Implementing a C++ extension: class namespaces

Vincent Ordy

Technical Report n^o0813, avril 2008
revision 1884

In C++, programmers are used to distinguish definition from implementation. But the syntax for C++ classes is repetitive and tedious, especially with templates and nested classes. We propose to extend the C++ grammar for the purpose of implementing member functions and initialize static data members already declared in the class definition. This work will be based on TRANSFORMERS' C++ grammar and transformation rules in STRATEGO Language.

En C++, les programmeurs séparent la définition de l'implémentation. Mais pour les classes, la syntaxe du C++ oblige l'utilisation d'une syntaxe répétitive, en particulier dans le cas de classes paramétrées ou imbriquées. On se propose donc de faire une extension de la grammaire du C++ permettant d'implémeter des méthodes et d'initialiser des attributs statiques déjà déclarés dans la définition de la classe. Dans ce but, nous utiliserons la grammaire du C++ implémentée dans TRANSFORMERS et des transformations écrites en STRATEGO.

Keywords
Transformers, program transformation, C++, STRATEGO/XT

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
vincent.ordy@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Introduction

The TRANSFORMERS project aims at designing a set of tools to manipulate C++ programs. Now that C++ parsing begins to be usable, we want to begin to use it to implement some basic transformations like it as been done for C-Transformers (Borghi et al., 2006). But there are still some limitations, revcpp the preprocessor not being integrated in TRANSFORMERS for example.

We will first have an overview of other transformations and related work. Then we will described how to extend TRANSFORMERS' C++ grammar for a transformation, and finally how to write the transformation itself.

We will be using the class namespace extension, as described by Daniel and Grund (2003), as an example all along the document. This addition to the C++ language allows us not to have long fully qualified names for methods, their return type and static members of classes.

Acknowledgements I want to thank Nicolas Pierron, Valentin David and Florian Quèze for their help to understand TRANSFORMERS. I also want to thank Akim Demaille, Maxime Van-Noppen and Florian Lesaint for their comments on earlier drafts of this report.

Chapter 1

State of the Art

First lets see other extensions and transformations, and how they have been implemented.

1.1 CodeBoost

The University of Bergen (Norway) is developing a C++ library, SOPHUS, providing high-level abstractions for implementing partial differential equation solvers. But due to high-level abstraction and the algebraic notation they wanted to use to simplify programming and readability, they encountered poor run-time performances.

One of the major problems is that the compiler uses temporary variables to evaluate each operation of an expression with user-defined types, including assignments. For example, let *a* and *b* be two objects of *C* class type.

```
a = a + b;
```

will be evaluated as

```
C temp;
temp = a + b;
a = temp;
```

With a predefined type like an integer, the compiler would not have used a temporary variable. But with temporary variable, we have to construct the new variable, and copy the result into the variable *a*. These operations can take a long time depending on the variable type.
With a matrix and without algebraic notation, we would have probably done it with a function like *opAdd (T& out, const T& in1, const T& in2)*, and used it like this: *opAdd (A, A, B);*. All we really need is some sugar to change the algebraic notation into non-algebraic and optimized notation.

They implemented these transformations in STRATEGO. Unlike TRANSFORMERS, CODEBOOST (O. S. Bagge, 2000) does not implement its own C++ parser, but uses OPENC++ (Chiba, 1997) and is limited to process a simplified version of the C++ (only what they really need to do the transformations they want).

Eventually TRANSFORMERS could be used as frontend to parse C++ for CODEBOOST. However TRANSFORMERS is too slow and disambiguation is still a problem at the moment.

1.2 CMM

CMM (Smith, 2000) (C++ with MultiMethods) is a program that implements source to source transformations for C++. It is fully written in C++. The parser, like CODEBOOST's, only parses a subset of the C++ syntax (only what is needed for the extensions).

Alternative Declaration Syntax

One of the extensions allows us to use an alternative declaration syntax, as described in *The Design and Evolution of C++* by Stroustrup (1994).

The CMM's website (Smith, 2000) gives as an example the *signal()* function prototype, which becomes in the alternative form:

```
signal: ( sig: int, func: ->(sig: int) void) -> (sig: int) void;
```

Embedded Functions

Another extension is the ability to have embedded functions. This is nothing other than anonymous functions. CMM gives them automatically an unique name and changes the anonymous function declaration to a call of the generated function.

This extension is useful when using STL algorithms or other functions that need a function as argument.

For example:

```
for_each(M.begin(), M.end(),
        compose1(mem_fun(&Object::Draw),
                select2nd<map>int, Object *>::value_type>()));
```

that can also be written

```
void fctDraw (Map::iterator it)
  {
    it->second->Draw();
  }

for_each(M.begin(), M.end(), fctDraw);
```

can be replaced with:

```
for_each(M.begin(), M.end(),
         void ()( Map::iterator it) { it->second->Draw();});
```

MultiMethods

The most important C++ extension implemented in CMM is the multimethod support. Let us first explain what are multimethods with the CMM example:

```
struct  Shape            {...};
struct  Square : Shape   {...};
struct  Triangle : Shape {...};

bool    Overlap( virtual Shape& a, virtual Shape& b);

bool    Overlap_( static Square&   a, static Triangle& b) {...}
bool    Overlap_( static Triangle& a, static Square&   b) {...}
bool    Overlap_( static Shape&    a, static Square&   b) {...}
bool    Overlap_( static Square&   a, static Shape&    b) {...}
```

CMM removes the virtual modifiers of the Overlap() function, and automatically generates the implementation for this function. This dispatch function calls the right Overlap_() function based on the arguments, using the *Run-Time Type Informations* (RTTI) from C++. RTTI allows to know at runtime the real type of an object.

1.3 Others

1.3.1 ASPECTC++

ASPECTC++ (Spinczyk, 2005) is an aspect-oriented extension of the C++. It uses PUMA (Pure Manipulation) (Urban, 1999) for C++ manipulation. The lexer is generated using flex++ and the parser using LEMON (Hipp, 2000).

1.3.2 ContractC with TRANSFORMERS

An extension for the C grammar(Borghi et al., 2006) has already been done with TRANSFORMERS to allow the "design by contract" support. It allows to give assertions on a function, before and after its execution. It enhances code readability, because it splits condition on input/output from the function algorithm.

Example from TRANSFORMERS's website:

```
double sqrt (double r)
    precondition
    {
        r >= 0.;
    }
    postcondition (result)
    {
        result >= 0.;
        equal_within_precision (result * result, r);
    };

double sqrt (double r)
{
  return _libc_sqrt (r);
}
```

1.3.3 GCC

GCC itself allows language extension for the C++.

But:

- GCC outputs a binary file, not a source file,
- the provided extensions may vary depending on GCC version.

In the 2.95 version for example, they were a '<?' operator, returning the minimum of two values. It is no more included in newer GCC versions.

All the extensions are described in the GCC manual (GCC Manual , GCC extensions).

Chapter 2

The Grammar Extension

2.1 Grammar Modularity

For some extensions, we need to extend the C++ standard grammar. But we do not want to be intrusive.

Hopefully, SDF allows us to do so by his syntax and because we can import other files. The process has already been described by Anisko et al. (2003).

2.2 Example: The Class Namespaces

Basically, we need to find what the extension should produce and what we need in input.

The easiest way to find out how to plug in into the C++ grammar is to find something that already exists in the C++ grammar and looks like our extension. Let the *namespace* definition be it.

```
NamedNamespaceDefinition    → NamespaceDefinition
OriginalNamespaceDefinition  → NamedNamespaceDefinition
"namespace" Identifier "{" NamespaceBody "}" → OriginalNamespaceDefinition
```

Now we have to find how are classes and structs described (in file '`ClassSpecifiers.esdf`').

```
"class"  → ClassKey
"struct" → ClassKey
"union"  → ClassKey

Identifier → ClassName

ClassHead "{" MemberSpecification? "}" → ClassSpecifier
```

By mixing the two previous definitions, we get

```
"namespace" ClassHead "{" MemberSpecification? "}" → ClassNamespace
```

But there is a problem here, *MemberSpecification* is too permissive: we only want to allow static member and method definitions, and nested class namespaces.

There are two ways to do it: we can either add to the grammar some rules that only accept what we need or use the previous rules and prune with attributes. We will now compare the two options.

2.2.1 Adding a rule to the grammar

It seems easier to add a rule in place of MemberSpecification to only accept what we need.

```
"namespace" ch:ClassHead "{" cnb:ClassNamespaceBody? "}" → ClassNamespace

%% ClassNamespaceBody can contain multiple methods, ...
ClassNamespaceBody+ → ClassNamespaceBody

%% method definition
FunctionDefinition → ClassNamespacesBody
%% static members
SimpleDeclaration → ClassNamespacesBody
%% nested namespace classes
ClassNamespace → ClassNamespacesBody
```

But this is not always as easy, because we have to propagate the attributes. It becomes tedious to copy and paste two rules that are mostly identical, but one does not allow something the first does. That is why we prefer to restrict existing grammar rules.

2.2.2 Restricting an existing grammar rule

To do so, we can use an attribute called *ok*. If set to *fail* on a term, the parse tree using that term will be removed from parse forest at disambiguation time.

```
ClassKey Identifier? → ClassHead
{attributes(disamb:
   root.ok = !ClassKey.key; if ?"test" then fail else id end
}
```

Unfortunately, we do not always have a direct access on the attributes we need. For example, a rule using *ClassHead* will not have access to the *ClassKey*'s key value. To have this attribute, we need to add a rule to the grammar that will manually propagate the attribute, and we are falling back to the first solution.

Chapter 3

Writing a Transformation

3.1 Syntax

We first need to describe how to write a transformation rule in STRATEGO. There are two ways to write such a rule. The first use the production constructor name and is called *abstract syntax*.

The other one directly uses C++ that is translated into the corresponding constructors.

3.1.1 Abstract Syntax

A *little* transformation using *abstract syntax* to replace every *return x;* where x is an integer to *return 42;* would be written that way:

```
JumpStatement(
  return(
    Some(
      Expression2(
        [ ConditionalExpression(
            LogicalOrExpression(
              LogicalAndExpression(
                InclusiveOrExpression(
                  ExclusiveOrExpression(
                    AndExpression(
                      EqualityExpression(
                        RelationalExpression(
                          ShiftExpression(
                            AdditiveExpression(
                              MultiplicativeExpression(
                                PmExpression(
                                  CastExpression(
                                    UnaryExpression(
                                      PostfixExpression(
                                        PrimaryExpression(
                                          Literal(IntegerLiteral(INTEGER-LITERAL(x)))
                                        )
))))))))))))))) ])))))

→

JumpStatement(
  return(
    Some(
      Expression2(
        [ ConditionalExpression(
            LogicalOrExpression(
              LogicalAndExpression(
                InclusiveOrExpression(
                  ExclusiveOrExpression(
```

```
                    AndExpression(
                      EqualityExpression(
                        RelationalExpression(
                          ShiftExpression(
                            AdditiveExpression(
                              MultiplicativeExpression(
                                PmExpression(
                                  CastExpression(
                                    UnaryExpression(
                                      PostfixExpression(
                                        PrimaryExpression(
                                          Literal(IntegerLiteral(INTEGER-LITERAL(42)))
                                        )
))))))))))))))) ])))))
```

But it is rather difficult to write such a rule. Moreover, adding or removing rules to the grammar can lead to a constructor name modification. This is intolerable in a context where we want to apply different transformations with grammar extensions at the same time.

3.1.2 Concrete Syntax

The other way to do it is to extend the STRATEGO language itself. This is done by creating a .meta file that contains the name of the grammar file.

For example:

```
Meta ([Syntax ("Ret42Cxx")])
```

Listing 3.1: In the file Ret42.meta

```
exports
  context-free syntax
    "|[" JumpStatement "]|"      -> \StrategoTerm { cons("ToTerm"), prefer }

%% [...]

exports
  variables
    x                            -> IntegerLiteral       { prefer }
```

Listing 3.2: In the file Ret42Cxx.sdf

In the transformation file, we are now able to write this:

```
|[ return x; ]| -> |[ return 42; ]|
```

This is easier to read and write than when using abstract syntax, but this is not perfect. As we are using C++ grammar inside STRATEGO, and C++ grammar is ambiguous, we sometimes need to disambiguate the rules by hand.

3.2 Transformation Pipeline

For now, the transformations were applied by using shell scripts and pipes, calling *parse-cxx* (the C++ parser generated), then the transformation program. This is a bad approach, because A | B executes B, even if A fail.

Worse, the exit code is the one of the last command. So if A fails and B returns 0, we will not be able to know that an error has occurred. Some shells like bash and ksh have an option *pipefail* to return the exit code of the first command returning a code not equal to 0. zsh and bash have a PIPESTATUS variable that contains all the return codes of the executed commands. There is

also a shell script by Cheusov (2007) that claims to support more shells. Anyway, this is not portable on any shell.

In addition, the TRANSFORMERS C++ parser and the STRATEGO programs need some environment variables like XTC_REPOSITORY (that defines the path to a file that contains a database of some STRATEGO files paths). It seems to be a better idea to use a STRATEGO program to do that, like it is done for *cxx-parser*. The *cxx-parser* program already call strategies on the tree, to disambiguate and remove the attributes. As some transformations might need the attributes (so they need to be called before attribute cleaning in *parse-cxx*), it seems to be a good idea to add an option to *parse-cxx* to execute the given transformation before *attr-clean*[1], and another to do it after *attr-clean*.

3.3 Example: The Class Namespaces

This transformation first have to remove "namespace class" and gets the classname. Then we can add the class name where it is needed.

3.3.1 Removing "namespace class"

In order to do this, we first need to define concrete syntax variables, to be able to match and get the terms we need.

```
module CnCxxVariables

imports
  ClassNamespaces

exports
  variables

    "cnb"[0-9]*[\']*     → ClassNamespacesBody? { prefer }
    "ck"[0-9]*[\']*      → ClassKey             { prefer }
    [xyz][0-9]*[\']*     → Identifier           { prefer }
```

Listing 3.3: CnCxxVariables.sdf

Now we can remove "namespace class":

```
UnClassNamespace : |[ namespace ck x { cnb } ]| → |ClassNamespacesBody?[ cnb ]|
```

Listing 3.4: class-namespaces.str

where *ck* is the class key (struct or class), *x* is the classname and *cnb* the class namespace body.

[1] attr-clean: program to remove all the attributes

3.3.2 Lookup

As the user can give a fully or partially qualified name for the class namespace name, we have to lookup into the environment table to find the right type to use for method return type and template.

First example:

```
namespace foo
{
  class bar
  {
    typedef int baz_t;
    baz_t my_method ();
  };
}

namespace class foo::bar
{
  // here baz_t must be replaced by foo::bar::baz_t
  baz_t my_method ()
  {
    // ...
  }
}
```

Second example:

```
namespace foo
{
  typedef int baz_t;
  class bar
  {
    baz_t my_method ();
  };
}

namespace class foo::bar
{
  // here baz_t must be replaced by foo::baz_t
  baz_t my_method ()
  {
    // ...
  }
}
```

Third example:

```
namespace foo
{
  typedef int baz_t;
  class bar
  {
    baz_t my_method ();
  };

  namespace class bar
  {
    // here baz_t must be replaced by foo::baz_t
    baz_t my_method ()
    {
      // ...
    }
  }
}
```

3.3.3 Adding the class namespace name

We now need to add the following rules to the meta file:

```
    "ret"[0-9]*[\']*     -> DeclSpecifierSeq?     { prefer } // function return type
    "fname"[0-9]*[\']*   -> Declarator            { prefer } // function name
    // attribute initializers in constructors
    "finit"[0-9]*[\']*   -> CtorInitializer?      { prefer }
    "fbody"[0-9]*[\']*   -> FunctionBody          { prefer } // function body
    "ftryb"[0-9]*[\']*   -> FunctionTryBlock      { prefer }
```

Listing 3.5: CnCxxVariables.sdf

And the transformation becomes:

```
cnReturnType(|x) : |DeclSpecifierSeq?[ ret ]|
              -> |DeclSpecifierSeq?[ ~DeclSpecifier:x'~ ~DeclSpecifier:ret~ ]|
    where
        <ns-lookup-name> (<get-attribute> (x, "disamb", "ns"),
                          x,
                          <get-attribute> (x, "disamb", "table"))
        x
        => x'

cnExpandName(|x) : |ClassNamespacesBody[ ret fname ftryb ]|
                -> |ClassNamespacesBody[ ret' fname' ftryb ]|
    where <cnFunction(|x)> fname => fname';
          <cnReturnType(|x)> ret => ret'

UnClassNamespace : |[ namespace ck x { cnb } ]| -> |ClassNamespacesBody?[ cnb' ]|
    where <alltd(cnExpandName(|x))> cnb => cnb'
```

Listing 3.6: class-namespaces.str

The *where* clause allows us to apply the rule cnExpandName on the class namespace body.

The cnExpandName is a rule that tries to match the method and the static declaration and then call other rules for method return type, method name and template if the method is parameterized.

In cnReturnType we need to add the class namespace name before the return type, but only if the return type is not a simple type specifier (like int, float, ...).

Conclusion

Even if TRANSFORMERS has some limitations (no preprocessor, bugged template disambiguation (Seine, 2008)), it is usable for simple transformations, like the class namespace extension. Some changes have been done in the TRANSFORMERS pipeline and in the attributes to be able to do such transformations.

We probably want now to start to implement some of the transformations like:

- those that were described by Desprès (2004) to help other LRDE projects,
- program slicing (Quèze, 2008),
- removing useless includes,
- ...

Implementing Centaur (the framework that simplifies the transformations) is still needed to make TRANSFORMERS usable by everyone (Raud, 2008).

References

Anisko, R., David, V., and Vasseur, C. (2003). Transformers: a C++ program transformation framework. Technical Report 0310, LRDE.

Borghi, A., David, V., and Demaille, A. (2006). C-Transformers — A framework to write C program transformations. ACM Crossroads, 12(3). http://www.acm.org/crossroads/xrds12-3/contractc.html.

Cheusov, A. (2007). pipestatus for unix/posix shell (http://sourceforge.net/projects/pipestatus/).

Chiba, S. (1997). Openc++ (http://opencxx.sourceforge.net/).

Daniel, C. and Grund, H. (2003). Proposed addition to c++: Class namespaces (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1420.pdf).

Desprès, N. (2004). C++ transformations panorama. Technical report, EPITA Research and Development Laboratory (LRDE).

GCC Manual (GCC extensions). Gcc manual (gcc extensions) (http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/C_002b_002b-Extensions.html).

Hipp, R. (2000). Lemon (http://www.hwaci.com/sw/lemon/lemon.html).

O. S. Bagge, M. Haveraaen, E. V. (2000). Codeboost (http://www.codeboost.org/).

Quèze, F. (2008). Toward a concrete application of Transformers. Technical report, EPITA Research and Development Laboratory (LRDE).

Raud, C. (2008). Centaur: A generic framework simplifying C++ transformation. Technical report, EPITA Research and Development Laboratory (LRDE).

Seine, W. (2008). C++ template disambiguation with Transformers attribute grammars. Technical report, EPITA Research and Development Laboratory (LRDE).

Smith, J. (2000). Cmm (http://www.op59.net/cmm/).

Spinczyk, O. (2005). Aspectc++ (http://www.aspectc.org/).

Stroustrup, B. (1994). The Design and Evolution of C++. ACM Press/Addison-Wesley Publishing Co.

Urban, M. (1999). Puma (http://ivs.cs.uni-magdeburg.de/~puma/).