An Object-Oriented preprocessor fit for C++

E.D.Willink and V.B.Muchnick

Abstract: C++ retains the ANSI C preprocessor, although its limitations have been widely recognised. We describe FOG, a meta-compiler for a super-set of C++, that provides replacement preprocessing and introduces static meta-programming, while preserving the spirit of C++. We show how implementation of preprocessor functionality in an Object-Oriented style eliminates unnecessary replication from practical C++ programs, and supports recent Object-Oriented Programming developments to a much greater extent than existing tools.

1 Introduction

Cpp, the C preprocessor, has always been an essential accompaniment for the C language. Apart from some minor rationalisation for the ANSI C standard [1], Cpp has survived unchanged as an important part of C++ [2]. Stroustrup, in [3], identifies elimination of the preprocessor as a major goal for C++, devoting the final chapter to a discussion of its weaknesses, and identifying some remedies that C++ provides. In the final paragraph, Stroustrup writes: “I’d like to see Cpp abolished. However the only realistic and responsible way of doing that is first to make it redundant, ...”. Alternative preprocessors such as m4 can be used for C++, but they also operate independently of the underlying language.

A preprocessor supports reconfiguration of source text at compile time using techniques such as file inclusion, conditional compilation and text replacement. Cheatham [4] identified three kinds of macro replacements:

- Text macros (text replaced by text - as exemplified by Cpp macros)
- Computational macros (text replaced by the result of a computation - as exemplified by inline functions and templates)
- Syntax macros (text replaced by the syntax tree representing a linguistically consistent construct)

The Flexible Object Generator (FOG) renders Cpp redundant, introducing syntax macros that integrate with C++ and solve the problems associated with text macros. Where Cpp performs lexical manipulation without regard to context, FOG provides extensive meta-level facilities and has a full understanding of C++ declarations.

Programmers exploit whatever tools are available to reuse ideas and avoid repetition: subroutines and classes can encapsulate some forms of reusable functionality; templates, particularly when used imaginatively, can provide reusable solutions to many more problems [5]. However when a problem of reuse is, or is perceived to be, insoluble within the programming language, programmers must resort to extra-lingual approaches. Lexical pasting with the preprocessor is inelegant and error prone, but to be preferred over abandoning reuse and replicating code.

Use of simple patterns [6] or idioms [7] leads to very predictable coding sequences. Many of these cannot be captured by a single C++ construct and so paradoxically C++ programs often use the preprocessor more than their C predecessors. Even with the assistance of the preprocessor, it is difficult to represent patterns that cut across multiple classes, and so the pattern used for design is lost from the implementation [8].

In this paper we show how a relatively simple subset of the FOG meta-level facilities may be used for practical applications without requiring an understanding of the underlying meta-concepts. Space does not permit more than a hint of how further use of the meta-level concepts can support weaving and Aspect-Oriented Programming [9]. A description of the meta-concepts may be found in [10], and an AOP example in [11].

We start by showing how the basic facilities of Cpp are replaced, using very simple examples, that are gradually reworked as more powerful facilities are described and exploited. We then examine a more typical example, before reviewing related work and relevance to other languages.

2 Traditional Preprocessing

2.1 Lexical substitution

Lexical substitution enables common definitions to be shared, given sensible names, and factored out if alternative definitions are needed in different contexts. When used responsibly, this leads to a considerable improvement in code quality, and is one of the main reasons for the widespread use of the preprocessor. However it is very easy for unfortunate substitutions to occur, and the presence of all names from all header files in a single name space is a source of many problems.

IEE Proceedings online no. 20000573

DOI: 10.1049/ip-sen:20000573

Paper first received 17th September 1999 and in revised form 29th March 2000

E.D. Willink is with Racal Research Limited, Worton Drive, Reading RG2 0SB, UK
E-mail: ed.willink@rrl.co.uk

V.B. Muchnick is with the School of Electronic and Electrical Engineering, Information Technology and Mathematics, University of Surrey, Guildford GU2 5XH, UK
E-mail: v.muchnick@ee.surrey.ac.uk

C++ has removed the need for many substitutions by the introduction of initialised `consts` and scoped enumerations. However, even where these are appropriate, the need for a non-integral type may defeat C++ enhancements.

Problems with Cpp substitution stem from the single namespace and from forceful substitution irrespective of context. Resolution of the namespace problem in FOG will be dealt with later. The problem of over-enthusiastic substitution is resolved by changing to a policy of substitution by invitation, rather than substitution by imposition. In FOG, instantiation of the definition of `NAME` is invited by `$NAME`, with the fallback of `${NAME}` when subsequent characters could cause an unwanted meaning. The increased safety incurs the cost of the trigger characters to invite the substitution. These characters are not too out of place in a cryptic language such as C. The syntax should be familiar to Unix shell or make programmers.

2.2 Name concatenation

Name concatenation is useful for generating a new name derived from some stem. Thus an implementation of the NullObject pattern [12] may automatically define a Null class derived from its AbstractObject by suffixing Null to the class name of the AbstractObject.

```
class AbstractObjectNull
     : public AbstractObject
{ /*...*/ };
```

This can be realised directly in FOG, where unseparated identifiers and literals (numbers, strings and characters) are concatenated[1].

```
class ${ABSTRACTOBJECT}Null
     : public $ABSTRACTOBJECT
{ /*...*/ };
```

Cpp provides the ## concatenation operator that can only be used within a macro:

```
#define NULLOBJECT_INTERFACE(ABSTRACTOBJECT) \
     class ABSTRACTOBJECT ## Null \
       : public ABSTRACTOBJECT { /*...*/ };
```

2.3 String conversion

It is sometimes necessary, particularly for diagnostic purposes, to use a name as both an identifier and a string.

```
const char *Class::class_name() const
     { return "Class"; }
```

This may be expressed directly in FOG, exploiting concatenation of an empty string to perform a lexical cast, since the result of a concatenation is of the same kind as the first contribution.

```
const char *${CLASS}::class_name() const
     { return ""$CLASS; }
```

Cpp provides the # operator for use within macros.

```
#define CLASS_NAME_IMPLEMENTATION(CLASS) \
   const char *CLASS::class_name() const \
       { return # CLASS; }
```

2.4 Text replacement

The preprocessor `#define` directive is used to define object-like macros

```
#define PI 3.14159
```

and function-like macros

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

supporting usage as

```
a = max(sin(2*PI*f),0.5)
```

The replacement text is an arbitrary sequence of preprocessor tokens that are substituted without regard to context. Errors, particularly in nested definitions, are difficult to diagnose, because substitution occurs before any language interpretation is applied; few compilers or debuggers support tracing back to the source once substitution has occurred. Long definitions require the use of backslashed continuation lines, which are inconvenient and unreliable to edit or read. Readability is further impaired by the need to use parentheses to guard against the possibility of accidental association problems.

FOG provides a meta-level where conventional run-time concepts can be used at (meta-)compile time. Meta-variables replace object-like macros and meta-functions replace function-like macros. Meta-variables and meta-functions are declared and typed in a very similar way to normal C++ variables and functions, save for the new use of the `auto` keyword and the introduction of meta-types:

```
auto double PI = 3.14159; // Meta-variable

auto expression max(expression a, expression b)
{                          // Meta-function
    $a > $b ? $a : $b;
}
```

for use as

```
a = $max(sin(2*$PI*f),0.5)
```

The `auto` keyword is almost totally obsolete in C++, where `auto` is only permitted within functions. `auto` is reused outside of functions in FOG to declare meta-functionality. Readers may choose to pronounce `auto` as `meta`, throughout this paper.

The meta-types correspond to the basic kinds of token (`identifier`, `string` and `character`), the numeric types (`bool`, `double`, `int` and etc.) and also to productions such as `declaration` and `expression` from the C++ grammar [2].

Use of meta-types enables the parser to ensure that arguments are passed and returned compatibly, and to diagnose errors more helpfully. When appropriate, conversions between the basic kinds are performed automatically.

Substitution within the meta-function replaces each invocation by its corresponding argument expression[2].

The simple meta-function implementation of `max` solves the parenthesis problem, works for arbitrary types but remains prone to side effects. The invocation

```
$max(a++, b++)
```

will result in one argument receiving a double increment just as in Cpp.

2.5 Conditional compilation

Conditional compilation is essential to support a variety of configuration options, often to resolve distinctions between different operating systems. It may be appropriate to define

```
static const char *temp_path = "/tmp/";
```

for use under Unix whereas NT might require

```
static const char *temp_path = "C:\\Temp\\";
```

FOG elevates C++ run-time statements such as `if ... else ...` for use at the meta-level, so that the selection may be made using an apparently conventional test:

```
auto if ($UNIX)
  static const char *temp_path = "/tmp/";
else
  static const char *temp_path = "C:\\Temp\\";
```

[1] Obscure syntax incompatibilities requiring spaces in `extern "C"` and around `and` are described in [13] .

[2] Substitution is syntax-tree-based, rather than token-based as in the ANSI C preprocessor, or character-based as in the K&R preprocessor.

Cpp provides line-oriented conditional directives that mark-up rather than form part of the source text:

```
#if defined(UNIX)
  static const char *temp_path = "/tmp/";
#else
  static const char *temp_path = "C:\\Temp\\";
#endif
```

C++ statements occur only within functions. The use of the `auto` prefix is therefore redundant in the above example.

2.6 Other directives

A description of the replacements for other Cpp directives may be found in [13]. In summary, FOG provides a more integrated form of `#error`, a more disciplined context for `#pragma` and a form of `#include` that solves the problems of multiple inclusion. `#line` is not replaced.

3 Object-oriented preprocessing

The facilities described above provide consistent replacement for Cpp behaviour. Most of the extensions could be regarded as extensions to C rather than C++. Reviewing and generalising the facilities within the context of C++ leads to a much more powerful programming environment in which predictable program structures can be coded effectively.

3.1 Scopes

Meta-variables and meta-functions may be scoped and inherited, and meta-statements may occur within declaration scopes.

Revisiting the conditional compilation example of Section 2.5 from an Object-Oriented perspective, we find no need for conditional compilation. The characteristics of each configuration option may be packaged as meta-variables (and meta-functions) of a (meta-)class[3].

```
class OsTraits_Abstract
{
  auto bool NT = false;    // default value
  auto bool UNIX = false;
  //...
};
```

```
class OsTraits_NT : public OsTraits_Abstract
{                          // derived class
  auto bool NT = true;     // overriding value
  auto string temp_path = "C:\\Temp\\";
  //...
};
```

```
class OsTraits_Unix : public OsTraits_Abstract
{
  auto bool UNIX = true;
  auto string temp_path = "/tmp/";
  //...
};
```

`OsTraits_NT` may be configured as the implementation of `OsTraits`, by specifying the value of `OS` on the FOG command line

```
fog ... -D OS=NT ...
```

and using the built-in meta-function

```
auto string std::get_cpp(string macroName)
```

to access it from

```
class OsTraits
 : public OsTraits_$std::get_cpp("OS") {};
```

thereby creating the equivalent declaration

```
class OsTraits : public OsTraits_NT {};
```

This maps the required configuration to `OsTraits`, so that an operating system specific file may be defined using the temporary path by

```
const char *fileName =
       $OsTraits::temp_path "results.dat";
```

This is then resolved at compile-time to

```
const char *fileName = "C:\\Temp\\results.dat";
```

Having isolated the configuration in separate classes and an associated header file, a new operating system can be supported by providing a prefix file characterising the new system and invoking it with an appropriate command line. Existing source files need no change. This could be achieved directly using multiple layers of name substitutions with C preprocessor, but it never is. Modularisation is much easier when supported by the programming environment. This cannot be achieved using C++ templates, which lack the ability to perform string manipulations.

3.2 Compilation model

C++ supports a two stage translation process involving multiple independent compilations followed by a link editing stage to produce a final executable. The independent compilations are consistent provided the One Definition Rule [2 (§3.2)] is observed. Simply stated, this rule requires that a declaration in one compilation must not have a different meaning in any other. In practice, placing declarations in header (interface) files, which are included by each compilation session that requires them, usually satisfies the One Definition Rule.

From the perspective of a compiler writer, the One Definition Rule is very useful, if not essential. From the perspective of the programmer, the One Definition Rule is very inconvenient. Declarations must be provided twice, once in the interface file and again in the implementation file. Declarations cannot be freely interleaved. In more serious applications, a conflict arises between language constraints and the programmer's need to organise code to suit algorithmic or functional perspectives. Code has to be organised to suit the compiler. Patterns cannot be preserved in the code [8] and Aspect-Oriented Programming [9] is not readily supported.

A preprocessor for C++, that performs its processing prior to compilation, can bridge the gap between the organisational requirements of the programmer and the integrity requirements of the compiler. FOG operates in this way using an augmented compilation model as shown in Fig. 1.

The centre and right hand sides show the conventional C++ compilation model. Interface files provide the declarations to be shared by independent compilations, which produce object files to be linked together with libraries to produce an executable. (The complexities of static construction and template instantiation are conveniently hidden by the 'Linker2' activity.) Meta-compilation adds the extra stages on the left hand side. The conventional C++ interface and implementation files are generated by one or more meta-compilations from source files (the forward arrows) and from frozen interfaces (the reverse arrows). Sources may be shared between meta-compilations, and a single meta-compilation may generate any number of interfaces and/or implementations.

Clearly the One Definition Rule must still be respected by the interface and implementation files fed to the compiler. However a more relaxed Composite Definition Rule can now be imposed on the source files. Simply stated, the composite meaning of all like declarations must

[3] In FOG, every class and built-in type has an identically named meta-class, so for the purposes of this paper classes and meta-classes may be considered equivalent.

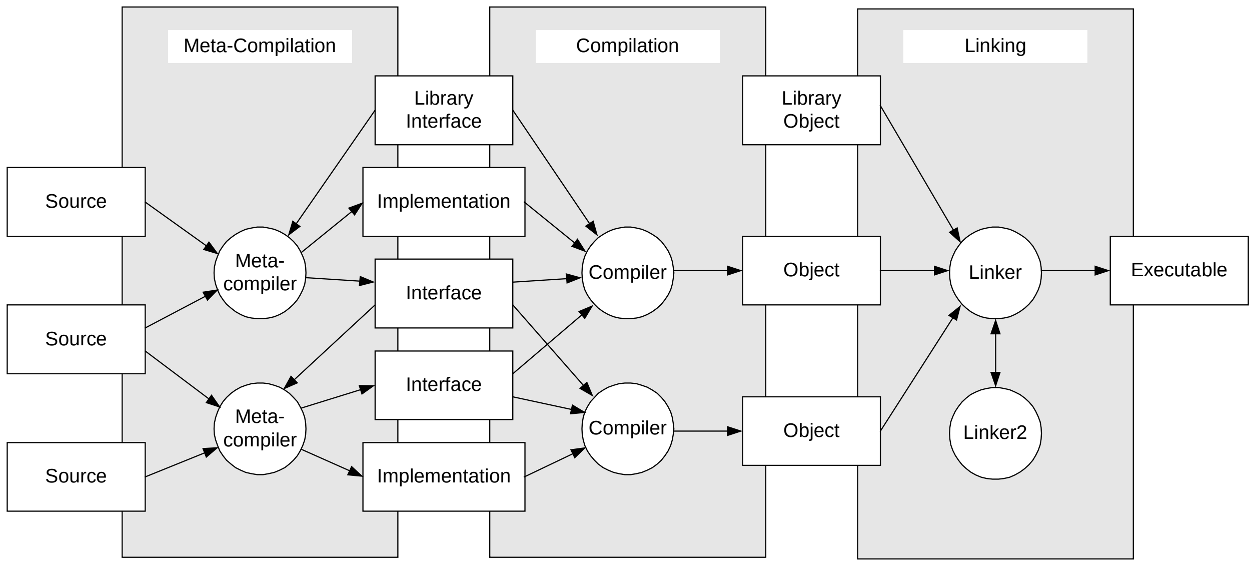

Fig. 1 *Meta-compilation model*

be the same in each meta-compilation. The composite meaning is explained in Section 3.4.

3.3 Joint interface and implementation

Introduction of a meta-compiler that synthesises interface and implementation files eliminates the need for independent interface and implementation declarations. It is appropriate to generalise C++ declarations to remove the distinction between interface-specific and implementation-specific declarations. This generalisation turns out to be almost entirely semantic, since the C++ grammar already permits an interface-specific keyword such as `virtual` to accompany a *function-definition*[4]. It is only necessary to allow an *access-specifier* (e.g. `protected`) as part of a *decl-specifier* (the type part of a declaration), and to permit a full *id-expression* (e.g. `Scope::name`) where previously only an *identifier* was allowed.

Programmers may then use an implementation style of declaration for parts of interfaces

```
public typedef size_t Class::SizeType;
```

or provide complete implementations in interfaces:

```
class Class
{
  protected virtual void f(int x = 0) = 0
     { std::cout << x; }
public:
  static double y = 0;
};
```

A complete solution to the `class_name()` example from Section 2.3 may now be captured by the single meta-function

```
auto declaration ClassName()
{
  public virtual !inline
     const char *class_name() const
       { return ""$Scope; }
};
```

which can be invoked as

```
class NamedClass
{
  $ClassName();
};
```

The reserved meta-variable `Scope` refers to the prevailing scope, avoiding the need to pass it as a parameter.

The negated keyword `!inline` ensures that the function body is not inlined. Similarly `!static` would provide for explicit rather than default programming intent.

The single meta-function invocation generates the equivalent C++ interface

```
class NamedClass
{
public:
  virtual const char *class_name() const;
};
```

and implementation

```
const char *NamedClass::class_name() const
     { return "NamedClass"; }
```

This requires a pair of macros when implemented using Cpp.

```
#define CLASS_NAME_INTERFACE() \
  virtual const char *class_name() const;
#define CLASS_NAME_IMPLEMENTATION(CLASS) \
  const char *CLASS::class_name() const \
     { return # CLASS; }
```

and a corresponding pair of invocations one from the interface

```
CLASS_NAME_INTERFACE()
```

and one from the implementation

```
CLASS_NAME_IMPLEMENTATION(NamedClass)
```

3.4 Composition

In C++, multiple declarations are an error. In FOG, multiple compatible declarations are composed; only incompatible declarations are an error. Space does not permit more than a superficial exposition of the composition rules.

Composed declarations merge their components, and so a variable qualified with `static` carries the `static` with it when merged with another variable that has no `static` specification, but provokes an error message if merged with a `!static`.

Overloaded function declarations compose independently. Default values may be repeated but may not conflict.

Arrays and enumerations extend to accommodate all contributions. Duplicate initialisations must match. Holes

[4] The italicised terms correspond to productions in the C++ grammar [2].

in arrays are zero filled. The GNU C [14] extension[5] is supported so that sparse arrays can be defined and composed.

```
bool is_prime[] = { [2] true, true, [5] true,
                    [7] true, [11] true };
```

The constructor initialisers for a particular constructor are composed and must not conflict. Unspecified initialisers for non-copy constructors are obtained from member variable initialisers. For example, code to support an error handling aspect may add a member variable with a default initialiser:

```
public bool Class::_error_generated = false;
```

A constructor independently added in support of some other aspect

```
Class::Class(PersistenceManager&) /*...*/;
```

provides the requisite initialisation.

Classes expand to encompass all distinct member declarations, with repeated declarations composed recursively.

Function (and constructor) bodies are composed by concatenating code contributions within named regions, which are in turn concatenated to form the overall function body. The regions named `entry` and `exit` typically provide for variable declaration and initialisation and a return statement, ensuring a predictable structure. Regions named `pre` and `post` provide code to operate before or after the default `body` region of the function. Function definitions are extended to support a declarative scope within which regions are prefixed by their name.

```
public bool Manager::do_it()
:{               // Start of declarative scope
  entry { bool exitStatus = true; };
  exit { return exitStatus; };
};
```

defines a framework for a composed function. A return variable is initialised in the `entry` region, and returned by the `exit` region. With the framework in place, code concerned with a particular aspect may contribute code to the function:

```
private Aspect Manager::_aspect;

public bool Manager::do_it()
{
  if (!_aspect.do_it())
     exitStatus = false;
}
```

FOG weaves the contributions together to produce the equivalent C++ declarations:

```
class Manager
{
private:
  Aspect _aspect;
public:
  bool do_it();
};

bool Manager::do_it()
{
  bool exitStatus = true;
  if (!_aspect.do_it())
     exitStatus = false;
  return exitStatus;
}
```

Readers who have programmed extensively with a macro assembler may recognise that the ability to extend classes, function code regions, enumerations and arrays at will gives each declaration space the attributes of a program section.

It is possible to define a meta-function that performs extension of an enumeration and a text array so that numeric and text declarations are automatically synchronised.

```
auto declaration NamedEnum(identifier aName)
{
  public enum Enum { $aName };
  public static const char *names[] =
                               { ""$aName };
}
```

Invocation as

```
class Colours
{
  $NamedEnum(RED);
  $NamedEnum(GREEN);
  $NamedEnum(BLUE);
};
```

provides successive entries for `Colours::Enum` and corresponding entries for `Colours::names[]`, as if the user had typed:

```
class Colours
{
public:
  enum Enum { RED, GREEN, BLUE };
  static const char *names[];
};
```

and

```
const char *Colours::names[] =
     { "RED", "GREEN", "BLUE" };
```

The conversion of a single name such as RED into multiple interleaved declarations cannot generally be achieved using the preprocessor or C++ templates.

3.5 Derivation rules

There are many idioms that require entirely predictable code to be provided by derived classes in order to comply with a protocol defined by a base class. The `class_name()` method of Section 3.3 provides one example. In C++, a declaration applies to the scope for which it is specified. In FOG, this scope is referred to as the root scope for that declaration. An optional derivation rule specifies how that declaration may be automatically redefined in the inheritance tree of scopes that derive from the root scope. Refining the example from Section 3.3

```
auto declaration ClassName()
{
  public virtual !inline
    const char *class_name() const
  :{
     derived(true) { return ""@Scope; };
  };
};
```

A declarative scope has been introduced to prefix a derivation rule to the function body. The predicate of `derived(true)` is always true and so the declaration is applied throughout the entire inheritance tree, that is at the root scope and all derived scopes.

The change of substitution operator from `$` to `@`, changes the evaluation time. `$` is an early substitution operator, evaluated when source tokens are first parsed to create a potential declaration in its associated root scope, at which point `Scope` resolves to the root scope. `@` is a late substitution operator, evaluated when a potential declaration becomes an actual declaration in its eventual scope, at which point `Scope` resolves to the actual scope. (If the `$` operator were used in the example, all derived scopes would return the name of the root scope.)

Derivation rules can apply to the declaration of any entity. Michael Tiemann provided a solution [3] to the problem of providing a mnemonic name for the primary base class

[5] A [*constant-expression*] preceding an array initializer specifies the array index to be initialized.

```
class foreman : public employee {
  typedef employee inherited;
  //...
  void print();
};
```

```
class manager : public foreman {
  typedef foreman inherited;
  //...
  void print();
};
```

enabling a derived class to refer to its base class mnemonically as `inherited` rather than explicitly.

```
void manager::print()
{
  inherited::print();
  //...
}
```

In FOG, the entire hierarchy of `typedef`s can be expressed by a single declaration.

```
private typedef @Super employee::inherited
  :{ derived(!is_root()); };
```

This provides a `typedef` for all derived classes. The derivation predicate inhibits the declaration at the root, where the built-in meta-variable `Super` may have no valid resolution for the primary base class.

The Prototype pattern [6], virtual constructor, or cloning idiom [15] is also provided very easily using a derivation rule. The conventional approach requires that a `clone` method be defined for every non-abstract class in an inheritance hierarchy

```
class ConcreteClass /* ... */
{
     /* ... */
     virtual RootClass *clone() const;
};
```

```
RootClass *ConcreteClass::clone() const
     { return new ConcreteClass(*this); }
```

This requires the programmer to manually weave the code in to every class. This is potentially error prone and costs at least one line per interface and one line per implementation file of every class. Using FOG, the idiom can be fully defined by a meta-function:

```
auto declaration Prototype()
{
  public virtual $Scope *clone() const = 0
  :{
     derived(!is_pure())
       { return new @{Scope}(*this); };
  };
}
```

The `!is_pure()` derivation predicate specifies that the declaration contributes code to all derived classes that have no pure virtual functions.

Instantiation requires a single line in the base class that defines the protocol. No code is required in derived classes.

```
class Base
{
  $Prototype();
};
```

(`$Scope` may be changed to `@Scope` to use the derived type as the return type.)

These two examples demonstrate FOG at its most advantageous: one line in the base class guarantees protocol observance and replaces one or more lines in each derived class. A more realistic example will now be examined.

4 A Real Example

One of the activities of a compiler involves selection of appropriate machine instructions (such as ADD or MOVE) to implement the program, usually represented by a tree of Abstract Syntax Tree nodes [16]. An effective approach to solving this problem involves a Bottom-Up Rewrite System [17], which searches the tree from the leaves upwards identifying the lowest cost solution that has each node covered exactly once by a machine instruction. The tree may then be rewritten in terms of the selected machine instructions. In order to support multiple target architectures, alternative instruction sets must be supported. Implementation of this diversity is assisted by the use of a Bottom-Up Rewrite Generator to transform a description of each machine instruction into the form needed for an efficient tree search. An example of this form of generator is lburg that forms part of the lcc C compiler [18].

lburg is a compact C program comprising just three files. lburg.c has 690 lines and 4652 (non-comment, non-whitespace preprocessor) tokens. lburg.h has 66 lines and 259 tokens. Additionally gram.y is a 19 rule, 37 state yacc parser grammar.

lburg supports single dispatch architectures (such as SPARC). An enhanced version was required in order to support less conventional processor architectures, and so a highly Object Oriented C++ rewrite was undertaken using reference counting and smart pointers to share common partial instructions. The resulting program was substantially larger, due to the extra declarations for encapsulated C++ classes, rather than the original free access to structure elements, and due to the added functionality. Preprocessor macros were used extensively to factor out common declarations.

A further revision to exploit FOG without any other change of functionality forms the basis of the following comparison. An implementation based on the use of preprocessor macros is compared with an implementation using meta-functions, meta-variables and derivation rules.

Use of FOG reduced the token count by 14%, from 20250 to 17500. The per-class reduction varied between 9 and 48%. The larger reductions occurred in small classes, where the benefits of derivation rules and simplification of interface and implementation declarations were most apparent.

A reduction in token count is an easily measured reduction in programming effort. Less easily measured are the more aesthetic improvements of better modularity, improved expression of programming intent, and automatic compliance with programming protocols. A pair of short before/after extracts are provided in the Appendix for readers to make their own judgements. The code is complete save for the removal of 4 functions whose lexical structure exactly duplicates functions that remain. Code for this example is chosen because it is shortest, and so demonstrates the changes more clearly. Providing the large number of unaffected function body lines from a more typical module would not provide extra insight. Space does not permit the definitions of the preprocessor macros or meta-functions to be shown. The two are of comparable lexical size, the meta-function has a higher token count through the use of `$` operators and meta-type names, but a lower token count through the use of more appropriate facilities. The meta-functions are modular, having fewer interdependencies than the preprocessor macros, and more readable through the use of more conventional structuring and the elimination of back-slash continuation lines.

The original preprocessor macros almost vanish completely. The `CUSTOM_RTTI` support is provided automatically by derivation. The remaining six macros supporting smart pointers are all subsumed by

`MapOfSmartPointerSpecialisations`. Other meta-functions such as `Mutate` just implement simple idioms.

5 Related Work

Although extensively criticised [3], relatively little work has been published on alternatives to the C preprocessor [1]. Weise [19] provides a review of earlier work, and describes the use of a syntactic macro, which has fully-typed argument and return values. Invocation of the macro occurs within the context of a parse tree, so there is no opportunity for the unpleasant side effects that occur with conventional macros that lack sufficient parentheses. The usage of fully meta-typed arguments for meta-functions in FOG is influenced by Weise, however the FOG notation is more compact and supports both character- and syntax-based replacement. Weise introduces nine extra lexical operators, requires an explicit return, and produces examples that are unpleasant and sometimes difficult to read. FOG introduces only two extra lexical operators (`$` and `@`) and by treating the entire meta-function body as the return achieves a simpler substitution model. Weise considers only ANSI C, whereas FOG revisits the concepts with an Object-Oriented and meta-level perspective.

Researchers in many fields have chosen to use C++, but found it inadequate for their purposes. There are therefore many domain specific extensions to C++, just some of which are discussed below.

A low level understanding of object layout is necessary for persistent storage of objects in databases or for marshalling objects, whether for signalling between nodes in a communication network, or distribution across nodes in a parallel processor. Wilson and Lu [20] provides extended articles by 16 of the leading research teams using C++ for parallel processing. Some researchers used only library classes and run-time support code, and so remain entirely within the normal confines of the C++ language. Others introduce language extensions, which are variously implemented as translators to C++, or modified C++ compilers. MPC++ [21] exploits meta-level facilities to support an extended syntax within a ‘standard’ C++ compiler. Many of the C++ extensions appear unnecessary and some authors recognise that more imaginative use of C++ facilities, particularly those not readily available at the start of their research could have reduced the need for divergence.

Domain specific extensions, when fully integrated with C++, can provide a clean solution to the domain problem. Many extensions are poorly integrated because of the size and complexity of C++ and so provide little more than a research tool. Many of the problems dealt with in a domain specific fashion can be resolved in a domain independent fashion by using the meta-level programming facilities of FOG. However FOG meta-programming is restricted to declarations and so the more radical changes of C** [22] in which data parallel semantics are introduced to expressions could certainly not be addressed.

The concepts of meta-classes were first defined for Smalltalk. Languages such as CLOS have been extended with a Meta-Object Protocol (MOP) [23]. Even Java has a meta-class object for every class. C++ has rather lagged behind, perhaps through a mismatch of the run-time characteristics of traditional MOPs and the statically compiled philosophy of C++, perhaps through the compiler writer’s desire to prevent further explosion of language complexity. FOG provides statically compiled meta-functionality, which can be used to define customised run-time meta-functionality. It is difficult to answer the critique that C++ is too large, and that adding meta-functionality is an enhancement too far. However it is also difficult to avoid recognising that the absence of meta-functionality is restrictive for some domains and an inhibition to reuse for all.

C++ has no compile-time meta-level capabilities. Chiba [24] describes an extended form of C++ called Open C++, that allows library developers to write code to analyse class layout and so ensure that persistence can be resolved transparently. Meta-classes are used as adjuncts of the normal C++ class structures, with a wide variety of reserved meta-functions available for re-implementation to enable the parse tree to be adjusted during the compilation process. This provides considerable power, but because Open C++ programming is very closely related to the compiler internals, this approach is not suitable for normal programming. FOG also uses meta-classes, but every C++ class or built-in type is a FOG meta-class, and the meta-code creates declarations by using declaration statements directly, with the result that the facilities available at the meta-level in FOG are very similar in syntax and behaviour to those already available in C++. FOG is not able to rewrite expressions in the same way that Open C++ does, but there may be no need to. Relevant expression terms can be encapsulated within inline functions and templates, which FOG provides the ability to manipulate at compile-time.

The use of patterns has provoked considerable interest since the original patterns book [6], although most attempts to represent patterns in code are informal. Soukup addresses the problems of implementing patterns in [8].

Aspect-Oriented Programming [9] has recognised that an aspect may cut across many objects, requiring the contributions to each class from each aspect to be woven together.

Generative Programming [25] seeks to configure re-usable components, making extensive use of C++ templates to perform the compile-time configuration. This approach works well for components that can be fully defined by a single type or function. FOG complements the template approach by providing a compile-time flexibility to configure multiple declarations as well as single types or functions.

6 Other languages

This paper has concentrated on C++, where meta-level facilities are very limited and preprocessing is more extensively used than in other languages. In C++, an efficient program is produced at compile-time avoiding the need for run-time activities. Preprocessing contributes to resolving problems at compile-time. The compile-time processing available from static meta-programming is compatible with the C++ philosophy. It is less relevant and probably unwelcome for languages such as Smalltalk or CLOS that have a tradition of run-time flexibility.

Java has its origins in C++, and also seeks to resolve problems at compile-time although run-time performance is less critical. The absence of a preprocessor in Java eliminates one of the programmer’s options. The lexical and meta-level concepts in this paper could certainly be applied to Java, where the cleaner syntax and existing meta-classes could be exploited.

Significant revision of the syntax is necessary to make the concepts compatible with languages such as Ada 95 or Eiffel, for which some form of meta-level programming would be beneficial, although a cryptic C/C++ lexical style would be out of place.

7 Performance

FOG is available as source, NT and Solaris binaries from [26]

The current implementation of FOG is at the level of a research tool and so the efficiency falls well below that of a production compiler.

The potential efficiency depends on the way in which the meta-compiler is used. If used to meta-compile on a class by class basis, the need to understand the context of declarations may incur a significant penalty, similar to the costs of instantiating templates one at a time without precompiled headers. If used on a subsystem by subsystem basis, the costs are amortised and can be similar to those of a conventional compiler.

A meta-compiler operating as an independent preprocessor duplicates much of the parsing and semantic analysis of the subsequent compiler. Using an appropriate intermediate representation to communicate between meta-compilation and compilation can reduce the meta-compilation overhead.

8 Conclusion

A meta-compiler has been described that makes the C preprocessor redundant

- concatenation and substitution replace # and ##
- meta-variables and meta-functions replace object-like and function-like macros
- meta-statements replace `#if`

The meta-compiler extends C++ but preserves its essential characteristics

- costs are incurred only at compile-time
- syntax-based substitution respects program structure
- meta-types define meaning and support error diagnosis
- meta-classes define the scope of meta-variables and meta-functions
- meta-statements support meta-programming at compile-time
- derivation rules support and enforce inheritance hierarchy protocols

Although each extension is not new to computer science, it is their combination that provides the unique ability to avoid repetition in practical examples such as cloning.

Finally the composition of declarations frees the programmer from the constraints imposed by the One Definition Rule, providing the flexibility to encapsulate solutions to patterns as meta-functions and to weave together the declarations from each separate concern of an Aspect Oriented Program.

9 Acknowledgements

The authors thank an anonymous referee for suggesting a clearer exposition and Matthew Ellis, Brian Hillam and Mike Quinlan for reading drafts diligently. The first author is very grateful to Racal Research Ltd. for providing the time and resources to pursue this research.

10 References

1 ANSI X3.159-1989.: ‘American National Standard for Information Systems - Programming Language - C’. (American National Standards Institute, 1990)
2 ISO/IEC 14882:1998(E).: ‘International Standard - Programming Languages - C++’. (American National Standards Institute, 1998)
3 STROUSTRUP, B.: ‘The design and evolution of C++’. (Addison-Wesley, 1994)
4 CHEATHAM, T.E.: ‘The introduction of definitional facilities into higher level programming languages’. Proceedings of the 1966 Fall Joint Computer Conference, AFIPS, 1966, **29**. pp. 623-637
5 VELDHUIZEN, T.: ‘Using C++ template metaprograms’. C++ Report, 1995, **7**, (4), pp. 36-43. (http://extreme.indiana.edu/~tveldhui/papers/meta-art.ps)
6 GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J.: ‘Design patterns, elements of reusable object-oriented software’. (Addison-Wesley, 1995)
7 COPLIEN, J.O.: ‘Advanced C++ programming styles and idioms’. (Addison-Wesley, 1992)
8 SOUKUP, J.: ‘Taming C++: Pattern classes and persistence for large projects’. (Addison-Wesley, 1994)
9 KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J-M., and IRWIN, J.: ‘Aspect-Oriented Programming’. Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP’97, June 1997, (Springer-Verlag, LNCS 1241) (http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Kiczales-ECOOP97/for-web.pdf)
10 WILLINK, E.D., and MUCHNICK, V.B.: ‘Preprocessing C++: Meta-class aspects’. Proceedings of the Eastern European Conference on Technology of Object Oriented Languages and Systems, June 1999, Blagoevgrad, Bulgaria, pp. 136-149, (ISBN 954-90484-1-1). http://www.ee.surrey.ac.uk/Research/CSRG/fog/FogToolsEE2.pdf
11 WILLINK, E.D., and MUCHNICK, V.B.: ‘Weaving a way past the C++ one definition rule’. Position paper for the Aspect-Oriented Programming Workshop at ECOOP’99, June 1999. (http://www.ee.surrey.ac.uk/Research/CSRG/fog/AopEcoop99.pdf)
12 MARTIN, R.C., RIEHLE, D., and BUSCHMANN, F.: ‘Pattern languages of program design 3’. (Addison-Wesley, 1997)
13 WILLINK, E.D.: ‘Meta-compilation for C++’. PhD Thesis, Computer Science Research Group, University of Surrey, January 2000 (http://www.ee.surrey.ac.uk/Research/CSRG/fog/FogThesis.pdf)
14 STALLMAN, R.M.: ‘Using and porting GNU C’. (Free Software Foundation, Inc., Cambridge MA, 1998. Available as part of the gcc-2.8.0 distribution.)
15 STROUSTRUP, B.: ‘The C++ programming language’. (Addison-Wesley, 1997) 3rd edn.
16 AHO, A.V., SETHI, R., and ULLMAN, J.D.: ‘Compilers: Principles, techniques and tools’. (Addison-Wesley, 1986)
17 PROEBSTING, T.A.: ‘BURS Automata generation’. ACM Transactions on Programming Languages and Systems, 1995, **17**, (3), pp. 461-486
18 FRASER, C.W., and HANSON, D.R.: ‘A retargetable C compiler: Design and implementation’. (Benjamin/Cummings 1995)
19 WEISE, D., and CREW, R.: ‘Programmable syntax macros’. Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, 1993, **28**, (6), pp. 156-165
20 WILSON, G.V., and LU, P.: ‘Parallel Programming using C++’. (MIT Press, 1996)
21 ISHIKAWA, Y., HORI, A., TEZUKA, H., MATSUDA, M., KONAKA, H., MAEDA, M., TOMOKIYO, T., NOLTE, J., and SATO, M.: ‘MPC++’, in WILSON, G.V., and LU, P.: ‘Parallel Programming using C++’. (MIT Press, 1996), pp. 429-464
22 LARUS, J.R., RICHARDS, B., and VISWANATHAN, G.: ‘C**’, in WILSON, G.V., and LU, P.: ‘Parallel Programming using C++’. (MIT Press, 1996), pp. 297-342
23 KICZALES, G., DES RIVIÉRES, J., and BOBROW, D.G.: ‘The art of the metaobject protocol’. (MIT Press, 1991)
24 CHIBA, S.: ‘A metaobject protocol for C++’. Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA’95, ACM SIGPLAN Notices, 1995, **30**, (10), pp. 285-299
25 CZARNECKI, K., and EISENECKER, U.W.: ‘Synthesizing objects’. Proceedings of the 13th European Conference on Object-Oriented Programming ECOOP’99, June 1999, Lisbon, Portugal, pp. 18-42, (Springer, LNCS 1628)
26 http://www.ee.surrey.ac.uk/Research/CSRG/fog

11 Appendix

11.1 Original interface file

```
#ifndef ENTRY_HXX
#define ENTRY_HXX
#include <Burg.h>
#include <Id.hxx>                            // A smart string class
#include <Object.hxx>
#include <ReferenceCount.hxx>
#include <SmartPointer.H>

class Entry : public Object
{
  CUSTOM_RTTI_DECLARATION(Entry, Object)
  REFERENCE_COUNT_DECLARATION(Entry)
  NULL_OBJECT_DECLARATION(Entry)
private:
  const Burg& _burg;
  const IdHandle _id;                        // Handle for a smart string
private:
  Entry(const Entry&);                       // No copy
  Entry& operator=(const Entry&);            // No assign
protected:
  Entry();
  Entry(Burg& aBurg, const Id& anId);
public:
  const Burg& burg() const { return _burg; }
  const Id& id() const { return *_id; }
  virtual Term *is_term();
  const Term *is_term() const { return ((Entry *)this)->is_term(); }
  virtual void mark_reachable();
  virtual ostream& print_this(ostream& s) const;
};
#endif
```

11.2 Original implementation file

```
#include <Entry.hxx>
#include <Burg.hxx>
#include <MapOfSmartPointer.H>

CUSTOM_RTTI_IMPLEMENTATION(Entry, Object)
REFERENCE_COUNT_IMPLEMENTATION(Entry)
NULL_OBJECT_IMPLEMENTATION(Entry)
SMART_POINTER_IMPLEMENTATION(Entry)
MAP_OF_SMART_POINTER_IMPLEMENTATION(Entry)

Entry::Entry() : _burg(Burg::null_object()) {}

Entry::Entry(Burg& aBurg, const Id& anId) : _burg(aBurg), _id(anId) { aBurg.add_entry(*this); }

Term *Entry::is_term() { return 0; }
void Entry::mark_reachable() {}
ostream& Entry::print_this(ostream& s) const { return s << _id; }
```

11.3 Revised FOG code, with use of FOG extensions italicised

```
using "Burg.fog";                            // Improved form of #include.

class Entry : public Object
{
  using/interface "Burg.h";                  // Need a #include <Burg.h>
  $NoCopy();                                 // Entry(const Entry&);
  $NoAssign();                               // Entry& operator= (const Entry&)
  $Mutate();                                 // Entry& mutate() const { return *(Entry *)this; }
private:
  const Burg& _burg = Burg::null_object();
  const IdHandle _id;

protected:
  !inline Entry() {}                         // Uses default initialiser value

public:
  const Burg& burg() const { return _burg; }
  const Id& id() const { return *_id; }
  virtual Term *is_term() { return 0; }
  const Term *is_term() const { return mutate().is_term(); }
  virtual void mark_reachable() {}
  virtual ostream& print_this(ostream& s) const { return s << _id; }

};

$MapOfSmartPointerSpecialisations(Entry);

protected Entry::Entry(Burg& aBurg, const Id& anId) : _burg(aBurg), _id(anId)
     { aBurg.add_entry(*this); }
```