OpenC++ 2.5 Reference Manual

Shigeru Chiba

Institute of Information Science and Electronics
University of Tsukuba
Email: `chiba@is.tsukuba.ac.jp`

1 Overview

OpenC++ is a toolkit for C++ translators and analyzers. It was designed to enable the users to develop those tools without concerning tedious parts of the development such as the parser and the type system. There are a number of tools that OpenC++ facilitates the development of. For example, the users can easily develop a C++ translator for implementing a language extension to C++ or for optimizing the compilation of their class libraries. Moreover, OpenC++ is useful to develop a source-code analyzer such as one for producing the class-inheritance graph of a C++ program.

The programmer who want to use OpenC++ writes a *meta-level* program, which specifies how to translate or analyze a C++ program. It is written in C++ and defines a small number of classes. Then the meta-level program is compiled by the OpenC++ compiler and (dynamically or statically) linked to the compiler itself as a compiler plug-in. The resulting compiler translates or analyzes a source program (it is called a *base-level* program for distinction) as the meta-level program specifies. See Figure 1.

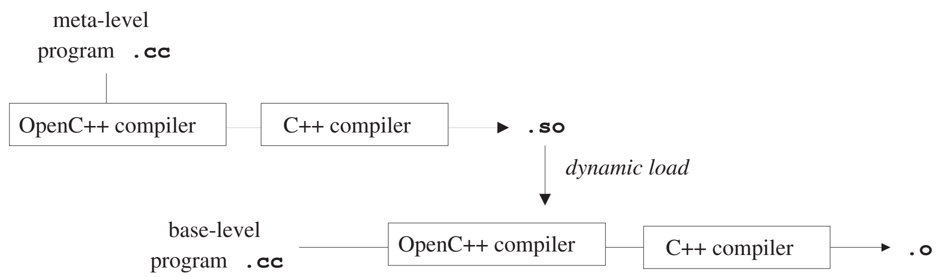

Figure 1: Overview (The meta-level program is dynamically linked)

The meta-level program is written according to the programming interface called the OpenC++ MOP (Metaobject Protocol.) Through this interface, the internal structure of the compiler is exposed to the programmers with object-oriented abstraction.

The base-level program is first preprocessed by the C++ preprocessor, and then divided into small pieces of code. These pieces of code are translated by class metaobjects and assembled again into a complete C++ program. In the OpenC++ MOP, the pieces of code is represented by `Ptree` metaobjects in the form of parse tree (that is, linked list). Although the metaobjects are identical to regular C++ objects, they exist in the compiler and represent a meta aspect of the *base-level* program. This is why they are not simply called *objects* but *metaobjects*.

The class metaobject is selected according to the static type of the translated piece of code. For example, if the piece of code is a member call on a `Point` object:

```
p0->move(3, 4)
```

Then it is translated by the class metaobject for `Point` (the type of `p`.) It is given to the class metaobject in the form of parse tree and translated, for example, into this;

```
(++counter, p0->move(3, 4))
```

This translation is similar to the one by Lisp macros, but it is type-oriented. The translation by the metaobjects is applied not only a member call but also other kinds of code involved with the C++ class system, such as data member access and class declaration.

The programmer who wants to customize the source-to-source translation writes a meta-level program to define a new class metaobject. This class metaobject is associated with a particular class in the base-level program and controls the translation of the code involved with the class. Thus the translation is applied only to the particular class and the rest of the code involved with the other classes remains *as is*.

The class metaobject can use other aspects of the base-level program during the source-code translation. In addition to the parse tree, it can access the semantic information such as static types and class definitions. These various aspects of the program facilitates the implementation of complex source-code translation and analysis. Furthermore, the OpenC++ MOP enables syntax extensions so that the base-level programmers can write annotations to help the translation or the analysis.

The meta architecture of OpenC++ might look very different from the architecture of other reflective languages. However, note that the class metaobject still controls the behavior of the base-level objects, which are instances of the class. The uniqueness of the OpenC++ MOP is only that the class metaobject does not interpret the base-level program in the customized way, but rather translates that program at compile time so that the customized behavior is implemented. The readers will find that, as in other reflective languages, the class metaobject has a member function for every basic action of the object, such as member calls, data reading/writing, object creation, and so forth, for customizing the object behavior.

2 Base-Level Language (OpenC++)

This section addresses the language specification of OpenC++. OpenC++ is identical to C++ except two extensions. To connect a base-level program and a meta-level program, OpenC++ introduces a new kind of declaration into C++. Also, new extended syntax is available in OpenC++ if the syntax is defined by the meta-level program.

2.1 Base-level Connection to the MOP

OpenC++ provides a new syntax for metaclass declaration. This declaration form is the only connection between the base level and the meta level. Although the default metaclass is `Class`, programmers can change it by using this declaration form:

- `metaclass` *metaclass-name* [*class-name* [`(` *meta-arguments* `)`]] `;` [1]

This declares the metaclass for a class. It must appear before the class is defined. If the class name is not specified, this declaration means nothing except that the metaclass is loaded into the compiler. *meta-arguments* is a sequence of identifiers, type names, literals, and C++ expressions surrounded by `()`. The elements must be separated by commas. The identifiers appearing in *meta-arguments* do not have to be declared in advance. What should be placed at *meta-arguments* is specified by the metaclass.

The code shown below is an example of metaclass declaration:

```
metaclass PersistentClass Point;
class Point {
public:
    int x, y;
};
```

The metaclass for `Point` is `PersistentClass`. This syntax was chosen so that it looks like a variable declaration such as:

```
class Point p0;
```

The former declaration defines a class metaobject `Point` as an instance of metaclass `PersistentClass`, and the latter defines an object `p0` as an instance of class `Point`.

[1] [ ] means an optional field.

2.2 Syntax Extensions

The extended syntax described here is effective if programmers define it by the MOP. By default, it causes a syntax error. To make it available, the programmers must register a new keyword, which is used in one of the following forms:

- Modifier *keyword* [(*function-arguments*)]

A keyword can be registered to lead a modifier. It may appear in front of class declarations, the `new` operator, or function arguments. For example, these statements are valid:

```
distribute class Dictionary { ... };
Point* p = remote(athos) new Point;
void append(ref int i, int j);
```

Here, `distribute`, `remote`, and `ref` are registered keywords.
Also, a modifier can be placed in front of a member declaration. For example,

```
class Point {
public:
    sync int x, y;
};
```

The keyword `sync` is a modifier.

- Access Specifier *keyword* [(*function-arguments*)] :

Programmers may define a keyword as a member-access specifier. It appears at the same place that the built-in access specifier such as `public` can appears. For example, if `after` is a user-defined keyword, then programmers may write:

```
class Window {
public:
    void Move();
after:
    void Move() { ... }     // after method
}
```

- While-style Statement

pointer `->` *keyword* (*expression*){ *statements* }

object . *keyword* (*expression*){ *statements* }
class-name : : *keyword* (*expression*){ *statements* }

A user-defined keyword may lead something like the `while` statement. In the grammar, that is not a statement but an expression. It can appear at any place where C++ expressions appear. *expression* is any C++ expression. It may be empty or separated by commas like function-call arguments. Here is an example of the while-style statement:

```
Matrix m2;
m2.forall(e){
    e = 0;
}
```

A user-defined keyword can also lead other styles of statements.

- For-style Statement

pointer -> *keyword* (*expr* ; *expr* ; *expr*){ *statements* }
object . *keyword* (*expr* ; *expr* ; *expr*){ *statements* }
class-name : : *keyword* (*expr* ; *expr* ; *expr*){ *statements* }

The for-style statement takes three expressions like the `for` statement. Except that, it is the same as the while-style statement.

- Closure Statement

pointer -> *keyword* (*arg-declaration-list*){ *statements* }
object . *keyword* (*arg-declaration-list*){ *statements* }
class-name : : *keyword* (*arg-declaration-list*){ *statements* }

The closure statement takes an argument declaration list instead of an expression. That is the only difference from the while-style statement. For example, programmers may write something like this:

```
ButtonWidget b;
b.press(int x, int y){
    printf("pressed at (%d, %d)\n", x, y);
}
```

This might be translated into this:

```
void callback(int x, int y){
    printf("pressed at (%d, %d)\n", x, y);
}
    :
ButtonWidget b;
b.press(callback);    // register a callback function
```

2.3 Loosened Grammar

Besides extended syntax, OpenC++'s grammar is somewhat loosened as compared with C++'s grammar. For example, the next code is semantically wrong in C++:

```
Point p = { 1, 3, 5 };
```

The C++ compiler will report that `p` cannot be initialized by `{ 1, 3, 5 }`. Such an aggregate can be used only to initialize an array. The OpenC++ compiler simply accepts such a semantically-wrong code. It ignores semantical correctness expecting that the code will be translated into valid C++ code.

3 Metaobject Protocol (MOP)

At the meta level, the (base-level) programs are represented by objects of a few predefined classes (and their subclasses that programmers define). These objects are called *metaobjects* because they are *meta* representation of the programs. Source-to-source translation from OpenC++ to C++ is implemented by manipulating those metaobjects.

The following several sections show details of such metaobjects. They reflect various aspects of programs that are not accessible in C++. Although most of metaobjects provide means of introspection, some metaobjects represent a behavioral aspect of the program and enables to control source-to-source translation of the program. Here is the list of metaobjects:

- `Ptree` metaobjects:

They represent a parse tree of the program. The parse tree is implemented as a nested-linked list.

- `Environment` metaobjects:

They represent bindings between names and types. Since this MOP is a compile-time MOP, the runtime values bound to names are not available at the meta level.

- `TypeInfo` metaobjects:

They represent types that appear in the program. The types include derived types such as pointer types and reference types as well as built-in types and class types.

- `Class` metaobjects:

As well as they represent class definitions, they control source-to-source translation of the program. Programmers may define subclasses of `Class` in order to tailor the translation.

- `Member` metaobjects:

They represent class members. They inform whether the member is a constructor, an inline function, a data member, a public member, or so forth.

Distinguishing `TypeInfo` metaobjects and `Class` metaobjects might look like wrong design. But this distinction is needed to handle derived types.

`TypeInfo` metaobjects were introduced to deal with derived types and fundamental types by using the same kind of metaobjects.

4 Representation of Program Text

Program text is accessible at the meta level in the form of parse tree. The parse tree is represented by a `Ptree` metaobject. It is implemented as a nested linked-list of lexical tokens — the S expressions in the Lisp terminology. For example, this piece of code:

```
int a = b + c * 2;
```

is parsed into:

```
[[static] [int] [[a = [b + [c * 2]]]] ;]
```

Here, `[]` denotes a linked list. Note that operators such as `=` and `+` make sublists. The sublists and their elements (that is, lexical tokens such as `a` and `=`) are also represented by `Ptree` metaobjects.

4.1 Basic Operations

To manipulate linked lists, the MOP provides many `static` member functions on `Ptree`, which are familiar to Lisp programmers:

- `static Ptree* First(Ptree* lst)`

This returns the first element of `lst`.

- `static Ptree* Rest(Ptree* lst)`

This returns the rest of `lst` except the first element, that is, the *cdr* field of `lst`.

- `static Ptree* Second(Ptree* lst)`

This returns the second element of *lst*.

- `static Ptree* Third(Ptree* lst)`

This returns the third element of *lst*.

- `static Ptree* Nth(Ptree* lst, int n)`

This returns the `n`-th element of `lst`. `Nth(lst, 0)` is equivalent to `First(lst)`.

- `static Ptree* Last(Ptree* lst)`

This returns the last cons cell, which is a list containing only the last element of `lst`.

• `static Ptree* ListTail(Ptree* lst, int k)`
This returns a sublist of `lst` obtained by omitting the first `k` elements. `ListTail(lst, 1)` is equivalent to `Rest(lst)` .

• `static int Length(Ptree* lst)`
This returns the number of the elements of `lst`.

• `static Ptree* Cons(Ptree* a, Ptree* b)`
This returns a cons cell whose *car* field is `a` and whose *cdr* is `b`.

• `static Ptree* List(Ptree* e1, Ptree* e2, ...)`
This returns a list whose elements are `e1`, `e2`, ... `List()` returns a null list `nil`.

• `static Ptree* Append(Ptree* lst1, Ptree* lst2)`
This concatenates `lst1` and `lst2`. It returns the resulting list.

• `static Ptree* CopyList(Ptree* lst)`
This returns a new list whose elements are the same as `lst` 's.

• `static Ptree* ReplaceAll(Ptree* lst, Ptree* orig, Ptree* subst)`
This returns a list in which all occurrences of `orig` in `lst` are replaced with `subst`. This is not a destructive operation.

• `static bool Eq(Ptree* lst, char x)`
• `static bool Eq(Ptree* lst, char* x)`
• `static bool Eq(Ptree* lst, Ptree* x)`
This returns `true` if `lst` and `x` are equal. If `x` is `Ptree*`, this determines the equivalence by comparing the pointers.

• `static bool Equal(Ptree* x, Ptree* y)`
This recursively compares `x` and `y` and returns `true` if they are equivalent.

Furthermore, the following member functions are available on `Ptree` metaobjects:

• `bool IsLeaf()`
This returns `true` if the metaobject indicates a lexical token.

• `void Display()`
This prints the metaobject on the console for debugging. Sublists are surrounded by `[` and `]`.

• `char* ToString()`
This converts the parse tree into a character string and returns it.

• `int Write(ostream& out)`
This writes the metaobject to the file specified by `out`. Unlike `Display()` , sublists are not surrounded by `[` and `]`. This member function returns the number of written lines.

• `ostream& operator <<(ostream& s, Ptree* p)`
The operator `<<` can be used to write a `Ptree` object to an output stream. It is equivalent to `Write()` in terms of the result.

The parse tree is basically a long list of the lexical tokens that appear in the program although some of them are grouped into sublists. The order of the elements of that list is the same as the order in which the lexical tokens appear. But if some fields such as the type field are omitted in the program, then `nil` is inserted at those places. For example, if the return type of a function declaration is omitted as follows:

```
main(int argc, char** argv){ }
```

then `nil` list is inserted at the head of the list:

```
[nil nil [main ( [[[int] [argc]] , [[char] [* * argv]]] )] [{
    nil
}]
```

Since the function body is also omitted, `nil` list is inserted between `{` and `}`.

4.2 Construction

Programmers can make `Ptree` metaobjects. Because the MOP provides a conservative garbage collector, they don't need to care about deallocation of the metaobjects. The next `static` member functions on `Ptree` are used to make a `Ptree` metaobjects.

• `static Ptree* Make(char* format, [Ptree* sublist, ...])`

This makes a `Ptree` metaobject according to the `format`. The `format` is a null-terminated string. All occurrences of `%c` (character), `%d` (integer), `%s` (character string), and `%p` (`Ptree`) in the `format` are replaced with the values following the `format`. `%%` in the `format` is replaced with `%`.

- `static Ptree* GenSym()`

This generates a unique symbol name (aka identifier) and returns it. The returned symbol name is used as the name of a temporary variable, for example.

The `Ptree` metaobject returned by `Make()` is not a real parse tree.[2]It is just a unparsed chunk of characters. Although programmers can use `Ptree` metaobjects generated by `Make()` as they use other `Ptree` metaobjects, the structure of those metaobjects does not reflect the code they represent.

Using `Make()`, programmers can easily generate any piece of code to substitute for part of the original source code. For example, suppose `array_name` is `xpos` and `offset` is `3`. The following function call:

```
Ptree::Make("%p[%d]", array_name, offset)
```

makes a `Ptree` metaobject that represents:

```
xpos[3]
```

`%p` simply expand a given `Ptree` metaobject as a character string. Thus programmers may write something like:

```
Ptree::Make("char* GetName(){ return \"%p\"; }",
            array_name);
```

Note that a double quote `"` must be escaped by a backslash `\` in a C++ string. `\"%p\"` makes a string literal. The function call above generates the code below:

```
char* GetName(){ return "xpos"; }
```

Although `Make()` follows the old `printf()` style, programmers can also use a more convenient style similar to Lisp's backquote notation. For example,

```
Ptree::Make("%p[%d]", array_name, offset)
```

[2]At least, for the time being.

The expression above can be rewritten using qMake() as follows:

```
Ptree::qMake("`array_name`[`offset`]")
```

Note that the "backqouted" C++ expressions array_name and offset are directly embedded in the C++ string. Their occurrence are replaced with the value of the expression. This replacement cannot be implemented in regular C++. It is implemented by the metaclass for Ptree.

- static Ptree* qMake(char* text)

This makes a Ptree metaobject that represents the text. Any C++ expression surrounded by backquotes ` can appear in text. Its occurrence is replaced with the value denoted by the expression. The type of the expression must be Ptree*, int, or char*.

Except the difference in the notation, qMake() is equivalent to Make(). Programmers can choose either one they prefer at any place.

4.3 Pattern Matching

The MOP provides a static member function on Ptree metaobjects for pattern matching.

- static BOOL Match(Ptree* list, char* pattern, [Ptree** sublist, ...])

This compares the pattern and list. If they match, this function returns true and binds the sublists to appropriate sublists of the list, as specified by the pattern. Note that the type of sublist is pointer to Ptree*.

For example, the function Match() is used as follows:

```
if(Ptree::Match(expr, "[%? + %?]", &lexpr, &rexpr))
    cout << "this is an addition.";
else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
    cout << "this is a subtraction.";
else
    cout << "unknown";
```

The pattern [%? + %?] matches a linked list that consists of three elements if the second one is +. If an expression expr matches the pattern, lexpr gets bound to the first element of expr and rexpr gets bound to the third element.

The `pattern` is a null-terminated string. Since `Match()` does not understand the C++ grammar, lexical tokens appearing in the pattern must be separated by a white space. For example, a pattern `a+b` is regarded as a single token. The pattern is constructed by these rules:

1. A word (characters terminated by a white space) is a pattern that matches a lexical token.
2. `%[`, `%]`, and `%%` are patterns that match `[`, `]`, and `%`.
3. `[]` is a pattern that matches a null list (`nil`).
4. `[`*pat1 pat2* ... `]` is a pattern that matches a list of *pat1*, *pat2*, ...
5. `%*` is a pattern that matches any token or list.
6. `%?` is a pattern that matches any token or list. The matched token or list is bound to `sublist`.
7. `%_` is a pattern that matches the rest of the list (the `cdr` part).
8. `%r` is a pattern that matches the rest of the list. The matched list is bound to `sublist`.

4.4 Reifying Program Text

If a `Ptree` metaobject represents a literal such as an integer constant and a string literal, we can obtain the value denoted by the literal.

• `static BOOL Reify(unsigned int& value)`
This returns `true` if the metaobject represents an integer constant. The denoted value is stored in `value`. Note that the denoted value is always a positive number because a negative number such as `-4` generates two distinct tokens such as `-` and `4`.

• `static BOOL Reify(char*& string)`
This returns `true` if the metaobject represents a string literal. A string literal is a sequence of character surrounded by double quotes `"`. The denoted null-terminated string is stored in `string`. It does not include the double quotes at the both ends. Also, the escape sequences are not expanded.

Note: the character string returned by `Reify()` is allocated in the heap area. However, because the MOP provides a conservative garbage collector, programmers do not need to deallocate the string by themselves.

4.5 Support Classes

The MOP provides two support classes `PtreeIter` and `PtreeArray` to help programmers to deal with `Ptree` objects. `PtreeIter` is useful to perform iteration on a list of `Ptree` objects. Suppose that `expr` is a list:

```
PtreeIter next(expr);
Ptree* p;
while((p = next()) != nil){
    // compute on p
}
```

Each element of `expr` is bound to `p` one at a time. The operator `()` on `PtreeIter` objects returns the next element. Programmers may call `Pop()` instead of the operator `()`. Since the two functions are equivalent, the program above can be rewritten to be:

```
PtreeIter next(expr);
Ptree* p;
while((p = next.Pop()) != nil){
    // compute on p
}
```

If the reader prefers the `for`-loop style, she may also say:

```
for(PtreeIter i = expr; !i.Empty(); i++){
    // compute on *i
}
```

Although this interface is slightly slower, it distinguishes the end of the list and a `nil` element. If `expr` includes `nil`, `Pop()` cannot correctly detect the end of the list.

Another support class is `PtreeArray` for dealing with an unbounded array of `Ptree` objects. It is used as follows (suppose that `expr` is a `Ptree` object):

```
PtreeArray a;            // allocate an array
a.Append(expr);          // append expr to the end of the array
Ptree* p = a[0];         // get the first element
Ptree* p2 = a.Ref(0);    // same as a[0]
int n = a.Number();      // get the number of elements
Ptree* lst = a.All();    // get a list of all the elements
a.Clear();               // make the array empty
```

5 Representation of Environments

`Environment` metaobjects represent bindings between names and types. If the name denotes a variable, it is bound to the type of that variable. Otherwise, if the name denotes a type, it is bound to the type itself. Programmers can look up names by the following member functions on `Environment` metaobjects:

- `bool Lookup(Ptree* name, bool& is_type_name, TypeInfo& t)`
This looks up the given `name` into the environment and returns `true` if found. The type of `name` is returned at `t`. If the name is a type name, `is_type_name` is set to `true`. If it is a variable name, `is_type_name` is set to `false`.

- `bool Lookup(Ptree* name, TypeInfo& t)`
This is an alias of `Lookup(Ptree*, bool&, TypeInfo&)` described above.

- `Class* LookupClassMetaobject(Ptree* class_name)`
This looks up the given `class_name` and returns the `Class` metaobject of the type. If the `class_name` is not found, this function returns `nil` (`class_name` may be a variable name.)

- `void Dump()`
This is for debugging and prints the elements in the environment on stderr.

6 Representation of Types

`TypeInfo` metaobjects represent types. Because C++ deals with derived types such as pointer types and array types, `Class` metaobjects are not used for primary representation of types. `TypeInfo` metaobjects do not treat `typedef`ed types as independent types. They are treated just as aliases of the original types.

The followings are member functions on `TypeInfo` metaobjects:

- `TypeInfoId WhatIs()`
This returns an `enum` constant that corresponds to the kind of the type: `BuiltInType`, `ClassType` (including `class`, `struct`, and `union`), `EnumType`, `TemplateType`, `PointerType`, `ReferenceType`, `PointerToMemberType`, `ArrayType`, `FunctionType`, `TemplateType`, or `UndefType` (the type is unknown).

- `Ptree* FullTypeName()`
This returns the full name of the type if the type is a built-in type, a class type, an `enum` type, or a template class type. Otherwise, this returns `nil`. For example, if the type is a nested class `Y` defined within a class `X` , this returns `X::Y`.

- `bool IsConst()`
This returns `true` if the type is const.

- `bool IsVolatile()`
This returns `true` if the type is volatile.

- `uint IsBuiltInType()`
This returns a bit field that represents what the built-in type is. If the type is not a built-in type, it simply returns `0` (`false`). To test the bit field, these masks are available: `CharType`, `IntType`, `ShortType`, `LongType`, `SignedType`, `UnsignedType`, `FloatType`, `DoubleType`, `LongDoubleType`, `BooleanType`, and `VoidType` . For example, `IsBuiltInType() & LongType` is `true` if the type is `long`, `unsigned long`, or `signed long`.

- `bool IsPointerType()`
This returns `true` if the type is a pointer type.

- `bool IsReferenceType()`

This returns `true` if the type is a reference type.

- `bool IsFunction()`
This returns `true` if the type is a function type.

- `bool IsArray()`
This returns `true` if the type is an array type.

- `bool IsPointerToMember()`
This returns `true` if the type is a pointer to member.

- `bool IsTemplateClass()`
This returns `true` if the type is a class template.

- `bool IsEnum()`
This returns `true` if the type is a enum type.

- `bool IsClass(Class*& metaobject)`
This returns `true` if the type is a class type. The `Class` metaobject representing the class type is stored in `metaobject`.

- `Class* ClassMetaobject()`
This returns a `Class` metaobject that represent the type. If the type is not a `class` type, it simply returns `nil`.

The `TypeInfo` metaobjects also provide methods for computing the dereferenced type. For example, those methods are used to get the type of the value that a pointer points to. Suppose that the type of the pointer is `int*`. If the dereferenced type of that pointer type is computed, then `int` is obtained.

- `void Dereference(TypeInfo& t)`
This returns the dereferenced type in `t` . If dereferencing is not possible, the `Undef` type is returned in `t` .

- `void Dereference()`
This is identical to `Dereference(TypeInfo&)` except that the `TypeInfo` metaobject itself is changed to represent the dereferenced type.

- `void Reference(TypeInfo& t)`
This returns the referenced type in `t` . For example, if the type is `int*`,

then the referenced type is `int**`.

• `void Reference()`
This is identical to `Reference(TypeInfo&)` except that the `TypeInfo` metaobject itself is changed to represent the referenced type.

The dereferenced type of a function type is the type of the return value. For example, if the function type is `void f(char)`, then the dereferenced type is `void` . If no return type is specified (e.g. constructors), the dereferenced type of the function type is "no return type."

• `bool IsNoReturnType()`
This returns `true` if the return type of the function is not specified.

The `TypeInfo` metaobjects also provide a method to obtain the types of function arguments.

• `int NumOfArguments()`
This returns the number of the arguments. If the type is not a function type, then it returns -1.

• `bool NthArgument(int nth, TypeInfo& t)`
If the type is `FunctionType` , this returns the type of the `nth` (≥ 0) argument in `t`. If the type is not `FunctionType` or the `nth` argument does not exist, this function returns `false`. If the `nth` argument is `...` (ellipses), then the returned type is an ellipsis type (see below.)

• `bool IsEllipsis()`
This returns `true` if the type is an ellipsis type.

Finally, we show a convenient method for constructing a `Ptree` metaobject that represents the declaration of a variable of the type.

• `Ptree* MakePtree(Ptree* varname = nil)`
This makes a `Ptree` metaobject that represents the declaration (or a function) of a variable of the type. For example, if the type is pointer to integer, this returns `[int *` *varname* `]`. `varname` may be `nil` .

If the type is a function type, `MakePtree()` returns a function prototype matching that type. The argument names in the prototype is omitted.

To construct a function prototype including argument names, programmers need to write as follows. Suppose that `ftype` is the type of the function, `atype` is the type of the argument, the function name is `Set`, and the argument name is `width`:

```
Ptree* arg = atype.MakePtree(Ptree::Make("width"));
Ptree* func = Ptree::qMake("Set(`arg`)");
ftype.Dereference();
Ptree* proto = ftype.MakePtree(func);    // function prototype
```

7 Class Metaobjects

Class metaobjects are the most significant metaobjects of the MOP. Although other metaobjects only represent a structural aspect of the program, the class metaobjects not only represent such a structural aspect but also allow programmers to define a subclass and alter the behavior of the program.

The default class for the class metaobjects is `Class`, which provides member functions for accessing the class definition. To alter a behavioral aspect of the class, the programmer define a subclass of `Class` that overrides `virtual` functions controlling source-to-source translation involved with the class.

7.1 Selecting a Metaclass

In general, the class of a metaobject is selected by the `metaclass` declaration at the base level. For example:

```
metaclass PersistentClass Point;
```

declares that the metaclass for `Point` is `PersistentClass`. This means that the compiler instantiates `PersistentClass` and makes the instantiated object be the class metaobject representing `Point`. Since `PersistentClass` is a regular C++ class but its instance is a class (metaobject), `PersistentClass` is called "metaclass". This might look weird, but regard a class metaobject as being identical to the class.

Programmers may specify a metaclass in a way other than the `metaclass` declaration. The exact algorithm to select a metaclass is as described below:

1. The metaclass specified by the `metaclass` declaration.
2. The metaclass specified by the keyword attached to the class declaration if exists.
3. Or else, the metaclass for the base classes. If they are different, an error is caused.
4. Otherwise, the default metaclass `Class` is selected.

Programmers may specify a metaclass by a user-defined keyword. For example,

```
distribute class Dictionary { ... };
```

This means that the metaclass associated with the user-defined keyword `distribute` is selected for `Dictionary`. If there is also a metaclass declaration for `Dictionary`, then an error occurs.

Although the default metaclass is `Class`, programmers can change it to another metaclass:

- `static void ChangeDefaultMetaclass(char* name)`

This changes the default metaclass to `name`.

7.2 Constructor

Class metaobjects may receive a meta argument when they are initialized. The meta argument is specified by programmers, for example, as follows:

```
metaclass PersistentClass Point("db", 5001);
```

The `Ptree` metaobject `["db" , 5001]` is a meta argument to the class metaobject for `Point`. Also, the programmers may specify a meta argument in this syntax:

```
distribute("db", 5001) class Dictionary { ... };
```

The user-defined keyword `distribute` can lead a meta argument. The class metaobject for `Dictionary` receives the same meta argument that the class metaobject for `Point` receives in the example above.

The member function `InitializeInstance()` on `Class` (and its subclasses) is responsible to deal with the meta argument. By default, the meta argument is simply ignored:

- `Class()`

This constructor performs nothing. The initialization is performed by `InitializeInstance()` invoked just after the constructor. For this reason, the member functions supplied by `Class` are not executable in the constructors of the subclasses of `Class`.
Note: only the OpenC++ compiler can call this constructor. The user programs should not call it.

- `void InitializeInstance(Ptree* definition, Ptree* meta_arg)`

This is automatically invoked just after the constructor is invoked. It initializes the data members of the class metaobject and processes the meta arguments. `definition` is a `Ptree` metaobject representing the class declaration. If a meta argument is not given, `meta_arg` is `nil`. This member function is not *overridable*; `InitializeInstance()` of the subclasses of `Class` must call the base-class'es `InitializeInstance()` at the *beginning*. Note: This has been separeted from the constructor. Otherwise, the constructor of `Class` would take two arguments and thus all the metaclasses have to have a constructor just for passing the arguments to the constructor of `Class`.

Note: only the OpenC++ compiler can call this member function. The user programs should not call it.

Another constructor is provided for the programmers to produce a new class. This is an example of the use of this constructor:

```
void MyClass::TranslateClass(Environment* e)
{
    Member m;
    Class* c = new Class(e, "Bike");
    LookupMember("move", m);
    c->AppendMember(m);
    AppendAfterToplevel(e, c);
}
```

A new class named `Bike` is created, a member named `move` is retrieved from the class represented by this class metaobject, and the retrieved member is copied to that new class. The created class `Bike` is then inserted in the source code after the declaration of the class represented by this class metaobject.

- `Class(Environment* e, char* name)`

This constructor creates a class with the given `name`. The created class has no member. If this constructor is invoked, `InitializeInstance()` is not called. No subclass of `Class` can inherit or invoke this constructor.

- `Class(Environment* e, Ptree* name)`

This constructor creates a class with the given `name`. The created class has no member. If this constructor is invoked, `InitializeInstance()` is not called. No subclass of `Class` can inherit or invoke this constructor.

- `void InsertBeforeToplevel(Environment* e, Class* c)`

This inserts the class specified by the metaobject `c` just before the toplevel declaration.

• `void AppendAfterToplevel(Environment* e, Class* c)`
This appends the class specified by the metaobject `c` just before the toplevel declaration.

7.3 Introspection

Since a class metaobject is the meta representation of a class, programmers can access details of the class definition through the class metaobject. The followings are member functions on class metaobjects. The subclasses of `Class` cannot override them.

• `Ptree* Name()`
This returns the name of the class.

• `Ptree* BaseClasses()`
This returns the base classes of the class. For example, if the class declaration is:

```
class C : public A, private B { ... };
```

Then, `BaseClasses()` returns a `Ptree` metaobject:

```
[: [public A] , [private B]]
```

• `Ptree* Members()`
This returns the body of the class declaration. It is a list of member declarations. It does not include `{` and `}`.

• `Ptree* Definition()`
This returns the `Ptree` metaobject representing the whole class declaration.

• `char* MetaclassName()`
This returns the name of the metaclass.

• `Class* NthBaseClass(int n)`
This returns the `n`-th (≥ 0) base class.

• `Ptree* NthBaseClassName(int n)`

This returns the name of the `n`-th (≥ 0) base class.

• `bool IsSubclassOf(Ptree* class_name)`
This returns `true` if the class is a subclass of `class_name`.

• `bool IsImmediateSubclassOf(Ptree* class_name)`
This returns `true` if the class is an immediate subclass of `class_name`.

• `bool NthMember(int n, Member& m)`
This returns `true` if the `n`-th (≥ 0) member, including data members and member functions, exists. The member metaobject representing the `n`-th member is returned in `m`. If the class is a subclass, the member is an inherited one from the base class.

• `bool LookupMember(Ptree* name, Member& m, int i = 0)`
This returns `true` if the member named `name` exists. The member metaobject representing that member is returned at `m`. The member may be an inherited one. If there are more than one members named `name`, the `i`-th (≥ 0) member is returned.

• `bool LookupMember(Ptree* name)`
This returns `true` if the member named `name` exists. The member may be an inherited one.

• `bool LookupMember(char* name, Member& m, int i = 0)`
This returns `true` if the member named `name` exists. The member metaobject representing that member is returned at `m`. The member may be an inherited one. If there are more than one members named `name`, the `i`-th (≥ 0) member is returned.

• `bool LookupMember(char* name)`
This returns `true` if the member named `name` exists. The member may be an inherited one.

7.4 Translation

Class metaobjects control source-to-source translation of the program. Expressions involving a class are translated from OpenC++ to C++ by a member function on the class metaobject.[3] Programmers may define a subclass of `Class` to override such a member function to tailor the translation.

The effective class metaobject that is actually responsible for the translation is the *static* type of the object involved by the expression. For example, suppose:

```
class Point { public: int x, y; };
class ColoredPoint : public Point { public: int color; };
        :
Point* p = new ColoredPoint;
```

Then, an expression for data member read, `p->x`, is translated by the class metaobject for `Point` because the variable `p` is a pointer to not `ColoredPoint` but `Point`. Although this might seem wrong design, we believe that it is a reasonable way since only static type analysis is available at compile time.

7.4.1 Class Definition

The class definition is translated by `TranslateClass()`. For example, if a member function `f()` is renamed `g()`, the member function `TranslateClass()` should be overridden to be this:

```
void MyClass::TranslateClass(Environment* e)
{
    Member m;
    LookupMember("f", m);
    m.SetName(Ptree::Make("g"));
    ChangeMember(m);
}
```

First, the member metaobject for `f()` is obtained and the new name `g()` is given to that member metaobject. Then, this change is reflected on the class by `ChangeMember()`. The class `Class` provides several member functions, such as `ChangeMember()`, for translating a class definition. Programmers can override `TranslateClass()` to call these functions and implement the translation they want.

• `void TranslateClass(Environment* env)`
This may call the member functions shown below and translate the declaration of the class.

[3] In the current version, the translated code is not recursively translated again. So the metaobjects have to translate code from OpenC++ to C++ rather than from OpenC++ to (less-extended) OpenC++. This limitation will be fixed in future.

— *Default implementation by* `Class`
This performs nothing.

• `void RemoveClass()`
This removes the whole declaration of the class from the source code.

• `void ChangeName(Ptree* new_name)`
This changes the name of the class. Note that this member function only substitutes `new_name` for the identifier following the `class` keyword in the declaration. The constructors or any other occurence of the class name are not changed.

• `void ChangeBaseClasses(Ptree* base_classes)`
This replaces the base-classes field of the class declaration with the given `base_class`.

• `void RemoveBaseClasses()`
This removes the base classes from the class declaration. The class does not inherit other classes after the translation.

• `void AppendBaseClass(Class* c, int specifier = Public, bool is_virtual = false)`
This appends a given class `c` to the list of the base classes. `specifier` is either `Class::Public`, `Class::Protected`, or `Class::Private`. If `is_virtual` is `true`, then the appended base class is `virtual`.

• `void AppendBaseClass(Ptree* class_name, int specifier = Public, bool is_virtual = false)`
This appends a given class named `class_name` to the list of the base classes. `specifier` is either `Class::Public`, `Class::Protected`, or `Class::Private`. If `is_virtual` is `true`, then the appended base class is `virtual`.

• `void AppendBaseClass(char* class_name, int specifier = Public, bool is_virtual = false)`
This appends a given class named `class_name` to the list of the base classes. `specifier` is either `Class::Public`, `Class::Protected`, or `Class::Private`. If `is_virtual` is `true`, then the appended base class is `virtual`.

• `void ChangeMember(Member& changed_member)`
This alter a member according to `changed_member`. The member metaob-

ject `changed_member` must be the object returned by `LookupMember()`. Note that the change of the member metaobject is not reflected until this member function is called.

- `void RemoveMember(Member& removed_member)`

This removes the member specified by `removed_member`. The member metaobject `removed_member` must be the object returned by `LookupMember()`.

- `void AppendMember(Member& added_member, int specifier = Public)`

This appends a new member to the class. `specifier` is either `Class::Public`, `Class::Protected`, or `Class::Private`. This member function is used to append a member similar to an existing one. For example, `added_member` may be the object returned by`LookupMember()`and called `SetName()` on to change the member name.

- `void AppendMember(Ptree* text)`

This inserts `text` after the member declarations in the original class declaration. `text` can be not only a member declaration but also a nested class declaration, an access specifier, and any kind of program text.

The implementation of member functions is translated by `TranslateMemberFunction()`. For example,

```
void Point::Move(int rx, int ry)
{
    x += rx;
    y += ry;
}
```

To translate this function implementation, the compiler calls `TranslateMemberFunction()` on the class metaobject for `Point`. The arguments are an environment and a member metaobject for `Move()`. If this member metaobject is changed by member functions such as `SetName()`, the change is reflected on the program. Unlike class declarations, no explicit function call for the reflection is not needed. For example,

```
void MyClass::TranslateMemberFunction(Environment* env, Member& m)
{
    m.SetFunctionBody(Ptree::Make("{}"));
}
```

This customizes the member function so that it has an empty body.

• `void TranslateMemberFunction(Environment* env, Member& m)`
This translates the implementation of a member function specified by `m`. The compiler does not call this function if the member function is inlined in the class declaration. For example,

```
class Point {
public:
    void Reset() { x = y = 0; }
    void Move(int, int);
    int x, y;
};
inline void Point::Move(int rx, int ry) { ... }
```

`TranslateMemberFunction()` is called only for `Move()`. The implementation of `Reset()` can be translated by `TranslateClass()`. Note that, even if the implementation of `Move()` is translated by `TranslateMemberFunction()`, the member declaration of `Move()` in the class declaration is not translated. It needs to be explicitly translated in `TranslateClass()`.
— *Default implementation by* `Class`
This performs nothing.

• `void InsertBeforeToplevel(Environment* e, Member& m)`
This inserts the member function specified by `m` just before the toplevel declaration. It should be used to insert the implementation of a member function derived from the argument `m` of `TranslateMemberFunction()`. Note that `AppendMember()` only appends a member declaration in the class declaration.

• `void AppendAfterToplevel(Environment* e, Member& m)`
This appends the member function specified by `m` just before the toplevel declaration. It should be used to append the implementation of a member function derived from the argument `m` of `TranslateMemberFunction()`. Note that `AppendMember()` only appends a member declaration in the class declaration.

7.4.2 Expressions

Class metaobjects also control the translation of expressions. An expressions, such as member calls, are translated by one of the following `virtual` functions on the class metaobject involved with the expression. For example, if the expression is a member call on a `Point` object, it is translated by the class metaobject for the `Point` class.

• `Ptree* TranslateInitializer(Environment* env, Ptree* var_name, Ptree* expr)`

This translates a variable initializer `expr`, which would be `[=` *expression*`]` or `[( [`*expression*`] )]`. The two forms correspond to C++'s two different notations for initialization. For example:

```
complex p(2.3, 4.0);
complex q = 0.0;
```

The initializers are `[( [2.3 , 4.0] )]` and `[= 0.0]`, respectively.

The argument `var_name` indicates the name of the variable initialized by `expr`.

— *Default implementation by* `Class`

This translates `expr` by calling `TranslateExpression()` on the second element of `expr`.

• `Ptree* TranslateAssign(Environment* env, Ptree* object, Ptree* assign_op, Ptree* expr)`

This translates an assignment expression such as `=` and `+=`. `object` is an instance of the class, which the value of `expr` is assigned to. `assign_op` is an assignment operator. `object` and `expr` have not been translated yet.

— *Default implementation by* `Class`

This calls `TranslateExpression()` on `object` and `expr` and returns the translated expression.

• `Ptree* TranslateBinary(Environment* env, Ptree* lexpr, Ptree* binary_op, Ptree* rexpr)`

This translates a binary expression. `binary_op` is the operator such as `*`, `+`, `<<`, `==`, `|`, `&&`, and `,` (comma). `lexpr` and `rexpr` are the left-side expression and the right-side expression. They have not been translated yet. The class metaobject that this function is called on is for the type of `rexpr`.

— *Default implementation by* `Class`

This calls `TranslateExpression()` on `lexpr` and `rexpr` and returns the translated expression.

• `Ptree* TranslateUnary(Environment* env, Ptree* unary_op, Ptree* object)`

This translates a unary expression. `unary_op` is the operator, which are either `*`, `&`, `+`, `-`, `!`, `~`, `++`, or `--`. `sizeof` is not included. `object` is an instance of the class, which the operator is applied to. `object` has not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and returns the translated expression.

• `Ptree* TranslateSubscript(Environment* env, Ptree* object, Ptree* index)`
This translates a subscript expression (array access). `object` is an instance of the class, which the operator `[]` denoted by `index` is applied to. `index` is a list `[\[` *expression* `\]]` . `object` and `expr` have not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and `index` and returns the translated expression.

• `Ptree* TranslatePostfix(Environment* env, Ptree* object, Ptree* post_op)`
This translates a postfix increment or decrement expression (`++` or `--`). `object` is an instance of the class, which the operator `post_op` is applied to. `object` has not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and returns the translated expression.

• `Ptree* TranslateFunctionCall(Environment* env, Ptree* object, Ptree* args)`
This translates a function call expression on `object`. Note that it is not for translating a member function call. It is invoked to translate an application of the call operator `()`. `object` is an instance of the class. `object` and `args` have not been translated yet. For example:

```
class Iterator { ... };
    :
Iterator next;
while(p = next())
    record(p);
```

`TranslateFunctionCall()` is called on the class metaobject for `Iterator` to translate `next()`. In this case, `object` indicates `next`.

— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and `TranslateArguments()` on `args` , and returns the translated expression.

• `Ptree* TranslateNew(Environment* env, Ptree* header, Ptree* new_op, Ptree* placement, Ptree* type_name, Ptree* arglist)`
This translates a `new` expression. `header` is a user-defined keyword (type modifier), `::` (if the expression is `::new`), or `nil`. `new_op` is the new operator. `type_name` may include an array size surrounded by `[]`. `arglist` is arguments to the constructor. It includes parentheses `()` . `placement`, `type_name`, and `arglist` have not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateArguments()` on `placement` and `arglist`, and `TranslateNewType()` on `type_name`. Then it returns the translated expression.

• `Ptree* TranslateDelete(Environment* env, Ptree* delete_op, Ptree* object)`
This translates a delete expression on the `object`. `delete_op` is the delete operator. Note that this function is not called on `::delete` or `delete []` expressions.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on the `object` and returns the translated expression.

• `Ptree* TranslateMemberRead(Environment* env, Ptree* object, Ptree* op, Ptree* member)`
This translates a member read expression on the `object`. The operator `op` is `.` (dot) or `->`. `member` specifies the member name. `object` has not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on the `object` and returns the translated expression.

• `Ptree* TranslateMemberRead(Environment* env, Ptree* member)`
This translates a member read expression on the `this` object. That is, it is invoked if the object is not explicitly specified.
— *Default implementation by* `Class`
This returns `member`.

• `Ptree* TranslateMemberWrite(Environment* env, Ptree* object, Ptree* op, Ptree* member, Ptree* assign_op, Ptree* expr)`
This translates a member write expression on the `object` . The operator `op` is . (dot) or `->`. `member` specifies the member name. `assign_op` is an assign operator such as `=` and `+=`. `expr` specifies the right-hand expression of the assign operator. `object` and `expr` have not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` and `expr` and returns the translated expression.

• `Ptree* TranslateMemberWrite(Environment* env, Ptree* member, Ptree* assign_op, Ptree* expr)`
This translates a member write expression on the `this` object. That is, it is invoked if the object is not explicitly specified. `member` specifies the member name. `assign_op` is an assign operator such as `=` and `+=` . `expr` specifies the right-hand expression of the assign operator. `expr` has not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on `expr` and returns the translated expression.

• `Ptree* TranslateMemberCall(Environment* env, Ptree* object, Ptree* op, Ptree* member, Ptree* arglist)`
This translates a member function call on the `object`. The operator `op` is . (dot) or `->` . `member` specifies the member name. `arglist` is arguments to the function. It includes parentheses `()` . `object` and `arglist` have not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateExpression()` on `object` , and `TranslateArguments()` on `arglist` . Then it returns the translated expression.

• `Ptree* TranslateMemberCall(Environment* env, Ptree* member, Ptree* arglist)`
This translates a member function call on the `this` object. That is, it is invoked if the object is not explicitly specified. `member` specifies the member name. `arglist` is arguments to the function. It includes parentheses `()` . `arglist` has not been translated yet.
— *Default implementation by* `Class`
This calls `TranslateArguments()` on `arglist` and returns the translated expression.

• `Ptree* TranslatePointer(Environment* env, Ptree* variable_name)`
This translates occurrences of the pointer variables, such as `this`, indicating instances of this class. It translates those variables even if they are the left values of assignment expressions. Note that this function is also called on the target expression of the `->` operator from within (the default) `TranslateMemberCall()` etc.
— *Default implementation by* `Class`
This returns `variable_name` as is.

• `Ptree* TranslateUserStatement(Environment* env, Ptree* object, Ptree* op, Ptree* keyword, Ptree* rest)`
This translates a user-defined statement, which is a while-style, for-style, or closure statement. The first three elements of the statement are specified by `object`, `op`, and `keyword`. The rest of the statement, the `()` part and the `{}` part, is specified by `rest`. For example:

```
matrix.forall(e) { e = 0.0; }
```

If `forall` is a user-defined keyword for the while-style statement, `object` is `matrix`, `op` is `.` (dot), `keyword` is `forall`, and `rest` is
`[( e ) [{ [[e = 0.0] ;] }]`.

To recursively translate the `{}` part, `TranslateExpression()` should be called on the last element of `rest`, that is, `rest->Last()`.
— *Default implementation by* `Class`
This causes an error and returns `nil`.

• `Ptree* TranslateStaticUserStatement(Environment* env, Ptree* keyword, Ptree* rest)`
This translates a user-defined statement beginning with a class name such as:

```
Lambda::closure(int i) { return i + 1; }
```

Here, `closure` is a user-defined keyword. The meaning of the arguments to this function is the same as that of `TranslateUserStatement()`.

To recursively translate the `{}` part, `TranslateExpression()` should be called on the last element of `rest`, that is, `rest->Last()`.

This member function is named after that the syntax is similar to one for `static` member function calls.

— *Default implementation by* `Class`
This causes an error and returns `nil`.

The MOP does not allow programmers to customize array access or pointer operations. Suppose that `p` is a pointer to a class `A`. Then the class metaobject for `A` cannot translate expressions such as `*p` or `p[3]`. This design decision is based on C++'s one. For example, C++'s operator overloading on `[]` does not change the meaning of array access. It changes the meaning of the operator `[]` applied to not an array of objects but an object.

If the MOP allows programmers to customize array access and pointer operations, they could implement an inconsistent extension. For example, they want to translate an expression `p[2]` into `p->get(2)`, where `p` is a pointer to a class `X`. Then, what should this expression `*(p + 2)` be translated into? Should the MOP regard it as an array access or a pointer dereference? Because C++ provides strong pointer arithmetic, designing an interface to consistently customize array access and pointer operations is difficult.

The class `Class` also provides functions for translating expressions and actual arguments. These functions are not overridden but rather called by other functions shown above:

• `Ptree* TranslateExpression(Environment* env, Ptree* expr)`
This translates an expression.

• `Ptree* TranslateExpression(Environment* env, Ptree* expr, TypeInfo& t)`
This translates an expression and stores its type in `t`.

• `Ptree* TranslateArguments(Environment* env, Ptree* args)`
This translates an actual-argument list.

• `Ptree* TranslateNewType(Environment* env, Ptree* type_name)`
This translates the type name included in a `new` expression. If the created object is an array, it calls `TranslateExpression()` on the expression specifying the array size. This is called by `TranslateNew()`.

7.5 Registering Keywords

To make user-defined keywords available at the base level, programmers must register the keywords by the `static` member functions on `Class` shown below. Those member functions should be called within `Initialize()`.

- `static void RegisterNewModifier(char* keyword)`

This registers `keyword` as a new modifier. If this appears in front of the new operator, the translation is performed by `TranslateNew()`. If it appears in front of a function argument, then it is automatically eliminated after the translation. Its existence can be inspected by `GetUserArgumentModifiers()` on the member metaobject.

- `static void RegisterNewMemberModifier(char* keyword)`

This registers `keyword` as a new member modifier. It is automatically eliminated after the translation. Its existence can be inspected by `GetUserMemberModifier()` on the member metaobject.

- `static void RegisterNewAccessSpecifier(char* keyword)`

This registers `keyword` as a new access specifier. It is automatically eliminated after the translation. Its existence can be inspected by `GetUserAccessSpecifier()` on the member metaobject.

- `static void RegisterNewWhileStatement(char* keyword)`

This registers `keyword` as a new `while`-style statement.

- `static void RegisterNewForStatement(char* keyword)`

This registers `keyword` as a new `for`-style statement.

- `static void RegisterNewClosureStatement(char* keyword)`

This registers `keyword` as a new closure-style statement.

- `static void RegisterMetaclass(char* keyword, char* metaclass)`

This registers `keyword` as a new modifier and associates it with `metaclass`. If this keyword appears in front of a class declaration, then `metaclass` is selected for the declared class.

The translation of the registered keyword for the while-, the for-, or the closure-style statement is the responsibility of the class metaobject. It is processed by `TranslateUserStatement()` and `TranslateStaticUserStatement()`.

7.6 Initialization and Finalization

The MOP provides functions to initialize and finalize class metaobjects:

• `static bool Initialize()`
This is a class initializer; it is invoked only once on each metaclass (not on each class metaobject) right after the compiler starts (if it is statically linked) or the metaclass is dynamically loaded. It returns `true` if the initialization succeeds. The subclasses of `Class` may define their own `Initialize()` but they must not call their base classes' `Initialize()`.
— *Default implementation by* `Class`
This does nothing except returning `true`.

• `Ptree* FinalizeInstance()`
This is invoked on each class metaobject after all the translation is finished. The returned `Ptree` object is inserted at the end of the translated source file. This member function is not *overridable*; `FinalizeInstance()` of the subclasses of `Class` must call the base-class'es `FinalizeInstance()` .
— *Default implementation by* `Class`
This does nothing except returning `nil`.

• `static Ptree* FinalizeClass()`
This is invoked on each metaclass after all the translation is finished if it exists. The returned `Ptree` object is inserted at the end of the translated source file. The subclasses of `Class` may define their own `FinalizeClass()` but they must not call their base classes' `FinalizeClass()`.
— *Default implementation by* `Class`
This returns `nil`.

• `static ClassArray& AllClasses()`
This is available only within `FinalizeInstance()`. It returns an array of all the classes appearing in the base-level program. The returned array `a` is used as follows:

```
int n = a.Number();       // get the number of elements
Class* c = a[0];          // get the first element
Class* c2 = a.Ref(0);     // same as a[0]
```

• `int Subclasses(ClassArray& result)`

This is available only within `FinalizeInstance()`. It returns the number of all the subclasses of the class. Those subclasses are also stored in `result`.

• `int ImmediateSubclasses(ClassArray& result)`
This is available only within `FinalizeInstance()`. It returns the number of all the immediate subclasses of the class. Those subclasses are also stored in `result`. The immediate subclass means only a child class but not a grand child.

• `static int InstancesOf(char* metaclas_name, ClassArray& result)`
This is available only within `FinalizeInstance()`. It returns the number of all the classes that are instances of the metaclass specified by `metaclass_name`. Also those classes are stored in `result`.

7.7 Inserting Statements

Class metaobjects can not only replace expressions but also insert statements into the translated source code:

• `void InsertBeforeStatement(Environment* e, Ptree* s)`
This inserts the statement `s` just before the statement currently translated.

• `void AppendAfterStatement(Environment* e, Ptree* s)`
This appends the statement `s` just after the statement currently translated.

• `void InsertBeforeToplevel(Environment* e, Ptree* s)`
This inserts the statement `s` just before the toplevel declaration, such as function definitions, that are currently translated.

• `void AppendAfterToplevel(Environment* e, Ptree* s)`
This appends the statement `s` just after the toplevel declaration, such as function definitions, that are currently translated.

• `bool InsertDeclaration(Environment* e, Ptree* d)`
This inserts the declaration statement `d` at the beginning of the function body. For example,

```
1: void Point::Move(int new_x, int new_y)
```

```
2: {
3:      x = new_x; y = new_y;
4: }
```

The declaration statement `d` is inserted between the 2nd line and the 3rd line.

This function returns `true` if the insertion succeeds.

• `bool InsertDeclaration(Environment* e, Ptree* d, Ptree* key, void* client_data)`
This inserts the declaration statement `d` at the beginning of the function body, and also records `client_data` with `key`. The recorded client data last while the function body is translated. This function returns `true` if no client data is recorded with `key` and the insertion succeeds.

• `void* LookupClientData(Environment* e, Ptree* key)`
This returns the client data associated with `key`. If the client data is not recorded, this function returns `nil`.

7.8 Command Line Options

Class metaobjects can receive command line options. For example, if the user specify the `-M` option:

```
% occ -Mclient -Mmode=tcp sample.cc
```

Then the class metaobjects can receive the command line options `client` and `mode` (the value is `tcp`) by the following functions:

• `static bool LookupCmdLineOption(char* option_name)`
This returns `true` if the option specified by `option_name` is given from the command line.

• `static bool LookupCmdLineOption(char* key, char*& value)`
This returns `true` if the option specified by `option_name` is given from the command line. The value of the option is stored in `value`. If the option value is not given, `value` is `nil`.

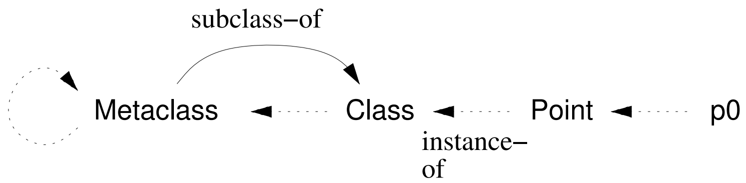

Figure 2: Instance-of Relationship

7.9 Error Message

The following functions reports an error that occurs during the source-to-source translation.

- `void ErrorMessage(Environment* env, char* message,`
  `Ptree* code, Ptree* where)`

This prints an error message. For example, `message` is `"wrong type:"` and `code` is a `Ptree` metaobject representing `Point`, then the printed messages is something like this:

```
sample.cc:25: wrong type: Point
```

The file name and the line number point to the location of the code specified by `where`. If `where` is `nil`, no file name or line number are not printed.

The first argument `env` can be omitted. In this case, the printed line number may be wrong.

- `void WarningMessage(Environment* env, char* message,`
  `Ptree* name, Ptree* where)`

This prints a warning message. The meaning of the arguments are the same as `ErrorMessage()`. The first argument `env` can be omitted.

7.10 Metaclass for Class

Since OpenC++ is a self-reflective language, the meta-level programs are also in OpenC++. They must be compiled by the OpenC++ compiler. Because of this self-reflection, metaclasses also have their metaclasses. The metaclass for `Class` and its subclasses must be `Metaclass`. However, programmers do not have to explicitly declare the metaclass for their metaclasses because the subclasses of `Class` inherit the metaclass from `Class`.

`Metaclass` makes it easy to define a subclass of `Class`. It automatically inserts the definition of `MetaclassName()` of that subclass and also generates house-keeping code internally used by the compiler.

Since `Metaclass` is a subclass of `Class`, its metaclass is `Metaclass` itself. This relationship is illustrated in Figure 2.

8 Member Metaobjects

Member metaobjects provide the ability of introspection and source-code translation of the members. They can be obtained by calling `NthMember()` or `LookupMember()` on a class metaobject. The following is the list of the member functions provided by the member metaobjects.

8.1 Introspection

First, we show the member functions for introspection.

- `Member(Member&)`

This is a constructor to make a copy of a member metaobject.

- `Ptree* Name()`

This returns the member name.

- `Ptree* ArgumentList()`

This returns the formal argument list of the member. For example, if the member is `int f(int, char*)`, then this function returns `[[[int] [nil]] , [[char] [*]]]`. If the member is a data member, this function returns `nil`.

- `Ptree* Arguments()`

This returns the argument list of the member. Unlike `ArgumentList()`, the returned list does not include the types. It is a list of the argument names. If the member is `int f(int p, char* q)`, then this function returns `[p , q]`. Even if the argument name is not given in the argument list, it is automatically filled by this function. In this case, the formal argument list of the member is also changed to include that argument name. If the member is a data member, this function returns `nil`.

- `Ptree* MemberInitializers()`

This returns the member initializers if the member is a constructor. Otherwise, it returns `nil`. For example, if the member is:

```
X::X() : p(3), q(1) {
... }
```

Then this function returns `[: [p ( [3] )] , [q ( [1] )]]`.

• `Ptree* FunctionBody()`
This returns the function body if the member is a function. The returned text includes braces `{}`.

• `int Nth()`
If the member is the `i` -th member, this returns `i`. Otherwise, if the member is not declared, it returns -1.

• `void Signature(TypeInfo& t)`
This returns the type of the member in `t`. If the member is a member function, the returned type is the function type.

• `Class* Supplier()`
This returns the class supplying this member. If the member is inherited from the base class, then the returned class is that base class.

• `bool IsConstructor()`
This returns `true` if the member is a constructor.

• `bool IsDestructor()`
This returns `true` if the member is a destructor.

• `bool IsFunction()`
This returns `true` if the member is a member function.

• `bool IsPublic()`
This returns `true` if the member is a `public` member.

• `bool IsProtected()`
This returns `true` if the member is a `protected` member.

• `bool IsPrivate()`
This returns `true` if the member is a `private` member.

• `bool IsStatic()`
This returns `true` if the member is a `static` member.

• `bool IsMutable()`
This returns `true` if the member is a `mutable` member.

• `bool IsInline()`
This returns `true` if the member is a `inline` member function.

• `bool IsVirtual()`
This returns `true` if the member is a `virtual` member function.

OpenC++ allows syntax extensions for access specifiers and argument lists. The following members are used for dealing with such syntax extensions.

• `Ptree* GetUserAccessSpecifier()`
This returns an user-defined access specifier for the member. For example, suppose that `sync` is a user-defined keyword:

```
class Window {
public:
    void Move();
sync:
    void Resize();
};
```

Then `GetUserAccessSpecifier()` called on `Resize()` returns `[sync :]`. The user-defined access specifier is effective until another access specifier appears. For example:

```
class X {
public:
    void f1();      // public
sync:
    void f2();      // public, sync
private:
    void g1();      // private
sync:
    void g2();      // private, sync
};
```

The user-defined access specifiers are automatically eliminated. The programmer does not have to be concerned about it.

• `Ptree* GetUserMemberModifier()`
This returns the member modifier for the member. If no member modifier is specified, this returns `nil`. The member modifier is automatically eliminated. The programmer does not have to be concerned about it.

• bool `GetUserArgumentModifiers(PtreeArray& modifiers)`
This computes user-defined type modifiers attached to the argument types. If successful, it returns `true` and stores the result in `modifiers`. The result is a `PtreeArray` of user-defined type modifiers. The `i`-the element is one for the `i`-th argument. If no modifier is specified, the element is `nil`. For example, if `ref` is a user-defined type modifier,

```
class C {
public:
    void f(ref int p1, int p2);
};
```

Then `GetUserArgumentModifiers()` called on `f` returns an array `{ [ref], nil }`.
All the user-defined type modifiers are automatically eliminated. The programmer does not have to be concerned about it.

8.2 Translation

The member metaobjects also provide functions for customizing the member. The changes are not actually reflected on the source-code translation until `ChangeMember()` or `AppendMember()` is called on the class metaobject.

• void `SetName(Ptree* name)`
This changes the member name to `name`.

• void `SetQualifiedName(Ptree* name)`
This changes the member name to `name`. Unlike `SetName()`, this function substitutes `name` for the member name including the qualified class name. It is useful in `Class::TranslateMemberFunction()`. For example, if the member is:

```
void Rect::Enlarge(int rx, int ry) { ... }
```

Then, `SetQualifiedName(Ptree::Make("Point::Move"))` changes this member to:

```
void Point::Move(int rx, int ry) { ... }
```

• void `SetArgumentList(Ptree* arglist)`

This changes the formal argument list of the member function to `arglist`.

- `void SetMemberInitializers(Ptree* init)`

This changes the member initializers of the constructor to `init`.

- `void SetFunctionBody(Ptree* body)`

This changes the function body of the member to `body`.

The member functions for introspection such as `Name()` does not reflect the customization in the results. For example, `Name()` returns the original member name even if `SetName()` specifies a new name. To get the new value specified by the above functions such as `SetName()`, the following functions are used:

- `void NewName()`

This returns the new member name substituted for the original one.

- `void NewArgumentList()`

This returns the new argument list substituted for the original one.

- `void NewMemberInitializers()`

This returns the new member initializers substituted for the original one.

- `void NewFunctionBody()`

This returns the new function body substituted for the original one.

Command Reference

NAME

`occ` — the Open C++ compiler

SYNOPSIS

occ [-l] [-s] [-V] [-v] [-c] [-E] [-n] [-p]
    [-m*file_name*] [--regular-c++] [-I*include_directory*]
    [-D*name*[=*def*]] [-M*option*[=*value*]]
    [-- *C++ compiler options* [.o *and* .a *files*]] *source_file*

DESCRIPTION

`occ` compiles an OpenC++ program into an object file. It first invokes the C++ preprocessor with the predefined macro `__opencxx` and generates a `.occ` file, then translates it into a `.ii` file according to the meta-level program. The `.ii` file is compiled by the back-end C++ compiler, and finally an `a.out` file is produced. If `occ` is run with the `-c` option, it generates a `.o` file but suppresses linking.

For example, to compile a base-level program `sample.cc` with the meta-level program `MyClass.mc`, the user should do as follows:

```
% occ -m MyClass.mc
```

First, `MyClass.mc` should be compiled into shared libraries `MyClass.so` and `MyClass-init.so`. The produced shared libraries must be under the directory specified by `LD_LIBRARY_PATH`. Then, the user can compile the base-level program:

```
% occ -- -o sample sample.cc
```

If `sample.cc` requires a metaclass `MyClass`, `occ` dynamically loads and links `MyClass.so` and `MyClass-init.so`. Then `sample.cc` is compiled according to the metaclass `MyClass` and an executable file `sample` is produced.

The separate compilation of meta-level programs is also supported. Suppose that `MyClass` is implemented by `foo.mc` and `bar.mc`. The user should compile them as follows:

```
% occ -c -m foo.mc
```

```
% occ -c -m bar.mc
```

This produces `foo.o`, `bar.o`, and `MyClass-init.so`. Although the second invocation of `occ` overrides `MyClass-init.so` produced by the first invocation, this is not a problem. To get the shared library, `foo.o` and `bar.o` have to be linked by hand into `MyClass.so` by:

```
% occ -mMyClass foo.o bar.o
```

For the reason of efficiency, the user can statically link the meta-level program with the OpenC++ compiler. To do this, the user must not specify the `-m` option:

```
% occ -- -o myocc opencxx.a MyClass.mc
```

First, `MyClass.mc` should be compiled and linked to the OpenC++ compiler. The command shown above produces the OpenC++ compiler that `MyClass.mc` is embedded in. `opencxx.a` is the archive of the original OpenC++ compiler. (Note: The Solaris and Linux users have to add the `-ldl` option after `opencxx.a`.)

Then, the produced compiler `myocc` is used to compile the base-level program:

```
% myocc -- -o sample sample.cc
```

This compiles `sample.cc` and produces an executable file `sample`.

OPTIONS

`-D` Define a macro *name* as *def*.

`-E` Don't run the back-end C++ compiler. Stop after generating a `.ii` file.

`-I` Add a *directory* to the search path of the `#include` directive.

`-M` Specify an *option* with *value*. It is passed to metaobjects.

`-V` Show the version number.

`-c` Suppress linking and produce a `.o` file.

`-l` Print the list of statically loaded metaclasses.

`-m` Produce a shared library (a `.so` file.) This is used to compile a metaclass. If *file_name* is specified, the name of the shared library is *file_name*`.so`. If the `-c` option is specified together, `occ` produces a `.so` file, which should be linked by the user to be a shared library.

`-n` Suppress invoking the preprocessor.

`-p` Stop after the parsing stage. No translation is done.

`-s` Print the whole parse tree of the given source program. Don't perform translation or compilation. If no source file is given, `occ` reads from the standard input.

`-v` Specify the verbose mode.

`--regular-c++` Inhibit the extended syntax. This enables the keyword `metaclass` to be used as a variable name. This option is useful when parsing legacy code being not intended to translation. When this option is used, the symbol `__opencxx` is not defined.

`--` The following options are interpreted as options for the back-end C++ compiler. For example, if you type

```
occ -I.. -- -g foo.c
```

Then the `-g` option is passed to the C++ compiler. Note that these options are not passed to the C++ preprocessor. The `-D` and `-I` options need to be placed before `--`.

FILES

`file.{cc,C,c,cpp,cxx,mc}`	source file.
`file.occ`	output file after C++ preprocessing.
`file.ii`	output file after translation.
`file.o`	object file.
`file.so`	shared library dynamically loaded by `occ`.
`opencxx.a`	library to link with meta-level program.

NOTES

- While the C++ processor is running, the macro `__opencxx` is predefined.
- The programs compiled by `occ` do not need any runtime libraries or a garbage collector unless the meta-level program requires them at the base level.

COPYRIGHT

AUTHOR

Shigeru Chiba, University of Tsukuba, Japan.
Email: `chiba@is.tsukuba.ac.jp`