FOG: A Meta-compiler for C++ Patterns

EDWARD D. WILLINK[1] AND VYACHESLAV B. MUCHNICK[2]

[1] *Racal Research Limited, Worton Drive, Reading, Berks, England*
Ed.Willink@rrl.co.uk

[2] *Department of Electronic and Electrical Engineering, University of Surrey, Guildford, England*
V.Muchnick@ee.surrey.ac.uk

SUMMARY

Modern assemblers provide powerful macro facilities to enable programmers to create their own high level constructs. In contrast, many high level languages have at best very limited macro capabilities. The macro facility provided by the C preprocessor has been almost universally derided. We examine the limitations imposed by the lack of macro facilities for pattern programming, object persistence and object communication. We propose a meta-compiler (a translator from a superset C++ language) that provides macros integrated into the language as meta-functions and meta-variables of meta-classes. We show how a variety of simple facilities provided at the meta-level can allow patterns to be expressed directly, substantially reduce the lexical size of C++ programs and their consequent maintenance cost, offer a more versatile programming environment, and provide facilities needed to support persistence and communication of objects.

KEY WORDS: object-oriented language; pattern; C++; metaclass; metaobject; composition

INTRODUCTION

Assembler languages lack any high level programming constructs, and so macros are provided to enable programmers to create their own constructs. High level languages provide high level constructs directly and do not generally provide macros. Elimination of macros provides for a more uniform language, but severely restricts the ability to write interesting programs.

The C preprocessor [1] alleviates some of these restrictions, but its lack of integration with the language and its very limited capabilities cause many problems. Stroustrup [23] identifies elimination of the preprocessor as a major goal for C++. New C++ constructs such as templates, inline functions and constant objects help, but do not tackle the need for conditionalisation or lexical substitution.

Paradoxically, C++ has increased the need for a preprocessor. Programmers have recognised that there is considerable behavioural commonality between the objects in different applications. This common behaviour has been called a pattern [9] and stimulated a new field of research. C++ in common with other languages that predate patterns, has no constructs to support them, and so a pattern must be expressed using the available constructs. This approach works fine for simple patterns. Some more complicated patterns can be realised with the aid of the preprocessor, or by using considerable ingenuity with templates [7], [11], but most patterns are lost once they have been coded [19].

19-June-1998

University of Surrey, Computer Science Research Group,
Technical Report CSRG / 98-01.

Relationships and Patterns

Development of Entity Relationship Diagrams forms an important stage of an early form of object analysis: data modelling [4]. Availability as part of standard tools such as Software through Pictures [22] encouraged their use within hybrid structured analysis and design processes. The diagram identifies the major data elements and their relationships, with distinct icons for entities and relationships.

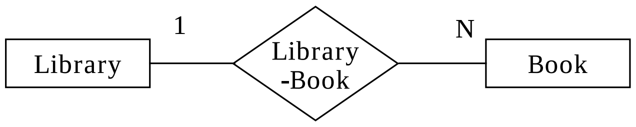

Figure 1 Entity Relationship Diagram

Object Orientation has focused on the entities, so that an entity becomes an object that encapsulates its data and provides a suitable behavioural interface. The change of focus is reflected in OO diagrams for which the various notations provide very elaborate capabilities to document an object, and the object's perspective of the relationship. The icon for the relationship is considered redundant, and so the relationship is shown as just an arc between objects. The UML notation [8] can qualify the relationship significantly so that the diagram in Figure 2 shows that a Library indexes each book by title and has an indeterminate number of copies of each.

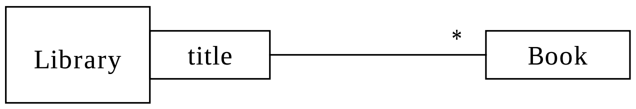

Figure 2 Unified Modeling Language Diagram

Patterns can be viewed as a natural extension of OO: associating behaviour with the relationships as well as with the entities. Clearly, an object may have more than one relationship, and a relationship may involve more than one object. Patterns and objects necessarily exhibit a many to many relationship. A programming language that supports patterns[1] must support definition and instantiation of objects and patterns as required to express a program.

The lack of support for patterns in OO languages and the simple nature of the most simple relationships causes the relationship perspective of a program to be neglected. The most trivial relationship, *has-datum*, is simply expressed by a member variable. The slightly more complicated, *has-object*, is equally easily expressed by a member variable. However in practice, different implementations are required to reflect

has-exactly-1-object
has-0-or-1-object
has-exactly-1-of-N-objects
has-0-or-1-of-N-objects

1. We cannot talk about a *pattern language* in this context, because the pattern community has chosen to use the phrase pattern language to refer to a language whose constructs are patterns.

with further distinction as to whether *has-exactly-1-of-N-objects* is by-value or by-reference. Practical considerations may require the chosen implementation to change to accommodate a legal object construction (and destruction) order.

More complicated one-sided relationships can be almost fully captured by templates such as `list<>`, `set<>`, `stack<>`, `vector<>`. Unfortunately, a template is rarely a complete solution, since template type names are clumsy. A `typedef` or inline function is often required to provide a more acceptable interface. This is the limit of C++'s capabilities.

Simple multi-sided relationships such as bidirectional pointers, parent-child pointers, or intrusive collections, cannot be captured by a single C++ declaration. It is necessary to use multiple sets of declarations, often one set of interface declarations and another set of implementation declarations for each participant in the relationship.

Proper support for a pattern requires one program declaration to define the pattern, so that a single invocation clause provides complete instantiation. Since a pattern may involve more than one class, and C++ classes (as distinct from namespaces) cannot be declared piecemeal, it is not possible to use a single preprocessor macro to satisfy the need for instantiation by a single clause.

Multi-context programming

The evolution of C++ from C sought to preserve C characteristics. The C compilation model supports independent compilation of sources followed by a link editing phase to produce an executable. C++ appears to exhibit the same model. In practice the need for construction and destruction of static objects requires the behaviour of the linker to be modified. The complexities of automatic template instantiation require significant elaboration of the compilation and or linking phases to ensure that the required and only the required instantiations occur. However, in spite of the new complexities, the apparent behaviour is the same. Independent compilation is supported, provided the One Definition Rule [24] is satisfied. Simply stated, the One Definition Rule requires that the same declaration appearing in different compilations must have the same meaning. Some violations of the One Definition Rule can be detected by the compiler and linker, others go undiagnosed, but result in an ill-formed program and consequent undefined behaviour.

It is conventional to place declarations that are used by more than one compilation in a shared include file. Each compilation is then likely to see the same definition. However the ill-disciplined behaviour of the C preprocessor, inconsistent include file ordering, or inconsistent template specialisation may subvert the Rule.

The One Definition Rule is essential to support reliable separate compilation. A simple ordering of declarations is convenient for compiler writers, and so C++ requires the entire interface of one class to be complete before the interface of another (non-nested) class can begin. This is not a convenience for pattern programmers, since it prohibits a pattern from contributing to multiple classes. Multi-context programming is supported by assemblers through the concept of program segments. However, it is missing from many high level languages. This omission has a significant impact on the way in which programs are structured: declarations must be organised so that one context is defined completely before another. Applications whose programming focuses more upon algorithms rather than data suffer as the algorithm code is partitioned over the data.

This paper describes a meta-compiler and its support for multiple contexts, whose use allows applications to be structured according to the algorithms. The meta-compiler distributes declarations from a single source file per algorithm over the data, rather than the programmer maintaining multiple source files per algorithm, with the files shared between multiple algorithms and multiple programmers.

The remainder of the paper gradually introduces the facilities that a C++ meta-compiler can offer and the solutions required to integrate a meta-compiler in a sensible compilation process. Very simple examples are provided while describing each facility. A more substantial example is then provided before comparing this approach with related work, assessing the relevance to languages other than C++ and then concluding.

The discussion of related work is rather wide ranging, partly because prior work does not consider Object Orientation, and partly because the facilities available with a meta-compiler appear to offer alternative solutions to many fields in which the facilities of C++ are limiting. The potential to generate code in response to declarations would appear to offer alternative approaches to managing persistence, whether as database schema or message structures between parallel processors.

META-COMPILATION

The need for a meta-compiler may be justified in many ways. The practical problems of implementing patterns have been outlined above. The inconvenience of editing separate interface and implementation files, and the substantial lexical redundancy of C++ code provide alternative reasons.

This section describes a particular C++ meta-compiler called FOG, a Flexible Object Generator. FOG started with a relatively modest goal of reducing the editing cost in a large inheritance hierarchy. However, it soon became clear that many other C++ problems could be addressed as well.

FOG can be used in a variety of ways:

- As a simple tool to automatically derive a pair of interface and implementation files from a combined source file.
- To achieve a useful reduction in lexical source size.
- To completely reorganise code, so that source code can be pattern or algorithm-based.

The source language for FOG is designed as a superset of C++. The intention is that C++ code can be used unchanged as FOG source, and gradually upgraded to exploit the greater facilities. This goal prohibits the introduction of any new reserved words. FOG re-uses existing words by giving meaning to syntactically valid but semantically invalid C++ statements. FOG also adds new words in a non-reserved fashion. The most extensively re-used keyword is `auto`, which is perhaps best pronounced “meta”.

Compilation Model

An augmented compilation model for a meta-compiler is shown in Figure 3.

The centre and right hand side show the conventional C++ model, with interface files 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

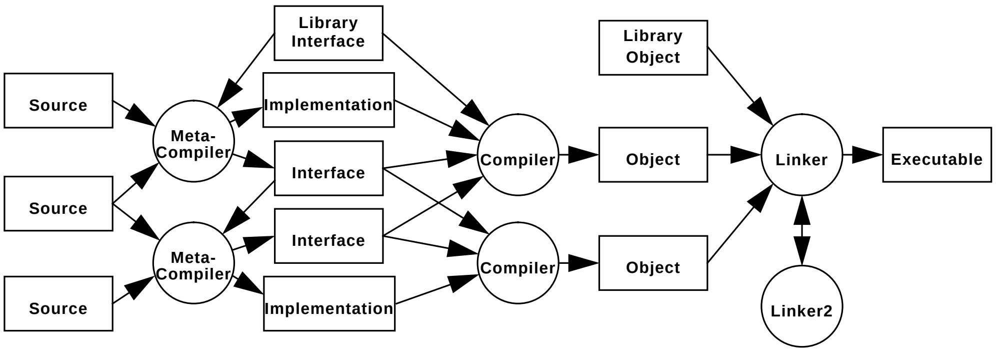

Figure 3 Meta-compilation model

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 meta-sources (the forward arrows) and from frozen interfaces (the reverse arrows). Sources may be shared between meta-compilations. A single meta-compilation may generate any number of interfaces and or implementations.

Syntax Generalisation

The most important difference between FOG and C++ is almost entirely semantic. In FOG, the distinction between declaration and definition is eliminated, so that we need only refer to declarations, and the requirement for declarations to occur exactly once and within a unique pair of scope delimiters is removed. These relaxations allow declarations to be freely interspersed providing the freedom required to implement patterns. Declarations can be added to classes, enumerations, arrays or functions as required, in the same way that program sections can be extended in a macro assembler. Multiple declarations are combined to form a composite declaration.

This change does not affect the C++ grammar because the two syntax definitions[1]

simple-declaration:
    decl-specifier-seq$_{opt}$ *init-declarator-list* ;

function-definition:
    decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*
    decl-specifier-seq$_{opt}$ *declarator function-try-block*

are all-embracing. Meaningless combinations such as

```
typedef long virtual static int const double x = 5;
```

are resolved semantically in C++, rather than syntactically.

A *decl-specifier* is extended to incorporate an *access-specifier*, allowing e.g. `private` to appear as a part of a declaration rather than somewhere before. The use of an *access-specifier* has no effect on the default access for subsequent declarations. Although this change can make declarations look a bit more like Java [10], the *access-specifier* has the normal C++ meaning; no packaging concepts have been adopted.

1. All usage of C++ grammar follows the syntax, naming and style of [24], which is very close to the imminent ANSI standard.

With this simple generalisation, a C++ declaration can be specified using a definition style:

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

and definitions may be specified using a declaration style:

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

The generalisation avoids the need to specify function interfaces and implementations separately. A single definition suffices, except that the `inline` keyword alone cannot encompass all the possible alternatives. *function-specifier* is therefore revised to introduce new alternatives.

function-specifier:

```
inline                   // Code to be inlined in the interface file
! inline                 // Code to be out-of-line
inline /interface        // Code to be inlined in the interface file
inline /implementation   // Code to be inlined in the implementation file
virtual                  // Normal meaning
! virtual                // Function is not virtual
explicit                 // Normal meaning
```

The negations (`!virtual`, and also `!const`, `!static` and `!volatile`) provide for unambiguous behaviour when multiple declarations are composed.

The semantics of a member variable initializer are generalised to permit initializers for non-static members. The initializer provides a default for use by all non-copy constructors for which no *ctor-initializer* is provided.

Classes can be defined piecemeal, with the semantics of a *declarator-id* relaxed to permit fully scoped identifiers.

```
class First
{
    class Nested {};                        // Conventional nested class
    class ::Second                          // Not a nested class
    {
        void f();                           // ::Second::f()
    };
};

class Second                                // Extends class
{
    void g();                               // ::Second::g()
};

private bool Second::_initialized = false;  // Further extends class
```

Composition

C++ programs must observe the One Definition Rule, so with few exceptions a declaration or definition occurs exactly once.

FOG eliminates this restriction. Any declaration or definition may be repeated. Multiple contributions are combined to produce a composite declaration. Provided the composition is consistent, the repeated declaration is legal. Thus a `virtual` function may be composed with a function that has no `virtual` specification, but not with one that has a `!virtual` specification.

Space does not permit a full specification of the composition rules for each declaration, although most are intuitive. Functions, variables, typedefs and enumerators can only have a single existence, so multiple contributions must not conflict and must result in a complete declaration. Each variant of an overloaded function is distinct and so composes independently. Arrays, enumerations, classes, unions and namespaces extend for distinct contributions but compose multiple contributions.

A function may have more than one *function-body* either through multiple declaration of the function or the direct specification of more than one body.

function-body:
    compound-statement function-body-context$_{opt}$ *function-body*$_{opt}$

function-body-context:
    function-body-context function-body-context$_{opt}$
    derivation-rule
    function-segment
    function-use

function-segment:

```
    /entry          // Segment for any required prologue
    /pre            // Segment for pre-condition code
    /construct      // Segment for construction ordered code
    /body           // Segment for normal code
    /destruct       // Segment for destruction ordered code
    /post           // Segment for post-condition code
    /exit           // Segment for epilogue, or return
```

Multiple function bodies are composed by independent concatenation within each of 7 segments. For most purposes only the body segment is used and this is the default. The pre and post segments provide a distinct region where patterns may place pre-condition and post-condition code. The entry and exit segments may contain initialized variables and a return statement.

When multiple function bodies are derived and composed together, it may be important to ensure that the various contributions are placed in an appropriate order. The well-defined order for the components of a constructor and destructor is used. Multiple contributions to the pre and construct segments are arranged in construction order. Multiple contributions to other segments are organised in destruction order. Multiple contributions that do not originate from distinct elements are organised in order of occurrence.

Function composition is intended to be used very sparingly. It may be used to combine code from multiple sources, typically to enable the `do_it()` of some manager class to invoke the `do_it()` methods of all its clients, without the author of the manager class needing to be aware of the number or identity of its clients (as would be the case in C++).

Manager.fog

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

Client1.fog

```
private Client1 Manager::_client1;          // Extend Manager to provide data context for client1
bool Manager::do_it() {_client1.do_it();}// Extend Manager to provide execute context for client1
```

Client2.fog

```
private Client2 Manager::_client2;
using Manager::do_it
{
   if (!_client2.do_it())
       exitStatus = false;
}
```

The final contribution shows an extension to the *using-declaration* syntax, to avoid the need to repeat redundant signature information. This can ease maintenance of signatures, since the signature of a function can be specified just once in a base class, and re-used for all derived versions. This of course requires that all derived versions use the same parameter names, which ensures greater uniformity, but introduces a more global import to parameter names.

The generalised *using-declaration* dispenses with the distinct *using-declaration* and adds `using` to the permitted alternatives for *decl-specifier*. In FOG the keyword `using` indicates that an existing declaration is to be exploited. When applied to auto-generated code, additional partial declarations can be composed to refine the default behaviour:

```
class X
{
private:
    using virtual X& operator=(const X&);
    using !inline X(const X& x) : _copy_of(x) { _copies.add(*this); }
};
```

Use of a standard function such as the copy constructor, indicates that the auto-generated behaviour is to be used as a default, although extra or replacement functionality may be composed. In the example, the default behaviour of the assignment operator is adjusted for use as a private virtual function and a copy constructor is forcibly generated out-of-line, with one member variable explicitly initialized. Other members retain the default member-wise copy. Additional code is provided for the constructor body. It is not necessary to manually write the default behaviour just to use a little non-default behaviour. It is not necessary to manually write the entire default behaviour just to control the access or placement of auto-generated code.

Meta-Classes

A C++ class has two sets of attributes: members and static members. Members, or object attributes, are attributes of the instantiated class or object. Static members are class attributes. Pure OO languages such as Smalltalk, treat each class as an instantiation of a meta-class, enabling classes to be treated uniformly as objects. With the advent of Run-Time Type Information, C++ acquires some of these characteristics, but the lack of integration of static members with RTTI prevents direct use of classes as objects.

The characteristics of C++ classes cannot be enhanced, but additional behaviour can be added for use during meta-compilation. In FOG, every class and built-in type is a meta-instance of a corresponding meta-class, which may have static and non-static, meta-variables and meta-functions. The meta-class hierarchy comprises the equivalent class hierarchy augmented by any meta-base-classes. Explicit meta-base-classes are specified by allowing use of the keyword `auto` as an *access-specifier* in a *base-specifier.* In addition all meta-classes ultimately meta-inherit from the root meta-class whose name is `auto`.

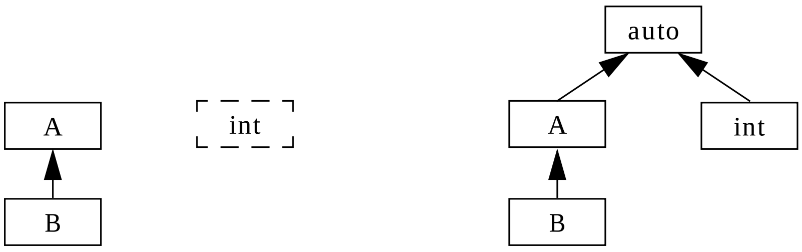

Figure 4 Class and Meta-class Inheritance Hierarchies

Figure 4 shows the meta-class hierarchy corresponding to a very simple class hierarchy.

The declaration syntax of meta-variables and functions is the same as for member variables and functions save for the extra keyword `auto`. This does not constitute a change to C++ grammar, since the keyword `auto` although very rarely used in C++ programs has syntactic validity in all declarations, (`auto` only has semantic validity within C++ function declarations).

The *type-specifier*s for meta-functions and meta-variables must be one of the reserved set of meta-types, most of which broadly correspond to grammar productions such as `identifier`, `string_literal`, `constant_expression` and `declaration_seq`. The exact correspondence is a matter for precise specification. For instance, assignment to a value of `expression` meta-type requires the assigned value to satisfy the syntax and semantics of an *expression*, whereas use of the same value corresponds to the use of a parenthesised expression which is a *primary-expression*. This enables a value of `expression` meta-type to be used wherever a *primary-expression* is allowed, and allows any of the intermediate expression productions to be used where an `expression` meta-type is required. A single `number` meta-type is used for arbitrary precision type-less arithmetic[1].

The following example defines the class `Derived` with a meta-function that defines an enumerator and a corresponding entry in a text array. The class has additional meta-inheritance from `MetaCounter` that provides a meta-variable to count the number of invocations of the meta-function. The `$` prefix indicates invocation of a meta-function or meta-variable.

```
class MetaCounter
{
    auto number values = 0;                                    // Meta-variable
};
```

1. Use of the built-in types would be more consistent, but invites extension to templated types at meta-compile time, potentially before their definition or specialisation is complete.

```
class Derived : public Base, auto MetaCounter                // Additional meta-inheritance
{
    public enum Enum {};
    public static const char *texts[] = {};
    auto static declaration_seq define(identifier aName, constant_expression aValue)
    {                                                         // Meta-function
        enum Enum { $aName = $aValue };
        static const char *texts[] = { [$aValue] ""$aName };
        $values++;                                            // Meta-expression
    }
};
```

Invocation of the meta-function as

```
$Derived::define(LABEL, 40);
```

causes the parameters `LABEL` and `40` to be located and parsed to satisfy the syntax of the meta-types `identifier` and `constant_expression`. The meta-function returns a `declaration_seq` value comprising an enumeration, an initialized static array, and an unevaluated meta-expression. Names are resolved within the scope of the meta-function prior to returning the value, with the result that `::Derived::Enum` is returned as the enumeration name. Instantiation of the return value in the invoking context installs the two potential declarations and executes the meta-expression to update the counter. Multiple invocations result in multiple enumerations and array initialisations, each of which are composed to provide a single composite declaration. Enumeration composition is straightforward. Array initializer composition exploits the GNU C extension [21] whereby a `[`*constant-expression*`]` prefix to an array initializer specifies the index to be initialized. The array element initializer comprises the value of `aName`, interpreted as a string because the prefix (empty) string acts as a cast.

The net result of each invocation is synchronisation between the value of an enumeration and an entry in an array of text strings. The example can be usefully simplified and made more reliable by using the sequentially allocated enumerator values directly:

```
    auto static declaration_seq define(identifier aName)
    {
        enum Enum { $aName };
        static const char *texts[] = { [$aName] ""$aName };
    }
```

The above example cannot be expressed in C++. Conventional practice requires that a maintainer update both enumeration and array of text strings consistently. The FOG pattern updates both at once, and provides freedom for each invocation of `Derived::define` to be located with code associated with the invocation, avoiding the need to fragment code to satisfy compiler constraints.

A very simple form of the Flyweight pattern [9] can be implemented using the C preprocessor and gives a useful contrast between the FOG and preprocessor approaches. The pattern provides an interface and an implementation for a function that returns a reference to the flyweight object.

```
const Fly& Fred::fly()
{
    static const Fly flyWeight("fly");
    return flyWeight;
}
```

When the preprocessor is used, a pair of implementation and interface macros are required for each class that may have flyweight instances.

Fly1.hxx

```
#define FLY1_INTERFACE(NAME) static const Fly1& NAME();
#define FLY1_IMPLEMENTATION(NAME, VALUE) \
const Fly1& NAME() { \
   static const Fly1 flyWeight(VALUE); \
   return flyWeight; }
```

Fred.hxx

```
class Fred
{
public:
   FLY1_INTERFACE(fly);
};
```

Fred.cxx

```
FLY1_IMPLEMENTATION(Fred::fly, "fly");
```

The entire functionality can be declared as a FOG meta-function

```
auto declaration Fly::flyweight(identifier aName, constant_expression initValue)
{
    public static const Fly& ${aName}()
    {
        static const Fly flyWeight($initValue);
        return flyWeight;
    }
}
```

enabling complete instantiation by a single invocation statement

```
class Fred
{
    $Fly::flyweight(fly, "fly");
};
```

The clumsiness of the C preprocessor approach discourages encapsulation of the pattern behaviour. As a result, simple patterns tend to get coded directly and only the original author is aware that a pattern has been used. The equivalent approach using FOG is more compact, polymorphic, properly typed and scoped, and so encourages direct instantiation of patterns.

Static meta-variables can be used to replace object-like preprocessor macros. Static meta-functions can be used to replace function-like preprocessor macros. Non-static meta-members provide distinct instances for each derivation. Meta-members are scoped, and inherited and can be used at meta-compile time in a similar way to static members at run-time.

Patterns such as Flyweight often need a different implementation for different data classes, so multiple implementations are required. These are readily provided as a meta-member of each data class, so that flyweights can be instantiated in a polymorphic fashion.

A meta-function provides a mechanism to generate the multiple code entities required to support a pattern. For simple patterns a template is able to offer similar facilities, and so a brief comparison of the two approaches is merited. A meta-function can contribute to multiple contexts and so can implement many complicated patterns, but requires an explicit invocation to instantiate the pattern. A template creates a new complete context, but can be automatically instantiated. A template is therefore very suitable for patterns that can be fully captured by a single function or type. A meta-function or ad hoc code must be used for more complicated patterns. Whether to use a meta-function to express a pattern that can be expressed as a template is a matter of programming style and context. The explicit invocation associated with a meta-function may avoid performance problems with template instantiation mechanisms. Alternatively the obligation on the template instantiation system to instantiate only those aspects of the template that are used and to perform that instantiation just once, avoids the need to site the meta-function invocation appropriately. The two approaches can be complementary since a meta-function may return multiple declarations, some of which may be templates. In the simplest case, a meta-function invocation may just return a `typedef` for a template.

Meta-replacement

The C preprocessor defines names in a single global name space and performs token-based replacement whenever a preprocessor token matches a defined name. The use of a single name space causes severe hazards for large programs. The arbitrary replacement of tokens can result in radical disruption of program meaning. The constraints of token-based replacement require the use of # and ## operators to achieve concatenation or string conversion. Replacement within strings is awkward.

Meta-replacement in FOG is syntax-based and occurs by invitation rather than by imposition. An early replacement is invited by use of

```
$name                       $name(...)
${name}                     ${name(...)}
```

where `name` may be arbitrarily scoped, and `(...)` denotes the argument list for a meta-function. The `{}` form is used when necessary to ensure that the longest possible character sequence is the required invocation. Replacement occurs naturally at the token level, in which case the `$` invocation is lexically isolated by whitespace or punctuation. Token-level replacement cannot occur within strings or characters. However the illusion of such replacement can be created by recognising that in ANSI C a concatenation of string literals composes to a single string literal. FOG generalises this so that a concatenation of character literals composes to a single character literal. Concatenation of identifiers is supported by interpreting the longest possible sequence of identifier elements and `$` invocations as a single identifier.[1] Whitespace can be used whenever such concatenation is unwanted. Type-conversion between the four low-level meta-types `identifier`, `number`, `string_literal` and `character_literal` is automatically performed so that all contributions to a concatenation share the type of the first contribution. Concatenation of numbers is possible by using a concatenated string in a context that requires a number.

```
a${five}"0"${part}_identifier
two $tokens
"String containing "${substituted}" text"
''$a_character
```

Syntax-based replacement ensures that source text satisfies the syntax requirements of the syntax-defining meta-type. Subsequent use of the meta-type value preserves its atomic characteristic, so there is no opportunity for the problems that occur when insufficient parentheses are used in conventional C macros. Meta-type conversion is performed in accordance with the normal C++ grammar, thus an `identifier` may be used as an

1. Concatenation of reserved words and character sequences could be similarly supported, but appears to offer only an opportunity for obscure programming and worse.

expression. Other conversions may be performed using the member functions of the meta-types. Space does not permit further elaboration.

Explicit specification of class names within meta-functions can be highly restrictive, or require needless parameter passing. The meta-variables, This, Scope, NameSpace and Super are reserved to identify the current scope, enclosing scope, enclosing namespace, and the first base class of the current scope. The flyweight pattern may therefore be rewritten:

```
class Fly
{
    auto static declaration flyweight(identifier aName, constant_expression initVal)
    {
        static const $This& ${aName}()
        {
            static const $This flyWeight($initVal);
            return flyWeight;
        }
    }
};
```

The enclosing class scope establishes the definition of This. Elimination of redundant mentions of Fly provides some assistance if the text is copied in another context.

The $ substitution is an early replacement and occurs as the meta-code is analysed into potential declarations. (The bodies of meta-functions are analysed upon invocation rather than upon definition.) An alternative @ substitution provides late replacement and occurs as actual declarations are created from potential declarations.

Rewriting the pattern to use @ substitution

```
class Fly
{
    auto static declaration flyweight(identifier aName, constant_expression initVal)
    {
        static const @This& ${aName}()
        {
            static const @This flyWeight($initVal);
            return flyWeight;
        }
    }
};
```

changes the behaviour so that the pattern operates correctly to create flyweights of the base class or its derived classes. Note that in the earlier example the $This defining the returned meta-type is expanded as the source text is analysed to define the Fly::flyweight meta-function, at which point This resolves to Fly. In the later example @This is expanded when the meta-function is invoked, and so if the meta-function is invoked as DerivedFromFly::flyweight the invocation occurs in the context of DerivedFromFly where This resolves to DerivedFromFly. The use of $This or @This for the internal variable does not affect behaviour.

Derivation rules

There are many patterns in which the pattern defines the behaviour of a base class and requires appropriate implementation in derived classes. A simple pattern of this form is the Prototype pattern [9], in which an existing object is cloned to create a copy.

The base class defines the (possibly abstract) method clone

```
class Base
{
    virtual Base *clone() const = 0;
};
```

Derived classes must declare and implement the protocol.

```
class Derived : public Base
{
    virtual Base *clone() const;
};
Base *Derived::clone() const { return new Derived(*this); }
```

A preprocessor macro can capture the behaviour. However, two macros are required if the implementation is to be kept out of the interface file. The need to instantiate these macros for each derived class provides undue opportunities for errors, particularly when instantiation failures cannot be detected at a compile time.

FOG introduces the concept of a derivation rule, to specify how a potential declaration provided in one scope generates an actual declaration in that scope (the root scope) and regenerates further actual declarations in derived scopes. The default behaviour is compatible with C++ and has no regeneration.

```
derivation-rule:                // Optional qualification of declarator, function-body, ctor-initializer
    /derived = root
    /derived = ! root               // below root
    /derived = branch               // between root and leaf
    /derived = leaf
    /derived = tree                 // root and branch and leaf
    /derived = pure                 // some pure-virtual in class
    /derived = boundary
    /derived = concrete             // no pure-virtual in class
```

Two categories of derivation rules are currently supported. Structural predicates support derivation at combinations of the root of the derivation hierarchy, leaves of the hierarchy and intermediate inheritance levels. Pure predicates support derivation in classes necessarily containing a pure virtual function, in classes not necessarily containing a pure virtual function, and at the inheritance boundary where there are no pure virtual functions.

The clone pattern can be implemented using a derivation rule as

```
class Base
{
    public virtual Base *clone() const = 0
        { return new @{This}(*this); }/derived=concrete
};
```

The conventional C++ part of the code defines `Base::clone` as a pure virtual function. The derivation rule qualifying the function body, causes regeneration of the function body (and its declaration) in all derived contexts that satisfy the `concrete` predicate. That is in derived classes that have no pure virtual functions. Use of `@This` ensures that the derived class name is used in the derived declaration.

This implementation may be expressed as a reusable pattern

```
auto declaration Prototype()
{
    public virtual $This *clone() const = 0
        { return new @{This}(*this); }/derived=concrete
}
```

which may be instantiated as

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

There is no need for any code in any derived class. There is no opportunity for omission or inconsistent implementation of the code in derived classes.

Another simple example, solves the problem of providing an indirect name for the inherited base class, to avoid problems with code copying or evolution when the direct name is used. Stroustrup [23] credits Michael Tiemann with the solution

```
class Derived : public Base
{
    typedef Base Inherited;
};
```

enabling usage as

```
void Derived::f()
{
    Inherited::f();
}
```

Using a derivation rule, FOG avoids the need to provide the one line declaration in every derived class, and the consequent inheritance leap-frogging that could occur from an omission or error.

```
    private typedef @Super Base::Inherited/derived=!root;
```

The presence of the rule causes the declaration to appear in all derived classes satisfying the `!root` predicate. That is in all derived classes excluding `Base`, which is the root class for the derivation rule. Application must be suppressed at the derivation root, where use of `@Super` would produce a meta-compilation error, unless `Base` is a derived class.

File disposition

A conventional C or C++ compilation processes many input files and generates a single output file. The input files and output file are readily specified on the compilation command line in conjunction with additional information to identify search paths for include files.

The same policy does not extend directly to a meta-compiler that may generate many output files, whose existence may be unknown to the author of the command line. The default behaviour of FOG is to emit an implementation and an interface file for each non-nested class and each namespace encountered in its source files. Template specialisations are emitted with the primary template. Command line options allow default paths to be specified for output directories, output file prefixes (such as `sys/`), output file suffixes (such as `.hxx`), and to provide a file name for the global namespace.

The default behaviour may be adjusted by adding file disposition directives.

```
class Base
{
    set/implementation "Bases.cxx";          // emit implementation to Bases.cxx
    set/interface "Bases.hxx";               // emit interface to Bases.hxx
};
class Derived : public Base
{
    set/implementation Base;                 // emit implementation with implementation of Base
    set/interface Base;                      // emit interface with interface of Base
};
```

```
class Local
{
    set/implementation Base;                // emit implementation with implementation of Base
    set/interface Base/implementation;      // emit interface with implementation of Base
};
```

Each scope has an associated interface and implementation file whose default may be overridden explicitly as for Base, or by reference to the file used by another entity. Interfaces and implementation can be intermixed as for Local, whose interface and implementation are both directed to the implementation file used by Base.

The implementation of a class may be emitted to multiple files through the use of file spaces. A similar technique may be used to impose a file structure upon the global namespace or upon C source.

```
class A
{
    namespace/file FirstRegion
    {
        // ...
    };
    namespace/file SecondRegion
    {
        set/implementation B;
        // ...
    };
};
```

Declarations within FirstRegion form part of class A, but are emitted to an implementation file derived by applying path, prefixes and suffixes to FirstRegion. Declarations within SecondRegion are similarly part of class A, but are emitted to the file used by the implementation of B.

Files emitted by FOG have include file guards around declarations and file inclusions that are arranged to avoid forward reference violations. Include file references and forward declarations are generated by analysis, and as a result are often tighter than those produced by hand. The current implementation of FOG does not analyse function bodies or expressions, so FOG may miss dependencies. This may be resolved by explicit specification:

```
class Base
{
    use/implementation "stdlib.h";          // Every part of implementation uses stdlib.h
    use/interface string;                   // Interface uses string
    public void f() { ... }/use=iostream    // Implementation of Base::f uses iostream
};
```

The class level specification indicates that the class and all its members have the specified dependency. The qualification on the function body provides a dependency for just the function body.

file-disposition-directive: *// extends declaration*
  set[1] /implementation *file-name*
  set /interface *file-name*
  use /implementation *file-name*
  use /interface *file-name*

1. Use of set does not cause an incompatibility with the STL template set, since when the template name occurs as part of a declaration, it cannot be part of an expression and so there is no possibility that the / could be a division operator.

file-name:
    file-ident
    entity-name — *// enum, class, typedef, function, variable, ...*
    file-name `/implementation`
    file-name `/interface`

file-ident:
    string-literal — *// "name"*
    file-ident `/prefix =` *string-literal* — *// override command line default for file prefix*
    file-ident `/suffix =` *string-literal* — *// override command line default for file suffix*
    file-ident `/noguard` — *// omit include file guard from file*
    file-ident `/guard =` *string-literal* — *// override algorithmic default for include file guard*
    file-ident `/path =` *string-literal* — *// override command line default for file path*

File utility

It is rarely appropriate to emit interfaces and implementations for every class, since this generates new interfaces for standard classes such as `iostream`. It might be convenient to generate a new interface with an extra virtual function, but this requires availability of the source and an ability to recompile all code that makes use of `iostream`.

In practice, certain external classes must be immutable. In order to support multiple meta-compilations, it is also desirable to group application classes into sub-systems which are mutually immutable.

This problem is resolved by extrapolating from the concept of a class utility [3]. A class utility is free-standing code independent of the current application. FOG determines a (degree of) utility as each declaration is created. The utility indicates whether the declaration belongs to a frozen free-standing external class utility, to a pool of declarations, or whether the declaration is to be emitted. The default behaviour is for all declarations arising from `#include` files to be treated as frozen utility declarations that must not be changed or re-emitted. A replacement syntax for `#include` provides qualifiers to select the nature of included declarations. The replacement syntax is only valid as a global-scope declaration. Inclusion only occurs upon the first encounter, and so include file guards are unnecessary.

```
using "file.h";                     // Include file.h preserving prevailing utility
using/utility "string.h";           // Include string.h as utility declarations
using/pool "shared.h";              // Include shared.h as pooled declarations
```

include-file-declaration: — *// extends declaration*
    `using` *string-literal* `;`
    `using /emit` *string-literal* `;`
    `using /pool` *string-literal* `;`
    `using /utility` *string-literal* `;`

The utility or pool attributes apply throughout the included file and its nested inclusions. The prevailing mode is restored after the include completes.

Declarations acquired while in utility mode provide information that enables FOG to correctly analyse and emit the wanted code, but do not directly cause emission of the utility code. Utility classes may indirectly contribute to emitted code by providing derivation rules that contribute code to classes that are emitted. Any attempt to change the functionality of utility classes can be diagnosed and rejected.

The interface and implementation files required by a C++ compiler encourage a physical structuring of application code to use a separate pair of files for each (major) class. For many applications this is quite appropriate, although the need for two rather than one file per class is unfortunate, and eliminated by using FOG. For other applications it may be better to structure the code with a separate file for each major algorithm, even though that algorithm is associated with more than one class. The extra level of processing available with a meta-compiler provides the freedom to organise the source code in one manner to suit the programmer, and to allow the meta-compiler to reorganise to suit the more restrictive needs of the compiler.

Sources needing reorganisation may be included by multiple meta-compilations as pooled sources, indicating that only those declarations that contribute to emitted declarations are to be used. Those that contribute to utility classes are to be ignored, presumably because the source was also included in the meta-compilation that generated those utilities.

The following example extends a base class to support the requirements of `do_algorithm` and provides all the functionality to implement the algorithm in multiple classes. The class hierarchies are defined elsewhere. Other algorithms may be defined in other files.

```
class Base
{
    protected typedef @Super Super/derived=!root;
    private bool _done_algorithm = false;                   // Default for constructors
    protected bool done_algorithm() const { return _done_algorithm; }
    public virtual void do_algorithm()
        { _done_algorithm = true; }/body/derived=root
        { if (!done_algorithm()) \{ }/entry/derived=!root    // Unmatched { escaped
        { Super::do_algorithm(); }/body/derived=!root
        { \} }/exit/derived=!root                            // Unmatched } escaped
};
using Derived1::do_algorithm
{
    // ....
}
using Derived2::do_algorithm
{
    useful_code();
}
```

The multiple function bodies for `Base::do_algorithm` provide a base class implementation with run-time guards against multiple invocation of the algorithm whilst also ensuring that all inherited versions of the algorithm are executed. The actual code emitted for `Derived2::do_algorithm` would be

```
void Derived2::do_algorithm()
{
    if (!done_algorithm())          // From Base::do_algorithm /entry
    {                               // Unmatched { from Base::do_algorithm /entry
        useful_code();              // From Derived2::do_algorithm
        Super::do_algorithm();      // From Base::do_algorithm /body
    }                               // Unmatched } from Base::do_algorithm /exit
}
```

Only the unique part of the derived code needs to be explicitly provided.

The source code shown is solely responsible for supporting a particular algorithm, and when located in a separate file provides a complete encapsulation of the support necessary for that algorithm. Other files defining the class hierarchy and other algorithms are completely isolated in so far as the code has been fully separated enabling independent concurrent development of algorithms. There is no programming isolation, since multiple contributions to the classes are combined by the meta-compiler, where some unfortunate name clashes may at least produce errors, while others could lead to undesirable composition. Equally, there is no isolation that prevents the code for one algorithm accessing facilities provided for another algorithm. Both algorithms are composed into the same class, where there is no distinction between the private parts that support one algorithm or another.

If name isolation between independent algorithms is required, each algorithm may define a nested class to encapsulate its meta-context, reducing the potential for name-clashes to the nested class name.

EXAMPLE

An implementation of the Visitor pattern [9] provides a more complete example of some of the facilities of FOG.

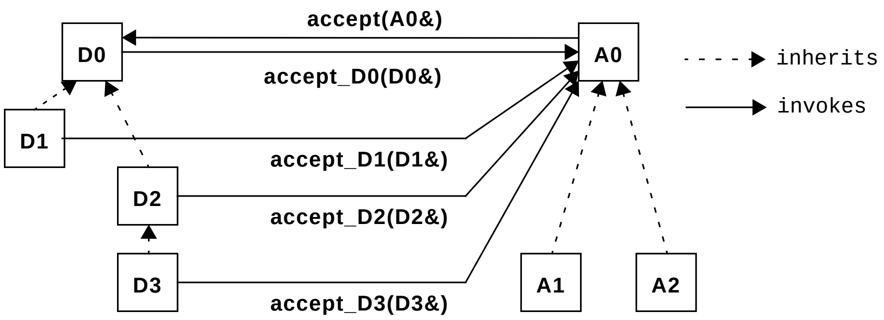

Figure 5 Visitor classes

The pattern involves a hierarchy of data classes (D1, D2...) and a number of algorithms that may be performed on the data. The algorithms are realised by algorithm classes (A1, A2...) and inherit from the abstract Visitor class A0. The data classes similarly inherit from an abstract class D0.

Usage of the pattern requires the appropriate data and algorithm dependent action to be performed. This is achieved by an invocation of the virtual method `D0::accept(A0&)` for which the derived implementation invokes the virtual method `A0::accept_D`*n*`(D`*n*`&)` and whose optional derived implementation performs the required action. With A algorithms and D data classes there may be as many as A*D functions to be declared and implemented. The actual code in the algorithm classes performs the required actions. The scaffolding code required in the data element classes can be generated automatically by FOG.

The pattern has two degrees of freedom, which contributes to the inconvenience of a conventional manual approach. Addition of an extra algorithm class is relatively benign, requiring just that the new algorithm class implements as many of the data functions as required. Addition of an extra data class requires that the data class complies with the inherited protocol and that an additional method be defined for the abstract algorithm. It may also be necessary to implement this method in every derived algorithm class.

The example implementation uses two meta-functions, one to be invoked in the root data class (D0), and another to be invoked in each derived data class (D*n*).

The derivation rule described above cannot satisfy the derivation requirements of this pattern. In the earlier definition, a derivation rule could regenerate from potential declarations in derived contexts. In this case, derivation of a data element class (D*n*) needs to export a declaration to the abstract visitor class (A0). A generalisation of the rule could be considered, but appears to be necessary only for this pattern. The cost of a one-line invocation in each derived data element class is small compared to the associated response code in the algorithm classes, and provides flexibility for deliberate omission if some levels of the data element hierarchy do not need support.

```
auto declaration_seq VisitorBaseElement(identifier V)
{
    auto identifier Visitor = $V;
    public virtual void accept($V& aVisitor) = 0;
};
auto declaration_seq VisitorDerivedElement()
{
    public virtual void accept($Visitor& aVisitor)
        { aVisitor.accept_${This}(*this); }
    public virtual void ${Visitor}::accept_${This}($This& aDatum) {}
};
```

The abstract classes may invoke the pattern as

```
class AbstractAlgorithm /* ... */
{
    // ...
};
class AbstractDataClass /* ... */
{
    $VisitorBaseElement(AbstractAlgorithm);
    // ...
};
```

Invocation of `VisitorBaseElement` initializes the `AbstractDataClass::Visitor` meta-variable with the value `AbstractAlgorithm`, avoiding the need to pass the same parameter to the invocations in derived classes:

```
class DerivedData1 : /* ... */ public AbstractDataClass /* ... */
{
    $VisitorDerivedElement();
    //...
};
class DerivedData2 : /* ... */ public AbstractDataClass /* ... */
{
    $VisitorDerivedElement();
    //...
};
class DerivedDerivedData : /* ... */ public DerivedData2 /* ... */
{
    $VisitorDerivedElement();
    //...
};
```

Definition of the derived algorithm classes, like the abstract algorithm classes requires no explicit code. The declarations are provided automatically by the second declaration in the `VisitorDerivedElement` meta-function, which also provides a default empty algorithm implementation. Since the signatures of the algorithm are defined in the base class, implementation of the derived algorithm code need only mention the name.

```
class DerivedAlgorithm : public AbstractAlgorithm {};
```

```
using DerivedAlgorithm::accept_DerivedData1
{
    // ...
}
```

The pattern is expressed compactly, and instantiated so that its use is clear. Compliance with the pattern is ensured because the pattern provides all the relevant declarations. The manually contributed code is reduced to that necessary to provide the actual implementation. The scaffolding is almost completely removed.

PERFORMANCE

In principle, a program written in FOG generates precisely the same code as the equivalent program written in C++. However the FOG source is significantly smaller, often by a factor of two. In practice, the ability to instantiate patterns directly may lead to a slightly larger program through the re-use of general purpose rather than hand-written declarations.

The current implementation of FOG, performs C preprocessing, FOG and C++ source analysis, and declaration composition before emitting C++ sources. These sources are then processed by a C++ compiler which has to repeat much of the parsing and analysis already performed by FOG. There is therefore a possible doubling of processing time, assuming FOG were coded as efficiently as a production C++ compiler.

The overall compilation time may be reduced a little, since FOG emits accurate include file references and consequently the C++ code may exhibit reduced dependencies and so may require fewer source lines to be parsed.

The overall recompilation time after a small change may well be increased significantly, if FOG is performing substantial code reorganisation so that a minor change to a FOG source affects many emitted files. This problem may be particularly severe if `#line` information is used to cause debuggers and compilers to regard the FOG source rather than the emitted C++ as the primary source. The minor change may cause a FOG source line number to change and so require many emitted files to be regenerated just to update the line numbers[1].

FOG does not regenerate an emitted file if the new file content matches the former content. When line numbers are suppressed, or the FOG sources only require minimal reorganisation, the compilation cost of FOG can therefore be kept low. When such control is not possible, minor recompilations may require total recompilation at about twice the cost of a normal C++ total recompilation.

The costs of using a meta-compiler could be substantially reduced by using an intermediate format to communicate between meta-compiler and compiler. This not only eliminates the double processing of source code, but also avoids the adverse interaction between the dependencies of each FOG meta-compilation and the dependencies of each C++ compilation.

RELATED WORK

Although, extensively criticised [23], relatively little work has been published on alternatives to the C preprocessor [1]. Weise [27] provides a review of earlier work, and

1. FOG output is pretty-printed and apart from the lack of comments resembles code that could have been written by hand. Debugging FOG output is therefore practical.

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 a form of character-based as well as syntax-based replacement. Weise introduces 9 extra lexical operators, requires an explicit return, and produces examples that are unpleasant and sometimes difficult to read. FOG introduces only 2 extra lexical operators ($ and @) and by treating the entire function body as the return achieves a simpler substitution model. Weise considers only ANSI C, whereas FOG revisits the concepts from an OO perspective.

Custom source translators have no doubt been developed by many authors, although their simple lexical behaviour has not merited publication.

C++ has been extended in minor ways by practical compilers [21] and [25], and a few isolated language extensions have been published. Baumgartner [2] introduces signatures and pointers to signatures so that a class can comply with a protocol without necessarily inheriting from a specific abstract base class. Porat [18] advocates extension to support pre-conditions, post-conditions and invariants in the style of Eiffel. FOG provides a few hooks in the form of informally reserved function segments where patterns could place pre-condition and post-condition code.

C++ maintains compatibility with C and so continues the failed experiment of declaring variables and functions in the style of their usage. The consequences of this cause unwarranted difficulties to every C++ compiler or tool vendor. C++ is very difficult to parse. Werther [28] provides a sensible proposal for a completely new syntax using more conventional syntactical styles like Ada or Pascal. Use of a simple unambiguous grammar would make FOG simpler and more accurate.

Migration of an object in time or space has caused considerable difficulties for database vendors and parallel program researchers, each of whom have developed a variety of C++ translators or variants.

Persistence in database applications requires that an object be stored on and retrieved from disk. This requires a low level knowledge of the object layout and an ability to store pointers and referenced objects in a way that can be resolved when an object is subsequently restored. Resolution of pointers must also include invisible pointers such as the virtual function table pointer, since there is no guarantee that the table has the same address in different applications sharing a database, or even in recompilations of the same application. Full restoration should not assume invariant object layout between different applications [19]. Park [17] identifies two standard approaches. Some databases apply a class library binding and so only support persistence for classes that inherit from a special base class. Alternatively a data manipulation language is developed as a minor variant on C++ with an extra keyword such as `persistent` or a variant of `new`. Park proposes a hybrid approach that provides the efficiency of the enhanced C++ approach while supporting persistence for arbitrary classes. Whatever approach is taken, the database requires a description of each stored object. An enhanced compiler can provide that information directly. Without compiler enhancement it is necessary to define object layouts as database schema, which are then translated to C++ declarations. Meta-level programming provides an opportunity for the descriptions to be emitted from source code without the need to use schema, and for the special support code to resolve pointers to be generated directly.

Parallel programming applications have a similar need to copy objects, but between address spaces. The problem here is simplified when all processors are of the same type, and execute the same source code.

The parallel processing community has investigated extensions to support concurrency. Wilson and Lu [29] provide extended articles by 16 of the leading research teams. Some of these are realised by 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++ [12] 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. The extra facilities available with FOG could well provide scope for implementing many of the extensions that are based on additional declarations. The more radical changes of C** [14] in which data parallel semantics are introduced to expressions could certainly not be addressed by FOG.

The use of patterns has provoked considerable interest since the original patterns book [9]. The annual PLOPD conferences [6], [15] and [26] have added many more patterns, but only Soukup [20] really addresses the problems of implementing patterns.

The pattern for implementing patterns [20] summarises the much more extensive consideration in [19]. Soukup describes the very uniform approach to implementing patterns adopted by the CodeFarms library. Each pattern is realised by static member functions of a data-less class, which is declared to be a friend of all the participating classes. The relationship of every data member is declared as a use of a pattern, using a private language that looks like preprocessor instantiations. A simple lexical conversion of the private language defines preprocessor macros that inject the required code in to application classes, provided the application classes invoke the appropriate macros. This approach is clumsy for very simple patterns, but reasonable for complicated patterns and has a very beneficial side effect of dramatically reducing compilation dependencies. FOG relaxes C++ syntax to allow declarations for different classes to be interspersed. This is exactly the relaxation required to support the CodeFarms approach. The CodeFarms patterns could be defined as global FOG meta-functions eliminating the need for preprocessor tricks.

The concepts of meta-classes are well defined for languages such as Smalltalk. Chiba [5] considers an extended form of C++ called Open C++, to allow 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 very considerable power, but because programming is very closely related to 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. Expression terms that need manipulation can be realised as inline functions, which FOG can adjust.

Meta-level programming has been established in CLOS in the form of the MOP (Meta-Object Protocol) [13]. C++ has a much less regular syntax than CLOS, does not treat functions as first class objects, and requires all declarations to be checked at compile-time. Restricting meta-behaviour to classes at compile-time is necessary to remain close to the C++ language philosophy. Safe composition of functions as a cascade of functional applications deriving from independent MOPs has been considered by [16]. A Mixin-like orthogonality between contributions is recommended but cannot be imposed. The unsafe approach adopted by FOG is clearly subject to the same recommendations.

FURTHER WORK

The description of meta-functions and meta-variables indicates that most C++ concepts can be usefully adopted as part of a meta-compiler, just by adding a meta- prefix. Meta-expressions have been used in a limited way to define values of meta-variables. The issue of meta-statements has not been addressed. Syntactically, there is little to prevent C++ control statements being used to support conditionalisation and iteration at the meta-level. One of the major complexities of C++ parsing is the co-existence of declaration and expression syntax within function bodies. Since such co-existence is resolvable within a function body, it is also resolvable outside.

Support for a *selection-statement* (`if` or `switch`) at the meta-level would make the C preprocessor fully redundant. Command line defines could be accessed as the meta-members of a special namespace, possibly `std`. `$std::FILE` could replace `__FILE__`.

Support for an *iteration-statement* (`while` or `for`) could open up the full field of meta-level programming [13]. The main challenge is to support iteration over the useful domains of the actual declarations, in particular the non-static-member-variables so that the appropriate data structures can be generated to support persistence. A meta-iterator of meta-type pointer to the `auto` root meta-class, can point at any meta-class. Provided accesses through the iterator use the dynamic rather than static type, then the behaviour of the actual meta-class can be used. Introduction of a single meta-iterator would appear to avoid the need to support any other form of pointer-to or reference-to meta-type. This seems a worthwhile simplification, but we now have a Smalltalk behaviour in C++.

Generation of data structures to manage persistence must be performed after the class layouts have been frozen. It cannot readily be performed as part of some declaration. An appropriate opportunity to create inter-class interaction and to implement global analysis code can be found by introducing meta-construction and meta-destruction stages to the meta-compilation activity:

(1) ANSI C preprocessing
(2) Source code analysis
(3) Meta-construction
(4) Resolution of `using` references
(5) Application of derivation rules
(6) Meta-destruction
(7) Declaration dependency analysis
(8) Generated file dependency analysis
(9) File emission

Meta-construction involves execution of any meta-code that may have been defined for the meta-constructor. Use of a meta-constructor enables the Visitor pattern to be captured by a single meta-function:

```
auto declaration_seq VisitorBaseElement(identifier V)
{
    auto identifier Visitor = $V;
    public virtual void accept($V& aVisitor) = 0
        { aVisitor.accept_@{This}(*this); }/derived=!root
    auto ${This}::${This}()                                    // Meta-constructor
    {
        public virtual void ${Visitor}::accept_@{This}(@This& aDatum) {}
    }/derived=!root
}
```

The meta-constructor provides an implementation for each derived class. Once source analysis is complete, all meta-constructors are invoked in a least-derived first order. Execution of the derived data class meta-constructor creates the requisite declaration in the `Visitor` base class. It is not necessary to invoke `VisitorDerivedElement` in each derived class.

Meta-destruction invokes the meta-destructor of each meta-class once in a most-derived to least-derived order, invoking `auto::~auto()` last of all. Meta-destructors may only update declarations, so any declarations must first appear in a meta-constructor, if not already defined by conventional programming.

The following code indicates how persistent classes could be managed.

```
    auto auto::~auto()
    {
        for (auto iterator p = member_variables; p; ++p)
            if (p->is_persistent)              // test meta-variable predicate
                p->do_something();             // invoke meta-function
    }/derived=tree;
```

All classes are meta-classes, and all meta-classes inherit from `auto`, so the example code fragment is invoked for every meta-class. The code performs an iteration over the member variables using the meta-iterator. At each iteration step, the iterator tests the `is_persistent` meta-variable, and if true then invokes the `do_something` meta-function to arrange for whatever action is necessary to support persistence. Since `is_persistent` and `do_something` are accessed with respect to each member variable, an appropriately overloaded implementation of `do_something` can be provided for each data type, including the built-in types.

The example is written without using any `$` meta-accessors. This makes the code easier to read, but leaves very little visual distinction between meta-code and ordinary code. The initial `auto` prefix clearly marks the presence of meta-code allowing subsequent names to be resolved unambiguously within meta-namespaces.

It may well be better to require `$` prefixes for all meta-accesses; to create a visual distinction, provide consistency of usage between declaration and statement contexts, and to avoid reintroducing the name clash hazards associated with the C preprocessor. If prefixes are avoided, addition of the following line by a library developer

```
    auto number auto::i = 0;
```

could seriously disrupt all application meta-code using the symbol `i`. If `$` prefixes are mandated, then replacement can only occur where such replacement is requested, and consequently any introduction of the same names in base classes should be occluded by application code.

Meta-constructors and meta-destructors are not inherited, since inheritance is easily provided as above by `/derived=tree`.

Access constraints are not applied to meta-functions or meta-variables. They are all treated as `public`. There is no reason why access constraints should not be applied, since there is a very strong correspondence between static members at run-time and meta-members at meta-compile time. It just doesn't seem appropriate. Access protection is useful in C++ to preserve encapsulation. Use of meta-members occurs in contexts where a number of different classes are being declared together. This is perceived to be a collaborative activity within a programming team. Imposition of access barriers would appear to impede legitimate requirements without offering any improvements in integrity.

OTHER OO LANGUAGES

The discussion on the difficulties of implementing patterns has focused on C++, but many of the comments apply equally to other languages, which also require complete declaration of one class before declaration of another class.

The problems are particularly severe in C++, where strong typing can often require code to be repeated for a variety of types, and where the absence of a run-time meta-class deprives the programmer of useful tools. Two of the examples in this paper: cloning and base class access are not an issue at all in Eiffel or Smalltalk for which the languages provide direct solutions.

The orthogonality between objects and patterns, when viewed as extensions of entities and relationships necessitates intermingling of partial class declarations. As has been shown for C++, this does not require a new language, merely a relaxation of unnecessary constraints during an early (meta-)compilation stage, and the provision of relevant macro facilities to enable the partial class declarations of a pattern to be instantiated directly. Similar relaxations and macro facilities are required to implement patterns effectively in other OO languages.

CONCLUSION

The inability to define powerful user-level programming constructs has been shown to be a severe obstacle to implementing patterns in C++ (or indeed other OO languages). Simple patterns can be resolved with the basic language facilities. Slightly more complicated patterns require extreme and possibly undue programming ingenuity. Arbitrary patterns require some form of translator.

Once the need for a translator is accepted, the compiler-friendly perspective of C++ can be reviewed to create a more user-friendly language. Elimination of the One Definition Rule and a syntax relaxation to allow arbitrary intermixing of declarations for more than one scope creates the flexibility required to support patterns.

AVAILABILITY

It is intended to put the entire source code for FOG into the public domain at

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

The code is experimental, and not as efficient as a production compiler, but only by a modest two- to four-fold factor. As the first program to process user code, FOG should perform as much semantic validation and diagnosis as possible. The error messages from FOG are not comparable to a good C++ compiler, so the user may have to wait to perform meta-compilation and then use diagnostics from the subsequent C++ compiler.

Since FOG reinterprets and rearranges declarations, it must fully parse all declarations. FOG handles the full proposed ANSI C++ syntax, but at present takes a number of short cuts.

ACKNOWLEDGEMENTS

Lui Melechi contributed to some early discussions, and provided timely reassurance that the overall direction was credible. Brian Hillam read the manuscript very diligently. The first author is very grateful to Racal Research Ltd. for providing the time and resources to pursue this research.

REFERENCES

1 ANSI X3.159-1989. *American National Standard for Information Systems - Programming Language - C*. American National Standards Institute, 1990.

2 Gerald Baumgartner and Vincent F. Russo. *Implementing signatures for C++*. ACM Transactions on Programming Languages and Systems, vol. 19, no. 1, 153-187, January 1997.

3 Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin / Cummings, 1994.

4 P. Pin-Shan Chen. *The entity-relationship model - towards a unified view of data*. ACM Transactions on Database Systems, vol. 1, no. 1, 9-36, March 1976.

5 Shigeru Chiba. *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, vol. 30, no. 10, 285-299, October 1995.

6 James O. Coplien and Douglas C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

7 James O. Coplien. *Curiously recurring template patterns*. C++ Report, 24-27, February 1995.

8 Martin Fowler with Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

9 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design patterns, Elements of reusable object-oriented software*. Addison-Wesley, 1995.

10 J. Gosling, B. Joy and G. Steele. *The Java language specification*. Addison-Wesley, 1997.

11 Rich Hickey. *Callbacks in C++ using template functors*. C++ Report, 42-50, February 1995.

12 Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Motohiko Matsuda, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Jörg Nolte and Mitsuhisa Sato. *MPC++*. In Gregory V. Wilson and Paul Lu. *Parallel Programming using C++*. 429-463, MIT Press, 1996.

13 G. Kiczales, J. des Rivières and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

14 James R. Larus, Brad Richards and Guhan Viswanathan. *C***. In Gregory V. Wilson and Paul Lu. *Parallel Programming using C++*. 297-341, MIT Press, 1996.

15 Robert C. Martin, Dirk Riehle and Frank Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

16 Philippe Mulet, Jacques Malenfant and Pierre Cointe. *Towards a methodology for explicit composition of metaobjects*. Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'95, ACM SIGPLAN Notices, vol. 30, no. 10, 316-330, October 1995.

17 Chong-Mok Park, Kyu-Young Whang, Il-Yeol Song and Shamkant Navathe. *Forced inheritance: A new approach for providing orthogonal persistence to C++*. Journal of Object Oriented Programming, 65-71, March / April 1996.

18 Sara Porat and Paul Fertig. *Class assertions in C++*. Journal of Object Oriented Programming, 30-37, May 1995.

19 Jiri Soukup. *Taming C++: Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994.

20 Jiri Soukup. *Implementing Patterns*. In James O. Coplien and Douglas C. Schmidt. *Pattern Languages of Program Design*. 395-412, Addison-Wesley, 1995.

21 Richard M. Stallman. *Using and porting GNU C*. Free Software Foundation, Inc., Cambridge MA, January 1998. Available as part of the gcc-2.8.0 distribution.

22 *Software through Pictures - User Manual*. Interactive Development Environments, San Francisco, June 1991.

23 Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

24 Bjarne Stroustrup. *The C++ programming language. Third edition*. Addison-Wesley, 1997.

25 *C++ Language Reference*. In *Microsoft Visual C++ 5.0 Programmer's Reference Set, Volume 4*. Of *Microsoft Visual C++ Language Reference*, Microsoft Press, 1997.

26 John M. Vlissides, James O. Coplien and Norman L. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

27 Daniel Weise and Roger Crew. *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.

28 Ben Werther and Damian Conway. *A Modest Proposal: C++ Resyntaxed*. ACM SIGPLAN Notices, vol. 31, no. 11, 74-82, November 1996.

29 Gregory V. Wilson and Paul Lu. *Parallel Programming using C++*. MIT Press, 1996.