Open C++ Tutorial[*]

Shigeru Chiba
Institute of Information Science and Electronics
University of Tsukuba
chiba@is.tsukuba.ac.jp

1 Introduction

OpenC++ is an extensible language based on C++. The extended features of OpenC++ are specified by a meta-level program given at compile time. For distinction, regular programs written in OpenC++ are called base-level programs. If no meta-level program is given, OpenC++ is identical to regular C++.

The meta-level program extends OpenC++ through the interface called the OpenC++ MOP. The OpenC++ compiler consists of three stages: preprocessor, source-to-source translator from OpenC++ to C++, and the back-end C++ compiler. The OpenC++ MOP is an interface to control the translator at the second stage. It allows to specify how an extended feature of OpenC++ is translated into regular C++ code.

An extended feature of OpenC++ is supplied as an *add-on* software for the compiler. The add-on software consists of not only the meta-level program but also runtime support code. The runtime support code provides classes and functions used by the base-level program translated into C++. The base-level program in OpenC++ is first translated into C++ according to the meta-level program. Then it is linked with the runtime support code to be executable code. This flow is illustrated by Figure 1.

The meta-level program is also written in OpenC++ since OpenC++ is a self-reflective language. It defines new metaobjects to control source-to-source translation. The metaobjects are the meta-level representation of the base-level program and they perform the translation. Details of the metaobjects are specified by the

[*]This document was originally written as part of: Shigeru Chiba, "OpenC++ Programmer's Guide for Version 2," Xerox PARC Technical Report, SPL-96-024, 1996.

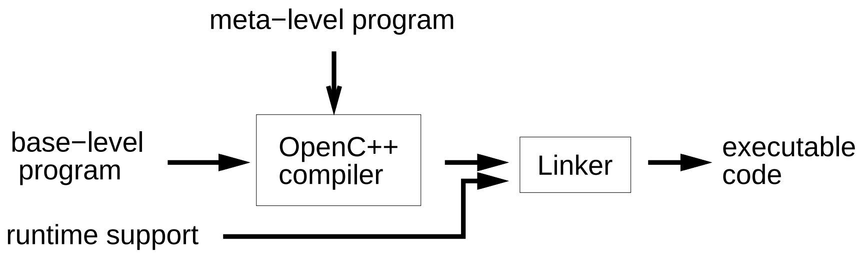

Figure 1: The OpenC++ Compiler

OpenC++ MOP. In the following sections, we go through several examples so that we illustrate how the OpenC++ MOP is used to implement language extensions.

2 Verbose Objects

A MOP version of "hello world" is verbose objects, which print a message for every member function call. We choose them as our first example.

The MOP programming in OpenC++ is done through three steps: (1) decide what the base-level program should look like, (2) figure out what it should be translated into and what runtime support code is needed, and (3) write a meta-level program to perform the translation and also write the runtime support code. We implement the verbose objects through these steps.

2.1 What the base-level program should look like

In the verbose objects example, we want to keep the base-level program looking the same as much as possible. The only change should be to put an annotation that specifies which class of objects print a message for every member function call. Suppose that we want to make a class `Person` verbose. The base-level program should be something like:

```
// person.cc
#include <stdio.h>

metaclass VerboseClass Person;     // metaclass declaration
class Person {
public:
    Person(int age);
    int Age() { return age; }
    int BirthdayComes() { return ++age; }
private:
    int age;
};

main()
{
    Person billy(24);
    printf("age %d\n", billy.Age());
    printf("age %d\n", billy.BirthdayComes());
}
```

Note that the `metaclass` declaration in the first line is the only difference from regular C++ code. It specifies that `Person` objects print a message for every member function call.

2.2 What the base-level program should be translated

In order to make the program above work as we expect, member function calls on `Person` objects must be appropriately translated to print a message. For example, the two expressions:

```
billy.Age()
billy.BirthdayComes()
```

must be translated respectively into:

```
(puts("Age()"), billy.Age())
(puts("BirthdayComes()"), billy.BirthdayComes())
```

Note that the resulting value of the comma expression `(x , y)` is `y`. So the resulting values of the substituted expressions are the same as those of the original ones.

2.3 Write a meta-level program

Now, we write a meta-level program. What we should do is to translate only member function calls on `Person` objects in the way shown above. We can easily do that if we use the MOP.

In OpenC++, classes are objects as in Smalltalk. We call them class metaobjects when we refer to their meta-level representation. A unique feature of OpenC++ is that a class metaobject translates expressions involving the class at compile time. For example, the class metaobject for `Person` translates a member function call `billy.Age()` since `billy` is a `Person` object.

By default, class metaobjects are identity functions; they do not change the program. So, to implement our translation, we define a new metaclass — a new class for class metaobjects — and use it to make the class metaobject for `Person`.

The metaclass for a class is specified by the `metaclass` declaration at the base level. For example, recall that the base-level program `person.cc` contains this:

```
metaclass VerboseClass Person;     // metaclass declaration
```

This declaration specifies that the class metaobject for `Person` is an instance of `VerboseClass`.

A new metaclass must be a subclass of the default metaclass `Class`. Here is the definition of our new metaclass `VerboseClass`:

```
// VerboseClass.mc
#include "mop.h"

class VerboseClass : public Class {
public:
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*,
                               Ptree*, Ptree*);
};

Ptree* VerboseClass::TranslateMemberCall(Environment* env,
        Ptree* object, Ptree* op, Ptree* member, Ptree* arglist)
{
    return Ptree::Make("(puts(\"%p()\"), %p)",
                       member,
                       Class::TranslateMemberCall(env, ob-
ject, op,
                                                  member, arglist));
}
```

The metaclass `VerboseClass` is just a regular C++ class. It inherits from `Class` and overrides one member function. `TranslateMemberCall()` takes an expression such as `billy.Age()` and returns the translated one. Both the given expression and the translated one are represented in the form of parse tree. `Ptree` is the data type for that representation.

Since the class metaobject for `Person` is responsible only for the translation involving the class `Person`, `TranslateMemberCall()` does not have to care about other classes. It just constructs a comma expression:

`(puts("` *member-name*`"),` *member-call*`)` from the original expression. `Ptree::Make()` is a convenience function to construct a new parse tree. `%p` is replaced with the following argument.

We do not need many concepts to write a meta-level program. As we saw above, the key concepts are only three. Here, we summarize these concepts:

class metaobject: The representation of a class at the meta level.

metaclass: A class whose instances are class metaobjects.

metaclass `Class`: The default metaclass. It is named because its instances are class metaobjects.

2.4 Compile, debug, and run

We first compile the meta-level program and extend the OpenC++ compiler, which is used to compile the base-level program. Because OpenC++ is a reflective language, the meta-level program is compiled by the OpenC++ compiler itself.

```
% occ -m -- -g VerboseClass.mc
```

The option `-m` means that a metaclass is compiled. The other option `-g` following `--` is passed to the back-end C++ compiler. `VerboseClass.mc` is compiled and a shared library (dynamically loadable library) is produced.

Next, we compile the base-level program `person.cc` with the compiled metaclass:

```
% occ -- -g -o person person.cc
```

The OpenC++ compiler dynamically loads `VerboseClass` if it encounters the `metaclass` declaration.

Now, we got an executable file `person`. It prints member function names if they are executed:

```
% person
Age()
age 24
BirthdayComes()
age 25
%
```

The OpenC++ MOP also provides some functions for debugging. First, programmers may use `Display()` on `Ptree` metaobjects to debug a compiler. This function prints the parse tree represented by the `Ptree` metaobject. For example, if the debugger is `gdb`, programmers may print the parse tree pointed to by a variable `object` in this way:

```
% gdb occc
    :
(gdb) print object->Display()
billy
$1 = void
(gdb)
```

Note that `gdb` cannot handle dynamically loaded code. To debug a compiled metaclass, it should be statically linked to the compiler. See the command reference section of the manual.

Furtheremroe, the OpenC++ compiler accepts the `-s` option to print the whole parse tree of the given program. The parse tree is printed in the form of nested list:

```
% myocc -s person.cc
[typedef [char] [* __gnuc_va_list] ;]
    :
[metaclass VerboseClass Person nil ;]
[[[class Person nil [{ [
    [public :]
    [nil [Person ( [[[int] [i]]] )] [{ [
        [[age = i] ;]
    ] }]]
    [[int] [Age ( nil )] [{ [
        [return age ;]
    ] }]]
    [[int] [BirthdayComes ( nil )] [{ [
        [return [++ age] ;]
    ] }]]
    [private :]
    [[int] [age] ;]
] }]]] ;]
[nil nil [main ( nil )] [{ [
    [[Person] [billy ( [24] )] ;]
    [[printf [( ["age %d\n" , [billy . Age [( nil )]]] )]] ;]
    [[printf [( ["age %d\n" , [billy . BirthdayComes ...
] }]]
%
```

This option makes the compiler just invoke the preprocessor and prints the parse tree of the preprocessed program. `[ ]` denotes a nested list. The compiler does not perform translation or compilation.

3 Syntax Extension for Verbose Objects

In the verbose object extension above, the base-level programmers have to write the `metaclass` declaration. The extension will be much easier to use if it provides easy syntax to declare verbose objects. Suppose that the base-level programmers may write something like this:

```
// person.cc
verbose class Person {
public:
    Person(int age);
    int Age() { return age; }
    int BirthdayComes() { return ++age; }
private:
    int age;
};
```

Note that the class declaration begins with a new keyword `verbose` but there is no `metaclass` declaration in the code above.

This sort of syntax extension is easy to implement with the OpenC++ MOP. To make the new keyword `verbose` available, the meta-level program must call `Class::RegisterMetaclass()` during the initialization phase of the compiler. So we add a `static` member function `Initialize()` to the class `VerboseClass`. It is automatically invoked at beginning by the MOP.

```
// VerboseClass2.mc
#include "mop.h"
class VerboseClass : public Class {
public:
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*,
                               Ptree*, Ptree*);
    static bool Initialize();
};

bool VerboseClass::Initialize()
{
    RegisterMetaclass("verbose", "VerboseClass2");
    return TRUE;
}
```

`RegisterMetaclass()` defines a new keyword `verbose`. If a class declaration begins with that keyword, then the compiler recognizes that the metaclass is `VerboseClass2`. This is all that we need for the syntax extension. Now the new compiler accepts the `verbose` keyword.

4 Matrix Library

The next example is a matrix library. It shows how the OpenC++ MOP works to specialize an optimization scheme for a particular class. The `Matrix` class is a popular example in C++ to show the usage of operator overloading. On the other hand, it is also famous that the typical implementation of the `Matrix` class is not efficient in practice. Let's think about how this statement is executed:

```
a = b + c - d;
```

The variables `a`, `b`, `c`, and `d` are `Matrix` objects. The statement is executed by invoking the operator functions `+`, `-`, and `=`. But the best execution is to inline the operator functions in advance to replace the statement:

```
for(int i = 0; i < N; ++i)
    a.element[i] = b.element[i] + c.element[i] - d.element[i];
```

C++'s `inline` specifier does not do this kind of smart inlining. It simply extracts a function definition but it does not fuse multiple extracted functions into efficient code as shown above. Expecting that the C++ compiler automatically performs the fusion is not realistic.

We use the OpenC++ MOP to implement this smart inlining specialized for the `Matrix` class. Again, we follow the three steps of the OpenC++ programming.

4.1 What the base-level program should look like

The objective of the matrix library is to provide the matrix data type as it is a built-in type. So the base-level programmers should be able to write:

```
Matrix a, b, c;
double k;
    :
a = a * a + b - k * c;
```

Note that the last line includes both a vector product `a * a` and a scalar product `k * c`.

4.2 What the base-level program should be translated

We've already discussed this step. The expressions involving `Matrix` objects are inlined as we showed above. We do not inline the expressions if they include more than one vector products. The gain by the inlining is relatively zero against two vector products.

Unlike the verbose objects example, we need runtime support code in this example. It is the class definition of `Matrix`. Note that the base-level programmers do not define `Matrix` by themselves. `Matrix` must be supplied as part of the compiler add-on for matrix arithmetics.

4.3 Write a meta-level program

To implement the inlining, we define a new metaclass `MatrixClass`. It is a metaclass only for `Matrix`. `MatrixClass` overrides a member function `TranslateAssign()`:

```
// MatrixClass.mc

Ptree* MatrixClass::TranslateAssign(Environment* env,
                        Ptree* object, Ptree* op, Ptree* expr)
{
    if(we can inline on the expression)
        return generate optimized code
    else
        return Class::TranslateAssign(env, object, op, expr);
}
```

This member function translates an assignment expression. `object` specifies the L-value expression, `op` specifies the assignment operator such as `=` and `+=`, and `expr` specifies the assigned expression. If the inlining is not applicable, this function invokes `TranslateAssign()` of the base class. Otherwise, it parses the given `expr` and generate optimized code.

Since `expr` is already a parse tree, what this function has to do is to traverse the tree and sort terms in the expression. It is defined as a recursive function that performs pattern matching for each sub-expression. Note that each operator makes a sub-expression. So an expression such as `a + b - c` is represented by a parser tree:

```
[[a + b] - c]
```

The OpenC++ MOP provides a convenience function `Ptree::Match()` for pattern matching. So the tree traverse is described as follows:

```
static bool ParseTerms(Environment* env, Ptree* expr, int k)
{
    Ptree* lexpr;
    Ptree* rexpr;

    if(expr->IsLeaf()){      // if expr is a variable
        termTable[numOfTerms].expr = expr;
        termTable[numOfTerms].k = k;
        ++numOfTerms;
        return TRUE;
    }
    else if(Ptree::Match(expr, "[%? + %?]", &lexpr, &rexpr))
        return ParseTerms(env, lexpr, k)
```

```
                && ParseTerms(env, rexpr, k);
    else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
        return ParseTerms(env, lexpr, k)
                && ParseTerms(env, rexpr, -k);
    else if(Ptree::Match(expr, "[( %? )]", &lexpr))
        return ParseTerms(env, lexpr, k);
    else if(Ptree::Match(expr, "[- %?]", &rexpr))
        return ParseTerms(env, rexpr, -k);
    else
        return FALSE;
}
```

This function recursively traverses the given parse tree `expr` and stores the variables in `expr` into an array `termTable`. It also stores the flag (`+` or `-`) of the variable into the array. The returned value is `TRUE` if the sorting is successfully done.

After `ParseTerms()` is successfully executed, each term in the expression is stored in the array `termTable`. The rest of the work is to construct an inlined code from that array:

```
static Ptree* DoOptimize0(Ptree* object)
{
    Ptree* index = Ptree::GenSym();
    return Ptree::Make(
        "for(int %p = 0; %p < %s * %s; ++%p)\n"
        "    %p.element[%p] = %p;",
        index, index, SIZE, SIZE, index,
        object, index, MakeInlineExpr(index));
}
```

`Ptree::GenSym()` returns a symbol name that has not been used. It is used as a loop variable. `MakeInlineExpr()` looks at the array and produces an inlined expression:

```
static Ptree* MakeInlineExpr(Ptree* index_var)
{
    int i;
    Ptree* expr;
    Ptree* inline_expr = nil;

    for(i = numOfTerms - 1; i >= 0; --i){
        char op;
        if(termTable[i].k > 0)
            op = '+';
        else
            op = '-';
        expr = Ptree::Make("%c %p.element[%p]",
                           op, termTable[i].expr, index_var);
        inline_expr = Ptree::Cons(expr, inline_expr);
    }

    return inline_expr;
}
```

The complete program of this example is `MatrixClass.mc`, which is distributed together with the OpenC++ compiler. See that program for more details. It deals with the scalar and vector products as well as simple `+` and `-` operators.

4.4 Write runtime support code

Writing the runtime support code is straightforward. The class `Matrix` is defined in regular C++ except the `metaclass` declaration:

```
// matrix.h
const N = 3;

metaclass MatrixClass Matrix;
class Matrix {
public:
    Matrix(double);
    Matrix& operator = (Matrix&);
        :
    double element[N * N];
};

Matrix& operator + (Matrix&, Matrix&);
Matrix& operator - (Matrix&, Matrix&);
Matrix& operator * (Matrix&, Matrix&);
Matrix& operator * (double, Matrix&);
```

Note that the class `Matrix` is a complete C++ class. It still works if the `metaclass` declaration is erased. For more details, see the sample program `matrix.cc`. They must be compiled by the OpenC++ compiler.

5 Syntax Extension for the Matrix Library

Initializer

We can also implement syntax sugar for the matrix library. First of all, we enable the following style of initialization:

```
Matrix r = { 0.5, -0.86, 0, 0.86, 0.5, 0, 0, 0, 1 };
```

This notation is analogous to initialization of arrays. In regular C++, however, an object cannot take an aggregate as its initial value. So we translate the statement shown above by `MatrixClass` into this correct C++ code:

```
double tmp[] = { 0.5, -0.86, 0, 0.86, 0.5, 0, 0, 0, 1 };
Matrix r = tmp;
```

To do this translation, `MatrixClass` must override a member function `TranslateInitializer()`:

```
// MatrixClass.mc

Ptree* MatrixClass::TranslateInitializer(Environment* env, Ptree* name,
            Ptree* init)
{
    Ptree* sep = Ptree::First(init);
    Ptree* expr = Ptree::Second(init);
    if(sep->Eq('=') && Ptree::Match(val, "[{ %* }]")) {
        Ptree* tmp = Ptree::GenSym();
        InsertBeforeStatement(Ptree::Make("double %p[] = %p;\n",
                              tmp, expr));
        return Ptree::Make("= %p", tmp);
    }
    else
        return Class::TranslateInitializer(env, init, before,
                                           after);
}
```

This member function translates the initializer of a `Matrix` object. For example, it receives, as the argument `init`, the initializer `= { 0.5, ... }` of `r`. If the initializer is an aggregate, this member function translates it as we mentioned above. The temporary array `tmp` is inserted before the variable declaration by `InsertBeforeStatement()`.

The `forall` statement

The second syntax sugar we show is a new kind of loop statement. For example, the programmer may write:

```
Matrix m;
    :
m.forall(e){ e = 0.0; }
```

`e` is bound to each element during the loop. The programmer may write any statements between `{` and `}`. The loop statement above assigns `0.0` to all the elements of the matrix `m`. This new loop statement should be translated into this:

```
for(int i = 0; i < N; ++i){
    double& e = m.element[i];
    e = 0.0;
}
```

The OpenC++ MOP allows programmers to implement a new kind of statement such as `forall`. To implement this statement, first we have to register a new keyword `forall`:

```
// MatrixClass.mc

bool MatrixClass::Initialize()
{
    RegisterNewWhileStatement("forall");
    return TRUE;
}
```

`Initialize()` is a member function automatically invoked at the beginning of compilation.

We also have to define what the `forall` statement is translated into. `MatrixClass` overrides a member function `TranslateUserStatement()`:

```
Ptree* MatrixClass::TranslateUserStatement(Environment* env,
         Ptree* object, Ptree* op, Ptree* keyword, Ptree* rest)
{
    Ptree *tmp, *body, *index;

    Ptree::Match(rest, "[([%?]) %?]", &tmp, &body);
    index = Ptree::GenSym();
    return Ptree::Make(
        "for(int %p = 0; %p < %s * %s; ++%p){\n"
        "    double& %p = %p%p element[%p];\n"
        "    %p }\n",
        index, index, SIZE, SIZE, index,
        tmp, object, op, index, TranslateStatement(env, body));
}
```

The `forall` statement is parsed so that `object`, `op`, and `keyword` are bound to `m . forall`, respectively. `rest` is bound to the rest of code `(e){ e = 0.0; }`. `TranslateUserStatement()` uses those arguments to construct the substituted code. `TranslateStatement()` is called to recursively translate the `body` part of the `forall` statement.

6 Before-Method

CLOS provides a useful mechanism called before- and after- methods. They are special methods that are automatically executed before or after the primary method is executed.

6.1 What the base-level program should look like

We implement before-methods in OpenC++. For simplicity, if the name of a member function is `before_f()`, then our implementation regards this member function as the before-method for the member function `f()`. We don't introduce any syntax extension. For example,

```
metaclass Queue : BeforeClass;
class Queue {
public:
    Queue(){ i = 0; }
    void Put(int);
    void before_Put();
    int Peek();

private:
    int buffer[SIZE];
    int i;
};
```

`Put()` has a before-method `before_Put()` whereas `Peek()` does not since `before_Peek()` is not defined in the class `Queue`.

The before-method is automatically executed when the primary method is called. If the programmer say:

```
Queue q;
   :
q.Put(3);
int k = q.Peek();
```

The execution of `q.Put(3)` is preceded by that of the before-method `q.before_Put()`. Since `Peek()` does not have a before-method, the execution of `q.Peek()` is not preceded by any other function.

6.2 What the base-level program should be translated

In this extension, the class declaration does not require any change. Only member function calls need to be translated. For example,

```
q.Put(3)
```

should be translated into:

```
((tmp = &q)->before_Put(), tmp->Put(3))
```

This expression first stores the address of `q` in a temporary variable `tmp` and then calls `before_Put()` and `Put()`. The address of `q` should be stored in the temporary variable to avoid evaluating `q` more than once. Also, the temporary variable must be declared in advance.

6.3 Write a meta-level program

The metaclass `BeforeClass` overrides `TranslateMemberCall()` to implement the translation mentioned above. The complete program of `BeforeClass` is `BeforeClass.mc` in the distribution package. Here, we explain some important topics in the program.

First of all, we have to decide whether there is a before-method for a given member function. `BeforeFunction()` does this work:

```
Ptree* BeforeClass::BeforeFunction(Ptree* name)
{
    Ptree* before = Ptree::Make("before_%p", name);
    if(LookupMember(before))
        return before;
    else
        return nil;
}
```

In the first line, this produces the name of the before-method by `Ptree::Make()`. Then it calls `LookupMember()` supplied by `Class`. `LookupMember()` returns `true` if the class has a member that matches the given name.

The next issue is a temporary variable. We have to appropriately insert a variable declaration to use a temporary variable. The name of the temporary variable is obtained by calling `Ptree::GenSym()`. The difficulty is how to share the temporary variable among member function calls. To do this, we record the temporary variable in the environment when we first declare the temporary variable.

```
Ptree* class_name = Name();
Ptree *tmpvar = (Ptree*)LookupClientData(env, class_name);
if(tmpvar == nil) {
    tmpvar = Ptree::GenSym();
    Ptree* decl = Ptree::Make("%p * %p;", class_name, tmpvar);
    InsertDeclaration(env, decl, class_name, tmpvar);
}

return Ptree::Make("((%p=%c%p)->%p(), %p->%p%p)",
                   varname, (op->Eq('.') ? '&' : ' '), object,
                   before_func, varname, member, arglist);
```

This is the core part of `TranslateMemberCall()` supplied by `BeforeClass`. It first calls `LookupClientData()` and looks for a temporary variable that is already declared. If it is not found, a variable declaration `decl` is produced by `Make()` and it is inserted into the translated program by `InsertDeclaration()`. `InsertDeclaration()` also records the temporary variable for future reference.

7 Wrapper Function

A wrapper function is useful to implement language extensions such as concurrency. A wrapper function is generated by the compiler and it intercepts the call of the original “wrapped” function. For example, the wrapper function may perform synchronization before executing the original function. The original function is not invoked unless the wrapper function explicitly calls it.

7.1 What the base-level program should be translated

We show a metaclass `WrapperClass` that generates wrapper functions. If `WrapperClass` is specified, it generates wrapper functions for the member functions of the class. And it translates the program so that the wrapper functions are invoked instead of the wrapped member functions. For example, suppose that the program is something like this:

```
metaclass WrapperClass Point;
class Point {
public:
    void Move(int, int);
    int x, y;
};

void Point::Move(int new_x, int new_y)
{
    x = new_x; y = new_y;
}

void f()
{
    Point p;
    p.Move(3, 5);    // call Move()
}
```

The compiler renames `Move()` to be `orig_Move()` and generates a wrapper function `Move()` for `orig_Move()`. The translated program should be this:

```
class Point {
public:
    void orig_Move(int, int);
    int x, y;
public:
    void Move(int, int);
};

void Point::orig_Move(int new_x, int new_y)    // renamed
{
    x = new_x; y = new_y;
}

void Point::Move(int p1, int p2)  // generated wrapper
{
    // should do something here in a real example
    Move(p1, p2);
}
```

For simplicity, we make the wrapper function just invoke the wrapped function `Move()` without doing anything else. In practice, it would do something necessary.

7.2 Write a meta-level program

`WrapperClass` has to do only two things: (1) to rename member functions and (2) to generate wrapper functions. `WrapperClass` overrides `TranslateClass()` for (1) and (2), and `TranslateMemberFunction()` for (1).

First, we show `TranslateClass()`. Its work is to translate the body of a class declaration. It examines a member of the class and, if the member is a function, it inserts the declaration of the wrapper function for that member.

```
Ptree* WrapperClass::TranslateBody(Environment* env, Ptree* body)
{
    Member member;
    int i = 0;
    while(NthMember(i++, member))
        if(member.IsPublic() && member.IsFunction()
           && !member.IsConstructor() && !member.IsDestructor()){
            Member wrapper = member;
            Ptree* org_name = NewMemberName(member.Name());
            member.SetName(org_name);
            ChangeMember(member);
            MakeWrapper(wrapper, org_name);
            AppendMember(wrapper, Class::Public);
    }
}

Ptree* WrapperClass::NewMemberName(Ptree* name)
{
    return Ptree::Make("org_%p", name);
}
```

In the `while` loop, we first examine whether the `i`-th member is a `public` member function. If so, we make a copy of that member and change the name of the original one to be `org_name`. This change is reflected on the class declaration by `ChangeMember()`. Then we modify the copy of that member so that is is a wrapper function. `MakeWrapper()` performs this modification. The modified copy is finally appended to the class declaration by `AppendMember()`.

`MakeWrapper()` substitutes an expression calling the original member for the function body of the given member:

```
void WrapperClass::MakeWrapper(Member& member, Ptree* org_name)
{
    Ptree* body = MakeWrapperBody(member, org_name);
    member.SetFunctionBody(Ptree::Make("{ %p }\n", body));
}

Ptree* WrapperClass::MakeWrapperBody(Member& member, Ptree* org_name)
{
    TypeInfo t;
    Ptree* call_expr = Ptree::Make("%p(%p)", org_name, member.Arguments())
    member.Signature(t);
    t.Dereference(); // get the return type
    if((t.IsBuiltInType() & VoidType) || t.IsNoReturnType())
        return Ptree::Make("%p;\n", call_expr);
    else{
        Ptree* rvar = Ptree::Make("_rvalue");
        Ptree* rvar_decl = t.MakePtree(rvar);
        return Ptree::Make("%p = %p;\n"
                           "return %p;\n", rvar_decl, call_expr, rvar);
    }
}
```

For example, `MakeWrapper()` alters the function body of a given member `f()` and generates something like this:

```
int f(int i){
    int _rvalue = org_f(i);
    return _rvalue;
}
```

To implement this translation, we use `Member` and `TypeInfo` metaobjects.

First, we must call `Arguments()` on a `Member` metaobject to construct an expression for calling the original member. `Arguments()` returns the formal-argument names of that member. If some of the names are omitted, `Arguments()` implicitly inserts names. For example, if the original member is:

```
int f(int, int j) { reutrn j; }
```

Then the call of `Arguments()` on this member translates this member into:

```
int f(int _p0, int j) { reutrn j; }
```

And returns a `Ptree` metaobject representing the argument list `[_p0 , j]`.

To obtain the return type of the member, we call `Signature()` on the `Member` metaobject. This returns a `TypeInfo` metaobject representing the signautre of the member. The return type of the member is the dereferenced type of that signature, which is obtained by calling `Dereference()` on the signature.

Once we obtain the `TypeInfo` metaobject representing the return type of the member, we can construct the declaration of a variable of that type. If we call `MakePtree()` on a `TypeInfo` metaobject, then it returns a declaration of the variable given through the argument. Suppose that a `TypeInfo` metaobject `t` is the pointer type to integer. Then:

```
t.MakePtree(Ptree::Make("i"))
```

returns:

```
int* i
```

If `MakePtree()` is called without an argument, then it returns:

```
int*
```

If `t` is a function type, we can give a function name and get a parse tree representing the function declaration. For example:

```
t.MakePtree(Ptree::Make("foo"))
```

returns if `t` is the function type that takes two integer arguments and returns a pointer to a character:

```
char* foo(int, int)
```

If `t` is the pointer type to that function, the returned parse tree is:

```
char* (*foo)(int, int)
```

To finish the implementation of `WrapperClass`, we also have to override `TranslateMemberFunction()`. `TranslateClass()` that we defined above renames member functions appearing in the class declaration, but it does not process the (not inlined) implementation of those functions.

```
void WrapperClass::TranslateMemberFunction(Environment* env, Mem-
ber& member)
{
    if(member.IsPublic() && !member.IsConstructor()
       && !member.IsDestructor())
        member.SetName(NewMemberName(member.Name()));
}
```

`TranslateMemberFunction()` renames the given member function if it is `public` but not a constructor or a destructor. Note that `ChangeMember()` is not called in this member function unlike `TranslateClass()`. The changes of `member` is always reflected on the source code after this function finishes.

7.3 Subclass of `WrapperClass`

The complete program of `WrapperClass` is found in `WrapperClass.h` and `WrapperClass.mc`, which is distributed together with the OpenC++ compiler. Although the wrapper functions generated by `WrapperClass` do not perform anything except calling the original member function, we can define a subclass of `WrapperClass` to generate the wrapper functions that we need. (Note that, to make the subclass effective, we also have to edit the metaclass declaration so that the compiler selects the subclass for `Point`.)

For example, suppose that we need a wrapper function that perform synchronization before calling the original member function. This sort of wrapper function is typical in concurrent programming. To implement this, we just define a subclass `SyncClass` and override `WrapperBody()`:

```
Ptree* SyncClass::MakeWrapperBody(Member& member, Ptree* name)
{
    Ptree* body = WrapperClass::MakeWrapperBody(member, name);
    return Ptree::qMake("synchronize();\n ‘body‘");
}
```

This inserts `synchronize();` before the `return` statement.

As we see above, carefully designed metaclasses can be reused as a super class of another metaclass. Such metaclasses, that is, metaclass libraries, make it easier to write other metaclasses. Indeed, `MatrixClass` in the matrix example should be re-implemented so that other metaclasses such as `ComplexClass` can share the code for inlining with `MatrixClass`.