Module 10

Inheritance, Virtual Functions, and Polymorphism

Table of Contents

This module discusses three features of C++ that directly relate to object-oriented programming: inheritance, virtual functions, and polymorphism. Inheritance is the feature that allows one class to inherit the characteristics of another. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. Built on the foundation of inheritance is the virtual function. The virtual function supports polymorphism, the "one interface, multiple methods" philosophy of object-oriented programming.

CRITICAL SKILL 10.1: Inheritance Fundamentals

In the language of C++, a class that is inherited is called a base class. The class that does the inheriting is called a derived class. Therefore, a derived class is a specialized version of a base class. A derived class inherits all of the members defined by the base class and adds its own, unique elements.

C++ implements inheritance by allowing one class to incorporate another class into its declaration. This is done by specifying a base class when a derived class is declared. Let's begin with a short example that illustrates several of the key features of inheritance. The following program creates a base class called TwoDShape that stores the width and height of a two-dimensional object, and a derived class called Triangle. Pay close attention to the way that Triangle is declared.

```cpp
// A simple class hierarchy.

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

// A class for two-dimensional objects.
class TwoDShape {
public:
  double width;
  double height;

  void showDim() {
    cout << "Width and height are " <<
            width << " and " << height << "\n";
  }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {          ←————— Triangle inherits TwoDShape.
public:                                             Notice the syntax.
  char style[20];

  double area() {
    return width * height / 2;               ←———— Triangle can refer to the
  }                                                 members of TwoDShape as
                                                    if they were part of Triangle.
  void showStyle() {
    cout << "Triangle is " << style << "\n";
  }
};
```

```
int main() {
   Triangle t1;
   Triangle t2;

   t1.width = 4.0;                    All members of Triangle are
   t1.height = 4.0;                   available to Triangle objects, even
   strcpy(t1.style, "isosceles");     those inherited from TwoDShape.

   t2.width = 8.0;
   t2.height = 12.0;
   strcpy(t2.style, "right");

   cout << "Info for t1:\n";
   t1.showStyle();
   t1.showDim();
   cout << "Area is " << t1.area() << "\n";

   cout << "\n";
   cout << "Info for t2:\n";
   t2.showStyle();
   t2.showDim();
   cout << "Area is " << t2.area() << "\n";

   return 0;
}
```

The output from this program is shown here:

```
Info for t1:
Triangle is isosceles
Width and height are 4 and 4
Area is 8

Info for t2:
Triangle is right
Width and height are 8 and 12
Area is 48
```

Here, TwoDShape defines the attributes of a "generic" two-dimensional shape, such as a square, rectangle, triangle, and so on. The Triangle class creates a specific type of TwoDShape,inthis case, a triangle. The Triangle class includes all of TwoDShape and adds the field style,the function area( ), and the function showStyle( ). A description of the type of triangle is stored in style, area( ) computes and returns the area of the triangle, and showStyle( ) displays the triangle style.

The following line shows how Triangle inherits TwoDShape:

class Triangle : public TwoDShape {

Here, TwoDShape is a base class that is inherited by Triangle, which is a derived class. As this example shows, the syntax for inheriting a class is remarkably simple and easy-to-use.

C++ A Beginner's Guide by Herbert Schildt

Because Triangle includes all of the members of its base class, TwoDShape, it can access width and height inside area( ). Also, inside main( ), objects t1 and t2 can refer to width and height directly, as if they were part of Triangle. Figure 10-1 depicts conceptually how TwoDShape is incorporated into Triangle.

One other point: Even though TwoDShape is a base for Triangle, it is also a completely independent, stand-alone class. Being a base class for a derived class does not mean that the base class cannot be used by itself.

The general form for inheritance is shown here:

class derived-class : access base-class { // body of derived class }

Here, access is optional. However, if present, it must be public, private, or protected. You will learn more about these options later in this module. For now, all inherited classes will use public. Using public means that all the public members of the base class will also be public members of the derived class.
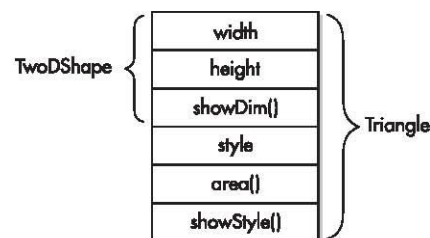
Figure 10-1 A conceptual depiction of the **Triangle** class

A major advantage of inheritance is that once you have created a base class that defines the attributes common to a set of objects, it can be used to create any number of more specific derived classes. Each derived class can precisely tailor its own classification. For example, here is another class derived from TwoDShape that encapsulates rectangles:

```cpp
// A derived class of TwoDShape for rectangles.
class Rectangle : public TwoDShape {
public:
  bool isSquare() {
    if(width == height) return true;
    return false;
  }

  double area() {
    return width * height;
  }
};
```

The Rectangle class includes TwoDShape and adds the functions isSquare( ), which determines if the rectangle is square, and area( ), which computes the area of a rectangle.

Member Access and Inheritance

As you learned in Module 8, members of a class are often declared as private to prevent their unauthorized use or tampering. Inheriting a class does not overrule the private access restriction. Thus, even though a derived class includes all of the members of its base class, it cannot access those members of the base class that are private. For example, if width and height are made private in TwoDShape, as shown here, then Triangle will not be able to access them.

```
// Access to private members is not granted to derived classes.

class TwoDShape {
  // these are now private
  double width;          ←——————————— width and height are now private.
  double height;
public:
  void showDim() {
    cout << "Width and height are " <<
            width << " and " << height << "\n";
  }
};
// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
public:
  char style[20];

  double area() {
    return width * height / 2; // Error! Can't access. ←——————
  }                                                    Can't access private members
                                                       of a base class.
  void showStyle() {
    cout << "Triangle is " << style << "\n";
  }
};
```

The Triangle class will not compile because the reference to width and height inside the area( ) function causes an access violation. Since width and height are now private, they are accessible only by other members of their own class. Derived classes have no access to them.

At first, you might think that it is a serious restriction that derived classes do not have access to the private members of base classes, because it would prevent the use of private members in many situations. Fortunately, this is not the case, because C++ provides various solutions. One is to use protected members, which is described in the next section. A second is to use public functions to provide access to private data. As you have seen in the preceding modules, C++ programmers typically grant access to the private members of a class through functions. Functions that provide access to private data are called accessor functions. Here is a rewrite of the TwoDShape class that adds accessor functions for width and height:

```cpp
// Access private data through accessor functions.

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

// A class for two-dimensional objects.
class TwoDShape {
  // these are private
  double width;
  double height;
public:

  void showDim() {
    cout << "Width and height are " <<
            width << " and " << height << "\n";
  }
  // accessor functions
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
};
```

The accessor functions for **width** and **height**

```cpp
// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
public:
  char style[20];

  double area() {
    return getWidth() * getHeight() / 2;
  }
```

Use the accessor functions to obtain the width and height.

```cpp
  void showStyle() {
    cout << "Triangle is " << style << "\n";
  }
};
int main() {
  Triangle t1;
  Triangle t2;

  t1.setWidth(4.0);
  t1.setHeight(4.0);
  strcpy(t1.style, "isosceles");

  t2.setWidth(8.0);
  t2.setHeight(12.0);
  strcpy(t2.style, "right");
```

```
   cout << "Info for t1:\n";
   t1.showStyle();
   t1.showDim();
   cout << "Area is " << t1.area() << "\n";

   cout << "\n";
   cout << "Info for t2:\n";
   t2.showStyle();
   t2.showDim();
   cout << "Area is " << t2.area() << "\n";

   return 0;
}
```

Progress Check

1. How is a base class inherited by a derived class?
2. Does a derived class include the members of its base class?
3. Does a derived class have access to the private members of its base class?

CRITICAL SKILL 10.2: Base Class Access Control

As explained, when one class inherits another, the members of the base class become members of the derived class. However, the accessibility of the base class members inside the derived class is determined by the access specifier used when inheriting the base class. The base class access specifier must be public, private, or protected. If the access specifier is not used, then it is private by default if the derived class is a class. If the derived class is a struct, then public is the default. Let's examine the ramifications of using public or private access. (The protected specifier is described in the next section.)

Ask the Expert

Q: I have heard the terms superclass and subclass used in discussions of Java programming. Do these terms have meaning in C++?

A: What Java calls a superclass, C++ calls a base class. What Java calls a subclass, C++ calls a derived class. You will commonly hear both sets of terms applied to a class of either language, but this book will

continue to use the standard C++ terms. By the way, C# also uses the base class, derived class terminology.

When a base class is inherited as public, all public members of the base class become public members of the derived class. In all cases, the private elements of the base class remain private to that class and are not accessible by members of the derived class. For example, in the following program, the public members of B become public members of D. Thus, they are accessible by other parts of the program.

```cpp
// Demonstrate public inheritance.

#include <iostream>
using namespace std;

class B {
  int i, j;
public:
  void set(int a, int b) { i = a; j = b; }
  void show() { cout << i << " " << j << "\n"; }
};

class D : public B {          Here, B is inherited as public.
  int k;
public:
  D(int x) { k = x; }
  void showk() { cout << k << "\n"; }

  // i = 10; // Error! i is private to B and access is not allowed.
};
                                        Can't access i because it is private to B.
int main()
{
  D ob(3);

  ob.set(1, 2); // access member of base class
  ob.show();    // access member of base class

  ob.showk();   // uses member of derived class

  return 0;
}
```

Since set( ) and show( ) are public in B, they can be called on an object of type D from within main( ). Because i and j are specified as private, they remain private to B. This is why the line

// i = 10; // Error! i is private to B and access is not allowed.

is commented-out. D cannot access a private member of B.

The opposite of public inheritance is private inheritance. When the base class is inherited as private, then all public members of the base class become private members of the derived class. For example,

the program shown next will not compile, because both set( ) and show( ) are now private members of D, and thus cannot be called from main( ).

```cpp
// Use private inheritance. This program won't compile.

#include <iostream>
using namespace std;

class B {
  int i, j;
public:
  void set(int a, int b) { i = a; j = b; }
  void show() { cout << i << " " << j << "\n"; }
};

// Public elements of B become private in D.
class D : private B {      // Now, inherit B as private.
  int k;
public:
  D(int x) { k = x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  D ob(3);

  ob.set(1, 2); // Error, can't access set()    // Now, set( ) and show( ) are inaccessible through D.
  ob.show();    // Error, can't access show()

  return 0;
}
```
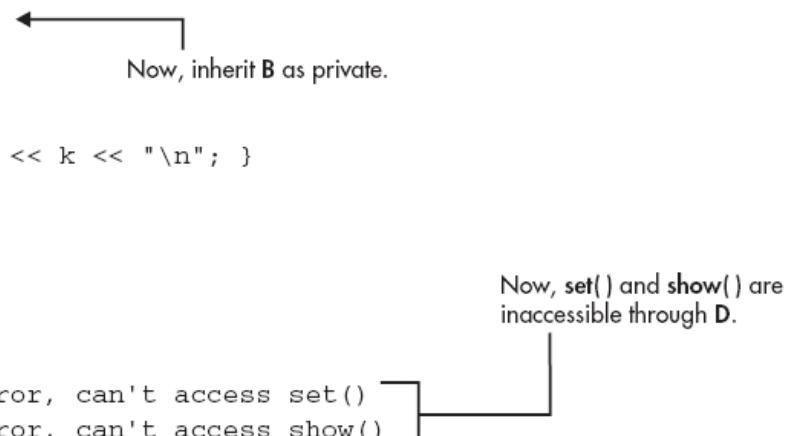
To review: when a base class is inherited as private, public members of the base class become private members of the derived class. This means that they are still accessible by members of the derived class, but cannot be accessed by other parts of your program.

CRITICAL SKILL 10.3: Using protected Members

As you know, a private member of a base class is not accessible by a derived class. This would seem to imply that if you wanted a derived class to have access to some member in the base class, it would need to be public. Of course, making the member public also makes it available to all other code, which may not be desirable. Fortunately, this implication is wrong because C++ allows you to create a protected member. A protected member is public within a class hierarchy, but private outside that hierarchy.

A protected member is created by using the protected access modifier. When a member of a class is declared as protected, that member is, with one important exception, private. The exception occurs

when a protected member is inherited. In this case, the protected member of the base class is accessible by the derived class. Therefore, by using protected, you can create class members that are private to their class but that can still be inherited and accessed by a derived class. The protected specifier can also be used with structures.

Consider this sample program:

```cpp
// Demonstrate protected members.

#include <iostream>
using namespace std;
                              Here, i and j are protected.
class B {
protected:
  int i, j; // private to B, but accessible to D
public:
  void set(int a, int b) { i = a; j = b; }
  void show() { cout << i << " " << j << "\n"; }
};

class D : public B {
  int k;
public:
  // D may access B's i and j         D can access i and j because
  void setk() { k = i*j; }            they are protected, not private.

  void showk() { cout << k << "\n"; }
};

int main()
{
  D ob;

  ob.set(2, 3); // OK, set() is public in B
  ob.show();    // OK, show() is public B

  ob.setk();
  ob.showk();

  return 0;
}
```

Here, because B is inherited by D as public and because i and j are declared as protected, D's function setk( ) can access them. If i and j were declared as private by B, then D would not have access to them, and the program would not compile.

When a base class is inherited as public, protected members of the base class become protected members of the derived class. When a base class is inherited as private, protected members of the base class become private members of the derived class.

The protected access specifier may occur anywhere in a class declaration, although typically it occurs after the (default) private members are declared and before the public members. Thus, the most common full form of a class declaration is

class class-name{

// private members by default protected:

// protected members public:

// public members };

Of course, the protected category is optional.

In addition to specifying protected status for members of a class, the keyword protected can also act as an access specifier when a base class is inherited. When a base class is inherited as protected, all public and protected members of the base class become protected members of the derived class. For example, in the preceding example, if D inherited B, as shown here:

class D : protected B {

then all non-private members of B would become protected members of D.

Progress Check

1. When a base class is inherited as private, public members of the base class become private members of the derived class. True or false?
2. Can a private member of a base class be made public through inheritance?
3. To make a member accessible within a hierarchy, but private otherwise, what access specifier do you use?

Ask the Expert

Q: Can you review public, protected, and private?

A: When a class member is declared as public, it can be accessed by any other part of a program.

When a member is declared as private, it can be accessed only by members of its class. Further, derived classes do not have access to private base class members. When a member is declared as protected, it can be accessed only by members of its class and by its derived classes. Thus, protected allows a member to be inherited, but to remain private within a class hierarchy.

When a base class is inherited by use of public, its public members become public members of the derived class, and its protected members become protected members of the derived class. When a base class is inherited by use of protected, its public and protected members become protected members of

the derived class. When a base class is inherited by use of private, its public and protected members become private members of the derived class. In all cases, private members of a base class remain private to that base class.

Constructors and Inheritance

In a hierarchy, it is possible for both base classes and derived classes to have their own constructors. This raises an important question: what constructor is responsible for building an object of the derived class, the one in the base class, the one in the derived class, or both? The answer is this: the constructor for the base class constructs the base class portion of the object, and the constructor for the derived class constructs the derived class part. This makes sense because the base class has no knowledge of or access to any element in a derived class. Thus, their construction must be separate. The preceding examples have relied upon the default constructors created automatically by C++, so this was not an issue. However, in practice, most classes will define constructors. Here you will see how to handle this situation.

When only the derived class defines a constructor, the process is straightforward: simply construct the derived class object. The base class portion of the object is constructed automatically using its default constructor. For example, here is a reworked version of Triangle that defines a constructor. It also makes style private since it is now set by the constructor.

```
// Add a constructor to Triangle.

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

// A class for two-dimensional objects.
class TwoDShape {
  // these are private
  double width;
  double height;
public:
  void showDim() {
    cout << "Width and height are " <<
            width << " and " << height << "\n";
  }

  // accessor functions
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
  char style[20]; // now private
public:

  // Constructor for Triangle.
  Triangle(char *str, double w, double h) {
   // Initialize the base class portion.
    setWidth(w);                                       Initialize the TwoDShape
    setHeight(h);                                      portion of Triangle.

    // Initialize the derived class portion.
    strcpy(style, str);    ←                           Initialize style, which
  }                                                    is specific to Triangle.

  double area() {
    return getWidth() * getHeight() / 2;
  }

  void showStyle() {
    cout << "Triangle is " << style << "\n";
  }
};
```

Here, Triangle's constructor initializes the members of TwoDShape that it inherits along with its own style field.

When both the base class and the derived class define constructors, the process is a bit more complicated, because both the base class and derived class constructors must be executed.

CRITICAL SKILL 10.4: Calling Base Class Constructors

When a base class has a constructor, the derived class must explicitly call it to initialize the base class portion of the object. A derived class can call a constructor defined by its base class by using an expanded form of the derived class' constructor declaration. The general form of this expanded declaration is shown here:

derived-constructor(arg-list) : base-cons(arg-list); {

body of derived constructor

}

Here, base-cons is the name of the base class inherited by the derived class. Notice that a colon separates the constructor declaration of the derived class from the base class constructor. (If a class inherits more than one base class, then the base class constructors are separated from each other by commas.)

The following program shows how to pass arguments to a base class constructor. It defines a constructor for TwoDShape that initializes the width and height properties.

```cpp
// Add a constructor to TwoDShape.

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

// A class for two-dimensional objects.
class TwoDShape {
  // these are private
  double width;
  double height;
public:

  // Constructor for TwoDShape.
  TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  void showDim() {
    cout << "Width and height are " <<
            width << " and " << height << "\n";
  }

  // accessor functions
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
  char style[20]; // now private
public:

  // Constructor for Triangle.
  Triangle(char *str, double w,
           double h) : TwoDShape(w, h) {   ◄——— Call the TwoDShape construct
    strcpy(style, str);
  }

  double area() {
    return getWidth() * getHeight() / 2;
  }

  void showStyle() {
```

```
    cout << "Triangle is " << style << "\n";
  }
};

int main() {
  Triangle t1("isosceles", 4.0, 4.0);
  Triangle t2("right", 8.0, 12.0);

  cout << "Info for t1:\n";
  t1.showStyle();
  t1.showDim();
  cout << "Area is " << t1.area() << "\n";

  cout << "\n";
  cout << "Info for t2:\n";
  t2.showStyle();
  t2.showDim();
  cout << "Area is " << t2.area() << "\n";

  return 0;
}
```

Here, Triangle( ) calls TwoDShape with the parameters w and h, which initializes width and height using these values. Triangle no longer initializes these values itself. It need only initialize the value unique to it: style. This leaves TwoDShape free to construct its subobject in any manner that it so chooses. Furthermore, TwoDShape can add functionality about which existing derived classes have no knowledge, thus preventing existing code from breaking.

Any form of constructor defined by the base class can be called by the derived class' constructor. The constructor executed will be the one that matches the arguments. For example, here are expanded versions of both TwoDShape and Triangle that include additional constructors:

```
// Add a constructor to TwoDShape.

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

// A class for two-dimensional objects.
class TwoDShape {
  // these are private
  double width;
  double height;
public:
```

```
  // Default constructor.
  TwoDShape() {
    width = height = 0.0;
  }

  // Constructor for TwoDShape.
  TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Construct object with equal width and height.
  TwoDShape(double x) {
    width = height = x;
  }

  void showDim() {
    cout << "Width and height are " <<
            width << " and " << height << "\n";
  }

  // accessor functions
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
  char style[20]; // now private
public:

  /* A default constructor. This automatically invokes
     the default constructor of TwoDShape. */
  Triangle() {
    strcpy(style, "unknown");
  }

  // Constructor with three parameters.
  Triangle(char *str, double w,
           double h) : TwoDShape(w, h) {
    strcpy(style, str);
  }

  // Construct an isosceles triangle.
  Triangle(double x) : TwoDShape(x) {
    strcpy(style, "isosceles");
```

Various **TwoDShape** constructors

Various **Triangle** constructors

```cpp
  }

  double area() {
    return getWidth() * getHeight() / 2;
  }

  void showStyle() {
    cout << "Triangle is " << style << "\n";
  }
};

int main() {
  Triangle t1;
  Triangle t2("right", 8.0, 12.0);
  Triangle t3(4.0);

  t1 = t2;

  cout << "Info for t1: \n";
  t1.showStyle();
  t1.showDim();
  cout << "Area is " << t1.area() << "\n";

  cout << "\n";

  cout << "Info for t2: \n";
  t2.showStyle();
  t2.showDim();
  cout << "Area is " << t2.area() << "\n";

  cout << "\n";

  cout << "Info for t3: \n";
  t3.showStyle();
  t3.showDim();
  cout << "Area is " << t3.area() << "\n";

  cout << "\n";

  return 0;
}
```

Here is the output from this version:

```
Info for t1:
Triangle is right
Width and height are 8 and 12
Area is 48
Info for t2: Triangle is right Width and height are 8 and 12
Area is 48
Info for t3: Triangle is isosceles Width and height are 4 and 4
```

```
Area is 8
```


Progress Check

1. How does a derived class execute its base class' constructor?
2. Can parameters be passed to a base class constructor?
3. What constructor is responsible for initializing the base class portion of a derived object, the one defined by the derived class or the one defined by the base class?

Project 10-1 Extending the Vehicle Class

This project creates a subclass of the Vehicle class first developed in Module 8.

As you should recall, Vehicle encapsulates information about vehicles, including the number of passengers they can carry, their fuel capacity, and their fuel consumption rate. We can use the Vehicle class as a starting point from which more specialized classes are developed. For example, one type of vehicle is a truck. An important attribute of a truck is its cargo capacity. Thus, to create a Truck class, you can inherit Vehicle, adding an instance variable that stores the carrying capacity. In this project, you will create the Truck class. In the process, the instance variables in Vehicle will be made private, and accessor functions are provided to get their values.

Step by Step

1. Create a file called TruckDemo.cpp, and copy the last implementation of Vehicle from Module 8 into the file.
2. Create the Truck class, as shown here:

```cpp
// Use Vehicle to create a Truck specialization.
class Truck : public Vehicle {
  int cargocap; // cargo capacity in pounds
public:

  // This is a constructor for Truck.
  Truck(int p, int f,
            int m, int c) : Vehicle(p, f, m)
  {
    cargocap = c;
  }

  // Accessor function for cargocap.
  int get_cargocap() { return cargocap; }
};
```

C++ A Beginner's Guide by Herbert Schildt

Here, Truck inherits Vehicle, adding the cargocap member. Thus, Truck includes all of the general vehicle attributes defined by Vehicle. It need add only those items that are unique to its own class.

3. Here is an entire program that demonstrates the Truck class:

```cpp
// Create a subclass of Vehicle called Truck.

#include <iostream>
using namespace std;

// Declare the Vehicle class.
class Vehicle {
  // These are private.
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon
public:
  // This is a constructor for Vehicle.
  Vehicle(int p, int f, int m) {
    passengers = p;
    fuelcap = f;
    mpg = m;

  }

  // Compute and return the range.
  int range() { return mpg * fuelcap; }

  // Accessor functions.
  int get_passengers() { return passengers; }
  int get_fuelcap() { return fuelcap; }
  int get_mpg() { return mpg; }
};

// Use Vehicle to create a Truck specialization
class Truck : public Vehicle {
  int cargocap; // cargo capacity in pounds
public:

  // This is a constructor for Truck.
  Truck(int p, int f,
            int m, int c) : Vehicle(p, f, m
  {
    cargocap = c;
  }

  // Accessor function for cargocap.
  int get_cargocap() { return cargocap; }
};
```

```
int main() {

  // construct some trucks
  Truck semi(2, 200, 7, 44000);
  Truck pickup(3, 28, 15, 2000);
  int dist = 252;

  cout << "Semi can carry " << semi.get_cargocap() <<
                  " pounds.\n";
  cout << "It has a range of " <<
          semi.range() << " miles.\n";
  cout << "To go " << dist << " miles semi needs " <<
          dist / semi.get_mpg() <<
          " gallons of fuel.\n\n";

  cout << "Pickup can carry " << pickup.get_cargocap() <<
                  " pounds.\n";
  cout << "It has a range of " <<
          pickup.range() << " miles.\n";

    cout << "To go " << dist << " miles pickup needs " <<
            dist / pickup.get_mpg() <<
            " gallons of fuel.\n";

    return 0;
  }
```

4. The output from this program is shown here:

```
Semi can carry 44000 pounds.
It has a range of 1400 miles.
To go 252 miles semi needs 36 gallons of fuel.

Pickup can carry 2000 pounds.
It has a range of 420 miles.
To go 252 miles pickup needs 16 gallons of fuel.
```

5. Many other types of classes can be derived from Vehicle. For example, the following skeleton creates an off-road class that stores the ground clearance of the vehicle:

```
// Create an off-road vehicle class
class OffRoad : public Vehicle {
  int groundClearance; // ground clearance in inches
public:
  // ...
};
```

The key point is that once you have created a base class that defines the general aspects of an object, that base class can be inherited to form specialized classes. Each derived class simply adds its own, unique attributes. This is the essence of inheritance.

CRITICAL SKILL 10.5: Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies consisting of only a base class and a derived class. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a derived class as a base class of another. For example, given three classes called A, B, and C, C can be derived from B, which can be derived from A. When this type of situation occurs, each derived class inherits all of the traits found in all of its base classes. In this case, C inherits all aspects of B and A.

To see how a multilevel hierarchy can be useful, consider the following program. In it, the derived class Triangle is used as a base class to create the derived class called ColorTriangle.

ColorTriangle inherits all of the traits of Triangle and TwoDShape, and adds a field called color, which holds the color of the triangle.

```cpp
// A multilevel hierarchy.

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

// A class for two-dimensional objects.
class TwoDShape {
  // these are private
  double width;
  double height;
public:

  // Default constructor.
  TwoDShape() {
    width = height = 0.0;
  }

  // Constructor for TwoDShape.
  TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Construct object with equal width and height.
  TwoDShape(double x) {
    width = height = x;
  }

  void showDim() {
    cout << "Width and height are " <<
            width << " and " << height << "\n";
  }
```

```cpp
  // accessor functions
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // now private
 public:

    /* A default constructor. This automatically invokes
       the default constructor of TwoDShape. */
    Triangle() {
      strcpy(style, "unknown");
    }

    // Constructor with three parameters.
    Triangle(char *str, double w,
             double h) : TwoDShape(w, h) {
      strcpy(style, str);
    }

    // Construct an isosceles triangle.
    Triangle(double x) : TwoDShape(x) {
      strcpy(style, "isosceles");
    }

    double area() {
      return getWidth() * getHeight() / 2;
    }

    void showStyle() {
      cout << "Triangle is " << style << "\n";
    }
};
```

```
// Extend Triangle.
class ColorTriangle : public Triangle {      ◄────────── ColorTriangle inherits Triangle,
  char color[20];                                         which inherits TwoDShape.
public:
  ColorTriangle(char *clr, char *style, double w,
                double h) : Triangle(style, w, h) {
    strcpy(color, clr);
  }

  // Display the color.
  void showColor() {
    cout << "Color is " << color << "\n";
  }
};

int main() {
  ColorTriangle t1("Blue", "right", 8.0, 12.0);
  ColorTriangle t2("Red", "isosceles", 2.0, 2.0);

  cout << "Info for t1:\n";
  t1.showStyle();
  t1.showDim();
  t1.showColor();
  cout << "Area is " << t1.area() << "\n";

  cout << "\n";

  cout << "Info for t2:\n";
  t2.showStyle();─┐
  t2.showDim();   ├──────────────────── A ColorTriangle object can call
  t2.showColor();─┘                     functions defined by itself and its
  cout << "Area is " << t2.area() << "\n";    base classes.

  return 0;
}
```

The output of this program is shown here:

```
Info for t1:
Triangle is right
Width and height are 8 and 12
Color is Blue
Area is 48

Info for t2:
Triangle is isosceles
Width and height are 2 and 2
Color is Red
Area is 2
```

Because of inheritance, ColorTriangle can make use of the previously defined classes of Triangle and TwoDShape, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point. In a class hierarchy, if a base class constructor requires parameters, then all derived classes must pass those parameters "up the line." This is true whether or not a derived class needs parameters of its own.

CRITICAL SKILL 10.6: Inheriting Multiple Base Classes

In C++, it is possible for a derived class to inherit two or more base classes at the same time. For example, in this short program, D inherits both B1 and B2:

```
// An example of multiple base classes.

#include <iostream>
using namespace std;

class B1 {
protected:
  int x;
public:
  void showx() { cout << x << "\n"; }
};

class B2 {
protected:
  int y;
public:
  void showy() { cout << y << "\n"; }
};

// Inherit multiple base classes.
class D: public B1, public B2 {          Here, D inherits both B1
public:                                  and B2 at the same time.
  /* x and y are accessible because they are
     protected in B1 and B2, not private. */
  void set(int i, int j) { x = i; y = j; }
};

int main()
{
  D ob;

  ob.set(10, 20); // provided by D
  ob.showx();       // from B1
  ob.showy();       // from B2

  return 0;
}
```

As this example illustrates, to cause more than one base class to be inherited, you must use a comma-separated list. Further, be sure to use an access specifier for each base class inherited.

CRITICAL SKILL 10.7: When Constructor and Destructor Functions Are Executed

Because a base class, a derived class, or both can contain constructors and/or destructors, it is important to understand the order in which they are executed. Specifically, when an object of a derived class comes into existence, in what order are the constructors called? When the object goes out of existence, in what order are the destructors called? To answer these questions, let's begin with this simple program:

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

class B {
public:
  B() { cout << "Constructing base portion\n"; }
  ~B() { cout << "Destructing base portion\n"; }
};

class D: public B {
public:
  D() { cout << "Constructing derived portion\n"; }
  ~D() { cout << "Destructing derived portion\n"; }
};

int main()
{
  D ob;

  // do nothing but construct and destruct ob

  return 0;
}
```

As the comment in main( ) indicates, this program simply constructs and then destroys an object called ob, which is of class D. When executed, this program displays

Constructing base portion Constructing derived portion Destructing derived portion Destructing base portion

As the output shows, first the constructor for B is executed, followed by the constructor of D. Next (since ob is immediately destroyed in this program), the destructor of D is called, followed by that of B.

The results of the foregoing experiment can be generalized as follows: When an object of a derived class is created, the base class constructor is called first, followed by the constructor for the derived class. When a derived object is destroyed, its destructor is called first, followed by that of the base class. Put differently, constructors are executed in the order of their derivation. Destructors are executed in reverse order of derivation. In the case of a multilevel class hierarchy (that is, where a derived class becomes the base class for another derived class), the same general rule applies: Constructors are called

in order of derivation; destructors are called in reverse order. When a class inherits more than one base class at a time, constructors are called in order from left to right as specified in the derived class' inheritance list. Destructors are called in reverse order right to left.

Progress Check

1. Can a derived class be used as a base class for another derived class?
2. In a class hierarchy, in what order are the constructors called?
3. In a class hierarchy, in what order are the destructors called?

Ask the Expert

Q: Why are constructors called in order of derivation, and destructors called in reverse order?

A: If you think about it, it makes sense that constructors are executed in order of derivation.

Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from, and possibly prerequisite to, any initialization performed by the derived class. Therefore, the base class constructor must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Since the base class underlies a derived class, the destruction of the base class implies the destruction of the derived class. Therefore, the derived destructor must be called before the object is fully destroyed.

CRITICAL SKILL 10.8: Pointers to Derived Types

Before moving on to virtual functions and polymorphism, it is necessary to discuss an important aspect of pointers. Pointers to base classes and derived classes are related in ways that other types of pointers are not. In general, a pointer of one type cannot point to an object of another type. However, base class pointers and derived objects are the exceptions to this rule. In C++, a base class pointer can also be used to point to an object of any class derived from that base. For example, assume that you have a base class called B and a class called D, which is derived from B. Any pointer declared as a pointer to B can also be used to point to an object of type D. Therefore, given

```
B *p;   // pointer to object of type B
B B_ob; // object of type B
D D_ob; // object of type D
```

both of the following statements are perfectly valid:

```
p = &B_ob; // p points to object of type B
p = &D_ob; /* p points to object of type D,
              which is an object derived from B */
```

A base pointer can be used to access only those parts of a derived object that were inherited from the base class. Thus, in this example, p can be used to access all elements of D_ob inherited from B_ob. However, elements specific to D_ob cannot be accessed through p.

Another point to understand is that although a base pointer can be used to point to a derived object, the reverse is not true. That is, you cannot access an object of the base type by using a derived class pointer.

As you know, a pointer is incremented and decremented relative to its base type. Therefore, when a base class pointer is pointing at a derived object, incrementing or decrementing it will not make it point to the next object of the derived class. Instead, it will point to (what it thinks is) the next object of the base class. Therefore, you should consider it invalid to increment or decrement a base class pointer when it is pointing to a derived object.

The fact that a pointer to a base type can be used to point to any object derived from that base is extremely important, and fundamental to C++. As you will soon learn, this flexibility is crucial to the way C++ implements runtime polymorphism.

References to Derived Types

Similar to the action of pointers just described, a base class reference can be used to refer to an object of a derived type. The most common application of this is found in function parameters. A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

CRITICAL SKILL 10.9: Virtual Functions and Polymorphism

The foundation upon which C++ builds its support for polymorphism consists of inheritance and base class pointers. The specific feature that actually implements polymorphism is the virtual function. The remainder of this module examines this important feature.

Virtual Function Fundamentals

A virtual function is a function that is declared as virtual in a base class and redefined in one or more derived classes. Thus, each derived class can have its own version of a virtual function.

What makes virtual functions interesting is what happens when a base class pointer is used to call one. When a virtual function is called through a base class pointer, C++ determines which version of that function to call based upon the type of the object pointed to by the pointer. This determination is made at runtime. Thus, when different objects are pointed to, different versions of the virtual function are executed. In other words, it is the type of the object being pointed to (not the type of the pointer) that determines which version of the virtual function will be executed. Therefore, if a base class contains a virtual function and if two or more different classes are derived from that base class, then when different types of objects are pointed to through a base class pointer, different versions of the virtual

function are executed. The same effect occurs when a virtual function is called through a base class reference.

You declare a virtual function as virtual inside a base class by preceding its declaration with the keyword virtual. When a virtual function is redefined by a derived class, the keyword virtual need not be repeated (although it is not an error to do so).

A class that includes a virtual function is called a polymorphic class. This term also applies to a class that inherits a base class containing a virtual function.

The following program demonstrates a virtual function:

```
// A short example that uses a virtual function.

#include <iostream>
using namespace std;

class B {                                        Declare a virtual function.
public:
  virtual void who() { // specify a virtual function   ◄────────┐
    cout << "Base\n";
  }
};

class D1 : public B {
public:
  void who() { // redefine who() for D1   ◄──────── Redefine the virtual function for D1.
    cout << "First derivation\n";
  }
};

class D2 : public B {
public:
  void who() { // redefine who() for D2   ◄──────── Redefine the virtual function a
    cout << "Second derivation\n";                   second time for D2.
  }
};

int main()
{
  B base_obj;
  B *p;
  D1 D1_obj;
  D2 D2_obj;

  p = &base_obj;
  p->who();  // access B's who ──┐

  p = &D1_obj;
  p->who(); // access D1's who ──┼──────── Call the virtual function through a
                                           base class pointer.
  p = &D2_obj;
  p->who();  // access D2's who ─┘

  return 0;
}
```

This program produces the following output:

```
Base
First derivation
Second derivation
```

Let's examine the program in detail to understand how it works.

As you can see, in B, the function who( ) is declared as virtual. This means that the function can be redefined by a derived class. Inside both D1 and D2, who( ) is redefined relative to each class. Inside main( ), four variables are declared: base_obj, which is an object of type B; p, which is a pointer to B

objects; and D1_obj and D2_obj, which are objects of the two derived classes. Next, p is assigned the address of base_obj, and the who( ) function is called. Since who( ) is declared as virtual, C++ determines at runtime which version of who( ) to execute based on the type of object pointed to by p. In this case, p points to an object of type B, so it is the version of who( ) declared in B that is executed. Next, p is assigned the address of D1_obj. Recall that a base class pointer can refer to an object of any derived class. Now, when who( ) is called, C++ again checks to see what type of object is pointed to by p and, based on that type, determines which version of who( ) to call. Since p points to an object of type D1, that version of who( ) is used. Likewise, when p is assigned the address of D2_obj, the version of who( ) declared inside D2 is executed.

To review: When a virtual function is called through a base class pointer, the version of the virtual function actually executed is determined at runtime by the type of object being pointed to.

Although virtual functions are normally called through base class pointers, a virtual function can also be called normally, using the standard dot operator syntax. This means that in the preceding example, it would have been syntactically correct to access who( ) using this statement:

D1_obj.who();

However, calling a virtual function in this manner ignores its polymorphic attributes. It is only when a virtual function is accessed through a base class pointer (or reference) that runtime polymorphism is achieved.

At first, the redefinition of a virtual function in a derived class seems to be a special form of function overloading. However, this is not the case. In fact, the two processes are fundamentally different. First, an overloaded function must differ in its type and/or number of parameters, while a redefined virtual function must have exactly the same type and number of parameters. In fact, the prototypes for a virtual function and its redefinitions must be exactly the same. If the prototypes differ, then the function is simply considered to be overloaded, and its virtual nature is lost. Another restriction is that a virtual function must be a member, not a friend, of the class for which it is defined. However, a virtual function can be a friend of another class. Also, it is permissible for destructors, but not constructors, to be virtual.

Because of the restrictions and differences between overloading normal functions and redefining virtual functions, the term overriding is used to describe the redefinition of a virtual function.

Virtual Functions Are Inherited

Once a function is declared as virtual, it stays virtual no matter how many layers of derived classes it may pass through. For example, if D2 is derived from D1 instead of B, as shown in the next example, then who( ) is still virtual:

```
// Derive from D1, not B.
class D2 : public D1 {
public:
  void who() { // define who() relative to second_d
    cout << "Second derivation\n";
  }
};
```

When a derived class does not override a virtual function, then the function as defined in the base class is used. For example, try this version of the preceding program. Here, D2 does not override who( ):

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

class B {
public:
  virtual void who() {
    cout << "Base\n";
  }
};

class D1 : public B {
public:
  void who() {
    cout << "First derivation\n";
  }
};

class D2 : public B {          ◄────────── D2 does not override who().
// who() not defined
};

int main()
{
  B base_obj;
  B *p;
  D1 D1_obj;
  D2 D2_obj;

  p = &base_obj;
  p->who();   // access B's who()

  p = &D1_obj;
  p->who(); // access D1's who()

  p = &D2_obj;
  p->who(); /* access B's who() because   ◄──────── This calls the who() defined by B.
             D2 does not redefine it */

  return 0;
}
```

The program now outputs the following:

```
Base
First derivation
Base
```

Because D2 does not override who( ), the version of who( ) defined in B is used instead.

Keep in mind that inherited characteristics of virtual are hierarchical. Therefore, if the preceding example is changed such that D2 is derived from D1 instead of B, then when who( ) is called on an object of type D2, it will not be the who( ) inside B, but the version of who( ) declared inside D1 that is called since it is the class closest to D2.

Why Virtual Functions?

As stated earlier, virtual functions in combination with derived types allow C++ to support runtime polymorphism. Polymorphism is essential to object-oriented programming, because it allows a generalized class to specify those functions that will be common to all derivatives of that class, while allowing a derived class to define the specific implementation of some or all of those functions. Sometimes this idea is expressed as follows: the base class dictates the general interface that any object derived from that class will have, but lets the derived class define the actual method used to implement that interface. This is why the phrase "one interface, multiple methods" is often used to describe polymorphism.

Part of the key to successfully applying polymorphism is understanding that the base and derived classes form a hierarchy, which moves from greater to lesser generalization (base to derived). When designed correctly, the base class provides all of the elements that a derived class can use directly. It also defines those functions that the derived class must implement on its own. This allows the derived class the flexibility to define its own methods, and yet still enforces a consistent interface. That is, since the form of the interface is defined by the base class, any derived class will share that common interface. Thus, the use of virtual functions makes it possible for the base class to define the generic interface that will be used by all derived classes.

At this point, you might be asking yourself why a consistent interface with multiple implementations is important. The answer, again, goes back to the central driving force behind object-oriented programming: It helps the programmer handle increasingly complex programs. For example, if you develop your program correctly, then you know that all objects you derive from a base class are accessed in the same general way, even if the specific actions vary from one derived class to the next. This means that you need to deal with only one interface, rather than several. Also, your derived class is free to use any or all of the functionality provided by the base class. You need not reinvent those elements.

The separation of interface and implementation also allows the creation of class libraries, which can be provided by a third party. If these libraries are implemented correctly, they will provide a common interface that you can use to derive classes of your own that meet your specific needs. For example, both the Microsoft Foundation Classes (MFC) and the newer .NET Framework Windows Forms class library support Windows programming. By using these classes, your program can inherit much of the

functionality required by a Windows program. You need add only the features unique to your application. This is a major benefit when programming complex systems.

Applying Virtual Functions

To better understand the power of virtual functions, we will apply it to the TwoDShape class. In the preceding examples, each class derived from TwoDShape defines a function called area( ). This suggests that it might be better to make area( ) a virtual function of the TwoDShape class, allowing each derived class to override it, defining how the area is calculated for the type of shape that the class encapsulates. The following program does this. For convenience, it also adds a name field to TwoDShape. (This makes it easier to demonstrate the classes.)

```cpp
// Use virtual functions and polymorphism.

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

// A class for two-dimensional objects.
class TwoDShape {
  // these are private
  double width;
  double height;

  // add a name field
  char name[20];
public:

  // Default constructor.
  TwoDShape() {
    width = height = 0.0;
    strcpy(name, "unknown");
  }

  // Constructor for TwoDShape.
  TwoDShape(double w, double h, char *n) {
    width = w;
    height = h;
    strcpy(name, n);
  }

  // Construct object with equal width and height.
  TwoDShape(double x, char *n) {
    width = height = x;
    strcpy(name, n);
  }
```

```
  void showDim() {
    cout << "Width and height are " <<
             width << " and " << height << "\n";
  }

  // accessor functions
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
  char *getName() { return name; }

  // Add area() to TwoDShape and make it virtual.
  virtual double area() {                           ←——— The area() function is
    cout << "Error: area() must be overridden.\n";        now virtual.

    return 0.0;
  }

};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
  char style[20]; // now private
public:

  /* A default constructor. This automatically invokes
     the default constructor of TwoDShape. */
  Triangle() {
    strcpy(style, "unknown");
  }

  // Constructor with three parameters.
  Triangle(char *str, double w,
           double h) : TwoDShape(w, h, "triangle") {
    strcpy(style, str);
  }

  // Construct an isosceles triangle.
  Triangle(double x) : TwoDShape(x, "triangle") {
    strcpy(style, "isosceles");
  }

  // This now overrides area() declared in TwoDShape.
  double area() {                          ←——— Override area()
    return getWidth() * getHeight() / 2;         in Triangle.
  }

  void showStyle() {
    cout << "Triangle is " << style << "\n";
  }
};
```

```
// A derived class of TwoDShape for rectangles.
class Rectangle : public TwoDShape {
public:

  // Construct a rectangle.
  Rectangle(double w, double h) :
    TwoDShape(w, h, "rectangle") { }

  // Construct a square.

  Rectangle(double x) :
    TwoDShape(x, "rectangle") { }

  bool isSquare() {
    if(getWidth() == getHeight()) return true;
    return false;
  }

  // This is another override of area().
  double area() {                              <──────────── Override area() again
    return getWidth() * getHeight();                         in Rectangle.
  }
};

int main() {
  // declare an array of pointers to TwoDShape objects.
  TwoDShape *shapes[5];

  shapes[0] = &Triangle("right", 8.0, 12.0);
  shapes[1] = &Rectangle(10);
  shapes[2] = &Rectangle(10, 4);
  shapes[3] = &Triangle(7.0);
  shapes[4] = &TwoDShape(10, 20, "generic");

  for(int i=0; i < 5; i++) {
    cout << "object is " <<
            shapes[i]->getName() << "\n";

    cout << "Area is " <<
            shapes[i]->area() << "\n";      <──────────── The proper version of area()
                                                          for each object is now called.
    cout << "\n";
  }

  return 0;
}
```

The output from the program is shown here:

```
object is triangle Area is 48
object is rectangle Area is 100
object is rectangle
```

```
Area is 40
object is triangle Area is 24.5
object is generic
Error: area() must be overridden.
Area is 0
```

Let's examine this program closely. First, area( ) is declared as virtual in TwoDShape class and is overridden by Triangle and Rectangle. Inside TwoDShape, area( ) is given a placeholder implementation that simply informs the user that this function must be overridden by a derived class. Each override of area( ) supplies an implementation that is suitable for the type of object encapsulated by the derived class. Thus, if you were to implement an ellipse class, for example, then area( ) would need to compute the area of an ellipse.

There is one other important feature in the preceding program. Notice in main( ) that shapes is declared as an array of pointers to TwoDShape objects. However, the elements of this array are assigned pointers to Triangle, Rectangle,and TwoDShape objects. This is valid because a base class pointer can point to a derived class object. The program then cycles through the array, displaying information about each object. Although quite simple, this illustrates the power of both inheritance and virtual functions. The type of object pointed to by a base class pointer is determined at runtime and acted on accordingly. If an object is derived from TwoDShape,then its area can be obtained by calling area( ). The interface to this operation is the same no matter what type of shape is being used.

Progress Check

1. What is a virtual function?
2. Why are virtual functions important?
3. When an overridden virtual function is called through a base class pointer, which version of the function is executed?

CRITICAL SKILL 10.10: Pure Virtual Functions and Abstract Classes

Sometimes you will want to create a base class that defines only a generalized form that will be shared by all of its derived classes, leaving it to each derived class to fill in the details. Such a class determines the nature of the functions that the derived classes must implement, but does not, itself, provide an implementation of one or more of these functions. One way this situation can occur is when a base class is unable to create a meaningful implementation for a function. This is the case with the version of TwoDShape used in the preceding example. The definition of area( ) is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a function to have no meaningful definition in the context of its base class. You can handle this situation two ways. One way,

as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have functions that must be overridden by the derived class in order for the derived class to have any meaning. Consider the class Triangle. It has no meaning if area( ) is not defined. In this case, you want some way to ensure that a derived class does, indeed, override all necessary functions. The C++ solution to this problem is the pure virtual function.

A pure virtual function is a function declared in a base class that has no definition relative to the base. As a result, any derived class must define its own version—it cannot simply use the version defined in the base. To declare a pure virtual function, use this general form: virtual type func-name(parameter-list) = 0;

Here, type is the return type of the function, and func-name is the name of the function. Using a pure virtual function, you can improve the TwoDShape class. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the preceding program declares area( ) as a pure virtual function inside TwoDShape. This, of course, means that all classes derived from TwoDShape must override area( ).

```
// Use a pure virtual function.

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

// A class for two-dimensional objects.
class TwoDShape {
  // these are private
  double width;
  double height;

  // add a name field
  char name[20];
public:

  // Default constructor.
  TwoDShape() {
    width = height = 0.0;
    strcpy(name, "unknown");
  }

  // Constructor for TwoDShape.
  TwoDShape(double w, double h, char *n) {
    width = w;
    height = h;
    strcpy(name, n);
  }
```

```cpp
    // Construct object with equal width and height.
    TwoDShape(double x, char *n) {
      width = height = x;
      strcpy(name, n);
    }

    void showDim() {
      cout << "Width and height are " <<
              width << " and " << height << "\n";
    }

    // accessor functions
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
    char *getName() { return name; }

    // area()is now a pure virtual function
    virtual double area() = 0;                          ←———————— area( ) is now a pure
                                                                  virtual function.
};

// Triangle is derived from TwoDShape.
class Triangle : public TwoDShape {
  char style[20]; // now private

 public:

    /* A default constructor. This automatically invokes
       the default constructor of TwoDShape. */
    Triangle() {
      strcpy(style, "unknown");
    }

    // Constructor with three parameters.
    Triangle(char *str, double w,
             double h) : TwoDShape(w, h, "triangle") {
      strcpy(style, str);
    }

    // Construct an isosceles triangle.
    Triangle(double x) : TwoDShape(x, "triangle") {
      strcpy(style, "isosceles");
    }

    // This now overrides area() declared in TwoDShape.
    double area() {
      return getWidth() * getHeight() / 2;
    }
```

```cpp
    void showStyle() {
      cout << "Triangle is " << style << "\n";
    }
};

// A derived class of TwoDShape for rectangles.
class Rectangle : public TwoDShape {
public:

  // Construct a rectangle.
  Rectangle(double w, double h) :
    TwoDShape(w, h, "rectangle") { }

  // Construct a square.
  Rectangle(double x) :
    TwoDShape(x, "rectangle") { }

  bool isSquare() {
    if(getWidth() == getHeight()) return true;
    return false;
  }
  // This is another override of area().
  double area() {
    return getWidth() * getHeight();
  }
};

int main() {
  // declare an array of pointers to TwoDShape objects.
  TwoDShape *shapes[4];

  shapes[0] = &Triangle("right", 8.0, 12.0);
  shapes[1] = &Rectangle(10);
  shapes[2] = &Rectangle(10, 4);
  shapes[3] = &Triangle(7.0);

  for(int i=0; i < 4; i++) {
    cout << "object is " <<
            shapes[i]->getName() << "\n";

    cout << "Area is " <<
            shapes[i]->area() << "\n";

    cout << "\n";
  }

  return 0;
}
```

If a class has at least one pure virtual function, then that class is said to be abstract. An abstract class has one important feature: there can be no objects of that class. To prove this to yourself, try removing the

override of area( ) from the Triangle class in the preceding program. You will receive an error when you try to create an instance of Triangle. Instead, an abstract class must be used only as a base that other classes will inherit. The reason that an abstract class cannot be used to declare an object is because one or more of its functions have no definition. Because of this, the shapes array in the preceding program has been shortened to 4, and a generic TwoDShape object is no longer created. As the program illustrates, even if the base class is abstract, you still can use it to declare a pointer of its type, which can be used to point to derived class objects.

Module 10 Mastery Check

1. A class that is inherited is called a _____ class. The class that does the inheriting is called a _____ class.
2. Does a base class have access to the members of a derived class? Does a derived class have access to the members of a base class?
3. Create a derived class of TwoDShape called Circle. Include an area( ) function that computes the area of the circle.
4. How do you prevent a derived class from having access to a member of a base class?
5. Show the general form of a constructor that calls a base class constructor.
6. Given the following hierarchy:

```
class Alpha { ...

class Beta : public Alpha { ...

Class Gamma : public Beta { ...
```

   in what order are the constructors for these classes called when a Gamma object is instantiated?

7. How can protected members be accessed?
8. A base class pointer can refer to a derived class object. Explain why this is important as it relates to function overriding.
9. What is a pure virtual function? What is an abstract class?
10. Can an object of an abstract class be instantiated?
11. Explain how the pure virtual function helps implement the "one interface, multiple methods" aspect of polymorphism.