
    fout << s.str() << endl;
    fout.close();
}

int {@ obj.name @}::_toLua(stringstream &s,string indent)
{
    s << " {" << endl;
        {% for name,member in obj.members.items() %}
```

```
            {% if not member.compound %}
                {% if member.type == "basic_string<char,std::char_traits<char>,std::allocator<cha
        s << indent << "    " << "{@ name @} = \"" << {@ name @} << "\"" ;
                {% else %}
        s << indent << "    " << "{@ name @} = " << {@ name @};
                {% endif %}
            {% else %}
        s << indent << "    " << "{@ name @} = ";
        {@ name @}._toLua(s,indent + "    ");
            {% endif%}

        {% if not loop.last %}
        s << ",";
        {% endif %}
        s << endl;

        {% endfor %}
    s << indent << "}";
}
{% endmacro %}
```

toLua Main

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;

{% include 'toLua.cmf' %}

//---------------------- Code

class simple
{
public:
    simple();
    int a;
    int b;
    string c;
    float d;
    {@ proto_toLua() -@}
};

class complicated {
public:
    complicated();
    string name;
    int something;
    simple component;
    {@ proto_toLua() -@}
};
```

```
simple::simple()
{
    a = 42;
    b = 13;
    c = "This is something simple";
    d = 10.0f/6.0f;
}

complicated::complicated()
: component()
{
    name = "My name";
    something = 1;
}

//---------------------- Autogen
{@ make_toLua('simple') @}
{@ make_toLua('complicated') @}

int main()
{
    complicated thing;


    {% print "thing.toLua(\"output.lua\");" %}
    return 0;
}
```

B.1.2 fromLua

fromLua MetaCode

```
#include <lua5.1/lua.hpp>
#include <lua5.1/lauxlib.h>
#include <lua5.1/lualib.h>
//---------------------- MetaCode

{% macro proto_fromLua -%}
public:
    int fromLua(string fname);
    int _fromLua(lua_State *L);
{% endmacro -%}

{% macro make_fromLua objname %}
{% set obj = ast.classes[objname] %}
int {@ obj.name @}::fromLua(string fname)
{
    lua_State *L;
    L = lua_open();
    luaL_reg *lib;
    luaL_reg lualibs[] =
```

```
    {
        {"base",luaopen_base},
        {NULL,NULL}
    };

    for (lib = lualibs; lib->func != NULL; lib++)
    {
        lib->func(L);
        lua_settop(L,0);
    }


    luaL_dofile(L,fname.c_str());

    lua_getglobal(L,"{@obj.name@}");
    if(!lua_istable(L,-1))
    {
        cerr << "Warning, {@obj.name@} is not a table" << endl;
        return 0;
    }

    _fromLua(L);

    lua_pop(L,1);
    lua_close(L);
    return 0;
}

int {@ obj.name @}::_fromLua(lua_State *L)
{
  {% for name, member in obj.members.items() %}
    {% if member.compound %}
        lua_pushstring(L,"{@ name @}");
        lua_gettable(L,-2);
        {@name@}._fromLua(L);
        lua_pop(L,1);

    {% else %}
      {% if member.type ==
"basic_string<char,std::char_traits<char>,std::allocator<char> >" -%}
        lua_pushstring(L,"{@ name @}");
        lua_gettable(L,-2);
        {@ name @} = lua_tostring(L,-1);
        lua_pop(L,1);
      {% else %}
        lua_pushstring(L,"{@ name @}");
        lua_gettable(L,-2);
        {@ name @} = static_cast<{@member.type@}>(lua_tonumber(L,-1));
        lua_pop(L,1);
      {% endif %}
    {% endif %}
  {% endfor %}
    return 0;
```

```
}
{% endmacro %}
```

fromLua Main

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;

{% include 'fromLua.cmf' %}
//---------------------- Code

class simple
{
public:
    simple();
    int a;
    int b;
    string c;
    float d;
    {@ proto_fromLua() -@}
};

class complicated {
public:
    complicated();
    string name;
    int something;
    simple component;
    {@ proto_fromLua() -@}
};

simple::simple()
{
    a = 42;
    b = 13;
    c = "This is something simple";
    d = 10.0f/6.0f;
}

complicated::complicated()
: component()
{
    name = "My name";
    something = 1;
}

//---------------------- Autogen
{@ make_fromLua('simple') @}
```

```
{@ make_fromLua('complicated') @}

int main()
{
    complicated thing;


    {% print "thing.fromLua(\"input.lua\");" %}
    cout << "thing.name        : " << thing.name << endl;
    cout << "thing.something   : " << thing.something << endl;
    cout << "thing.component.a : " << thing.component.a << endl;
    cout << "thing.component.b : " << thing.component.b << endl;
    cout << "thing.component.c : " << thing.component.c << endl;
    cout << "thing.component.d : " << thing.component.d << endl;
    return 0;
}
```

Bibliography

[1] Codesmith tools. http://www.codesmithtools.com/, 2007. [last access March 2008].

[2] James Clark, Steve(Inso Corp. DeRose, and Brown University). Xml path language (xpath). http://www.w3.org/TR/xpath, 1999. [last access February 2008].

[3] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2004.

[4] The SCons Foundation. Scons: A software construction tool. http://www.scons.org/, 2004. [last access February 2008].

[5] Python Foundaton. Python programming language. http://www.python.org/, January 2007. Webpage, Python was developed by Guido van Rossum and the Python Software Foundation [last access February 2008].

[6] Aleksei Gurtovoy and David Abrahams. The boost c++ metaprogramming library. http://www.boost.org/libs/mpl/doc/index.html. [last access February 2008].

[7] Kitware. Gcc-xml. http://www.gccxml.org/, January 2007. GCC-XML was developed by King, Brad at Kitware [last access February 2008].

[8] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. http://dinosaur.compilertools.net/lex/index.html. [last access February 2008].

[9] Libxml++. http://libxmlplusplus.sourceforge.net/, January 2007. Developed by Ari Johnson, Christophe de Vienne and Murray Cumming [last access February 2008].

[10] Mike Orr and Tavis Rudd. Cheetah - the python-powered template engine. http://www.cheetahtemplate.org/, 2001. [last access February 2008].

[11] Armin Ronacher and the Pocoo Team. Jinja templates. http://jinja.pocoo.org/, January 2007. [last access February 2008].

[12] Hermanpreet Singh. Introspective c++. Master's thesis, Virginia Tech.

[13] AT&T Bell Laboratories Stephen C. Johnson. Yacc: Yet another compiler-compiler. http://dinosaur.compilertools.net/yacc/index.html. [last access February 2008].

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] Ryan Tomayko. Kid language specification. http://www.kid-templating.org/language.html, 2005. [last access February 2008].