IMPLEMENTATION OF OBJECT-ORIENTED CONSTRUCTS

There are at least two parts of language support for object-oriented programming

- storage structures for instance variables (Instance Data Storage)
- dynamic bindings of messages to methods

Instance Data Storage

In C++, classes are defined as extensions of C's record structures—structs. This similarity suggests a storage structure for the instance variables of class instances—that of a record. This form of this structure is called a class instance record (CIR). The structure of a CIR is static, so it is built at compile time and used as a template for the creation of the data of class instances.

Every class has its own CIR. When a derivation takes place, the CIR for the subclass is a copy of that of the parent class, with entries for the new instance variables added at the end. Because the structure of the CIR is static, access to all instance variables can be done as it is in records, using constant offsets from the beginning of the CIR instance. This makes these accesses as efficient as those for the fields of records.

12.11.2 Dynamic Binding of Method Calls to Methods

• Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR

– Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR

– The storage structure is sometimes called virtual method tables (vtable)

– Method calls can be represented as offsets from the beginning of the vtable

Consider the following Java example, in which all methods are dynamically bound:

```
public class A
{
public int a, b;
```

```
public void draw() { . . . }

public int area() { . . . }

}

public class B extends A

{

public int c, d;

public void draw() { . . . }

public void sift() { . . . }

}
```

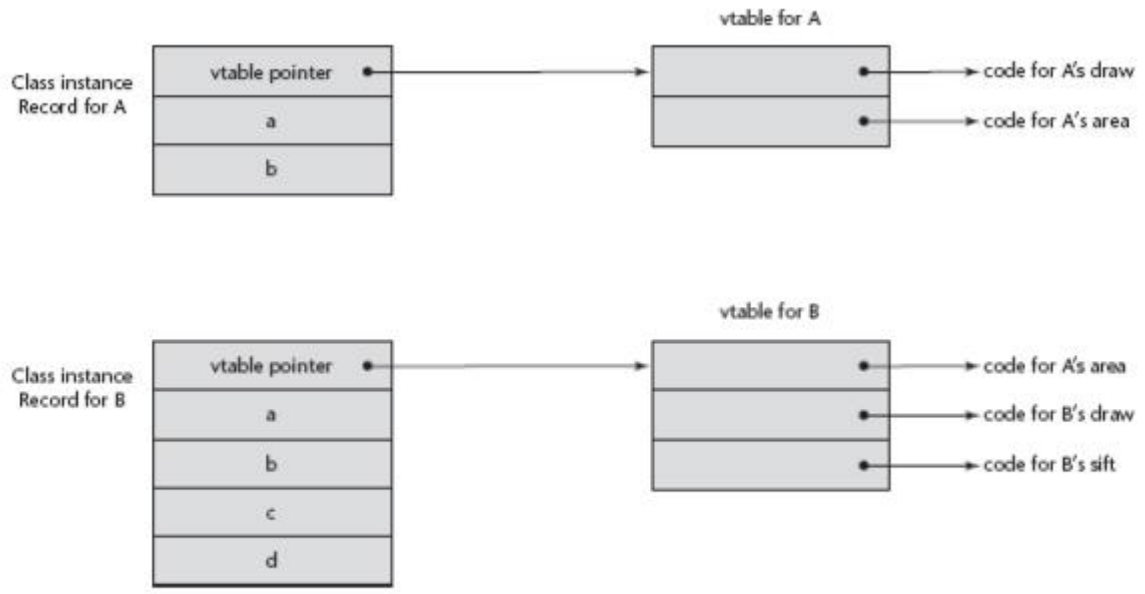

Figure 12.7

An example of the CIRs with single inheritance

The CIRs for the A and B classes, along with their vtables, are shown in Figure 12.7. Notice that the method pointer for the area method in B's vtable points to the code for A's area method. The reason is that B does not override A's area method, so if a client of B calls area, it is the area method inherited from A. On the other hand, the pointers for draw and sift in B's vtable point to B's draw and sift. The draw method is overridden in B and sift is defined as an addition in B.

Multiple inheritance complicates the implementation of dynamic binding.

Consider the following three C++ class definitions:

```
class A {
public:
int a;
virtual void fun() { . . . }
virtual void init() { . . . }
};
class B {
public:
int b;
virtual void sum() { . . . }
};
class C : public A, public B {
public:
int c;
virtual void fun() { . . . }
virtual void dud() { . . . }
};
```

The C class inherits the variable a and the init method from the A class. It redefines the fun method, although both its fun and that of the parent class A are potentially visible through a polymorphic variable (of type A). From B, C inherits the variable b and the sum method. Cdefines

its own variable, c, and defines an uninherited method, dud. A CIR for C must include A's data, B's data, and C's data, as well as some means of accessing all visible methods. Under single inheritance, the CIR would include a pointer to a vtable that has the addresses of the code of all visible methods.

There must also be two vtables: one for the A and C view and one for the B view. The first part of the CIR for C in this case can be the C and A view, which begins with a vtable pointer for the methods of C and those inherited from A, and includes the data inherited from A. Following this in C's CIR is the B view part, which begins with a vtable pointer for the virtual methods of B,which is followed by the data inherited from B and the data defined in C. The CIR for C is shown in Figure 12.8.

An example of a subclass CIR with multiple parents

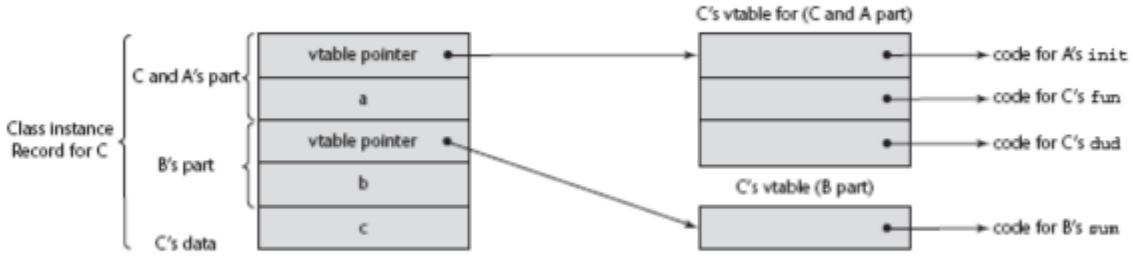

Figure 12.8

An example of a subclass CIR with multiple parents