Preprocessing C++ : Substitution and Composition

Edward D. Willink
Racal Research Limited,
Worton Drive, Reading, England
+44 118 923 8278
Ed.Willink@rrl.co.uk

Vyacheslav B. Muchnick
Department of Computing,
University of Surrey, Guildford, England
+44 1483 300800 x2206
V.Muchnick@ee.surrey.ac.uk

ABSTRACT

Problems with the lexical substitution mechanism of the C preprocessor are well known. We resolve these problems with a new syntax-based substitution mechanism for C++ based on meta-variables and meta-functions. Implementation of these meta-concepts in a C++ style leads naturally to meta-expressions and then meta-statements and a generalisation of C++ syntax. We show how meta-compilation renders the C preprocessor redundant, and offers a more intuitive and powerful programming language in which pattern solutions can be provided and Aspect-Oriented programming practiced. In a companion paper [21] we go one stage further, putting the concepts together in the context of meta-classes where meta-programming and reflection are supported at compile-time.

Keywords

Object-Oriented Language; Preprocessor; C++; Meta-Level; Composition; Weaving; Aspect-Oriented Programming; Pattern Implementation

1 INTRODUCTION

C++ [3] is a very widely used industrial strength language with support for Object Orientation. The popularity of C++ is in part attributable to a very high degree of compatibility with C [2]. Portability and run-time efficiency are some of C's and consequently C++'s attractions to programmers. Efficiency is achieved in C by providing programming constructs that are relatively low level. Efficiency is preserved in C++ by using a restrictive form of Object Model that enables C++ to resolve at compile-time what many other Object-Oriented languages resolve at run-time. Programmers as well as compiler writers seek to trade compile-time for run-time activity. In order to improve run-time efficiency, a programmer may identify better algorithms, select a more efficient compiler, and structure code to exploit the good, or avoid the poor, characteristics of that compiler. In order to improve compile-time, there appear to be few alternatives, although different coding styles and appropriate management of include file dependencies and compilation unit sizes can show surprising benefits. Dramatic improvements in programming time may be achieved when an automatic code generator such as lex or yacc is applicable, or when an application generator such as a GUI builder is suitable. For many more mundane programming applications, the structure of the code is in some way predictable, but not of sufficient complexity to justify the development of a custom code generator. For these applications, the programmer is forced to perform more editing than should be necessary, exploiting whatever tools are available. Cpp (the C preprocessor) was for a long time the main tool available to C and C++ programmers. Prior to the introduction of templates, it was standard practice to use some very large preprocessor macros to define generic classes for containers. Templates provide a powerful solution to problems that can be characterised by the requirement to define a single type or function. For many other problems, Cpp remains the only alternative.

Compatibility with C required C++ to preserve Cpp, although its limitations as a programming tool have long been recognised. While C++ introduces a number of new syntaxes that eliminate some traditional uses of Cpp, other uses remained. The power and complexity of Object Orientation and the increasing use of simple patterns [8] or idioms [7] considerably increases the need for programmers to program at compile-time and as a result Cpp is perhaps more, rather than less, important to C++ than to C.

Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems, TOOLS EE 99, Blagoevgrad, Bulgaria, June 1999.

In this paper we very briefly review the well-known deficiencies of Cpp, and then propose a new form of substitution that integrates with, rather than operates in spite of, C++. We find that each Cpp problem can be resolved by making some conventional C++ run-time characteristic available at compile-time, and that by following through the natural corollaries of each enhancement, we end up with an extension of C++ that supports meta-programming. There is too much ground to cover to describe all the improvements in one paper, and so we stop short of meta-programming and leave that to a companion paper [21].

The extended syntax is a pure-superset of C++, with no new reserved words and no changed semantics. The new substitution syntax is achieved through two new operators using the `$` and `@` tokens. All other changes are realised by generalising existing syntax, and giving a useful meaning to declarations that have no valid meaning in C++. A practical implementation of the extended syntax is provided by FOG (a Flexible Object Generator). FOG is a source to source translator converting extended syntax source files to conventional C++. In order to preserve compatibility, FOG contains a full implementation of Cpp, however there is no need to use it, since all of its facilities have been made redundant.

This paper is organised as follows. In Section 2 we review the problems with the traditional Cpp substitution mechanism, and then propose a better mechanism using compile-time meta-variables and meta-functions. In Section 3 we replace Cpp conditionalisation by using meta-variables and meta-functions in meta-expressions whose evaluation may be resolved by a meta-statement at compile-time. In Section 4 we generalise C++ syntax to remove the need for duplication between declarations and definitions, and find that this enables programs to be structured to suit programming requirements and supports the weaving required by Aspect Orientation [12]. Then in Section 5 we compare our approach with some related work, and consider its relevance to other languages in Section 6.

2 SUBSTITUTION

2.1 Traditional Functionality

Cpp supports the definition of and replacement of object-like and function-like macros.

An object-like macro associates an identifier with a replacement sequence of preprocessor tokens.

```
#define OCTAL_CASES \
        '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7'
```

The replacement tokens replace the macro identifier wherever it occurs.

```
switch (c)
{
    case OCTAL_CASES: /* ... */ break;
    case ALPHABETIC_CASES: /* ... */ break;
    case '%': /* ... */ break;
}
```

Substitution occurs at a very low level, offering the programmer considerable flexibility. In the above example, the replacement sequence apparently omits an initial `case` keyword and a trailing `:` token. The missing tokens accompany the instantiation. Readers may form their own opinion as to whether the unusual definition leads to a dangerously obscure or aesthetically pleasing implementation of the `switch` statement.

A function-like macro associates an identifier and a list of formal parameters with a replacement sequence.

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Invocation of the macro provides the actual arguments that replace the formal parameters in the replacement sequence. Macro substitution is simple but prone to accidents. Substitution occurs at a lexical level and so ignores any logical structure that may be present in the source code. The intention that `MAX` returns an expression from a pair of expression arguments is only realised when the actual usage is appropriate. The apparently redundant parentheses in the macro definition avoid the surprising evaluation that would otherwise result from the interpretation of

```
price + MAX(current_rate, fixed_rate) * commission
```

as

```
(price + current_rate) > fixed_rate) ? current_rate : (fixed_rate * commission)
```

A different set of problems occurs if one of the arguments has a side-effect.

```
c = MAX(a++, ++b)
```

One of the arguments is evaluated twice, and so receives a double increment. If the first argument is greater, the result is obtained after one increment has occurred. It is unlikely that the program will function as required.

Perhaps the most serious problem with the preprocessor is that of name capture. All names occur in a single namespace and so every conventional use of a name that is defined as a macro can malfunction. For instance the enumeration

```
struct Options { enum { LEFT, RIGHT, UP, DOWN, MAX }; };
```

should operate quite satisfactorily with `Options::MAX` denoting the number of options. However the definition of the `Options` enumeration and the `MAX` macro will probably be provided in include files, and when both

include files are used, any reference to `Options::MAX` will fail. If the macro is defined before the enumeration, a syntax error will spring up in the enumeration. This form of error is obscure and confusing. It can appear to be intermittent since compilations that do not use both include files succeed. Novice programmers are confused. Experienced programmers may take a little time to detect the hand of the preprocessor.

The traditional functionality may be characterised as substitution by imposition. A macro definition is imposed upon all subsequent code.

2.2 Invited Substitution

We propose a new substitution mechanism, in which the substitution is invited rather than imposed. This incurs a lexical penalty through the introduction of a substitution operator, but cures the danger of unwanted code changes. The presence of a substitution operator also serves as a clear indication that a substitution is intended and is occurring.

Object-like substitution occurs as

```
$object_name                or      ${object_name}
```

and function-like substitution as

```
$function_name(args)        or      ${function_name(args)}
```

This cures the problem of unwanted substitutions.

The independence of macros from the underlying language is resolved by replacing the concepts of object-like and function-like macros by meta-variables and meta-functions, which are declared in a very similar fashion to conventional variables and functions.

```
auto number PI = 3.14159;
auto expression MAX(expression a, expression b) { a > b ? a : b }
```

Each meta-variable or meta-function or meta-function parameter is defined with a meta-type that corresponds to a terminal (e.g. `const`, `"string"` or `}`) or non-terminal (e.g. *declaration* or *statement*) of the C++ grammar[1]. The examples use the meta-types `number` and `expression`.

The `number` meta-type describes any value corresponding to a C++ *integer-literal*, *character-literal*, *floating-literal* or *boolean-literal*. The `expression` meta-type corresponds directly to an *expression* in the C++ grammar.

The `auto` keyword is totally redundant in C++. It is only permitted within function bodies, where its use may assist the reader in identifying local declarations. We therefore avoid introducing any new keywords by reusing the existing keyword to identify meta-functionality for declarations that do not appear within functions.

The similarities with normal C++ declarations extend to default parameter values, but not to overloading or exception specifications.

Associating a meta-type with each meta-function or meta-variable preserves the style of C++ declarations and enables the programmer to select the style of substitution. The meta-type determines the syntax of parsed arguments, removing potential ambiguity and improving error diagnosis.

Many implementations of the original Kernighan and Ritchie C preprocessor [9] performed character-based substitution. The replacement characters were inserted between the surrounding characters and the resulting character stream was then re-analysed. This offered considerable flexibility, but different implementations varied in their treatment of obscure recursions and encountered difficulties when a composite token such as `+=` arose as a result of character concatenation.

The ANSI C preprocessor changed to token-based substitution. The preprocessor identifies the tokens and substitution replaces a sequence of tokens. Composite tokens can only arise through explicit use of the `##` operator.

The new substitution mechanism supports syntax-based substitution. Syntax elements are inserted in place of the invocation. The original syntactical structure is preserved. The need for protective parentheses is therefore removed. However multiple substitution may still occur, so the problem of side effects is not resolved. Inline functions provide an adequate solution to such problems.

Although the new substitution is syntax-based, token-based substitution is also supported through use of the `token` meta-type. `token` is the most primitive and generic terminal of the C++ grammar. Every number, string, identifier or piece of punctuation such as `>>=` is a *token*.

```
auto token OCTAL_CASES[] =
    { '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7' }
```

1. In this paper we use the same style as in Appendix A of [15], which is substantially the same as [3]. *Italics* are used to denote non-terminals. A `fixed` width font is used for literal text and for meta-type names. The C++ grammar comprises both lexical and syntactical levels, and so strictly only the ASCII characters, the reserved words and punctuation sequences are terminals. In this paper we concentrate on the syntactical level and so refer to a lexical production such as a *string-literal* or *identifier* as a terminal rather than a non-terminal.

The declaration of `OCTAL_CASES` as a list (array) of tokens causes syntax-based substitution to occur at the token level and so provides token-based substitution. The use of a free format language grammar avoids the need for the \ continuation characters in the Cpp line-based grammar.

2.3 Concatenation and Stringizing

The character-based substitution of the K&R C preprocessor enabled composite tokens (normally an extended identifier) to be formed by causing the character sequences to abut.

```
begin/**/_and_/**/end                          // begin_and_end
```

An identifier could be converted to a string by substitution within a macro:

```
#define STRINGIZE(s) "s"
STRINGIZE(text)                                // "text"
```

In the ANSI C preprocessor, the change to token-based substitution and the requirement that a comment be replaced by a whitespace character lost this flexibility necessitating the introduction of the ## operator to request concatenation, and the # operator to support stringizing.

```
#define CONC3(a,b,c) a ## b ## c
CONC3(begin,_and_,end)                         // begin_and_end
#define STRINGIZE(s) #s
STRINGIZE(text)                                // "text"
```

The major syntax elements of C++ comprise whitespace, punctuation, characters, numbers, strings, identifiers and keywords. Characters and strings are surrounded by single or double quotes and are unambiguously distinct from surrounding text. Numbers, identifiers and keywords have no associated punctuation and so must be separated from other numbers, identifiers and keywords by punctuation or whitespace. Punctuation must also be separated by whitespace in those cases where an ambiguity could arise. A consequence of these practical constraints is that there are only two contexts where adjacent characters, identifiers, keywords, numbers or strings can occur without intervening punctuation or whitespace in valid C++ programs:

Adjacent strings (with or without intervening whitespace) are concatenated and treated as a single string.

```
"This " "is"" ""a" " single"" string"           // "This is a single string"
```

A string may occur between a keyword and an identifier in the case of a *linkage-specification*.

```
extern"C"size_t f;
```

although few programmers would choose to be so economical with whitespace in this case.

We may therefore define the following concatenation property (incurring only a trivial compatibility problem with respect to unusual formatting of a little used form of a *linkage-specification*):

A token sequence comprising a character, identifier, number or string followed, without any intervening whitespace, by an arbitrary mix of characters, identifiers, keywords, numbers and strings is concatenated using a character-based representation to yield an extension of the first token. If the resulting text does not correspond to a possible extension, the program is ill formed, and an error is reported.

This restores the flexibility of character-based substitution and eliminates the need for a special concatenation or stringizing operator. An empty string or character can be used to provide separation between elements that require concatenation.

```
begin""_and_''end                              // begin_and_end
```

An empty string (or character) can be used as a meta-cast for stringizing. The subsequent text acquires the string (or character) characteristics of the start of the sequence.

```
""text                                         // "text"
```

These examples show the new mechanism, but in a context where it would be simpler to write the intended text in the first place. The utility of concatenation only arises when one or more of the concatenated elements is a $ expression.

2.4 Scoping

Meta-variables and meta-functions are declared in a similar fashion to conventional variables and functions. This similarity extends to scopes and templates. Whereas preprocessor macros contribute to a single global namespace irrespective of the prevailing language context, meta-objects are declared as part of the language.

```
class Simple
{   /* ... */
    auto declaration StaticFlyWeight(identifier name, expression init = 0)
    {
        static const Simple& ${name}()
        {
            static const Simple staticInstance($init);
            return staticInstance;
        }
    }
};
```

```
class Smart
{   /* ... */
    auto declaration StaticFlyWeight(identifier name, expression init = 0)
    {
        static const Smart& ${name}()
        {
            static const auto_ptr<Smart> staticInstance(new Smart($init));
            return *staticInstance;
        }
    }
};
```

Each class provides an isomorphic `StaticFlyWeight` meta-function that returns a `declaration` comprising a static function declaration. The body of each function creates a static instance of its class as a local variable, using a construction procedure appropriate to the rest of the class implementation (not shown). This provides a solution to the FlyWeight pattern [8], that is applicable for concrete flyweights that can be enumerated at compile-time. Application code may define such flyweights as

```
class Application
{   /*...*/
public:
    $Simple::StaticFlyWeight(easy);
    $Smart::StaticFlyWeight(null);
};
```

Providing isomorphic implementations of the same meta-function avoids the invoker needing to understand the different construction protocols of each class. It is then possible to introduce an extra level of indirection and invoke as

```
auto identifier Naivety = Smart;
$${Naivety}::StaticFlyWeight(null);
```

It would of course be possible to use the preprocessor. However the preprocessor approach is unnatural and so programmers tend to avoid its use. Meta-functions fit within the context of the language and so provide a useful addition to a programmer's toolbox.

Defining a template for a generic case and specialisations for variations could also solve this particular example.

```
template <class T, T::InitType Init = 0>
class StaticFlyWeight
{
    const T& operator()() const { static const T iT(Init); return iT; }
};
template <>
class StaticFlyWeight<Smart, Smart::InitType Init = 0>
{
    const Smart& operator()() const
        { static const auto_ptr<Smart> iT(new Smart(Init)); return *iT; }
};
class Application
{   /*...*/
private:
    static const StaticFlyWeight<Simple> easy;
    static const StaticFlyWeight<Smart> null;
};
```

When more than a single declaration is required, the template approach fails, although the preprocessor approach may remain viable. For instance, three declarations are required to define a member variable (*name*), a protected set accessor (set_*name*) and a public get accessor (get_*name*). The following meta-function defines these declarations for a simple scalar type such as `bool`.

```
auto declaration_seq ScalarMember(identifier type, identifier name, expression init = 0)
{
    private[2] $type $name = $init[3];
    protected void set_${name}(const $type& aValue) { $name = aValue; }
    public const $type& get_${name}() const { return $name; }
};
```

Similar isomorphic meta-functions can be provided to cover alternate data types such as smart pointers.

Member variables and their standard accessors may then be defined as

```
class Application
{   /* ... */
    $ScalarMember(bool, is_valid, false);         // bool is_valid; ...
};
```

2.5 Semantic Meta-Functions

For a few applications it is very desirable to avoid the lexical cost of invited substitution.

2. The use of `private` as a *type-specifier* rather than `private:` as a prefix to a *member-specification* is a FOG extension that avoids invocation of a meta-function disrupting the prevailing access for subsequent declarations.
3. The use of an initializer for member variables is a FOG extension that avoids accidental omission of an initializer from a constructor. The specified initializer is incorporated into all non-copy constructors that do not provide an explicit initializer.

```
auto declaration persistent(declaration aDeclaration) { /* meta-function body */ }
```
invoked as:
```
$persistent(int aValue = 0);
```
lacks the clarity of an invocation such as:
```
persistent int aValue = 0;
```
An alternate form of meta-function declaration is therefore provided comprising a return meta-type, a braced syntax-template comprising a sequence of identifier, punctuation or parenthesised meta-parameter, followed by a braced meta-function body. The syntax-template must start with an identifier, which becomes a reserved word within the prevailing namespace. The semantic alternative that supports the more direct usage is:
```
auto declaration { persistent (declaration aDeclaration) } { /* meta-function body */ }
```
This declares a syntax-template comprising the *identifier* `persistent` followed by a *declaration* that may be accessed as `aDeclaration` in the subsequent meta-function body. For the application programmer, this appears to be a seamless language extension.

3 META-STATEMENTS

The C preprocessor supports conditionalisation using various forms of `#if` directive to test the state of a lexical expression. Meta-variables would be of limited utility if they too did not support conditionalisation.

Within function bodies, the C++ grammar involves a sequence of statements, which may comprise expressions, declarations or control statements. Outside functions the grammar involves only declarations. The C++ grammar is a little irregular, and the distinction between expressions and declarations may require careful analysis, and in some cases remains ambiguous. The ambiguity is resolved by definition in favour of interpretation as a declaration. It is therefore safe to extend the C++ grammar to allow expression and control statements outside of functions, and to apply the same resolution rule for ambiguities. This extension provides for meta-expressions and meta-statements that are evaluated at compile time.

We may therefore replace a preprocessor `#if` directive
```
class X
{   /* ... */
#if DEBUG               // Preprocessor directive executed at (pre-)compile time
    void debug_print_out(ostream& s) const;       // Conditionalised declaration
#endif
};
```
by an equivalent meta-statement
```
class X
{   /* ... */
    if ($debug)                       // Meta-statement executed at (meta-)compile-time
        void debug_print_out(ostream& s) const;       // Conditionalised declaration
};
```
In an object hierarchy, we may want to use smart pointers and an intrusive reference count [7]. The reference count functionality must be provided exactly once. We can define a meta-function to provide the intrusive reference count functionality and use a meta-variable to prevent multiple copies.
```
class ReferenceCount
{
private:
    mutable size_t _shares;
public:
    ReferenceCount() : _shares(1) {}
    ReferenceCount(const ReferenceCount&) : _shares(1) {}
    ReferenceCount& operator=(const ReferenceCount&) { return *this; }
    ~ReferenceCount() { /* ASSERT(_shares == 1); */ }
    bool annul() const { return (_shares == 1) ? false : (_shares--, true); }
    void share() const { _shares++; }
    auto declaration_seq install()
    {
        if (!has_reference_count)
        {
            auto number has_reference_count = true;
            private ReferenceCount _shares;
            public void annul() const
                { if (!_shares.annul()) delete ($This[4] *)this; }
            public void share() const { _shares.share(); }
            friend inline void annul(const $This *anObject)
                { if (anObject) anObject->annul(); }
        }
    }
};
auto number has_reference_count = false;
```

4. The built-in meta-variable `This` resolves to the current declaration scope, which is the `ReferenceCounted` class in this example. Use of the built-in variable avoids the need to pass the declaration scope as a formal parameter. It also avoids the need to specify redundant information.

The first few lines of the example define the `ReferenceCount` class, which encapsulates the required behaviour of the reference count member variable of a reference counted object. The counter is always constructed with a value 1, and is unaffected by assignment. The reference count is adjusted by `share()` and `annul()`. The latter method returns `true` if the reference counted object should continue to exist, `false` if the reference counted object should be destroyed. The destructor may contain an assertion that the reference count is 1 when the reference counted object destructs. The functionality is incomplete: additional declarations are required in the reference counted object. These extra declarations are provided when the reference counted object invokes the install meta-function as in

```
class ReferenceCounted /* ... */
{   /* ... */
    $ReferenceCount::install();
};
```

The meta-function starts with a conditional meta-statement to guard against double installation. When first invoked, the `has_reference_count` meta-variable is not defined in the class or any of its base classes and so the reference is resolved by the global definition on the last line of the example. This has the value `false`, and so the conditional succeeds. The `has_reference_count` meta-variable is then defined within the `ReferenceCounted` class with a `true` value. As a result, any reinvocation of the meta-function is suppressed by the conditional. The guard code is then followed by the definition of a member variable, two member functions and a friend function. The `_shares` member variable provides the run-time storage needed by the share count, and interacts via its constructors and destructors with the equivalent functions for the reference counted class. The `share()` member function simply delegates the member variable. The `annul()` member function completes the share counting protocol by deleting the reference counted object when the final share is removed. The `annul()` friend function just provides a more convenient destruction option avoiding the need to worry about null pointers.

Accidental double installation within the same class may seem unlikely for conventionally structured code, however when more than one meta-function is invoked as in

```
class ReferenceCounted /* ... */
{   /* ... */
    $NonIntrusiveList::install();
    $NonIntrusiveMap::install();
};
```

a multiple installation could occur indirectly. Accidental double installation within an inheritance tree can easily occur, but is trapped by the inheritance of `has_reference_count` with a true value. This would be very difficult to achieve with the preprocessor and cannot be achieved using templates.

An improved implementation eliminates the global variable, and guarantees a compile-time error when a reference counted object is destroyed in any way other than the use of `annul()`. This is unfortunately beyond the scope of this paper, but will appear in [22].

Using meta-functions, the single line `$ReferenceCount::install()` is all that is required to add share count functionality to an application class. We have therefore achieved the separation of concerns [1] that is the goal of Aspect Orientation [12].

4 COMPOSITION

Meta-functions and meta-variables have been introduced as outright extensions to C++. A few other incidental extensions have been mentioned in the main text or alluded to in footnotes. The earlier examples have involved only a single class, or at most a master class controlling a slave class. All code has been written so that it appears inline as part of the class interface. Many problems involve more than one class, and it is generally undesirable to abuse inline functions. It leads to code bloat and cache misses at run-time [5] and excessive include file dependencies at compile-time [13]. We therefore need to ensure that the use of meta-functions to generate declarations does not lead to unwarranted problems.

4.1 Splitting

Separation of function bodies from function declarations requires that their interfaces be specified twice, once without a body in an interface include file, and again with a body in an implementation file. The repeated interface uses a slightly different syntax (no `virtual` keyword, no default parameter initializers, but with a template and scope qualification).

This duplication is an inconvenience to programmers, it doubles the number of files and declarations to create and maintain, and introduces opportunities for inconsistency. The duplication also slightly discourages the factoring of repetitive code into preprocessor macros, since two macros must be provided, one to contribute to an interface and another to contribute to an implementation.

FOG is a source to source translator. It converts FOG source files (a C++ superset) into conventional C++ files. FOG does not require separate interface and implementation files. FOG accepts contributions from as many

source files as are provided, and then emits pairs of interface and implementation files for the appropriate classes[5]. There is no need for interface and implementation to be separated in FOG source files, and the potential inlining problems are resolved by providing the programmer with more control.

In the absence of explicit inlining information, very small[6] functions are inlined by FOG and other functions are not. Explicit inlining in the interface file occurs in response to the `inline` keyword. The converse `!inline` prevents inlining. An intermediate form of inlining within the implementation file only may be specified by `inline/implementation`.

Since FOG eliminates the need for distinct interface and implementations, FOG must also eliminate the syntactical differences. Specifying many non-trivial functions within the confines of a single class declaration would result in very unwieldy classes. Upward compatibility with C++ requires that both the interface and implementation style of definition be supported. FOG therefore generalises the syntax to allow declarations to use either an interface or implementation style.

C++ requires independent interface and implementation for an implemented pure virtual function:

```
class Application
{   /* ... *
public:
    virtual void function(int aValue = 0) = 0;
};
```

and

```
void Application::function(int aValue) { /* ... */ }
```

FOG supports a complete declaration in an interface style

```
class Application
{
    public virtual void function(int aValue = 0) = 0 { /* ... */ }
};
```

or in an implementation style

```
public virtual void Application::function(int aValue = 0) = 0 { /* ... */ }
```

4.2 Composing

This means that a class declaration is not closed. A class may be extended by the presence of other declarations defined in an implementation style, or just by the respecification of the class. The following three declarations

```
class Application
{
    public int _first_variable = 1;
};
/* ... */
public int Application::_second_variable = 2;
/* ... */
class Application
{
    public int _third_variable = 3;
};
```

therefore define a class with (at least) three member variables.

This is a total violation of one of the major principles of C++: the One Definition Rule, which simply stated requires exactly one declaration of each entity. FOG is a translator to C++, and so while the One Definition Rule must be satisfied by the generated C++, the translator may take a more liberal view for its sources. FOG implements a Composite Definition Rule, composing the class members, enumerators, array initializers and function bodies from many potentially very disparate sources to present a single composite definition to the subsequent C++ compilation.

4.3 Weaving

Soukup [13] makes a persuasive case for implementing patterns using pattern classes. A pattern class is just a grouping of functions templated by the types of each collaborator. The pattern class contains no member variables and is a friend of each collaborator. A pattern is used by invoking the static member function of the pattern class with the appropriate collaborator instances as parameters. Since the pattern class is a friend of each collaborator the function is free to peek and poke the working variables in the collaborator objects to perform the required actions. This approach achieves a very regular style of implementation, and has very beneficial effects in reducing include file dependencies. However the extensive use of friend declarations runs counter to normal programming practice.

Instantiation of a pattern using a pattern class requires instantiation of the pattern class, and insertion of friendship declarations and working variables into the collaboration classes. Instantiation of the pattern class is

5. Ensuring that multiple many-source-file to many-output-file translations leads to consistent results introduces some configuration management complexities that are beyond the scope of this paper.

6. A command line option defaulting to 10 non-whitespace lexical tokens.

readily resolved by conventional C++ template instantiation. Insertion of declarations into the collaborators could be performed by manual editing. [13] describes a custom preprocessor for the CodeFarms library that performs this insertion automatically, provided the programmer has left a hook in each class of the form.

```
class State
{
    MEMBER_State                // Hook to enable Cpp to insert arbitrary declarations
    /*...*/
};
class Town
{
    MEMBER_Town                 // Hook to enable Cpp to insert arbitrary declarations
    /*...*/
};
```

The custom preprocessor scans pattern class instantiations such as

```
WHOLE_PART(State,Town)          // Declaration that State has many Towns
```

to produce conventional Cpp definitions such as

```
#define MEMBER_Town \
    friend class WholePart<State,Town>; \
    State *_whole;
```

Multiple patterns are readily accommodated. The Cpp macro just grows.

This approach demonstrates that implementation of a particular pattern solution requires that declarations be injected into the code for collaborator classes. Only for the degenerate case of a pattern involving a single class can injection be avoided. In the terminology of Aspect Orientation [12], the declarations associated with each aspect (or pattern) must be woven together to create composite declarations acceptable to the C++ compiler. FOG supports this weaving eliminating the need for a custom preprocessor and for the preprocessor hooks to support the CodeFarms library.

```
template <class Whole, class Part>
auto declaration_seq WholePart::install()
{
    class $Whole
    {
        friend class $Static;
        private list<$Part> _parts;
        /* optional construction, delegations and destruction */
    };
    class $Part
    {
        friend class $Static;
        private $Whole *_whole;
        /* optional construction, delegations and destruction */
    };
}
$WholePart<State,Town>::install();
```

Invocation of the installation function just adds the required friend declaration and member variable to each collaborator class identified by the template parameters. The semantics of meta-function execution involve replacement only of formal parameters in the scope of the meta-function, before returning the declarations in the meta-function body for interpretation within the invocation scope. The meta-function apparently has no formal parameters. It actually has four. All meta-functions have two built-in formal parameters `Static` and `Dynamic` corresponding to the declared scope of the meta-function and the actual scope, which differs if invoked for a derived class. In addition, each template parameter is also a formal parameter. The usage of `$Whole` rather than `Whole` therefore ensures that a replacement occurs before the body is returned to the calling scope where `Whole` may be undefined or differently defined.

The example shows only the minimum to activate the pattern solution. Additional declarations could enforce appropriate construction and destruction protocols, and provide delegation so that users are unaware that a pattern class is in use.

It is not necessary to use Soukup's pattern class approach, although it has a pleasant symmetry. Installation can be organised with respect to a dominant collaborator, probably the Whole class in this case. It is then only necessary to perform code injection into the other collaborators. However whatever approach is adopted, a pattern solution with more than one collaborator class requires manual editing to spread the pattern solution or automatic code injection.

5 RELATED WORK

Stroustrup has highlighted the inadequacies of the C preprocessor in [14], where he calls for its eventual demise. Very little work has been published on practical alternatives. Straightforward lexical alternatives such as m4 suffer from many of the same problems by operating independently of the underlying language. Operation in collaboration with C++ involves tackling the challenge of C++ syntax, which is difficult to parse and difficult to extend. Werther [19] provides a sensible proposal for a completely new C++ syntax using more conventional syntactical styles like Ada or Pascal. Within a clean syntactical framework, it would be much easier for

researchers to examine alternative syntaxes, and it would be possible for a meta-program to perform syntax extension.

Cheatham [6] and Leavenworth [10] introduced the concepts of a syntactic macro whose arguments have types corresponding to parts of the language grammar, and Vidart [16] gave these concepts a sound and efficient foundation with an Abstract Syntax Tree (AST) interpretation.

Weise [18] applies these ideas to ANSI C and exploits a Lisp-like backquote to support a pattern template for substitution, avoiding the need for extensive call trees to rearrange the AST. Weise's approach is very much an extension to ANSI C introducing new keywords and 9 lexical operators. Our approach in FOG is in some ways very similar to Weise's, however by giving existing concepts a compile-time meaning, and retaining a degree of consistency on all the corollaries, FOG achieves a notation that is more compact, supports character-, token- or syntax-based substitution with only two new lexical operators (`$` and `@`) and no new reserved words. Where Weise needs backquotes and an explicit return statement to activate source-like declarations, FOG just treats all declarations as the return. In FOG all concepts are put into a C++ Object Oriented perspective.

The term code weaving has been popularised in conjunction with AspectJ [11], which supports the programmer in partitioning code to locate each aspect in distinct source units. AspectJ then weaves the various contributions together to create Java classes that incorporate multiple aspects. A form of code weaving is provided by the custom preprocessor for the CodeFarms pattern classes [13]. A more general purpose capability is provided by SNIP [20], although the enhancements effectively introduce two new languages.

Although few researchers have extended C++ with a view to resolving pre-processor problems, many have introduced variants to support the requirements of concurrency and persistence. [23] provides articles by many of the leading parallel processing researchers, whose activities have been pursued in C++. The different approaches demonstrate that language extension can occur at three different levels:

> Library classes and run-time environments can be developed without any language or compiler changes. FOG's increased capabilities at compile-time provide library developers with more options, perhaps supporting simpler interfaces, reduced requirements for user support code, or stronger compile-time detection of protocol violations.
>
> Translators that recognise one or two extra reserved words require development of the translator but do not affect the underlying compiler. FOG provides an ability to introduce custom extensions to C++, enabling many of the characteristics of custom translators to be achieved by a general-purpose translator.
>
> New forms of code generation necessitate significant revision to both language and compiler. FOG offers very little to applications that need to rewrite the basic compiler.

The original patterns book [8] has provoked considerable interest and a number of specialised conferences and workshops. Patterns and their evolution concern identifying appropriate contexts, the forces that influence problems and different solutions that resolve particular combinations of forces. The difficulty of actually fielding a reusable implementation of a particular solution is rather neglected. Vlissides participated in the development of GUI tool to generate code for patterns automatically [4], but then expresses considerable reservations in his book [17]. It is indeed difficult to conceive of an automatic generator that will have sufficient flexibility to balance all the conflicting forces and select the appropriate cookbook implementation. There is rightly much generality and ambiguity in pattern descriptions. However programmers regularly reuse particular pattern implementations with which they are familiar, and providing an improvement over cut and paste for such reuse is essential. Aspect Orientation has recognised that an aspect (such as fault diagnosis, or synchronization) of an application is beneficially isolated from other aspects, and that the realisation of the code for an aspect, often using patterns, cuts across class boundaries. Integration of code partitioned according to aspects requires a weaver to combine class declarations together. AspectJ [11] provides this for Java. FOG provides weaving, a replacement preprocessor and meta-programming for C++.

6 OTHER LANGUAGES

In this paper we have been resolving deficiencies with the C preprocessor and the paper concentrates on C++. The solutions are applicable to C as well, although while C has structures it does not have member functions, so introduction of meta-functions to more than the global scope would not be appropriate.

The C preprocessor and other lexical tools can be used for other languages, where ignorance of the underlying language avoids language dependence but also incurs problems from lack of language compatibility. Java is perhaps particularly relevant in this context. Java has replacements for preprocessor constructs such as `#include`, and provides a concept of portability that undermines much traditional usage of `#if`. However the elimination without replacement of `#define`, `#if` and `#line` appears retrograde. It causes some unwarranted problems for Java code generators such as AspectJ [11].

Programming involves repetition at many levels, and programmers naturally seek to factor the repetition into some parameterisable reusable construct, which may be a loop, subroutine, class, template, macro, file or library. Omission of any of these capabilities simplifies a language, but limits the programmer's or the

program's efficiency. Some form of macro to perform lexical processing and meta-programming is therefore beneficial to all languages, although the precise syntax must be carefully chosen to fit within the traditional style of each language.

7 PERFORMANCE

FOG is a research tool fully described in [22]. It is available as source, Solaris and NT binaries from

http://www.ee.surrey.ac.uk/Research/CSRG/fog

As might be expected of a research tool, the efficiency falls well below that of a production compiler. FOG is relatively complete, covering all but a few dark corners of the C++ syntax.

There is no run-time cost associated with the use of FOG, because the source translation just offers the programmer alternative and more compact ways of writing the same program. The opportunity to exploit a package of reusable meta-functions may incur small penalties as often occurs through the use of general purpose solutions rather than locally optimised implementations.

Compile time efficiency is dependent upon the way in which FOG is used. FOG comprises a full C++ parser and so there is a possible doubling of compile time as the source is parsed once by FOG and again by the C++ compiler. The exact ratio depends on how many extraneous declarations are processed by each stage as a result of include files. Careless organisation of build procedures can result in a very high penalty comparable to that of instantiating templates one at a time with early C++ compilers. Subsystem by subsystem organisation of compilations can substantially reduce the meta-compilation costs of using FOG in proportion to the compilation costs, although it may be that a full subsystem recompilation will occur for most changes to the subsystem or its dependencies.

Since the parsing within FOG duplicates much of the activity of the subsequent compiler, a much more efficient system can be envisaged in which the results of the translation are communicated to the compiler directly.

8 CONCLUSION

A new substitution syntax has been proposed for C++, used to render Cpp redundant, resolve its long standing limitations and prepare the way for compile-time meta-programming.

Unwanted token replacement is avoided by performing replacement by invitation rather than imposition.

Incorrect substitution caused by the surrounding context is avoided by the use of meta-types to drive syntax-based substitution.

Independence from the language and the need for backslash continuations are avoided by defining all behaviour as part of the main language.

The problems of a single namespace for Cpp macros are resolved by using C++ namespaces and scopes.

Concatenation and stringizing have been implemented using a simple concept of adjacency without the need for any additional operators.

Extending C++ so that statements occurring outside of functions denote meta-execution provides conditionalisation.

C++ syntax has been generalised to remove the distinction between and need for distinct interfaces and implementations. This has been shown to allow code to be organised to suit the programmer allowing declarations to be woven in support of Aspect Oriented Programming.

These ideas are developed further in the companion paper where the compile-time capability of the meta-level is extended to support meta-programming and static compile-time reflection.

9 ACKNOWLEDGEMENTS

The authors thank Brian Hillam for a diligent reading of the manuscript. The first author is grateful to Racal Research Ltd. for providing the time and resources to pursue this research.

10 REFERENCES

[1] Aksit, M. Composition and Separation of Concerns in the Object-Oriented Model. ACM Computing Surveys, vol. 28A, no. 4es, 148, December 1996.
http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/al48-aksit/al48-aksit.html

[2] ANSI X3.159-1989. American National Standard for Information Systems - Programming Language - C. American National Standards Institute, 1990.

[3] ANSI X3J16/96-0225. Working Paper for Draft Proposed International Standard for Information Systems - Programming Language - C++. American National Standards Institute, 1996 (also known as Committee Draft 2 and now adopted as the Standard). http://www.maths.warwick.ac.uk/cpp/pub/dl/cd2/CD2-PDF.tar.Z

[4] Budinsky, F.J., Finnie, M.A., Vlissides, J.M., and Yu, P.S. Automatic Code Generation from Design Patterns. IBM Systems Journal, vol. 35, no. 2, 151-171 1996.
[5] Carroll, M., and Ellis, M. Designing and Coding Reusable C++. Addison-Wesley, 1995.
[6] Cheatham, T.E. The Introduction of Definitional Facilities into Higher Level Programming Languages. Proceedings of the 1966 Fall Joint Computer Conference, AFIPS, vol. 29. 623-637, 1966.
[7] Coplien, J.O. Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992.
[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
[9] Kernighan, B.W., and Ritchie, D.M. The C Programming Language. Prentice Hall, 1978.
[10] Leavenworth, B.M. Syntax Macros and Extended Translation. Communications of the ACM, vol. 9, no. 11, 790-793, November 1966.
[11] Lopes, C.V., and Kiczales, G. Recent Developments in AspectJ. In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98. In Workshop Reader of the European Conference on Object-Oriented Programming(ECOOP), LNCS ?, Springer-Verlag, June 1998.
http://wwwtrese.cs.utwente.nl/aop-ecoop98/papers/Lopes.pdf
[12] Mens, K., Lopes, C., Tekinerdogan, B. and Kiczales, G. Aspect-Oriented Programming Workshop Report. In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97. In Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1357, Springer-Verlag, June 1997 http://www.parc.xerox.com/spl/projects/aop/ecoop97/aop-ecoop97-proceedings.pdf
[13] Soukup, J. Taming C++: Pattern Classes and Persistence for Large Projects. Addison-Wesley, 1994.
[14] Stroustrup, B. The Design and Evolution of C++. Addison-Wesley, 1994.
[15] Stroustrup, B. The C++ Programming Language. Third edition. Addison-Wesley, 1997.
[16] Vidart, J. Extensions Syntactiques dans une Contexte LL(1). Thése pour Obtenir le Grade de Docteur de Troisiéme Cycle, 1974.
[17] Vlissides, J. Pattern Hatching: Design Patterns Applied. Addison-Wesley, 1998.
[18] Weise, D., and Crew, R. Programmable Syntax Macros. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, vol. 28, no. 6, 156-165, June 1993.
[19] Werther, B., and Conway, D. A Modest Proposal: C++ Resyntaxed. ACM SIGPLAN Notices, vol. 31, no. 11, 74-82, November 1996.
[20] Wild, F. Instantiating Code Patterns. Dr. Dobb's Journal, 72,74-76,88-91, June 1996.
[21] Willink, E.D., and Muchnick, V.B. Preprocessing C++ : Meta-Class Aspects. Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems, June 1999. http://www.ee.surrey.ac.uk/Research/CSRG/fog/FogToolsEE2.{gz,pdf,ps,zip}
[22] Willink, E.D. Meta-compilation for C++. PhD Thesis, Department of Computing, University of Surrey, (in preparation) 1999. http://www.ee.surrey.ac.uk/Research/CSRG/fog/FogThesis.{gz,pdf,ps,zip}
[23] Wilson, G.V., and Lu, P. Parallel Programming Using C++. MIT Press, 1996.