MODULE 4
Support for Object Oriented Programming

Support for Object Oriented Programming

- Introduction
- Inheritance
- Dynamic Binding
- Design Issues for Object Oriented Languages
- Support for Object Oriented Programming in C++
- Implementation of Object-oriented Constructs

Introduction

- Paradigm Evolution

1. Procedural - 1950s-1970s (procedural abstraction)
2. Data-Oriented - early 1980s (data-oriented)
3. OOP - late 1980s (Inheritance and dynamic binding)

Introduction

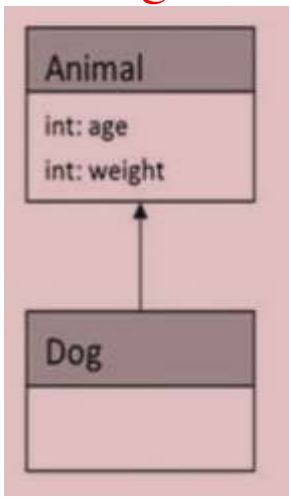
- object-oriented languages include Java, C++, C#, Python, PHP, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, and Smalltalk
- The first object oriented language was Simula-67;
- Smalltalk followed soon after as the first "pure" object-oriented language
- C++ (also supports procedural and data oriented programming)
- Ada 95 (also supports procedural and data oriented programming)
- CLOS (also supports functional programming)
- Scheme (also supports functional programming)

Introduction

- A language that is object oriented must provide support for three key language features:
  - abstract data types,
  - inheritance, and
  - dynamic binding of method calls to methods

Inheritance

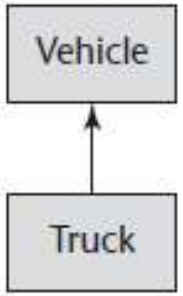
- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class
- Inheritance is an "is-a" relation, which inherits the attributes and behaviours from its parent class.
- For example, dog is an animal. It means animal is a parent class and Dog is the child class.
- The child class "Dog" inherits the attributes like age and weight from the parent class , which is an animal.



- As a simple example of inheritance, consider the following: Suppose we have a class named *Vehicles*, which has variables for year, color, and make.
- A natural specialization, or subclass, of this would be *Truck*, which could inherit the variables from *Vehicle*, but would add variables for hauling capacity and number of wheels.
- Figure 12.1 shows a simple diagram to indicate the relationship between the Vehicle class and the Truck class, in which the arrow points to the parent class.

Figure 12.1

A simple example of inheritance



- Using inheritance we can define a new ***derived*** or ***child class*** based on an existing ***parent class*** or ***superclass***.
- The derived class
  - Inherits all fields and methods of the superclass,
  - Can define additional fields and methods, and
  - Can override existing fields and methods.

Inheritance: Definition

- **inheritance**: a parent-child relationship between classes
  - allows sharing of the behavior of the parent class into its child classes, code sharing between classes through inheritance
  - child class can add new behavior or override existing behavior from parent. The new class is a specialized version of the existing class
- All the data and methods available to the parent class also appear in the child class with the same names

Inheritance terms

- **superclass**, **base class**, **parent class**: terms to describe the parent in the relationship, which shares its functionality

- **subclass**, **derived class**, **child class**: terms to describe the child in the relationship, which accepts functionality from its parent

- **extend**, **inherit**, **derive**: become a subclass of another class

Syntax of Inheritance

```
class widget {
   ...
   void paint();
   ...
};

class push_button : public widget {
   ...
   void paint();
   ...
};
```

- **C++**

  class push_button : public widget { … }

- **Java**

  public class push_button extends widget { … }

- **Ada**

  type push_button is new widget with …

Inheritance - Visibility in Derived Classes

- Derived classes can restrict the visibility of its base class's members in objects of the derived class.

class A : **public** B { … }

- All methods have the same visibility in the derived class as in the base class.

class A : **protected** B { … }

- Public and protected members of the base class become protected in the derived class. Private members remain private.

class A **: private** B { … }

- All members of the base class become private in the derived class.

Class Access Specifiers-In c++

Class access specification: determines how private, protected, and public members of base class are inherited by the derived class

1. Public members are visible anywhere the class declaration is in scope.
2. Private members are visible only inside the class's methods.
3. Protected members are visible inside methods of the class or its descendants.

- If a new class is a subclass of a single parent class, then the derivation process is called **single inheritance**.
- If a class has more than one parent class, the process is called **multiple inheritance**.
- When a number of classes are related through single inheritance, their relationships to each other can be shown in a derivation tree.
- The class relationships in a multiple inheritance can be shown in a derivation graph.

Disadvantage of inheritance

- One disadvantage of inheritance as a means of increasing the possibility of reuse is that it creates dependencies among the classes in an inheritance hierarchy.
- This result works against one of the advantages of abstract data types, which is that they are independent of each other.
- However, it may be difficult, if not impossible, to increase the reusability of abstract data types without creating dependencies among some of them.

Dynamic Binding

- The third characteristic (after abstract data types and inheritance) of object-oriented programming languages is a kind of polymorphism provided by the **dynamic binding** of messages to method definitions. This is sometimes called **dynamic dispatch**.

Static and Dynamic Binding

- STATIC BINDING
- The **binding which can be resolved at compile time by compiler** is known as static or early binding.
- The binding of static, private and final methods is compile-time.
- **These method cannot be overridden and the type of the class is determined at the compile time**
- DYNAMIC BINDING
- **When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding**.
- Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed.
- **The type of object is determined at the run time so this is known as dynamic binding**.

STATIC BINDING IN JAVA

```
class Human{
public static void eat()
    {
System.out.println("Human is
eating");
    }
}
class Boy extends Human{
    //Overriding method
    public static void eat(){
 System.out.println("Boy is
eating");
 }
}
```

```
public class Methover{
public static void main(String
args[])
{
  Boy obj = new Boy();
  Human ob=new Human();
  Human ob1=new Boy();
      obj.eat();
      ob.eat();
      ob1.eat();
   }
}
```

Boy is eating
Human is eating
Human is eating

DYNAMIC BINDING IN JAVA

```
class Human
{
     public  void eat()
   {
System.out.println("Human is eating");
   }
}
class Boy extends Human{
   //Overriding method
   public  void eat(){
 System.out.println("Boy is eating");
 }
}
```

```
public class Methover{
public static void main(String args[])
{
  Boy obj = new Boy();
  Human ob=new Human();
  Human ob1=new Boy();
                obj.eat();
     ob.eat();
     ob1.eat();
   }

}
```

Boy is eating
Human is eating
Boy is eating

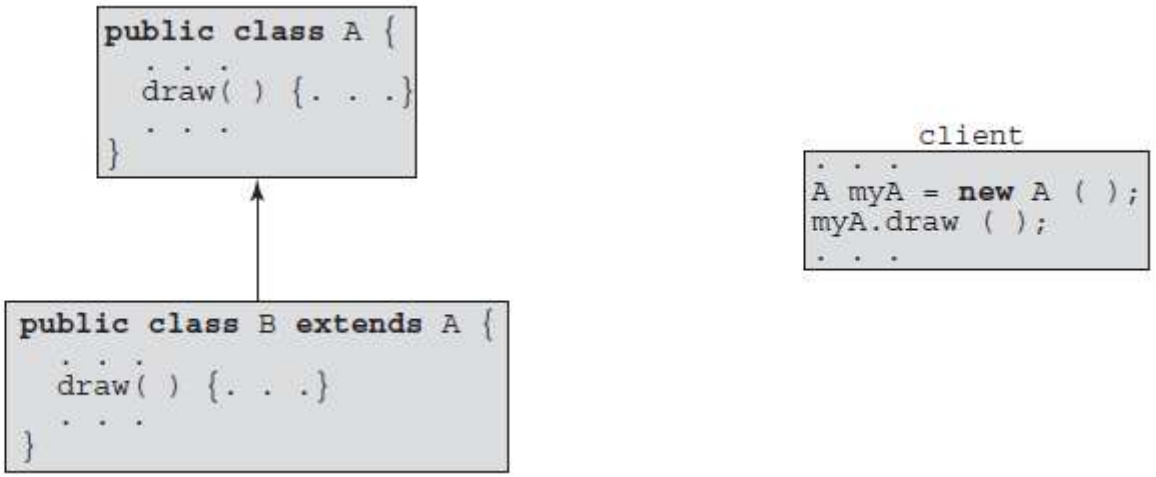
Dynamic Binding

- In inheritance a derived class D has all the members—data and subroutines—of its base class B.
- If D does not hide any of the publicly visible members of B ,it is allowed that an object of class D to be used in any context that expects an object of class B:
- In Ada terminology, a derived class that does not hide any publicly visible members of its base class .So derived class is a subtype of that base class.
- The ability to use a derived class in a context that expects its base class is called subtype polymorphism.

Dynamic Binding

Figure 12.2

Dynamic binding



- Consider the following situation: There is a base class, A, that defines a method **draw** that draws some figure associated with the base class. A second class, B, is defined as a subclass of A. Objects of this new class also need a **draw** method that is like that provided by A but a bit different because the subclass objects are slightly different. So, the subclass **overrides** the inherited draw method.

Static Vs Dynamic binding

- Which is more efficient: static or dynamic method binding?
- Static method binding is more efficient
- **Static method binding:**
  - The method invoked is determined by the type of the variable through which the object is accessed.
  - Languages with static method binding: Simula, C++, Ada 95
- **Dynamic method binding:**
  - The method invoked is determined by the type of the accessed object.
  - Languages with dynamic method binding: Smalltalk, Modula 3, Java, Eiffel

Design Issues for Object-Oriented Languages

- A number of issues must be considered when designing the programming language features to support inheritance and dynamic binding.

1. The Exclusivity of Objects

a. Everything is an object

- advantage - elegance and purity
- disadvantage - slow operations on simple objects (e.g., float)

b. Add objects to a complete typing system

- Advantage - fast operations on simple objects
- Disadvantage - results in a confusing type system

c. Include an imperative-style typing system for primitives but make everything else objects

- Advantage - fast operations on simple objects and a relatively small typing system
- Disadvantage - still some confusion because of the two type systems

2. Are Subclasses Subtypes?

- The issue here is : Does an "is-a" relationship hold between a derived class and its parent class?
- From a purely semantics point of view, if a derived class is a parent class, then objects of the derived class must expose all of the members that are exposed by objects of the parent class.
- An is-a relationship guarantees that in a client a variable of the derived class type could appear anywhere a variable of the parent class type was legal, without causing a type error.
- The derived class objects should be behaviorally equivalent to the parent class objects.
- The subtypes of Ada are examples of this simple form of inheritance for data. For example,

  subtype Small_Int **is** Integer **range** -100..100;

  - Variables of Small_Int type have all of the operations of Integer variables but can store only a subset of the values possible in Integer.
  - Furthermore, every Small_Int variable can be used anywhere an Integer variable can be used. That is, every Small_Int variable is, in a sense, an Integer variable.

3. Single and Multiple Inheritance

- Another simple issue is: Does the language allow multiple inheritance (in addition to single inheritance)?
- Maybe it's not so simple.
- The purpose of multiple inheritance is to allow a new class to inherit from two or more classes.
- Because multiple inheritance is sometimes highly useful, why would a language designer not include it? The reasons lie in two categories:

  complexity and efficiency.

- The additional complexity is illustrated by several problems.

First, note that if a class has two unrelated parent classes and neither defines a name that is defined in the other, there is no problem.

However, suppose a subclass named C inherits from both class A and class B and both A and B define an inheritable method named **display**.

If C needs to reference both versions of display, how can that be done? This ambiguity problem is further complicated when the two parent classes both define identically named methods and one or both of them must be overridden in the subclass.

Another issue arises if both A and B are derived from a common parent, Z, and C has both A and B as parent classes. This situation is called **diamond** or **shared inheritance**.

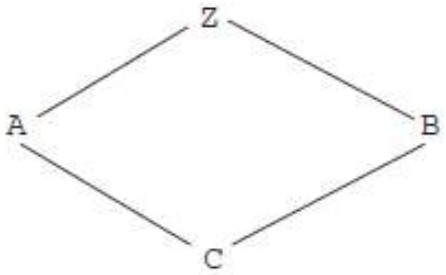
In this case, both A and B should include Z's inheritable variables.

Suppose Z includes an inheritable variable named **sum**. The question is whether C should inherit both versions of sum or just one, and if just one, which one?

There may be programming situations in which just one of the two should be inherited, and others in which both should be inherited.

Figure 12.3

An example of diamond inheritance



- Interfaces are an alternative to multiple inheritance.
- Interfaces provide some of the benefits of multiple inheritance but have fewer disadvantages.

4. Allocation and Deallocation of Objects

- There are two design questions concerning the allocation and deallocation of objects.
- The first of these is the place from which objects are allocated.
- If they behave like the abstract data types, then perhaps they can be allocated from anywhere. If objects are heap dynamic.

- If objects are stack dynamic, there is a problem with regard to subtypes. If class B is a child of class A and B is a subtype of A, then an object of B type can be assigned to a variable of A type.
  - For example, if b1 is a variable of B type and a1 is a variable of A type, then

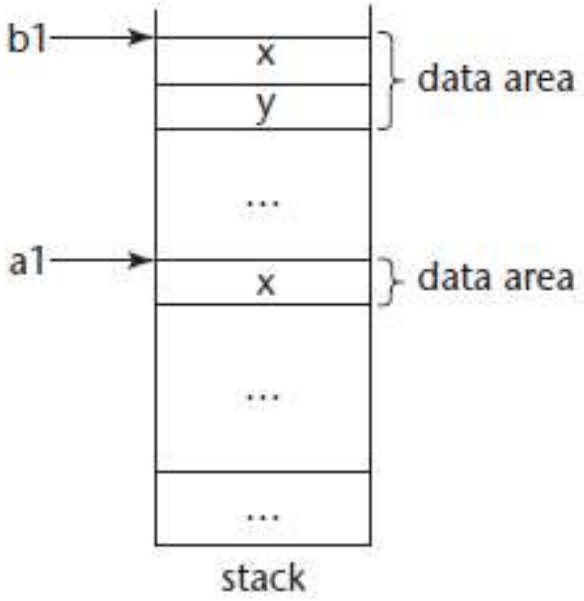
    a1 = b1;

  is a legal statement.
- If a1 and b1 are references to heap-dynamic objects, there is no problem—the assignment is a simple pointer assignment.
- However, if a1 and b1 are stack dynamic, then they are value variables and, if assigned the value of the object, must be copied to the space of the target object. If B adds a data field to what it inherited from A, then a1 will not have sufficient space on the stack for all of b1. The excess will simply be truncated, which could be confusing to programmers who write or use the code. This truncation is called **object slicing.**

```
class A {
   int x;
   . . .
};
class B : A {
   int y;
   . . .
}
```

Figure 12.4

An example of object slicing



- The second question here is concerned with those cases where objects are allocated from the heap.
- The question is whether deallocation is implicit, explicit, or both.
- If deallocation is implicit, some implicit method of storage reclamation is required.
- If deallocation can be explicit, that raises the issue of whether dangling pointers or references can be created.

5. Dynamic and Static Binding

- The dynamic binding of messages to methods is an essential part of object-oriented programming.
- The question here is whether all binding of messages to methods is dynamic.

6. Nested Classes

- The class in which the new class is nested is called the **nesting class**.
- The most obvious design issues associated with class nesting are related to visibility.
- Specifically, one issue is: Which of the facilities of the nesting class are visible in the nested class?
- The other main issue is the opposite: Which of the facilities of the nested class are visible in the nesting class?

7. Initialization of Objects

- The initialization issue is whether and how objects are initialized to values when they are created.
- This is more complicated than may be first thought.
- The first question is whether objects must be initialized manually or through some implicit mechanism.
- When an object of a subclass is created, is the associated initialization of the inherited parent class member implicit or must the programmer explicitly deal with it.

Support for Object Oriented Programming in C++

- C++ was the first widely used object-oriented programming language, and is still among the most popular.

➢ General Characteristics

- To main backward compatibility with C, C++ retains the type system of C and adds classes to it.
- Therefore, C++ has both traditional imperative-language types and the class structure of an object-oriented language.
- This makes it a hybrid language, supporting both procedural programming and object- oriented programming.

- The objects of C++ can be static, stack dynamic, or heap dynamic.
- Explicit deallocation using the **delete** operator is required for heap-dynamic objects, because C++ does not include implicit storage reclamation.
- Many class definitions include a destructor method, which is implicitly called when an object of the class ceases to exist.
- The **destructor** is used to deallocate heap-allocated memory that is referenced by data members.
- It may also be used to record part or all of the state of the object just before it dies, usually for debugging purposes.

Inheritance

- A C++ class can be derived from an existing class, which is then its parent, or base class.
- The data defined in a class definition are called *data members* of that class, and the functions defined in a class definition are called *member functions* of that class (member functions in other languages are often called methods).
- Some or all of the members of the base class may be inherited by the derived class, which can also add new members and modify inherited member functions.

- All C++ objects must be initialized before they are used. Therefore, all C++ classes include **at least one constructor** method that initializes the data members of the new object.
- Constructor methods are implicitly called when an object is created. If any of the data members are pointers to heap-allocated data, the constructor allocates that storage.
- If a class has a parent, the inherited data members must be initialized when the subclass object is created. To do this, the parent constructor is implicitly called.

- Class members can be private, protected, or public.
- **Private members** are accessible only by member functions and friends of the class.
- **Public members** are visible everywhere.
- **Protected members** are like private members, except in derived classes, whose access is described next. Derived classes can modify accessibility for their inherited members.
- The syntactic form of a derived class is

class derived_class_name : derivation_mode base_class_name

{data member and member function declarations};

The derivation_mode can be either **public** or **private**.

- The public and protected members of a base class are also public and protected, respectively, in a **public-derived class**.
- In a **private-derived class**, both the public and protected members of the base class are private. So, in a class hierarchy, a private-derived class cuts off access to all members of all ancestor classes to all successor classes, and protected members may or may not be accessible to subsequent subclasses (past the first).
- Private members of a base class are inherited by a derived class, but they are not visible to the members of that derived class and are therefore of no use there.
- Consider the following example:

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class {...};
class subclass_2 : private base_class {...};
```

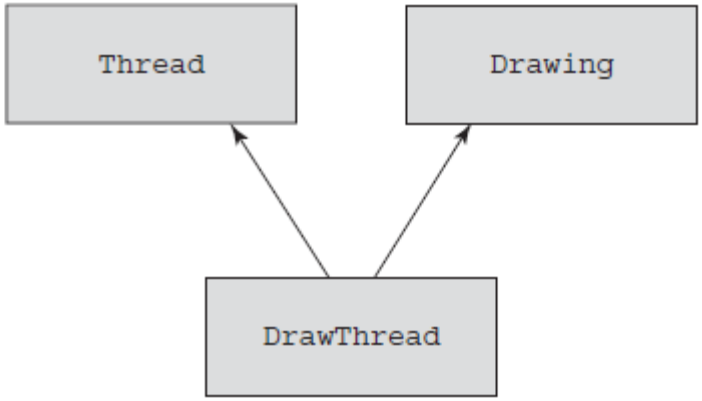
- In subclass_1, b and y are protected, and c and z are public.
- In subclass_2, b, y, c, and z are private.
- No derived class of subclass_2 can have members with access to any member of base_class.
- The data members a and x in base_class are not accessible in either subclass_1 or subclass_2.

- C++ provides multiple inheritance, which allows more than one class to be named as the parent of a new class.
- For example,

```
class Thread { . . . };
class Drawing { . . . };
class DrawThread : public Thread, public Drawing { . . . };
```

Figure 12.5

Multiple Inheritance



Dynamic Binding

- A C++ object could be manipulated through a value variable, rather than a pointer or a reference. (Such an object would be static or stack dynamic.)
- However, in that case, the object's type is known and static, so dynamic binding is not needed.
- C++ does not allow value variables (as opposed to pointers or references) to be polymorphic.
- When a polymorphic variable is used to call a member function overridden in one of the derived classes, the call must be dynamically bound to the correct member function definition. Member functions that must be dynamically bound must be declared to be virtual functions by preceding their headers with the reserved word **virtual**, which can appear only in a class body.

Consider the situation of having a base class named Shape, along with a collection of derived classes for different kinds of shapes, such as circles, rectangles, and so forth.

If these shapes need to be displayed, then the displaying member function, draw, must be unique for each descendant, or kind of shape.

These versions of draw must be defined to be virtual. When a call to draw is made with a pointer to the base class of the derived classes, that call must be dynamically bound to the member function of the correct derived class.

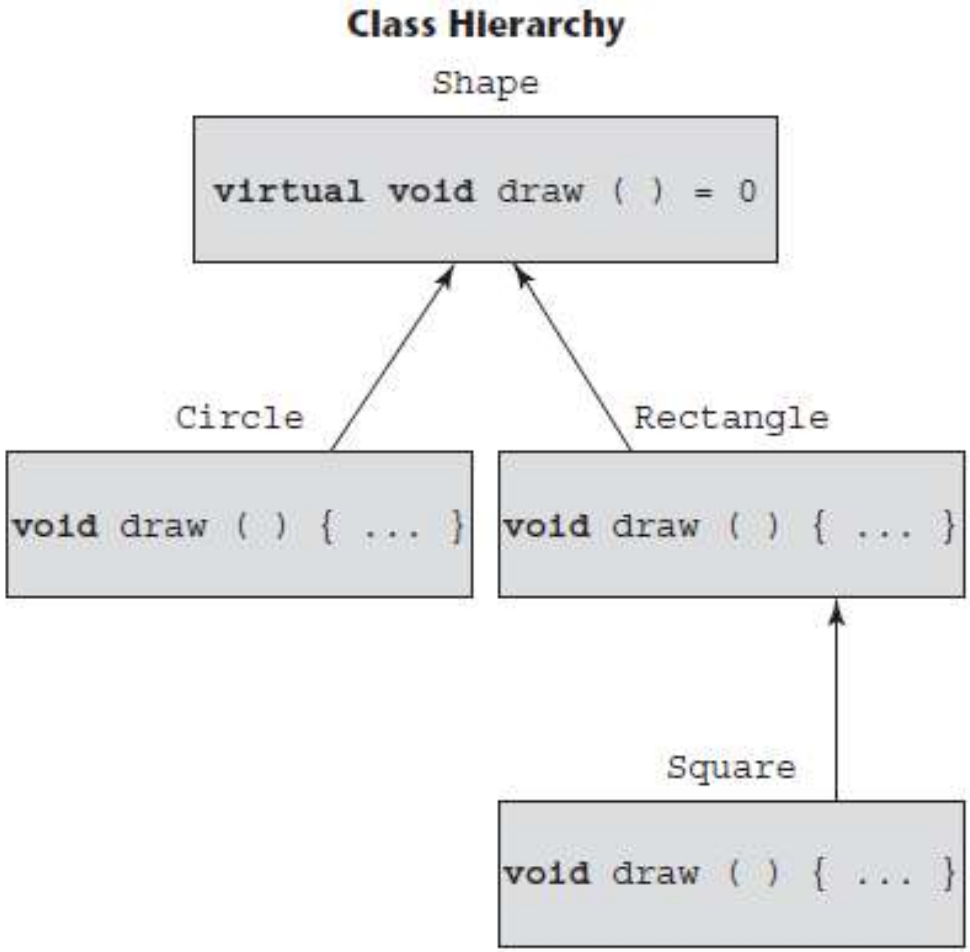
```
class Shape {
  public:
    virtual void draw() = 0;
  ...
};
class Circle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Rectangle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Square : public Rectangle {
  public:
    void draw() { ... }
  ...
};
```

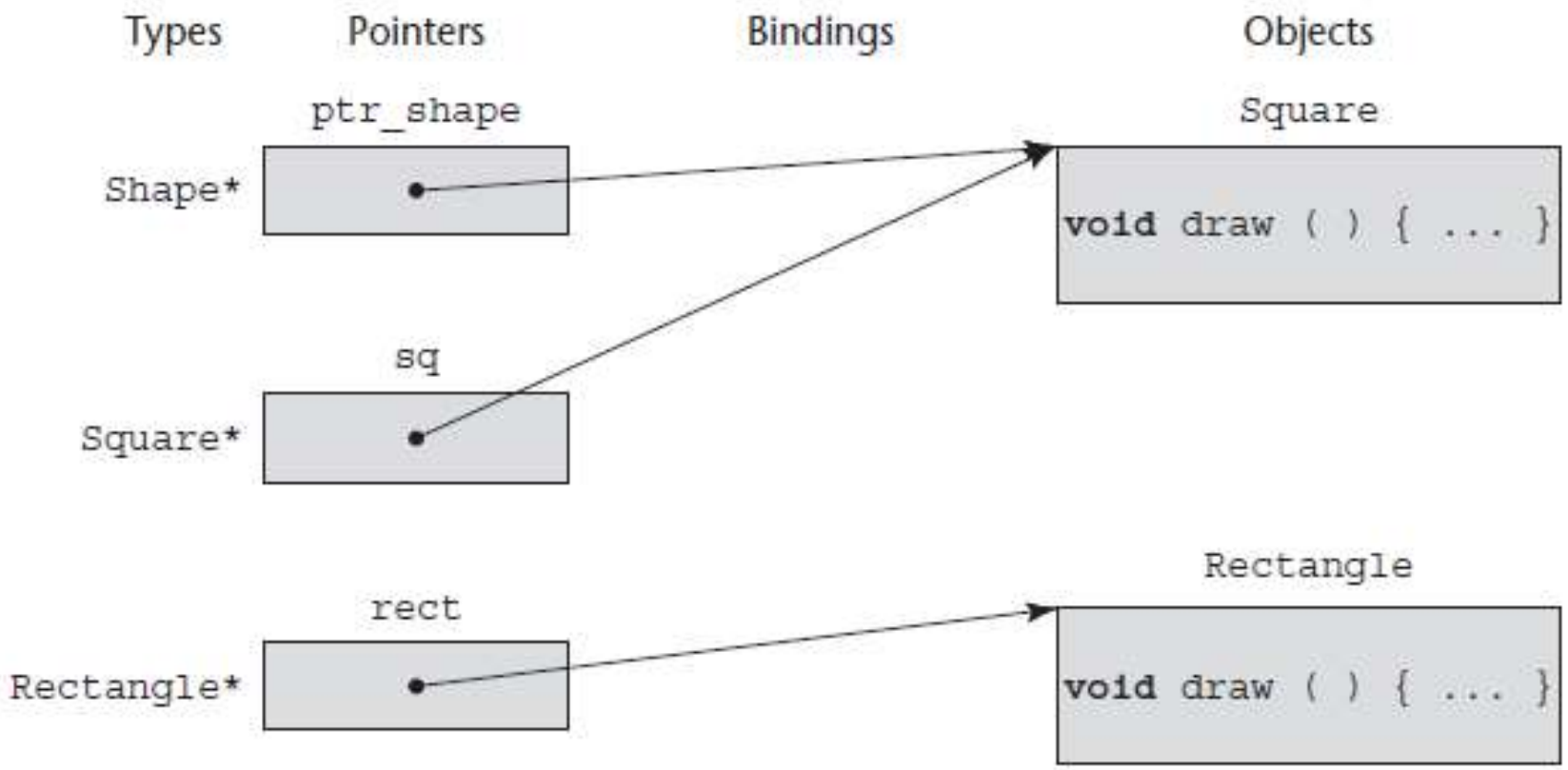
- Given these definitions, the following code has examples of both statically and dynamically bound calls:

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
ptr_shape = sq;          // Now ptr_shape points to a
                         //  Square object
ptr_shape->draw();       // Dynamically bound to the draw
                         //  in the Square class
rect->draw();            // Statically bound to the draw
                         //  in the Rectangle class
```

Figure 12.6

Dynamic binding




Types
Pointers
Bindings
Objects
ptr_shape
Shape*
Square
void draw ( ) { ... }
sq
Square*
Rectangle
rect
Rectangle*
void draw ( ) { ... }

Evaluation

- compare the object-oriented features of C++ with those of Smalltalk.
- The inheritance of C++ is more intricate than that of Smalltalk in terms of access control.
  - By using both the access controls within the class definition and the derivation access controls, and also the possibility of friend functions and classes, the C++ programmer has highly detailed control over the access to class members.
- C++ provides multiple inheritance and Smalltalk does not.
- In C++, the programmer can specify whether static binding or dynamic binding is to be used.
- In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound.
- Because static binding is faster, this is an advantage for those situations where dynamic binding is not necessary.

- The dynamic binding in C++ is fast when compared with that of Smalltalk.
- The static type checking of C++ is an advantage over Smalltalk, where all type checking is dynamic (flexible, but somewhat unsafe).
  - Compiler-detected errors are less expensive to repair than those found in testing.

Implementation of Object-Oriented Constructs

- There are at least two parts of language support for object-oriented programming that pose interesting questions for language implementers: storage structures for instance variables and the 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.

Dynamic Binding of Method Calls to Methods

- Methods in a class that are statically bound need not be involved in the CIR for the class.
- However, methods that will be dynamically bound must have entries in this structure. Such entries could simply have a pointer to the code of the method, which must be set at object creation time.
- Calls to a method could then be connected to the corresponding code through this pointer in the CIR.
- The drawback to this technique is that every instance would need to store pointers to all dynamically bound methods that could be called from the instance.

- The list of dynamically bound methods that can be called from an instance of a class is the same for all instances of that class.
- Therefore, the list of such methods must be stored only once. So the CIR for an instance needs only a single pointer to that list to enable it to find called methods.
- The storage structure for the list is often called a **virtual method table (vtable).**
- Method calls can be represented as offsets from the beginning of the vtable.
- Virtual Method Tables (VMTs) are used for dynamic binding

- 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() { ... }
}
```

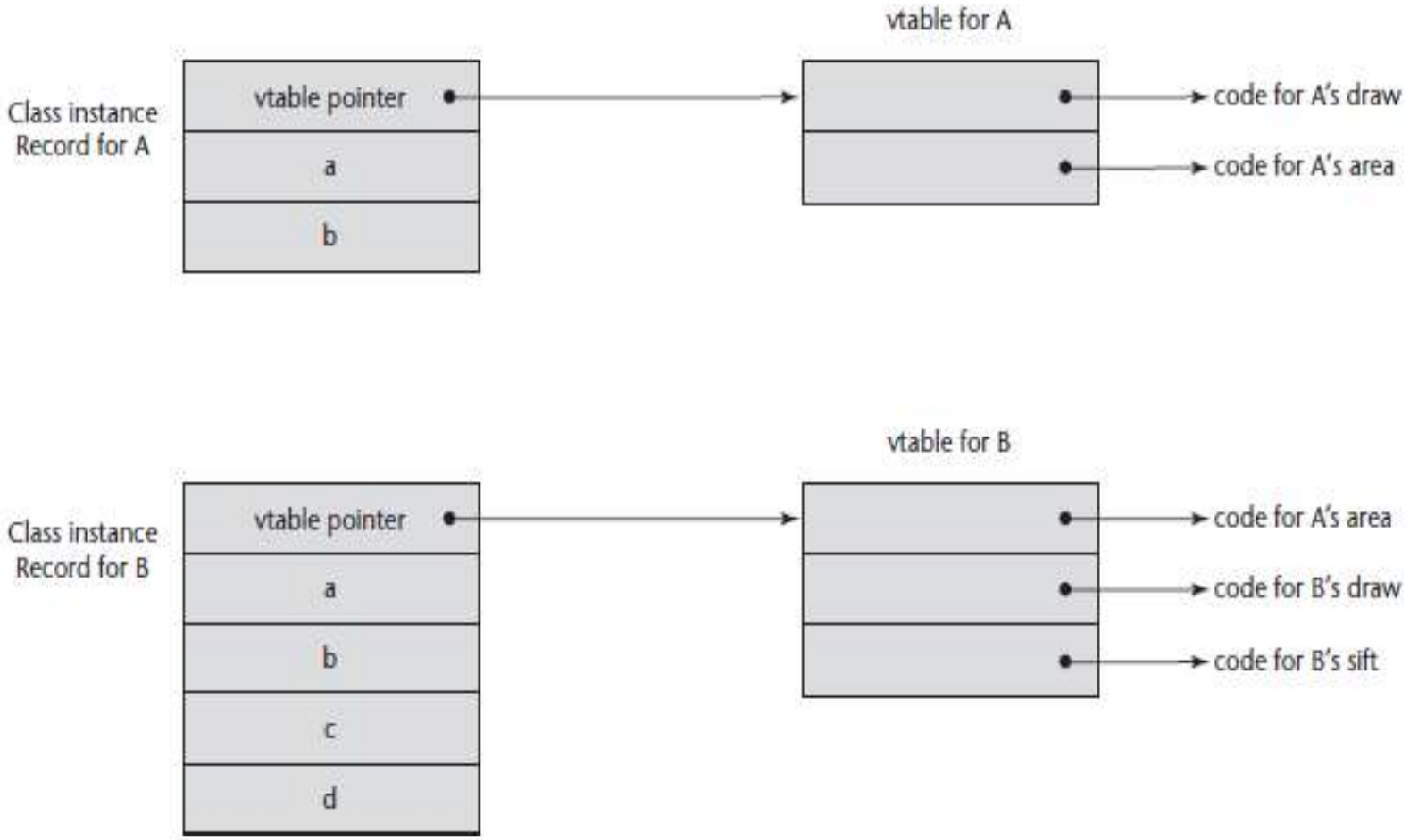


Figure 12.7

An example of the CIRs with single inheritance