

Chapter 1

Object-oriented features

The compiler's job for a procedural language like C is relatively straightforward, because C and most other compiled procedural languages have been designed to approximate what the computer does. Other language paradigms, however, are more independent of computer architecture, and so they are more difficult for compilers to translate, particularly if the efficiency of the translation is important.

Among these are the object-oriented paradigm, which presents a host of new challenges for compilers. In this section, we'll look at how a compiler can handle these challenges, using translations from Java to Intel assembly language for our examples.

1.1 Basics

An object-oriented programmer tends to think of an object as a conglomeration of variables and methods. In memory, however, each individual object of the same class shares the same methods, and so there is no reason to store methods with individual instances.¹ Thus, the representation of an object in memory includes the object's instance variable values. The memory representation of an instance is called its **class instance record**, or CIR.

Consider, for example, the following class declaration.

```
class Container {
    int capacity;
    int contents;

    Container(int capacity) {
        this.capacity = capacity;
        this.contents = 0;
    }

    int getRemaining() {
        return capacity - contents;
    }

    void add(int weight) {
        if(contents + weight < capacity) contents += weight;
    }
}
```

⁰©2003, Carl Burch. Copies may not be distributed in any form without permission.

¹This is not true for all object-oriented languages. Some allow the methods for an object to be altered over time. Java does not, due to the memory and the inefficiencies of associating methods with each object.

A Java compiler will decide that each Container needs to hold two integer values, for the instance variables `capacity` and `contents`. Thus, it will reason, it must allocate eight bytes for each Container instance. For the moment, we'll imagine that each Container CIR has only these eight bytes. (We'll see that there's more to it later.)

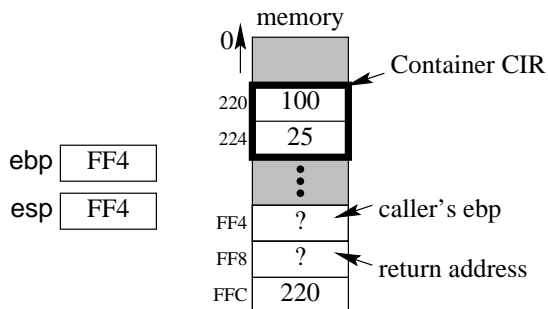
An instance method is similar to a function. There will be a subroutine at a fixed location in memory, and the compiled code can call this subroutine when the method is to be executed. One difference is that the code for the instance method needs to know which object it is addressing (that is, what `this` is). This can be done by giving each instance method an implicit first parameter, the address of `this`.

Thus, a compiler might produce the following for Container's `getRemaining` instance method.

```

1  Container_getRemaining:
2      pushl %ebp
3      movl %esp, %ebp
4      movl 8(%ebp), %ecx      # load memory address of this into ecx
5      movl 0(%ecx), %eax     # load this.capacity into eax
6      subl 4(%ecx), %eax     # subtract this.contents from eax
7      movl %ebp, %esp       # and return this value
8      popl %ebp
9      ret
    
```

After completing the entry template for this method, the memory would look something like the following.



In line 4, the method loads the implicit first parameter `this`, the address of a Container object, into a register. Line 5 illustrates the compiler fetching an instance variable from an object: it loads the `capacity` variable, which we're supposing is the first thing in a Container CIR. The `contents` variable, loaded in line 6, would be four bytes beyond this.

Constructor methods are similar.

```

1  Container_construct:
2      pushl %ebp
3      movl %esp, %ebp
4      movl 8(%ebp), %eax     # load memory address of this into ecx
5      movl 12(%ebp), %ecx   # copy capacity parameter into this.capacity
6      movl %ecx, 0(%eax)
7      movl $0, 4(%eax)     # put 0 into this.contents
8      movl %ebp, %esp
9      popl %ebp
10     ret
    
```

```

class Suitcase extends Container {
    boolean closed;

    Suitcase(int capacity) {
        super(capacity);
        this.closed = true;
    }

    void add(int weight) { if(!this.closed) super.add(weight); }

    void open() { this.closed = false; }

    void close() { this.closed = true; }
}

```

Figure 1.1: The Suitcase class.

Constructor methods are slightly different when they are called, however: Before calling a constructor method, the code must allocate space for the CIR. Suppose the compiler were to compile “new Container(100).” Since a Container occupies eight bytes, the first task is to allocate eight bytes of memory.

```

    pushl $8                # first we call malloc to allocate 8 bytes
    call malloc
    movl %eax, %ebx        # save the address in a callee-save register
    pushl $100            # now we call the constructor method
    pushl %eax            # the Container's address is the first parameter
    call Container_construct

```

Methods, then, aren’t very different from functions. Class methods are even less different — they don’t even require the implicit parameter.

Many object-oriented language features are irrelevant to compilation. Notable among these is the issue of **data protection**. It is a central feature of object-oriented programming. But protecting data (using the `private` keyword, for example) has no implication for the compiled code. This information is used only during compilation, when the compiler should flag protection violations. Because the compiler ensures that there are no protection violations, the executable program it generates needn’t bother with them.

1.2 Polymorphism

Polymorphism refers to the ability of an object of one class to masquerade as an object of a superclass. For example, suppose we define a Suitcase class extending Container, as in Figure 1.1. Then the following code would illustrate polymorphism, since it converts a Suitcase object into a Container object.

```
Container luggage = new Suitcase(100);
```

Like data protection, polymorphism is a central idea of object-oriented programming. Polymorphism, however, has major implications for code generation.

Each Suitcase object has three instance variables: In addition to the `closed` instance variable, it inherits the `capacity` and `contents` instance variables from the Container class. In order to be able to masquerade as a Container object, the compiler needs to place Container’s instance variables first in the CIR. Thus, at the address contained by the `luggage` variable initialized as above, we would find its `capacity` variable’s value. Four bytes beyond this, we would find its `contents` variable’s value. And four

bytes beyond this, we would find its `closed` variable's value. (The program wouldn't be able to access `luggage.closed` directly, since the `luggage` variable is declared as a `Container`, but it would still exist in memory.)

The compiler cannot place the `closed` variable elsewhere in a `Suitcase` CIR. For example, it couldn't go between `capacity` and `closed`. Doing this would prevent the already-compiled `Container` methods from working; for example, the compiled `getRemaining` method we saw relies on `contents` being four bytes beyond the `capacity` value in memory.

If the compiler places instance variables of the superclass before those of the subclass, then all the inherited instance methods work well. Indeed, if the compiler needs to generate code to handle a cast into a superclass, it can simply copy the existing CIR address.

1.3 Virtual methods

An important complication arises when we consider the fact that a class can **override** methods of its superclass. For example, the `Suitcase` class of Figure 1.1 overrides `Container`'s `add()` method. In object-oriented language parlance, a method that can be overridden is a **virtual method**. In Java, nearly (or virtually) all methods are virtual.

Compiling a virtual method isn't a major problem. The problem is in how the computer calls the method. Consider the following code, where `luggage` is a variable declared as a `Container`.

```
luggage.add(25);
```

Without considering overriding, the natural way to compile this is the following.

```
pushl $25
pushl %eax           # push the implicit first parameter, this
call Container_add
```

This is erroneous, though, because while `luggage` is declared as a `Container`, it may actually be a `Suitcase` due to polymorphism. This assembly code would call `Container`'s `add` method, and the 25 pounds would go into the suitcase whether or not its open. But overridden methods are to be in force even when the object has been cast into its parent class — in this case, if the suitcase is closed, the 25 pounds should not go in.

To support this, object-oriented compilers generally used a technique called the **virtual method table**, or VMT. For each class in a program, the compiler generates a fixed table of the virtual methods defined by the class.

| Container VMT | Suitcase VMT |
|---------------|--------------|
| 0 | 0 |
| 4 | 4 |
| 8 | 8 |
| 12 | 12 |

| |
|------------------------|
| Container_getRemaining |
| Container_add |

| |
|------------------------|
| Container_getRemaining |
| Suitcase_add |
| Suitcase_open |
| Suitcase_close |

The `Suitcase` VMT must follow the `Container` VMT for the methods that it inherits, but for methods that it overrides (the `add` method), it contains the address of the overriding method's first instruction instead.

We change the CIR format by always allocating the first four bytes to refer to the VMT of the instance's actual class. Thus, if `luggage` were a `Suitcase`, its CIR contains first the `Suitcase` VMT's memory address, followed by its `capacity` instance variable value, then its `contents` value, then its `closed` value. The constructor method would be responsible for initializing the VMT pointer for each class instance.

The following translation of “`luggage.add(25);`” would support overriding, where `eax` contains the address corresponding to `luggage`.

```

1      movl 0(%eax), %ecx      # load the VMT address for luggage into ecx
2      pushl $25
3      pushl %eax             # push the implicit first parameter, this
4      call *4(%ecx)          # call the add method given in the VMT

```

The first line looks at the memory pointed to by `luggage`; with our redefinition of the CIR format, the data here is the memory address of the VMT for `luggage`’s actual type. If `luggage` is a `Suitcase`, then this line would place the address of the `Suitcase` VMT into `ecx`. When the computer calls the routine in line 4, it determines the routine’s address from the second entry in the `Suitcase` VMT, which would be `Suitcase`’s `add` method. If `luggage` were a plain `Container`, however, then line 1 would load `Container`’s VMT address, and so it would enter `Container`’s `add` method in line 4, since this is where the second entry of `Container`’s VMT points.

1.4 Casting

Java permits a program to explicitly cast an object into another type.

```
Suitcase bag = (Suitcase) luggage; // luggage is a Container
```

Performing the conversion is easy to do, since a `Suitcase` variable and a `Container` variable pointing to the same object refer to the same address. But there is a complication: The Java language requires a `ClassCastException` to be thrown should `luggage` in fact not be a `Suitcase` object. Usually, this can only be done at run-time. Thus, for this line, the compiler must generate code to verify an object’s actual type. What should it generate?

We can solve this problem through the VMT. If `luggage` is actually a `Suitcase`, then the verification process is easy: The VMT pointer in `luggage` will point to `Suitcase`’s VMT, and so the compiler can easily add code to verify that this is in fact true. But what if `luggage` is a subclass of `Suitcase` — like `RollingSuitcase`? Or if it is a subclass of that? In these cases, the cast should still be legal. But the VMT of `luggage` would not be `Suitcase`’s VMT, and so the comparison would indicate that the cast is illegal.

What we can do is to change the format of virtual method tables so that the first four bytes of a class’s VMT will always point to the VMT of its superclass. Told to check whether `luggage` is actually a `Suitcase`, then, the assembly code can look into `luggage`’s VMT. If it doesn’t refer to `Suitcase`’s VMT, then we can check the VMT’s parent pointer, and that VMT’s parent pointer, and so on until finally we get to `Suitcase`’s VMT (in which case the cast is legal) or to `Object`’s VMT (in which case it is not).

1.5 Interfaces

Java also incorporates the concept of the *interface*, a set of methods that must be defined by any class that wants to claim that it implements the interface. Moreover, it permits a variable’s type to be an interface type, whose value can be any class that implements the given interface.

This poses a challenge for object-oriented compilers: If a program calls a method on a variable of an interface type, how can it determine where the method is located? The VMT approach does not apply directly here. With classes, the compiler could assign a VMT index to each virtual method of a class. It could be confident that any subclass would be able to use that same index, because Java requires each class to have at most one parent class, and so there is no potential for conflicts.

But a class can implement many interfaces, and so if the compiler assigned an index to each interface method, there is the potential that another interface would use the same index. Or, if the compiler assigned a unique index to each interface method throughout a program, then the VMT would be prohibitively large.²

How to best handle calling interface methods is still an open research question. Alpern, et al., proposed one good alternative.³ They proposed assigning each interface method both a unique identifier and a random ID between 0 and, say, 4. A class's VMT would include an array of 5 slots, called an *interface method table*; for any call to an interface method, the generated assembly code could find the object's VMT, containing its interface method table. The generated code would call the method found in this table at the method's assigned random ID.

For example, suppose we were to define an `Openable` interface.

```
interface Openable {
    void open();
    void close();
}
```

In compiling this, the compiler would choose a descriptor and a random ID for each method. Suppose it chose 1193 and 2 for the `open()` method and 1194 and 4 for the `close()` method. If we modified the `Suitcase` class of Figure 1.1 to implement the `Openable` interface, the compiler would generate `Suitcase`'s VMT as follows.

| | | |
|----|------------------------|------------------------|
| 0 | Container_VMT | parent class's VMT |
| 4 | 0 | |
| 8 | 0 | |
| 12 | Suitcase_open | interface method table |
| 16 | 0 | |
| 20 | Suitcase_close | |
| 24 | Container_getRemaining | |
| 28 | Suitcase_add | virtual method table |
| 32 | Suitcase_open | |
| 36 | Suitcase_close | |

We've supposed that the compiler chooses to place the address of the parent's VMT first (as required to support casting in Section 1.4), the interface method table next, followed by the regular virtual method table.

Of course, the problem with this system is that some interface methods could receive the same random ID. For this, Alpern, et al., propose that the compiler would generate a *stub method* that would receive a descriptor of the interface method and use this to decide which of the conflicting methods to enter. The address of this stub method would go into the VMT interface method table. In the generated code for calling an interface method, the method descriptor would be stashed before calling the interface method, so that, if the table contains a stub method, it can use the descriptor to choose which conflicting method to enter. We'll suppose that the method descriptor is stashed in the `edx` register.

²Some Java compilers have, in fact, used unique indexes for each interface method. This can only work for programs using a small number of classes and interfaces, however. Beyond this, it proves too wasteful of memory.

³Alpern, et al., "Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless," *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 2001

To compile “`o.open()` ;” where `o` is an `Openable` object and whose address is in `eax`, the compiler might generate the following code.

```

1      movl 0(%eax), %ecx      # load o's VMT
2      movl $1193, %edx      # store open's method descriptor in edx
3      pushl %eax            # call the open method
4      call *12(%ecx)

```

If it happened that both `open()` and `close()` got the same random ID of 2, then the compiler would notice that there is a conflict when it fills out `Suitcase`'s VMT.⁴ Therefore, the compiler would generate the following stub method.

```

100  Suitcase_invoke2:
101      movl 4(%esp), %ecx      # determine the this parameter
102      movl 0(%ecx), %ecx      # determine this's VMT
103      cmpl $1193, %edx      # is the method the open method?
104      je *32(%ecx)          # if so, jump into it
105      jmp *36(%ecx)         # otherwise, jump to close method

```

`Suitcase`'s VMT would refer to this stub method.

| | | |
|----|------------------------|------------------------|
| 0 | Container_VMT | parent class's VMT |
| 4 | 0 | |
| 8 | 0 | |
| 12 | Suitcase_invoke2 | interface method table |
| 16 | 0 | |
| 20 | 0 | |
| 24 | Container_getRemaining | |
| 28 | Suitcase_add | virtual method table |
| 32 | Suitcase_open | |
| 36 | Suitcase_close | |

When the calling code calls the routine in line 4, it would find `Suitcase_invoke2` in the interface method table and go there. The stub method jumps into the `open` or `close` method based on the method descriptor saved in `edx`. One peculiar thing about the stub method is that it jumps into the intended method rather than calling it (as in line 104). This works, because the stub method does not alter the stack in any way, so it will appear to the `open` method that it was called by line 4 directly.⁵

The size of the interface method table would be chosen to balance the expected number of ID conflicts against memory costs. Through experiments on various programs using an interface method table of 40 entries, Alpern, et al., estimate that calling interface methods using their technique took roughly 50% more time than calling class methods.

⁴When avoidable, it would be silly to assign the same random ID to two methods in the same interface, because doing so forces every implementor of the interface to use a stub method. Nonetheless, to save the trouble of following a more complex example, pretend that we have a silly compiler.

⁵Another peculiarity of line 104 is that it uses the virtual method table to determine the location of the `open`. You might wonder: Why not just call `Suitcase.open` directly, since that would save a memory reference? The reason it would find the method indirectly is so that a `Suitcase` subclass can override `open` without necessitating a new stub method.

1.6 Analysis

With a bit of ingenuity and care, the object-oriented features of Java can be compiled to relatively efficient code. This is not entirely surprising, because the efficiency of the compiled code was a major design criterion in selecting which features to include in Java.

Other object-oriented languages (like Smalltalk) were designed with a heavy emphasis on convenient programming and very little emphasis on computational efficiency of the resulting program. For example, Smalltalk provides the capability for a program to add new methods to an object at run-time. This adds new flexibility that Java does not share, but it can wreak tremendous damage with the run-time efficiency, both because methods must be stored with each object and because determining the location of a method requires a more complex process.

The object-oriented ideas introduced by Smalltalk were years ahead of its time — its first idea came in 1972, and object-oriented programming did not really take off until 1990. Though this historical gap was largely due to inertia (for Smalltalk was very different in many other ways too), but it was also because Smalltalk could not possibly compile to efficient programs. Only when Stroustrup developed C++, which took a conventional language and incorporated only those features that could translate to efficient code, did the programming community finally recognize the usefulness of object-oriented constructs.