

A Study of Compile-time Metaobject Protocol

By

Shigeru Chiba
`chiba@is.s.u-tokyo.ac.jp`

November 1996

A Dissertation Submitted to
Department of Information Science
Graduate School of Science
The University of Tokyo

In Partial Fulfillment of the Requirements
For the Degree of Doctor of Science.

Copyright ©1996 by Shigeru Chiba. All Rights Reserved.

Shigeru Chiba

A Study of Compile-time Metaobject Protocol

Ph.D. Dissertation, Graduate School of Science, The University of Tokyo.

November 1996

**A Metaobject Protocol
for Enabling Better C++ Libraries**

Shigeru Chiba

**Department of Information Science
The University of Tokyo
chiba@is.s.u-tokyo.ac.jp**

Copyright ©1996 by Shigeru Chiba. All Rights Reserved.

Abstract

C++ cannot be used to implement control/data abstractions as a library if their implementations require specialized code for each user code. This problem limits programmers to write libraries in two ways: it makes some kinds of useful abstractions that inherently require such facilities impossible to implement, and it makes other abstractions difficult to implement efficiently.

The OpenC++ MOP addresses this problem by providing libraries the ability to pre-process a program in a context-sensitive and non-local way. That is, libraries can instantiate specialized code depending on how the library is used and, if needed, substitute it for the original user code. The basic protocol structure of the OpenC++ MOP is based on that of the CLOS MOP, but the OpenC++ MOP runs metaobjects only at compile time. This means that the OpenC++ MOP does not imply runtime penalties caused by dispatching to metaobjects.

Biographical Sketch

Shigeru Chiba was born in Tokyo, Japan, in 1968. After living in Osaka for a few years, he spent most of his childhood in Yokohama, a suburb of Tokyo. As many other children living in a suburb of Tokyo do, he went to a high school in Tokyo, and first met meta computation there when his good friend at the high school let him borrow a book, *Gödel, Escher, Bach: an Eternal Golden Braid*, by Douglas R. Hofstadter. Looking back now, it was natural that he was captured by the idea of meta computation, which was plainly explained in the book.

But the life at university was attractive enough to have him forget meta computation. It had actually gone from his mind until it emerged again in front of him at the department of information science. He was supposed to walk away from it, but what he really did was to dance along a circle and come back to the same place. It might be his karma although he hopes he drew not a circle but a helix.

The second encounter eventually took him to California, where a guru is living. He has not yet known what is the next.

Acknowledgments

This thesis was supervised by Gregor Kiczales. I would like to express my deep gratitude to him. I also profoundly thank Takashi Masuda, my supervisor at The University of Tokyo. He continuously supported and encouraged me during this work. John Lamping gave me a plenty of constructive comments and criticism, which greatly helped me write this thesis. Finally, I greatly thank my thesis committee, Akinori Yonezawa, Kei Hiraki, Yoshio Oyanagi, Masami Hagiya, Hiroshi Imai, and Naoki Kobayashi.

I am grateful for financial support received from Japan Society for the Promotion of Science (Nihon Gakujyutsu Sinkokukai), from The Information-technology Promotion Agency, Japan (IPA), from Xerox Corporation, from Japan Foundation for the Education and Training of Information Processing (Jyouhousyori Kyouiku Kensyu Jyosei Zaidan), and from International Information Science Foundation, Japan (Jyouhou Kagaku Kokusai Koryu Zaidan). Also, Department of Information Science at The University of Tokyo let me extend my stay at PARC with exceptional courtesy.

This thesis project was conducted at Xerox Palo Alto Research Center (PARC) in Palo Alto, California, U.S.A.

Contents

1	Introduction	1
2	Limitations of C++	6
2.1	Inheritance	7
2.2	Template	12
2.3	Summary	17
3	Techniques for Processing a Program	19
3.1	Lisp Macros	19
3.2	3-Lisp	21
3.3	The CLOS MOP	27
3.4	Summary	36
4	The OpenC++ MOP	39
4.1	Overview	39
4.2	Context-sensitive and Non-local	42
4.3	Syntax Extension	47
4.4	What is New?	49
4.5	Summary	53
5	Meta Helix	55
5.1	Implementation Level Conflation	55
5.2	Inadequate Solutions	60
5.3	The Meta Helix Architecture	63
5.4	Implementing the Meta Helix	65
5.5	Summary	68
6	Libraries in OpenC++	70
6.1	Named Object Library	70
6.2	Distributed Object Library	72

6.3	Wrapper Library	79
6.4	Implementation of qMake()	85
6.5	Metaclass	87
6.6	Vector Library	90
6.7	The Standard Template Library	94
6.8	The OOPACK Benchmark Test	98
6.9	Summary	104
7	Conclusion	106
A	Backquote	115
A.1	Quote	115
A.2	Backquote	115
B	Reference Manual	117
B.1	Introduction	117
B.2	Tutorial	118
B.2.1	Verbose Objects	119
B.2.2	Syntax Extension for Verbose Objects	124
B.2.3	Matrix Library	125
B.2.4	Syntax Extension for the Matrix Library	129
B.2.5	Before-Method	131
B.2.6	Wrapper Function	134
B.3	Base-Level Language (OpenC++)	140
B.3.1	Base-level Connection to the MOP	140
B.3.2	Syntax Extensions	140
B.3.3	Loosened Grammar	142
B.4	Metaobject Protocol (MOP)	143
B.4.1	Representation of Program Text	143
B.4.2	Representation of Types	151
B.4.3	Representation of Environment	152
B.4.4	Class Metaobjects	154
B.4.5	Error Message	165
B.4.6	C++ Preprocessing	166
B.5	Command Reference	167
C	Programs	169
C.1	The Distributed Object Library	169
C.2	The Wrapper Function Metaclass Library	174
C.3	Vector Library	177

CONTENTS

vi

C.4 The Standard Template Library	179
C.5 OOPACK benchmark	185

List of Figures

1.1	The common part and the specialized part of a library	3
3.1	The CLOS MOP is metacircular	29
3.2	An improved implementation of 3-Lisp	31
3.3	An metacircular implementation	33
5.1	Two handles with the same abstraction	63
5.2	The implementation relation between interfaces	65
5.3	The Meta Helix supports n implementation levels.	66
6.1	The instance-of relationship among metaclasses	88
B.1	The OpenC++ Compiler	119
B.2	Instance-of Relationship	165

List of Tables

3.1	Lisp Macros, 3-Lisp, and the CLOS MOP	37
6.1	Execution performance of the vector library (μ sec.)	94
6.2	Execution performance of STL (msec.)	99
6.3	Execution time of the OOPACK benchmark (sec.)	104
B.1	<code>static</code> member functions on <code>Ptree</code>	144
B.2	<code>enum</code> constants returned by <code>Ptree::WhatIs()</code>	146

Chapter 1

Introduction

*In computing science, elegance is not a luxury
but a matter of life and death.*

— E. W. Dijkstra

Execution speed is the only metric in computer science.

— Kei Hiraki

One of today's significant concerns in software industry would be to decrease time and costs of software development. Anybody would agree that good libraries promote code reuse and thereby contribute to rapid and low-cost software development. This thesis deals with a language mechanism for programmers to write good libraries.

What are good libraries? Good libraries provide useful control/data abstractions, which are commonly used for a number of applications, high-level to improve readability of programs, and simple and intuitive to avoid leading the library users to misuse the libraries and cause serious errors. Also, efficiency is another criterion of good libraries. Since writing a good library is a very difficult task, the programmer is required to have not only good programming skill but also deep knowledge about the application domain in that the library helps programmers. For example, library developers need to know typical functions and data structures used in that domain.

Development of good libraries also needs assistance of language designers, who should provide language mechanisms for writing good libraries. This is a more realistic option for the designers than including all desirable control/data abstractions in the language specifications, since the designers cannot expect every desirable abstraction in advance and because the language would be too complex if built-in language mechanisms cover all the desirable abstractions, such as I/O, persistency, distribution, concurrency,

and so on. Programming languages should have a minimum set of built-in mechanisms and most of abstractions should be supplied by libraries.

Motivating problem

Language mechanisms that have been developed so far for writing libraries are not powerful enough to support every desirable control/data abstractions. For example, the C++ language [58] is one of languages that have a richest set of language mechanisms, but a number of useful abstractions cannot be included in a C++ library with satisfying ease-of-use and efficiency. As shown in Chapter 2, C++ programmers cannot write a library class that gives distribution extension to its subclasses.

Our observation on this problem is that such abstractions as distribution support are used to give extended features to another abstraction, and thus their implementations are tightly tangled with the implementation of the other abstraction. Those abstractions need a different implementation if they are used with a different abstraction. This means that those abstractions are difficult to include in a library, which is an independent software component and can provide only a single general-purpose implementation.

There are other kinds of abstractions the implementations of which are tightly tangled with other parts of the program. Abstractions such as a vector data type can be provided by a library but the implementation that the library can supply is not efficient. An implementation specialized for a particular user program can improve the execution performance although that implementation is effective only for the particular program and thus it cannot be included in the library.

This problem can be avoided if programming languages provide a mechanism for library developers to supply specialized code through a library to a particular user program. Current language mechanisms, however, allow only limited kinds of specialization of library code. For example, C++'s template mechanism allows only type parameterization; library developers can specialize only type names appearing in library code for a particular user program.

Solution by this thesis

To solve the problem mentioned above, this thesis proposes a new language mechanism for enabling libraries to supply specialized code and include more useful control/data abstractions. With the proposed mechanism, programmers can build a library consisting of two parts: commonly-used code and

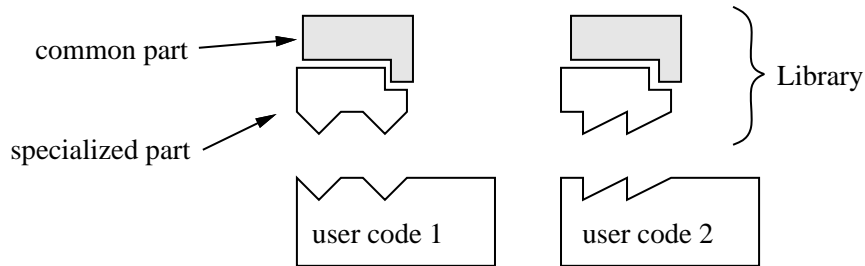


Figure 1.1: The common part and the specialized part of a library

specialized code for every user program (Figure 1.1). The commonly-used code is supplied *as is* to the user program, but the specialized code is automatically generated on demand for a particular user program. The specialized code fills a gap between the common part and the user program.

The generation of the specialized code is *programmed* by the library developer. The proposed mechanism allows to preprocess a program with interacting with a collection of code such as class definitions and member function calls. For example, it provides the ability to insert specialized code, to rewrite class definitions, to substitute different code for member function calls, and so on.

The language mechanism proposed by this thesis is the OpenC++ MOP [11], which is a metaobject protocol for C++. OpenC++ is the name of a version of C++ language with that metaobject protocol. This mechanism provides class metaobjects, which are regular objects representing a class, so that library developers can program the generation of specialized code. Although the class metaobjects might seem similar to Smalltalk's class objects [26], the class metaobjects receive source code at compile time and preprocess it if needed. The programmers can define a new metaclass (i.e., a class for class metaobjects) and thereby they can program desired preprocessing of code involved with the class. The reason that the proposed mechanism is called a metaobject protocol is that it is essentially a protocol for defining and accessing metaobjects. Protocol is the Smalltalk terminology and it means object interface.

The OpenC++ MOP is a more powerful mechanism than other similar mechanisms like Lisp macros. Unlike Lisp macros, the OpenC++ MOP provides contextual information of the processed code other than a abstract syntax tree. The contextual information includes the type of a variable, class members, a base class, and so forth. This feature makes it possible to per-

form context-sensitive preprocessing. Furthermore, implementing non-local processing is easy with the OpenC++ MOP. Preprocessing for implementing abstractions often spreads out over the whole source code, but the description of the preprocessing involved with a single class is centralized into the class metaobject. Since all code fragments are automatically dispatched according to the static types to an appropriate class metaobject, programmers can easily specify preprocessing that is effective only for a particular class.

Preceding techniques known as reflection has a notable influence on the OpenC++ MOP. Especially, we took the basic structure of the metaobject protocol from the CLOS MOP [36], while we took the basic architecture from Lisp macros. Thus the OpenC++ MOP is also metacircular as the CLOS MOP is, and it is easy to learn and to write an efficient meta-level program. The difference from the CLOS MOP is that the OpenC++ MOP employs static typing and executes class metaobjects only at compile time; this means that it does not involve runtime penalties due to class metaobjects. Since CLOS is a dynamically-typed language and thus the CLOS MOP executes metaobjects at runtime, avoiding runtime penalties in the CLOS MOP requires complex implementation techniques.

The structure of this thesis

From the next chapter, we present background, design details, and applications of the OpenC++ MOP. The structure of the rest of this thesis is as follows:

Chapter 2: Limitations of C++

We first discuss limitations of current language mechanisms provided by C++ for library developers. We illustrate that the inheritance mechanism and the template mechanism do not work for including some kinds of desirable abstractions in a library. And we claim that this problem is caused by the lack of the capability of C++ to preprocess a program in a context-sensitive and non-local way and instantiate specialized code for a particular program.

Chapter 3: Techniques for Processing a Program

Next, we overview existing mechanisms that other programming languages provide for processing a program. As the representatives, we show Lisp macros, 3-Lisp, and the CLOS MOP. The feature shared by the three systems is that they provide meta representation of programs for the program-

mers. We overview detailed architectures of the systems and discuss their pros and cons. The bottom line of the discussion is that metacircularity of the CLOS MOP and compile-time executability of Lisp macros are preferable properties.

Chapter 4: The OpenC++ MOP

On the basis of the discussion in the previous chapter, we propose a new C++ mechanism for processing a program. This mechanism is the OpenC++ MOP and it has the two preferable properties discussed in the previous chapter: metacircularity and compile-time executability. The OpenC++ MOP allows programmers to process a program in a context-sensitive and non-local way in order to include useful abstractions in a library that regular C++ cannot handle. We also mention comparison between the OpenC++ MOP and other early compile-time MOPs.

Chapter 5: Meta Helix

Although metacircularity is a good property, pure metacircular systems can lead to a problem we called implementation level conflation. This problem, found in the CLOS MOP, confuses programmers and often causes programming errors such as circular definition. To avoid this problem, we – a MOP designer – has adopted an improved version of metacircular architecture for the OpenC++ MOP. We present that this improved architecture, named the meta helix, preserves advantages of metacircularity and also addresses implementation level conflation.

Chapter 6: Libraries in OpenC++

The OpenC++ MOP makes it possible to include a number of control/data abstractions in a library. We show examples of these abstractions in this chapter. The abstractions shown here include abstractions that regular C++ cannot handle, a metaclass library for helping write a new metaclass, abstractions implemented by meta-metaclass for facilitating meta-level programming, and abstractions that the OpenC++ MOP makes more efficient than in regular C++.

Chapter 7: Conclusion

Finally, we conclude this thesis in Chapter 7. We present contributions of this thesis and future directions.

Chapter 2

Limitations of C++

C++ cannot pre-process a program in a context-sensitive and non-local way. The lack of this capability makes it impossible to include some useful control/data abstractions in a library, or makes it difficult to implement some abstractions efficiently as a library. This problem is solved if C++ has a mechanism to process a user program and allow the library to supply specialized code for a particular user program. Single general-purpose codes cannot implement those abstractions or have difficulty in making the abstractions efficient.

Existing C++ mechanisms for building a library — inheritance or templates — do not provide this capability sufficiently. This means that some practically important abstractions have not been included in a library with an ideal interface and efficiency. In fact, some abstractions have been even provided by a specialized C++ language in which the abstractions are embedded in, or implemented with a dedicated code generator. For example, in the academic world, a number of distributed C++ languages have been developed for making distributed objects available [48, 28]. In industry, programmers who want to use distributed objects have needed to write an extra program in an IDL (interface description language) and combine the code generated from the extra program with their C++ programs.

This chapter presents that C++'s inheritance mechanism or the templates mechanism do not work for including some kinds of abstractions in a library. Then it mentions that, to do this, C++ needs a mechanism for processing a user program in a context-sensitive and non-local way. This mechanism makes it possible to develop a library including the abstractions that regular C++ cannot handle.

2.1 Inheritance

The most basic mechanism C++ provides for library developers is instantiation. The idea is that the library writer writes a reusable class, which the library user can instantiate. This mechanism is limited in that it enables only abstract data types (ADTs); the library user has to use a library class *as is*. Even if the library class does not exactly fit her requirements, she cannot change the definition of the class at all, and thus she might need to write a new class even though a library provides a similar class. This means that the reusability of the library is significantly limited.

Inheritance is a more powerful mechanism for developing a good library. It allows the library user to incrementally define a new class extending a class provided by the library. However, the ability of the inheritance mechanism to reuse part of a library class is significantly limited. The inherited code must be always the same; the library cannot supply different code to different subclasses.

This limitation makes it impossible to develop a library that provides certain kinds of important abstractions such as distributed objects. These abstractions require that the inherited code be *adapted* to the user program. This section discusses this limitation of the inheritance mechanism and begins to outline what kind of support for adaptation of reused code needs to be provided.

Distributed objects

Distributed objects are extended objects, which are accessible over the network without concern for their location. An ideal library for distributed objects might allow the user to write something like this:

```
class Point : public Distribution {
public:
    int x, y;
    void Move(int nx, int ny) { x = nx; y = ny; }
};
```

This program defines a class of distributed objects called `Point`. The class `Point` inherits from the `Distribution` library class, and this is what gives it the distributed class. The user can deal with `Point` objects as regular objects even if they are on a remote machine:

```
Point* p = ...;
p->Move(3, 11);
```

In this ideal library, no special syntax is required to call a member function for a `Point` object `p` that might be on a remote machine.

Unfortunately, this ideal class `Distribution` is not feasible in regular C++ because it cannot supply all the member functions that the subclass `Point` needs to inherit. For example, to make the member function `Move()` callable from a remote machine, the class `Point` needs to inherit a member function for marshaling arguments to `Move()`. This marshaling function converts the arguments into a byte stream, which lower-level network routines can directly handle.

Within the confines of the inheritance mechanism, however, the class `Distribution` cannot supply the marshaling function to the subclass `Point`. Since the marshaling function for `Move()` performs a kind of type conversion, its implementation strongly depends on the signature of `Move()`, such as the number of arguments and their types. But the inheritance mechanism does not allow the class `Distribution` to alter the implementation of the marshaling function to adapt to the signature of `Move()`.

Named objects

Since the example of the distributed objects is too complex to show the details here, we instead use a much simpler example and articulate limitations of the inheritance mechanism and what C++ needs to handle some kinds of abstractions that it cannot currently handle. The problem we showed in the example of the distributed objects is that a super class cannot supply some kinds of member functions to a subclass even though supplying them seems desirable from the library users' viewpoint.

To discuss this problem, let's implement a simple library with only the inheritance mechanism. This simple library allows the users to get the class name of an object at runtime. This function is one of requirements of building the distributed object library. The users may write something like the following program:

```
class Complex : public NamedObject {
public:
    double r, i;
};

void f(Complex* x)
{
    cout << "x is " << x->ClassName();
}
```

If invoked, `f()` displays “`x is Complex`”. The class `NamedObject` is a library class, which supplies a member function `ClassName()` to `Complex`.

Although this particular function is already provided in regular C++ by the `typeid` operator of RTTI [59], we assume that there is no RTTI (Run-Time Type Information) in C++ and try to implement this function with the inheritance mechanism. Unfortunately, this is impossible because of limitations of the inheritance mechanism. For example, the following definition of `NamedObject` works for the class `Complex` but not others:

```
class NamedObject {
public:
    virtual char* ClassName() { return "Complex"; }
};
```

This version of the class `NamedObject` supplies a member function `ClassName()`, which returns a character string “`Complex`”, but this implementation of `ClassName()` is obviously wrong. If another class `Real` inherits from `NamedObject` and the member function `ClassName()` is called for a `Real` object, `ClassName()` returns an inappropriate character string “`Complex`”. To make `ClassName()` work for the class `Real`, the implementation of `ClassName()` should be:

```
virtual char* ClassName() { return "Real"; }
```

However, the class `NamedObject` cannot switch the implementations of `ClassName()` to make it work for different subclasses. It has to select either of the implementations and supply the selected one to all the subclasses.

We cannot implement the ideal version of the class `NamedObject` in regular C++. To make it feasible, we have to change the specifications of the library, but this change also makes the library less easy to use. For the new version of the library, the definition of the class `NamedObject` is as follows:

```
class NamedObject {
public:
    virtual char* ClassName() = 0; // not implemented
};
```

The class `NamedObject` does not supply a concrete implementation of `ClassName()` any more. `ClassName()` is implemented by a subclass of `NamedObject`. This means that the library users have to implement `ClassName()` by hand for their classes:

```

class Complex : public NamedObject {
public:
    double r, i;
    virtual char* ClassName() { return "Complex"; }
};

class Real : public NamedObject {
public:
    double value;
    virtual char* ClassName() { return "Real"; }
};

```

These specifications of the library are quite unsatisfying in terms of ease of use. Note that the definitions of the user classes `Complex` and `Real` now include the implementation of the member function `ClassName()`. In the ideal specifications, this should be supplied by the library class `NamedObject`.

What does C++ need?

The example of `NamedObject` shows us that C++ lacks the ability to allow libraries to supply code customized for the user programs. With the inheritance mechanism, the class `NamedObject` cannot provide an ideal interface because it cannot supply a differently implemented member function `ClassName()` to a subclass such as `Complex` and `Real`.

This problem is solved if C++ provides a mechanism for programmers to *program* source-code processing and include “the program” in a library. For example, the developer of the named object library would write a program to process the user program at source-code level and automatically insert the implementation of `ClassName()` customized for a subclass such as `Complex` and `Real`. Then the library user could avoid implementing `ClassName()` by hand for her class. She could write something like this:

```

class Complex : public NamedObject {
public:
    double r, i;
};

```

Note that the member function `ClassName()` is not included by the definition of the class `Complex` since it is automatically supplied by the library. The named class library reads this program before compilation and translates it into this program:

```

class Complex : public NamedObject {
public:

```

```

    double r, i;
    virtual char* ClassName() { return "Complex"; }
};

```

The implementation of `ClassName()` is inserted by the library. Note that the inserted implementation is specialized for the class `Complex`. It is not effective for other sibling subclasses like `Real`. The library class `NamedObject` declares the member function `ClassName()` but it does not implement it:

```

class NamedObject {
public:
    virtual char* ClassName() = 0; // not implemented
};

```

This is the same definition that we presented for the feasible but unsatisfying version of the named object library.

The macro mechanism might seem an alternative to the mechanism proposed above, but the macro mechanism allows very limited kinds of source-code processing, such as simple word-by-word replacement and concatenation of words. To implement the named object library with ideal interface, C++ is required to have a more powerful and sophisticated mechanism, which should satisfy the following two criteria:

- **Context-sensitive**

How a program is processed should be determined with referring various contextual information of the processed program. The contextual information includes what RTTI (Run-Time Type Information) already provides but not limited. It is a class definition, type information, program text, and so on. Providing contextual information means that the information can be used even if it is defined at other locations than where the source-code processing happens. For example, if an assignment expression to a variable is processed, the type of the variable may be declared at a different location.

Context-sensitivity is a crucial property of the proposed mechanism since the processing by the macro mechanism is independent of the contextual information of the processed program. Context-sensitivity gives an advantage against the macro mechanism to the proposed mechanism.

In the example of the named object library, the library has to examine the name of the subclass that inherits from the library class `NamedObject`. This name is directly embedded in the implementation of the member function `ClassName()`, which are inserted in the definition of the subclass. The

use of contextual information is more significant in the example of the distributed objects. For example, to supply a marshaling function, the library has to examine the signature of a member function and specialize the implementation of the marshaling function for the particular member function.

- **Non-local**

The proposed mechanism should make it easy to program not only local processing but also non-local one, which needs to deal with many code fragments spread out in various locations of the processed program. For ease of programming, locations where the processing must happen should be specified in a declarative way. If the programmer has to explicitly specify every location, the programming would be extremely difficult and unrealistic.

For the example of the named object library, the library has to read the whole program and insert the member function `ClassName()` in the definitions of all the subclasses of `NamedObject`. The mechanism should help the programmer easily specify locations to insert `ClassName()`. For example, it should allow the programmer to direct something like “insert `ClassName()` in a class declaration if the class is a subclass of `NamedObject`.”

C++’s macro functions are rather a mechanism for local processing. It processes only places where a macro name appears, and so programmers have to place the macro name by hand wherever the processing is needed. For example, if the named object library is developed with a macro function, the library users would need to write an awkward program like the following:

```
class Complex : public NamedObject {
public:
    double r, i;
    EXPAND_CLASSNAME_HERE(Complex)
};
```

The macro function is `EXPAND_CLASSNAME_HERE`, which is expanded into an appropriate implementation of `ClassName()`. The library users have to explicitly insert the macro in the definition of all the subclasses of `NamedObject`, but inserting it in all the places by hand is quite error-prone.

2.2 Template

In the previous section, we presented that a library including some kinds of abstractions need to supply customized code for a particular user program. The template mechanism of C++ achieves this ability to a certain degree,

but it cannot be a general solution of our problem. In this section, we mention the ability that the template mechanism provides and then we present why the mechanism is not a general solution.

Vector

With the template mechanism, programmers can write a library that supply customized code for a user program although the range of customization is limited. For example, the template mechanism enables a library for a vector abstraction. With this library, the users can write a program like this:

```
Vector<int> v1, v2, v3, v4;
    ⋮
v1 = v2 + v3 + v4;
```

The variables `v1`, `v2`, `v3`, and `v4` are vectors of integers. Using vectors for other types is also easy. If the users want to deal with vectors of characters, they should say `Vector<char>` instead of `Vector<int>`.

The inheritance mechanism does not enable such a useful vector library. If the vector library is implemented with the inheritance mechanism, the library users need to define a new subclass for every vector for a different type. This results from the same reason why the named object library is not feasible; to make vectors for various types available without subclassing, the vector library has to alter the implementation, depending on whether the vector is for integers or characters.

The template mechanism can absorb the difference of implementations if the difference is type names. It directs the C++ compiler to automatically produce the implementation adapted for a particular user program. The following program is the definition of the vector abstraction, which should be included in the vector library:

```
template <class T> class Vector {
    T elements[SIZE];
public:
    Vector operator + (Vector& a, Vector& b) {
        Vector c;
        for(i = 0; i < SIZE; ++i)
            c.elements[i] = a.elements[i] + b.elements[i];

        return c;
    }
    ⋮
};
```

For simplicity, this example assumes that the length of a vector is always `SIZE`. In the `Vector` template above, `T` is a type parameter. All occurrences of `T` in the template are replaced with an actual type given by a user program when the compiler produces the actual implementation of the vector abstraction. For example, the implementation that the compiler produces for `Vector<int>` is equivalent to this (pseudo) class definition:

```
class Vector<int> {
    int elements[SIZE];
public:
    Vector<int> operator + (Vector<int>& a, Vector<int>& b) {
        Vector<int> c;
        for(i = 0; i < SIZE; ++i)
            c.elements[i] = a.elements[i] + b.elements[i];

        return c;
    }
    :
};
```

Limitations of templates

The vector abstraction can be included by a library with ideal interface if the template mechanism is used, but this implementation of the vector abstraction is not satisfying with respect to execution performance. Consider how this expression is executed if the vector abstraction is implemented as we showed above:

```
v1 = v2 + v3 + v4;
```

Since the `+` operator is overridden for `Vector<int>`, the execution of this expression is divided into two function calls, each of which executes a `for` loop to compute an addition of two vectors. The called function receives two vectors, executes the `for` loop to compute the addition of each vector element, and returns the result.

To compute `v2 + v3 + v4`, this implementation eventually needs two `for` loops from the first through the last element of the vector, but executing this loop twice is redundant. The whole expression should be computed by the following more efficient implementation:

```
for(i = 0; i < SIZE; ++i)
    v1.elements[i] = v2.elements[i] + v3.elements[i]
                    + v4.element[i];
```

This implementation executes the loop only once, and it directly sets the summation of the three elements to `v1`.

This inefficiency is not due to the mechanism of operator overloading. Rather, it should be thought that is caused by limitations of the template mechanism. Operator overloading is a mechanism for syntax sugar, and the problem is not solved even though various kinds of operators can be overloaded. For example, suppose that the programmer can overload the three-operands `+` operator. Then the developer of the vector library would include the following code in the library:

```
template<class T>
Vector<T> operator + (Vector<T>& a, Vector<T>& b, Vector<T>& c)
{
    Vector<T> v;
    for(int i = 0; i < SIZE; ++i)
        v.element[i] = a.element[i] + b.element[i]
            + c.element[i];

    return v;
}
```

This operator function executes such an expression as `v2 + v3 + v4` more efficiently, but the library is still inefficient to deal with other kinds of expressions such as `v2 + v3 - v4` and `v2 + v3 + v4 + v5`. The library developer cannot overload all combinations of operators in advance.

C++ needs a mechanism for processing a program

The template mechanism cannot enable the efficient implementation, which executes a `for` loop only once. This is because of the limitations of the ability of the template mechanism to supply code adapted for a user program. The only adaptation that the template mechanism can perform is to simply fill out parameterized fields in the template with given type names.¹ This is not sufficient to implement the vector abstraction efficiently.

In general, the template mechanism is not suitable for this kind of inter-procedure optimization, which needs to process several operations at a time and substitutes specialized code for all the operations. To do this, C++ needs to be able to handle context of the processed operations: what the group of operations are computing. If C++ has a mechanism to process a program with that context sensitivity, then the user program:

¹A template parameter may be not only type names but also any constant value.

```

Vector<int> v1, v2, v3, v4;
    ⋮
v1 = v2 + v3 + v4;

```

can be translated by the vector library before compilation into:

```

Vector<int> v1, v2, v3, v4;
    ⋮
for(i = 0; i < SIZE; ++i)
    v1.elements[i] = v2.elements[i] + v3.elements[i]
                    + v4.element[i];

```

After the translation, the efficient loop is substituted for the vector expression. The overloaded + operator function is not called any more, but the additions are computed by the inlined loop.

Context-sensitivity and non-locality are also significant for this processing as in the example of the named object library. First, context-sensitivity is needed to determine which expressions should be translated. Since the translation is applied only to vector expressions, the library needs to look up the type of a variable in an expression and determine whether the translation is applied or not. Second, this processing is non local; All vector expressions in the whole program needs to be found and translated into efficient code. Without appropriate supports for programmers to specify places at which the translation should be done, programing the translation for the vector library would be difficult and awkward.

Can optimizing compilers do the same thing?

In the example above, we merged two distinct function calls and inlined the resulting optimized loop. This way of performance improvement is regarded as optimization that regular C++ compilers can perform. As we show in Chapter 6, however, the inter-procedure optimization seems difficult for practical compilers to perform within reasonable time and space. Since inter-procedure optimization requires deep flow-analysis and very clever code generation, we should not expect that compilers perform all possible inter-procedure optimization in general. What we can expect is that compilers may perform the optimization for some typical patterns of program.

Although inter-procedure optimization is difficult for C++ compilers, it is often obvious and straightforward to perform from the programmer's viewpoint. Since the programmers know semantic information of the program, they can easily find possible inter-procedure optimization without

complex flow-analysis. An advantage of our proposed mechanism is that it allows library developers to implement ad-hoc optimization which is obvious to the developers but difficult for compilers to automatically perform. The library developers can include the program for source-code processing in a library so that a user program using the library is pre-processed and efficiently compiled by a back-end compiler.

2.3 Summary

This chapter mentioned that some useful abstractions cannot be included in a library, and others are difficult to efficiently implement within the confines of C++. Such abstractions require different implementations for different library user programs, but existing C++ mechanisms do not provide the ability to do that enough to implement those abstractions.

In this chapter, first, we showed that the inheritance mechanism does not enable developers to implement the distributed object library or the named object library. As for the distributed object library, the developer cannot define a library class `Distribution` so that it supplies subclasses with a marshaling function, which needs to be differently implemented for different subclasses. This is because the inheritance mechanism forces a library class to supply member functions to subclasses *as is* without any adaptation.

Then, we presented that the template mechanism provides the limited ability to supply different implementations for different user programs, but this ability is not powerful enough to implement some abstractions such as the vector abstraction. Because of the ability to supply different implementations, the vector library implemented with the template mechanism allows the users to easily deal with vectors for various types. However, this vector library is not efficient because of the limitations of the ability of the template mechanism. If the library can supply cleverer implementations customized for a particular user program, the provided vector abstractions would be more efficient.

To solve the problem above, this chapter claimed that C++ needs a more powerful mechanism for processing a program at source-code level. If this mechanism is available, programmers can develop a library that pre-processes a user program and inserts code needed for implementing an abstraction in an adapted way to the user program. Through the discussion with the concrete examples, we presented that the proposed mechanism should have two important properties: context-sensitivity and non-locality. First, a user program should be processed in a context-sensitive way. For

example, to produce a marshaling function, the distributed object library needs contextual information of a user program such as signatures of member functions and type information of variables. Second, the proposed mechanism should make it easy for developers to program non-local processing, which deals with code fragments spread out over the user program. For example, the efficient vector library has to find all vector expressions included in a user program and translate them into efficient loops. Without appropriate supports, programming such non-local processing would be difficult and error-prone.

Chapter 3

Techniques for Processing a Program

The previous chapter presented that C++ needs a more sophisticated mechanism for processing a program. The existing mechanisms, the inheritance mechanism or the template mechanism, do not enable context-sensitive or non-local processing. This chapter overviews currently known mechanisms for program processing and mentions their pros and cons. The basic idea shared by these mechanisms is to provide the *meta representation* of the programs. The meta representation gives programmers the capability for context-sensitive and non-local processing, which we need for C++.

As the representatives of these mechanisms, this chapter shows Lisp macros, 3-Lisp, and the CLOS MOP. We compare the meta representation they provide, and discuss pros and cons. The focus of this chapter is on illustrating essential ideas behind the mechanisms rather than showing the exact specifications. Hence the description in this chapter is not exactly faithful to the original syntax or specifications. We carefully alter the syntax and the specifications so as to help clarify the differences among the mechanisms but not to lose the essence of their ideas.

3.1 Lisp Macros

Unlike C++ macros, which perform simple word-based replacement ignoring the syntax, Lisp macros allow the programmers to manipulate program text as data; A program is manipulated through an ordinary data structure. We call this data structure a *meta representation* of the program. In Lisp macros, the meta representation of a program is an abstract syntax tree.

The program text is represented in the form of the tree whose leaf nodes are the lexical tokens of the program text.

Lisp macros enable programmers to implement some kinds of abstractions that Lisp functions cannot implement. Those abstractions are called special forms in the Lisp terminology. For example, the following macro implements a special form `rbegin` that sequentially evaluates expressions from right to left (in the reverse order of `begin`) and returns the resulting value of the leftmost expression:

```
(define rbegin
  (macro exprs
    `(begin ,@(reverse exprs))))
```

A backquote (`'`) and `,@` are convenient notation for constructing a tree structure. If the readers are not familiar to this notation, see Appendix A).

This macro is used as follows:

```
(rbegin (+ 3 4) (list 1 2) (* 5 8))
```

This program is processed by the macro function `rbegin` before being executed. The macro function is an ordinary Lisp function except that it receives and returns program text. The macro argument `exprs` is bound to a list `((+ 3 4) (list 1 2) (* 5 8))`. Then the macro function returns this program:

```
(begin (* 5 8) (list 1 2) (+ 3 4))
```

This is substituted for the original program `(rbegin (+ ...))` before the execution. The resulting value of the program is that of this substituted program, that is, 7.

Note that a Lisp function `reverse` is called during the macro expansion. It reverses the order of the list that `exprs` is bound to. Since the processed program text is the first-class data represented in the tree structure,¹ this Lisp function can process it as it processes ordinary Lisp data. This capability makes Lisp macros different from other simple macros like C++ macros. Indeed, C++ macros cannot implement the special form `rbegin` since the given program text is not the first-class data. They can only perform the limited operations on the program text. Only word-by-word replacement and concatenation are allowed.

¹The first-class data are the data that the program can handle as the object of the computation. For example, numbers, symbols, lists, and vectors, are the first-class data in Lisp.

Applicability to our problem

Lisp macros give a partial solution of the problem we discussed in the previous chapter. They allow programmers to process a piece of code following a macro name. An advantage of Lisp macros is that programmers can *program* how the piece of code is expanded. This provides the ability to generate more specialized code than what the template mechanism can do by simple replacement of type names. On the other hand, Lisp macros cannot handle context-sensitivity or non-locality and thus just porting the Lisp macro system to C++ does not solve our problem.

3.2 3-Lisp

Lisp macros provide the meta representation of programs so that they can implement some kinds of abstractions that Lisp functions cannot implement. Although the meta representation in Lisp macros is only program text, 3-Lisp [54, 53] provides not only program text but also other information as the meta representation. This feature of 3-Lisp enhances the variety of abstractions that programmers can implement.

Meta representation

The meta representation of programs in 3-Lisp consists of program text, the current environment, and the current continuation. Adding the latter two makes it possible to implement abstractions that Lisp macros cannot handle. Suppose that we implement a special form `defined?`, which returns true if the given variable is defined in the current environment:

```
> (define x 1)
x
> (defined? x)
#t
> (defined? y)
#f
```

The special form `defined?` returns `#t` (true) for the variable `x`, but `#f` (false) for the variable `y`.

Because Lisp macros cannot examine the current environment, they cannot deal with this special form. However, 3-Lisp can do. See the following program written in 3-Lisp:²

²To make it easy to read, we modify the syntax of 3-Lisp. We believe that this modification does not affect the essential idea of 3-Lisp but rather helps articulate it.

```
(define defined?
  (meta (expr env cont)
        (if (is-bound? env (car expr))
            (cont #t)
            (cont #f))))
```

The special form `defined?` is implemented by a meta function (originally called *lambda reflect* in 3-Lisp). Unlike macro functions, which receive only program text, the meta function receives the current environment `env` and the current continuation `cont` as well as the program text `expr`. The current environment represents the bindings between symbol names and values, while the current continuation represents the control flow after this special form finishes. Note that the meta function may access and change the environment and the continuation since they are the first-class data within the meta function. In fact, to implement `defined?`, the meta function calls a built-in function `is-bound?` and looks up a symbol name obtained by `(car expr)` in the received environment `env`. Then it calls the received continuation `cont` with `#t` or `#f`, which is the resulting value of the special form.³

Base level and meta level

Meta functions are not extended macro functions that receive more arguments. There are significant difference between macro functions and meta functions. Meta functions run at runtime because they manipulate the current environment and the current continuation, which are only available at runtime. Hence a meta function directly interpret the received program text. The value returned by the meta function is the result of the interpretation. The meta function also may cause side effects on the environment as the result of the interpretation. On the other hand, macro functions may run at either runtime or compile time since they deal with only static information of the program, that is, program text. They are functions that receive program text and transform it, but they do not directly interpret it. The value returned by a macro function is the transformed text.

In 3-Lisp, both ordinary functions and meta functions run together at runtime. To distinguish the two kinds of functions, 3-Lisp has two execution levels, which are the base level and the meta level. The two execution levels

³In reality, the base-level value has different representation at the meta level. For example, `#t` (a boolean) becomes `'#t` (a symbol) at the meta level. Therefore, the meta function must convert `#t` into the meta representation before passing it to the continuation `cont`.

are identical except that the objects of the computation at the meta level is the interpretation of the base-level program. The result of the computation at the meta level *reflects* on the computation at the base level. This relation between the meta level and the base level is called *causal connectivity*. For example, meta functions may change the environment at the base level in order to define a new symbol name, or to change the value that a symbol name is bound to.

Meta functions can use a built-in meta function if they interpret the received program text in the default way. For example, the program below is another implementation of the special form `rbegin` with a meta function:

```
(define rbegin
  (meta (expr env cont)
        (let loop ((rest expr)
                   (c cont))
          (if (null? rest)
              (c '???)
              (loop (cdr rest)
                    (lambda (r)
                      (eval-expr env ; environment
                               (car rest) ; sub-expression
                               c))) ; continuation
```

Note that a built-in meta function `eval-expr` (`normalize` in the 3-Lisp terminology) is called to interpret each sub-expression of `rbegin`. According to the given sub-expression, `eval-expr` causes side effects on the environment and calls the continuation with the resulting value of the sub-expression.

Self modification

In the examples shown so far, we have introduced new keywords such as `rbegin` and `defined?`, and defined their meanings to implement new abstractions. Some kinds of abstractions, however, require altering the meanings of the existing keywords or syntax. In other words, they need to modify the default behavior of the language rather than to extend it. A feature of this *self modification* is that the effects of the modification are applied to programs even though the programs are not edited. Editing them or inserting new keywords are not necessary.

Some followers of 3-Lisp, such as Black [2], enables the self modification. They provide many built-in meta functions, which carry out a primitive base-level operation, and allow programmers to redefine them to change the default behavior of the operations. For example, they may alter the behavior of the language when reading a variable. By default, reading an undefined

variable causes an error. We alter this behavior so that `'undef` is returned if the variable is undefined:

```
(define change-eval-read
  (meta (expr env cont)
    (set! eval-read
      (lambda (expr env cont)
        (let ((var (car expr)))
          (if (is-bound? env var)
              (cont (lookup env var))
              (cont 'undef))))))
  (cont #t)))
```

If the meta function `change-eval-read` is executed, it calls `set!` to substitute a new meta function for the built-in meta function `eval-read`. Note that `set!` has to be invoked at the meta level. Otherwise, `set!` would replace a base-level function `eval-read`, if any, with the new one. The substituted meta function first checks whether the given variable is defined or not in the current environment. If not, it uses `'undef` for the value of the variable:

```
> x
ERROR: undefined symbol
> (change-eval-read)
#t
> x
UNDEF
```

This sort of self modification is difficult for Lisp macros. Some implementations of the Lisp macro system allow a macro function to override an existing keyword. For example, programmers may define a macro function named `if` to alter the behavior of the `if` special form. However, Lisp macros cannot handle such modification that we showed in the `change-eval-read` example because the expression for reading a variable is not preceded by any keyword. To process an expression by a Lisp macro, programmers have to explicitly place the macro name in front of the processed expression.

To use a Lisp macro and get the same result that `change-eval-read` provides, programmers have to use a programming convention when reading a variable. For example, they may have to write `(read-variable x)` instead of just writing `x`, whenever reading a variable `x`. This programming convention makes it possible for a macro `read-variable` to process the expression `x`. Another approach is to surround the whole program by a macro call. This technique is called the code walker. The macro function

receives the whole program as the argument, looks up expressions for reading variables, and translates the expressions to alter the behavior. Writing the code walker is not difficult in Lisp because the grammar is simple, but it is difficult in other languages such as C++.

Reflective languages

3-Lisp is one of the earliest languages that can handle richer meta representation of the programs than Lisp macros.⁴ 3-Lisp has been followed by many languages, and this family of languages are often called *reflective languages* or languages with a meta architecture. For example, Brown [60], Blond [18], and Black [2] are Lisp-based successors of 3-Lisp. These successors has been developed to study the semantics and the implementation of the infinite tower of the execution levels, that is, the base level, the meta level, the meta-meta level, and so on. This was also one of the main issues of the study of 3-Lisp.

The meta architecture developed by 3-Lisp has been also applied to object-oriented languages. Early representatives of such languages are CLOS [56, 36] and 3-KRS [38]. The two languages have been developed under different design strategies. Since we discuss CLOS in the next section, we introduce only 3-KRS here.

3-KRS is designed mainly for customizing the default behavior of the language on demands, rather than implementing new abstractions on top of the language. In 3-KRS, each base-level entity such as objects and messages is associated with a special object called *metaobject*. The metaobject is the meta representation of the base-level entity. Calling a method for the metaobject, programmers can obtain the meta information of the entity. For example, they can inspect what methods an object has through the metaobject for the object. Also, the metaobject provide similar capability of a built-in meta function in 3-Lisp. For example, the metaobject for an object has a method for invoking a method on the object.

As in 3-Lisp, programmers can alter the default behavior of 3-KRS. Instead of redefining built-in meta functions, the programmers define a new metaobject to implement the new behavior. Suppose that they alter the default behavior of the object creation so that the history of object creation is recorded. With this language customization, programmers may write something like this:

```
> *history*
```

⁴In Artificial Intelligence, a similar idea was proposed earlier than 3-Lisp.[3, 25, 62]

```

()
> (defclass point ()
  (variable x y) :meta recorded-object)
POINT
> (defclass rect ()
  (variable top bottom left right) :meta recorded-object)
RECT
> (define p (make-instance point))
P
> (define q (make-instance rect :meta recorded-object))
Q
> *history*
(RECT POINT)

```

If a class annotated with “:meta recorded-object” is instantiated, the class name of the instance is added to the list indicated by `*history*`. For the example above, since a `point` object and a `rect` object are created, the class `point` and the class `rect` are added to the list.

To implement this language customization, the programmer first defines a new class `recorded-object` for metaobjects:⁵

```

(defclass recorded-object (metaobject-for-object)
  (defmethod create-object (expr env)
    (let ((class-name (car expr))
          (h (env-ref env '*history*)))
      (env-set! env '*history* (cons class-name h))
      (<- super create-object expr env))))

```

The class `recorded-object` inherits from the default class `metaobject-for-object` and overrides a method `create-object` to maintain `*history*`. The method `create-object` is invoked when an object is created. The new `create-object` first adds the class name to the list indicated by the base-level variable `*history*`. Then it calls the overridden method of the super class `metaobject-for-object`, which creates a new object in the default way. Note that `create-object` has to call built-in meta functions `env-ref` and `env-set!` to access the base-level variable `*history*`. It cannot directly access this variable since `create-object` is at the meta level. Any base-level entity must be dealt with through built-in meta functions.

Other object-oriented reflective languages include ABCL/R [61, 41, 40], Ferber’s language [21], RbCl [32], The MIP for C++ [9], OpenC++ version 1 [13], AL-1/D [46, 45], CodA [42], a reflective version of BETA [6], and Iguana [29, 30]. They have explored various applications of the meta

⁵Again, we use an altered version of 3-KRS for clarifying our argument.

architecture. ABCL/R is a parallel language, and it allows programmers to customize the default scheduling policy. OpenC++ version 1 enables programmers to implement language extensions such as distribution, persistence [57], and fault-tolerance [20], within the confines of the language. AL-1/D's application is similar to ABCL/R. It allows programmers to customize the policy of object migration. Coda employs the meta architecture to run the same Smalltalk program on different platforms. For example, if the programmer ports a program written for a single processor machine to run on a multi-processor machine, she has only to define new metaobjects. Since the platform-dependent code is separated into the metaobjects, the programmer does not need to edit the base-level program.

Applicability to our problem

Reflective languages represented by 3-Lisp show us how meta representation of a program should be exposed to the programmer. Especially, object-oriented reflective languages such as 3-KRS shows that metaobjects can be good abstraction to deal with complexity of the meta representation.

However, reflection cannot be a solution of our problem since programmers describe interpretation of a base-level program to define a new mechanism or change an existing mechanism. Although programmers can describe interpretation that causes the same effects that we want, describing source-code processing is more intuitive and straightforward as for our motivating applications. Also, runtime penalties implied by the interpretation is another problem of reflection.

3.3 The CLOS MOP

We claim that the reflective languages introduced in the previous section, such as 3-Lisp and 3-KRS, should be called non-metacircular, or weakly-metacircular, languages. They are metacircular in a certain sense because base-level and meta-level programs are written in the same syntax, but the two "languages" for the base-level and meta-level programs are not identical.

This section discusses (truly) metacircular reflective languages represented by the CLOS MOP. Unlike 3-Lisp or 3-KRS, its base level and meta level programs are written in an identical language. This means that the customization by the meta-level program affects not only the base-level language but also the meta-level language in which the meta-level program is written. This feature gives some benefits to reflective languages.

Metacircularity

Metaobject protocols (MOPs) are another name to indicate a meta architecture such as 3-Lisp's one. Particularly, MOPs mean programming interfaces for customizing the language. The word "metaobject protocol" has been first used in CLOS (Common Lisp Object System) [56]. The CLOS MOP [36] enables programmers to incrementally customize CLOS in CLOS itself with a meta architecture.

A unique feature of the CLOS MOP is that the system is metacircular (Figure 3.1). In 3-Lisp, meta functions and base-level functions are written in the same language, but the two "languages" for the meta and base-level functions may not be identical; they may be two distinct instances of the same language. At beginning, the two languages are identical and are thus meta-circular in some sense. But once the base language is customized, the meta language is left unchanged and the two languages are therefore different.

This fact would be clear if we reason about the object of language customization; the language customized by a meta-level program is just the language for a base-level program. For example, a new special form defined by a meta function is only available for base-level functions but not available for other meta functions. The customization by meta functions reflects only on the language for base-level functions, but it does not circularly reflect on the language for the meta functions.

The reason for this non-metacircularity would be to avoid apparent infinite regression caused by a circular definition. If 3-Lisp is naively made metacircular, programmers would easily define a meta function with using the special form defined by the meta function itself and the special form would cause infinite regression. But this problem can be avoided with keeping metacircularity if another scope control mechanism is introduced.

To keep metacircularity while letting programmers avoid a circular definition, the CLOS MOP uses a class system for controlling the scope of language customization. Hence both base-level programs and meta-level programs are written in the same instance of the language, and language customization by a meta-level program is also applied to the language in that the meta-level program is written. The relation is circular between the semantics of the customized language and the meta-level program customizing the language.

Despite of the metacircularity, CLOS programmers can avoid a wrong circular definition.⁶ In the CLOS MOP, customization is specified on a class

⁶The solution by the CLOS MOP is not a complete solution. We revisit this issue in

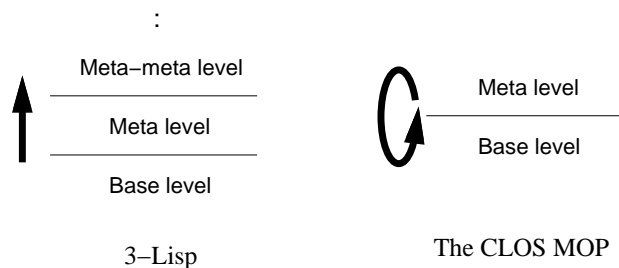


Figure 3.1: The CLOS MOP is metacircular

basis; it is applied only to particular classes and their instances. Thus, if programmers carefully distinguish classes, they can avoid a circular definition. Suppose that a meta-level program alters the behavior of a class `Point`. Since the CLOS MOP is metacircular, the programmer may use the customized class `Point` in the meta-level program, but she can still avoid a circular definition unless she explicitly implements the customization with `Point` to be a circular definition.

The origin of the metacircular architecture would be found in Smalltalk-80 [26]. In fact, Foote reported the use of the metacircularity in Smalltalk-80 in his paper [22]. Also, CommonLoops [4] should be noted as an early metacircular language. Furthermore, the metacircular architecture is found in ObjVlisp [17], Classtalk [8], EuLisp [7] and STklos [24].

Direct access to the base level

The CLOS MOP provides the capability for customizing the behavior and implementation of the object system of Common Lisp (i.e., CLOS). For example, programmers may define a new metaobject for classes to alter the rule of multiple inheritance. This customization makes it easy that CLOS runs programs written in other Lisp-based languages such as Loops [5] and Flavors [10].

Despite the metacircularity, the way of customization with the CLOS MOP is quite similar to the way of customization in non-metacircular languages such as 3-KRS. In the previous section, we showed the 3-KRS implementation of the language customization for recording object creation. In the CLOS MOP, this customization is implemented by this program:⁷

Chapter 5.

⁷Again, we use an altered syntax for emphasizing difference from the 3-KRS imple-

```
(defclass recorded-class (metaobject-for-class)
  (defmethod create-object (init-args)
    (let ((class-name (<- self name)))
      (set! *history* (cons class-name *history*))
      (<- super create-object init-args))))
```

Since the CLOS MOP does not provide the metaobject for an object, the program defines a new metaclass, which means a new class for class metaobjects. It inherits from the default class `metaobject-for-class` and overrides a method `create-object`. When an object is created, the new `create-object` first calls a method `name` for the metaobject (`self`) to obtain the class name. Then it updates the history of object creation and calls the overridden method for the super class, which creates an object in the default means.

Because of metacircularity, there is no explicit (syntactical) distinction between base-level programs and meta-level programs in the CLOS MOP. The two kinds of programs are written in the same language and run under the same runtime environment. They may even coexist in a single function or method. Therefore, metaobjects in the CLOS MOP can directly access the base-level data. The metaobjects can use the base-level primitives to access the base-level data as the objects can do. There is no difference in primitives that the metaobjects and the objects can use. For example, the method `create-object` directly reads and writes the variable `*history*` with base-level primitives like `set!`. Recall that `create-object` in 3-KRS has to access through built-in meta functions `env-ref` and `env-set!` since the base-level objects and the metaobjects are run at distinct execution levels.

Ease of learning

Base and meta levels in non-metacircular systems are written in different languages. This separation makes access across levels be complex and tend to be inefficient. On the other hand, the metacircularity of the CLOS MOP avoids gratuitous differences between levels, while still being effective, and gives a few advantages.

First, the meta architecture employing metacircularity is easy to learn. Since programmers can use base-level primitives to access base-level data from the meta level, they do not have to learn built-in meta functions unless they need to access meta-level data, which there is no base-level primitive to access.

mentation.

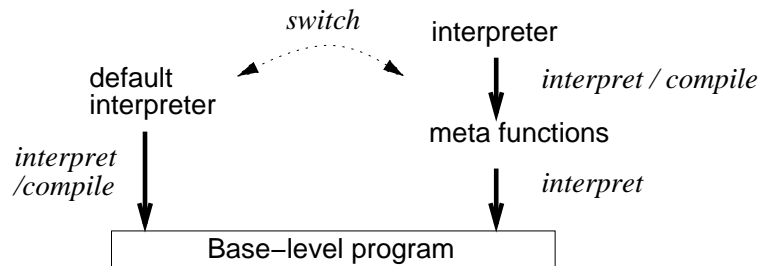


Figure 3.2: An improved implementation of 3-Lisp

For example, a metaobject in the CLOS MOP can call a method for an object in the same syntax that an object calls it at the base level. The metaobject uses a built-in meta function only when it needs to access a meta representation of the program, for example, the class name of an object. Recall that `create-object` of `recorded-class` executes `(<- self name)` to obtain the class name. On the other hand, a metaobject in 3-KRS has to call a built-in meta function even when it calls a method for an object. In 3-KRS, any object must be indirectly manipulated through the metaobject associated with that object. To call a method for the object, an appropriate built-in meta function for the metaobject has to be called.

Ease of implementing efficiently

Another advantage of metacircularity is execution performance. Metacircular systems make it intuitive and straightforward for implementors to develop an efficient interpreter and compiler. To do the same thing in non-metacircular systems like 3-Lisp, they need sophisticated implementation techniques or they need to force programmers to use advanced programming techniques.

If naively implemented, 3-Lisp is extremely slow. The typical implementation of 3-Lisp uses two interpreters for keeping two distinct execution levels. The first interpreter executes the second interpreter, while the second interpreter executes base-level functions. In this implementation, meta functions are regarded as part of the program of the second interpreter. They are executed by the first interpreter and execute base-level functions. This double interpretation maintains the causal connectivity between the base level and the meta level, but it significantly decreases the execution speed of the base-level program even though no meta functions are used.

Improving the execution performance is relatively easy if the base-level program does not use meta functions. We can prepare the third interpreter, named the default interpreter, and switch to it while the base-level program runs without meta functions (Figure 3.2). Note that the first interpreter and the default interpreter are not identical; at least, they are distinct instances of an interpreter and they maintain different runtime environments. Because the default interpreter does not handle meta functions, it can achieve as good efficiency as an optimized interpreter for ordinary Lisp. Or, we may use a compiler for better efficiency instead of an interpreter.

To use the default interpreter from a meta function, programmers need to call a built-in meta function like `eval-expr` and explicitly switch to it. Recall the implementation of `create-object` in 3-KRS. It can be rewritten to explicitly switch to the default interpreter:

```
(defmethod create-object (expr env)
  (let* ((class-name (car expr))
        (expr2 '(set! *history*
                      (cons ,class-name *history*))))
    (eval-expr env expr2)
    (<- super create-object expr env)))
```

The built-in meta function `eval-expr` executes the expression `expr2` with the environment `env` by the default interpreter, so that the expression is executed faster. For example, since the expression accesses the base-level variable `*history*` twice, the default interpreter may memoize the memory location of the variable and reuse it for the second access. However, the optimization that the default interpreter can do for `eval-expr` is limited to runtime optimization since `eval-expr` receives an expression and an environment at runtime. Even if a compiler is used instead of the interpreter, no static optimization is applicable.

On the other hand, metacircular systems naturally give more freedom to the default interpreter or compiler. For example, in the CLOS MOP, there is no explicit distinction between the base level and the meta level, so the default interpreter can execute both base-level and meta-level programs (Figure 3.3). This means that programs implicitly switch direct interpretation by the default interpreter and double interpretation via metaobjects. Recall the implementation of `create-object` in the CLOS MOP:

```
(defmethod create-object (init-args)
  (let ((class-name (<- self name)))
    (set! *history* (cons class-name *history*)))
    (<- super create-object init-args)))
```

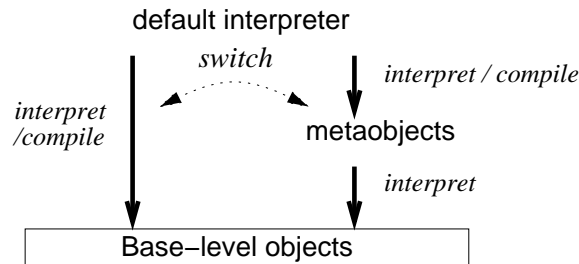


Figure 3.3: An metacircular implementation

This method directly accesses a base-level variable `*history*` with the base-level primitive `set!`. Since the expression `(set! ...)` is written in the syntax for the base-level program, it can be directly executed by the default interpreter, and hence the system implicitly switches to direct execution by the default interpreter.

This implicit switching gives a lot of room for optimization especially to a compiler. Unlike the way by using `eval-expr`, an executed expression is statically given to the default interpreter and hence various optimization techniques using static information are naturally applicable.

To get equivalent effects in non-metacircular systems, a sophisticated optimizing compiler or complex programming techniques are needed. For example, an optimizing compiler using a technique called partial evaluation [19, 23] or inlining will compile the following program in 3-KRS:

```
(defmethod create-object (expr env)
  (let* ((class-name (car expr))
        (expr2 '(set! *history*
                      (cons ,class-name *history*))))
    (eval-expr env expr2)
    (<- super create-object expr env)))
```

into as efficient code as the equivalent program in the CLOS MOP. The compiler will statically determine the values of `env` and `expr2` as much as possible, and specialize `eval-expr` to be more efficient. This approach has been studied by a few researchers including the author [52, 14, 39], but implementing this technique is very difficult in practical languages such as C++; for example, no compiler using partial evaluation has been developed for C++ yet.

Another approach is to provide more built-in meta functions for programmers to be able to optimize a meta function by hand:

```
(defmethod create-object (expr env)
  (let ((compiled-code (lookup-hash expr)))
    (if (null? compiled-code)
        (let* ((class-name (car expr))
              (expr2 '(begin
                       (set! *history*
                             (cons ,class-name *history*))
                       ,expr)))
          (set! compiled-code (compile-expr expr2)))
        (record-hash expr compiled-code)))
    (execute env compiled-code)))
```

The underlined functions are built-in meta functions. The above implementation explicitly compiles an expression and memoizes it when it is first executed, and reuses the compiled code from the second time. Although this implementation will run a bit faster, the built-in meta functions for optimization make the language complicated and difficult to use. Programmers have to learn how the built-in meta functions help for optimization.

When metaobjects run

Normally, metaobjects in metacircular systems run at runtime as in non-metacircular systems. Metacircularity has nothing to do with when metaobjects run at runtime or compile time. However, because running metaobject at runtime impairs execution performance, the actual CLOS MOP employs a technique called *currying*⁸ so that the metaobjects less frequently run at runtime. This technique is based on the observation that some of computation by metaobjects can be statically performed, or, once it is done, the result can be memoized and it does not need to be performed again. So the currying technique explicitly splits the protocol into one for computation that has to be done at runtime and the rest, and let the underlying interpreter improve performance by avoiding redundant execution of that computation.

For example, if the currying technique is not adopted, a method `call-method` for class metaobjects is implemented as something like this (Note: this is pseudo code):

⁸This technique should not be called the memoization technique. It is a technique for protocol designers, who perform currying on documented functions so that protocol implementors can perform memoization.

```

(defmethod call-method (object method-name args)
  (let ((method (<- self lookup method-name)))
    (apply method (cons object args))))

(defmethod lookup (method-name)
  (let ((entry (assq method-name
                    (<- self direct-methods))))
    (if entry
        (cdr entry)
        (let loop ((supers (<- self super-classes)))
          (if (null? supers)
              (error "invalid method name")
              (or (<- (car supers) lookup method-name)
                  (loop (cdr supers))))))))))

```

The method `call-method` is a built-in meta function for calling the method specified by `method-name` for the object. The implementation is divided into two parts. It first calls a helper method `lookup`, which finds the function body of the targeted method. Then `call-method` actually invokes the found method. `lookup` first checks methods directly supplied by the class, and, if not found, it searches super classes.

This implementation is not efficient because the targeted method is looked up in the class hierarchy for every call. It should be looked up only once when it is first called, and the found method should be memoized and reused for all subsequent calls to the method.

To avoid this inefficiency, the actual protocol has been designed with the currying technique. In the actual design, the computation by `call-method` shown above is split into two parts. The redesigned `call-method` receives only `method-name`, looks up the targeted method, and returns a function:

```

(defmethod call-method (method-name)
  (let ((method (<- self lookup method-name)))
    (lambda (object args)
      (apply method (cons object args)))))

```

If called, the function returned by `call-method` receives `object` and `args` and invokes the targeted method with them. The targeted method is not invoked until the returned function is called. Note that the targeted method is looked up only once when `call-method` runs. It is never looked up when the returned function runs.

The CLOS interpreter and compiler can employ this curried protocol to improve execution performance. It should call `call-method` in advance at load time or at compile time, and memoize the returned function with the method name. Then, if the method is actually called at runtime, it can

directly call the memoized function for invoking the method. No look up is needed at runtime. The targeted method is looked up only once at load time or at compile time.

Applicability to our problem

The CLOS MOP shows that there is another design of the meta system of a reflective language. Because of its metacircularity, a meta-level program is not an interpreter of the base-level program but can be regarded as a collection of base-level code substituted for parts of the original base-level program. Recall that a meta-level program describes base-level behavior with the base-level primitives instead of built-in meta functions.

This fact means that metacircular reflection can be the basis of a solution of our problem. In the previous chapter, we presented that C++ needs a mechanism for processing source code with context sensitivity and non locality. Metacircular reflection allows programmers to specify source-code substitution with context sensitivity and non locality.

However, we cannot use metacircular reflection represented by the CLOS MOP *as is* to solve our problem. It is in principle a runtime system, which may involve overheads, and is difficult to handle optimization such as the example of the vector library in the previous chapter. It is not suitable for dealing with adjacent but independent operations at a time.

3.4 Summary

This chapter illustrated currently known mechanisms for processing a program. We showed the tree representative mechanisms, Lisp macros, 3-Lisp, and the CLOS MOP. As an object-oriented version of 3-Lisp, we also showed 3-KRS.

Table 3.1 summarizes the features of the three mechanisms. All of them provide meta representations of the base-level programs for the programmers. The meta representations enable the programmers to process the programs as computable data. As the meta representations, Lisp macros provide program text in the form of the abstract syntax tree. 3-Lisp provides not only program text but also the current environment and the current continuation. The CLOS MOP provides classes, generic functions, methods, and so on.

Although providing meta representations allows to implement a new kind of abstraction that are not available within the confines of the original language, some kinds of abstractions also need to alter the default behavior

Table 3.1: Lisp Macros, 3-Lisp, and the CLOS MOP

	<i>Lisp Macros</i>	<i>3-Lisp</i>	<i>CLOS MOP</i>
Meta representation	\triangle	\circ	\circ
Self modification	\times	\circ	\circ
Metacircular	\circ	\times	\circ
When running	CT or RT	RT	(CT and) RT

RT: runtime, CT: compile time

of the language. This self modification of the language is allowed only by 3-Lisp and the CLOS MOP. Lisp macros cannot change the behavior of the language when reading a variable, for example. 3-Lisp and the CLOS MOP, therefore, enable a larger number of kinds of abstractions than Lisp macros.

Metacircularity is a good property because metacircular systems are easy to learn and easy to implement efficiently. Only Lisp macros and the CLOS MOP have this property. 3-Lisp is not metacircular; although the base level and the meta level uses the same language, the customization by meta functions is not applied to the language for the meta level. The customization is applied only to the language for the base level.

Furthermore, macro functions may run at either runtime or compile time, whereas meta functions in 3-Lisp and metaobjects in the CLOS MOP run at runtime. This means that Lisp macros have an advantage in terms of execution performance since macro functions run at compile time. 3-Lisp has to process a program at runtime even though the processing can be done at compile time. To avoid the performance problem of 3-Lisp, the CLOS MOP employs the currying technique and performs most of meta computation at the load time or at compile time.

Through this chapter, we have discussed that a meta architecture with metacircularity, such as the CLOS MOP, has great benefits for processing a program. But this architecture has a drawback with respect to execution performance because metaobjects normally run at runtime unless an elaborate technique like currying is used. Lisp macros have an advantage for the performance issue; macro functions can run at compile time and they can involve no performance penalties at runtime. This feature of Lisp macros is significant to develop a mechanism for processing a program in C++ since C++ programmers are particular about execution performance.

In the next chapter, we propose a new C++ mechanism with both advantages of the CLOS MOP and Lisp macros. It enables source-code processing in a context-sensitive and non-local way so that programmers can develop better libraries in C++.

Chapter 4

The OpenC++ MOP

This chapter presents a new C++ mechanism for processing a program. The proposed mechanism enables context-sensitive and non-local processing, which is not supported by existing C++ mechanisms, such as the inheritance mechanism and the template mechanism. Because of those two features, this mechanism makes it possible to include some kinds of useful abstractions in a library and implement other abstractions efficiently. In regular C++, those abstractions are impossible to include in a library or, otherwise, they are difficult to implement efficiently.

The proposed mechanism is called the OpenC++ MOP (Metaobject Protocol). OpenC++ is an enhanced version of the C++ language for this mechanism.¹ The OpenC++ MOP has been developed by a synthesis of ideas of the techniques illustrated in the previous chapter. Especially, we took the basic protocol structure from the CLOS MOP [36], and we took the basic architecture from Lisp macros. The OpenC++ MOP is also influenced by Intrigue [37], Anibus [50, 51], and MPC++ [33, 34].

4.1 Overview

The OpenC++ compiler is fed with two kinds of code. One is ordinary source code and the other is meta code that specifies how the source code is processed. Both of them are written in OpenC++. The compiler first runs the C++ preprocessor, and then performs source-to-source translation from OpenC++ to regular C++. This translation is specified by the meta code. The translated code is passed to the back-end C++ compiler and processed into executable code.

¹To distinguish OpenC++ version 1, this language is called OpenC++ version 2.

Introductory Example

The OpenC++ MOP is a protocol that the meta code uses for specifying source-code processing. The protocol structure of the OpenC++ MOP is based on that of the CLOS MOP. Programmers define a new class metaobject to specify a new kind of source-code processing. For example, we show the implementation of the language extension that records object creation. This is the example that we repeatedly used in the previous chapter:

```
class RecordedClass : public Class {
public:
    Ptree* TranslateNew(Environment* env, Ptree* header,
                       Ptree* new_op, Ptree* placement,
                       Ptree* type_name, Ptree* arglist)
    {
        return Ptree::Make("(history[n++]=\"%p\",%p)",
                           type_name,
                           Class::TranslateNew(env, header, new_op,
                                                placement, type_name,
                                                arglist));
    }
};
```

We define a new metaclass, which is a class for class metaobjects. The new metaclass `RecordedClass` corresponds to `recorded-class` that the previous chapter showed while mentioning the CLOS MOP. It inherits from the default metaclass `Class` and overrides a member function² `TranslateNew()` as the class `recorded-class` inherits from `metaobject-for-class` and overrides `create-object`.

Although `create-object` in the CLOS MOP directly interprets the program and executes the object creation, `TranslateNew()` in the OpenC++ MOP just translates the program at the source-code level as Lisp macros do. It receives program text and returns the resulting text of the translation.

To use this language extension, programmers write something like this:

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

void f(){
    . . .
```

²A member function means a method in the C++ terminology. Similarly, in C++, an instance variable is called a data member, and a super class is called a base class.

```

    Point* p = new Point;
    . . .
}

```

The first line beginning with `metaclass` is a metaclass declaration. It declares that the class metaobject for `Point` is an instance of `RecordedClass`. Thereby, the program above is translated by the class metaobject into this program:

```

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

void f(){
    . . .
    Point* p = (history[n++]="Point",new Point);
    . . .
}

```

Note that the metaclass declaration is eliminated and the `new` expression “`new Point`” is replaced with “`(history[n++]=. . .)`”. If the function `f()` is executed, a character string “`Point`” is stored in an array `history`. The variable `n` specifies the number of the recorded class names.

What Class Metaobjects do

The translation shown above is performed by `TranslateNew()`. It receives the program text for the `new` expression and the environment, and returns the translated `new` expression. The environment represents the bindings between names and their static types. Unlike the environment in 3-Lisp, it does not represent the dynamic bindings between names and their runtime values. The received program text is represented in the form of the parse tree as in Lisp macros and 3-Lisp. `Ptree` is the type for the parse tree. For convenience, `TranslateNew()` does not receive the `new` expression as a single tree. The tree is divided into several subtrees before passed to `TranslateNew()`. For example, the fifth argument `type_name` is bound to the type-name field of the `new` expression, that is, `Point`.

`TranslateNew()` constructs a new parse tree that is substituted for the original `new` expression. To do this, `TranslateNew()` calls a built-in function `Ptree::Make()`. Here we show the function body of `TranslateNew()` again:

```

return Ptree::Make("(history[n++]=\"%p\\\",%p)",
                  type_name,

```

```

Class::TranslateNew(env, header, new_op,
                   placement, type_name,
                   arglist));

```

The expression `Class::TranslateNew(env, ...)` calls the overridden member function of the base class `Class`. Since `Class` is the default metaclass, this member function call returns the `new` expression without any change. The function `Ptree::Make()` constructs a parse tree according to the format given as the first argument. It substitutes given subtrees for the occurrences of `%p` in the format. For example, the first occurrence of `%p` surrounded by double quotes is replaced with the subtree indicated by `type_name`.

A class metaobject handles all kinds of expressions involved with the class. They can handle class declaration, the `new` expression, member function calls, and even data member accesses. The default metaclass `Class` has member functions for translating each kind of expression. Although these member functions of `Class` do not transform the received program text, programmers can define a new metaclass to override them and customize the translation. Note that a class metaobject handles only the expressions involved with the class. Hence programmers can restrict the customized translation to only some classes. Even though a new metaclass is defined, expressions are not translated unless the metaclass is specified for the class that the expressions are involved with.

4.2 Context-sensitive and Non-local

The OpenC++ MOP enables context-sensitive and non-local processing of programs, which the inheritance mechanism or the template mechanism cannot do. This feature of the OpenC++ MOP makes it possible for programmers to write libraries that they cannot do in regular C++.

Context Sensitive

The OpenC++ MOP provides rich meta representation of programs so that programmers can refer to various contextual information of the programs during the processing. For example, the programmers can refer to the program text represented in a parse tree, the static type environment, and class definitions. The OpenC++ MOP provides different metaobjects for each kind of information. The programmers use these metaobjects and determine how to process the programs.

We below present brief overviews of all the kinds of metaobjects: `Ptree`, `Environment`, `TypeInfo`, and `Class`. The detailed specifications are shown

in Appendix B. These metaobjects represent program text, static types, type definitions, class definitions, respectively. They cover information needed to determine the semantics of a given code fragment.

• Ptree metaobject

The `Ptree` metaobjects represent the parse tree of a program. The parse tree is implemented by a linked list of lexical tokens. For example, this program:

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

is represented by a `Ptree` metaobject:

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

Here, `()` denotes a linked list. We denote a parse tree with the notation that Lisp uses for the `S` expression. Note that operators such as `=` and `+` make sublists.

The OpenC++ provides many functions for manipulating a parse tree. Most of them were taken from Lisp. For example, to obtain the second sublist of the list that a variable `expr` is bound to, the programmer writes:

```
Ptree::Second(expr)
```

The function `Second()` is a `static` member function³ of the class `Ptree`. It returns the `Ptree` metaobject that represents the second sublist.

Moreover, since the grammar of C++ is relatively complex, the `Ptree` metaobjects provide a member function `WhatIs()` for examining the kind of the syntax represented by the parse tree. `WhatIs()` returns a unique constant according to the kind of the syntax. If the parse tree represents a declaration, `WhatIs()` returns `PtreeDeclarationId`; if the tree represents a class name, `WhatIs()` returns `LeafClassNameId`.

As we already saw, `Ptree` metaobjects can be constructed by calling `Ptree::Make()`. This `static` member function constructs a `Ptree` metaobject according to the given format. All occurrences of `%c` (character), `%d` (integer), `%s` (character string), and `%p` (`Ptree`) in the format are replaced with the arguments following the format. For example, suppose that `array_name` is "xpos" and `offset` is 3. This function call:

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

³A `static` member function is a sort of class method in C++.

constructs a `Ptree` metaobject that represents:

```
xpos [3]
```

Unfortunately, the current implementation of `Ptree::Make()` does not construct a fully-capable `Ptree` metaobject. Since the C++ grammar is context sensitive, `Ptree::Make()` cannot correctly parse the constructed metaobject without syntactic context and hence `WhatIs()` does not work for it. Except this limitation, however, programmers can use the `Ptree` metaobject returned by `Ptree::Make()` at any place in a meta-level program.

The function `Ptree::Make()` makes it significantly easy to write a meta-level program. It is a conversion function from standard C++ syntax given as a C++ string to a `Ptree` metaobject. Thus programmers can construct a `Ptree` metaobject with standard C++ syntax, which is more intuitive and easy to handle than bare `new` operators.

The OpenC++ MOP also provides a function for pattern matching. The `static` member function `Ptree::Match()` compares the given pattern and the given `Ptree` object. If they match, it returns `TRUE` and binds the given variables to the appropriate sublists. See the following sample program:

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

The pattern `[%? + %?]` matches a `Ptree` metaobject if the length of the linked list is three and the second element is `+`. If `expr` matches the pattern, `lexpr` gets bound to the first element of `expr`, and `rexpr` gets bound to the third element. Note that the type of `lexpr` and `rexpr` is `Ptree`.

- **Environment metaobject**

The `Environment` metaobjects represent bindings between names and their static types. Programmers can call `Lookup()` for the metaobjects to determine the type of a variable name. The returned type is represented by a `TypeInfo` metaobject.

- **TypeInfo metaobject**

The `TypeInfo` metaobjects represent types. The types are not limited to class types. They also include other kinds of types, such as built-in types like `int`, pointer types, function types, template types, and so on.

The most important member function of `TypeInfo` is `WhatIs()`. It returns a unique constant according to the kind of the type. For example, if the type is a class type, `WhatIs()` returns `ClassType`.

The `TypeInfo` metaobjects also supply member functions for obtaining detailed information of each kind of type:

- `uint IsBuiltInType()`
This works for built-in types. It returns what the built-in type is, `char`, `int`, `double`, or others.
- `Class* ClassMetaobject()`
This works for class types. It returns the `Class` metaobject for the class.
- `void Dereference(TypeInfo& t)`
This works for derived types, such as pointer types, reference types, and function types. It returns the dereferenced type of the type. For example, if the type is `int*`, `Dereference()` returns the `TypeInfo` metaobject for `int`. If the type is a function type, `Dereference()` returns the return type of the function.
- `BOOL NthArgument(int nth, TypeInfo& t)`
This works for function types. It returns the type of the `nth` argument to the function. If there is not the `nth` argument, `NthArgument()` returns `FALSE`.

• Class metaobject

The `Class` metaobjects supply member functions for introspection. Programmers can call these functions to inspect the class; for example, they can obtain the class metaobject for the base class. The followings are part of these member functions:

- `Ptree* Name()`
This returns the name of the class.
- `char* MetaclassName()`
This returns the name of the metaclass.
- `Class* NthBaseClass(Environment* env, int nth)`
This returns the `nth` base class of the class. Recall that C++ allows multiple inheritance. The leftmost base class is the first, and the

rightmost is the last. Note that this member function returns only immediate base classes. It does not return a base class of a base class.

- `Ptree* NthMemberName(int nth)`
This returns the name of the `nth` member, which is either a member function or a data member.
- `BOOL LookupMemberType(Environment* env, Ptree* name, TypeInfo& t)`
This returns the type of the member specified by `name`.

Non Local

The OpenC++ MOP also enables non-local processing. Because a single class metaobject processes all the code fragments relevant to the class, programmers usually define only one new metaclass for implementing one abstraction. Even if the processed fragments spread out over the whole program, they are automatically dispatched to the class metaobject. Although programmers have to define more than one metaclasses if the processing involves multiple classes, we believe that a class is good granularity for source-code processing in C++. This is because most of abstractions are usually implemented by a single class in C++.

For non-local processing, the `Class` metaobjects supply a different member function for each kind of program text. One of these functions is `TranslateNew()` presented in Section 4.1. These functions receive program text and translate it. The translated text is substituted for the original text in the program. Although the member functions supplied by `Class` do not change the received text at all, a subclass of `Class` can override them and implement new source-code translation. The followings are some of the member functions that the subclass can override (For the complete list, see Appendix B):

- `Ptree* TranslateClassName(Environment* env, Ptree* keyword, Ptree* name)`
This translates the class name appearing in the program.
- `Ptree* TranslateSelf(Environment* env)`
This translates the class declaration.
- `Ptree* TranslateMemberFunctionBody(Environment* env, Ptree* name, BOOL inlined,`

Ptree* body)

This translates the body of a member function supplied by the class.

- Ptree* TranslateUnary(Environment* env, Ptree* op,
 Ptree* object)

This translates an expression including a unary operator `op`. It is called if the `object` that the operator is applied to is an instance of the class.

- Ptree* TranslateNew(Environment* env, Ptree* ...)

This translates the `new` expression.

- Ptree* TranslateMemberRead(Environment* env, Ptree* member)

This translates an expression for reading a data member of an object of the class.

- Ptree* TranslateMemberCall(Environment* env, Ptree* member,
 Ptree* arglist)

This translates a member function call on an object of the class.

4.3 Syntax Extension

The OpenC++ MOP allows limited syntax extension. Because the C++ grammar is heavily context dependent, the full extensibility for syntax is difficult to provide in C++. However, programmers can define new keywords and implement the following kinds of new syntax:

- Type Modifier

Programmers can define a new type modifier. It may appear in front of type names, the `new` operator, or class declarations. For example, programmers may define new keywords, `distributed` and `remote`, and then they can write:

```
distributed class Dictionary { ... };
remote Point* p = remote(athos) new Point;
```

Here, `distributed`, `remote`, and `remote(athos)` are new type modifiers.

- Access Specifier

Programmers can define a new member-access specifier, which appears

within class declarations. `public`, `protected`, and `private` are the built-in access specifiers. For example, if `after` is a keyword for a new access specifier, then programmers may write:

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

- **Statement**

Programmers can define a new kind of statement that is similar to either the `if` statement or the `while` statement. The following examples are valid syntax extensions:

```
Matrix m1;
m1.forall(e){ e = 0.0; }    // extended syntax

ButtonWidget b;
b.press(int x, int y){    // extended syntax
    cout << "pressed at" << x << ", " << y;
};
```

In these examples, `forall` and `press` are new keywords. Like the `while` statement, they are followed by a `()` expression and a block statement.

To make these syntax extensions available, the programmers first register the keywords to the parser. The `Class` metaobjects supply the following registration functions. Each one registers a keyword for a specific syntax extension (the followings are not all the functions. See Appendix B for more details):

- `void RegisterNewModifier(char* keyword)`
This registers a keyword for a new type modifier.
- `void RegisterNewAccessSpecifier(char* keyword)`
This registers a keyword for a new access specifier.

- `void RegisterNewWhileStatement(char* keyword)`
This registers a `keyword` for a new kind of statement that is similar to the `while` statement.

The semantics of a new syntax extension is defined by the class metaobject that is involved with the extension. For example, the `forall` statement in this program:

```
Matrix m1;
m1.forall(e){ e = 0.0; }
```

Should be translated into regular C++ code by the class metaobject for `Matrix`. The programmer, therefore, has to define a new class metaobject to handle the translation. The new class metaobject will override this member function:

- `Ptree* TranslateUserStatement(Environment* env,`
`Ptree* object, Ptree* op,`
`Ptree* keyword, Ptree* rest)`

This translates a new kind of statement. The default implementation by `Class` causes a syntax error.

If the `forall` statement is translated, the arguments `object`, `op`, and `keyword` are bound to `m1`, `.` (dot), and `forall`, respectively. The last argument `rest` is bound to the rest of the statement, that is, `(e){ e = 0.0; }`. The overridden `TranslateUserStatement()` should use these arguments and construct the parse tree for the translated statement.

4.4 What is New?

The OpenC++ MOP has been developed by a synthesis and re-engineering of ideas of other known techniques. Especially Lisp macros and the CLOS MOP influence the design of the OpenC++ MOP. This section discusses comparison between the OpenC++ MOP and other techniques.

Comparison with Lisp Macros

In the OpenC++ MOP, class metaobjects process a program as Lisp macros do. Their member functions receive program text and return the translated text, which is substituted for the original text.

However, the OpenC++ provides more contextual information than Lisp macros when translating a program. The member functions for the translation receive the environment as well as program text. They can use it to examine the static types of variables. Lisp macros do not provide the environment; macro functions have to translate a program only with syntactical information but without contextual information.

The OpenC++ MOP also enables self modification whereas Lisp macros do not. With only a simple annotation, it can alter the behavior of the objects of only a specific class. Error-prone programming conventions are not needed. Lisp macros require programmers to explicitly call them to translate programs. The expressions that a Lisp macro should process have to be explicitly preceded by the macro name. Hence, if programmers want to change the behavior of the language when reading a variable, they have to write something like `(read-variable x)` instead of just writing `x`. Here `read-variable` is a macro name. Otherwise, programmers have to write the code walker. On the other hand, the OpenC++ MOP does not need such a programming convention. Once a metaclass is declared, all the expressions involved with the class are automatically processed by the class metaobject. Programmers can alter the behavior of the objects without editing the original program to insert something like macro names. Moreover, programmers can restrict the range of the self modification within a specific class. The behavior for the other classes are kept as is.

Comparison with the CLOS MOP

The CLOS MOP is the immediate ancestor of the OpenC++ MOP. Both the CLOS MOP and the OpenC++ MOP are metacircular, and they employ class metaobjects instead of the metaobjects for objects. The difference is that the OpenC++ MOP is specially designed to run at compile time.

Although the currying technique allows metaobjects in the CLOS MOP to mostly run at compile time, some computation by the metaobjects is still performed at runtime. At least, which metaobject is selected for given base-level code is determined at runtime. See Figure 3.3 in page 33 again. The default interpreter (or compiler) has to determine whether it executes each expression through a metaobject or not, and which metaobject it selects if so. In the CLOS MOP, this is done at runtime.

The OpenC++ performs all meta computation at compile time. It also determines which metaobject is selected at compile time. Because of this feature and the metacircularity, the OpenC++ MOP does not imply any performance penalty at runtime. The OpenC++ MOP uses static typing to

select a metaobject at compile time. It statically types all variables and expressions in the program and determines which metaobject is responsible for the translation. Then it calls the metaobject for translating the expression and directly substitutes the result for the original expression. This means that even the code for selecting a metaobject and dispatching to it does not run at runtime.

The OpenC++ MOP is regarded as a good synthesis of the CLOS MOP and Lisp macros. We below present the definition of `RecordedClass` written in pseudo Lisp. Comparing this definition with the equivalent definition in the CLOS MOP, the readers will intuitively understand the synthesis. First, we show the definition in the OpenC++ MOP:

```
(defclass recorded-class (metaobject-for-class)
  (defmethod create-object (env expr)
    (let ((class-name (<- self name)))
      '(begin (set! *history* (cons ,class-name *history*))
            ,(<- super create-object env expr))))))
```

The next is the definition in the CLOS MOP:

```
(defclass recorded-class (metaobject-for-class)
  (defmethod create-object (init-args)
    (let ((class-name (<- self name)))
      (set! *history* (cons class-name *history*))
      (<- super create-object init-args))))
```

The readers can see that the two definitions are significantly close to each other. The primary difference is that the definition in the OpenC++ MOP returns an expression instead of directly executing the expression. This feature is the influence by Lisp macros.

Comparison with Early Compile-time MOPs

Several MOPs running at compile time, called compile-time MOPs, have been developed earlier than the OpenC++ MOP. The earliest compile-time MOPs are *Intrigue* [37] and *Anibus* [50, 51]. They are designed for Scheme [16] and parallel Scheme although they are written in CLOS. However, their design is quite different from the OpenC++ MOP. They provide meta interface mainly for customizing the behavior of the compiler rather than source-code translation. In fact, metaobjects of *Intrigue* are internal components of the compiler, such as a parser and an optimizer. Metaobjects of *Anibus* are nodes of a parse tree and controls translation from parallel Scheme to regular Scheme including low-level primitives for parallel computing. *Anibus*'s

MOP might look similar to the *OpenC++* MOP since both MOPs control program translation, but it does not provide non-locality based on the class system (or the type system because Scheme does not include a class system). Moreover, Anibus's program translation is to transform a parse tree. The metaobjects perform the transformation instead of generating substituted source code.

CRML [31] has a compile-time MOP for ML [43]. Its MOP provides the capability similar to Lisp macros and makes it possible to use new syntax in ML. However, it also involves the same limitations as Lisp macros. For example, it does not enable self modification.

The *MPC++* MOP [33, 34] is another compile-time MOP for C++. As in *OpenC++*, it allows programmers to control source-code translation. The most significant difference between *MPC++* and *OpenC++* is that the *MPC++* MOP does not provide non-locality based on the class system. Instead, each piece of program is dispatched to a metaobject for translation according to the kind of the parse tree, such as a declaration, an `if` statement, a `+` operator, and so forth. Metaobjects of *MPC++* are nodes of the parse tree of the processed program. If a piece of program is declaration of a variable, for example, then it is dispatched to the declaration metaobject (corresponding to the `Ptree` metaobject in *OpenC++*) for the translation, rather than the class metaobject involved with the type of the declared variable.

Because of this feature, the *MPC++* MOP is not suitable for customization that is specific to a particular class and involves several statements and expressions. If the customization needs to translate declaration statements and `->` expressions on a specific class, then the programmer needs to define two new metaclasses for declaration and the `->` operator. Then the new metaclasses must explicitly determine the class of the processed code and translate the code only if it is the specific class. This computation is implicitly performed in *OpenC++* because it provides non locality.

Moreover, the *MPC++* MOP does not provide the meta-meta level. Thus programmers cannot write a meta-meta level program to make it easier to write a meta-level program. Since the *OpenC++* MOP provides the meta-meta level, for example, programmers can enjoy special syntax when writing a meta-level program.

4.5 Summary

This chapter proposed the `OpenC++` MOP, which is a mechanism for processing a program. This mechanism provides rich meta representation of the processed program and enables context-sensitive and non-local processing. For context-sensitive processing, the `OpenC++` MOP provides `Ptree` metaobjects (for program text), `Environment` metaobjects, `TypeInfo` metaobjects, and `Class` metaobjects. These metaobjects are the meta representation of various aspects of the program. The `Class` metaobjects also work for non-local processing. They control the translation of all the expressions that are spread out over the whole program but involved with the class.

Like Lisp macros, the `OpenC++` MOP also enables syntax extensions. Programmers can register a new keyword and define a new type modifier, an member-access specifier, or a new kind of statement. They are appropriately translated into regular C++ code by `Class` metaobjects as other code is translated.

These features are advantages of the `OpenC++` MOP against other existing C++ mechanisms such as inheritance and templates, and they enable C++ libraries that have been impossible or difficult to make them efficient. To enable those libraries, an extended C++ compiler specially prepared for the libraries has been necessary so far. The `OpenC++` MOP allows programmers to write a meta-level program to implement such an extension to C++ on top of the compiler.

The `OpenC++` MOP has been developed by a synthesis of ideas from Lisp macros and the CLOS MOP shown in the previous chapter. However, it is not just a C++ version of Lisp macros or the CLOS MOP. It is something considerably different from Lisp-style macros or traditional reflective languages.

The basic architecture of the `OpenC++` MOP was taken from Lisp macros. In both of them, the meta-level program receives program text and returns the translated one. Moreover, the meta-level program can run at compile time to improve execution performance of the base-level program. On the other hand, the `OpenC++` MOP provides richer meta representation than Lisp macros. It provides not only program text but also an environment and type information. Also, it provides non-locality based on the class system of C++. Furthermore, self modification is possible in the `OpenC++` MOP. Programmers can alter the behavior of the language without programming conventions or the code walker.

The protocol structure of the `OpenC++` MOP is based on that of the

CLOS MOP. The OpenC++ MOP is metacircular, and class metaobjects are the subjects of processing a program. However, the OpenC++ MOP is designed so that metaobjects run at compile time. All the meta computation, that is, source-code processing, is performed at compile time. Even which metaobject should be selected for processing is determined at compile time. The CLOS MOP performs this selection at runtime although other meta computation can be moved to compile time with the currying technique.

Chapter 5

Meta Helix

Like the CLOS MOP, the OpenC++ MOP is metacircular since metacircular systems are easy to learn and make it easier to write efficient meta programs. Unfortunately, pure metacircularity can lead to a problem we call implementation level conflation. This problem confuses programmers and causes errors, hence we could not adopt pure metacircularity *as is* for the OpenC++ MOP.

Instead, to avoid the problem, we have developed an improved version of metacircular architecture, named the meta helix [15]. This chapter presents a problem of the pure metacircular architecture, and proposes the meta helix as a solution. The meta helix preserves advantages of metacircularity but also addresses the problem we present.

5.1 Implementation Level Conflation

In a metacircular system like the CLOS MOP, the language can be customized in the customized language itself. This means not only that the base-level program and the meta-level program are written in the same language, but also that the customization by the meta-level program reflects on the language in which the meta-level program itself is written. This feature gives advantages that we presented in Chapter 3, but it can also lead to a problem which we call *implementation level conflation*.

Lisp macros

First of all, we show an example of implementation level conflation in Lisp macros, which is also metacircular. Since Lisp macros are relatively simpler

than the CLOS MOP, this example might look trivial. However, we believe that showing this example help the readers understand a more serious example we show later.

Suppose that we modify the behavior of a special form `define` so that more than one variables can be defined at the same time. With this extension, for example, we can define variables `x` and `y` with the initial value 7 by this single expression:

```
(define x y 7)
```

This expression should be expanded by a macro function named `define` into this:¹

```
(begin (define x 7) (define y 7))
```

Therefore, the definition of the macro should be this:

```
(define-macro (define . args)
  (let ((vars (list-head args (- (length args) 1)))
        (init (car (last args))))
    `(begin ,@(map vars
                   (lambda (v) `(define ,v ,init))))))
```

Unfortunately, this macro does not work; it will fall down into an infinite loop. Because the symbol `define` is now a macro name, occurrences of `define` in the expanded expression are repeatedly processed by the macro function. For example, the expanded expression:

```
(begin (define x 7) (define y 7))
```

Will be expanded again into:

```
(begin (begin (define x 7)) (begin (define y 7)))
```

And this expression also invokes macro expansion, and so on.

This problem can be fixed although the way of fixing is quite implementation dependent. To fix this problem, we have to define the macro as follows:

¹For simplicity, we assume that the sub-expression for the initial value does not cause side-effects.

```
(define define* define) // alias
(define-macro (define . args)
  (let ((vars (list-head args (- (length args) 1)))
        (init (car (last args))))
    `(begin ,(map vars
                  (lambda (v)
                    `(define* ,v ,init)))))) // use define*
```

This program first binds a symbol `define*` to the original `define` special form. Now `define*` is an alias of `define`. Then, the program defines a macro `define` and uses `define*` for the expanded expression. Using the alias makes the macro function avoid the infinite loop since the expanded expression does not include the macro name `define` any more.

A More Serious Example

The problem of the example shown above is that two distinct concepts, which are the `define` special form and the `define` macro, are conflated into a single name. We call this problem implementation level conflation because the implemented level (`define` macro) and the implementing level (`define` special form) are conflated into a single structure.

This conflation becomes more serious in metacircular systems like the CLOS MOP. The power of the CLOS MOP is that programmers can enjoy metacircularity and use the existing facilities of CLOS as much as possible when they write meta-level programs. But this is a double-edged blade. It can also easily lead to conflation of a new abstraction and the facilities implementing that abstraction.

To illustrate more serious implementation level conflation, we show an example of an extension written with the CLOS MOP. This extension records all accesses to slots (i.e. data members) of objects. The following program is an example of a program that uses this extension:²

```
> (defclass point ()
    (variable x y) :meta history-class)
POINT
> (define p1 (make-instance point))
P1
> (<- p1 x 3)           ; set x to 3
3
> (<- p1 y 2)           ; set y to 2
2
> (<- p1 x)             ; read x
```

²Again, we use the same altered syntax that we used in Chapter 3.

```

3
> (slot-history p1) ; show the access log
((GET X) (SET Y) (SET X))

```

Since the metaclass for `point` is `history-class`, all the accesses to the slots of `point` objects are recorded, and the access log is available through the function `slot-history`.

Slots with access history can be implemented using the existing slot mechanism. The way this works is simply for instances of `point` to actually have three slots, the two visible slots `x` and `y` as well as a third, “hidden” slot `history` for storing the access log. The definition of `history-class` implemented in this way is as follows:

```

(defclass history-class (metaclass-for-class)
  ; what slots do the instances have?
  (defmethod compute-slots ()
    (cons 'history (<- super compute-slots)))

  ; read a slot of an instance
  (defmethod read-slot (object slot-name)
    (<- object history
      '((get ,slot-name) ,@(<- object history)))
    (<- super read-slot object slot-name))

  (defmethod set-slot ...))

```

This metaclass overrides three methods. First, it overrides `compute-slot` so that every history class has an extra slot to store the access log. Also it overrides `read-slot` and `set-slot`, which reads and writes a slot of the instance of history classes. They update the access log before actually reading or writing. Since the access log is stored in the hidden slot `history`, they read and write the slot to update the access log.

Unfortunately, this example includes implementation level conflation. There are two distinct but conflated concepts in this implementation. One is a `point` object that has extended slots. The accesses to the slots are recorded in the access log. The other concept is a `point` object that has three non-extended slots, `x`, `y`, and hidden `history`. This object is used to implement the former object. Since the CLOS MOP is metacircular, the two `point` objects are identical — the same object. Again we are dealing with the implemented level (extended slots) and the implementing level (non-extended slots) but the two levels are conflated into a single structure (`point` object).

Problems for Users of the Extension

Implementation level conflation results in confusion for users of the extension. For example, consider that a programmer uses one of the MOP's introspective facilities to ask what slots a class has:

```
> (class-slots (find-class 'point))
(HISTORY X Y)
```

According to the specifications of the CLOS MOP, `class-slots` returns the list of the slot names that the given class has. The result, however, includes the `history` slot. Is this right? What does “the given class” mean in the specifications? Is it the implementing class or the implemented class? It should be the implemented class and hence, since the `history` slot exists only for the implementation, including the slot is entirely inappropriate.

This problem particularly shows up when using browsers and debuggers that rely on the introspective part of the CLOS MOP to work. Exposing this detail of how slots with history is implemented can leave programmers confused, or worse yet, can tempt them to rely on this implementation details in ways that they should not.

Problems for the Implementor of the Extension

Implementation level conflation also can cause problems for the implementor of the extension. A careful reading of the method `read-slot` supplied by `history-class` provides an example of this.

The implementation of `read-slot` has a bug which manifests itself as infinite recursion although it is better understood as resulting from the implementation level conflation. Operationally, the bug is that the body of the method, as part of updating the access log, must read the `history` slot, which runs this method recursively, which starts to update the access log, which reads the `history` slot, and so on ad infinitum. This bug happens because `read-slot` is implemented with a non-extended slot `history` but `read-slot` alters the language to record all the accesses to the slots, including `history`.

The standard solution to this problem is to introduce a special purpose test that prevents the infinite recursion. So the revised code ends up looking something like this:

```
(defmethod read-slot (object slot-name)
  (unless (eq slot-name 'history)
    (<- object history)))
```

```

      ‘((get ,slot-name) ,@(<- object history)))
    (<- super read-slot object slot-name))

```

This solution, while effective, seems ad hoc, can be difficult to reason about, and is not effective in general.

The Importance of an N-to-N Correspondence

The problems shown above can be better described as having to do with implementation level conflation. As mentioned at the beginning, there are two concepts of slots in play: extended slots `x` and `y`, the accesses to which are recorded, and non-extended slots `x`, `y`, and `history`, which are used to implement the extended slots. But, because of the conflation, there is only one structure, or a handle to refer to the two concepts. The bug happens since the handle always refers to the extended slots. If, somehow, the slots within the body of `read-slot` could refer to non-extended slots, then the ad-hoc solution to the infinite recursion could be avoided.

Fundamentally, if there are n distinct important views of an object — or any other structure — there needs to be n distinct handles to it. For example, we need some way for instances of the class `point` to be viewed in terms of either the implemented functionality or the implementing functionality, not a conflation of both. The users of the extension want a view in terms of the implemented functionality, but the implementor of the extension wants to be able to take one view or the other at different times.

5.2 Inadequate Solutions

Implementation level conflation is a flaw of the metacircular architecture. Despite the advantages, this architecture makes metaobject protocols lose elegance; the metaobject protocols become inconsistent and difficult to use. Because of this fact, we could not adopt the pure metacircular architecture for the OpenC++ MOP. Instead, we have developed a improved metacircular architecture since the OpenC++ MOP should be elegant, simple, and easy to use, as much as possible.

Before presenting our solution to the problem of implementation level conflation, we first present two earlier solutions that, in different ways, fail to our needs. These solutions serve to further flesh out the criteria which the more general solution should meet.

Change the Implementation

One possible solution to this problem involves implementing the extension in a different way, specifically by storing the access log in the class metaobject rather than directly in the objects themselves. The following program implements history classes in this way:

```
(defclass history-class (metaobject-for-class)
  (variable history) ; place to store the access log

  (defmethod read-slot (object slot-name)
    (let ((log (assq object (<- self history))))
      (set-cdr! log '(get ,slot-name) ,(cdr log))
      (<- super read-slot object slot-name)))

  (defmethod set-slot ...))
```

All the access logs for the instances are stored in the slot `history` of the class metaobjects (that is, a class variable in the Smalltalk terminology). So the value of `history` is a list of pairs of an object and its access log. This association list can be searched by `assq` with using `object` as a search key.

This solution solves the specific problems mentioned in the previous section, but it loses advantages of metacircularity, which we would like to keep. This solution must manually implement the mapping from individual objects to their access logs, even though that basic functionality is already present in CLOS. Implementing that mapping is not needed if each access log is directly stored in a slot of the object. This solution is not only redundant, but also difficult to achieve sufficient execution performance. The implementor of `history-class` has to implement as efficient a mapping mechanism as the default one, which is a slot of an object. At least, the implementation shown above does not satisfy this performance criterion; the association list is significantly inefficient.

Tiny CLOS MOP

If we give up metacircularity, we can avoid implementation level conflation. Non-metacircular systems such as 3-Lisp and 3-KRS do not involve this problem because they have two distinct handles for implemented functionality and implementing functionality. In 3-KRS, if programmers want a view in terms of the implemented functionality, they should refer to the base-level objects, but if they want a view in terms of the implementing functionality, they should refer to the associated metaobjects. Since there is no metacircularity, the base-level objects and the metaobjects are never

conflated. The behavior of the metaobjects does not change depending on the metaobjects themselves.

The Tiny CLOS MOP developed by Gregor Kiczales et al avoids implementation level conflation by partially giving up metacircularity. The Tiny CLOS MOP provides two different abstractions for per instance storage: slots and fields. Fields are a lower-level abstraction used to implement slots; they represent memory image allocated for implementing each object. The base-level Tiny CLOS programs never know the fields exist.

In the specific example of the history class, the extension works by allocating an extra *field* for each object. So, for example, `point` objects have two slots `x` and `y`; but they have three fields, for holding the `x` and `y` slots and the slot access log. The implementation of `history-class` in Tiny CLOS looks like:

```
(defclass history-class (metaobject-for-class)
  (variable history-index) ; the index of the field that will
                           ; store the access log for instances
                           ; of the class

  ; allocate an extra field and remember its index
  (defmethod compute-fields ()
    (<- self history-index (<- self allocate-field))
    (<- super compute-fields))

  (defmethod read-slot (object slot-name)
    (let* ((index (<- self history-index)))
      (<- self set-field object index
        '((get ,slot-name)
          ,@(<- self get-field object index))))
    (<- super read-slot object slot-name))

  (defmethod set-slot ...))
```

Fields are accessed through the methods `get-field` and `set-field`, and each field is specified by its index instead of its name. Fields have a more primitive naming mechanism in terms of indices.

Again, this solution solves the specific problems mentioned in the previous section. Implementation level conflation is avoided because (1) no “hidden” slot for the accesses log is added, and (2) the access log stored in an object is retrieved through the lower-level methods `get-field` and `set-field`, which do not invoke `read-slot` recursively. Since the Tiny CLOS MOP provides two distinct handles, slots and fields, for implemented functionality and implementing functionality, programmers can select an appropriate view by switching the handles.

Unfortunately, this solution has significant problems of its own which make it unsuitable as a general solution. First, this solution loses advantages of metacircularity. Because the Tiny CLOS MOP is not metacircular in terms of slots, programmers have to learn lower-level abstractions and writing efficient meta-level programs is difficult. Second, this solution is only effective in the presence of a single extension to slot functionality. If, for example, someone wanted an additional extension (i.e. to store the objects in a persistent database [47]) there would still be conflation. This is because in such a situation there needs to be (at least) three views. The view of persistent objects with a slot access history, built on top of the view of persistent objects, built on top of ordinary objects. But the Tiny CLOS MOP provides only two levels, so there will still be some conflation. To avoid implementation level conflation in n levels of implementation, the MOP must provide support for n different views.

5.3 The Meta Helix Architecture

The common idea underlying the two unsatisfactory solutions is to distinguish the implemented and implementing functionality by using a different “handle” for each. In the first proposed solution, the objects are the handle to the implemented functionality, and the class metaobject is the handle to the implementing functionality. In the second proposed solution, slots are the handle to the implemented functionality, while fields are the handle to the implementing functionality.

But the problem with both of these solutions is that the benefits of metacircularity is destroyed by the fact that the two handles are distinct abstractions. The idea of our proposed solution is to address its problem by providing two handles, but to retain the benefits of metacircularity by the use of the same abstraction for the two handles. For example, in the case of

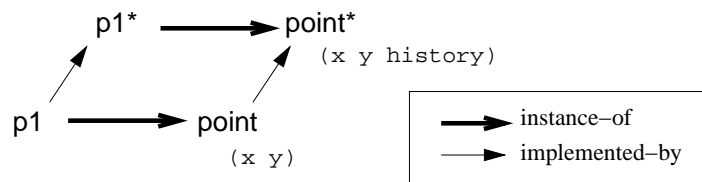


Figure 5.1: Two handles with the same abstraction

the `history-class` extension, our solution has two class metaobjects, `point` and `point*`, to represent different implementation levels (Figure 5.1). The class `point` corresponds to the class in the extended language, which keeps slot access histories, while the class `point*` corresponds to the class in the non-extended language, which is used to implement the extended one.

Note that the two handles indicate the same entity. For example, the class `point` and the class `point*` corresponds to the same class. They are just handles to distinct views of the same entity as slots and fields are handles to distinct views of per-instance storage in the Tiny CLOS MOP. `point` is the handle to the implemented view and `point*` is the handle to the implementing view.

Although the two handles, such as `point` and `point*`, are often very similar, the relationship between the handles does not fit the usual `subclass-of` relationship. It cannot be either the `instance-of` or `base-meta` relationships. To capture the relationship between the two handles, we introduce an `implemented-by` relationship. In Figure 5.1, the class `point` is implemented-by the class `point*`, and the object `p1` is implemented-by the object `p1*`. The objects `p1` and `p1*` are instances-of `point` and `point*`, respectively.

If the `implemented-by` relationship is used, the problems by implementation level conflation can be easily solved. First, the programmer who want to ask what slots a class has can obtain an appropriate result:

```
> (class-slots (find-class 'point))
(X Y)
> (class-slots (implemented-by (find-class 'point)))
(HISTORY X Y)
```

Second, the method `read-slot` becomes more simple and easy to read. No ad-hoc techniques for avoiding infinite recursion are needed any more:

```
(defmethod read-slot (object slot-name)
  (let ((object* (implemented-by object)))
    (<- object* history
      '((get ,slot-name) ,@(<- object* history)))
    (<- super read-slot object slot-name)))
```

Note that `read-slot` is the method for not the class metaobject for `point*` but the class metaobject for `point`. The behavior of the object `p1`, which has slots with history, is controlled by the class metaobject for `point`.

Our choice of the name meta-helix for this architecture is best seen when thinking in terms of the relation between handles that the different solutions use. As shown in Figure 5.2, in the pure metacircular approach, the

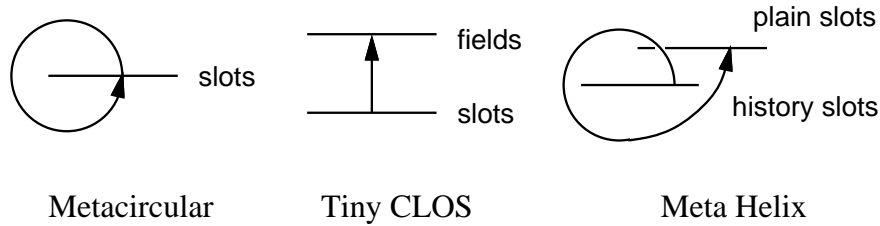


Figure 5.2: The implementation relation between interfaces

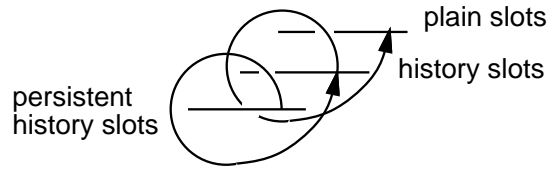
implementation loops directly back onto itself—leading to conflation. In the Tiny CLOS approach, the implementation maps between two distinct functionalities—leading to added complexity. In the meta-helical approach, the implementation spirals between two distinct handles of (nearly) identical functionality—preserving what is good about the metacircular approach, while still reifying the distinction that prevents conflation of implementation levels.

Similarly, the meta helix works when there are more than two implementation levels. So, for example, in the case of the persistent history class mentioned in the previous section, we can create the three levels that are needed to maintain separate views, and relate them using implemented-by relationships. This is shown in Figure 5.3, which illustrates the helical nature of this architecture.

While distinguishing implementation levels, the meta helix architecture preserves the benefits of metacircularity because the meta helix architecture is a super set of the metacircular architecture. Except the implemented-by relationship, the meta helix architecture is quite identical to the metacircular architecture. So, for example, unlike the Tiny CLOS MOP, programmers do not need to learn new abstractions such as fields to implement an extended concept of slots. Also, writing an efficient meta-level program is still easy.

5.4 Implementing the Meta Helix

The OpenC++ MOP is based on the meta helix architecture. This section first shows this fact, especially focusing on how the OpenC++ reifies the implemented-by relationship. The way of realizing the implemented-by relationship is a main issue for implementing the meta helix architecture. Then this section shows how the implemented-by relationship should be reified for the CLOS MOP. Through this example, we present that the meta



A Three level Meta Helix

Figure 5.3: The Meta Helix supports n implementation levels.

helix architecture is applicable to not only the OpenC++ MOP but also other kinds of MOPs.

The OpenC++ MOP is Meta-Helical

In the OpenC++ MOP, the implemented and implementing levels are naturally separated; the implemented level is the original program before translation and the implementing level is the resulting program of the translation. To implement the meta helix architecture, therefore, we should be just able to distinguish classes in the original program from the classes in the resulting program, and relate the corresponding classes by the implemented-by relationship.

The implemented-by relationship in the OpenC++ MOP is reified with an aliasing technique, which we used to avoid implementation level conflation caused by the macro `define` in Section 5.1. To do this, the OpenC++ MOP includes the following rule to be meta-helical:

- Any class appearing in the original program must be renamed in the resulting program of the translation.

This rule reifies the implemented-by relationship. For example, in the example of history classes, the class `point` should be renamed `point*`. This aliasing gives programmers two handles to the implemented and implementing functionality. One handle is the class `point` for the implemented functionality, and the other is the class `point*` for the implementing functionality. By switching the two class names, programmers can avoid implementation level conflation. First, they are not confused by the introspective part of the MOP any more:³

³For making the argument clearer, we use Lisp-style syntax to show an OpenC++ program.

```
> (class-of 'point)
(X Y)
> (class-of 'point*)
(X Y HISTORY)
```

Also, the method `read-slot` is implemented without confusion in an intuitive way (This implementation uses nested backquotes for emphasizing the similarity to other versions of implementations. Although nested backquotes might make the program look complex, the complexity does not result from the meta helix.):

```
(defmethod read-slot (env object slot-name)
  `(begin (<- ,object history
           `((get ',slot-name) ,@(<- ,object history)))
         ,(<- super read-slot env object slot-name)))
```

For example, this method translates an expression:

```
(<- p1 x)
```

into this expression:

```
(begin (<- p1 history
        `((get x) ,@(<- p1 history)))
      (<- p1 x))
```

Note that the type of `p1` is not the class `point` but the class `point*` after the translation. During the translation, all occurrences of the class name `point` are replaced with `point*`. Therefore, after the translation, the expression `(<- p1 history)` is not recursively processed by the class metaobject for `point`. Rather, it is processed by the class metaobject for `point*`, which is a distinct metaobject and would be the default one. If programmers want to recursively process, they can declare that the type of `p1` is still the class `point`.

The OpenC++ MOP also supports multiple implementation levels. The persistent history slots shown in Figure 5.3 can be implemented by just specifying a non-default metaclass `persistent-class` for `point*`. If the metaclass is not the default one, all the expressions involved with the class `point*` are recursively translated by the class metaobject for `point*`. This process is repeated until the metaclass becomes the default one.

A Meta-Helical Version of the CLOS MOP

The meta helix is applicable to other kinds MOPs such as the CLOS MOP. In the OpenC++ MOP, the implemented and implementing functionality are naturally separated into the original program and the translated program. In the CLOS MOP, however, the implemented and implementing functionality coexist in the same runtime environment. So we need a different technique to reify the implemented-by relationship.

We present that delegation works for reifying the implemented-by relationship for the CLOS MOP. In a meta-helical version of the CLOS MOP, a class metaobject is responsible for defining the class that will implement it. The class metaobject for `point` will produce a class `point*` equivalent to the definition:

```
(defclass point* ()
  (variable x y history))
```

As in the OpenC++ MOP, the class `point*` is the handle to the implementing functionality. The class metaobject for `point` delegates most of its work to the class metaobject for `point*`. For example, when an instance of `point` is created, the class `point` asks the class `point*` to create a `point*` object. The primitive function `implemented-by` on a `point` object returns the `point*` object. The slot access primitive `read-slot` is then specialized by the class metaobject for `point`. Its implementation is:

```
(defmethod read-slot (object slot-name)
  (let ((object* (implemented-by object)))
    (<- object* history
      '((get ,slot-name) ,@(<- object* history)))
    (<- super read-slot object slot-name)))
```

The method `read-slot` first gets the implementing object `object*` for that object and then updates its `history` slot. The method finally invokes `read-slot` supplied by the super class, which delegates the actual work of implementation to the class metaobject for `point*`.

5.5 Summary

This chapter presented a new analysis of a problem that arises in existing metacircular systems. This analysis shows how pure metacircularity causes confusion when there are not clearly distinguished views of the implemented and implementing functionality. We call this problem implementation level

conflation. This chapter first shows an example of the confusion in Lisp macros, which is a simple metacircular system. Then it shows a more serious example with the CLOS MOP. The confusion makes troubles for both the users of the language extension and the implementor of the extension.

Because of implementation level conflation, we did not adopt the pure metacircular architecture for the OpenC++ MOP. Although implementation level conflation does not reduce the capability of the MOP for processing programs, it severely impacts the elegance of the design of the MOP. The elegance is a significant matter because making it really easy for programmers to process programs is one of the design goals of metaobject protocols in general.

To avoid implementation level conflation while keeping benefits of metacircularity, we have developed a improved metacircular architecture, named the meta helix, for the OpenC++ MOP. It addresses the problem by providing two distinct handles to the implemented and implementing functionality while keeping benefits of metacircularity by using the same abstraction for the two handles. Programmers can enjoy metacircularity and, if needed, switch the handles to distinguish the implementing level from the implemented level. The meta helix is not only for the OpenC++ MOP but also other kinds of MOPs. To show this, this chapter also presented a meta-helical version of the CLOS MOP.

Chapter 6

Libraries in OpenC++

Because of the ability for context-sensitive and non-local processing, the OpenC++ MOP makes it possible to include useful control/data abstractions in a library and, if it is already possible in regular C++, to implement the abstractions more efficiently. This chapter illustrates examples of libraries that the OpenC++ MOP enables but regular C++ does not.

The first two examples show abstractions that the OpenC++ MOP enables, and the next example presents that we can write a class library for metaclasses and make it easier to write similar metaclasses. Then, we mention that the OpenC++ MOP is also effective for meta-level programming. A few abstractions provided by the OpenC++ MOP for meta-level programmers are implemented by the MOP itself. Finally, we show examples of abstractions that the OpenC++ MOP can make efficient.

6.1 Named Object Library

The OpenC++ MOP makes it possible to implement the named object library that we presented in Chapter 2. With this library, the users can get the class name of an object at runtime. For example, the users may write something like this:

```
class Complex : public NamedObject {
public:
    double r, i;
};

void f(Complex* x)
{
    cout << "x is " << x->ClassName();
}
```

```
}

```

If invoked, a function `f()` displays “`x is Complex`”.

To implement this library, the developer needs to write two kinds of programs: a base-level program and a meta-level program. The base-level program defines a library class `NamedObject` and it is linked with the user program. The meta-level program defines a metaclass for processing the user program. We first show the base-level program:

```
metaclass NamedObject : NamedObjectClass;
class NamedObject {};

```

The base-level program defines a library class `NamedObject`. Also, the program declares that the metaclass of `NamedObject` is `NamedObjectClass`. Note that the class `NamedObject` does not include the member function `ClassName()`. It is automatically inserted to `NamedObject` and its subclasses by the meta-level program. For example, the subclass `Complex` defined by the library user is translated by the meta-level program into:

```
class Complex : public NamedObject {
public:
    double r, i;
    virtual char* ClassName() { return "Complex"; }
};

```

The meta-level program defines the metaclass `NamedObjectClass`, which performs the translation mentioned above. Since a subclass of `NamedObject` inherits the metaclass from `NamedObject`, the metaclass `NamedObjectClass` controls the translation on the class `Complex` as well although there is no explicit metaclass declaration. The definition of `NamedObjectClass` is as follows:

```
class NamedObjectClass : public Class {
public:
    NamedObjectClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateBody(Environment* env, Ptree* body){
        Ptree* mf = Ptree::qMake(
            "public:"
            "virtual char* ClassName() {"
            "    return \"'Name()'\";"
            "}"
            "\n");
        return Ptree::Append(body, mf);
    }
};

```

This metaclass overrides the member function `TranslateBody()`, which is invoked to translate members included in a class declaration. The overridden member function just constructs a `Ptree` metaobject that represents the member function `ClassName()`, and appends it to the other members. `Ptree::qMake()` is a member function provided by the OpenC++ MOP. It constructs a `Ptree` metaobject according to the given format. Unlike a similar function `Ptree::Make()`, it computes an expression appearing in the format if the expression is surrounded by back-quotes (```). In the example above, `Name()` is computed at compile time and the resulting value is embedded in the constructed `Ptree` metaobject. `Name()` returns the name of the class metaobject. The difference between `qMake()` and `Make()` is analogous to the difference between the back-quote notation and the quote notation in Lisp (As for the back-quote notation, see Appendix A). We show the implementation of `qMake()` later in this chapter since its implementation needs the OpenC++ MOP; The member function `qMake()` cannot be implemented in regular C++.

To use this library, programmers first need to compile the meta-level program by the OpenC++ compiler, and then they have to link the compiled code with the original OpenC++ compiler. Suppose that the file name of the meta-level program is `nameclass.cc`:

```
% occ -- -o myocc opencxx.a nameclass.cc
```

The resulting executable module `myocc` is an extended OpenC++ compiler with which programmers can use the metaclass `NamedObjectClass`. The linked archive `opencxx.a` is an archive including the original OpenC++ compiler `occ`.

The library users compile their program by `myocc`. For example, to compile a source file `complex.cc`, the users may say something like this:

```
% myocc complex.cc
```

This command compiles `complex.cc` by the extended OpenC++ compiler and produces an executable module.

6.2 Distributed Object Library

Distributed objects are another example of data abstractions that regular C++ cannot include in a library. This section illustrates how the OpenC++ MOP works for including this abstraction in a library. This example needs

more member functions to be inserted in library users' classes by a metaclass. Hence, through this example, we illustrate how to use various metaobjects like a `TypeInfo` metaobject for meta-level programming.

Developing a library with the OpenC++ MOP follows three steps: (1) determine what a user program should look like, (2) figure out what the user program should be translated into and what runtime library is needed to run the translated program, and (3) write a meta-level program to perform the translation and also write the necessary runtime library. We present the implementation of the distributed object library in the order of the three steps.

What the user program should look like

The distributed object library helps the users write a program with distributed objects. The users should be able to define a distributed object as easy as they define a non-distributed object.

For example, the users write something like this code:

```
metaclass Rectangle : DistributionClass;
class Rectangle {
public:
    Rectangle(int l, int h) { length = l; height = h; }
    int Stretch(int l, int h) {
        length += l; height += h; return length * height;
    }
    :
    int length, height;
};
```

Note that a class `Rectangle` turns to a class for distributed objects by just putting a metaclass declaration.

Once putting the declaration, the user can create a `Rectangle` object on a server machine and access it from a client machine without concern of the location of the object:

```
// server side
main()
{
    Rectangle* r = new Rectangle(3, 4);
    Export(r, "rect", Rectangle);
    ServerLoop();
}
```

This program first creates a `Rectangle` object and then exports the object for clients. The (macro) function `Export()` is a library function for exporting a distributed object with a global name. The first argument is the exported object, the second argument is the global name, and the third argument is the type name of the exported object. In the example above, a `Rectangle` object `r` is exported with a global name `"rect"`. The function `ServerLoop()` is another library function, which starts waiting for requests from a client. The server program has to call this library function after all distributed objects are ready.

After `ServerLoop()` is called, the client program can freely access the distributed object on the server machine:

```
// client side
main()
{
    StartupClient("calvin");
    Rectangle* obj = Import("rect", Rectangle);
    cout << "new size: " << obj->Stretch(1, 3);
}
```

The client program first calls a library function `StartupClient()`, which connects the client to the server machine specified by the argument. In the example above, the client connects to the server machine named `"calvin"`. Then the client program imports a distributed object from the server by a library (macro) function `Import()`. Once the distributed object is imported, the client program can deal with the object in the same way that it deals with ordinary objects.

What the user program should be translated into

To run the user program shown above, the meta-level program of the library needs to translate the user program and inserts marshalling and unmarshalling code. The marshalling code converts function arguments into a byte stream so that lower-level functions can handle and send them to a remote machine. The unmarshalling code performs the reverse conversion.

First, the member function call appearing in the client program should be translated so that marshalling code is inserted. The underlined code in the original client program:

```
// client side
main()
{
    StartupClient("calvin");
```

```

    Rectangle* obj = Import("rect", Rectangle);
    cout << "new size: " << obj->Stretch(1, 3);
}

```

should be translated into this:

```

// client side
main()
{
    int i;
    StartupClient("calvin");
    Rectangle* obj = Import("rect", Rectangle);
    cout << "new size: "
         << (i = 0, *(int*)&mBuf[i]=1, i+=sizeof(int),
             *(int*)&mBuf[i]=3, i+=sizeof(int),
             CallRemote(i, obj, 2));
}

```

The substituted code shown by the underline copies two integer arguments 1 and 3 into an array of characters `mBuf`. This copying involves type conversion.¹ The variable `i` means that the size of the copied arguments. Then, a library function `CallRemote()` is called for sending the arguments stored in `mBuf` to the server machine. `CallRemote()` deals with the arguments in `mBuf` just as a simple byte stream. Since the implementation of the marshalling code depends on the signature of `Stretch()`, it should be inserted by the meta-level program.

At the server side, unmarshalling code needs to be inserted. The user program should be translated into the following program:

```

class Rectangle {
public:
    Rectangle(int l, int h) { length = l; height = h; }
    int Stretch(int l, int h) {
        length += l; height += h; return length * height;
    }
    :
    int length, height;
    void Dispatch(int*, void*, int);
};

void Rectangle::Dispatch(int* buf, void* obj, int member)

```

¹The type conversion shown above assumes that all machines are based on the same architecture. For real systems, it should absorb difference between architectures, such as little endian and big endian.

```

{
    switch(member){
        :
        case 2 :    // if Stretch() is called
        {
            int s = 0;
            int p1 = *(int*)&buf[s];
            s += sizeof(int);
            int p2 = *(int*)&buf[s];
            s += sizeof(int);
            *(int*)buffer = ((Rectangle*)obj)->Stretch(p1, p2);
        }
        break;
        :
    };
}

```

After the translation, a member function `Dispatch()` is appended to the class `Rectangle`. `Dispatch()` is used to invoke a member function for a distributed object when a client program calls the member function. It receives a byte stream (`buf`), a pointer to the object (`obj`), and an integer indicating the called member function (`member`). It unmarshals the byte stream to function arguments according to the value of `member`, and invokes the called member function with the unmarshalled arguments. Note that the unmarshalling code also depends on the signature of the called function and hence it needs to be produced by the meta-level program.

Write a runtime library

After determining what a user program should be translated into, the library developer writes a runtime library that the translated user program uses to run. It is an ordinary library written in regular C++ and includes such functions as `Export()`, `Import()`, `ServerLoop()`, `CallRemote()`, and so on. We show the implementation of the runtime library in Appendix C.1. The readers who are interested in details may see it.

Write a meta-level program

The rest of the work that the developer has to do is to write a meta-level program for the translation mentioned above. The meta-level program defines the metaclass `DistributionClass`, which overrides member functions `TranslateSelf()` and `TranslateMemberCall()` inherited from the default

metaclass `Class`. `TranslateSelf()` controls the translation of a class definition and `TranslateMemberCall()` controls the translation of a member function call expression. Since the whole meta-level program is about a hundred lines, we show it in Appendix C.1 and here present just highlights of the program.

The role of the overridden `TranslateSelf()` is to produce a member function `Dispatch()` and inserts it into the user program. This is the implementation of `TranslateSelf()`:

```
Ptree* DistributionClass::TranslateSelf(Environment* env)
{
    Ptree* name;
    TypeInfo t;
    int i;
    Ptree* code = nil;
    for(i = 0; (name = NthMemberName(i)) != nil; ++i){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId
           && whatis != PtreeDestructorId
           && LookupMemberType(env, name, t)
           && t.WhatIs() == FunctionType)
        {
            code = AppendDecoder(code, name, i, t);
        }
    }

    AppendAfterToplevel(Ptree::qMake(
        "void 'Name()'::Dispatch(int* buf, void* object,"
        "int member){\n"
        "switch(member){\n 'code' }"}));

    return Class::TranslateSelf(env);
}
```

This member function calls `AppendDecoder()` for every member function that the translated class has. In the `while` loop, each member is retrieved by `NthMemberName()` and the type of the member is examined to determine whether the member is a member function or a data member. The member type is represented by a `TypeInfo` metaobject `t` returned by `LookupMemberType()`. If the member is neither a data member, a constructor, or a destructor, then the member function `AppendDecoder()` is called for the member. `AppendDecoder()` produces a `case` block and appends it to `code`. For example, `AppendDecoder()` produces the following code for the `Stretch()` member function:

```
case 2 :
```

```

{
  int s = 0;
  int p1 = *(int*)&buf[s];
  s += sizeof(int);
  int p2 = *(int*)&buf[s];
  s += sizeof(int);
  *(int*)buf = ((Rectangle*)obj)->Stretch(p1, p2);
}
break;

```

The code produced by `AppendDecoder()` is included by the implementation of `Dispatch()`, which is eventually inserted by `AppendAfterToplevel()` just after the translated class definition.

`TypeInfo` metaobjects are also used in `AppendDecoder()`. For example, the following for loop is part of `AppendDecoder()`:

```

TypeInfo atype;
for(i = 0; t.NthArgument(i, atype); ++i){
  Ptree* argtype = atype.MakePtree();
  code = Ptree::Snoc(code, Ptree::qMake(
    "'argtype' p'i' = *('argtype')&buf[s];\n"
    "s += sizeof('argtype');\n"));
}

```

This for loop uses `TypeInfo` metaobjects to produce the code for retrieving arguments from a network message stored in `buf`. The variable `t` is the `TypeInfo` metaobject for the type of the processed member function. Note that the type of each argument is obtained by calling `NthArgument()` for this metaobject. `MakePtree()` is another important member function of `TypeInfo`. It converts the `TypeInfo` metaobject to a `Ptree` metaobject that represents the type name. In the code shown above, `MakePtree()` is used to obtain the type name of each argument.

The metaclass `DistributionClass` also overrides `TranslateMemberCall()`, which translates a member function call expression. For example, it translates an expression in the user program:

```
obj->Stretch(1, 3)
```

into something like this expression:

```

int i;
  :
(i = 0, *(int*)&mBuf[i]=1, i+=sizeof(int),
 *(int*)&mBuf[i]=3, i+=sizeof(int), CallRemote(i, obj, 2))

```

The implementation of `TranslateMemberCall()` is similar to the implementation of `AppendDecoder()`. It uses `TypeInfo` metaobjects and produces an expression that converts function arguments to a network message stored in `mBuf`:

```
Ptree* DistributionClass::TranslateMemberCall(
    Environment* env, Ptree* object,
    Ptree* op, Ptree* member, Ptree* arglist)
{
    TypeInfo ftype, atype;
    int id = IsMember(member);

    PtreeIter next(Ptree::Second(arglist));
    Ptree* code = nil;
    Ptree* tmp = Ptree::GenSym();

    env->InsertDeclaration(Ptree::qMake("int 'tmp';"));
    LookupMemberType(env, member, ftype);
    for(int i = 0; ftype.NthArgument(i, atype); ++i){
        Ptree* p = next();
        Ptree* tname = atype.MakePtree();
        code = Ptree::Snoc(code, Ptree::qMake(
            "*( 'tname'*)&mBuf['tmp']"
            "= 'TranslateExpression(env, p)',"
            "'tmp' += sizeof('tname'),"));
        next(); // skip ,
    }

    return Ptree::qMake("( 'tmp'=0, 'code'"
        "CallRemote('tmp', 'object', 'id')"));
}
```

This member function first inserts a variable declaration in the processed program by calling `InsertDeclaration()` for `env`. This declares a temporary variable used in the translated expression. The name of the temporary variable is given by calling `Ptree::GenSym()`. Then `TranslateMemberCall()` looks up the type of the called member function and produces the code for converting function arguments to a network message. The produced code is finally connected with other code and returned as the result of the translation.

6.3 Wrapper Library

Since a similar kind of abstraction often requires similar code translation, programmers may write a library to help meta-level programming for the

similar code translation. Such a library should be called *a metaclass library*. In this section, we present an example of metaclass libraries.

A wrapper function is a useful technique for implementing abstractions such as concurrent objects. It is a function that wraps another function in itself and, if invoked, simply calls the wrapped function. But it may also performs some computation before or after calling it. For example:

```
int f(int i) { return i + 1; }

int wrap_f(int i) {
    cout << "f() is called.\n"
    return f(i);
}
```

Here, `wrap_f()` is a wrapper function for `f()`. It prints a message and then calls the wrapped function `f()`.

A number of abstractions can be implemented by metaclasses that produce wrappers for all member functions of a class. Suppose that a metaclass `MyWrapperClass` does such a thing. This metaclass translates the user program shown below:

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

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

void Point::Move(int diff_x, int diff_y) {
    x += diff_x; y += diff_y;
}

void f(Point& p)
{
    p.Move(3, 5);    // call Move()
    p.rMove(-1, 2); // call rMove()
}
```

into the following program including wrapper functions for `Move()` and `rMove()`:

```

class Point {
public:
    void Move(int, int);
    void rMove(int, int);
    int x, y;
public:
    void wrapper_Move(int, int);
    void wrapper_rMove(int, int);
};

void Point::Move(int new_x, int new_y) { ... }

// inserted wrapper function for Move()
void Point::wrapper_Move(int p1, int p2)
{
    cout << "Move() is called.\n";
    Move(p1, p2);
}

void Point::rMove(int diff_x, int diff_y) { ... }

// inserted wrapper function for rMove()
void Point::wrapper_rMove(int p1, int p2)
{
    cout << "rMove() is called.\n";
    rMove(p1, p2);
}

void f(Point& p)
{
    p.wrapper_Move(3, 5);    // call the wrapper
    p.wrapper_rMove(-1, 2); // call the wrapper
}

```

Note that all occurrences of member calls for `Point` objects are substituted by the call of the wrapper function. For example, the call of `Move()` in `f()` is substituted by the call of the wrapper function.

This kind of wrapper metaclass is found in implementations of many abstractions. The translation by those wrapper metaclasses are quite similar and the only difference is what the produced wrapper functions perform before or after calling the wrapped functions. For example, the wrappers shown above just print a message, but, if they instead perform synchronization before calling the wrapped function, then `Point` objects will be concurrent objects.

A meta-level program using a metaclass library

Since wrapper metaclasses like `MyWrapperClass` are quite similar to each other, we should write a base class of these metaclasses and provide it as a metaclass library. Library developers can make their wrapper classes inherit from the base class and focus on what the wrapper functions perform before or after calling the wrapped function.

Let the name of the base class be `WrapperClass`. With the metaclass library, `MyWrapperClass` should be defined by the following simple code:

```
class MyWrapperClass : public WrapperClass {
public:
    MyWrapperClass(Ptree* d, Ptree* m) : WrapperClass(d, m){}
    Ptree* WrapperBody(Environment*, Ptree*, Ptree*, int,
                       TypeInfo&);
};

Ptree* WrapperBody(Environment* e, Ptree* name, Ptree* wrapper,
                   int nargs, TypeInfo& ftype)
{
    Ptree* body = Class::WrapperBody(e, name, wrapper, nargs,
                                     ftype);
    return Ptree::qMake(
        "cout << \"'name'() is called.\n\"; 'body'");
}
```

Note that the metaclass `MyWrapperClass` inherits from `WrapperClass` and overrides only a member function `WrapperBody()`, which produces the body of a wrapper function. The base class performs the rest of the translation, which is to insert the declarations of wrapper functions, to replace all occurrences of member calls with calls of the wrapper functions, and so forth.

A metaclass library

The metaclass `WrapperClass` provided by the metaclass library needs to execute three things: (1) to insert wrapper functions in the user program, (2) to substitute calls of wrapper functions for calls of the wrapped functions, and (3) to provide a member function that a subclass of `WrapperClass` can override for specifying the behavior of wrapper functions. We below present overviews of how the three things are implemented. The complete implementation of `WrapperClass` is found in Appendix C.2.

For (1), `WrapperClass` overrides two member functions. One is `TranslateBody()`, which controls the translation of a class definition. It is over-

ridden to insert the declarations of wrapper functions in the class definition. For example, the definition of the class `Point` is translated into this code:

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

The underlined code is the inserted declaration. To do this translation, `TranslateBody()` examines each 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)
{
    Ptree* decl = Ptree::qMake("public:\n");
    Ptree* name;
    TypeInfo t;
    int i = 0;
    while((name = NthMemberName(i++)) != nil){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId
           && whatis != PtreeDestructorId
           && LookupMemberType(env, name, t)
           && t.WhatIs() == FunctionType){
            Ptree* m = t.MakePtree(WrapperName(name));
            decl = Ptree::qMake("'decl' 'm';\n");
        }
    }
    return Ptree::Append(body, decl);
}
```

Here, the member function `WrapperName()` a member function of the class `WrapperClass`. It returns the name of the wrapper function for the given member function.

The other member function overridden by `WrapperClass` is `TranslateMemberFunctionBody()`, which translates the body of a member function. It is overridden to produce the definitions of wrapper functions. For example, `TranslateMemberFunctionBody()` processes the definition of a member function `Move()` and inserts the definition of the wrapper function `wrapper_Move()` right after `Move()`:

```

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

void Point::wrapper_Move(int p1, int p2)
{
    cout << "Move() is called.\n";
    Move(p1, p2);
}

```

`TranslateMemberFunctionBody()` constructs the definition of `wrapper_Move()` from the `TypeInfo` metaobject for the type of `Move()`. First, it derives argument types and the return type from that `TypeInfo` metaobject and converts those types to `Ptree` metaobjects by calling `MakePtree()`. Then it assembles the converted `Ptree` metaobjects with the wrapper name and the body of the wrapper function, and constructs the complete definition of `wrapper_Move()`. The overall structure of `TranslateMemberFunctionBody()` is as follows:

```

Ptree* WrapperClass::TranslateMemberFunctionBody(...)
{
    :
    Ptree* arglist = argument list of the wrapper function
    :
    Ptree* body = WrapperBody(env, name, wrapper_name, i - 1, t);
    Ptree* head = Ptree::qMake(
        "Name() ':: 'wrapper_name' ('arglist'");
    :
    AppendAfterToplevel(Ptree::Make("head '{body}'\n"));
    return Class::TranslateMemberFunctionBody(...);
}

```

The produced definition of `wrapper_Move()` is inserted by calling `AppendAfterToplevel()`. Note that `TranslateMemberFunctionBody()` calls `WrapperBody()` to make a function body so that (3) a subclass of `WrapperClass` can override it and specify the behavior of wrapper functions. The default `WrapperBody()` supplied by `WrapperClass` returns an expression that just calls the wrapped function.

Finally, we show `TranslateMemberCall()`, which `WrapperClass` overrides for (2). It substitutes calls of wrapper functions for calls of the wrapped functions. To do this substitution, it just calls `TranslateMemberCall()` supplied by `Class` with the name of a wrapper function:


```
Ptree* WrapperClass::TranslateMemberCall(Environment* env,
                                         Ptree* member, Ptree* arglist)
{
    return Class::TranslateMemberCall(env, WrapperName(member),
                                       arglist);
}
```

Note that the second argument to `TranslateMemberCall()` supplied by `Class` is not the name of the wrapped function, that is, `member`, but the name of the wrapper function. Thus a member function call for a wrapper-class object:

```
p.Move(3, 5)
```

is translated into this expression:

```
p.wrapper_Move(3, 5)
```

6.4 Implementation of `qMake()`

The OpenC++ MOP provides a member function `Ptree::qMake()` (quoted `make`), which constructs a `Ptree` metaobject according to the given format. Although this member function is more convenient than a similar member function `Make()`, it cannot be implemented within regular C++, but requires meta-level programming to be implemented. Without meta-level programming, only `Make()` is available.

The implementation of `qMake()` is an example of meta-meta level programming in OpenC++. The OpenC++ MOP uses itself to implement metaobjects and their member functions such as `qMake()`, so that the OpenC++ MOP provides better abstractions and programming interface for programmers to write metaclasses. Through this example, we present that OpenC++ can naturally deal with meta-meta level programming and it actually uses meta-meta level programming for implementing abstractions that make it easier to write metaclasses.

What the user program should look like

Recall the usage of `qMake()`. If a variable `tmp` is a pointer to a `Ptree` metaobject representing a variable name `xyz`, and a variable `i` is an integer `3`, then programmers may write something like this:

```
Ptree* exp = Ptree::qMake("int 'tmp' = 'i';");
```

This program constructs a `Ptree` metaobject “`int xyz = 3`”. The expressions surrounded by back-quotes are expanded when `qMake()` is invoked.

The program shown above can be rewritten into a program using `Make()`:

```
Ptree* exp = Ptree::Make("int %p = %d", tmp, i);
```

Unlike `qMake()`, `Make()` takes a format and some parameters, which are substituted for the occurrences of `%p` and `%d` in the format. This kind of programming interface is popular in C and C++, but it becomes error-prone as the number of parameters increases. Typical errors caused by this interface are to give a wrong number of parameters and to place parameters in a wrong order.

What the user program should be translated into

Function calls of `qMake()` are translated into a combination of several function calls. For example, this program:

```
Ptree* exp = Ptree::qMake("int 'tmp' = 'i';");
```

is translated into:

```
Ptree* exp = (Ptree*)(PtreeHead()+"int "+tmp+" = "+i+");
```

Note that the program after the translation is in regular C++. For example, the character string “`tmp`” becomes a variable name `tmp` after the translation. Since regular C++ cannot convert a character string to a variable name at runtime, the conversion should be done by the translation at compile time.

`PtreeHead()` returns a `PtreeHead` object, which is a stream object producing a `Ptree` metaobject. The `+` operator is used to input a character string, a `Ptree` metaobject, and so on, to the stream object. The inputted data are concatenated into a `Ptree` metaobject, and the concatenated `Ptree` metaobject is obtained by explicitly casting the `PtreeHead` object into the type `Ptree*`. The cast operator is overloaded to return the concatenated `Ptree` metaobject.

Write a meta-level program

Writing a metaclass for performing the translation mentioned above is quite straightforward. To do this translation, the metaclass `QuoteClass` just overrides a member function `TranslateMemberCall()`:

```

Ptree* QuoteClass::TranslateMemberCall(Environment* env,
                                       Ptree* member, Ptree* args)
{
    Ptree* name = SimpleName(member);
    char* str;

    if(Ptree::Eq(name, "qMake")){
        Ptree* arg1 = Ptree::First(Ptree::Second(args));
        if(arg1->Reify(str) && str != nil)
            return ProcessBackQuote(FALSE, str);
        else
            ErrorMessage("bad argument for qMake()", arg1);
    }
    else
        return Class::TranslateMemberCall(env, member, args);
}

```

This member function translates a given function-call expression if the called function is `qMake()`, otherwise it delegates the translation to the member function supplied by the base class `Class`.

Since the argument `member` may be not a simple member name but a qualified name such as `Ptree::qMake()`, `TranslateMemberCall()` first calls `SimpleName()` to strip the class name and double colons off:

```

Ptree* QuoteClass::SimpleName(Ptree* qualified_name)
{
    if(qualified_name->IsLeaf())
        return qualified_name;
    else
        return Ptree::First(Ptree::Last(qualified_name));
}

```

After that, `TranslateMemberCall()` determines whether the member name is `qMake`, and if so, it converts the first argument from a `Ptree` metaobject to a character string. This conversion is done by calling `Reify()` for `arg1`. Then `TranslateMemberCall()` calls `ProcessBackQuote()` with the converted character string to translate the member-call expression. `ProcessBackQuote()` is a member function of `QuoteClass`.

6.5 Metaclass

Only the implementation of `qMake()` is not an example of use of meta-meta level programming. The default metaclass `Class` and its subclasses are also implemented with using meta-meta level programming, so that programmers

can easily define a new metaclass. For this reason, the metaclass `Class` and its subclasses are also class metaobjects, and they are instances of the metaclass `Metaclass`. Figure 6.1 shows this instance-of relationship.

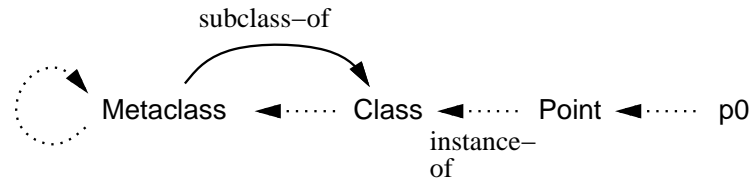


Figure 6.1: The instance-of relationship among metaclasses

Protocol without meta-meta level programming

The definitions of metaclasses seen so far have not explicitly showed all the protocol that the metaclasses must obey. They need to be translated by the metaclass `Metaclass` so that they satisfy all the protocol.

To obey all the protocol, (i) a new metaclass has to have a member function `MetaclassName()`, and (ii) a function that instantiates the metaclass must be registered. For example, a new metaclass `MyClass` should be translated into something more complex than what we have seen:

```

class MyClass : public Class {
public:
    MyClass(Ptree* d, Ptree* m) : Class(d, m) {}
    :
    char* MetaclassName() { return "MyClass"; }
};

static Class* CreateMyClass(Ptree* d, Ptree* m)
{
    return new MyClass(d, m);
}

static ListOfMetaclass myClassObject("MyClass", CreateMyClass,
                                     MyClass::Initialize());
  
```

After the translation, a member function `MetaclassName()`, a function `CreateMyClass()`, and an object `myClassObject` are inserted. `CreateMyClass()` is a function that instantiates the metaclass, and the object `myClassObject` is created at the beginning of runtime and registers `CreateMyClass()`. The registered function is used to implement a function that

receives a class name in the form of character string and instantiates the specified class. This function is internally used by the OpenC++ compiler when instantiating a metaclass, because the `new` operator does not take a character string to specify the instantiated class:

```
class Point { ... };
Point* p0 = new Point;           // valid
char* c = "Point";
Point* p1 = new c;              // invalid
```

Note that `new Point` is a valid expression but `new c` is not since `c` is not a class name but a variable.

The behavior implemented by the inserted functions and variable cannot be inherited from the base class; it must be explicitly implemented by every new metaclass. This is because implementing the behavior needs the definition of the new metaclass as the example of named object that we have shown in Chapter 2.

Protocol with meta-meta level programming

Since the definition obeying all the protocol is complex and error prone, the actual OpenC++ MOP has meta-level programmers write a simpler definition of a metaclass, and automatically translates it into the regular definition that obeys all the protocol. To do this, the OpenC++ MOP provides a metaclass `Metaclass`, which is the metaclass of all metaclasses.

Like other metaclasses, the metaclass `Metaclass` is a subclass of `Class`. It overrides `TranslateSelf()` and `TranslateBody()`:

```
Ptree* Metaclass::TranslateSelf(Environment* env)
{
    Ptree* name = Name();
    Ptree* tmpname = Ptree::GenSym();

    AppendAfterToplevel(Ptree::Make(
        "static Class* %p(Ptree* def, Ptree* marg){\n"
        "    return new %p(def, marg); }\n"
        "static ListOfMetaclass %p(\"%p\", %p,\n"
        "    %p::Initialize());\n",
        tmpname, name,
        Ptree::GenSym(), name, tmpname, name));

    return Class::TranslateSelf(env);
}

Ptree* Metaclass::TranslateBody(Environment* env, Ptree* body)
```

```

{
    Ptree* mem = Ptree::Make("public: char* MetaclassName() {\n"
        "    return \"%p\"; }",
        Name());
    return Class::TranslateBody(env, Ptree::Append(body, mem));
}

```

The member function `TranslateSelf()` inserts a function and a variable for instantiating the metaclass, and `TranslateBody()` inserts a member function `MetaclassName()` in the declaration of the metaclass. For the reason of bootstrapping, `Metaclass` uses not `Ptree::qMake()` but `Ptree::Make()`.

Since a subclass inherits the metaclass from the base class, programmers do not need to explicitly write a metaclass declaration for new metaclasses. The OpenC++ compiler automatically selects `Metaclass` for the new metaclasses since they are subclasses of `Class`. No ad-hoc implementation is required; the OpenC++ MOP can naturally deal with this and programmers can enjoy a simpler protocol by this mechanism.

6.6 Vector Library

All the examples shown above are of abstractions that regular C++ cannot handle but the OpenC++ MOP can do. The OpenC++ MOP also makes some kinds of abstractions more efficient than in regular C++. From this section, we show a few examples of such abstractions.

The first example is the vector library. In Chapter 2, we showed that the template mechanism of C++ enables a vector abstraction for any type, but the implementation with the template mechanism was not as efficient as an ideal implementation. If the OpenC++ MOP is used, however, the vector abstraction is implemented more efficiently.

Vector library in regular C++

As we showed in Chapter 2, a vector abstraction is implemented in regular C++ by the following template:

```

template <class T> class Vector {
    T elements[SIZE];
public:
    Vector operator + (Vector& a, Vector& b) {
        Vector c;
        for(i = 0; i < SIZE; ++i)
            c.elements[i] = a.elements[i] + b.elements[i];
    }
};

```



```

Ptree* index = Ptree::GenSym();
return Ptree::qMakeStatement(
    "for(int 'index' = 0;'index' < SIZE; ++'index')\n"
    "  'object'.element['index']"
    "  = 'Inline(env, expr, index)';\n");
}

```

The member function `TranslateAssign()` translates an assignment expression such as `=` and `+=`. It translates an assignment expression into an efficient for loop if the operator is `=`. The right-side expression is passed to a member function `Inline()` and translated into an appropriate expression:

```

Ptree* VectorClass::Inline(Environment* env, Ptree* expr,
                          Ptree* index)
{
    Ptree* lexpr;
    Ptree* rexpr;

    if(expr->IsLeaf())
        return Ptree::qMake("'expr'.element['index']");
    else if(Ptree::Match(expr, "[%? + %?]", &lexpr, &rexpr))
        return Ptree::qMake("'Inline(env, lexpr, index)'"
                              "+ 'Inline(env, rexpr, index)'"");
    else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
        return Ptree::qMake("'Inline(env, lexpr, index)'"
                              "- 'Inline(env, rexpr, index)'"");
    else if(Ptree::Match(expr, "[ ( %? ) ]", &lexpr))
        return Ptree::qMake("'Inline(env, lexpr, index)'"");
    else if(Ptree::Match(expr, "[ - %? ]", &lexpr))
        return Ptree::qMake("-'Inline(env, lexpr, index)'"");
    else if(Ptree::Match(expr, "[%? * %?]", &lexpr, &rexpr))
        return Ptree::qMake(
            "'lexpr' * 'Inline(env, rexpr, index)'"");
    else{
        ErrorMessage("invalid vector expression", expr);
        return nil;
    }
}
}

```

The member function `Inline()` tests whether the given expression matches a pattern and, if it matches, calls `Inline()` recursively to process the sub-expressions. `Inline()` can deal with not only the `+` operator but also the `-` and `*` operators and parentheses `()`.

Experiments

The metaclass `VectorClass` translates an assignment expression on a vector into an efficient for loop. For example, this program:


```

Vector<double> v1, v2, v3, v4;
    ⋮
v1 = v2 + v3 + v4;

```

is translated into this:

```

Vector<double> v1, v2, v3, v4;
    ⋮
for(i = 0; i < SIZE; ++i)
    v1.elements[i] = v2.elements[i] + v3.elements[i]
                    + v4.element[i];

```

This translation drastically improves execution performance. To show the improvement, we ran a micro benchmark program and measured execution time of `Vector` expressions. The micro benchmark program computes the sum of various numbers of vectors of `double`. First, we ran the program with the vector library written in regular C++, then ran the same program with the vector library written in OpenC++. Since different compilers perform different optimization techniques, we used GNU C++ 2.7.2.1 (with option `-O3`) and Sun C++ 3.0.1 (with option `-fast`) for regular C++. The OpenC++ compiler uses GNU C++ for the backend compiler. Moreover, we ran a program that is equivalent to the micro benchmark but optimized by hand without the vector library. This hand-coded program uses arrays of `double` instead of objects and was compiled by GNU C++. When measuring the execution time, we changed the length of each vector between 8 and 64, and also the number of the vectors in the assigned expression from 1 to 4. When the number of the vectors is 1, the expression is `v0 = v1`; No addition is executed. All the benchmark programs used in this experiment are in Appendix C.3.

The results of the experiment are listed in Table 6.1. The vector library implemented with the OpenC++ MOP achieved as good performance as the hand-coded program. Although the OpenC++ program is slightly slower than the hand-coded program when the number of vectors is 1 and the vector length is 64, this fact is caused by the implementation difference in copying a vector. The hand-coded program copies a vector by explicitly copying each element, but the OpenC++ program copies it by using the default object copy mechanism, which is compiled into a call of `memcpy()`. The implementation strategy of copying a vector is also the reason that Sun C++ achieved the best performance when the number of vectors is 1 and the length of each vector is 64. The Sun C++ compiler inlines `memcpy()` when an object is copied.

Table 6.1: Execution performance of the vector library ($\mu\text{sec.}$)

# of vectors	length 8				length 64			
	1	2	3	4	1	2	3	4
Sun C++	0.5	1.5	3.0	4.4	3.3	10.1	20.0	30.1
GNU C++	0.3	1.7	3.1	4.5	6.9	21.7	36.2	51.1
OpenC++	0.3	0.9	1.3	1.7	6.9	6.5	9.8	13.1
Hand-coded	0.9	0.9	1.3	1.7	6.5	6.5	9.8	13.0

Average of 1,000,000 (size 8) or 300,000 (size 64) iterations.

SPARC Station 20/61, SunOS 4.1.3

This experiment also shows that real C++ compilers do not perform the optimization that the metaclass `VectorClass` performs. Although an ideal compiler should automatically perform the optimization, it seems difficult for real compilers to do that within reasonable space and time. Our claim is that such an optimization should be done by a metaclass rather than a compiler's optimizer. Such an optimization is difficult for a general-purpose optimizer as our experiment showed, but on the other hand, it is not difficult for a metaclass because the metaclass is written by the library developer, who knows semantic information about the library code. We believe that a compiler's optimizer should focus on general optimizations and metaclasses should perform a special optimization that is effective only for a particular class. Although forcing end-programmers to write a metaclass is not realistic, we believe that it is acceptable that library programmers write metaclasses since they are usually experienced programmers and their code is reused by many end programmers.

6.7 The Standard Template Library

The Standard Template Library (STL) [44] is another example of abstractions that The OpenC++ MOP can implement more efficiently. STL is a C++ library included by the ANSI standard of C++. Despite of the high-level abstractions that STL provides, a program using STL is often slower than an equivalent program written without STL. The OpenC++ MOP contributes to avoid performance drawbacks caused by STL with keeping

its high-level abstractions.

Brief Overview of STL

A unique feature of STL is that the library consists of independent components and the users can flexibly combine the components to obtain the functionality they need. The main components of STL are containers and generic algorithms. The containers are objects that store a collection of other objects, and the generic algorithms are functions that process containers. Since type names are parameterized with using the template mechanism, STL users may use a generic algorithm with any kind of containers. They do not have to use a different version of the generic algorithm for a different kind of containers.

For example, STL users may write something like this:

```
list<double> a1 = ... ;
set<double> a2 = ... ;
n1 = count(a1.begin(), a1.end(), 3.14);
n2 = count(a2.begin(), a2.end(), 3.14);
```

This program computes the number of 3.14 stored in containers `a1` and `a2`, respectively. `a1` is a `list` container and `a2` is a `set` container. `count` is a generic algorithm to determine the number of elements in a container that are equal to a given value. It takes pointers to the first and the last element in the container and the value that it counts the number of. Note that the same generic algorithm `count` is used for two different kinds of containers `list` and `set`. The single generic algorithm serves all kinds of containers.

The connectivity between containers and generic algorithms is enabled by another kind of STL component called iterators. In the program above, iterators are the values returned by `a1.begin()` and `a1.end()`. They are pointer-like components that all kinds of containers provide as common interface to access the elements. Generic algorithms use the iterators to traverse elements stored in a container. For example, iterators for `list` containers are defined as follows:

```
class iterator {
public:
    iterator(list<T>* p) { ptr = p; }
    list<T>* ptr;
    int eof() { return ptr == 0; }
    int operator != (iterator& a) { return ptr != a.ptr; }
    T operator * () { return ptr->value; }
    iterator& operator ++ () { ptr = ptr->next; return *this; }
```

```

        iterator operator ++ (int) {
            iterator prev = *this;
            ptr = ptr->next;
            return prev;
        }
};

```

Iterators are objects for which pointer operators such as `*` and `++` are overloaded. Generic algorithms use the iterators as if they are C++ pointers to arrays; for example, the next template function is an implementation of the generic count algorithm:

```

template <class I, class T>
int count(I first, I last, T value)
{
    int n = 0;
    while(first != last)
        if(*first++ == value)
            ++n;

    return n;
}

```

The template argument `I` is the type of iterators and `T` is the type of container elements. Note that the variables `first` and `last` are used as if they are pointers to an array of the type `T`.

Performance improvement by the OpenC++ MOP

Although iterators give great flexibility to STL, they also involve performance drawbacks if compared with an equivalent program written without iterators. Since a generic algorithm must indirectly access elements in a container through an iterator, its execution performance tends to be slower. If programmers give up generality of the generic algorithm and specialize the algorithm to work only for a particular kind of container, then the specialized algorithm will be more efficient because iterators are not needed any more. For example, the following function is a specialized count algorithm for counting elements only in a `list<int>` container:

```

int count(List<int>* first, List<int>* last, int value)
{
    int n = 0;
    while(first != last){
        if(first->value == value)
            ++n;
    }
}

```

```

        first = first->next;
    }

    return n;
}

```

Note that now the variables `first` and `last` are not iterators but actual pointers to `list<int>` objects. Hence reading an element is done by the `->` operator instead of the `*` operator. Also, the `++` operator is replaced with the expression `first = first->next`.

The OpenC++ MOP reduces the overheads by iterators. In OpenC++, the STL implementor can write a metaclass that specializes a generic algorithm for a particular kind of container and translates a program to use that specialized algorithm. Suppose that a program uses the generic `count` algorithm with a `list<int>` container. The metaclass converts iterators for `list<int>` into actual pointers to `list<int>` and it replaces calls of the generic `count` algorithm with calls of a `count` algorithm specialized for `list<int>`. First, the definition of iterators for `list` containers:

```
class iterator { ... };
```

is translated from a class type into this pointer type:

```
typedef list<T>* iterator;
```

Then the template function shown below is derived as a specialized algorithm from the generic `count` algorithm. It is substituted for the generic `count` algorithm called with `list` containers:

```

template <class I, class T>
int count_for_list_int(I first, I last, T value)
{
    list<int>* tmp;
    int n = 0;
    while(first != last)
        if((tmp=first,first=first->next,tmp)->value == value)
            ++n;

    return n;
}

```

This template function supposes that `I` is bound to `list<int>*`.

The translation mentioned above is easily implemented with about 40 lines of meta-level program. The metaclass for this translation is effective

not only for the generic `count` algorithm. It deals with combinations of `list` containers and any generic algorithm. For the complete definition of the metaclass, see in Appendix C.4. We also show a metaclass for `set` containers in the same place.

Experiment

To illustrate the performance improvement by the metaclasses, we measured the execution time of the generic `count` algorithm with/without the metaclasses. As for the containers, we used `list<int>` and `set<int>`. When the metaclasses were not used, the measured program was compiled by GNU C++ 2.7.2 with option `-O3` and Sun C++ 4.1 with option `-fast`. When the metaclasses are used, the program was compiled by the OpenC++ compiler, which uses GNU C++ for the backend compiler. Also, we measured the execution time of hand-optimized `count` algorithms for `list<int>` and `set<int>`. These algorithms were compiled by GNU C++. All the programs used in this experiment are shown in Appendix C.4.

The results of this experiment is listed in Table 6.2. As for `list` containers, the generic `count` algorithm involves 73% (Sun C++) or 36% (GNU C++) overheads against the hand-optimized version, but with the OpenC++ MOP, the overheads are reduced to only 9%. As for `set` containers, the generic `count` algorithm involves 39% (Sun C++) or 21% (GNU C++) overheads. But the OpenC++ MOP reduces the overheads to 10%. These results show that generic algorithms of STL work for any kind of containers but this adaptability causes serious performance degradation. The OpenC++ MOP recovers this performance degradation from one half to one fourth while keeping the adaptability of STL.

6.8 The OOPACK Benchmark Test

The last example is the OOPACK benchmark test [49]. This benchmark test is a program for testing the ability of a C++ compiler for compiling a program written in object-oriented programming (OOP) as efficiently as a program in non-OOP. The program contains a suite of tests, each of which consists of two equivalent routines written in OOP and non-OOP. The OOP routines are written with higher-level abstractions, whereas the non-OOP routines are written in C style for efficiency. In other words, the non-OOP routines are hand-optimized versions of the corresponding OOP routines.

If the OOP routines are slower than the non-OOP routines, that performance degradation means costs to use higher-level abstractions implemented

Table 6.2: Execution performance of STL (msec.)

	<code>list</code> (ratio)	<code>set</code> (ratio)
Sun C++	57 (1.73)	860 (1.39)
GNU C++	45 (1.36)	750 (1.21)
OpenC++	36 (1.09)	680 (1.10)
Hand-coded	33 (1.00)	620 (1.00)

Average of 100 (`list`) or 10 (`set`) iterations.

SPARC Station 20/514, Solaris 5.3

with objects under the compiler. A C++ compiler should be able to compile the OOP routines as efficient as the non-OOP routines since it can in principle transform the OOP routines into the non-OOP routines by inlining member functions and performing constant propagation and strength reduction.

Improvement with the OpenC++ MOP

Real C++ compilers have difficulty in compiling the OOP routines efficiently as we show later, but this inefficiency is fairly recovered by the OpenC++ MOP. Although the OpenC++ MOP cannot improve the execution speed without changing the benchmark program, it can extend the language syntax to allow the programmer to write more efficient OOP code than the OOP routines in the original benchmark program. The extended syntax is used for annotating compilation hints without directly describing lower-level details of the implementation. It does not affect the level of abstraction of the OOP routines.

For example, the following code is the `Matrix` test in the benchmark. It computes multiplication of two matrices in OOP style:

```
void MatrixBenchmark::oop_style() const
{
    Matrix c(L, L, C);
    Matrix d(L, L, D);
    Matrix e(L, L, E);
    for(int i = 0; i < e.rows; i++)
        for(int j = 0; j < e.cols; j++){
```

```

        double sum = 0;
        for(int k=0; k<e.cols; k++)
            sum += c(i,k) * d(k,j);

        e(i,j) = sum;
    }
}

```

The OpenC++ MOP extends the syntax to make a `foreach` statement available for this test and improve the execution performance with the new statement. The next is the code rewritten using the `foreach` statement:

```

void MatrixBenchmark::oop_style() const
{
    Matrix c(L, L, C);
    Matrix d(L, L, D);
    Matrix e(L, L, E);

    c.foreach(i){
        for(int j = 0; j < e.cols; ++j){
            double sum = 0;
            d.foreach(k){
                sum += c(k) * d(j);
            };

            e(i,j) = sum;
        }
    };
}

```

Here, `c.foreach(i)` means iterating the following block statement for each row of the matrix `c`. `i` is bound to the index of the row currently processed. In the block statement, `c(k)` indicates the `k`-th column in the row. Unlike the `for` statement, the `foreach` statement explicitly indicates that the loop is executed for traversing the rows of a matrix, so that the statement can be translated into optimized code for the traversing. In fact, the metaclass for `Matrix` translates the program above into this efficient one:

```

void MatrixBenchmark::oop_style() const
{
    Matrix c(L, L, C);
    Matrix d(L, L, D);
    Matrix e(L, L, E);

    for(int i=c.rows, t1=c.cols, t2=(c.rows-1)*c.cols;
        --i >= 0;
        t2 -= t1)

```



```

    {
        double const* t3 = &(c.Data())[t2];
        for(int j = 0; j < e.cols; ++j){
            double sum = 0;
            for(int k=d.rows, t4=d.cols, t5=(d.rows-1)*d.cols;
                --k >= 0;
                t5 -= t4)
            {
                double const* t6 = &(d.Data())[t5];
                sum += t3[k] * t6[j];
            };
            e(i,j) = sum;
        }
    };
}

```

Note that the `foreach` statements are translated into `for` statements.

The second test in the benchmark program is `Iterator`. It computes dot-product of two vectors implemented by arrays of `double`. The `Iterator` objects in this test is similar to STL's iterator, but they also contain the length of a vector and provide a member function `done()` to check whether there are no more elements:

```

void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    for(Iterator ai(A,N), bi(B,N);
        !ai.done();
        ai.next(), bi.next())
    {
        sum += ai.look() * bi.look();
    }
    IteratorResult = sum;
}

```

Here, `A` and `B` are arrays of `double`, and `N` is the length of the arrays.

The OpenC++ MOP provides a statement `foreach` for the class `Iterator`. With the `foreach` statement, the program above is rewritten into this:

```

void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    Iterator ai(A,N), bi(B,N);
    ai.foreach(v){
        sum += v * bi.look();
    }
}

```

```

        bi.next();
    };

    IteratorResult = sum;
}

```

The `foreach` statement for `Iterator` provides a control abstraction similar to `foreach` for `Matrix`. It iterates the following block statement for each element of the vector. In the block statement, a specified variable (`v` in the program above) indicates the element currently processed. However, the `foreach` statement for `Iterator` is translated differently way from `Matrix`. This is `oop_style()` after the translation:

```

void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    Iterator ai(A,N), bi(B,N);
    for(int t7 = 0, t8 = ai.Limit(); t7 < t8; ++t7){
        const double& v = ai.Array(t7);
        sum += v * bi.look();
        bi.next();
    };

    IteratorResult = sum;
}

```

The length of the vector that `ai` points to is stored in a local variable before starting iteration. This eliminates accesses to `ai` when the loop-termination condition is checked.

The last test in the benchmark is `Complex`. This test computes a complex-valued “SAXPY” operation, and measures how efficiently a C++ compiler compiles a `Complex` object, which represents a complex number and is very popular in scientific computing:

```

void ComplexBenchmark::oop_style() const
{
    Complex factor(0.5, 0.86602540378443864676);
    for(int k = 0; k < N; k++)
        Y[k] = Y[k] + factor * X[k];
}

```

Here, `X` and `Y` are arrays of `Complex` objects.

The execution performance of this test can be improved by a similar technique that we showed for the vector library in Section 6.6. No extended syntax is needed. The program shown above is translated by a metaclass into this:

```
void ComplexBenchmark::oop_style() const
{
    double factor_re = 0.5;
    double factor_im = 0.86602540378443864676;
    for(int k = 0; k < N; k++){
        Y[k].re = Y[k].re+factor_re*X[k].re-factor_im*X[k].im;
        Y[k].im = Y[k].im+factor_re*X[k].im+factor_im*X[k].re;
    }
}
```

Note that a `Complex` object `factor` is broken down into two `double` variables, `factor_re` and `factor_im`. This leads a C++ compiler to allocate `factor` on registers rather than a stack frame, and eventually contributes to performance improvement. Without this translation, C++ compilers such as GNU C++ and Sun C++ do not allocate objects on registers even though allocating them is possible in principle.

Experiments

We measured execution time of the OOPACK benchmark test under different settings. We first ran the benchmark program compiled by Sun C++ 4.1 with `-fast` option, and then the program compiled by GNU C++ 2.7.2 with `-O3` option. We also compiled the program by the OpenC++ compiler with metaclasses for the translation shown above, and measured the execution time of the compiled code. When compiling with those metaclasses, the benchmark program was rewritten to utilize the extended syntax. The OpenC++ compiler used GNU C++ 2.7.2 with option `-O3` for the back-end compiler. All the programs used for the experiment are presented in Appendix C.5.

The results of the experiment are listed in Table 6.3. The OOPACK benchmark consists of four tests: `Max`, `Matrix`, `Complex`, and `Iterator`. We did not develop a metaclass for the `Max` test, which is for measuring how well a C++ compiler inlines a function. But for the remaining three tests, OpenC++ showed better performance than Sun C++ and GNU C++. If OpenC++ is used, the OOP routines involves only 10% to 20% overheads against the non-OOP routines written in C style. On the other hand, the OOP routines compiled by Sun C++ is twice or three times slower than the non-OOP routines. The OOP routines compiled by GNU C++ also involves 60% or 20% overheads for `Complex` and `Iterator`, but the `Matrix` test is more than 8 times slower.

Table 6.3: Execution time of the OOPACK benchmark (sec.)

		Sun C++	GNU C++	OpenC++
Max	(C-style)	8.1	9.0	9.0
	(OOP)	8.5	12.0	12.0
	Ratio	1.1	1.3	1.3
Matrix	(C-style)	10.9	9.8	9.8
	(OOP)	31.3	80.7	11.2
	Ratio	2.9	8.2	1.1
Complex	(C-style)	13.2	11.2	11.2
	(OOP)	23.3	18.2	13.1
	Ratio	1.8	1.6	1.2
Iterator	(C-style)	7.1	7.1	7.1
	(OOP)	15.2	8.2	8.1
	Ratio	2.1	1.2	1.1

50000 (Max), 500 (Matrix), 20000 (Complex), 50000 (Iterator) iterations.
SPARC Station 20/514, Solaris 5.3

6.9 Summary

This chapter presented eight examples of libraries that the OpenC++ MOP enables. The first two examples, named `objects` and `distributed objects`, presented useful data abstractions that regular C++ cannot handle. Through the examples, we also illustrated how various metaobjects like `TypeInfo` are used in meta-level programming.

The next example, wrapper library, is an example of metaclass libraries. Implementing abstractions like named objects and distributed objects is facilitated if there is a metaclass library which provides typical meta code for metaclass writers. This example showed a metaclass `WrapperClass`, which helps programmers write a metaclass handling wrapper functions.

`qMake()` and `MetaClass` are examples of meta-meta level programming. The OpenC++ MOP naturally allows meta-meta level programming in order to make useful abstractions available at meta level as well as base level. `qMake()` is a meta-level function providing convenient interface for metaclass writers. Its interface cannot be implemented without the OpenC++

MOP. `Metaclass` is a meta-metaclass for all metaclasses. It simplifies the protocol for writing a new metaclass.

The remaining three examples showed that the OpenC++ MOP can be used to improve execution performance of some kinds of abstractions. The OpenC++ MOP makes it possible to specialize library code for user code and improve execution performance. This specialization includes elimination of unnecessary indirection and encapsulation and it helps C++ compilers to generate more efficient object code. Furthermore, the OpenC++ MOP allows syntax extension, which is used for putting annotations for efficient compilation. In the example of the OOPACK benchmark, we showed that such syntax extensions actually improve execution performance while keeping the level of abstraction.

Chapter 7

Conclusion

This thesis has discussed the OpenC++ MOP, our new language mechanism for writing better libraries. This mechanism enables better C++ libraries and will contribute to rapid and low-cost software development, which is one of major issues in today's software industry.

Contributions

Direct contributions by this thesis are summarized as follows:

- This thesis proposed a new language mechanism for pre-processing a program in a context-sensitive and non-local way. This mechanism allows a library to instantiate specialized code depending on how the library is used and substitute it for the original user code.
- Then this thesis presented that the proposed mechanism makes it possible to write better C++ libraries than in regular C++. For being high level and easy to use, the implementation of some kinds of useful control/data abstractions requires the ability to instantiate specialized code for the user code. The proposed mechanism provides that ability for libraries to include such abstractions.
- Also, this thesis showed that the proposed mechanism improves efficiency of C++ libraries. Some kinds of control/data abstractions are difficult to implement efficiently because the ability of C++ compilers to optimize the library code is limited due to time and space. The proposed mechanism allows library developers to specify specialization of the library code so that the implemented abstractions run efficiently.

The library developers can specify the specialization assuming optimization performed by a backend C++ compiler. This is an approach integrating general optimization by a compiler and ad-hoc optimization that is apparent to programmers.

- Compared with other reflective languages, a unique feature of the proposed mechanism is that it is a compile-time MOP. It exploits static types to run metaobjects only at compile time. This means that the proposed mechanism does not imply any runtime penalties due to dispatching to metaobjects.
- Furthermore, this thesis proposed the meta helix architecture, which is an improved version of the metacircular architecture of the CLOS MOP. It fixes a problem we call implementation level conflation, which is involved by the CLOS MOP, while keeping benefits of metacircularity — ease of learning and ease of writing an efficient meta program.

These contributions suggest a new design approach for programming languages. The proposed mechanism makes it feasible for language designers to keep a language simple and consistent and to implement most of desirable control/data abstractions as a “language extension” library. Keeping a core language simple and consistent has a number of advantages; especially, a simple and consistent language is easy for programmers to learn and for compiler implementers to develop an optimizing compiler.

Designing a simple and consistent language has been had a few disadvantages. First, a number of language extensions have been impossible to implement as a library and, second, language extensions provided by a library have been less efficient than ones implemented as built-in features. The latter disadvantage can be solved if compilers support a special optimization technique for the library, but this solution takes long time since compilers do not support it until the extensions are accepted and deployed. The proposed mechanism solves these disadvantages and motivates language designers to avoid a rich and fat language and design a simple and consistent language.

Future directions

Possible future directions of this study are followings.

Apply to other languages

The idea of the OpenC++ MOP will be applicable to other static-typed languages such as Java [27]. Although Java is still a simple and small language, it will be getting complicated and difficult to understand as it is widely used for developing real applications, because real programmers tend to desire richer language mechanisms. This is also the same path that other major languages such as Fortran [1], C [35], and Lisp [56], have followed. But if Java has a mechanism like the OpenC++ MOP, it will be able to avoid following the undesirable path. Although reflective mechanisms for Java have been already proposed, their capability is limited since, like RTTI of C++, they support only introspection about the classes and the objects [55].

Make meta representation richer

Unfortunately, the OpenC++ MOP does not enable all kinds of desirable control/data abstractions because the meta representation provided by the MOP is limited. For example, the OpenC++ MOP does not include metaobjects representing control and data flow of the processed program. The lack of this information makes it difficult to efficiently implement some kinds of abstractions. The OpenC++ MOP should be enhanced to support such metaobjects.

*Real programmers program in C++ or C,
Real programmers demand efficiency,
Real programmers are never happy with existing languages.*

— Anon, *A Metaobject Protocol for Real Programmers*

Bibliography

- [1] American National Standards Institute, Inc., *American National Standard Programming Language FORTRAN*, 1978. ANSI X3.9-1978.
- [2] Asai, K., S. Matsuoka, and A. Yonezawa, “Duplication and Partial Evaluation,” *Lisp and Symbolic Computation*, vol. 9, pp. 203–241, 1996.
- [3] Bobrow, D. and T. Winograd, “An Overview of KRL, A Knowledge Representation Language,” *Cognitive Science*, vol. 1, no. 1, pp. 1–46, 1977.
- [4] Bobrow, D. G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, “CommonLoops, Merging Lisp and Object-Oriented Programming,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 17–29, ACM, September 1986.
- [5] Bobrow, D. G. and M. Stefik, *The LOOPS Manual*. Xerox PARC, December 1983.
- [6] Brandt, S. and R. W. Schmidt, “The Design of a Meta-Level Architecture for the BETA Language,” in *Proc. of Ecoop Workshop in Advances in Metaobject Protocol and Reflection (META’95)*, 1995.
- [7] Bretthauer, H., H. Davis, J. Kopp, and K. Playford, “Balancing the EuLisp Metaobject Protocol,” in *Proc. of the Int’l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 113–118, 1992.
- [8] Briot, J. and P. Cointe, “Programming with Explicit Metaclasses in Smalltalk-80,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 419–431, ACM, October 1989.

- [9] Buschmann, F., K. Kiefer, F. Paulisch, and M. Stal, “The Meta-Information-Protocol: Run-Time Type Information for C++,” in *Proc. of the Int’l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 82–87, 1992.
- [10] Cannon, H. I., “Flavors: A Non-Hierarchical Approach to Object-Oriented Programming,” 1982.
- [11] Chiba, S., “A Metaobject Protocol for C++,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, no. 10 in Sigplan Notices vol. 30, pp. 285–299, ACM, 1995.
- [12] Chiba, S., “OpenC++ Programmer’s Guide for Version 2,” Technical Report SPL-96-024, Xerox PARC, 1996.
- [13] Chiba, S. and T. Masuda, “Designing an Extensible Distributed Language with a Meta-Level Architecture,” in *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pp. 482–501, Springer-Verlag, 1993.
- [14] Chiba, S. and T. Masuda, “Open C++ and Its Optimization (Extended Abstract),” in *Proc. of OOPSLA ’93 Workshop on Reflection and Meta-level Architectures*, Oct. 1993.
- [15] Chiba, S., G. Kiczales, and J. Lamping, “Avoiding Confusion in Metacircularity: The Meta-Helix,” in *Proc. of the 2nd Int’l Symp. on Object Technologies for Advanced Software (ISOTAS)*, LNCS 1049, pp. 157–172, Springer, Mar. 1996.
- [16] Clinger, W. and J. Rees, *Revised⁴ Report on the Algorithmic Language Scheme*, November 1991.
- [17] Cointe, P., “Metaclasses are first class: The ObjVlisp model,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 156–167, 1987.
- [18] Danvy, O. and K. Malmkjær, “Intensions and Extensions in a Reflective Tower,” in *Proc. of ACM Conf. on Lisp and Functional Programming*, pp. 327–341, 1988.
- [19] Ershov, A., “On the Essence of Compilation,” in *Formal Description of Programming Concepts* (E. Neuhold, ed.), pp. 391–420, North-Holland, 1978.

- [20] Fabre, J., V. Nicomette, T. Perennou, R. J. Stroud, and Z. Wu, “Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming,” in *Proc. of the 25th IEEE Symp. on Fault-Tolerant Computing Systems*, pp. 489–498, 1995.
- [21] Ferber, J., “Computational Reflection in Class based Object Oriented Languages,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 317–326, 1989.
- [22] Foote, B. and R. E. Johnson, “Reflective Facilities in Smalltalk-80,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 327–335, 1989.
- [23] Futamura, Y., “Partial Computation of Programs,” in *Proc. of RIMS Symposia on Software Science and Engineering*, no. 147 in LNCS, pp. 1–35, 1982.
- [24] Gallesio, E., “STk Main page.” <http://kaolin.unice.fr/STk>.
- [25] Genesereth, M., R. Greiner, and D. Smith, “MRS manual,” Memo HPP-80-24, Heuristic Programming Project, Stanford University, 1980.
- [26] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [27] Gosling, J., B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.
- [28] Gourhant, Y. and M. Shapiro, “FOG/C++: a Fragmented-Object Generator,” in *Proc. of USENIX C++ Conference*, pp. 63–74, 1990.
- [29] Gowing, B. and V. Cahill, “Making Meta-Object Protocols Practical for Operating Systems,” in *Proc. of 4th International Workshop on Object Orientation in Operating Systems*, (Lund, Sweden), pp. 52–55, Aug. 1995.
- [30] Gowing, B. and V. Cahill, “Meta-Object Protocols for C++: The Iguana Approach,” in *Proc. of Reflection 96*, pp. 137–152, Apr. 1996.
- [31] Hook, J. and T. Sheard, “A Semantics of Compile-time Reflection,” Technical Report 93-019, Dept. of Computer Science and Engineering, Oregon Graduate Institute, Portland, Oregon, 1993.

- [32] Ichisugi, Y., S. Matsuoka, and A. Yonezawa, “RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel,” in *Proc. of the Int’l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 24–35, 1992.
- [33] Ishikawa, Y., “Meta-Level Architecture for Extendable C++,” Technical Report 94024, Real World Computing Partnership, Japan, 1994.
- [34] Ishikawa, Y., A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota, “Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach —,” in *Proc. of Reflection 96*, pp. 153–166, Apr. 1996.
- [35] Kernighan, B. W. and D. Ritchie, *The C Programming Language*. Prentice Hall, 2nd ed., 1988.
- [36] Kiczales, G., J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. The MIT Press, 1991.
- [37] Lamping, J., G. Kiczales, L. Rodriguez, and E. Ruf, “An Architecture for an Open Compiler,” in *Proc. of the Int’l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 95–106, 1992.
- [38] Maes, P., “Concepts and Experiments in Computational Reflection,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 147–155, 1987.
- [39] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa, “Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 300–315, 1995.
- [40] Masuhara, H., S. Matsuoka, T. Watanabe, and A. Yonezawa, “Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 127–144, 1992.
- [41] Matsuoka, S., T. Watanabe, and A. Yonezawa, “Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming,” in *Proc. of European Conf. on Object-Oriented Programming ’91*, no. 512 in LNCS, pp. 231–250, Springer-Verlag, 1991.

- [42] McAffer, J., “Meta-level Programming with CodA,” in *Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, pp. 190–214, Springer-Verlag, 1995.
- [43] Milner, R., M. Tofte, and R. Harper, *The Definition of Standard ML*. The MIT Press, 1990.
- [44] Musser, D. R. and A. Saini, *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.
- [45] Okamura, H. and Y. Ishikawa, “Object Location Control Using Meta-level Programming,” in *Proc. of the 8th European Conference on Object-Oriented Programming*, LNCS 821, pp. 299–319, Springer-Verlag, 1994.
- [46] Okamura, H., Y. Ishikawa, and M. Tokoro, “AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework,” in *Proc. of the Int’l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 36–47, 1992.
- [47] Paepcke, A., “PCLOS: Stress testing CLOS Experiencing the metaobject protocol,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 194–211, 1990.
- [48] Parrington, G. D., “Reliable Distributed Programming in C++: The Arjuna Approach,” in *Proc. of USENIX C++ Conference*, pp. 37–50, 1990.
- [49] Robison, A., “OOPACK: a Benchmark for Comparing OOP vs. C-style programming.” <http://www.kai.com/oopack/oopack.html>, 1995.
- [50] Rodriguez Jr., L. H., “Coarse-Grained Parallelism Using Metaobject Protocols,” Technical Report SSL-91-61, XEROX PARC, Palo Alto, CA, 1991.
- [51] Rodriguez Jr., L. H., “A Study on the Viability of a Production-Quality Metaobject Protocol Based Statically Parallelizing Compiler,” in *Proc. of the Int’l Workshop on Reflection and Meta-Level Architecture* (A. Yonezawa and B. C. Smith, eds.), pp. 107–112, 1992.
- [52] Ruf, E., “Partial Evaluation in Reflective System Implementation,” in *Proc. of OOPSLA’93 Workshop on Reflection and Metalevel Architectures*, 1993.

- [53] Smith, B. C., “Reflection and Semantics in Lisp,” in *Proc. of ACM Symp. on Principles of Programming Languages*, pp. 23–35, 1984.
- [54] Smith, B., “Reflection and Semantics in a Procedural Languages,” Tech. Rep. MIT-TR-272, M.I.T. Laboratory for Computer Science, 1982.
- [55] Soft, J., “Java Core Reflection, API and Specification.” draft paper (9/3/96), 1996.
- [56] Steele, G., *Common Lisp: The Language*. Digital Press, 2nd ed., 1990.
- [57] Stroud, R. J. and Z. Wu, “Using Metaobject Protocols to Implement Atomic Data Types,” in *Proc. of the 9th European Conference on Object-Oriented Programming*, LNCS 952, pp. 168–189, Springer-Verlag, 1995.
- [58] Stroustrup, B., *The C++ Programming Language*. Addison-Wesley, 2nd ed., 1991.
- [59] Stroustrup, B., *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [60] Wand, M. and D. P. Friedman, “The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower,” in *Meta-Level Architectures and Reflection* (P. Maes and D. Nardi, eds.), pp. 111–134, Elsevier Science Publishers B.V., 1988.
- [61] Watanabe, T. and A. Yonezawa, “Reflection in an Object-Oriented Concurrent Language,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 306–315, 1988.
- [62] Weyhrauch, R. W., “Prolegomena to a Theory of Mechanized Formal Reasoning,” *Artificial Intelligence*, vol. 13, no. 1, pp. 133–170, 1980.

Appendix A

Backquote

“backquote” is a convenient mechanism for constructing a list structure in Lisp. This appendix briefly introduces this mechanism for the readers who are not familiar to Lisp.

A.1 Quote

We should start from the “quote” mechanism. It is used to include literal constants in programs. The quotes suppress evaluation; the quoted symbols or expressions are not evaluated:

```
a          => (the value that the symbol a is bound to)
'a         => a   (symbol a)
(+ 1 2)    => 3   (function application)
'(+ 1 2)   => (+ 1 2) (equivalent to (list '+ 1 2))
```

Numerical constants and so on need not be quoted. They evaluate to themselves:

```
735        => 735 (number)
'735       => 735 (number)
```

A.2 Backquote

The backquote (‘) mechanism is similar to the quote mechanism but it allows some sub-expressions to be evaluated in a (back)quoted expression. It evaluates a sub-expression if it follows a comma:

Appendix B

Reference Manual

This document is a programming guide for OpenC++ version 2. It was originally published as a technical report from Xerox PARC [12]. The copyright of this document and the original one belongs to the author of this thesis.

B.1 Introduction

The goal of the OpenC++ project is to make the C++ language extensible. The project started in 1992 at the University of Tokyo in Japan. The first version was released in 1993 as one of the early C++ MOPs (metaobject protocols) and has been used as a research platform at many sites, including University of Newcastle upon Tyne in UK, LAAS in Toulouse, France, Universidade Federal do RGS in Brazil, and so on. After that, the project moved to Xerox Palo Alto Research Center (PARC) in US and joined the Open Implementation Group in 1994. OpenC++ met compile-time MOPs there and OpenC++ Version 2 was developed as the result.

The OpenC++ language enables programmers to extend C++ so that they can use language features that are not available in regular C++. These language features include distribution, persistence, and fault-tolerance. Although they are available in regular C++ with less integrated syntax, for example, OpenC++ enables programmers to use the member function call syntax when accessing a remote object.

Moreover, OpenC++ makes it possible to customize an optimization scheme for a particular class. The OpenC++ compiler manipulates a program at the source code level for optimizing the execution performance. Programmers can customize that program manipulation on a particular class. This benefit got feasible in the version 2 because its MOP is a compile-time

MOP.

This document shows detailed specifications of OpenC++ Version 2. It consists of three parts. First, we give a brief tutorial of programming in OpenC++. It will help the readers get overview of the language. Then, we mention the base-level specifications of OpenC++. The difference from regular C++ is shown here. Last, we present the meta-level specifications, that is, the OpenC++ MOP. It is an interface to deal with the base-level program and customize the language.

B.2 Tutorial

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, 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: pre-processor, 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 B.1.

The meta-level program is written in C++, accurately in OpenC++ because 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 OpenC++ MOP. In the followings, we go through a few examples so that we illustrate how the OpenC++ MOP is used to implement language extensions.

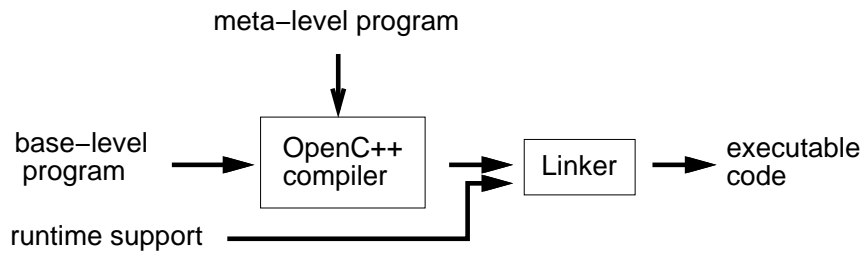


Figure B.1: The OpenC++ Compiler

B.2.1 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.

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 Person : VerboseClass;    // 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.

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 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.

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 Person : VerboseClass;    // 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`:

```
// verbose.cc
#include "mop.h"

class VerboseClass : public Class {
public:
    VerboseClass(Ptree* d, Ptree* m) : Class(d, m) {}
    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, object, 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 key 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.

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. Then, the compiled code is linked with the original compiler (`opencxx.a`)¹ and a new extended compiler is produced. Let's name the extended compiler `myocc`:

```
% occ -- -g -o myocc opencxx.a verbose.cc
```

The options following `--`, such as `-g`, are passed to the back-end C++ compiler. `verbose.cc` is compiled with the `-g` option and linked with `opencxx.a`. The produced file is `myocc` specified by the `-o` option. Unless the `-c` option is given, the OpenC++ compiler produces an executable file.

Next, we compile the base-level program `person.cc` with the extended compiler `myocc`:

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

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

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

¹In the current version, the OpenC++ compiler cannot dynamically load meta-level programs.

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

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

Similarly, 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 Person : VerboseClass [] ;]
[[[class Person [] [{ [
  [public :]
  [[] [Person ( [[[[int] [i]]] )] [{ [
    [age = i] ;]
  ] }]]
  [[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 ...
] ]]]]
%]
```

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.

B.2.2 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.

```
// verbose.cc
class VerboseClass : public Class {
public:
    VerboseClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*,
                               Ptree*, Ptree*);
    static BOOL Initialize();
};

BOOL VerboseClass::Initialize()
{
    RegisterMetaclass("verbose", "VerboseClass");
    return Class::Initialize();
}
```

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

B.2.3 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.

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`.

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.

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.cc

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::MakeStatement(
        "for(int %p = 0; %p < %s * %s; ++%p)\\"
        "    %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. `Ptree::MakeStatement()` is a similar function to `Ptree::Make()`. It constructs a parse tree representing a statement instead of an expression. `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.cc`, 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.

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 Matrix : MatrixClass;
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.

B.2.4 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.cc

Ptree* MatrixClass::TranslateInitializer(Environment* env,
    Ptree* init, PtreeArray& before, PtreeArray& after)
{
    Ptree* sep = Ptree::First(init);
    Ptree* expr = Ptree::Second(init);
    if(sep->Eq('=') && expr->WhatIs() == PtreeBraceId){
        Ptree* tmp = Ptree::GenSym();
        before.Append(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, ... }` following `tmp[]`. If the initializer is an aggregate, this member function translates it as we mentioned above. The temporary array is stored in `before`. The `Ptree` objects stored in `before` are inserted before the variable declaration after the translation.

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; };    // ; is always necessary
```

`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`. Because the new kind of statement is regarded as an expression in grammar, programmers can write it at any place an expression appears. However, they have to put a semicolon (`;`) at the end of the statement.

To implement this statement, first we have to register a new keyword `forall`:

```
// matrixclass.cc

BOOL MatrixClass::Initialize()
{
    RegisterNewWhileStatement("forall");
    return Class::Initialize();
}
```

`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::MakeStatement(
    "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. Note that it calls `MakeStatement()` instead of `Make()`. This is because the constructed code is not an expression but a statement. `TranslateStatement()` is called to recursively translate the body part of the `forall` statement.

B.2.5 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.

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.

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.

Write a meta-level program

The metaclass `BeforeClass` overrides `TranslateMemberCall()` to implement the translation mentioned above. The complete program of `BeforeClass` is `before.cc` in the distribution package. Here, we explain some key 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(IsMember(before) != 0)
        return before;
    else
        return nil;
}
```


In the first line, this produces the name of the before-method by `Ptree::Make()`. Then it calls `IsMember()` supplied by `Class`. `IsMember()` returns non-zero 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. We can define a subclass of `Environment::ClientData` and record the object in an environment. Let's define the subclass:

```
class TempVarName : public Environment::ClientData {
public:
    TempVarName(Ptree* p) { varname = p; }
    Ptree* varname;
};
```

A `TempVarName` object includes a single member `varname`, which is the name of a temporary variable. We record this object in the environment when we first declare the temporary variable. Then we use this object to check whether a temporary variable has been already declared.

```
Ptree* class_name = Name();
TempVarName* tmpvar
    = (TempVarName*)env->LookupClientData(this, class_name);
if(tmpvar != nil)
    varname = tmpvar->varname;
else{
    varname = Ptree::GenSym();
    tmpvar = new TempVarName(varname);
    Ptree* decl = Ptree::MakeStatement("%p* %p;",
                                      class_name, varname);
    env->InsertDeclaration(decl, this, 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 looks for a `TempVarName` object by calling `LookupClientData()` with two search keys. 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 a `TempVarName` object for future reference.

B.2.6 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.

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 Point : WrapperClass;
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 generates a wrapper function `wrapper_Move()` for `Move()`. The call of `Move()` in `f()` is substituted by the call of the wrapper function. For simplicity, we make the wrapper function just invoke the wrapped function `Move()` without doing anything else. The translated program should be this:

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

```

public:
    void wrapper_Move(int, int);
};

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

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

void f()
{
    Point p;
    p.wrapper_Move(3, 5); // call the wrapper
}

```

Write a meta-level program

`WrapperClass` has to do three things: (1) to insert member declarations for wrapper functions, (2) to generate the definitions of the wrapper functions, and (3) to replace a call of a member function with a call of the wrapper function. `WrapperClass` overrides `TranslateBody()` for (1), `TranslateMemberFunctionBody()` for (2), and `TranslateMemberCall()` for (3).

First, we show `TranslateBody()`. 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. To get a member name, we use `NthMemberName()`:

```

Ptree* WrapperClass::TranslateBody(Environment* env, Ptree* body)
{
    Ptree* decl = Ptree::Make("public:\n");
    Ptree* name;
    TypeInfo t;
    int i = 1;
    while((name = NthMemberName(i++)) != nil){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId
           && whatis != PtreeDestructorId)
            if(LookupMemberType(env, name, t))
                if(t.WhatIs() == FunctionType){
                    Ptree* mem = t.MakePtree(WrapperName(name));
                    decl = Ptree::Make("%p %p;\n",

```

```

                                decl, mem);
                                }
                                }
    return Ptree::Append(body, decl);
}

Ptree* WrapperClass::WrapperName(Ptree* name)
{
    return Ptree::Make("wrapper_%p", name);
}

```

In the `while` loop, we first check that the member is not a constructor or a destructor. If it is a constructor or a destructor, the parse tree indicated by `name` is `LeafClassNameId` or `PtreeDestructorId`. Then we get the type of the member by calling `LookupMemberType()`. If the member is a function, the type is `FunctionType`.

After we make sure that the member is a function, we generate the declaration of the wrapper function. We use `MakePtree()` to generate it. This member function of `TypeInfo` makes a parse tree that represents the type name. Suppose that a `TypeInfo t` is the pointer type to integer. Then `t.MakePtree()` returns:

```
int*
```

We can also give a variable name to `MakePtree()`. For example:

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

returns:

```
int* i
```

Similarly, if `t` is a function type, we can give a function name and get a parse tree that represents 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)
```

Next, we mention `TranslateMemberFunctionBody()`. It generates the definition of the wrapper function if the member function is not a constructor or a destructor. The actual generation is done by `MakeWrapper()`:

```
void WrapperClass::MakeWrapper(Environment* env, Ptree* name,
                               Ptree* wrapper_name, TypeInfo& t,
                               BOOL inlined)
{
    TypeInfo atype;
    TypeInfo rtype;
    int i;
    Ptree* wrapper;
    Ptree* arglist = nil;

    for(i = 1; t.NthArgument(i, atype); ++i){
        Ptree* arg = atype.MakePtree(Ptree::Make("p%d", i));
        if(i == 1)
            arglist = arg;
        else
            arglist = Ptree::Make("%p,%p", arglist, arg);
    }

    t.Dereference(rtype);
    Ptree* body = WrapperBody(env, name, wrapper_name, i-1, t);
    Ptree* head = Ptree::Make("%p::%p(%p)",
                              Name(), wrapper_name, arglist);
    if(rtype.WhatIs() != UndefType)
        head = rtype.MakePtree(head);

    if(inlined)
        wrapper = Ptree::Make("inline %p{%p}\n",
                              head, body));
    else
        wrapper = Ptree::Make("%p{%p}\n", head, body));

    AppendAfterToplevel(wrapper);
}
```

By the first `for` loop, this member function constructs the argument list `arglist`. The name of the arguments are `p1`, `p2`, `p3`, and so on. To obtain the argument type, we call `NthArgument()` of `TypeInfo`. Then we call `MakePtree()` to construct each argument declaration.

The return type of the function is attached by calling `MakePtree()` on the return type `rtype`, which is obtained by calling `Dereference()` on the

function type `t`. The argument passed to `MakePtree()` is the rest of the function header such as `X::f(int p1, char p2)`.

The constructed definition of the wrapper function is finally inserted into the translated program by `AppendAfterToplevel()`. It inserts the constructed parse tree just after the (translated) definition of the original member function. This avoids unnecessary duplicated copies of the wrapper functions. If we generate the wrapper functions when `TranslateBody()` processes the class declaration, the wrapper functions would be duplicated every time the class declaration is included by a different source file.

Finally, we show `TranslateMemberCall()`, which substitutes the wrapper function for the member function originally called. Its definition is simple. It calls the member function of the base class `Class` with the wrapper function's name instead of the original member's name:

```
Ptree* WrapperClass::TranslateMemberCall(Environment* env,
                                         Ptree* member,
                                         Ptree* arglist)
{
    return Class::TranslateMemberCall(env, WrapperName(member),
                                       arglist);
}
```

Subclass of WrapperClass

The complete program of `WrapperClass` is found in `wrapper.cc`, 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 uses the subclass.)

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()`:

```
#include "wrapper.h"
class SyncClass : public WrapperClass {
public:
    SyncClass(Ptree* d, Ptree* m) : WrapperClass(d, m) {}
    Ptree* WrapperBody(Environment*, Ptree*, Ptree*, int,
                       TypeInfo&);
};
```

`WrapperBody()` is a virtual function and it is called by `MakeWrapper()` to construct the function body of the wrapper function. `WrapperBody()` supplied by `WrapperClass` returns a `return` statement such as:

```
return original-function(p1, p2, ...);
```

So we define `WrapperBody()` supplied by `SyncClass` as shown below:

```
Ptree* SyncClass::WrapperBody(Environment* env, Ptree* name,
                               Ptree* wrapper_name, int num_of_args,
                               TypeInfo& ftype)
{
    Ptree* ret = WrapperClass::WrapperBody(env, name, wrapper_name,
                                           num_of_args, ftype);
    return Ptree::Make("synchronize(); %p", ret);
}
```

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

As we see above, carefully designed metaclasses can be reused as the base 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`.

B.3 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.

B.3.1 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 class-name : metaclass-name [(meta-arguments)];`²

This declares the metaclass for a class. It must appear before the class is defined. *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 Point : PersistentClass;
class Point {
public:
    int x, y;
};
```

The metaclass for `Point` is `PersistentClass`.

B.3.2 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, programmers must register a new keyword, which is used in one of the following forms:

²[] means an optional field.

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

The keyword registered to lead a type modifier may appear in front of type names, the `new` operator, or class declarations. For example, these statements are valid:

```

distribute class Dictionary { ... };
remote Point* p = remote(athos) new Point;

```

Here, `distribute` and `remote` are registered keywords.

- 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 appear. For example, if `after` is a registered 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 registered 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;
};

```

Note the last semicolon “;”. It is needed because the `forall` statement is an “expressions” in the grammar.

A registered 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);
};
```

B.3.3 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.

B.4 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.

This section shows 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.
- **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.
- **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.
- **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.

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.

B.4.1 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

Table B.1: static member functions on Ptree

-
- Ptree* First(Ptree* lst) returns the first element of lst.
 - Ptree* Rest(Ptree* lst) returns the rest of lst except the first element, that is, the *cdr* field of lst.
 - Ptree* Second(Ptree* lst) returns the second element of lst.
 - Ptree* Third(Ptree* lst) returns the third element of lst.
 - Ptree* Nth(Ptree* lst, int n) returns the n-th element of lst. Nth(lst, 0) is equivalent to First(lst).
 - Ptree* Last(Ptree* lst) returns the last cons cell, which is a list containing only the last element of lst.
 - Ptree* ListTail(Ptree* lst, int k) returns a sublist of lst obtained by omitting the first k elements. ListTail(lst, 1) is equivalent to Rest(lst).
 - int Length(Ptree* lst) returns the number of the elements of lst.
 - Ptree* Cons(Ptree* a, Ptree* b) returns a cons cell whose *car* field is a and whose *cdr* is b.
 - Ptree* List(Ptree* e1, Ptree* e2, ...) returns a list whose elements are e1, e2, ... List() returns a null list `[]`, which is denoted by `nil` or `NIL`.
 - Ptree* Append(Ptree* lst1, Ptree* lst2) concatenates lst1 and lst2. It returns the resulting list.
 - Ptree* CopyList(Ptree* lst) returns a new list whose elements are the same as lst's.
 - BOOL Eq(Ptree* lst, char x)
 - BOOL Eq(Ptree* lst, char* x)
 - BOOL Eq(Ptree* lst, Ptree* x) returns TRUE if lst and x are equal.
-

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:

```
[[[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.

Basic Operations

To manipulate linked lists, the MOP provides some `static` member functions on `Ptree`, which are familiar to Lisp programmers. Table B.1 shows those `static` member functions. In addition, 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].

- `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.

- `PtreeId WhatIs()`

This returns an `enum` constant that corresponds to the syntactical meaning of the code that the metaobject represents. For example, the metaobject represents a class name, this member function returns `LeafClassNameId`. If it represents a `if` statement, `PtreeIfStatementId` is returned. The returned constants are listed in Table B.2.

Table B.2: enum constants returned by `Ptree::WhatIs()`

-
- Toplevel declarations
`PtreeDeclarationId`, `PtreeFunctionId`, `PtreeTemplateDeclId`,
`PtreeMetaclassId`, `PtreeTypedefId`, `PtreeLinkageId`
 - Statements
`PtreeExprStatementId`, `PtreeLabelId`, `PtreeCaseLabelId`,
`PtreeIfStatementId`, `PtreeSwitchStatementId`,
`PtreeWhileStatementId`, `PtreeDoStatementId`,
`PtreeForStatementId`, `PtreeBreakStatementId`,
`PtreeContinueStatementId`, `PtreeReturnStatementId`,
`PtreeGotoStatementId`
 - Expressions
`PtreeInfixExprId`, `PtreeCondExprId` (conditional),
`PtreeCastId`, `PtreeUnaryExprId`, `PtreePostfixExprId`,
`PtreeMemberAccessExprId`, `PtreeSizeofExprId`,
`PtreeAssignExprId`, `PtreeSizeofTypeId`, `PtreePtrToMemExprId`
(pointer to member, `->*` or `.*`), `PtreeNewId`, `PtreeDeleteId`,
`PtreeFstyleCastId` (function-style cast), `PtreeUserStatementId`,
`PtreeStaticUserStatementId`, `PtreeActualArgsId`
 - Class Declarations
`PtreeEnumId`, `PtreeClassId`
 - Groups
`PtreeParenId` (parenthesis), `PtreeBracketId`, `PtreeBraceId`,
`PtreeAngleId`,
 - Leaf Nodes
`LeafTypeNameId`, `LeafClassNameId`, `LeafPointerId`,
`LeafReferenceId`, `LeafConstValueId`, `LeafNameId`,
`LeafReservedId`, `LeafThisId`, `LeafReservedModifierId`,
 - Others
`PtreeTemplateNameId`, `PtreeQClassNameId` (qualified class name),
`PtreeQnameId` (qualified name), `PtreePtrToMemberId` (pointer to
member, `::*`), `PtreeCvQualifierId` (const or volatile),
`PtreeDestructorId`, `PtreeOperatorFuncId`, `PtreeAccessCtrlId`,
`PtreeUserKeywordId`
-

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` lists `[]` are inserted at that place. For example, if the return type of a function declaration is omitted as follows:

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

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

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

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

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.

- `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 `%`.

- `Ptree* MakeStatement(char* format, [Ptree* sublist, ...])`

This is identical to `Make()` except the generated `Ptree` metaobject represents not an expression but a statement.

- `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()` and `MakeStatement()` is not a real parse tree.³ It is just a unparsed chunk of characters. Although

³At least, for the time being.

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()` and `MakeStatement()` follow 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)
```

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

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

Note that the "backquoted" 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`.

- `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*`.

- `Ptree* qMakeStatement(char* text)`

This is identical to `qMake()` except the generated `Ptree` metaobject represents not an expression but a statement.

Pattern Matching

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

- `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))
    puts("this is addition.");
else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
    puts("this is subtraction.");
else
    puts("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`.

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.

- `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`.

- `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`. The denoted string does not include the double quotes at the both ends. Also, the escape sequences are not expanded.

Convenience Classes `PtreeIter` and `PtreeArray`

The MOP provides a few convenience classes to help programmers to deal with `Ptree` objects. One of them is the class `PtreeIter`. It 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. This is the same as calling a member function `Pop()`. If the reader likes the `for`-loop style, she can also say:

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

Although this interface is slightly slow, it distinguishes the end of the list and a `nil` element.

Another class `PtreeArray` is for 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
```

B.4.2 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 `typedefed` 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`, `TemplateFunctionType`, 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`.

- `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`, and `VoidType`. For

example, `bit_field & LongType` is TRUE if the type is long, unsigned long, or signed long.

- `Class* ClassMetaobject()`

This returns a `Class` metaobject that represent the type. If the type is not a `class` type, it simply returns `nil`.

- `void Dereference(TypeInfo& t)`

This returns the dereferenced type in `t`. For example, if the type is `int**`, then `Dereference()` on the type returns a `TypeInfo` metaobject for `int*`. If a function type is dereferenced, `t` becomes its return type. 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.

- `BOOL NthArgument(int nth, TypeInfo& t)`

If the type is `FunctionType`, this returns the type of the `nth` (≥ 0) argument at `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 `UndefType`.

- `Ptree* MakePtree(Ptree* var_name = nil)`

This makes a `Ptree` metaobject that represents the type name. For example, if the type is pointer to integer, this returns `[[int* var_name]]`. `var_name` may be `nil`.

B.4.3 Representation of Environment

`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 these 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 changed to TRUE. If it is a variable name, `is_type_name` is FALSE.

- `BOOL Lookup(Ptree* name, TypeInfo& t)`

This is an alias of `Lookup(Ptree*, BOOL&, TypeInfo&)` shown above.

- `Class* LookupClassMetaobject(Ptree* name)`

This looks up the given `name` and returns a `Class` metaobject of the type. The `name` may be a variable name or a type name. If the `name` is not found, this function returns `nil`.

`Environment` metaobjects are also useful to store client data. Programmers can record client data in any `Environment` metaobject and look it up later. The client data must be a subclass of `Environment::ClientData`. The following member functions are for manipulating the client data:

- `BOOL AddClientData(Class* metaobject, Ptree* key, Environment::ClientData* data)`

This records `data` in the environment. `metaobject` and `key` are used to retrieve `data` later. If another data is recorded with the same pair of `metaobject` and `key`, this returns `FALSE`.

- `Environment::ClientData* LookupClientData(Class* metaobject, Ptree* key)`

This returns the client data recorded with `metaobject` and `key`. If the data is not found, this returns `nil`.

- `BOOL DeleteClientData(Environment::ClientData* data)`

This deallocates `data` recorded in the environment. If `data` is not found, this returns `FALSE`. (Note that `data` is automatically deallocated when the environment is deallocated.)

`Environment` metaobjects are also used to insert declarations in the translated program. These are member functions for that purpose:

- `void InsertDeclaration(Ptree* decl)`

This inserts the given code `decl` at the beginning of the function body that is currently processed.

- `BOOL InsertDeclaration(Ptree* decl, Class* metaobject, Ptree* key, Environment::ClientData* data)`

This inserts the given code `decl` at the beginning of the function body. The difference from the function above is that it also records the client data in the

outermost environment of the function body. It guarantees that the recorded data last while the function body is translated. Note that `AddClientData()` records the client data in the immediate environment (aka the innermost block). The client data is not visible out of that environment.

To insert code in the translated program, also see `InsertBeforeToplevel()` and `AppendAfterToplevel()` in Section B.4.4.

B.4.4 Class Metaobjects

Class metaobjects play the key role of the MOP. They represent class definitions and also control source-to-source translation of the program. Their default class is `Class`, but programmers may define a subclass of `Class` to control the source-code translation.

The class of a metaobject is selected by the `metaclass` declaration at the base level. If the metaclass for `Point` is `PersistentClass` at the base level, then the class metaobject for `Point` is an instance of `PersistentClass`. This semantics is natural because a metaclass means the class of a class.

Selecting a Metaclass

Base-level programmers may specify a metaclass in a way other than the `metaclass` declaration. The exact algorithm to select a metaclass (that is, the class of a class metaobject) is as shown below:

1. The metaclass specified by the `metaclass` declaration.
2. The metaclass specified by the keyword attached to the class declaration if exists (See `RegisterMetaclass()` in Section B.4.4 for more details).
3. Or else, the metaclass for the base classes. If the metaclass of each base class is different, an error is caused.
4. Otherwise, the default `Class` is selected.

If both the metaclass declaration and the keyword exist and they specify different metaclasses, then an error is caused.

Constructor

Class metaobjects may receive meta arguments from the base level when they are initialized. The constructor is responsible to deal with the meta arguments. By default, the meta arguments are simply ignored. Here is the prototype of the constructor:

- `Class(Ptree* definition, Ptree* meta_args)`

This constructor initializes the data members. `definition` is the whole piece of code of the class declaration. If meta arguments are not given, `meta_args` is `nil`.

Protocol for 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 field of the class declaration.

- `Ptree* BaseClasses()`

This returns the base-classes field of the class declaration. For example, if the class declaration is:

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

Then, `BaseClasses()` returns:

```
[[: [[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 whole of the class declaration. It is the `Ptree` metaobject passed to the constructor as `definition`.

- `char* MetaclassName()`

This returns the name of the metaclass.

- `Class* NthBaseClass(Environment* env, int n)`

This returns the *n*-th base class. *n* must be greater or equal to zero.

- `Ptree* NthMemberName(int n)`

This returns the name of the *n*-th member (including data members and member functions). It returns `nil` if *n*-th member does not exist. *n* must be greater or equal to zero.

- `int IsMember(Ptree* name)`

This returns `-1` if `name` is not a member of the class. Otherwise, it returns where the member is. `IsMember(NthMemberName(n))` is equal to *n*.

- `BOOL LookupMemberType(Environment* env, Ptree* name, TypeInfo& t)`

This looks up the type of the member specified by `name`. The found type is stored in `t`. The function returns `FALSE` if the member is not found.

Protocol for Translation

Class metaobjects controls 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.⁴ 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, it is a right way since only static type analysis is available for compile-time MOPs.

⁴In 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.

The `virtual` member functions on `Class` shown below control source-to-source translation on each kind of expression. They take an environment and an expression, then returns a translated expression. All of them are overridable.

- `Ptree* TranslateClassName(Environment* env, Ptree* keyword, Ptree* name)`

This translates the `name` of the class. All occurrences of the class name in the program are replaced with the returned name. `keyword` specifies a user-defined keyword (type modifier) attached to the class name. If none, `keyword` is `nil`.

— *Default implementation by Class*

This does not change the name but it just returns `name`. If `keyword` is not `nil`, it causes an error.

- `Ptree* TranslateSelf(Environment* env)`

This translates the declaration of the class. The declaration is not passed as an argument because it is available by `Definition()`.

— *Default implementation by Class*

This follows a layered subprotocol. It calls `TranslateClassName()`, `TranslateBaseClasses()`, and `TranslateBody()` so that class names in the declaration are correctly renamed.

- `Ptree* TranslateBaseClasses(Environment* env, Ptree* bases)`

This translates the base-classes field of class declaration.

— *Default implementation by Class*

This calls `TranslateClassName()` on each base class. A subclass of `Class` can call this with a modified `bases`. For example, it may append another base class to `bases`.

- `Ptree* TranslateBody(Environment* env, Ptree* body)`

This translates the body of class declaration. `body` has been already translated except user-defined access specifiers so that all the class names are renamed. This function can only append member functions written in C++ to `body`. The class names appearing in the appended member functions must be already renamed. This function also has to properly process user-defined access specifiers.

— *Default implementation by Class*

This returns `body`.

- `Ptree* TranslateMemberFunctionBody(Environment* env, Ptree* name, BOOL inlined, Ptree* body)`

This translates the body of a member function. `name` is the name of the member function. `inlined` is `TRUE` if the member function is inlined. `body` is statements surrounded by `{}`. The braces are not included by `body`. `env` contains the member name and the argument names.

— *Default implementation by Class*

This translates `body` by calling `TranslateStatement()` on each statement.

- `Ptree* TranslateInitializer(Environment* env, Ptree* expr, PtreeArray& before, PtreeArray& after)`

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 z(2.3, 4.0);
complex z = 0.0;
```

The `Ptree` objects stored in `before` and `after` are inserted before and after the declaration including the variable initializer. This is useful to implement, for example, a translation from

```
complex z = {2.3, 4.0};
```

into:

```
static double tmp[] = {2.3, 4.0};
complex z(tmp);
```

To store a `Ptree` object in an `PtreeArray` object, `Append(Ptree* list)` on `PtreeArray` is available.

— *Default implementation by Class*

This translates `expr` by calling `TranslateExpression()` on the initializing parameter (the second element of `expr`) and returns the result.

- `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 `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*` `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*` `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 the `()` operator. `object` is an instance of the class. `object` and `args` have not been translated yet.

— *Default implementation by Class*

This calls `TranslateExpression()` on `object` and `args` 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*` `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* 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 type of `rexpr` is the class.

— *Default implementation by Class*

This calls `TranslateExpression()` on `lexpr` and `rexpr` 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` denotes the lexical token `new`. `type_name` may include an array size surrounded by `[]`. `arglist` is arguments to the constructor. It includes parentheses `()`. `placement`, `tname`, and `arglist` have not been translated yet.

— *Default implementation by Class*

This calls `TranslateExpression()` on `placement` and `arglist`, and `TranslateNewType()` on `type_name`. Then it 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.

— *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. `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 `TranslateArgumentList()` 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. `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 `TranslateArgumentList()` on `arglist` and returns the translated expression.

- `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`.

— *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. This is named after that the syntax is similar to one for `static` member function calls. The meaning of the arguments is the same as that of `TranslateUserStatement()`.

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.

Protocol for Initialization and Finalization

The MOP provides functions to initialize and finalize class metaobjects.

- `static BOOL Initialize()`

This is called only once on each metaclass right after the compiler starts. It returns `TRUE` if the initialization succeeds. The subclasses of `Class` may define their own `Initialize()` but they have to call their base classes' `Initialize()`. This function is not overridable.

— *Default implementation by Class*

This does nothing except returning `TRUE`.

- `virtual Ptree* Finalize()`

This is called on each class metaobject after all the translation is finished. The returned `Ptree` object is inserted at the end of the translated source file.

— *Default implementation by Class*

This returns `nil`.

Protocol for 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 are called within `Initialize()` in Section B.4.4.

- `void RegisterNewModifier(char* keyword)`

This registers `keyword` as a new type modifier.

- `void RegisterNewAccessSpecifier(char* keyword)`

This registers `keyword` as a new access specifier.

- `void RegisterNewWhileStatement(char* keyword)`

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

- `void RegisterNewForStatement(char* keyword)`

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

- `void RegisterNewClosureStatement(char* keyword)`

This registers `keyword` as a new closure statement.

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

This registers `keyword` as a new type modifier and associates it with `metaclass`. If this keyword appears in front of a class declaration, then `metaclass` is selected for the declared class. See also Section B.4.4.

Service Functions

These are also `static` member functions on `Class`:

- `Ptree* TranslateNewType(Environment* env, Ptree* type_name)`
This translates the `type_name` field of `new` expressions.
- `Ptree* TranslateArgumentList(Environment* env, Ptree* arglist)`
This translates an argument list.
- `Ptree* TranslateIndex(Environment* env, Ptree* index)`
This translates the `index` field of subscript expressions.
- `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* TranslateStatement(Environment* env, Ptree* expr)`
This translates a statement, including a block statement surrounded by `{}`.

The next two member functions insert given code into the translated source code. As for insertion, also see `InsertDeclaration()` in Section B.4.3.

- `void InsertBeforeToplevel(Ptree* list)`
This inserts `list` before the toplevel declaration, such as function definitions, that are currently translated.
- `void AppendAfterToplevel(Ptree* list)`
This appends `list` after the toplevel declaration, such as function definitions, that are currently translated.

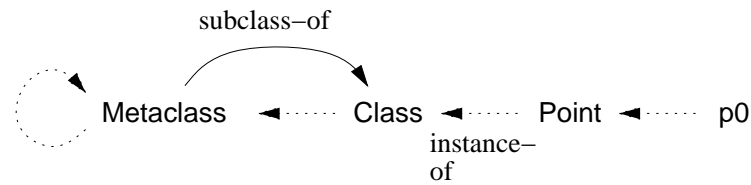


Figure B.2: Instance-of Relationship

Template Class

Metaclasses for template classes must be `TemplateClass` or its subclass programmers define. `TemplateClass` is a subclass of `Class` and it defines the following member functions for introspection:

- `Ptree* TemplateDefinition()`
This returns the whole definition of the template, including the keyword `template` and the template arguments.
- `Ptree* TemplateArguments()`
This returns the template argument.

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 B.2.

B.4.5 Error Message

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

B.5 Command Reference

NAME

`occ` — the Open C++ compiler

SYNOPSIS

```
occ [-l] [-s] [-v] [-c] [-E] [-Iinclude_directory]
    [-Dname[=def]] [-- C++ compiler options] source_file
```

DESCRIPTION

`occ` compiles an OpenC++ program into an object file. It first invokes the C++ preprocessor and generates a `.occ` file, then translates it into a `.ii` file according to meta-level code. 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.

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.
- `-c` Suppress linking and produce a `.o` file.
- `-l` Print the list of loaded metaclasses.
- `-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.
- `--` 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.

FILES

<code>file.cc</code>	source file.
<code>file.occ</code>	output file after C++ preprocessing.
<code>file.ii</code>	output file after translation.
<code>file.o</code>	object file.
<code>opencxx.a</code>	library to link with meta-level code.

NOTES

- When the C++ processor runs, the macro `__opencxx` is predefined.
- The current version of the compiler cannot dynamically load meta-level programs on demand.

COPYRIGHT

Copyright ©1995, 1996 Xerox Corporation. All Rights Reserved.

AUTHOR

Shigeru Chiba, The University of Tokyo.

Appendix C

Programs

C.1 The Distributed Object Library

Meta-level program (distobj.cc)

```
#include "mop.h"

metaclass DistributionClass : Metaclass;

class DistributionClass : public Class {
public:
    DistributionClass(Ptree* d, Ptree* m) : Class(d, m) {}

    Ptree* TranslateSelf(Environment*);
    Ptree* TranslateBody(Environment*, Ptree*);
    Ptree* AppendDecoder(Ptree*, Ptree*, int, TypeInfo&);

    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*, Ptree*, Ptree*);
};

#define BUF_NAME      "mBuffer"

Ptree* DistributionClass::TranslateSelf(Environment* env)
{
    Ptree* name;
    TypeInfo t;
    int i;
    Ptree* code = nil;
    for(i = 0; (name = NthMemberName(i)) != nil; ++i){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId && whatis != PtreeDestructorId)
            if(LookupMemberType(env, name, t) && t.WhatIs() == FunctionType){
                // Unless the member is a constructor, a destructor,
                // or a data member, then ...

                code = AppendDecoder(code, name, i, t);
            }
    }
}
```

```

AppendAfterToplevel(Ptree::qMake(
    "void 'Name()':::Dispatch(int* buffer, void* object, int member){\n"
    "switch(member){\n 'code' }})",);

return Class::TranslateSelf(env);
}

Ptree* DistributionClass::TranslateBody(Environment* env, Ptree* body)
{
    Ptree* decl = Ptree::qMake("public:"
        "static void Dispatch(int*, void*, int);\n");
    return Ptree::Append(body, decl);
}

Ptree* DistributionClass::AppendDecoder(Ptree* code, Ptree* name,
    int nth, TypeInfo& t)
{
    int i;
    TypeInfo atype;
    TypeInfo rtype;

    code = Ptree::Snoc(code, Ptree::qMake("case 'nth': {\n"
        "int s = 0;});");

    for(i = 0; t.NthArgument(i, atype); ++i){
        Ptree* argtype = atype.MakePtree();
        code = Ptree::Snoc(code, Ptree::qMake(
            "'argtype' p'i' = *('argtype*)&buffer[s];\n"
            "s += sizeof('argtype');\n"));
    }

    t.Dereference(rtype);
    if(rtype.WhatIs() != UndefinedType)
        code = Ptree::Snoc(code, Ptree::qMake(
            "*"int*)buffer = (('Name()'*object)->'name'("));
    else if(rtype.IsBuiltInType() & VoidType)
        code = Ptree::Snoc(code, Ptree::qMake(
            "('Name()'*object)->'name'("));
    else
        code = Ptree::Snoc(code, Ptree::qMake(
            "*"('rtype.MakePtree()'*buffer"
            "= (('Name()'*object)->'name'("));

    for(int j = 0; j < i; ++j)
        if(j + 1 >= i)
            code = Ptree::Snoc(code, Ptree::qMake("p'j'"));
        else
            code = Ptree::Snoc(code, Ptree::qMake("p'j',"));

    code = Ptree::Snoc(code, Ptree::qMake("};\nbreak;\n"));

    return code;
}

Ptree* DistributionClass::TranslateMemberCall(Environment* env, Ptree* object,

```

```

                                Ptree* op, Ptree* member,
                                Ptree* arglist)
{
    TypeInfo ftype, atype;
    int id = IsMember(member);
    if(id == 0){
        ErrorMessage("no such a member", member);
        return Class::TranslateMemberCall(env, object, op, member, arglist);
    }

    Ptree* p;
    PtreeIter next(Ptree::Second(arglist));
    Ptree* code = nil;
    Ptree* tmp = Ptree::GenSym();
    int i = 0;

    env->InsertDeclaration(Ptree::qMake("int 'tmp';"));
    LookupMemberType(env, member, ftype);
    for(i = 0; ftype.NthArgument(i, atype); ++i){
        p = next();
        Ptree* tname = atype.MakePtree();
        code = Ptree::Snoc(code, Ptree::qMake(
            *('tname'*)&" BUF_NAME '['tmp'] = 'TranslateExpression(env, p)',"
            "'tmp' += sizeof('tname');"));
        next();          // skip ,
    }

    return Ptree::qMake("( 'tmp'=0, 'code' CallRemote('tmp', 'object', 'id')");
}

```

Runtime Library (remote.h)

```

#ifndef __remote_h
#define __remote_h

void ExportObject(void * object, char* name, void* dispatch);
void ServerLoop();
void StartupClient(char* server_name);
int CallRemote(int numofargs, void* object, int member);
int ImportObject(char* name);

extern int* mBuffer;

#define Export(object,name,type) \
ExportObject((void*)object,name,(void*)&type::Dispatch)

#define Import(name,type)      (type*)ImportObject(name)

#endif __remote_h

```

Runtime Library (remote.cc)

```

#include <string.h>
#include "ipc.h"

```

```

#include "remote.h"

const int PORT = 4002;
const int MAX = 64;
const int LOOKUP = -1;

typedef void (*DispatchProc)(int*, void*, int);

static TcpServer* tcpServer;
static TcpClient* tcpClient;
static int tcpBuffer[32];

static struct {
    char* name;
    void* object;
    void* dispatch;
} exportTable[MAX];

static struct {
    void* object;
    void* dispatch;
} importTable[MAX];

static int numOfExport = 0;
static int numOfImport = 0;

/*
   The first four words are for house keeping: size, member,
   dispatcher, object.
*/
int* mBuffer = &tcpBuffer[4];

static int LookupExportedObject(char* name);

void ExportObject(void * object, char* name, void* dispatch)
{
    exportTable[numOfExport].name = name;
    exportTable[numOfExport].object = object;
    exportTable[numOfExport].dispatch = dispatch;
    ++numOfExport;
}

static int LookupExportedObject(char* name)
{
    for(int i = 0; i < numOfExport; ++i)
        if(strcmp(exportTable[i].name, name) == 0)
            return i;

    return -1;
}

void ServerLoop()
{
    int size;
    tcpServer = new TcpServer(PORT);
    for(;;){

```



```

        BOOL is_new_client;
        int fd = tcpServer->Wait(is_new_client);
        if(tcpServer->Recv(fd, (char*)&size, sizeof(size)) == 0)
            continue;

        tcpServer->Recv(fd, (char*)&tcpBuffer[1], size - sizeof(int));
        int member = tcpBuffer[1];
        if(member == LOOKUP){
            int index = LookupExportedObject((char*)&tcpBuffer[2]);
            tcpBuffer[0] = (int)exportTable[index].object;
            tcpBuffer[1] = (int)exportTable[index].dispatch;
            tcpServer->Send(fd, (char*)tcpBuffer, sizeof(int) * 2);
        }
        else{
            (*(DispatchProc)tcpBuffer[2])(mBuffer, (void*)tcpBuffer[3],
                member);
            tcpServer->Send(fd, (char*)mBuffer, sizeof(int));
        }
    }
}

void StartupClient(char* server_name)
{
    tcpClient = new TcpClient(server_name, PORT);
}

int CallRemote(int numofargs, void* object, int member)
{
    int size = sizeof(int) * (numofargs + 5);
    tcpBuffer[0] = size;
    tcpBuffer[1] = member;
    tcpBuffer[2] = (int)importTable[(int)object].dispatch;
    tcpBuffer[3] = (int)importTable[(int)object].object;
    tcpClient->Send((char*)tcpBuffer, size);
    tcpClient->Recv((char*)tcpBuffer, sizeof(int));
    return tcpBuffer[0];
}

int ImportObject(char* name)
{
    int size = sizeof(int) * 2 + strlen(name) + 1;
    strcpy((char*)&tcpBuffer[2], name);
    tcpBuffer[0] = size;
    tcpBuffer[1] = LOOKUP;
    tcpClient->Send((char*)tcpBuffer, size);
    tcpClient->Recv((char*)tcpBuffer, sizeof(int) * 2);
    importTable[numOfImport].object = (void*)tcpBuffer[0];
    importTable[numOfImport].dispatch = (void*)tcpBuffer[1];
    return numOfImport++;
}

```

C.2 The Wrapper Function Metaclass Library

Included File (wrapper.h)

```
#include "mop.h"

class WrapperClass : public Class {
public:
    WrapperClass(Ptree* d, Ptree* m) : Class(d, m) {}

    Ptree* TranslateBody(Environment*, Ptree*);
    virtual Ptree* WrapperName(Ptree* name);

    Ptree* TranslateMemberFunctionBody(Environment*, Ptree*, BOOL, Ptree*);
    virtual void MakeWrapper(Environment*, Ptree*, Ptree*, TypeInfo&, BOOL);
    virtual Ptree* WrapperBody(Environment*, Ptree*, Ptree*, int, TypeInfo&);

    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*, Ptree*,
                               Ptree*);
    Ptree* TranslateMemberCall(Environment*, Ptree*, Ptree*);
};
```

Implementation (wrapper.cc)

```
#include "wrapper.h"

// TranslateBody() inserts declarations for wrapper functions.

Ptree* WrapperClass::TranslateBody(Environment* env, Ptree* body)
{
    Ptree* decl = Ptree::qMake("public:\n");
    Ptree* name;
    TypeInfo t;
    int i = 0;
    while((name = NthMemberName(i++)) != nil){
        PtreeId whatis = name->WhatIs();
        if(whatis != LeafClassNameId && whatis != PtreeDestructorId)
            if(LookupMemberType(env, name, t) && t.WhatIs() == FunctionType){
                // if the member is not a constructor, a destructor,
                // or a data member, insert the declaration for the wrapper.
                Ptree* m = t.MakePtree(WrapperName(name));
                decl = Ptree::qMake("'decl' 'm';\n");
            }
    }

    return Ptree::Append(body, decl);
}

Ptree* WrapperClass::WrapperName(Ptree* name)
{
    return Ptree::qMake("_wrap_'name'");
}

/*
    TranslateMemberFunctionBody() defines wrapper functions.
```

```

They are defined just after the wrapped function is defined.
Note that if they are defined in the header file, their duplicated
copies would be unnecessarily produced.
*/
Ptree* WrapperClass::TranslateMemberFunctionBody(Environment* env,
                                                Ptree* name, BOOL inlined,
                                                Ptree* body)
{
    TypeInfo t;

    PtreeId whatis = name->WhatIs();
    if(whatis != LeafClassNameId && whatis != PtreeDestructorId)
        if(LookupMemberType(env, name, t))
            MakeWrapper(env, name, WrapperName(name), t, inlined);

    return Class::TranslateMemberFunctionBody(env, name, inlined, body);
}

/*
MakeWrapper() generates the member function such as:

<return type> CLASS::WRAPPER_NAME(<type> p1, <type> p2, ...)
{
    <The code returned by WrapperBody(>
}
*/
void WrapperClass::MakeWrapper(Environment* env, Ptree* name,
                              Ptree* wrapper_name, TypeInfo& t, BOOL inlined)
{
    TypeInfo atype;
    TypeInfo rtype;
    int i;
    Ptree* arglist = nil;

    for(i = 0; t.NthArgument(i, atype); ++i){
        Ptree* arg = atype.MakePtree(Ptree::qMake("p'i('"));
        if(i == 0)
            arglist = arg;
        else
            arglist = Ptree::qMake("'arglist', 'arg('");
    }

    t.Dereference(rtype);
    Ptree* body = WrapperBody(env, name, wrapper_name, i, t);
    Ptree* head = Ptree::qMake("'Name()'::'wrapper_name'('arglist('");
    if(rtype.WhatIs() != UndefType)
        head = rtype.MakePtree(head);

    if(inlined)
        AppendAfterToplevel(Ptree::qMake("inline 'head'{'body'}\n"));
    else
        AppendAfterToplevel(Ptree::qMake("'head'{'body'}\n"));
}

/*
WrapperBody() returns the body of the wrapper.  NAME is the name of

```

the member function wrapped by this wrapper. WRAPPER_NAME is the name of this wrapper. NUM_OF_ARGS is the number of the arguments needed to call the wrapped member function. The arguments are (p1, p2, ..., p<NUM_OF_ARGS>).

This function returns a return statement:

```

    return <the wrapped function>(p1, p2, ...);
*/
Ptree* WrapperClass::WrapperBody(Environment* env, Ptree* name,
                                Ptree* wrapper_name, int num_of_args,
                                TypeInfo& ftype)
{
    // first make the argument list needed to call the wrapped function.

    Ptree* arglist = nil;
    while(num_of_args > 0){
        if(num_of_args > 1)
            arglist = Ptree::qMake("p'--num_of_args' 'arglist'");
        else{
            arglist = Ptree::qMake("p0 'arglist'");
            --num_of_args;
        }
    }

    // then make a statement that calls the wrapped member function

    ftype.Dereference();
    if(ftype.IsBuiltInType() & VoidType)
        return Ptree::qMake("name('arglist');");
    else
        return Ptree::qMake("return 'name('arglist');");
}

// TranslateMemberCall() replaces the called function with the wrapper
// function.

Ptree* WrapperClass::TranslateMemberCall(Environment* env,
                                         Ptree* object, Ptree* op,
                                         Ptree* member, Ptree* arglist)
{
    return Class::TranslateMemberCall(env, object, op, WrapperName(member),
                                      arglist);
}

Ptree* WrapperClass::TranslateMemberCall(Environment* env,
                                         Ptree* member, Ptree* arglist)
{
    return Class::TranslateMemberCall(env, WrapperName(member), arglist);
}

```

C.3 Vector Library

Vector library (vector.h)

```

const SIZE = 8;

#ifdef __opencxx
metaclass Vector : VectorClass;
#endif

template <class T> class Vector {
public:
    T element[SIZE];

    Vector() {}

    Vector(T t){
        for(int i = 0; i < SIZE; ++i)
            element[i] = t;
    }

    Vector operator + (Vector& a){
        Vector<T> v;
        for(int i = 0; i < SIZE; ++i)
            v.element[i] = element[i] + a.element[i];

        return v;
    }

    Vector operator - (Vector& a){
        Vector<T> v;
        for(int i = 0; i < SIZE; ++i)
            v.element[i] = element[i] - a.element[i];

        return v;
    }
};

template <class T>
Vector<T> operator * (T k, Vector<T>& a){
    Vector<T> v;
    for(int i = 0; i < SIZE; ++i)
        v.element[i] = k * a.element[i];

    return v;
};

```

Micro benchmark program with the vector library

```

#include <iostream.h>
#include "vector.h"

extern "C" { long clock(); }

const N = 1000000;

```

```

main()
{
    Vector<double> v1(2.167), v2(10.95), v3(50196), v4(44.4077);
    Vector<double> v0;
    int i;

    long t0 = clock();
    for(i = 0; i < N; ++i)
        v0 = v1;

    long t1 = clock();
    for(i = 0; i < N; ++i)
        v0 = v1 + v2;

    long t2 = clock();
    for(i = 0; i < N; ++i)
        v0 = v1 + v2 + v3;

    long t3 = clock();
    for(i = 0; i < N; ++i)
        v0 = v1 + v2 + v3 + v4;

    long t4 = clock();

    cout << "N=" << N << ", SIZE=" << SIZE << "\n";
    cout << "1)" << (t1 - t0) / 1000 << "msec, 2)" << (t2 - t1) / 1000
        << "msec, 3)" << (t3 - t2) / 1000 << "msec, 4)"
        << (t4 - t3) / 1000 << "msec.\n";
}

```

Micro benchmark program written by hand

```

#include <iostream.h>

extern "C" { long clock(); }

const SIZE = 8;
const N = 1000000;

main()
{
    double v1[SIZE], v2[SIZE], v3[SIZE], v4[SIZE];
    double v0[SIZE];

    int i, j;

    for(i = 0; i < SIZE; ++i){
        v1[i] = 2.167;
        v2[i] = 10.95;
        v3[i] = 50196;
        v4[i] = 44.4077;
    }
}

```

```

long t0 = clock();
for(i = 0; i < N; ++i)
    for(j = 0; j < SIZE; ++j)
        v0[j] = v1[j];

long t1 = clock();

for(i = 0; i < N; ++i)
    for(j = 0; j < SIZE; ++j)
        v0[j] = v1[j] + v2[j];

long t2 = clock();

for(i = 0; i < N; ++i)
    for(j = 0; j < SIZE; ++j)
        v0[j] = v1[j] + v2[j] + v3[j];

long t3 = clock();

for(i = 0; i < N; ++i)
    for(j = 0; j < SIZE; ++j)
        v0[j] = v1[j] + v2[j] + v3[j] + v4[j];

long t4 = clock();

cout << "N=" << N << ", SIZE=" << SIZE << "\n";
cout << "1)" << (t1 - t0) / 1000 << "msec, 2)" << (t2 - t1) / 1000
    << "msec, 3)" << (t3 - t2) / 1000 << "msec, 4)"
    << (t4 - t3) / 1000 << "msec.\n";
}

```

C.4 The Standard Template Library

Meta-level program (stl-class.cc)

```

#include "mop.h"

// for List

class ListIteratorClass : public Class {
public:
    ListIteratorClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateSelf(Environment*);
    Ptree* TranslatePostfix(Environment*, Ptree*, Ptree*);
    Ptree* TranslateUnary(Environment*, Ptree*, Ptree*);
};

Ptree* ListIteratorClass::TranslateSelf(Environment* env)
{
    return Ptree::Make("typedef List<T>* iterator");
}

Ptree* ListIteratorClass::TranslateUnary(Environment* env, Ptree* op,

```

```

                                Ptree* object)
{
    if(op->Eq('*')){
        object = TranslateExpression(env, object);
        return Ptree::qMake("object'->value");
    }
    else if(op->Eq("++")){
        object = TranslateExpression(env, object);
        return Ptree::qMake("object'='object'->next");
    }
    else
        return Class::TranslateUnary(env, op, object);
}

Ptree* ListIteratorClass::TranslatePostfix(Environment* env, Ptree* object,
                                           Ptree* op)
{
    TypeInfo type;

    Ptree* tmp = Ptree::GenSym();
    object = TranslateExpression(env, object, type);
    Ptree* decl = Ptree::qMake("type.FullTypeName()" 'tmp');
    env->InsertDeclaration(decl);
    return Ptree::qMake("tmp'='object', 'object'='object'->next, tmp'");
}

// for Set

class SetIteratorClass : public Class {
public:
    SetIteratorClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateSelf(Environment*);
    Ptree* TranslatePostfix(Environment*, Ptree*, Ptree*);
    Ptree* TranslateUnary(Environment*, Ptree*, Ptree*);
};

Ptree* SetIteratorClass::TranslateSelf(Environment* env)
{
    return Ptree::qMake("typedef Set<T>* iterator");
}

Ptree* SetIteratorClass::TranslateUnary(Environment* env, Ptree* op,
                                       Ptree* object)
{
    if(op->Eq('*')){
        object = TranslateExpression(env, object);
        return Ptree::qMake("object'->value");
    }
    else if(op->Eq("++")){
        TypeInfo type;
        object = TranslateExpression(env, object, type);
        Ptree* tname = type.FullTypeName();
        return Ptree::qMake("object'=(tname')Tree::Right('object')");
    }
    else
        return Class::TranslateUnary(env, op, object);
}

```



```

}

Ptree* SetIteratorClass::TranslatePostfix(Environment* env, Ptree* object,
                                         Ptree* op)
{
    TypeInfo type;

    Ptree* tmp = Ptree::GenSym();
    object = TranslateExpression(env, object, type);
    Ptree* tname = type.FullTypeName();
    Ptree* decl = Ptree::qMake("'tname' 'tmp'");
    env->InsertDeclaration(decl);
    return Ptree::qMake("'tmp'='object',"
                       "'object'=('tname')Tree::Right('object',"
                       "'tmp')");
}

```

Base-level library (stl.h)

```

#define nil          0

template <class T> class List {
public:
    List<T>* next;
    T value;

    typedef T valuetype;

#ifdef __opencxx
    metaclass iterator : ListIteratorClass;
#endif
    class iterator {
    public:
        iterator(List<T>* p) { ptr = p; }
        List<T>* ptr;
        int eof() { return ptr == nil; }
        int operator != (iterator& a) { return ptr != a.ptr; }
        T operator * () { return ptr->value; }
        iterator& operator ++ () { ptr = ptr->next; return *this; }
        iterator operator ++ (int) {
            iterator prev = *this;
            ptr = ptr->next;
            return prev;
        }
    };

    iterator begin() { return iterator(this); }
    iterator end() { return iterator(nil); }
};

class Tree {
public:
    Tree* parent;
    Tree* right;
    Tree* left;
}

```

```

inline static Tree* RightMost(Tree* p){
    while(p->right != nil)
        p = p->right;

    return p;
}

inline static Tree* LeftMost(Tree* p){
    while(p->left != nil)
        p = p->left;

    return p;
}

inline static Tree* Right(Tree* p){
    if(p->right != nil){
        p = p->right;
        while(p->left != nil)
            p = p->left;

        return p;
    }
    else{
        Tree* q = p->parent;
        while(q != nil && p == q->right){
            p = q;
            q = q->parent;
        }

        return q;
    }
}
};

template <class T> class Set : public Tree {
public:
    T value;

#ifdef __opencxx
    metaclass iterator : SetIteratorClass;
#endif
    class iterator {
    public:
        iterator(Tree* p) { ptr = (Set<T>*)p; }
        Set<T>* ptr;
        int operator != (iterator& a) { return ptr != a.ptr; }
        T operator * () { return ptr->value; }
        iterator& operator ++ () {
            ptr = (Set<T>*)Tree::Right(ptr);
            return *this;
        }
        iterator operator ++ (int) {
            iterator prev = *this;
            ptr = (Set<T>*)Tree::Right(ptr);
            return prev;
        }
    }
}

```

```

};

    iterator begin() { return iterator(Tree::LeftMost(this)); }
    iterator end() { return iterator(0); }
};

```

Micro benchmark program (stl-test.cc)

```

#include <iostream.h>
#include "stl.h"

const N = 100000;
const R = 100;
const S = 10;
const L = 20;          // 1048574 elements

extern "C" { long clock(); }

template <class I, class T> int count(I first, I last, T value)
{
    int n = 0;
    while(first != last)
        if(*first++ == value)
            ++n;

    return n;
}

int list_count(List<int>* first, List<int>* last, int value)
{
    int n;
    for(n = 0; first != last; first = first->next)
        if(first->value == value)
            ++n;

    return n;
}

int set_count(Set<int>* first, Set<int>* last, int value)
{
    int n;
    for(n = 0; first != last; first = (Set<int>*)Tree::Right(first))
        if(first->value == value)
            ++n;

    return n;
}

// initialize routines

List<int>* list_init(){
    List<int>* lst = nil;
    for(int i = 0; i < N; ++i){
        List<int>* node = new List<int>;
        node->value = N;
        node->next = lst;
    }
}

```

```

        lst = node;
    }

    return lst;
}

Set<int>* MakeTree(int start, int size)
{
    Set<int>* node = new Set<int>;
    if(size == 1){
        node->left = node->right = nil;
        node->value = start;
        node->parent = nil;
        return node;
    }
    else{
        int size2 = (size - 1) / 2;
        node->value = start + size2;
        node->left = MakeTree(start, size2);
        node->left->parent = node;
        node->right = MakeTree(start + size2 + 1, size2);
        node->right->parent = node;
        node->parent = nil;
        return node;
    }
}

Set<int>* set_init()
{
    int n;
    int level = L;
    for(n = 1; level > 1; --level)
        n = n * 2 + 1;

    return MakeTree(1, n);
}

main()
{
    int i;
    List<int>* lst = list_init();
    Set<int>* set = set_init();

    int s0 = list_count(lst, nil, N);
    long t0 = clock();
    for(i = 0; i < R; ++i)
        s0 = list_count(lst, nil, N);

    t0 = clock() - t0;
    int s1 = count(lst->begin(), lst->end(), N);
    long t1 = clock();

    for(i = 0; i < R; ++i)
        s1 = count(lst->begin(), lst->end(), N);

    t1 = clock() - t1;
}

```

```

#ifdef __opencxx
    int s2 = set_count(set->begin(), nil, N);
    long t2 = clock();
    for(i = 0; i < S; ++i)
        s2 = set_count(set->begin(), nil, N);

    t2 = clock() - t2;
#else
    int s2 = set_count(set->begin().ptr, nil, N);
    long t2 = clock();
    for(i = 0; i < S; ++i)
        s2 = set_count(set->begin().ptr, nil, N);

    t2 = clock() - t2;
#endif

    int s3 = count(set->begin(), set->end(), N);
    long t3 = clock();
    for(i = 0; i < S; ++i)
        s3 = count(set->begin(), set->end(), N);

    t3 = clock() - t3;

    cout << "N=" << N << ", R=" << R;
    cout << ", L=" << L << ", S=" << S << "\n";
    cout << "list_count()=" << s0 << ", time: " << t0 / 1000 << "msec\n";
    cout << "count(list) =" << s1 << ", time: " << t1 / 1000 << "msec\n";
    cout << "set_count() =" << s2 << ", time: " << t2 / 1000 << "msec\n";
    cout << "count(set) =" << s3 << ", time: " << t3 / 1000 << "msec\n";
}

```

C.5 OOPACK benchmark

The OOPACK benchmark program

```

//=====
//
// OOPACK - a benchmark for comparing OOP vs. C-style programming.
// Copyright (C) 1995 Arch D. Robison
//
// This program is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation; either version 2 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// For a copy of the GNU General Public License, write to the Free Software
// Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
//

```

```

//=====
//
// OOPACK: a benchmark for comparing OOP vs. C-style programming.
//
// Version: 1.7
//
// Author: Arch D. Robison (robison@kai.com)
//         Kuck & Associates
//         1906 Fox Dr.
//         Champaign IL 61820
//
// Web Info: http://www.kai.com/oopack/oopack.html
//
// Last revised: November 21, 1995
//
// This benchmark program contains a suite of tests that measure the relative
// performance of object-oriented-programming (OOP) in C++ versus just writing
// plain C-style code in C++. All of the tests are written so that a
// compiler can in principle transform the OOP code into the C-style code.
// After you run this benchmark and discover just how much you are paying to
// use object-oriented programming, you will probably say: OOP? ACK!
// (Unless, of course, you have Kuck & Associates' Photon C++ compiler.)
//
// TO COMPILE
//
//   Compile with your favorite C++ compiler. E.g. 'CC -O2 oopack.C'.
//   On most machines, no special command-line options are required.
//   For Suns only, you need to define the symbol 'sun4'.
//   E.g. 'g++ -O -Dsun4 oopack.C'.
//
// TO RUN
//
//   To run the benchmark, run 'a.out Max=50000 Matrix=500 Complex=20000
//   Iterator=50000'.
//   This runs the four tests for the specified number of iterations.
//   E.g., the Max test is run for 50000 iterations. You may want to
//   adjust the number of iterations to be small enough to get
//   an answer in reasonable time, but large enough to get a reasonably
//   accurate answer.
//
// INTERPRETING THE RESULTS
//
//   Below is an example command line and the program's output.
//
//   $ a.out Max=5000 Matrix=50 Complex=2000 Iterator=5000
//   OOPACK Version 1.7
//
//   For results on various systems and compilers, examine this Web Page:
//   http://www.kai.com/oopack/oopack.html
//
//   Report your results by sending e-mail to oopack@kai.com.
//   For a run to be accepted, adjust the number of iterations for each test
//   so that each time reported is greater than 10 seconds.
//
//   Send this output, along with:
//

```

```

//      * your
//      + name -----
//      + company/institution ----
//
//      * the compiler
//      + name -----
//      + version number -----
//      + options used -----
//
//      * the operating system
//      + name -----
//      + version number -----
//
//      * the machine
//      + manufacturer -----
//      + model number -----
//      + processor clock speed --
//      + cache memory size -----
//
//
//      Test          Iterations      Seconds      Mflops
//      ----          -
//      Max           5000      1.3  1.3   3.8  4.0   1.0
//      Matrix        50       1.5  2.8   8.6  4.5   1.9
//      Complex       2000     1.5  5.3  10.8  3.0   3.6
//      Iterator      5000     1.1  1.6   9.4  6.3   1.5
//
//      The "Test" column gives the names of the four tests that are run.
//      The "Iterations" column gives the number of iterations that a test
//      was run. The The two "Seconds" columns give the C-style
//      and OOP-style running times for a test. The two "Mflops" columns
//      give the corresponding megaflop rates. The "Ratio" column gives
//      the ratio between the times. The value of 1.5 at the bottom, for
//      example, indicates that the OOP-style code for Iterator ran 1.5 times
//      more slowly than the C-style code.
//
//      Beware that a low "Ratio" could indicate either that the OOP-style
//      code is compiled very well, or that the C-style code is compiled poorly.
//      OOPACK performance figures for KAI's Photon C++ and some other compilers
//      can be found in http://www.kai.com/oopack/oopack.html.
//
//      Revision History
//      9/17/93  Version 1.0 released
//      10/ 5/93 Allow results to be printed even if checksums do not match.
//      10/ 5/93 Increased "Tolerance" to allow 10-second runs on RS/6000.
//      10/ 5/93 Version 1.1 released
//      1/10/94 Change author's address from Shell to KAI
//      1/13/94 Added #define's for conditional compilation of individual tests
//      1/21/94 Converted test functions to virtual members of class Benchmark.
//      10/11/94 Added routine to inform user of command-line usage.
//      10/11/94 Version 1.5 released.
//      11/21/95 V1.6 Added "mail results to oopack@kai.com" message in output
//      11/28/95 V1.7 Added company/institution to requested information
//
//=====

```

```

#include <assert.h>
#include <ctype.h>
#include <float.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>

//
// The source-code begins with the benchmark computations themselves and
// ends with code for collecting statistics. Each benchmark 'Foo' is
// a class FooBenchmark derived from class Benchmark. The relevant methods
// are:
//
//     init - Initialize the input data for the benchmark
//
//     c_style - C-style code
//
//     oop_style - OOP-style code
//
//     check - computes number of floating-point operations and a checksum.
//
const int BenchmarkListMax = 4;

class Benchmark {
public:
    void time_both( int iteration_count ) const;
    void time_one( void (Benchmark::*function)() const, int iterations,
                  double& sec, double& Mflop, double& checksum ) const;
    virtual const char * name() const = 0;
    virtual void init() const = 0;
    virtual void c_style() const = 0;
    virtual void oop_style() const = 0;
    virtual void check(int iterations, double& flops, double& checksum)
        const = 0;
    static Benchmark * find( const char * name );
private:
    static Benchmark * list[BenchmarkListMax];
    static int count;
protected:
    Benchmark() {list[count++] = this;}
};

// The initializer for Benchmark::count *must* precede the declarations
// of derived of class Benchmark.
int Benchmark::count = 0;
Benchmark * Benchmark::list[BenchmarkListMax];

//
// The 'iterations' argument is the number of times that the benchmark
// computation was called. The computed checksum that ensures that the
// C-style code and OOP code are computing the same result. This
// variable also prevents really clever optimizers from removing the
// the guts of the computations that otherwise would be unused.
//

```



```

// Each of the following symbols must be defined to enable a test, or
// undefined to disable a test. The reason for doing this with the
// preprocessor is that some compilers may choke on specific tests.
#define HAVE_MAX 1
#define HAVE_MATRIX 1
#define HAVE_COMPLEX 1
#define HAVE_ITERATOR 1

const int N = 1000;

#if HAVE_MAX
//=====
//
// Max benchmark
//
// This benchmark measures how well a C++ compiler inlines a function that
// returns the result of a comparison.
//
// The functions C_Max and OOP_Max compute the maximum over a vector.
// The only difference is that C_Max writes out the comparison operation
// explicitly, and OOP_Max calls an inline function to do the comparison.
//
// This benchmark is included because some compilers do not compile
// inline functions into conditional branches as well as they might.
//
const int M = 100;           // Dimension of vector
double U[M];                // The vector
double MaxResult;          // Result of max computation

class MaxBenchmark: public Benchmark {
private:
    const char * name() const {return "Max";}
    void init() const;
    void c_style() const;
    void oop_style() const;
    void check( int iterations, double& flops, double& checksum ) const;
} TheMaxBenchmark;

void MaxBenchmark::c_style() const           // Compute max of vector (C-style)
{
    double max = U[0];
    for( int k=1; k<M; k++ )                // Loop over vector elements
        if( U[k] > max )
            max=U[k];
    MaxResult = max;
}

inline int Greater( double i, double j )
{
    return i>j;
}

void MaxBenchmark::oop_style() const        // Compute max of vector (OOP-style)
{
    double max = U[0];

```

```

        for( int k=1; k<M; k++ )           // Loop over vector elements
            if( Greater( U[k], max ) )
                max=U[k];
        MaxResult = max;
    }

void MaxBenchmark::init() const
{
    for( int k=0; k<M; k++ )
        U[k] = k&1 ? -k : k;
}

void MaxBenchmark::check(int iterations, double& flops, double& checksum) const
{
    flops = (double)M*iterations;
    checksum = MaxResult;
}
#endif /* HAVE_MAX */

#ifdef HAVE_MATRIX
//=====
//
// Matrix benchmark
//
// This benchmark measures how well a C++ compiler performs constant
// propagation and strength-reduction on classes. C_Matrix multiplies
// two matrices using C-style code; OOP_Matrix does the same with
// OOP-style code. To maximize performance on most RISC processors, the
// benchmark requires that the compiler perform strength-reduction and
// constant-propagation in order to simplify the indexing calculations in
// the inner loop.
//
const int L = 50;           // Dimension of (square) matrices.

double C[L*L], D[L*L], E[L*L]; // The matrices to be multiplied.

class MatrixBenchmark: public Benchmark {
private:
    const char * name() const {return "Matrix";}
    void init() const;
    void c_style() const;
    void oop_style() const;
    void check( int iterations, double& flops, double& checksum ) const;
} TheMatrixBenchmark;

void MatrixBenchmark::c_style() const { // Compute E=C*D with C-style code.
    for( int i=0; i<L; i++ )
        for( int j=0; j<L; j++ ) {
            double sum = 0;
            for( int k=0; k<L; k++ )
                sum += C[L*i+k]*D[L*k+j];
            E[L*i+j] = sum;
        }
}
}

```

```

// Class Matrix represents a matrix stored in row-major format (same as C).
class Matrix {
private:
    double *data;          // Pointer to matrix data
public:
    int rows, cols;       // Number of rows and columns

    Matrix( int rows_, int cols_, double * data_ ) :
        rows(rows_), cols(cols_), data(data_)
    {}

    double& operator()( int i, int j ) { // Access element at row i, column j
        return data[cols*i+j];
    }
};

void MatrixBenchmark::oop_style() const{ // Compute E=C*D with OOP-style code.
    Matrix c( L, L, C );                // Set up three matrices
    Matrix d( L, L, D );
    Matrix e( L, L, E );
    for( int i=0; i<e.rows; i++ )      // Do matrix-multiplication
        for( int j=0; j<e.cols; j++ ) {
            double sum = 0;
            for( int k=0; k<e.cols; k++ )
                sum += c(i,k)*d(k,j);
            e(i,j) = sum;
        }
}

void MatrixBenchmark::init() const
{
    for( int j=0; j<L*L; j++ ) {
        C[j] = j+1;
        D[j] = 1.0/(j+1);
    }
}

void MatrixBenchmark::check(int iterations, double& flops, double& checksum)const
{
    double sum = 0;
    for( int k=0; k<L*L; k++ )
        sum += E[k];
    checksum = sum;
    flops = 2.0*L*L*L*iterations;
}
#endif /* HAVE_MATRIX */

#if HAVE_ITERATOR
//=====
//
// Iterator benchmark
//
// Iterators are a common abstraction in object-oriented programming, which
// unfortunately may incur a high cost if compiled inefficiently.
// The iterator benchmark below computes a dot-product using C-style code
// and OOP-style code. All methods of the iterator are inline, and in

```

```

// principle correspond exactly to the C-style code.
//
// Note that the OOP-style code uses two iterators, but the C-style
// code uses a single index. Good common-subexpression elimination should,
// in principle, reduce the two iterators to a single index variable, or
// conversely, good strength-reduction should convert the single index into
// two iterators!
//
double A[N];
double B[N];
double IteratorResult;

class IteratorBenchmark: public Benchmark {
private:
    const char * name() const {return "Iterator";}
    void init() const;
    void c_style() const;
    void oop_style() const;
    void check( int iterations, double& flops, double& checksum ) const;
} TheIteratorBenchmark;

void IteratorBenchmark::c_style() const // Compute dot-product with C-style code
{
    double sum = 0;
    for( int i=0; i<N; i++ )
        sum += A[i]*B[i];
    IteratorResult = sum;
}

class Iterator { // Iterator for iterating over array of double
private:
    int index; // Index of current element
    const int limit; // 1 + index of last element
    double * const array; // Pointer to array
public:
    double look() {return array[index];} // Get current element
    void next() {index++;} // Go to next element
    int done() {return index>=limit;} // True iff no more elements
    Iterator( double * array1, int limit1 ) :
        array(array1),
        limit(limit1),
        index(0)
    {}
};

// Compute dot-product with OOP-style code
void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    for( Iterator ai(A,N), bi(B,N); !ai.done(); ai.next(), bi.next() )
        sum += ai.look()*bi.look();
    IteratorResult = sum;
}

void IteratorBenchmark::init() const
{

```

```

    for( int i=0; i<N; i++ ) {
        A[i] = i+1;
        B[i] = 1.0/(i+1);
    }
}

void IteratorBenchmark::check(int iterations, double& flops,
                              double& checksum ) const {
    flops = 2*N*iterations;
    checksum = IteratorResult;
}
#endif /* HAVE_ITERATOR */

#if HAVE_COMPLEX
//=====
//
// Complex benchmark
//
// Complex numbers are a common abstraction in scientific programming.
// This benchmark measures how fast they are in C++ relative to the same
// calculation done by explicitly writing out the real and imaginary parts.
// The calculation is a complex-valued ‘SAXPY’ operation.
//
// The complex arithmetic is all inlined, so in principle the code should
// run as fast as the version using explicit real and imaginary parts.
//
class ComplexBenchmark: public Benchmark {
private:
    const char * name() const {return "Complex";}
    void init() const;
    void c_style() const;
    void oop_style() const;
    void check( int iterations, double& flops, double& checksum ) const;
} TheComplexBenchmark;

class Complex {
public:
    double re, im;
    Complex( double r, double i ) : re(r), im(i) {}
    Complex() {}
};

inline Complex operator+( Complex a, Complex b )          // Complex add
{
    return Complex( a.re+b.re, a.im+b.im );
}

inline Complex operator*( Complex a, Complex b )          // Complex multiply
{
    return Complex( a.re*b.re-a.im*b.im, a.re*b.im+a.im*b.re );
}

Complex X[N], Y[N];                                     // Arrays used by benchmark

void ComplexBenchmark::c_style() const // C-style complex-valued SAXPY operation
{

```

```

    double factor_re = 0.5;
    double factor_im = 0.86602540378443864676;
    for( int k=0; k<N; k++ ) {
        Y[k].re = Y[k].re + factor_re*X[k].re - factor_im*X[k].im;
        Y[k].im = Y[k].im + factor_re*X[k].im + factor_im*X[k].re;
    }
}

// OOP-style complex-valued SAXPY operation
void ComplexBenchmark::oop_style() const
{
    Complex factor( 0.5, 0.86602540378443864676 );
    for( int k=0; k<N; k++ )
        Y[k] = Y[k] + factor*X[k];
}

void ComplexBenchmark::init() const
{
    for( int k=0; k<N; k++ ) {
        X[k] = Complex( k+1, 1.0/(k+1) );
        Y[k] = Complex( 0, 0 );
    }
}

void ComplexBenchmark::check(int iterations, double& flops, double& checksum)
const{
    double sum = 0;
    for( int k=0; k<N; k++ )
        sum += Y[k].re + Y[k].im;
    checksum = sum;
    flops = 8*N*iterations;
}
#endif /* HAVE_COMPLEX */

//=====
// End of benchmark computations.
//=====

// All the code below is for running and timing the benchmarks.
#if defined(sun4) && !defined(CLOCKS_PER_SEC)
// Sun/4 include-files seem to be missing CLOCKS_PER_SEC.
#define CLOCKS_PER_SEC 1000000
#endif

//
// TimeOne
//
// Time a single benchmark computation.
//
// Inputs
//     function = pointer to function to be run and timed.
//     iterations = number of times to call function.
//
// Outputs
//     sec = Total number of seconds for calls of function.
//     Mflop = Megaflop rate of function.

```

```

//      checksum = checksum computed by function.
//
void Benchmark::time_one(void (Benchmark::*function)() const, int iterations,
                        double& sec, double& Mflop, double& checksum ) const
{
    // Initialize and run code once to load caches
    init();
    (this->*function)();

    // Initialize and run code.
    init();
    clock_t t0 = clock();
    for( int k=0; k<iterations; k++ )
        (this->*function)();
    clock_t t1 = clock();

    // Update checksum and compute number of floating-point operations.
    double flops;
    check( iterations, flops, checksum );

    sec = (t1-t0) / (double)CLOCKS_PER_SEC;
    Mflop = flops/sec*1e-6;
}

//
// The variable ‘‘C_Seconds’’ is the time in seconds in which to run the
// C-style benchmarks.
//
double C_Seconds = 1;

//
// The variable ‘‘Tolerance’’ is the maximum allowed relative difference
// between the C and OOP checksums. Machines with multiply-add
// instructions may produce different answers when they use those
// instructions rather than separate instructions.
//
// There is nothing magic about the 32, it’s just the result of tweaking.
//
const double Tolerance = 64*DBL_EPSILON;

Benchmark * Benchmark::find( const char * name ) {
    for( int i=0; i<count; i++ )
        if( strcmp( name, list[i]->name() )== 0 )
            return list[i];
    return NULL;
}

//
// Benchmark::time_both
//
// Runs the C and Oop versions of a benchmark computation, and print the
// results.
//
// Inputs
//      name = name of the benchmark

```

```

//      c_style = benchmark written in C-style code
//      oop_style = benchmark written in OOP-style code
//      check = routine to compute checksum on answer
//
void Benchmark::time_both( int iterations ) const {
    // Run the C-style code.
    double c_sec, c_Mflop, c_checksum;
    time_one( &Benchmark::c_style, iterations, c_sec, c_Mflop, c_checksum );

    // Run the OOP-style code.
    double oop_sec, oop_Mflop, oop_checksum;
    time_one(&Benchmark::oop_style, iterations, oop_sec, oop_Mflop, oop_checksum);

    // Compute execution-time ratio of OOP to C. This is also the
    // reciprocal of the Megaflop ratios.
    double ratio = oop_sec/c_sec;

    // Compute the absolute and relative differences between the checksums
    // for the two codes.
    double diff = c_checksum - oop_checksum;
    double min = c_checksum < oop_checksum ? c_checksum : oop_checksum;
    double rel = diff/min;

    // If the relative difference exceeds the tolerance, print an error-message,
    // otherwise print the statistics.
    if( rel > Tolerance || rel < -Tolerance ) {
        printf("%-10s: warning: relative checksum error of %g"
            " between C (%g) and oop (%g)\n",
            name(), rel, c_checksum, oop_checksum );
    }
    printf( "%-10s %10d %5.1f %5.1f %5.1f %5.1f %5.1f\n",
        name(), iterations, c_sec, oop_sec, c_Mflop, oop_Mflop, ratio );
}

const char * Version = "Version 1.7"; // The OOPACK version number

void Usage( int /*argc*/, char * argv[] ) {
    printf( "Usage:\t%s test1=iterations1 test2=iterations2 ...\n", argv[0] );
    printf( "E.g.:\ta.out Max=5000 Matrix=50 Complex=2000 Iterator=5000\n" );
    exit(1);
}

int main( int argc, char * argv[] )
{
    // The available benchmarks are automatically put into the list of available
    // benchmarks by the constructor for Benchmark.

    // Check if user does not know command-line format.
    if( argc==1 ) {
        Usage( argc, argv );
    }
    int i;
    for( i=1; i<argc; i++ ) {
        if( !isalpha(argv[i][0]) )
            Usage( argc, argv );
    }
}

```



```

// Print the request for results
printf("\n");
printf("OOPACK %s\n",Version);
printf("\n");
printf("For results on various systems and compilers, examine this Web");
printf("Page:\n  http://www.kai.com/oopack/oopack.html\n");
printf("\n");
printf("Report your results by sending e-mail to oopack@kai.com.\n");
printf("For a run to be accepted, adjust the number of iterations for");
printf(" each test\nso that each time reported is greater than 10 seconds.");
printf("\n\n");
printf("Send this output, along with:\n");
printf("\n");
printf("    * your\n");
printf("      + name ----- \n");
printf("      + company/institution ---- \n");
printf("\n");
printf("    * the compiler\n");
printf("      + name ----- \n");
printf("      + version number ----- \n");
printf("      + options used ----- \n");
printf("\n");
printf("    * the operating system\n");
printf("      + name ----- \n");
printf("      + version number ----- \n");
printf("\n");
printf("    * the machine\n");
printf("      + manufacturer ----- \n");
printf("      + model number ----- \n");
printf("      + processor clock speed -- \n");
printf("      + cache memory size ----- \n");
printf("\n");

// Print header.
printf("%-10s %11s %11s %5s\n", "", "", "Seconds ", "Mflops ", "" );
printf("%-10s %10s %5s %5s %5s %5s %5s\n",
       "Test", "Iterations", " C ", "OOP", " C ", "OOP", "Ratio" );
printf("%-10s %10s %11s %11s %5s\n", "-----", "-----",
       "-----", "-----", "-----" );

for( i=1; i<argc; i++ ) {
    const char * test_name = strtok( argv[i], "=" );
    const char * rhs = strtok( NULL, "" );
    if( rhs==NULL ) {
        printf("missing iteration count for test '%s'\n", test_name );
    } else {
        int test_count = (int)strtol( rhs, 0, 0 );
        Benchmark * b = Benchmark::find( test_name );
        if( b==NULL ) {
            printf("skipping non-existent test = '%s'\n", test_name );
        } else {
            b->time_both( test_count );
        }
    }
}
}

```

```

    /* Print blank line. */
    printf("\n");

    return 0;
}

```

The benchmark program for OpenC++ (only difference)

```

// Matrix benchmark

metaclass Matrix : MatrixClass;

// Class Matrix represents a matrix stored in row-major format (same as C).
class Matrix {
private:
    double *data;          // Pointer to matrix data
public:
    int rows, cols;       // Number of rows and columns

    Matrix( int rows_, int cols_, double * data_ ) :
        rows(rows_), cols(cols_), data(data_)
    {}

    double* Data() { return data; }
    double& operator()( int i, int j ) { // Access element at row i, column j
        return data[cols*i+j];
    }
};

void MatrixBenchmark::oop_style() const { // Compute E=C*D with OOP-style code.
    Matrix c( L, L, C );                // Set up three matrices
    Matrix d( L, L, D );
    Matrix e( L, L, E );

    c.foreach(i){
        for(int j = 0; j < e.cols; ++j){
            double sum = 0;
            d.foreach(k){
                sum += c(k) * d(j);
            };

            e(i,j) = sum;
        }
    };
}

// Iterator benchmark

metaclass Iterator : IteratorClass;

class Iterator {
private:
    int index;                // Index of current element
    const int limit;         // 1 + index of last element
    double * const array;    // Pointer to array
};

```

```

public:
    double look() {return array[index];} // Get current element
    void next() {index++;} // Go to next element
    int done() {return index>=limit;} // True iff no more elements
    Iterator( double * array1, int limit1 ) :
        array(array1),
        limit(limit1),
        index(0)
    {}
    const int Limit() { return limit; }
    double Array(int i) { return array[i]; }
};

// Compute dot-product with OOP-style code
void IteratorBenchmark::oop_style() const
{
    double sum = 0;
    Iterator ai(A,N), bi(B,N);
    ai.foreach(v){
        sum += v * bi.look();
        bi.next();
    };

    IteratorResult = sum;
}

```

Meta-level program

```

#include "../mop.h"

// Max benchmark

// I need a function template...

// Matrix benchmark

class MatrixClass : public Class {
public:
    MatrixClass(Ptree* d, Ptree* m) : Class(d, m) {}
    static BOOL Initialize();
    Ptree* TranslateUserStatement(Environment*, Ptree*, Ptree*, Ptree*,
                                Ptree*);
    Ptree* TranslateFunctionCall(Environment*, Ptree*, Ptree*);
};

class ForeachData
: public Environment::ClientData {
public:
    ForeachData(Ptree* p) { tempname = p; }
    Ptree* tempname;
};

BOOL MatrixClass::Initialize()
{
    RegisterNewWhileStatement("foreach");
}

```

```

    return TRUE;
}

Ptree* MatrixClass::TranslateUserStatement(Environment* env, Ptree* object,
                                           Ptree* op,
                                           Ptree* keyword, Ptree* rest)
{
    Ptree *index, *body, *tmp, *tmp2, *tmp3, *body2;

    if(!Ptree::Eq(keyword, "foreach"))
        return Class::TranslateUserStatement(env, object, op, keyword,
                                             rest);

    if(object->WhatIs() != LeafNameId){
        ErrorMessage("sorry, the object field must be a variable name",
                    keyword);
        return nil;
    }

    if(!Ptree::Match(rest, "[([?]) %?", &index, &body)){
        ErrorMessage("invalid foreach statement");
        return nil;
    }

    tmp = Ptree::GenSym();
    tmp2 = Ptree::GenSym();
    tmp3 = Ptree::GenSym();
    ForeachData* data = new ForeachData(tmp3);

    env->AddClientData(this, object, data);
    body2 = TranslateStatement(env, body);
    env->DeleteClientData(data);

    return Ptree::MakeStatement(
        "for(int %p = %p.rows, %p = %p.cols, %p = (%p.rows - 1) * %p.cols; "
        "    "--%p >= 0; %p -= %p){\n"
        "    double const* %p = &(%p%pData())[%p];\n"
        "    %p }\n",
        index, object, tmp, object, tmp2, object, object,
        index, tmp2, tmp,
        tmp3, object, op, tmp2, body2);
}

Ptree* MatrixClass::TranslateFunctionCall(Environment* env, Ptree* object,
                                           Ptree* args)
{
    ForeachData* ent = (ForeachData*)env->LookupClientData(this, object);
    if(ent == nil)
        return Class::TranslateFunctionCall(env, object, args);
    else
        return Ptree::Make("%p[%p]", ent->tempname,
                           Ptree::First(Ptree::Second(args)));
}

// Iterator benchmark

```

```

class IteratorClass : public Class {
public:
    IteratorClass(Ptree* d, Ptree* m) : Class(d, m) {}
    static BOOL Initialize();
    Ptree* TranslateUserStatement(Environment*, Ptree*, Ptree*, Ptree*,
                                Ptree*);
};

BOOL IteratorClass::Initialize()
{
    RegisterNewWhileStatement("foreach");
    return TRUE;
}

Ptree* IteratorClass::TranslateUserStatement(Environment* env, Ptree* object,
                                             Ptree* op,
                                             Ptree* keyword, Ptree* rest)
{
    Ptree *arg, *body, *index, *limit;

    if(!Ptree::Eq(keyword, "foreach"))
        return Class::TranslateUserStatement(env, object, op, keyword,
                                             rest);

    if(object->WhatIs() != LeafNameId){
        ErrorMessage("sorry, the object field must be a variable name",
                    keyword);
        return nil;
    }

    if(!Ptree::Match(rest, "[([%?]) %?]", &arg, &body)){
        ErrorMessage("invalid foreach statement");
        return nil;
    }

    index = Ptree::GenSym();
    limit = Ptree::GenSym();

    return Ptree::MakeStatement(
        "for(int %p = 0, %p = %p.Limit(); %p < %p; ++%p){\n"
        "    const double& %p = %p.Array(%p);\n"
        "    %p }\n",
        index, limit, object, index, limit, index,
        arg, object, index,
        TranslateStatement(env, body));
}

// Complex benchmark

class ComplexClass : public Class {
public:
    ComplexClass(Ptree* d, Ptree* m) : Class(d, m) {}
    Ptree* TranslateAssign(Environment*, Ptree*, Ptree*, Ptree*);
    BOOL IsComplex(Environment*, Ptree*);
    BOOL IsArray(Ptree*);
    Ptree* TransEachPart(Environment*, Ptree*, BOOL);
};

```

```

private:
    BOOL giveUp;
};

Ptree* ComplexClass::TranslateAssign(Environment* env, Ptree* object,
                                     Ptree* op, Ptree* expr)
{
    Ptree *repart, *impart, *array;

    if(!op->Eq('='))
        goto giveup_opt;

    if(!object->IsLeaf() && !IsArray(object))
        goto giveup_opt;

    if(expr->IsLeaf())          // e.g. a = b;
        goto giveup_opt;

    giveUp = FALSE;
    repart = TransEachPart(env, expr, TRUE);
    impart = TransEachPart(env, expr, FALSE);

    if(!giveUp)
        return Ptree::Make("%p.re = %p,\n%p.im = %p",
                           object, repart, object, impart);

giveup_opt:          // give up optimization
    return Class::TranslateAssign(env, object, op, expr);
}

BOOL ComplexClass::IsComplex(Environment* env, Ptree* var)
{
    Class* metaobj = env->LookupClassMetaobject(var);
    return BOOL(metaobj == this);
}

BOOL ComplexClass::IsArray(Ptree* expr)
{
    Ptree* array;

    return BOOL(Ptree::Match(expr, "[%? [%[ %* %]]]", &array)
                && array->IsLeaf());
}

Ptree* ComplexClass::TransEachPart(Environment* env, Ptree* expr,
                                    BOOL real_part)
{
    Ptree* lexpr;
    Ptree* rexpr;
    Ptree *l_re, *l_im, *r_re, *r_im;

    if(expr->IsLeaf()){
        if(IsComplex(env, expr))
            if(real_part)
                return Ptree::Make("%p.re", expr);
    }
}

```

```

        else
            return Ptree::Make("%p.im", expr);
    else
        return expr;
}
else if(IsArray(expr)){
    TypeInfo t;
    Ptree* expr2 = TranslateExpression(env, expr, t);
    if(t.ClassMetaobject() == this)
        if(real_part)
            return Ptree::Make("%p.re", expr2);
        else
            return Ptree::Make("%p.im", expr2);
    else
        return expr2;
}
else if(Ptree::Match(expr, "[%? + %?]", &lexpr, &rexpr))
    return Ptree::Make("%p+%p", TransEachPart(env, lexpr, real_part),
        TransEachPart(env, rexpr, real_part));
else if(Ptree::Match(expr, "[%? - %?]", &lexpr, &rexpr))
    return Ptree::Make("%p-%p", TransEachPart(env, lexpr, real_part),
        TransEachPart(env, rexpr, real_part));
else if(Ptree::Match(expr, "[( %? )]", &lexpr))
    return Ptree::Make("(%p)", TransEachPart(env, lexpr, real_part));
else if(Ptree::Match(expr, "[- %?]", &rexpr))
    return Ptree::Make("-%p", TransEachPart(env, rexpr, real_part));
else if(Ptree::Match(expr, "[%? * %?]", &lexpr, &rexpr)){
    l_re = TransEachPart(env, lexpr, TRUE);
    l_im = TransEachPart(env, lexpr, FALSE);
    r_re = TransEachPart(env, rexpr, TRUE);
    r_im = TransEachPart(env, rexpr, FALSE);
    if(real_part)
        return Ptree::Make("%p*%p-%p*%p", l_re, r_re, l_im, r_im);
    else
        return Ptree::Make("%p*%p+%p*%p", l_re, r_im, l_im, r_re);
}
else {
    giveUp = TRUE;
    return expr;
}
}

```