Open C++ Release 1.2
Programmer's Guide*

Shigeru Chiba

Department of Information Science
University of Tokyo
E-mail: `chiba@is.s.u-tokyo.ac.jp`

1 Introduction

Open C++ is a variant of the C++ language, in which method calls and variable access are made open-ended. The implementation of method calls and variable access is extensible. By using this facility, the programmer can implement a new language function, such as remote procedure calls, on top of the language. An object is controlled by its metaobject so that the metaobject alters the behavior of method calls to that object and also access to its variables. Because a metaobject is a C++ object, the programmer can easily define a new metaobject to implement a new function; rebuilding the compiler is not necessary.

Open C++ adopts a model in which an object has a single metaobject. An object does not share a metaobject with other objects. The ability of a metaobject is simple; a metaobject can control only method calls and variable access (member function calls and member variable access). It cannot customize the rule of multiple-inheritance unlike CLOS MOP. The simplicity of Open C++, however, enables efficient implementation of the language.

A program of Open C++ is a C++ program containing special directives that declare methods and variables controlled by a metaobject. The directives are embedded in the C++ program as comments that start with `//MOP`.* The programmer alter the behavior of an object by specifying the class of its metaobject. Such an object that behaves in a different manner by its metaobject is called a reflective object to distinguish it from a normal object, which has no metaobject and behaves in the default manner.

1.1 A Primary Example

First of all, we show a primary example of Open C++ programs. In this example, we attempt to print a message on the console every method call to an object. The class of the called object is `Person`. It is controlled by a metaobject,

*Shigeru Chiba, "Open C++ Programmer's Guide," Technical Report No. 93-3, Dept. of Information Science, University of Tokyo, 1993

*MOP means *metaobject protocol*.

which alters the implementation of method calls to that object. See the following code.

```
// ---- person.c ----

#include <stdio.h>
#include "prnmetaobj.h"      // declaration of a metaobject

class Person {
public:
    Person(char* name, int age);
    int Age();
//MOP reflect :
    int IncAge();

private:
    char* name;
    int   age;
};
```

The file "`prnmetaobj.h`" includes the class declaration of a metaobject that controls an object of the class `Person`. The details of this file are shown latter. The class `Person` has two instance variables `name` and `age`, and three instance methods. The method `IncAge()` increases the age by one and then returns the new age. A method `Age()` returns the current age. The line that starts with `//MOP` indicates that execution of the following methods, i.e. `IncAge()` in this case, is controlled by a metaobject. We call them `reflect` methods. The other methods are executed normally. Note that this file is a complete C++ code and accepted by the C++ compiler. The class `Person` is an ordinary C++ class.

```
// ---- person.c (cont.)----

//MOP reflect class Person : PrintMetaObj;

Person::Person(char* n, int a) {
    name = n;
    age = a;
}

int Person::IncAge() {
    return ++age;
}

int Person::Age() {
    return age;
}
```

The second comment line starting with `//MOP` :

```
//MOP reflect class Person :  PrintMetaObj;
```

indicates that an object of the class `Person` can be reflective and its metaobject is an object of a class `PrintMetaObj`. A reflective object is controlled by its metaobject to alter the implementation of method calls and variable access. The compiler reads this line and then produces a new subclass of the class `Person`. The name of that subclass is `refl_Person`. The compiler adds the keyword '`refl_`' at the head of the name of the original class. If you want to create a reflective object of the class `Person`, then you create it as an object of the subclass `refl_Person`. An object created as an instance of the class `Person` is not reflective but a normal C++ object. See the rest of the program.

```
// ---- person.c (cont.) ----

void main() {
    refl_Person student("Billy", 24);
    printf("age %d\n", student.Age());
    printf("age %d\n", student.IncAge());
}
```

Because the object `student` is reflective, a call `student.IncAge()` makes a metaobject print a message. A metaobject of the class `PrintMetaObj` alters the implementation of its object so that a message is printed on the console every time `reflect` methods of its object are called. Note that a call `student.Age()` does not make a metaobject print a message since the method `Age()` is not a `reflect` method. A metaobject can control only `reflect` methods of its object. Non-`reflect` methods are executed in the normal C++ manner. Now, we can compile and execute the program.

```
% OCC person.c
% OCC prnmetaobj.c                meta-program described later
% CC person.o prnmetaobj.o .occ/reflect.a libOpc.a
% a.out
age 24
*** IncAge() was called.
age 25
%
```

The Open C++ compiler (`OCC`) compiles a program (`person.c`) and produces an object file (`person.o`). It also produces a library file `reflect.a`, which must be linked together with the object file to run the program. This library file is produced at the directory `./.occ`.

1.2 The Backstage of the Example

Here, we show the backstage of the example described above. To obtain the metaobject used in the example, the class `PrintMetaObj` must be defined as:

```
// ---- prnmetaobj.h ----

#include "metaobj.h"

class PrintMetaObj : public MetaObj {
public:
    void Meta_MethodCall(Id method_id, Id category,
                         ArgPac& args, ArgPac& reply);
};
```

The classes of metaobjects must inherit from the class `MetaObj`. Except this restriction, they are normal C++ classes. The class `PrintMetaObj` redefines the method `Meta_MethodCall()` to print a message every time `reflect` methods are called. If the method `IncAge()` of the object `student` is called, then this call is trapped and the method `Meta_MethodCall()` of the metaobject is executed instead of the method `IncAge()`. The method `Meta_MethodCall()` takes four parameters. ‘`method_id`’ and ‘`category`’ specify the identifier and the category name of the called method, respectively. ‘`args`’ is a reference to calling parameters. ‘`reply`’ is a reference to a location at which the return value is stored.

```
// ---- prnmetaobj.c ----

#include <stdio.h>
#include "prnmetaobj.h"

void PrintMetaObj::Meta_MethodCall(Id m_id, Id category,
                                   ArgPac& args, ArgPac& reply) {
    printf("*** %s was called.\n", Meta_GetMethodName(m_id));
    Meta_HandleMethodCall(m_id, args, reply);
}
```

The method `Meta_MethodCall()` prints a message, and then executes the called method `IncAge()` by the method `Meta_HandleMethodCall()`, which is inherited from the class `MetaObj`. The method `Meta_HandleMethodCall()` actually executes the method specified by ‘`m_id`’. The return value is stored at the location referred by ‘`reply`’. After the execution of the method `Meta_MethodCall()`, the value stored at ‘`reply`’ is retrieved and passed to the caller as the return value of the called method `IncAge()`. Figure 1 illustrates this flow.

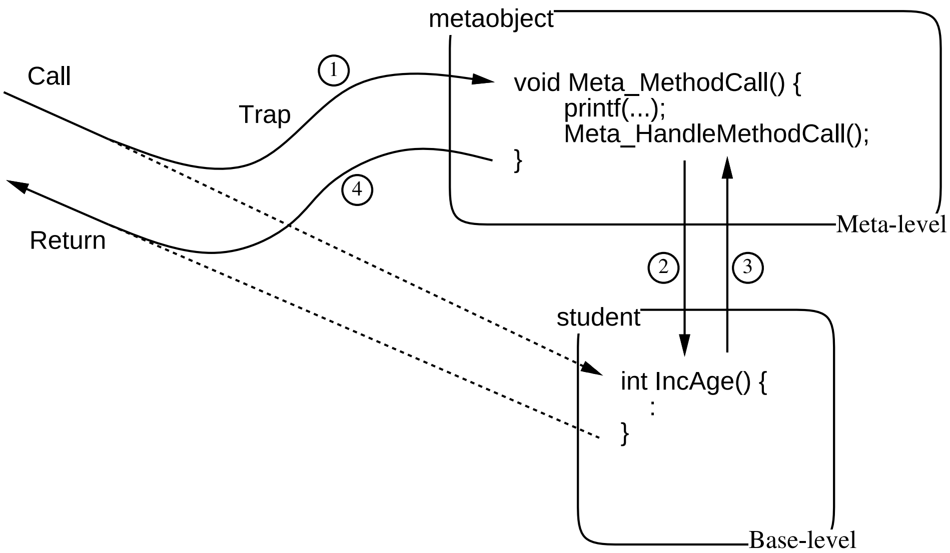

Figure 1: The Meta-Architecture of This System

1.3 Summary of the Concepts

Before going to the next step, we summarize the concepts and keywords that appear in this section. This is helpful for you to understand the following sections.

In Open C++, there are two kinds of objects: a reflective object and a non-reflective one. A reflective object is controlled by its metaobject so that the implementation of that object is altered according to the metaobject. On the other hand, a non-reflective object is a normal C++ object. It does not have a metaobject.

The class of a reflective object must be declared as being reflective. Part of its methods and variables are specified to be `reflect` methods and variables, which is controlled by a metaobject. An object of a reflective class is not always reflective. You can select it by using class names `refl_X` or `X` (where `X` is the name of a reflective class). A non-reflective object of a reflective class is completely the same as a normal C++ object. It cannot become a reflective object later. Also, `reflect` methods and variables of a non-reflective object are treated as normal methods and variables.

A metaobject controls method calls and variable access to its (reflective) object so that it implements them in a different way. Every method and variables of a reflective object is not controlled by a metaobject, however. Only `reflect` methods and variables are controlled. The other methods and variables are implemented in the same manner as normal C++ ones are.

2 Directives at the Base Level

To specify a reflective class and `reflect` methods/variables, special directives are embedded in the C++ code. There are two directives: both of them are comments that start with "`//MOP`".

The first directive is for specifying reflective classes.

`//MOP reflect class` *name1, name2,*`...` `:` *metaobject_class*`;`

This directive specifies that classes *name1, name2,...* are reflective and the class of their metaobjects is *metaobject_class*.

(!) Because of implementation matters, nothing except declarations can appear before this directive. Declarations involving memory assignment is also prohibited. You cannot write code that must appear only once in the program, for example, allocating global-variables, defining functions, and initializing `static` member variables.

The second directive is for specifying `reflect` methods and variables.

`//MOP reflect :` *or* `//MOP reflect(` *category_name* `) :`

This directive is similar to an access specifier, such as `public` and `private`. Methods and variables following this directive until the end of the class or until another access specifier is encountered are specified as `reflect` methods and variables. This directive can take a *category_name*. If a category name is specified, then a metaobject recognizes the `reflect` method or variable as to have the specified category name. A metaobject may change the implementation of method calls and variable access according to category names. The necessity of a category name depends on a metaobject. If a metaobject needs a category name, then you must specify it by this directive, otherwise specifying it is meaningless; even if you specify a category name, it is simply ignored.

The access rule of a `reflect` method and variable is made one specified by the access specifier preceding this directive. For example,

```
class X {
public:
    void f();
//MOP reflect:
    void g();
protected:
//MOP reflect:
    void h();
    int p;
};
```

the method `g()` is `public`, but the method `h()` and the variable `p` are `protected`.

Because of implementation matters, a `reflect` method and variable must satisfy the following conditions.

- a `reflect` method and variable are `public` or `protected`,
- a `reflect` method is a `virtual` function (*recommended*),
- a variable argument list are not available, and
- an overloaded operator cannot be reflective.

The second condition is recommended. Because a class `refl_X` is implemented as a subclass of `X`, if all the `reflect` methods are `virtual` functions, then an object of the class `refl_X` is dealt with completely as if it were an object of the class `X` except at the creation.

2.1 The Constructor of a Reflective Object

The constructor of a reflective object may take additional arguments besides the arguments of its original constructor. If the constructor of a metaobject takes arguments, then the constructor of its object needs those arguments. For example,

```
class M : public MetaObj {        // metaobject
public:
    M(int p);
    ...
};

class X {
public:
    X(int i, int j);
    ...
};

//MOP reflect class X : M;
```

Because the constructor of a metaobject takes an `int` argument, the constructor of a reflective object takes two arguments for that of the original class and one for that of the metaobject.

```
class refl_X {
public:
    refl_X(int i, int j, int p);
    ...
};
```

The arguments of a metaobject's constructor are appended at the tail of the arguments of its object's. In the current implementation, a metaobject cannot have more than one constructors, although an object can have any number of constructors. The default arguments of the metaobject's constructor are supported, whereas those of the object's constructors are not; they are simply ignored.

The constructor of a reflective object may be a `reflect` method. That is, the creation of a reflective object may be controlled by a metaobject. A reflective object is created according to a metaobject, however, only when it is allocated in the heap by the `new` operator. For example, suppose that the constructor of a reflective class `X` is a `reflect` method.

```
void f() {
    refl_X       x1;              // normal creation
    refl_X* p = new refl_X;       // reflective creation
    ...
}
```

Since a reflective object pointed to by `p` is created with the `new` operator, a metaobject controls its creation. On the other hand, since a reflective object `x1` is allocated on the stack, the creation of `x1` is not controlled by a metaobject. This is because the size of the object created according to a metaobject is unknown at the compile time. The metaobject may create an object of a class different from `X`. In C++, an object whose size is unknown at the compile time cannot be allocated on the stack.

2.2 Arguments of a `reflect` Method

When a `reflect` method is called, arguments of that method are marshaled and collected into a container object of the class `ArgPac`. Those arguments are retrieved from this object and unmarshaled when they are needed. An `ArgPac` object represents the stack frame of a method call. Marshaling and unmarshaling of atomic types, such as `int`, `char`, `float`, and `double`, are supported by the compiler. Pointers and references to any type, and aliases of atomic types are also supported. They are (un)marshaled automatically by the compiler. The other types must be (un)marshaled explicitly by the programmer.

An `ArgPac` object behaves as a stack. It has the push and pop methods for each atomic type. The class `ArgPac` defines the following methods:

```
void PushInt(int);
int PopInt();
void PushDouble(double);
double PopDouble();
void PushPtr(void*);
void* PopPtr();
void PushStr(char*);
char* PopStr();
void PushBin(char*, int);
char* PopBin();
```

The methods `PushStr()` and `PopStr()` are for a null terminated string. They copy a whole string from/into an `ArgPac` object although `PushPtr()` and `PopPtr()` copy only a pointer value. The methods `PushBin()` and `PopBin()` are for a binary data. The second argument of the method `PushBin()` is a byte-length of the binary data.

The compiler pushes arguments of a method call onto this object in the same manner as the C++ compiler pushes them onto the stack. The last argument is pushed first, and the first argument is pushed last. For example, suppose that the first argument is an integer value `a` and the second is a double-floating point value `b`. The compiler produces the following code to push them into an `ArgPac` object `args`.

```
args.PushDouble(b);
args.PushInt(a);
```

When the values are retrieved, the order of retrieving is reversed.

```
a = args.PopInt();
b = args.PopDouble();
```

If an argument is `class` (, `struct`, or `union`) type, then the programmer must define both marshaling and unmarshaling methods. The class of such an argument must define some `public` methods. For example, suppose that the class of the argument is *arg_class*. This class must define the following methods.

```
arg_class(ArgPac&);
void Pack(ArgPac&);
void Unpack(ArgPac&);
```

The first is a constructor of this class, which is used for unmarshaling. It takes an `ArgPac&` argument and initializes its own contents (instance variables), by retrieving data from the `ArgPac` object. The second is a marshaling method. The method `Pack()` takes an `ArgPac&` argument and stores its own contents in this `ArgPac` object. The last is another unmarshaling method, which works in the same manner as the first one. The first unmarshaling method is used for creating a new object, whereas the last one is for assigning a value to an existing object.

To facilitate the implementation of (un)marshaling methods, some utility classes are provided. For example, the class `argString` is used to represent a null-terminated string, instead of a `char` pointer. It defines marshaling and unmarshaling methods.

```
class argString {
public:
    argString(char* p = 0) { str = p; }
    char* CharP() { return str; }
    char* operator = (char* p) { str = p; return p; }
    operator char* () { return str; }

// for (un)marshaling
    argString(ArgPac& args) { str = args.PopStr(); }
    void Pack(ArgPac& args) { args.PushStr(str); }
    void Unpack(ArgPac& args) { str = args.PopStr(); }
private:
    char*       str;
};
```

In this example, the contents are a single instance variable. If there are several instance variables, they are pushed into and popped from an `ArgPac` object one-by-one in a fixed order. You can store any data in an `ArgPac` object. The protocol of marshaling and unmarshaling is responsibility of the programmer.

You may want to use the << and >> operators to implement (un)marshaling methods, as well as `PushInt()` etc. In `ArgPac`, those operators are overloaded for `char`, `int`, `double`, `void*`, and `char*`. The following codes, for example, are identical.

```
args << 3;
args.PushInt(3);
```

Here, `args` is an `ArgPac` object. Note that the operators overloaded for `char*` deal with a character string instead of a `char` pointer. You must take care when a `char` pointer value is (un)marshaled by these operators. Explicit casting to `void*` is needed.

The class `ArgPac` has also marshaling and unmarshaling methods. Thus, an `ArgPac` object can be an argument of a `reflect` method. Because this class is used internally by the compiler, however, you must take care of its use.

(!) After creating an `ArgPac` object, its `BeginSettingArgs()` method must be called. This method clears the contents and prepares to store values into this object. If `PushInt()` etc. are called without calling this method, any results are never guaranteed.

Because of this protocol, the programmer should use the class `argStack`, which is a subclass of `ArgPac`, if creation of `ArgPac` objects is necessary. The constructor of that class calls `BeginSettingArgs()` correctly. The class `argStack` can be an argument of a `reflect` method.

2.3 `reflect` Variables

You can specify a `reflect` variable as you specify a `reflect` method. The type of a `reflect` variable is restricted in the same way as arguments of a `reflect` method is. A `reflect` variable of a `class` type must have methods `Pack(ArgPac&)` and `Unpack(ArgPac&)` for (un)marshaling. One of its constructors must be a form like *class_name*`(ArgPac&)`. Furthermore, a `reflect` variable of a reference type is not supported in the current implementation.

When you read a `reflect` variable, you must add `()` to the name of the `reflect` variable. This parenthesis is not needed when you assign a value by the = operator.

```
class Y;

class X {
public:
//MOP reflect:
     int a;
     Y*   y;
     ...
};

//MOP reflect class X : M;       // M is the class of a metaobject.

refl_X objx;
```

For example, if you want to read the variable `a` of `objx`, then you describe `objx.a()` instead of `objx.a`. You can omit adding `()` in these cases:

- You access a member of the `reflect` variable by using the `->` operator. i.e. when you describe like `objx.y()->foo`, you can do like `objx.y->foo`.
- the type of the `reflect` variable is inferred implicitly in the expression, or explicitly by a cast. For example, if the `reflect` variable `a` appears in an expression `objx.a() + 1`, then `()` can be omitted.

These exceptions are because a `reflect` variable is implemented with a special object in which a cast operator and the =, `->` and `()` operator are overloaded so that you can access the `reflect` variable transparently.

A reflective object has two housekeeping methods mainly used by its metaobject. One is `Meta_refl_Copy()`, and the other is `Meta_refl_Init()`.

```
void refl_X::Meta_refl_Copy(ArgPac&);
void refl_X::Meta_refl_Init(ArgPac&);
```

By these methods, a metaobject copies whole contents of `reflect` variables at a time. The former pushes the values of all `reflect` variables onto an `ArgPac` object. The latter retrieves values from the `ArgPac` object and then reinitializes the `reflect` variables. These methods are used, for example, when a metaobject implements a replicated object.

You can define `Meta_Copy()` and `Meta_Init()` to extend these methods. They are called from the inside of `Meta_refl_Copy()` and `Meta_refl_Init()`. Due to implementation matters, `Meta_Copy()` and `Meta_Init()` must be `public` or `protected`. The method `Meta_refl_Copy()` calls `Meta_Copy()` before copying `reflect` variables. On the other hand, `Meta_refl_Init()` calls `Meta_Init()` after restoring `reflect` variables. Using these methods, you can define additional actions to do when a metaobject calls `Meta_refl_Copy()` and `Meta_refl_Init()`. For example, you can copy and restore non-`reflect` variables together with `reflect` variables. For example, suppose that you define as:

```
class X {
public:
    ...
    struct P {int x,y;} *v;   // not a reflect variable
//MOP reflect:
    int a, b;

protected:
    void Meta_Copy(ArgPac& p);
    void Meta_Init(ArgPac& p);
};

//MOP reflect class X : M;    // M is the class of a metaobject

void X::Meta_Copy(ArgPac& arg) {
    // do anything necessary
    arg.PushInt(v->x);        // copying v
    arg.PushInt(v->y);
}

void X::Meta_Init(ArgPac& arg) {
    v = new P;
    v->y = arg.PopInt();      // restoring v
    v->x = arg.PopInt();
    // do anything necessary
```

```
}
```

`Meta_Copy()` stores values pointed to by `v` into `arg`, and `Meta_Init()` restores those values explicitly. Note that the class `P` of the variable `v` does not need to have the methods `Pack()` and `Unpack()`. The following is the definition of methods `Meta_refl_Copy()` and `Meta_refl_Init()` of a class `refl_X`.

```
void refl_X::Meta_refl_Copy(ArgPac& p) {
    Meta_Copy(p);          // calling a programmer-defined method
    p.PushInt(a);
    p.PushInt(b);
}

void refl_X::Meta_refl_Init(ArgPac& p) {
    b = p.PopInt();
    a = p.PopInt();
    Meta_Init(p);          // calling a programmer-defined method
}
```

2.4 Utility Methods

A reflective object has two methods to get its metaobject. They are `Base_GetMetaObj()` and `Base_GetMetaObject()`. Although both of them return a pointer to the metaobject, the type of the returned pointer is different: the former is `MetaObj*`, which is a super class of every metaobject, whereas the other one is the real class of the metaobject.

3 Defining a Metaobject

You can define a new metaobject to implement a function that you want. The class of a metaobject must be a subclass of `MetaObj`. This section shows the metaobject protocol of Open C++ —how to define a subclass of `MetaObj`.

3.1 Metaobject Protocol

The class `MetaObj` defines methods for a metaobject to control its object. You obtain a new implementation of method calls and variable access by redefining those methods in your subclass. This is the declaration of the class `MetaObj`:

```
class MetaObj {
public:
    void Meta_MethodCall(Id method_id, Id category,
                         ArgPac& args, ArgPac& reply);
    void Meta_Assign(Id var_id, Id category, ArgPac& args);
    void Meta_Read(Id var_id, Id category, ArgPac& reply);

    void Meta_StartUp();
    void Meta_CleanUp();

protected:
    void Meta_HandleMethodCall(Id method_id, ArgPac& args, ArgPac& reply);
    void Meta_HandleAssign(Id var_id, ArgPac& args);
    void Meta_HandleRead(Id var_id, ArgPac& reply);
    void Meta_AssignValue(Id var_id, ArgPac& args);
```

```
    void Meta_ReadValue(Id var_id, ArgPac& reply);

    const char* Meta_GetClassName();
    const char* Meta_GetMethodName(Id method_id);
    const char*  Meta_GetVarName(Id var_id);
    ...
};
```

The methods defined by the class `MetaObj` are divided into three categories. The first is methods that implements method calls and variable access. The second is methods that a metaobject uses to manipulate its object. The last is miscellaneous methods. In addition to those methods, the class `MetaObj` defines methods to control object creation. Those methods are addressed in the following section.

The first category includes three methods: `Meta_MethodCall()`, `Meta_Assign()`, and `Meta_Read()`. They are executed when a `reflect` method is called or a `reflect` variable is accessed. You can consider that these methods substitute for the default implementation of method calls and variable access. A subclass of `MetaObj` redefines the methods of this category to obtain a new implementation of method calls and variable access.

- `Meta_MethodCall(Id method_id, Id category, ArgPac& args, ArgPac& reply)` *Overridable*
  This method implements method calls and is executed when a `reflect` method is called. The arguments `method_id` and `category` specify the identifier of the called method and its category name. The argument `args` specifies the argument list, which is marshaled. The argument `reply` is a value-result parameter, and specifies the return value of the called method. The value stored in this object is passed to the caller.

- `Meta_Assign(Id var_id, Id category, ArgPac& args)` *Overridable*
  This method implements updating a variable. It is executed when a `reflect` variable is attempted to be updated. The argument `var_id` and `category` specify the identifier of the updated variable and its category name. The argument `args` specifies the reference to the value that will be assigned to the variable. Note that `args` does not specify the assigned value itself.

- `Meta_Read(Id var_id, Id category, ArgPac& reply)` *Overridable*
  This method implements reading a variable. It is executed when a `reflect` variable is attempted to be read. The argument `var_id` and `category` specify the identifier of the read variable and its category name. The argument `reply` is a value-result parameter, and specifies the reference to the read variable. Note that `reply` specifies not the read value but the reference to the location at which the read value is stored. The value is read from the variable that `reply` refers to.

The identifiers of `reflect` methods and variables are assigned by the compiler. The constants `Meta_Init_ID` and `Meta_Copy_ID`, however, are always assigned to the methods `Meta_refl_Init()` and `Meta_refl_Copy()`. Because you cannot know the identifiers before the program is compiled, to distinguish methods and variables, you use category names. If a `reflect` method or variable has no category name, then its identifier is the constant `Category_none`, which is zero. You can define a new category name freely by defining constants in the program.

If you use a category name '*cname*', then you must define a constant `Category_`*cname* and assign any integer value other than zero to it. The identifier of a method with the category name '*cname*' is the constant `Category_`*cname*.

The second category includes five methods. A metaobject uses these methods to manipulate its object. They are language primitives and provide these functions: to execute a method, to assign a value to a variable, and to read a variable. You must not redefine these methods in a subclass.

- `Meta_HandleMethodCall(Id method_id, ArgPac& args, ArgPac& reply)`
  This method actually executes a method of this metaobject's object. The argument `method_id` specifies the identifier of the executed method. The actual argument list is retrieved from `args`, and the return value is stored in `reply`. This method is normally called by the method `Meta_MethodCall()`.

- `Meta_HandleAssign(Id var_id, ArgPac& args)`
  This method actually assigns a value to a variable of this metaobject's object. The argument `var_id` specifies the identifier of the variable. This method assigns the value that the pointer stored in `args` refers to. Note that it does not assign the value stored in `args`. This method is normally called by the method `Meta_Assign()`.

- `Meta_HandleRead(Id var_id, ArgPac& reply)`
  This method returns a reference to a variable of this metaobject's object. The variable is specified by the argument `var_id`. The argument `reply` is a value-result parameter, and the reference to the read value is stored in it. This method is normally called by the method `Meta_Read()`.

- `Meta_AssignValue(Id var_id, ArgPac& args)`
  This method assigns the value stored in `args` directly to the variable specified by `var_id`. Note that this method is different from `Meta_HandleAssign()`.

- `Meta_ReadValue(Id var_id, ArgPac& reply)`
  This method retrieves the value from the variable specified by `var_id`, and stores it in `reply`. The argument `reply` is a value-result parameter.

The last category includes house-keeping methods. The first two methods are for programmers to define startup and cleanup actions. You can redefine these methods freely in a subclass. The latter two methods are to get some information of an object.

- `Meta_Startup()` *Overridable*
  This method is called after a metaobject and its object is created and they are associated with each other. You cannot define code in a constructor of a metaobject to manipulate its object. Such code must be defined in this method. This is because most methods of the class `MetaObj` are not available in a constructor of a metaobject.

- `Meta_Cleanup()` *Overridable*
  This method is called just before a metaobject and its object is deleted. As in `Meta_Startup()`, you define code in this method to do a cleanup action. You must not define such a code in a destructor of a metaobject.

- `Meta_GetClassName()`
  This method returns a `const` pointer to the class name.

- `Meta_GetMethodName(Id method_id)`
  This method returns a `const` pointer to the name of the `reflect` method specified by `method_id`.

- `Meta_GetVarName(Id var_id)`
  This method returns a `const` pointer to the name of the `reflect` variable specified by `var_id`.

3.2 Reflective Creation

If a constructor of an object is a `reflect` method, the object creation by that constructor is controlled by a metaobject. The metaobject exactly does not control the creation, however. The class object of the metaobject controls it because the metaobject has not been created at the creation time of the object. In C++, the methods of a class object are defined as `static` member functions. The following methods defined in the class `MetaObj` are to control object creation. You can redefine these methods in your subclass to alter the implementation of object creation.

```
class MetaObj {
    ...
public:
    static MetaMsgReceiptor* Meta_Create(const char* class_name,
                LocalClassId id, Id cons_id, ArgPac& args);
protected:
    static MetaMsgReceiptor* Meta_HandleCreate(LocalClassId id,
                Id cons_id, ArgPac& args);
    static LocalClassId Meta_LookupClass(const char* class_name);
    ...
};
```

Two primitive types are introduced into the meta level. Type `MetaMsgReceiptor` is a unique supertype of any reflective object. Type `LocalClassId` represents the local identifier of a class, which is available only in the local address space. It must not be transferred to another address space. The global identifier of a class is a `const` string of its class name.

- `Meta_Create(const char* class_name, LocalClassId id,`
  `Id cons_id, ArgPac& args)` *Overridable*
  This method implements object creation. It is executed when an object is created by the `new` operator with a reflective constructor. The argument `class_name` specifies the class name of the created object. The argument `id` specifies the local identifier of the class of the created object. The argument `cons_id` specifies the identifier of the constructor, and the argument `args` specifies the argument list passed to the constructor. This method creates a reflective object and returns a pointer to it.

- `Meta_HandleCreate(LocalClassId id, Id cons_id, ArgPac& args)`
  This method actually creates a reflective object. It returns a pointer to the created object. The argument `id` specifies the local identifier of the class of the created object. The argument `cons_id` specifies the identifier

of the constructor used for the creation. The last argument `args` specifies the argument list passed to the constructor.

- `Meta_LookupClass(const char* class_name)`
  This method returns the local identifier of the class specified by the argument `class_name`.

3.3 class `NullMetaObj`

The class `NullMetaObj` is a subclass of `MetaObj`. It implements the normal semantics of C++ method calls and variable access. Four methods are overridden in the class `NullMetaObj`.

```
void NullMetaObj::Meta_MethodCall(Id m_id, argString category,
                                  ArgPac& args, ArgPac& reply) {
    Meta_HandleMethodCall(m_id, args, reply);
}

void NullMetaObj::Meta_Assign(Id var_id, Id, ArgPac& args) {
    Meta_HandleAssign(var_id, args);
}

void NullMetaObj::Meta_Read(Id var_id, Id, ArgPac& reply) {
    Meta_HandleRead(var_id, reply);
}

MetaMsgReceiptor* NullMetaObj::Meta_Create(const char*,
                LocalClassId class_id, Id cons_id, ArgPac& args) {
    return Meta_HandleCreate(class_id, cons_id, args);
}
```

Other subclasses of `MetaObj` redefine these methods differently to implement their own functions. For example, one may redefine them to print a message before executing a method, and another one may redefine to synchronize with other metaobjects before accessing variables.

3.4 Calling Methods of a Metaobject

An object can call a method of a metaobject directly beyond the boundary between base and meta levels. If a `reflect` method has the category name `metamethod`, then this method is overridden by its metaobject's method that has the same name. You can call the method of the metaobject, by calling that overridden `reflect` method. For example,

```
class X {
public:
//MOP reflect(metamethod):
    virtual void Meta_f(int p){}       // a dummy method
    ...
private:
    void g() { Meta_f(1); }
};

class M : public MetaObj {             // metaobject
public:
    void Meta_f(int p);
    ...
```

```
};
//MOP reflect class X : M;              // M is a metaobject of X
```

A method call `Meta_f(1)` in `X::g()` is interpreted as a method call to `M::Meta_f()` of a metaobject. Note that a `reflect (metamethod)` method should be `virtual`, and that the called method of a metaobject must be `public`.

The class `X` defines the overridden method `Meta_f()`, which is a dummy. Because this definition is not actually used, the definition of `Meta_f()` can be

```
virtual void Meta_f(int p){}          // empty body
```

or

```
virtual void Meta_f(int p) = nil;     // undefined
```

Open C++ adopts naming convention in which a method name of a metaobject begins with '`Meta_`'. When you define a new class of metaobjects, you should comply this naming convention.

3.5 Defining a Reflective Metaobject

Many reflective programming languages allow us to make an ascending tower of meta levels. An object is controlled by a metaobject, and a metaobject is also controlled by a meta-metaobject, and so on. Open C++ also allows us to make a metaobject reflective. You can define a reflective metaobject in the same manner as you define a reflective object.

We show an example using a reflective metaobject. Remember the example shown in the section 1. We modify the program "`person.c`".

```
// ---- person.c (modified) ----

#include <stdio.h>
#include "prnmetaobj.h"     // declaration of a metaobject

class Person {
    . . .
};

//MOP reflect class PrintMetaObj : PrintMetaObj;
//MOP reflect class Person : refl_PrintMetaObj;

Person::Person(char* n, int a) {
    . . .
```

Now, the metaobject of a `Person` object is the class `refl_PrintMetaObj`, which is reflective and has a meta-metaobject of the class `PrintMetaObj`. The metaobject of the class `PrintMetaObj` is defined recursively. We also modify the declaration of the class `PrintMetaObj` so that a method of `PrintMetaObj` is reflective.

```
// ---- prnmetaobj.h (modified) ----
```

```
#include "metaobj.h"

class PrintMetaObj : public MetaObj {
public:
//MOP reflect:
    void Meta_MethodCall(Id method_id, Id category,
                         ArgPac& args, ArgPac& reply);
};
```

Because `Meta_MethodCall()` of the class `PrintMetaObj` is a `reflect` method, a method call to `Meta_MethodCall()`, which is implicitly invoked by the runtime system, cause a meta-metaobject to print a message.

```
% OCC person.c
% OCC prnmetaobj.c
% CC person.o prnmetaobj.o .occ/reflect.a libOpc.a
% a.out
age 24
*** Meta_MethodCall() was called.       by a meta-metaobject
*** IncAge() was called.                by a metaobject
age 25
%
```

4 Open-Ended Compiler

The protocol of interaction between an object and its metaobject is embedded in the compiler. You can append a new module to the compiler, however, and switch the protocol during compilation. By selecting the most suitable protocol, the execution performance of the program may be optimized. If you add a new protocol to the compiler, you make a new subclass of a compiler object and then registers its instance. This is done at a building time of the compiler. A new protocol cannot be dynamically added after the compiler is build. This process is similar to that of UNIX kernel reconfiguration.

We don't discuss details of the means to add a new protocol in this document. Instead, we show how to switch protocols, and also an optional protocol `PtrArgOptimizer` available in default.

Protocols are switched by the following directive:

`//MOP Compiler =` *name_of_compiler_object*`;`

This means making an compiler object '*name_of_compiler_object*' to a current compiler. After reading this directive, the compiler uses a protocol specified by that compiler object. If '*name_of_compiler_object*' is `nil`, then the default compiler object is selected. Also, the programmer can send a message to the current compiler object to modify a protocol.

`//MOP Compiler->` *method_name* `(` *argument* `);`

This directive calls a method '*method_name*' of the current compiler object. The '*argument*' is optional. You can pass one or no identifier to a compiler object by this directive. Integers or characters cannot be passed.

The Open C++ compiler provides an optional compiler object as well as the default one. We address this compiler object `PtrArgOptimizer` below as an example of switching a protocol.

If `PtrArgOptimizer` is selected, the compiler interprets pointer and reference type arguments in a different manner. If a pointer-type argument appears, then the compiler interprets that the passed argument is the object pointed to by that pointer. The compiler, thus, pushes that object into an `ArgPac` object instead of the pointer itself. When the metaobject executes the `reflect` method, it retrieves this object from the `ArgPac` object, and then passes a pointer to this object to the `reflect` method. For example,

```
class X;

class Y {
public:
//MOP reflect:
    f(X* p);     //  a pointer argument
    ...
};

//MOP reflect class Y : NullMetaObj;

void g() {
    X objx;
    Y objy;
    X* ptr = &objx;
    objy.f(ptr);
}
```

Although the argument of the method call `objy.f()` in `g()` is a pointer `ptr`, an `ArgPac` object passed to the metaobject contains a copy of `objx` instead of `ptr`. If the metaobject actually executes the method `objy.f()`, `objy.f()` receives a pointer to the copy as its argument. This semantics can be represented as follows:

`objy.f(ptr)` $\Longrightarrow$ `objy.f((X objx2 = *ptr, &objx2));`

This protocol is useful when a metaobject implements remote procedure calls. In a distributed system, a local pointer has meanings only within local address space. Hence, to pass data to a remote object, the data itself must be passed instead of a pointer to the data. If passed data is large, however, this imposes some overheads in the current implementation of this compiler because the data is copied on the stack wastefully. The compiler object `PtrArgOptimizer` is provided to avoid such a wasteful copy. It transfers large data pointed to by a argument to a remote object in an efficient manner.

The compiler object `PtrArgOptimizer` treats `void*` type arguments in the same manner as the default compiler object does. When the current compiler object is `PtrArgOptimizer`, if you want to pass a pointer value itself to a `reflect` method, you must explicitly convert the type of the pointer to `void*`. Furthermore, `PtrArgOptimizer` cannot treat an argument that is a pointer to a pointer-type value or a reference to a reference-type value (e.g. `int**`). If such an argument appears, the compiler prints an error message.

Glossary

base-level A layer of a meta-level hierarchy. Objects belong to this layer.

default argument An argument that can be omitted when calling a method. If the argument is omitted, the default value is passed to the called method.

instance An entity created by a template. An object is an instance of a class. In some context, an instance simply means an object.

instance variable A variable contained by an object. In C++, it is called a member variable.

meta-level A layer of a meta-level hierarchy. Metaobjects belong to this layer.

metaobject An object that controls the behavior of an object. A metaobject alters the implementation of its object.

method A procedure related to an object. In C++, it is also called a member function.

reference A reference type is similar to a pointer type. A reference-type argument in C++ corresponds to a `VAR` argument in Pascal. That is, it is passed in the call-by-reference manner. A reference type is denoted by '`&`'. For example, the reference type to `int` is `int&`.

reflective Being controlled by a metaobject.

reflective class A class that can create a reflective object. You must add the keyword `refl_` at the head of the class name if you create a reflective object.

reflective object An object controlled by its metaobject.

reflect method A method controlled by a metaobject.

reflect variable An instance variable that a metaobject controls access to.

super class An inherited class. A base class in C++.

Command Reference

NAME

`OCC` — Open C++ compiler

SYNOPSIS

`OCC [-O] [-f] [-g] [-d` *archive_directory*`] [-a` *archive_name*`]`
`[-v] [-I`*include_directory*`][-D`*name*`[=`*def*`]] [--` *CC options*`]` *soruce_file*

DESCRIPTION

`OCC` is a compiler of Open C++. It compiles a given source file and generates a `*.o` file and an archive (`*.a`) file. The generated archive file contains code necessary to use reflective objects. At the linking stage, you have to link `*.o` files with this archive file as well as an Open C++ library. Use the `CC` command to link the files.

`OCC` produces a working file for each reflective class and compiles it by invoking a C++ compiler. The compiled code is stored in an archive file. Although the compiler does not doubly produce a working file that has been produced, it reproduces a working file if the declaration of a reflective class is updated. The compiler compares the timestamps of the working file and the source file, which may be an included file, to decide to reproduce or not.

OPTIONS

`-D` Define a symbol *name* to the preporcessor. This option is also passed to the C++ compiler.

`-I` Add a *directory* to the search path of the `#include` directive. The added directories are also passed to the C++ compiler.

`-O` Optimize the generated code.

`-a` Specify the file name of an archive file. The default name is `reflect.a`.

`-d` Specify the directory at which an archive file is produced. The compiler also produces working files at this directory. The default directory is `./.occ`.

`-f` Before compiling, delete the directory at which an archive file is produced.

`-g` Produce debug information. The temporary file `*-opc.cc` generated is not removed after the compilation. It is needed by a debugger like `dbx`. This option is passed to the C++ compiler.

`-v` Show the compiler version.

`--` Following options are interpreted as options of the C++ compiler. For example, if you type

```
OCC -a ./arch -- -fast foo.c
```

then the option `-fast` is passed to the C++ compiler.

`-I` and `-D` options are also taken from the environment variable `OCCOPT`. For example, you might tell `csh`:

```
setenv OCCOPT "-I/usr/local/occ/include -DTEST"
```

The environment variable is parsed before the command line options are.

EXAMPLE

Suppose that a source file is `foo.c`, and the Open C++ include files and library files are at `../include` and `../lib`. You compile the source file as follows:

```
% OCC -I../include foo.c
% ranlib .occ/reflect.a
% CC foo.o .occ/reflect.a ../lib/libOpc.a
```

FILES

`./.occ` The directory at which `OCC` produces an archive file.

`reflect.a` An archive file produced by `OCC`.

`libOpc.a` The library file of Open C++.

`libOc.a` An extended library of Open C++ that contains a class library for distributed computing. It needs to be linked `libtask.a` (AT&T C++ task library) together. This library includes `libOpc.a`.

NOTES

- To link `*.o` files produced by `OCC`, the libraries `libOpc.a` or `libOc.a` must be linked together. If you choose `libOc.a`, then `libtask.a` must be linked as well. Pass a link option `-ltask` to the linker.
- `friend` classes and functions of a reflective object have not been supported yet. The declaration is simply ignored.
- While `OCC` is compiling a source file, the macro name `__reflect` is defined.
- If the option `-DNOCHECK_OCLIB` is specified, then the compiler omits code that checks overflow of `ArgPac` objects. The maximum size of data stored into an `ArgPac` object is 8Kbytes.