The C++ Hybrid Imperative Meta-Programmer: CHIMP

John L. Kenyon, Frederick C. Harris Jr., Sergiu M. Dascalu
University of Nevada, Reno
Department of Computer Science and Engineering
1664 N. Virginia St., Reno, NV 89557
{jkenyon,fredh,dascalus}@cse.unr.edu

Abstract

Neither the C nor the C++ languages support reflection and their meta-programming capabilities are very limited. Both of these problems can be addressed by adding a pre-processing step, which can analyze and modify the code before it is passed to the actual compiler. By using this method, we can simplify many typical 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. This paper presents CHIMP and demonstrates the benefits of this kind of meta-programming by applying it to a frequently encountered XML problem.

1 Introduction

One of the goals of a programmer is to produce as much functionality as possible in the limited timeframe of the working day. For this reason, there is a perpetual arms race to develop new programming languages, integrated environments, and platforms, each designed to make the task of programming easier, faster, more reliable, and less repetitive. However, for various practical and historical 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++ in creative ways. This paper proposes an alternative approach tentatively entitled Imperative Meta-Programming (IMP).

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 allows us to automatically generate large amounts of code based on the structure and content of existing structures, and it enables us to do so without having to change the C/C++ compilers and languages definitions. In order to develop the imperative meta-programming idea, we have created the tool called CHIMP. CHIMP is written in Python and uses various other open source components.

The remainder of the document is laid out as follows: Section 2 covers related work and similar tools to CHIMP, Section 3 explains what imperative meta-programming is and how we implemented CHIMP, Section 4 presents the application of CHIMP to an XML serialization problem, and Section 5 draws the conclusions of the paper and explains future work.

2 Related Work

2.1 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* [4]. Programs like yacc [13] and lex [6] are also meta-programmering tools, 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 one want a container for another data type, one 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].

As such, template meta-programming is a powerful meta-programming technique. It has been used to great benefit in the Boost libraries [4]. However, there are various constraints on the C++ template engine, and several drawbacks to using it. The main obstacle to using template meta-programming stems from the fact that it is hard to use. 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.

Additionally, template meta-programming does not directly support reflection. Some attempts have been made to add reflection support to C++ template meta-programming. Many of them have the flaw that they require the user to explicitly list class members a second time, in a preprocessor macro [1]. 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 component.

2.2 Reflection

The second major gain from the proposed *imperative meta-programming* approach is that it allows reflection. 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 intelligent programming techniques. A prime example of reflection in action is the pickle module in Python [10]. 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 [10].

Since the C++ standard does not offer any reflection, attempts to add it often require stretching the language definition. There have been several attempts to do this. 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. While this does leave the language mostly intact, it still changes the language slightly. Additionally, it requires the developer to use the template meta-programming technique, which has various faults addressed before.

In contrast to these attempts, one goal of imperative meta-programming is to work with existing compilers as they are. Changes to the C++ language are a particularly bad idea since it would violate a well established standard, and it is very difficult to overcome the momentum that an old standard has. Additionally, it would require a fork of an existing compiler or an entirely new compiler, and either of these would be difficult to maintain. Thus, for practical reasons, sticking to established compilers such as GCC is key to our approach.

2.3 Code Generation

A code generator is a tool that will generate source code based on some input parameters, such as Yacc [13], Lex [6], Redwood [16], or COGENT [2].

The most similar tool to our idea is the development tool called CodeSmith [3], an extremely powerful code generation tool. CodeSmith uses the same web style template system that we are using in CHIMP. It allows users to generate massive amounts of C# or VB.NET code based on the contents and definitions of SQL databases and tables, or based on an XML schema[3]. One drawback is that CodeSmith was designed for the .NET plaftorm, and not for general use with C and C++. Additionally, it does not add any reflective mechanism to the languages, most likely because the languages it support already have reflection as a built-in feature.

3 Imperative Meta-Programming

The *imperative meta-programming* 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 are then processed and either evaluated or removed, resulting in a new file completely composed of the target language. From here the compilation process can procede normally. For the remainder of the document, we assume that C++ is the target language.

The two stages of the IMP preprocessor are analysis and application. The *analysis* phase collects information about classes, and the *application* phase executes logical structures included in the form of metacode.

3.1 Application Phase

The application phase is fairly straight forward 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++ with CHIMP), with small bits of dynamic content that are generated by the template language. Figure 1 shows these similarities by comparing PHP code embeded in HTML, along with some equivalent CHIMP code embedded in C++.

```
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 1. PHP embedded in HTML and CHIMP embedded in C++

These two side by side examples should clarify the basic idea, that we can embed logic directly in C++ code, without interfering with C++ syntax. This allows us to perform some very complex actions to generate repetitive or tricky code.

It is fairly easy to see how the web style template engine is used. In this case, we are using a modified version of the Jinja template engine, although there are several other template engines that were considered. Python has many popular template engines, such as Kid [15], PSP, Spyce, and Cheetah [9] [8]. 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 [9]. After some additional research, the Jinja template engine was selected for the CHIMP prototype.

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 advantage 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 one's 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.

3.2 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 produce much more useful code if we provide the application phase with more information about the code itself. Lets look at some simple repetitive tasks that are sensitive to existing code. One such example is a common function for debugging called 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 2 is a perfect example of tedious, 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 variable;`, the programmer would now need to go down to the DumpToScreen function for this struct, and add the line `cout << " variable : " << ptr->variable << endl;`. This is a cumbersome process, and it is an error prone one as well. 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 raw text, then one would need to go through all of these functions and fix them. This would be a vastly time consuming and tedious processes.

```
 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 2. C++ Code for the DumpToScreen function

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. In that case, we assume we can iterate over the members of the target struct.

C++ does not support any form of reflection, and thus 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 provide this information to our template engine, then our IMP approach would be able to write code equivalent to that shown in Figure 3.

In Figures 4 and 5 we address this problem with a com-

```
 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 member name : member value
12  }
```

Figure 3. PseudoCode for the DumpToScreen function

plete C++ program. The program shown in Figure 4 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 contents of struct definition, one could add as many members or change the types of any members, and the source code would be updated appropriately after the next compilation. This is very powerful because the DumpToScreen function is now almost 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.

To analyze the C++ code we need a C++ parser. This is by far the most complicated step in the process, and so we decided to use GCC-XML. GCC-XML will parse a C++ file and read all information about the top level objects into an XML file [5]. These objects include: 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.

4 XML Interfaces with CHIMP

One of the more tedious tasks in C++ 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 serialize and de-serialize objects. One of the more popular file formats for serialization is the XML format. In the following examples, CHIMP is used to automate the process for simple classes.

```
 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 @} : "
10           << ptr->{@ name @} << endl;
11      {% endfor %}
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  {@ DumpToScreen(ast.structs['ExampleStruct']) @}
23
24  int main()
25  {
26      ExampleStruct e;
27      DumpToScreen(&e);
28      return 0;
29  }
```

Figure 4. C++ metacode for DumpToScreen, written for CHIMP

For this purpose we used the LibXML++ library, created by the Gnome group [7]. It provides a convenient object-oriented interface to parsing and creating XML files, and provides XPath capabilities, which is used to parse the files. This example is a limited proof of concept, and as such it only handles primitive types (int, float, etc.), STL strings, and member objects. For simplicity it does not handle arrays, pointers, STL lists/vectors, or STL maps. These types will be addressed in a later release.

Writing the code to convert a class to an XML file is fairly simple, but it is still tedious and repetitive. For outputing XML code, using a library such as LibXML++ may seem excessive, but it gives us a guarantee that the output is valid XML. So our 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 are placed in a separate file, called "ToXML.cmf", where the cmf extension is short for "CHIMP Macro File." The main file is called "XML_example_01.mcpp", with the extension "mcpp" short for "Meta C Plus Plus." Figure 6 presents a partial listing of the file.

In order to properly produce XML, we have two functions, toXML and _toXML. The first, toXML, is a wrapper function that provides a clean interface, and the _toXML function is a recursive function. The toXML macro is shown in Figure 7 and the _toXML macro is shown in Figure 8.

```
 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 5. DumpToScreen generated from metacode

Reading data from an XML file is a lot harder then 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 from LibXML++ library, 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. Similar to the previous example, the code shall be separated into a metacode file and a source code file with a few metacode directives. Please see Figures 9 and 10 for the macros fromXML and _fromXML.

```
1  struct simple
2  {
3    simple() : a(42),b(13), c("c"),d(10.0f/6.0f) {}
4    int a;
5    string b;
6    float c;
7    {@ proto_toXML() -@}
8  };
9  {@ make_toXML('simple') @}
```

Figure 6. XML_example_01.mcpp : A sample C++ class and the metacode for XML

5 Conclusions and Future Work

Through use of CHIMP, we have automated several tasks that are both common and tedious. We have demonstrated how we can produce a C++ program where the code describes what we want to accomplish instead of what the language requires. The primary examples shown here were

```
 1  {% macro proto_toXML -%}
 2  public:
 3    int toXML(string fname);
 4    int _toXML(xmlpp::Element *lroot);
 5  {% endmacro -%}
 6
 7  {% macro make_toXML objname %}
 8  {% set obj = ast.classes[objname] %}
 9  int {@ obj.name @}::toXML(string fname)
10  {
11    xmlpp::Document doc;
12    xmlpp::Element *enode;
13    enode = doc.create_root_node("{@obj.name@}");
14    _toXML(enode);
15    doc.write_to_file(fname,"ISO-8859-1");
16  }
```

Figure 7. ToXML.cmf : Metacode to write an XML serialization function

```
17  int {@ obj.name @}::_toXML(xmlpp::Element *lroot)
18  {
19    stringstream conv;
20    xmlpp::Element *enode;
21    xmlpp::TextNode *tnode;
22    {% for name,member in obj.members.items() -%}
23      {% if not member.compound -%}
24        conv.str("");
25        conv << {@ name @};
26        enode = lroot->add_child("{@ name @}");
27        tnode = enode->add_child_text("quiet");
28        tnode->set_content(conv.str());
29      {% elif member.compound -%}
30        enode = lroot->add_child("{@ name @}");
31        {@ name @}._toXML(enode);
32      {% endif -%}
33    {% endfor %}
34  }
35  {% endmacro %}
```

Figure 8. ToXML.cmf (continued): Metacode to write an XML serialization function

for producing XML, however this is only one of many things that can be done with *imperative meta-programming*. We have developd various other working examples using CHIMP to produce code to interface with MySQL databases, interface with the LUA scripting engine, and implement a run time reflection mechanism. Examples of these have not been shown here due to space constraints.

In summary, we have shown how the proposed *imperative meta-programming* Technique can be used to revitalize and extend an existing language without having to modify that language's definition or compiler at all. Additionally, it can substantially reduce the amount of tedium in coding common tasks and patterns. Furthermore, the basic usage of such systems is reasonably easy to learn because of its similarity to web development.

In using CHIMP it has become clear that the Jinja/Django template language is fairly limited compared to the full power of a language such as Python. At present, we are working on a new template system based more closely on the PSP model, with a few variations that make it

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

Figure 9. FromXML.cmf : Metacode to write an XML deserialization function

more effective and easier to use with C++. This will allow us to harness the full power of Python in our templates. Especially interesting is the ability to have a template function generate an external file, such as an XML DTD or schema. Alternatively, one could read external data sources, such as an XML file or a MySQL database, and generate code based on their contents or description.

Near future work includes finalizing the documentation and making the CHIMP tool and its working examples available online.

```
21 int {@ obj.name @}::_fromXML(xmlpp::Element *lroot)
22 {
23   xmlpp::Element *enode;
24   xmlpp::Node *node;
25   string temp;
26   {% for name,member in obj.members.items() -%}
27    {% if not member.compound -%}
28     {% if member.type == "std::string" -%}
29     node = lroot->find("{@ name @}")[0];
30     enode = dynamic_cast<xmlpp::Element *>(node);
31     if(enode)
32     {
33      temp = enode->get_child_text()->get_content();
34      {@ name @} = temp;
35     }
36     {% else -%}
37     stringstream s_{@name@};
38     node = lroot->find("{@ name @}")[0];
39     enode = dynamic_cast<xmlpp::Element *>(node);
40     if(enode)
41     {
42      temp = enode->get_child_text()->get_content();
43      s_{@name@}.str(temp);
44      s_{@name@} >> {@ name @};
45     }
46     {% endif -%}
47    {% elif member.compound -%}
48     node = lroot->find("{@ name @}")[0];
49     enode = dynamic_cast<xmlpp::Element *>(node);
50     if(enode)
51     {
52      {@ name @}._fromXML(enode);
53     }
54    {% endif -%}
55   {% endfor %}
56 }
57 {% endmacro %}
```

Figure 10. FromXML.cmf (continued): Metacode to write an XML deserialization function

References

[1] G. Attardi and A. Cisternino. Reflection support by means of template metaprogramming. *Lecture Notes in Computer Science*, 2186:118–128, 2001.

[2] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

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

[4] A. Gurtovoy and D. Abrahams. The boost c++ metaprogramming library. http://www.boost.org/libs/mpl/doc/index.html, 2002. [last access February 2008].

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

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

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

[8] A. Martelli, A. Ravenscroft, and D. Ascher. *Python Cookbook*. O'Reilly Media, Inc., March 2005.

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

[10] Python Software Foundation. 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].

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

[12] H. Singh. Introspective c++. Master's thesis, Virginia Tech, 2004.

[13] A. B. L. Stephen C. Johnson. Yacc: Yet another compiler-compiler. http://dinosaur.compilertools.net/yacc/index.html, 2008. [last access February 2008].

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

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

[16] B. T. Westphal, F. C. Harris, and S. Dascalu. Snippets: Support for drag-and-drop programming in the redwood environment. *J. UCS*, 10(7):859–871, 2004.