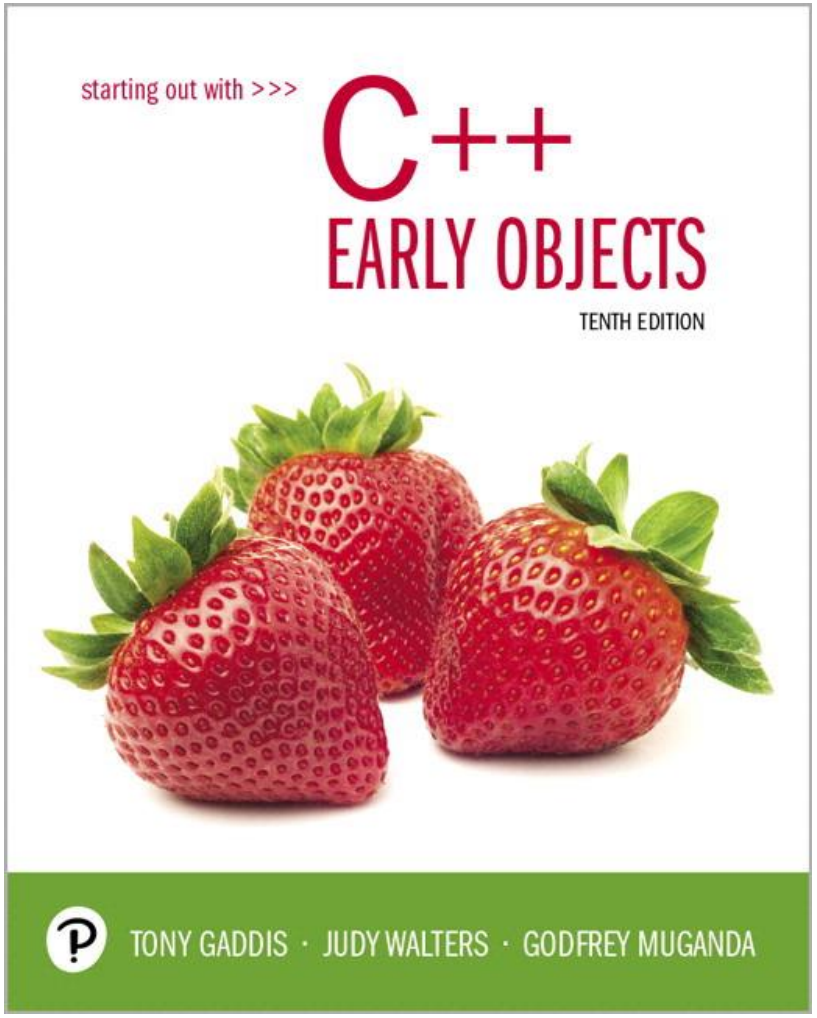
Starting Out with C++ Early Objects

Tenth Edition



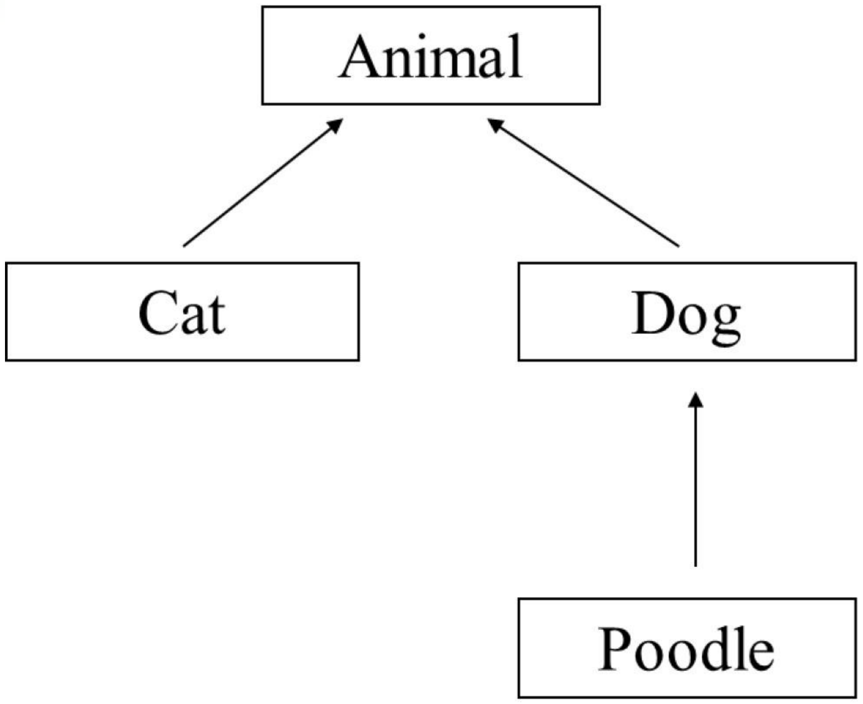
Chapter 15

Polymorphism and Virtual Functions

Topics

15.1 Type Compatibility in Inheritance Hierarchies

- Classes in a program may be part of an inheritance hierarchy
- Classes lower in the hierarchy are special cases of those above



Type Compatibility in Inheritance 1 of 2

- A pointer to a derived class can be assigned to a pointer to a base class.
- Another way to say this is:

  A base class pointer can point to

  derived class objects

```
Animal *pA = new Cat;
```

Type Compatibility in Inheritance 2 of 2

- Assigning a base class pointer to a derived class pointer requires a cast

```
Animal *pA = new Cat;
Cat *pC;
pC = static_cast<Cat *>(pA);
```

- The base class pointer must already point to a derived class object for this to work correctly.

Using Type Casts with Base Class Pointers

- C++ uses the declared type of a pointer to determine access to the members of the pointed-to object
- If an object of a derived class is pointed to by a base class pointer, all members of the derived class may not be accessible
- Type cast the base class pointer to the derived class (via **`static_pointer_cast`**) in order to access members that are specific to the derived class

Pointers, Inheritance, and Overridden Functions

If

- a derived class overrides a member function in the base class, and
- a base class pointer points to a derived class object,

then

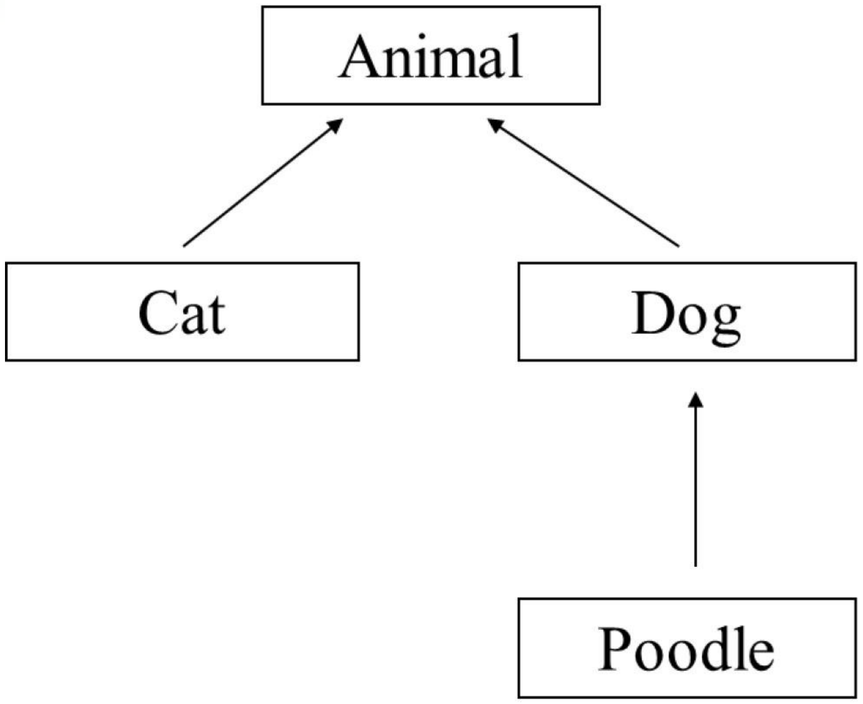
the compiler determines the version of the function to use by the type of the pointer, not by the type of the object.

15.2 Polymorphism and Virtual Member Functions

- Polymorphic code: Code that behaves differently when it acts on objects of different types
- Virtual Member Function: The C++ mechanism for achieving polymorphism

Polymorphism 1 of 5

Consider the Animal, Cat, Dog hierarchy where each class has its own version of the member function `id( )`



Polymorphism 2 of 5

```
class Animal{
 public: void id(){cout << "animal";}
};
class Cat : public Animal{
 public: void id(){cout << "cat";}
};
class Dog : public Animal{
 public: void id(){cout << "dog";}
};
```

Polymorphism 3 of 5

- Consider a collection of different Animal objects

```
Animal *pA[] = {new Animal, new Dog,
                new Cat};
```

and accompanying code

```
for(int k=0; k<3; k++)
    pA[k]->id();
```

- Prints: **`animal animal animal`**, ignoring the more specific versions of **`id()`** in **`Dog`** and **`Cat`**

Polymorphism 4 of 5

- The preceding code is not polymorphic: it behaves the same way even though **`Animal`**, **`Dog`** and **`Cat`** have different types and different **`id()`** member functions
- Polymorphic code would have printed "**`animal dog cat`**" instead of "**`animal animal animal`**"

Polymorphism 5 of 5

- The code is not polymorphic because in the expression

```
pA[k]->id()
```

  the compiler sees only the type of the pointer `pA[k]`, which is pointer to `Animal`

- The compiler does not see type of actual object pointed to, which may be `Animal`, or `Dog`, or `Cat`

Virtual Functions 1 of 3

Declaring a function `virtual` will make the compiler check the type of each object to see if it defines a more specific version of the virtual function

Virtual Functions 2 of 3

If the member functions `id()` are declared virtual, then the code

```
Animal *pA[] = {new Animal,
                new Dog,new Cat};
for(int k=0; k<3; k++)
    pA[k]->id();
```

will print **`animal dog cat`**

Virtual Functions 3 of 3

How to declare a member function virtual:

```
class Animal{
 public: virtual void id(){cout << "animal";}

};

class Cat : public Animal{
 public: virtual void id(){cout << "cat";}

};

class Dog : public Animal{
 public: virtual void id(){cout << "dog";}

};
```

Function Binding

- In **`pA[k]->id()`** , compiler must choose which version of **`id()`** to use. There are different versions in the **`Animal`**, **`Dog`**, and **`Cat`** classes
- Function binding is the process of determining which function definition to use for a particular function call
- The alternatives are *static* and *dynamic* binding

Static Binding

- Static binding chooses the function in the class of the base class pointer, ignoring any versions in the class of the object actually pointed to
- Static binding is performed at compile time

Dynamic Binding

- With Dynamic Binding, the function to be invoked is determined at execution time
- Can look at the actual class of the object pointed to and choose the most specific version of the function
- Dynamic binding is used to bind virtual functions

Overriding vs. Overloading

- Recall that overloaded functions have the same name but different parameter lists.
- Suppose a derived class has a function that overloads a virtual member function in the base class. If you want the derived class function to override the base class function, use the **`override`** key word at the end of the function header or prototype. This indicates that it should be treated as an overriding function instead of as an overloading function.

The `final` Key Word and Overriding

- The key word final can be used at the end of the header or prototype of a function in an inheritance hierarchy if you want to ensure that no classes below this one override this function.
- If an attempt is made to override this function in a derived class, a compiler error will occur.

15.3 Abstract Base Classes and Pure Virtual Functions 1 of 2

- An abstract class is a class that contains no objects that are not members of subclasses (derived classes)
- For example, in real life, Animal is an abstract class: there are no animals that are not dogs, or cats, or lions…

Abstract Base Classes and Pure Virtual Functions 2 of 2

- Abstract classes are an organizational tool. They are useful in organizing inheritance hierarchies
- Abstract classes can be used to specify an interface that must be implemented by all subclasses

Pure Virtual Functions 1 of 2

- The member functions specified in an abstract class do not have to be implemented
- The implementation is left to the subclasses
- A function without an implementation is a pure virtual function or an abstract function
- In C++, an abstract class is a class with at least one abstract member function

Pure Virtual Functions 2 of 2

- In C++, a member function of a class is declared to be an abstract function by making it virtual and replacing its body with `= 0;`

```
class Animal
{
  public:
    virtual void id()=0;
};
```

Abstract Classes

- An abstract class can not be instantiated
- An abstract class can only be inherited from; that is, you can derive classes from it
- Classes derived from an abstract class must override all pure virtual functions with concrete member functions in order to be instantiated.

15.4 Composition vs. Inheritance 1 of 2

- Inheritance models an 'is a' relation between classes. An object of a derived class 'is a(n)' object of the base class
- Example:
  - an **`UnderGrad`** is a **`Student`**
  - a **`Mammal`** is an **`Animal`**
  - a **`Poodle`** is a **`Dog`**

Composition vs. Inheritance 2 of 2

- When defining a new class:
- <u>Composition</u> is appropriate when the new class needs to use an object of an existing class
- <u>Inheritance</u> is appropriate when
  - objects of the new class are a subset of the objects of the existing class, or
  - objects of the new class will be used in the same ways as the objects of the existing class

Copyright