# Preprocessing C++ : Meta-Class Aspects

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

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

## ABSTRACT

C++ satisfies the previously conflicting goals of Object-Orientation and run-time efficiency within an industrial strength language. Run-time efficiency is achieved by ignoring the meta-level aspects of Object-Orientation. In a companion paper [15] we show how extensions that replace the traditional preprocessor lexical substitution by an Object-Oriented meta-level substitution fit naturally into C++. In this paper, we place those extensions in the context of a compile-time meta-level, in which application meta-programs can execute to browse, check or create program declarations. An extended example identifies the marshalling aspect as a programming concern that can be fully separated and automatically generated by an application meta-program.

## Keywords

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

## 1 INTRODUCTION

Prior to C++, Object Orientation, as exemplified by Smalltalk, was perceived to be inherently inefficient because of the run-time costs associated with message dispatch. C++ introduced a more restrictive Object Model that enabled most of the run-time costs to be resolved by compile-time computation. As a result Object Orientation in C++ is efficient and widely used.

C++ requires that the layout of objects be frozen at compile-time, and that the type of the recipient of any message is known. The layout constraint enables a single contiguous memory allocation for each object. The messaging constraint enables non-virtual and some virtual function calls to be implemented as simple function calls. The remaining virtual function calls require a single indirection from a known index into a relatively small dispatch table. These are pragmatic constraints. Elimination of run-time object flexibility eliminated the need for run-time code to manipulate, and run-time objects to describe, object structure. The meta-classes that are essential for languages such as Smalltalk were not necessary for C++ and so they are not part of the language.

It has been found that some degree of self-awareness is useful to an Object Oriented program. This may involve

- a knowledge of class names for diagnostic purposes
- availability of inheritance information as Run-Time Type Information to validate dynamic casts
- object layout information to support marshalling for communication
- object layout information for persistent storage in data bases
- full class descriptions for browsers or debuggers

The first two of these needs have been addressed as C++ has evolved from ARM [7] to ANSI standard [1].

When a Smalltalk or CLOS program reflects upon itself, this necessarily happens at run-time, since this is when object structure is defined. In C++, objects are defined at compile-time, and so an opportunity exists for a program to reflect upon itself at compile-time as well, or instead. If the purpose of that reflection is just to extract some information or perform some checking in a one-off fashion, it is clearly preferable for such code to execute at compile-time. If reflection is to happen continuously, then it must occur at run-time.

FOG is an extension of C++ that supports execution of meta-programs in an interpreted fashion at compile-time. Declarations and statements for use at compile-time are simple extensions of the equivalent run-time

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

declarations, and so there is relatively little new syntax for programmers to learn. Normal programs operate at run-time through the execution of functions. Variables contain values of built-in numeric types or user-defined types. In FOG, meta-programs operate at compile-time through the execution of meta-functions. Meta-variables reference declarations that have built-in meta-types.

Provision of compile-time meta-programs gives C++ programmers some of the capabilities available to programmers in other Object Oriented languages, without incurring any run-time costs. This provision is sufficient to satisfy a significant class of practical problems that require a one-off manipulation of the program and currently have to be resolved by custom preprocessors or more manual approaches. The more general problem of supporting run-time reflection is potentially soluble by arranging for compile-time reflection to prepare data structures for use by a subsequent run-time environment.

In Section 2, we contrast the C++ Object Model with an idealised Object Model and define the extension to the C++ model available for use at compile-time in FOG. We then describe the range of meta-types used by compile-time meta-objects and the meta-type methods available for meta-programming. In Section 3, we show how these facilities fit together to support generation of marshalling code automatically. In Section 4, we contrast the meta-level concepts in other languages and show how many of the potential problems are avoided by pragmatic restrictions.

## 2 THE META-LEVEL

### 2.1 Meta-Classes

Conventional programming involves programs that operate on application entities. Meta-programming involves programs that operate on program entities: the class, function and variable declarations that define a program.

In the same way that newcomers to Object-Orientation are easily confused by loose usage of the terms class, instance and object, newcomers to meta-programming are easily confused by loose usage of the term meta-class. One confusion arises because, in Object-Oriented Programming, the phrase *is-a* denotes an inheritance relationship, but in meta-programming *is-a* can alternatively be used to denote an instantiation relationship. Further confusion arises because the object models available to the programmer do not correspond to the underlying abstraction.

We will therefore briefly describe the very pure Object Model exemplified by ObjVlisp [6], before describing the C++ Model and the enhancements provided by FOG. A very simple three-class hierarchy is shown in Figure 1 in which class X inherits from class Y, which inherits from class Z. A single instance of each class is shown and named respectively anX, aY and aZ.

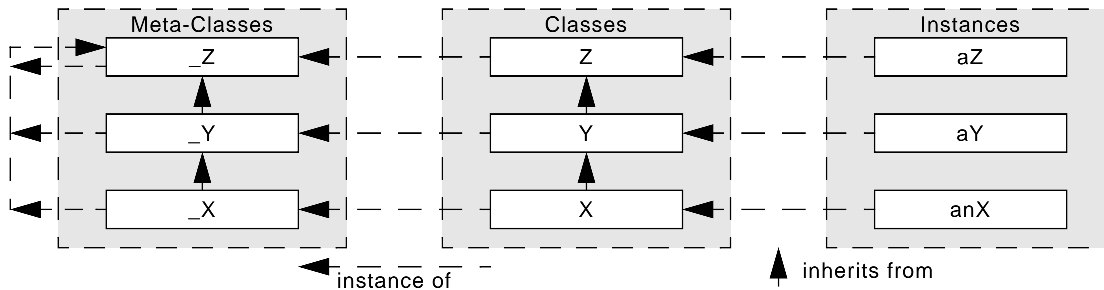


**Figure 1 Pure OO Object Model**

An Object Oriented program performs computation as a result of the interaction of its object instances at run-time, and in the simplest Object Model instances have meaning only as object instances at run-time and classes exist solely as an abstraction at compile time.

A more sophisticated Object Model enables the run-time objects to make use of class information, and in a pure Object Model, this information is provided by a run-time object for each class. Each class object provides a description of instances of its class. Every object must be an instance of some class, so it is necessary to define meta-classes that are instantiated as the class objects. The meta-classes are labelled _X, _Y and _Z in the figure. There are corresponding meta-class objects to describe each class. A potentially infinite recursion is avoided by ensuring that the instance of _Z is also a valid description of _Z. Every box in Figure 1 corresponds to a run-time object. Vertical arrows denote an inheritance relationship. Horizontal arrows denote an *instance-of* relationship from right to left or a *describes* relationship from left to right.

Meta-classes were originally introduced for languages such as CLOS and Smalltalk to assist in the construction of instances whose layout was entirely defined at run-time. More efficient languages such as C++ or Eiffel define object layouts at compile-time and compiler writers have no need to provide meta-classes. The available facilities in C++ are limited but they do exist. The C++ components corresponding to Figure 1 are shown in Figure 2. The instances in the right hand column comprise a contiguous piece of memory for the member

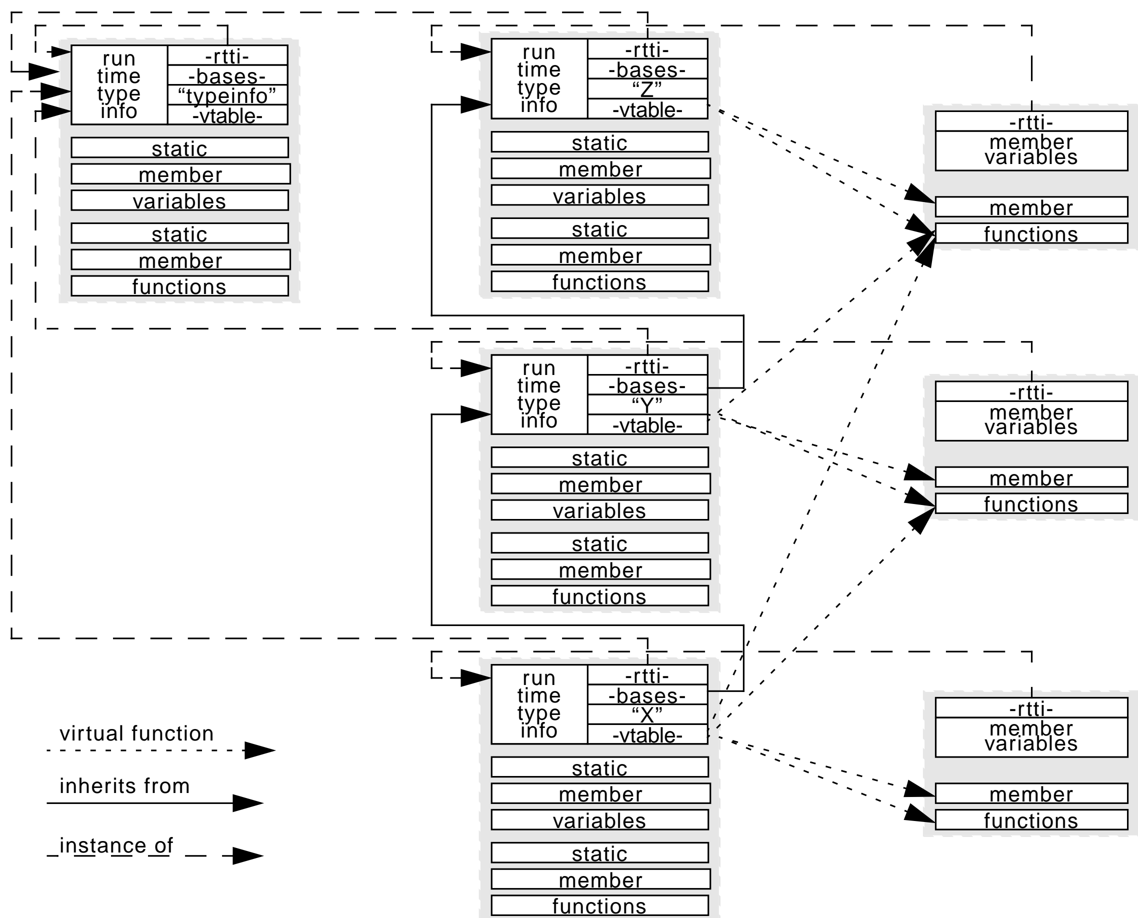


**Figure 2 C++ Object Model (OO Perspective)**

variables and `-rtti-` (a hidden pointer to the class description), in addition to some discontiguous memory areas containing the code for the member functions.

Each class (and meta-class) ‘object’ comprises a potentially contiguous area of memory containing the compiler generated run-time type information as a `typeinfo`, and generally discontiguous areas of memory for the static member variables and functions. The `typeinfo` typically comprises `-rtti-` (pointer to the class description), `-bases-` (list of pointers to the base class descriptions), a class name, and `-vtable-` (dispatch table for virtual functions). The `typeinfo` class has very limited functionality and the `typeinfo` instance that both describes and is an instance of the `typeinfo` class is the only meta-class in C++.

In common with other programming languages, C++ does not require the programmer to distinguish between class variables and instance variables. This is a programming convenience, but leads to some confusion between class and instance meanings, and undermines the pure perspective of the meta-class structure. The alternative presentation of Figure 3 is more appropriate. All visible names now appear in one rather than two columns. Only the Y class is shown. Readers are invited to imagine a similar picture for Z above and for X below Y.

C++ supports instance objects and, to a limited extent, class objects at run-time. FOG extends C++ to support class objects and, to a limited extent, meta-class objects at compile time. The programming perspective of the Object Model supported by FOG is shown in Figure 4. The run-time configuration is unchanged, although the run-time components are drawn more compactly to highlight the significant aspects and to ease comparison with compile time components. The compile-time components for a class comprise exactly two notional objects, one meta-class object to describe the run-time class, and one class object to describe the run-time instance. Since these two notional objects have the same inheritance and always exist as a pair, it is convenient to treat the pair of objects as a single object, which may be safely but loosely referred to as the meta-class.

A meta-class has instance variables and functions, as well as class variables and functions in just the same way as a run-time class has instance (member) variables and functions and class (static member) variables and functions. The meta-class members are referred to as (non-static) meta-members and static meta-members by direct analogy with (non-static) members and static members.

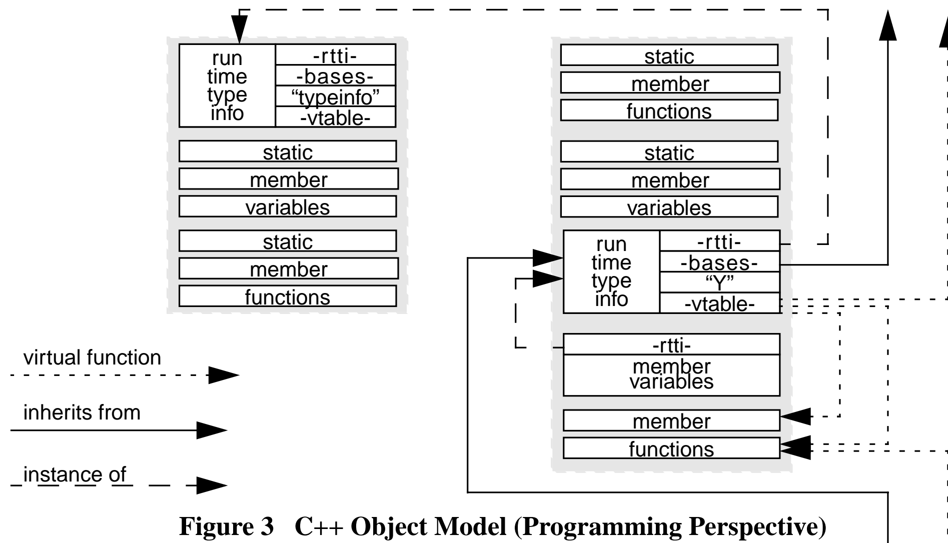


**Figure 3 C++ Object Model (Programming Perspective)**

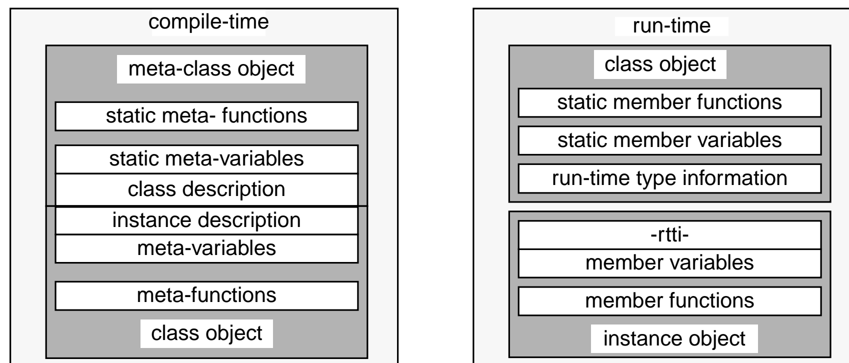


**Figure 4 FOG Object Model (Programming Perspective)**

## 2.2 Meta-Types

Each class declaration has a compile-time meta-object of the `class` meta-type. The meta-object name is the same as the declaration, so that a meta-program may access the meta-object by just specifying the declared name in order to interact with the meta-object.

`class` is just one of the meta-types supported by FOG. At compile-time there is a meta-object for each declaration and for each of its components. A partial listing of the meta-types in FOG is given in the following specialisation list (indentation denotes a specialisation from the preceding less indented meta-type).

```
token                                   --> string
    statement
        declaration
            entity                      --> identifier
                object
                    base_specifier      --> scope
                    enumerator
                        function
                    typedef
                    variable
                type
                    built_in
                    enum
                    scope
                        class
                        namespace
                        struct
                        union
                    typename
            expression
            character                   --> identifier, number, string
            number                      --> character, identifier, string
            string                      --> character, identifier, number
    list
        statement_seq                   --> statement
            declaration_seq
    iterator                            --> number
    identifier                          --> character, expression, number, string
```

A `union` is a specialisation of a `scope`, and everything is a specialisation of a `token`. A more specialised meta-type can be used in place of a less specialised meta-type, thus a `class` may be used where a `declaration` is required.The `-->` annotations against some meta-types indicate where certain free translations may be performed. An `enum` may therefore be used where an `identifier` is required, because an `enum` is more specialised than an `entity` which may be translated to an `identifier` by retaining just the entity name.

In principle, a meta-type could be defined for every production in the C++ grammar. However this would be excessive, since there can be no sensible requirement to distinguish between an *inclusive-or-expression*[1] and an *exclusive-or-expression*. The single `expression` meta-type suffices.

Meta-types are incorporated into the C++ grammar by extending relevant productions to incorporate meta-expressions (passed as a meta-function parameters or evaluated as a meta-function or meta-variable values). The enhanced grammar for an enumerator is therefore:

*enumerator:*
    *identifier*
    *enumerator-literal*

where *enumerator-literal* is a meta-expression known to be at least as specialised as `enumerator`. The grammar for a *meta-expression* is a subset of that for a *conditional-expression*, excluding constructors and casts and some aspects of pointers. A full description of the enhanced grammar will be found in [16].

## 2.3 Built-in Meta-functions

In order for meta-objects to be useful, it must be possible to manipulate them. Therefore each meta-type has an associated set of methods. The following hypothetical declaration of the `base_specifier` meta-type specifies a base class. (The declaration is hypothetical, since meta-types are built-in, although they can be extended).

```
auto class base_specifier : auto object       // meta-class base_specifier inherits from object
{
    virtual number is_auto();
    virtual number is_private();
    virtual number is_protected();
    virtual number is_public();
    virtual number is_virtual();
    virtual scope scope();
};
```

A meta-variable may be initialised with a base-specification:

```
    auto base_specifier aBaseSpecifier = public virtual BaseClass;
```

and then used as:

```
    if (aBaseSpecifier.is_virtual())
        class VirtualClass : $aBaseSpecifier { /* ... */ };
    else
        class NonVirtualClass : $aBaseSpecifier { /* ... */ };
```

A meta-program may navigate from a known class declaration using the `class::bases()` method that returns a list (array) of all the base specifiers of a class:

```
    class TestClass : public BaseClass { /* ... */ };
    if (TestClass::bases()[0].is_public()) /* ... */;
```

The behaviour is a direct extension of the run-time behaviour, making statements available as meta-statements at compile-time. Meta-expressions, meta-functions and meta-variables are used within meta-statements, in the same way as conventional expressions, functions and variables are used in statements. Access to the meta-level from the normal level (from a declaration) requires the use of a `$` expression in which a `$` prefixes or, when a lexical ambiguity arises, a `${}` surrounds a meta-expression.

All built-in meta-functions are virtual, having a default diagnostic implementation in `token`. This provides weakly-typed rather than strongly-typed execution at compile-time, and appears inconsistent with C++. C++ has strong typing in the interests of efficiency and ensuring that errors are detected at compile-time. Weak typing defers error detection till run-time. However compile-time meta-programs execute at compile-time and so weak typing is flexible and does not defer error detection till tun-time.

## 2.4 Lists and Iterators

With all built-in methods implemented as virtual functions, it is only necessary to provide a single polymorphic container for collections of meta-objects. This collection has the meta-type `list`, and was used transiently as the return from `class::bases()` in the last example.

Iteration is required to support real programming. In C++, this is achieved by `for`, `while` and `do while` statements. Each of these is supported as a meta-statement, and in order to provide some practical mechanism

1. In this paper we use the same style as in Appendix A of [14], which is substantially the same as [1]. *Italics* are used to denote non-terminals. A `fixed` width font is used for meta-type names.

for iterating over a `list` there is a further polymorphic built-in meta-type `iterator`. An `iterator` behaves as a pointer to a `token` and has the following notional declaration.

```
auto class iterator : auto token
{
public:
    iterator();
    iterator(list);
    iterator(iterator);
    nil operator=(list);
    nil operator=(iterator);
    nil operator++();
    nil operator--();
    token operator->();
    token operator*();
    operator number();
};
```

A typical idiomatic use of an iterator is shown in the following example

```
for (iterator i = ApplicationClass::variables(); i; ++i)
    if (i->is_static())
        const char *ApplicationClass::names[] = { ""$i->name() };
const char *ApplicationClass::names[] = { 0 };
```

The iteration loops over a list containing all the variable declarations of `ApplicationClass`. The iteration test uses the conversion to `number` which returns `true` so long as the iterator remains within the domain. The iteration step increments conventionally. Within the loop, the conditional statement exploits the pointer-like behaviour to indirect to the current variable declaration and determine whether it is for a static variable. If so, the following line is enabled. That line is a declaration for an array, which is assigned the name of the variable cast to a string by concatenation with the leading "" (see the companion paper [15] for details). Since the meta-expression to access the name occurs within a declaration and not a meta-statement, it is necessary to perform the access using a `$` expression. The loop apparently repeatedly declares the array for each static variable. This is not so. In FOG repeated initializations extend the array. The declarations within the loop build a list of all the names, which the declaration following the loop null-terminates. The list of static variable names is then available for use by run-time code.

Construction of, or assignment to, an iterator takes a copy of the identities of the meta-objects in the iteration domain. This avoids some problems of meta-circularity [5] that could occur if an iteration invoked a meta-function that added another object to the iteration domain.

## 2.5 Meta-Construction and Meta-Destruction

The iteration example shows an iterator at work extracting information from a class declaration at compile-time and formatting it for use at run-time. The example omits the important detail of how, when and where the code is declared and invoked. In a C program, application code is invoked because a call path exists from `main()` to the application code. In C++, application code may also be invoked as a result of construction or destruction of static objects.

The same three alternatives are provided in FOG at compile-time. Each class and built-in type may have a meta-constructor that is executed before the meta-main program, and a meta-destructor that is executed afterwards. Default empty implementations of all meta-constructors, meta-destructors and `main` are provided so that no programming is required for unused facilities. These functions are declared as:

```
auto Application::Application() { /* ... */ }             // meta-constructor.
auto main() { /* ... */ }                                  // meta main program.
auto unsigned long::~unsigned long() { /* ... */ }        // meta-destructor.
```

A conventional compilation may be viewed as having three phases:

- source file reading and analysis
- optimisation
- emission

FOG adds three phases for meta-programming:

- source file reading and analysis
- meta-construction
- meta-main execution
- meta-destruction
- optimisation
- emission

The precise behaviour of the three extra phases is a matter for further research. At present meta-constructors are invoked in an otherwise unspecified base-class to more derived class order. Meta constructors may define additional declarations. A meta-constructor cannot reliably iterate over all declarations (another meta-constructor may declare some more). Meta-construction is limited to defining facilities to be used later.

The meta-main program is currently not implemented, and perhaps should not be implemented. It is not entirely clear that a partially compiled description can be maintained efficiently in response to arbitrary application meta-programming. A potential for unpleasant effects arise if a meta-main program declares a class that then needs meta-construction, or if execution of a meta-function defines additional code for itself, or its callers.

Meta-destructors are also invoked in an otherwise unspecified base-class to more derived class order. Meta-destructors are not permitted to define additional declarations, their activity is restricted to extending already defined declarations, that is providing initializers for arrays, or code for functions. Since additional declarations are prohibited a meta-destructor can reliably iterate over the eventual declarations.

The earlier example traversing the variables of `ApplicationClass` can therefore be written as

```
auto ApplicationClass::ApplicationClass()
{
    public static const char *names[];
}
auto ApplicationClass::~ApplicationClass()
{
    for (iterator i = variables(); i; ++i)
        if (i->is_static())
            const char *names[] = { ""$i->name() };
    const char *names[] = { 0 };
}
```

These principles are now applied to a practical example.

## 3 MARSHALLING

Communication between programs requires messages to be passed between those programs. Each message is usefully represented as an object, and so the programmer is presented with the problem of transferring the contents of one object between programs. This is readily achieved using an Interface Definition Language and CORBA when such high level facilities are available, however when working at a lower level the problem must be solved by the programmer.

A typical approach involves the conversion of each object into a sequence of bytes with a common header that describes the format and length of the subsequent bytes. The sending program must marshal the data elements of each object into the byte stream and the receiving program perform the corresponding unmarshalling back into an object. Preparation of this marshalling code is straightforward, but not amenable to automation with conventional compilers. In order to show how this can be resolved by FOG, it is helpful to first show one possible conventional solution. The precise exposition exactly matches the subsequent automated solution. Numbered comments (//3.0) may assist the reader in correlating the two solutions.

All messages inherit from the `Message` class, that defines the marshalling and unmarshalling interfaces and an enumeration, whose values distinguish between each possible message format.

```
class Message
{   /* ... */
protected:
    enum MessageTypes                                                  //1.0
    {
        MESSAGE_StockReport /* , ... */                                //1.1
    };
public:
    virtual size_t marshal(unsigned char dataBuffer[]) const;          //2.0
    static Message *unmarshal(unsigned char dataBuffer[]);             //3.0
};
```

Invocation of the marshalling function fills a `dataBuffer` with the byte stream and returns the message size. The unmarshalling function is passed a byte stream and returns a pointer to an object if the message is valid, or 0 on failure. For this example, a single very simple message comprising just two data elements is used.

```
class StockReport : public Message
{   /* ... */
private:
    unsigned long _item_number;
    short _stock_level;
private:
    inline StockReport(unsigned char dataBuffer[]);                      //5.0
public:
    static StockReport *make(unsigned char dataBuffer[]);                //4.0
    virtual size_t marshal(unsigned char dataBuffer[]) const;            //2.1
};
```

The message-specific marshalling into `dataBuffer` is performed by a virtual function:

```
size_t StockReport::marshal(unsigned char dataBuffer[]) const            //2.2
{
    unsigned char *p = dataBuffer;                                       //2.3
    *p++ = MESSAGE_StockReport;            // Message type               //2.4
    *p++ = 6;                              // Message body length        //2.5
    *p++ = (_item_number >> 24) & 0xFF;                                  //2.6
    *p++ = (_item_number >> 16) & 0xFF;
    *p++ = (_item_number >> 8) & 0xFF;
    *p++ = _item_number & 0xFF;
    *p++ = (_stock_level >> 8) & 0xFF;                                   //2.7
    *p++ = _stock_level & 0xFF;
    return p - (dataBuffer + 2);                                         //2.8
}
```

Unmarshalling from `dataBuffer` must first select the message-specific routine from the message type:

```
Message *Message::unmarshal(unsigned char dataBuffer[])                  //3.1
{
    switch (dataBuffer[0])                                               //3.2
    {                                                                    //3.3
        case MESSAGE_StockReport:                                        //3.4
            return StockReport::make(dataBuffer);
        /* ... */
        default:                                                         //3.5
            return 0;        // 0 for bad message type error.
    }                                                                    //3.6
}
```

and then create the message-specific object, but only if the message length is valid:

```
StockReport *StockReport::make(unsigned char dataBuffer[])               //4.1
{
    if (dataBuffer[1] != 6)
        return 0;            // 0 for bad message length error.
    else
        return new StockReport(dataBuffer);
}
```

Finally construction performs the message specific unmarshalling:

```
StockReport::StockReport(unsigned char dataBuffer[])                     //5.1
{
    unsigned char *p = dataBuffer+2;                                     //5.2
    {                                                                    //5.3
        unsigned long temp = *p++;
        temp = (temp << 8) | *p++;
        temp = (temp << 8) | *p++;
        temp = (temp << 8) | *p++;
        _item_number = temp;
    }
    {                                                                    //5.4
        unsigned long temp = *p++;
        temp = (temp << 8) | *p++;
        _stock_level = short(temp);
    }
}
```

The marshalling and unmarshalling code is very predictable and in principle easy to write, however when there are many messages, it is tedious and error prone. When a data type is changed or a member variable added, there are many places where updates are required. It is preferable to generate the code automatically. This requires a meta-program that can reflect upon the message class declarations and generate code accordingly.

Using FOG, an application meta-program can achieve this automation, and satisfy the separation of concerns underlying Aspect Oriented Programming [10].

## 3.1 Application aspect

The marshalling support may be separated completely from the application code. The message classes now express their own inheritance relationships, their data contents, and any other application declarations that may be necessary.

```
class Message { /* ... */ };

class StockReport : public Message
{   /* ... */
private:
    unsigned long _item_number;
    short _stock_level;
};
```

## 3.2 Aspect weaving

The marshalling aspect is added (woven) by invoking the installation meta function of the `Marshal` meta-class.

```
using "Marshal.fog";                        // A better form of #include.
class Message
{
    $Marshal::install();
};
```

## 3.3 Marshalling aspect

In the following code, all declarations are provided in full to reduce the amount of unusual syntax. In practice a FOG extension to a *using-declaration* without a qualifying scope, allows a declaration to be re-used in its own scope. Repeated declarations such as

```
public virtual size_t marshal(unsigned char dataBuffer[]) const
```

can therefore be written as just

```
using marshal
```

saving many characters and reducing the maintenance burden for signature changes.

```
auto class Marshal {};
auto declaration_seq Marshal::install()
{
    auto scope MessageClass = $This;                                       //10
    protected enum MessageTypes {};                                        //1.0
    public virtual size_t marshal(unsigned char dataBuffer[]) const;       //2.0
    public static $MessageClass *unmarshal(unsigned char dataBuffer[])     //3.0,3.1
        [[pre]] { switch (dataBuffer[0]) \{ }                              //3.2,3.3
        [[post]] { default: return 0; \} }                                 //3.5,3.6
    auto ${This}()                                                         //11
    {
        protected enum ${MessageClass}::MessageTypes { MESSAGE_$This };     //1.1
        public static $MessageClass *${MessageClass}::unmarshal(unsigned char dataBuffer[])//3.1
            { case MESSAGE_$This: return ${This}::make(dataBuffer); }       //3.4
        public virtual size_t marshal(unsigned char dataBuffer[]) const;   //2.1,2.2
        public static $This *make(unsigned char dataBuffer[]);             //4.0
        private inline ${This}(unsigned char dataBuffer[])                 //5.0,5.1
            { unsigned char *p = dataBuffer + 2; }                         //5.2
    }
    auto ~${This}()                                                        //12.0
    {
        auto number byte_count = 0;                                        //12.1
        for (iterator i = variables(); i; ++i)                             //12.2
            if (!i->is_static())                                           //12.3
                i->type().marshal(i->name());                              //12.4
        public virtual size_t marshal(unsigned char dataBuffer[]) const    //2.1,2.2
            [[entry]] { unsigned char *p = dataBuffer;                     //2.3
                        *p++ = MESSAGE_$This;                              //2.4
                        *p++ = $byte_count; }                              //2.5
            [[exit]] { return p - (dataBuffer + 2); }                      //2.8
        public static $This *make(unsigned char dataBuffer[])              //4.1
            { if (*p++ != $byte_count) return 0;
              else return new ${This}(dataBuffer); }
    };
};
```

The `install` meta-function is invoked from a class declaration for `Message` and executes as part of the source file reading and analysis compilation phase. All lines declare declarations that are added to the `Message` class. A meta-variable (10), meta-constructor (11) and meta-destructor (12.0) are declared in addition to more conventional declarations (1.0, 2.0, 3.0).

The first declaration (10) caches the name of the invocation scope in the meta-variable `MessageClass` for later use by the meta-constructor. Then the marshalling interface is defined (1.0, 2.0, 3.0) for the `Message` class. The body of a C++ function is composed by FOG from the concatenation of contributions to five named regions: `entry`, `pre`, `body`, `post` and `exit`. Contributions within each region are simply concatenated, and in the absence of a region specification, contributions are placed in the `body` region. The `pre` (3.2) and `post` (3.5) regions are used to position the switch statement around contributions to the body (3.4). The loose braces of the switch statement (3.3, 3.6) are escaped to avoid a syntax conflict with the braces that delimit a *function-body.*

The meta-constructor is invoked for `Message` and all its derived classes. The meta-constructor first defines contributions to the root message class whose identity was cached in the `MessageClass` meta-variable. The

meta-constructor then defines the interface and some of the implementation for the derived message class, on whose behalf the meta-constructor is executing.

An enumerator is defined (1.1) in the `MessageTypes` enumeration of the `Message` class. The additional enumerator extends the enumeration and so acquires a unique value for each message class. The enumerator name is formed by concatenation of the prefix `MESSAGE_` to the derived message class name. A switch case is defined (3.4) for part of the default body region of the unmarshalling function of the root message class. The marshalling interface of the derived message is defined (2.1, 4.0, 5.0) and that part of the implementation that can be defined without knowledge of the contents of this class or other classes (5.2).

The meta-destructor is similarly invoked during the meta-destruction phase on behalf of each message class. It comprises a loop to resolve the member variable dependent code and count the number of bytes in the message. It finishes with those parts of the implementation that need knowledge of the byte count. The framework of the `marshal` routine is defined (2.1), in this case using `entry` and `exit` regions to surround body contributions[2]. The `make` routine is defined in its entirety (4.1).

The byte count is maintained in a meta-variable initialised to 0 (12.1). The loop (12.2) iterates over all member variables in the derived class, and (12.3) skips static member variables. Within the loop (12.4) invocation of the `marshal` meta-function for the data-type of each member variable causes emission of member specific marshalling code. The member variable name is passed as a parameter to the type-specific implementation.

```
auto declaration_seq unsigned long::marshal(identifier name)              //13.0
{
    byte_count += 4;                                                      //13.1
    public virtual size_t marshal(unsigned char dataBuffer[]) const       //2.1,2.2
    {
        *p++ = ($name >> 24) & 0xFF;                                      //2.6
        *p++ = ($name >> 16) & 0xFF;
        *p++ = ($name >> 8) & 0xFF;
        *p++ = $name & 0xFF;
    }
    private inline ${This}(unsigned char dataBuffer[])                    //5.1
    {
        {                                                                 //5.3
            unsigned long temp = *p++;                                    //5.4
            temp = (temp << 8) | *p++;
            temp = (temp << 8) | *p++;
            temp = (temp << 8) | *p++;
            $name = temp;
        }
    }
}
```

Meta-functions can be defined for built-in types as well as user defined types. The above declaration for the `unsigned long` 'class' supports the marshalling of `unsigned long` member variables. The formal parameter `name` is replaced throughout the body before the body is interpreted in the invoking context, that of the derived message class. The update of the `byte_count` (13.1) therefore maintains the counter of the derived message class, and the two declarations (2.1, 5.1) provide additional code for the body region of the derived message class routines. The member variable iteration is in declaration order, and the ordering of function body contributions is preserved, so the final ordering of the many contributions is well-defined. The contributed code (2.6, 5.3) just performs the very simple operations appropriate to the data type.

A similar isomorphic meta-function for `short` is needed to complete the example (2.7, 5.4), and further routines for every other primitive data type. Nested data types can be resolved by a nested iteration, which can be specified as a general-purpose meta-function:

```
auto declaration_seq Marshal::marshal(identifier name)
{
    for (iterator i = variables(); i; ++i)
        if (!i->is_static())
            i->type().marshal(${name}.$i->name());
};
```

It can be installed by meta-inheritance for use in a nested type

```
struct NestedDataType : auto Marshal { /* ... */ };
```

This declares `Marshal` as an additional base class of the `NestedDataType`, but only at (meta-)compile time. The names of `Marshal` are therefore visible to the derived class, providing the required resolution of `NestedDataType::marshal()`.

This example shows how application code can be generated in response to the actual application declarations. The code is fully under the programmer's control. The programmer can freely choose an alternate implementation using data tables to describe each message rather than monolithic functions. Inheritance of messages can be accommodated by extending the iterations to traverse base classes, or by changing the run-

2. In this example, use of `entry` and `exit` rather than `pre` and `post` is purely a matter of style. A `return` should only occur from `exit`. A variable used by the `return` should be declared in `entry`.

time code to invoke base class methods. More sophisticated code can be provided to support swizzling of pointer types for database applications.

The generated code is portable, since all members are referred to by name. The example code for `unsigned long::marshal` has a portability problem for processors with a greater than 32 bit `unsigned long`, but this is a limitation of the example solution, not of the approach.

# 4 RELATED WORK

Meta-classes were first introduced to support the configuration of objects at run-time in Smalltalk, and have subsequently become an important part of most Object Oriented languages. Limitations of the Smalltalk implementation and the ObjVlisp resolution are discussed by Pointe [6]. Maes [9] argues that the pure Object Model fails to distinguish the meta-level adequately. The restricted Object Model in FOG does precisely this; only meta-objects exist at compile-time, and only real objects at run-time. A more versatile Object Model allowing the inheritance of meta-classes to differ from their classes is supported by CLOS and SOM. The resulting compatibility problems are discussed in [2]. These problems do not arise in FOG since the two inheritance hierarchies are the same. They also do not arise since there is no object creation at meta-compile time and so no level traversal. A motivation for distinct meta-class inheritance is to avoid the propagation problem whereby a concrete class inherits the inappropriate property of abstractness from its abstract base class. This problem does not arise in FOG, since C++ offers at least two distinct solutions to the original problem using pure virtual functions or protected constructors.

The lack of meta-classes has always been a deficiency of C++, for which various proposals were suggested during standardisation. Eventually the standardisation committee compromised on the relatively limited functionality known as Run-Time Type Information (RTTI). A more substantive proposal [3], is largely proprietary and so it is difficult to assess accurately. It defines a run-time Meta-Information Protocol providing more extensive data structures with global functions to support iteration. FOG provides a compile-time meta-level, in which application meta-programs may be used to create whatever run-time data structures are appropriate. These may vary from just class names to large descriptive tables for use by a run-time environment that supports run-time meta-programming.

OpenC++ [4] also extends C++ to provide a compile-time meta-level. OpenC++ follows the traditional CLOS Meta Object Protocol approach [8], providing hooks for the meta-programmer to intercept and change the syntax tree during a wide variety of compilation activities such as declaring or accessing a member variable. This is a very powerful approach, but requires the meta-programmer to have a good understanding of the compiler internals. Code to modify the syntax tree can be a little unwieldy. The CLOS approach is appropriate for a language that configures its objects at run-time: the configuration calls already exist, it is just a matter of formalising them to allow a programmer to use them. The C++ philosophy is for the program structure to be resolved at compile time so that run-time activity is minimised. The meta-level in FOG is much more in keeping with C++ philosophy, the only hooks for meta-programming are in the meta-constructors and meta-destructors of each data type. These provide sufficient flexibility for meta-programs to browse their declarations and generate customised run-time type-dependent code. FOG does not support rewriting of expressions, since FOG operates only on declarations. However, if inline accessor functions are used for member variables, FOG can rewrite those functions creating a similar effect to rewriting expressions. Because FOG operates only on declarations, it is possible to define FOG's modifications to declarations using natural extensions of declaration syntax. This leads to compact and at first sight cryptic programs that are arguably still C++.

A difficult problem of meta-circularity arises when more than one meta-program operates upon the same program. Which meta-program operates first? Does the second meta-program operate on the source or the results of the first meta-program? How does a meta-program behave when it changes it's own declarations? Chiba [5] argues that each level of the reflection tower [13] must exist so that each meta-program operates consistently on the results of a more nested program and is isolated from its own and less nested meta-programs. The complexity of arranging this parallels the current failure to implement a meta-`main()` in FOG. The restrictive functionality of meta-construction and meta-destruction may be an appropriate comprise between infinite extensibility and practical meta-programming problems. Some difficult problems of compatibility also arise when multiple independently sound contributions to a single declaration (normally a function) conflict. Ossher [12] considers the problem to be a severe weakness for Aspect Oriented Programming. Mulet [11] examines this from a functional language perspective and argues for composition by nested function calls.

# 5 CONCLUSION

A companion paper [15] introduced meta-functions and meta-variables, meta-expressions and meta-statements and showed how they replaced the substitution and conditionalisation functionality of Cpp rendering Cpp redundant.

In this paper, meta-functions and meta-variables have been defined within the context of meta-classes, so that a meta-programmer perceives very similar programming constructs for meta programs as are used for normal programs. Meta-types associated with the meta-objects for each program declaration and its components have been defined. Built-in meta-functions that support meta-object usage in meta-programs have been identified.

A weakly meta-typed execution model has been described for compile-time meta-programs so that a single polymorphic list and associated iterator class is sufficient for practical meta-programming. This is demonstrated by a practical example involving data marshalling.

FOG supports meta-programming using syntax extensions to C++ and the programming styles of C++. The meta-level 'optimised away' by the C++ language definition has been restored at compile-time without compromising C++'s strengths. Run-time is reduced by compile-time activity. No run-time cost is incurred for unused functionality.

## 6 ACKNOWLEDGEMENTS

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

## 7 REFERENCES

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

[2] Bouraqadi-Saâdani, N.M.N., Ledoux, T., and Rivard, F. Safe Metaclass Programming. Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98, ACM SIGPLAN Notices, vol. 33, no. 10, 84-96, October 1998.

[3] Buschmann, F., Kiefer, K., Paulisch, F., and Stal, M. A Run-Time Type Information System for C++. 1992. http://jerry.cs.uiuc.edu/reflection/utrecht/paulisch.ps

[4] 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, vol. 30, no. 10, 285-299, October 1995. http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-oopsla95.ps.gz

[5] Chiba, S., Kiczales, G., and Lamping, J. Avoiding Confusion in Metacircularity: The Meta-Helix. Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software, ISOTAS'96, LNCS 1049, 157-172, 1996. http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-isotas96.ps.gz

[6] Cointe, P. Metaclasses are First Class: The ObjVlisp Model. Proceedings of the 1987 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, ACM SIGPLAN Notices, vol. 22, no. 12, 156-167, December 1987.

[7] Ellis, M.A., and Stroustrup, B. The Annotated C++ Reference Manual. Addison-Wesley, 1990.

[8] Kiczales, G., des Rivières, J., and Bobrow, D.G. The Art of the Metaobject Protocol. MIT Press, 1991.

[9] Maes, P. Concepts and Experiments in Computational Reflection. Proceedings of the 1987 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, ACM SIGPLAN Notices, vol. 22, no. 12, 147-155, December 1987.

[10] Mens, K., Lopes, C., Tekinerdogan, B. and Kiczales, G. Aspect-Oriented Programming workshop report. In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97. In Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1357, Springer-Verlag, June 1997 http://www.parc.xerox.com/spl/projects/aop/ecoop97/aop-ecoop97-proceedings.pdf

[11] Mulet, P., Malenfant, J., and Cointe, P. 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.

[12] Ossher, H., and Tarr, P. Operation-Level Composition: A Case in (Join) Point. In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98. In Workshop Reader of the European Conference on Object-Oriented Programming(ECOOP), LNCS?, Springer-Verlag, June 1998.
proceedings.pdfhttp://wwwtrese.cs.utwente.nl/aop-ecoop98/papers/Ossher.pdf

[13] Smith, B.C. Reflection and Semantics in Lisp. Proceedings of the 11th Annual Symposium on Principles of Programming Languages, 23-35, January 1984.

[14] Stroustrup, B. The C++ Programming Language. Third edition. Addison-Wesley, 1997.

[15] Willink, E.D., and Muchnick, V.B. Preprocessing C++ : Substitution and Composition. Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems, June 1999. http://www.ee.surrey.ac.uk/Research/CSRG/fog/FogToolsEE1.{gz,pdf,ps,zip}

[16] Willink, E.D. Meta-compilation for C++. PhD Thesis, Department of Computing, University of Surrey, (in preparation) 1999. http://www.ee.surrey.ac.uk/Research/CSRG/fog/FogThesis.{gz,pdf,ps,zip}