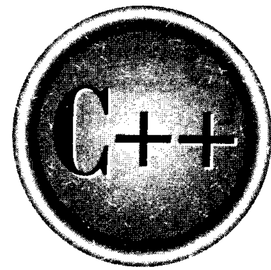
The Complete Reference



Chapter 17

Virtual Functions and Polymorphism

Polymorphism is supported by C++ both at compile time and at run time. As discussed in earlier chapters, compile-time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions, and these are the topics of this chapter.

Virtual Functions

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method.*

When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer. As discussed in Chapter 13, a base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time.* Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.

To begin, examine this short example:

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
```

```
};

class derived2 : public base {
public:
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()

  return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

As the program illustrates, inside **base**, the virtual function **vfunc( )** is declared. Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc( )** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)

In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc( )** is redefined relative to that class. Inside **main( )**, four variables are declared:

Name	Type
p	base class pointer
b	object of base
d1	object of derived1
d2	object of derived2

Next, **p** is assigned the address of **b**, and **vfunc( )** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc( )** is executed. Next, **p** is set to the address of **d1**, and again **vfunc( )** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc( )** to be executed. Finally, **p** is assigned the address of **d2**, and **p–>vfunc( )** causes the version of **vfunc( )** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc( )** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.

Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved. For example, assuming the preceding example, this is syntactically valid:

```
d2.vfunc(); // calls derived2's vfunc()
```

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc( )**.

At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term *overloading* is not applied to virtual function redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ. (In fact, when you overload a function, either the number or the type of the parameters *must* differ! It is through these differences that C++ can select the correct version of an overloaded function.) However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost. Another important restriction is that virtual functions must be

nonstatic members of the classes of which they are part. They cannot be **friends**. Finally, constructor functions cannot be virtual, but destructor functions can.

Because of the restrictions and differences between function overloading and virtual function redefinition, the term *overriding* is used to describe virtual function redefinition by a derived class.

Calling a Virtual Function Through a Base Class Reference

In the preceding example, a virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference. As explained in Chapter 13, a reference is an implicit pointer. Thus, a base-class reference can be used to refer to an object of the base class or any object derived from that base. When a virtual function is called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call.

The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter. For example, consider the following variation on the preceding program.

```
/* Here, a base class reference is used to access
   a virtual function. */
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
public:
```

```
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};

// Use a base class reference parameter.
void f(base &r) {
  r.vfunc();
}

int main()
{
  base b;
  derived1 d1;
  derived2 d2;

  f(b); // pass a base object to f()
  f(d1); // pass a derived1 object to f()
  f(d2); // pass a derived2 object to f()

  return 0;
}
```

This program produces the same output as its preceding version. In this example, the function **f( )** defines a reference parameter of type **base**. Inside **main( )**, the function is called using objects of type **base**, **derived1**, and **derived2**. Inside **f( )**, the specific version of **vfunc( )** that is called is determined by the type of object being referenced when the function is called.

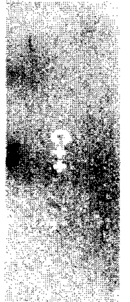
For the sake of simplicity, the rest of the examples in this chapter will call virtual functions through base-class pointers, but the effects are same for base-class references.

The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden. Put differently, no matter how many times a virtual function is inherited, it remains virtual. For example, consider this program:

```
#include <iostream>
using namespace std;
```

```
class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

/* derived2 inherits virtual function vfunc()
   from derived1. */
class derived2 : public derived1 {
public:
  // vfunc() is still virtual
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()
```



```
  return 0;
}
```

As expected, the preceding program displays this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

In this case, **derived2** inherits **derived1** rather than **base**, but **vfunc( )** is still virtual.

Virtual Functions Are Hierarchical

As explained, when a function is declared as **virtual** by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used. For example, consider this program in which **derived2** does not override **vfunc( )**:

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
```

```
public:
// vfunc() not overridden by derived2, base's is used
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // use base's vfunc()

  return 0;
}
```

The program produces this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

Because **derived2** does not override **vfunc( )**, the function defined by **base** is used when **vfunc( )** is referenced relative to objects of type **derived2**.

The preceding program illustrates a special case of a more general rule. Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical. This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used. For example, in the following program, **derived2** is derived from **derived1**, which is derived from **base**. However, **derived2** does not override **vfunc( )**. This means that, relative to **derived2**,

the closest version of **vfunc( )** is in **derived1**. Therefore, it is **derived1::vfunc( )** that is used when an object of **derived2** attempts to call **vfunc( )**.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public derived1 {
public:
/* vfunc() not overridden by derived2.
   In this case, since derived2 is derived from
   derived1, derived1's vfunc() is used.
*/
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()
```

```
  // point to derived2
  p = &d2;
  p->vfunc(); // use derived1's vfunc()

  return 0;
}
```

The program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived1's vfunc().
```

Pure Virtual Functions

As the examples in the preceding section illustrate, when a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

A *pure virtual function* is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

virtual *type func-name(parameter-list)* = 0;

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

The following program contains a simple example of a pure virtual function. The base class, **number**, contains an integer called **val**, the function **setval( )**, and the pure virtual function **show( )**. The derived classes **hextype**, **dectype**, and **octtype** inherit **number** and redefine **show( )** so that it outputs the value of **val** in each respective number base (that is, hexadecimal, decimal, or octal).

```
#include <iostream>
using namespace std;

class number {
```

```
protected:
  int val;
public:
  void setval(int i) { val = i; }

  // show() is a pure virtual function
  virtual void show() = 0;
};

class hextype : public number {
public:
  void show() {
    cout << hex << val << "\n";
  }
};

class dectype : public number {
public:
  void show() {
    cout << val << "\n";
  }
};

class octtype : public number {
public:
  void show() {
    cout << oct << val << "\n";
  }
};

int main()
{
  dectype d;
  hextype h;
  octtype o;

  d.setval(20);
  d.show(); // displays 20 - decimal

  h.setval(20);
  h.show(); // displays 14 - hexadecimal
```

```
    o.setval(20);
    o.show();  // displays 24 - octal

    return 0;
}
```

Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, **number** simply provides the common interface for the derived types to use. There is no reason to define **show( )** inside **number** since the base of the number is undefined. Of course, you can always create a placeholder definition of a virtual function. However, making **show( )** pure also ensures that all derived classes will indeed redefine it to meet their own needs.

Keep in mind that when a virtual function is declared as pure, all derived classes must override it. If a derived class fails to do this, a compile-time error will result.

Abstract Classes

A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes.

Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

Using Virtual Functions

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations. In concrete C++ terms, a base class can be used to define the nature of the interface to a general class. Each derived class then implements the specific operations as they relate to the type of data used by the derived type.

One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism. Using these features, you create a class hierarchy that moves from general to specific (base to derived). Following this philosophy, you define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function. In essence, in the base

class you create and define everything you can that relates to the general case. The derived class fills in the specific details.

Following is a simple example that illustrates the value of the "one interface, multiple methods" philosophy. A class hierarchy is created that performs conversions from one system of units to another. (For example, liters to gallons.) The base class **convert** declares two variables, **val1** and **val2**, which hold the initial and converted values, respectively. It also defines the functions **getinit( )** and **getconv( )**, which return the initial value and the converted value. These elements of **convert** are fixed and applicable to all derived classes that will inherit **convert**. However, the function that will actually perform the conversion, **compute( )**, is a pure virtual function that must be defined by the classes derived from **convert**. The specific nature of **compute( )** will be determined by what type of conversion is taking place.

```
// Virtual function practical example.
#include <iostream>
using namespace std;

class convert {
protected:
  double val1;  // initial value
  double val2;  // converted value
public:
  convert(double i) {
    val1 = i;
  }
  double getconv() { return val2; }
  double getinit() { return val1; }

  virtual void compute() = 0;
};

// Liters to gallons.
class l_to_g : public convert {
public:
  l_to_g(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.7854;
  }
};

// Fahrenheit to Celsius
class f_to_c : public convert {
```

```
public:
  f_to_c(double i) : convert(i) { }
  void compute() {
    val2 = (val1-32) / 1.8;
  }
};

int main()
{
  convert *p;  // pointer to base class

  l_to_g lgob(4);
  f_to_c fcob(70);

  // use virtual function mechanism to convert
  p = &lgob;
  cout << p->getinit() << " liters is ";
  p->compute();
  cout << p->getconv() << " gallons\n";  // l_to_g

  p = &fcob;
  cout << p->getinit() << " in Fahrenheit is ";
  p->compute();
  cout << p->getconv() << " Celsius\n";  // f_to_c

  return 0;
}
```

The preceding program creates two derived classes from **convert**, called **l_to_g** and **f_to_c**. These classes perform the conversions of liters to gallons and Fahrenheit to Celsius, respectively. Each derived class overrides **compute( )** in its own way to perform the desired conversion. However, even though the actual conversion (that is, method) differs between **l_to_g** and **f_to_c**, the interface remains constant.

One of the benefits of derived classes and virtual functions is that handling a new case is a very easy matter. For example, assuming the preceding program, you can add a conversion from feet to meters by including this class:

```
// Feet to meters
class f_to_m : public convert {
public:
  f_to_m(double i) : convert(i) { }
```

```
    void compute() {
      val2 = val1 / 3.28;
    }
};
```

An important use of abstract classes and virtual functions is in *class libraries*. You can create a generic, extensible class library that will be used by other programmers. Another programmer will inherit your general class, which defines the interface and all elements common to all classes derived from it, and will add those functions specific to the derived class. By creating class libraries, you are able to create and control the interface of a general class while still letting other programmers adapt it to their specific needs.

One final point: The base class **convert** is an example of an abstract class. The virtual function **compute( )** is not defined within **convert** because no meaningful definition can be provided. The class **convert** simply does not contain sufficient information for **compute( )** to be defined. It is only when **convert** is inherited by a derived class that a complete type is created.

Early vs. Late Binding

Before concluding this chapter on virtual functions and run-time polymorphism, there are two terms that need to be defined because they are used frequently in discussions of C++ and object-oriented programming: *early binding* and *late binding*.

Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

The opposite of early binding is *late binding*. As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.