Illustrations List *(Main Page)*

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Fig. 9.1 Some simple inheritance examples.

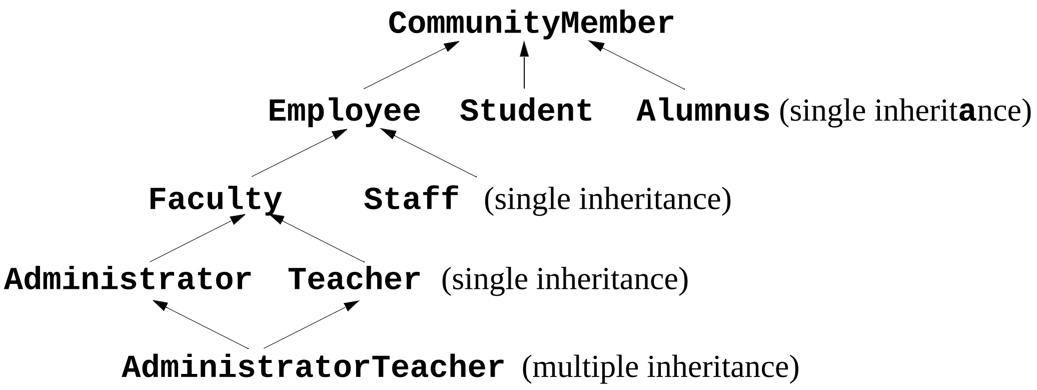
CommunityMember
Employee
Student
Alumnus (single inheritance)
Faculty
Staff (single inheritance)
Administrator
Teacher (single inheritance)
AdministratorTeacher (multiple inheritance)

Fig. 9.2 An inheritance hierarchy for university community members.

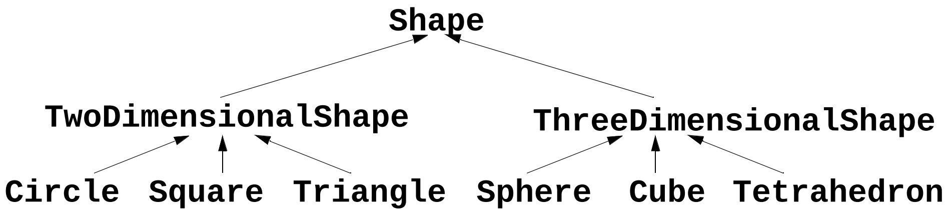

Fig. 9.3 A portion of a **`Shape`** class hierarchy.

```
// Fig. 9.4: point.h
// Definition of class Point
#ifndef POINT_H
#define POINT_H

class Point {
   friend ostream &operator<<( ostream &, const Point & );
public:
   Point( int = 0, int = 0 );      // default constructor
   void setPoint( int, int );      // set coordinates
   int getX() const { return x; }  // get x coordinate
   int getY() const { return y; }  // get y coordinate
protected:         // accessible by derived classes
   int x, y;       // x and y coordinates of the Point
};

#endif
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 1 of 6).

```
// Fig. 9.4: point.cpp
// Member functions for class Point
#include <iostream.h>
#include "point.h"

// Constructor for class Point
Point::Point( int a, int b ) { setPoint( a, b ); }

// Set x and y coordinates of Point
void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

// Output Point (with overloaded stream insertion operator)
ostream &operator<<( ostream &output, const Point &p )
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // enables cascaded calls
}
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 2 of 6).

```
// Fig. 9.4: circle.h
// Definition of class Circle
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream.h>
#include <iomanip.h>
#include "point.h"

class Circle : public Point {  // Circle inherits from Point
   friend ostream &operator<<( ostream &, const Circle & );
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius
   double area() const;        // calculate area
protected:
   double radius;
};

#endif
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 3 of 6).

```
// Fig. 9.4: circle.cpp
// Member function definitions for class Circle
#include "circle.h"

// Constructor for Circle calls constructor for Point
// with a member initializer then initializes radius.
Circle::Circle( double r, int a, int b )
   : Point( a, b )       // call base-class constructor
{ setRadius( r ); }

// Set radius of Circle
void Circle::setRadius( double r )
   { radius = ( r >= 0 ? r : 0 ); }

// Get radius of Circle
double Circle::getRadius() const { return radius; }

// Calculate area of Circle
double Circle::area() const
   { return 3.14159 * radius * radius; }

// Output a Circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<( ostream &output, const Circle &c )
{
   output << "Center = " << static_cast< Point >( c )
          << "; Radius = "
          << setiosflags( ios::fixed | ios::showpoint )
          << setprecision( 2 ) << c.radius;

   return output;   // enables cascaded calls
}
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 4 of 6).

```
// Fig. 9.4: fig09_04.cpp
// Casting base-class pointers to derived-class pointers
#include <iostream.h>
#include <iomanip.h>
#include "point.h"
#include "circle.h"

int main()
{
   Point *pointPtr = 0, p( 30, 50 );
   Circle *circlePtr = 0, c( 2.7, 120, 89 );

   cout << "Point p: " << p << "\nCircle c: " << c << '\n';

   // Treat a Circle as a Point (see only the base class part)
   pointPtr = &c;   // assign address of Circle to pointPtr
   cout << "\nCircle c (via *pointPtr): "
        << *pointPtr << '\n';

   // Treat a Circle as a Circle (with some casting)
   pointPtr = &c;   // assign address of Circle to pointPtr

   // cast base-class pointer to derived-class pointer
   circlePtr = static_cast< Circle * >( pointPtr );
   cout << "\nCircle c (via *circlePtr):\n" << *circlePtr
        << "\nArea of c (via circlePtr): "
        << circlePtr->area() << '\n';

   // DANGEROUS: Treat a Point as a Circle
   pointPtr = &p;   // assign address of Point to pointPtr

   // cast base-class pointer to derived-class pointer
   circlePtr = static_cast< Circle * >( pointPtr );
   cout << "\nPoint p (via *circlePtr):\n" << *circlePtr
        << "\nArea of object circlePtr points to: "
        << circlePtr->area() << endl;
   return 0;
}
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 5 of 6).

```
Point p: [30, 50]
Circle c: Center = [120, 89]; Radius = 2.70

Circle c (via *pointPtr): [120, 89]

Circle c (via *circlePtr):
Center = [120, 89]; Radius = 2.70
Area of c (via circlePtr): 22.90

Point p (via *circlePtr):
Center = [30, 50]; Radius = 0.00
Area of object circlePtr points to: 0.00
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 6 of 6).

```
// Fig. 9.5: employ.h
// Definition of class Employee
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee( const char *, const char * );  // constructor
   void print() const;  // output first and last name
   ~Employee();         // destructor
private:
   char *firstName;     // dynamically allocated string
   char *lastName;      // dynamically allocated string
};

#endif
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 1 of 5).

```
// Fig. 9.5: employ.cpp
// Member function definitions for class Employee
#include <string.h>
#include <iostream.h>
#include <assert.h>
#include "employ.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 ); // terminate if not allocated
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );  // terminate if not allocated
   strcpy( lastName, last );
}

// Output employee name
void Employee::print() const
   { cout << firstName << ' ' << lastName; }

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;   // reclaim dynamic memory
   delete [] lastName;    // reclaim dynamic memory
}
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 2 of 5).

```
// Fig. 9.5: hourly.h
// Definition of class HourlyWorker
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker( const char*, const char*, double, double );
   double getPay() const;  // calculate and return salary
   void print() const;     // overridden base-class print
private:
   double wage;            // wage per hour
   double hours;           // hours worked for week
};

#endif
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 3 of 5).

```
// Fig. 9.5: hourly.cpp
// Member function definitions for class HourlyWorker
#include <iostream.h>
#include <iomanip.h>
#include "hourly.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker( const char *first,
                            const char *last,
                            double initHours, double initWage )
   : Employee( first, last )   // call base-class constructor
{
   hours = initHours;  // should validate
   wage = initWage;    // should validate
}

// Get the HourlyWorker's pay
double HourlyWorker::getPay() const { return wage * hours; }

// Print the HourlyWorker's name and pay
void HourlyWorker::print() const
{
   cout << "HourlyWorker::print() is executing\n\n";
   Employee::print();   // call base-class print function

   cout << " is an hourly worker with pay of $"
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 ) << getPay() << endl;
}
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 4 of 5).

```
// Fig. 9.5: fig.09_05.cpp
// Overriding a base-class member function in a
// derived class.
#include <iostream.h>
#include "hourly.h"

int main()
{
   HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
   h.print();
   return 0;
}
```

```
HourlyWorker::print() is executing

Bob Smith is an hourly worker with pay of $400.00
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 5 of 5).

Base class member access specifier	Type of inheritance		
	`public` inheritance	**`protected`** inheritance	**`private`** inheritance
`public`	**`public`** in derived class. Can be accessed directly by any non-**`static`** member functions, **`friend`** functions and non-member functions.	**`protected`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.	**`private`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.
`protected`	**`protected`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.	**`protected`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.	**`private`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.
`private`	Hidden in derived class. Can be accessed by non-**`static`** member functions and **`friend`** functions through **`public`** or **`protected`** member functions of the base class.	Hidden in derived class. Can be accessed by non-**`static`** member functions and **`friend`** functions through **`public`** or **`protected`** member functions of the base class.	Hidden in derived class. Can be accessed by non-**`static`** member functions and **`friend`** functions through **`public`** or **`protected`** member functions of the base class.

Fig. 9.6 Summary of base-class member accessibility in a derived class.

```
// Fig. 9.7: point2.h
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
public:
   Point( int = 0, int = 0 );  // default constructor
   ~Point();    // destructor
protected:      // accessible by derived classes
   int x, y;    // x and y coordinates of Point
};

#endif
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 1 of 5).

```
// Fig. 9.7: point2.cpp
// Member function definitions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point( int a, int b )
{
   x = a;
   y = b;

   cout << "Point  constructor: "
        << '[' << x << ", " << y << ']' << endl;
}

// Destructor for class Point
Point::~Point()
{
   cout << "Point  destructor:  "
        << '[' << x << ", " << y << ']' << endl;
}
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 2 of 5).

```
// Fig. 9.7: circle2.h
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );

   ~Circle();
private:
   double radius;
};

#endif
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 3 of 5).

```
// Fig. 9.7: circle2.cpp
// Member function definitions for class Circle
#include "circle2.h"

// Constructor for Circle calls constructor for Point
Circle::Circle( double r, int a, int b )
   : Point( a, b )   // call base-class constructor
{
   radius = r;  // should validate
   cout << "Circle constructor: radius is "
        << radius << " [" << x << ", " << y << ']' << endl;
}

// Destructor for class Circle
Circle::~Circle()
{
   cout << "Circle destructor:  radius is "
        << radius << " [" << x << ", " << y << ']' << endl;
}
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 4 of 5).

```
// Fig. 9.7: fig09_07.cpp
// Demonstrate when base-class and derived-class
// constructors and destructors are called.
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

int main()
{
   // Show constructor and destructor calls for Point
   {
      Point p( 11, 22 );
   }

   cout << endl;
   Circle circle1( 4.5, 72, 29 );
   cout << endl;
   Circle circle2( 10, 5, 5 );
   cout << endl;
   return 0;
}
```

```
Point  constructor: [11, 22]
Point  destructor:  [11, 22]

Point  constructor: [72, 29]
Circle constructor: radius is 4.5 [72, 29]

Point  constructor: [5, 5]
Circle constructor: radius is 10 [5, 5]

Circle destructor:  radius is 10 [5, 5]
Point  destructor:  [5, 5]
Circle destructor:  radius is 4.5 [72, 29]
Point  destructor:  [72, 29]
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 5 of 5).

```
// Fig. 9.8: point2.h
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
   friend ostream &operator<<( ostream &, const Point & );
public:
   Point( int = 0, int = 0 );     // default constructor
   void setPoint( int, int );      // set coordinates
   int getX() const { return x; }  // get x coordinate
   int getY() const { return y; }  // get y coordinate
protected:        // accessible to derived classes
   int x, y;      // coordinates of the point
};

#endif
```

Fig. 9.8 Demonstrating class **`Point`** (part 1 of 3).

```
// Fig. 9.8: point2.cpp
// Member functions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point( int a, int b ) { setPoint( a, b ); }

// Set the x and y coordinates
void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

// Output the Point
ostream &operator<<( ostream &output, const Point &p )
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;          // enables cascading
}
```

Fig. 9.8 Demonstrating class **`Point`** (part 2 of 3).

```
// Fig. 9.8: fig09_08.cpp
// Driver for class Point
#include <iostream.h>
#include "point2.h"

int main()
{
   Point p( 72, 115 );   // instantiate Point object p

   // protected data of Point inaccessible to main
   cout << "X coordinate is " << p.getX()
        << "\nY coordinate is " << p.getY();

   p.setPoint( 10, 10 );
   cout << "\n\nThe new location of p is " << p << endl;

   return 0;
}
```

```
X coordinate is 72
Y coordinate is 115

The new location of p is [10, 10]
```

Fig. 9.8 Demonstrating class **`Point`** (part 3 of 3).

```
// Fig. 9.9: circle2.h
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
   friend ostream &operator<<( ostream &, const Circle & );
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );
   void setRadius( double );    // set radius
   double getRadius() const;    // return radius
   double area() const;         // calculate area
protected:          // accessible to derived classes
   double radius;   // radius of the Circle
};

#endif
```

Fig. 9.9 Demonstrating class **`Circle`** (part 1 of 5).

```
// Fig. 9.9: circle2.cpp
// Member function definitions for class Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle2.h"

// Constructor for Circle calls constructor for Point
// with a member initializer and initializes radius
Circle::Circle( double r, int a, int b )
   : Point( a, b )       // call base-class constructor
{ setRadius( r ); }

// Set radius
void Circle::setRadius( double r )
   { radius = ( r >= 0 ? r : 0 ); }

// Get radius
double Circle::getRadius() const { return radius; }
```

Fig. 9.9 Demonstrating class **`Circle`** (part 2 of 5).

```

// Calculate area of Circle
double Circle::area() const
   { return 3.14159 * radius * radius; }

// Output a circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<( ostream &output, const Circle &c )
{
   output << "Center = " << static_cast< Point > ( c )
          << "; Radius = "
          << setiosflags( ios::fixed | ios::showpoint )
          << setprecision( 2 ) << c.radius;

   return output;   //  enables cascaded calls
}
```

Fig. 9.9 Demonstrating class `Circle` (part 3 of 5).

```
// Fig. 9.9: fig09_09.cpp
// Driver for class Circle
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

int main()
{
   Circle c( 2.5, 37, 43 );

   cout << "X coordinate is " << c.getX()
        << "\nY coordinate is " << c.getY()
        << "\nRadius is " << c.getRadius();

   c.setRadius( 4.25 );
   c.setPoint( 2, 2 );
   cout << "\n\nThe new location and radius of c are\n"
        << c << "\nArea " << c.area() << '\n';

   Point &pRef = c;
   cout << "\nCircle printed as a Point is: " << pRef << endl;

   return 0;
}
```

Fig. 9.9 Demonstrating class `Circle` (part 4 of 5).

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area 56.74

Circle printed as a Point is: [2, 2]
```

Fig. 9.9 Demonstrating class `Circle` (part 5 of 5).

```
// Fig. 9.10: cylindr2.h
// Definition of class Cylinder
#ifndef CYLINDR2_H
#define CYLINDR2_H

#include "circle2.h"

class Cylinder : public Circle {
   friend ostream &operator<<( ostream &, const Cylinder & );

public:
   // default constructor
   Cylinder( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

   void setHeight( double );   // set height
   double getHeight() const;   // return height
   double area() const;        // calculate and return area
   double volume() const;      // calculate and return volume

protected:
   double height;              // height of the Cylinder
};

#endif
```

Fig. 9.10 Demonstrating class **`Cylinder`** (part 1 of 5).

```
// Fig. 9.10: cylindr2.cpp
// Member and friend function definitions
// for class Cylinder.
#include <iostream.h>
#include <iomanip.h>
#include "cylindr2.h"

```

Fig. 9.10 Demonstrating class **`Cylinder`** (part 2 of 5).

```
// Cylinder constructor calls Circle constructor
Cylinder::Cylinder( double h, double r, int x, int y )
   : Circle( r, x, y )   // call base-class constructor
{ setHeight( h ); }

// Set height of Cylinder
void Cylinder::setHeight( double h )
   { height = ( h >= 0 ? h : 0 ); }

// Get height of Cylinder
double Cylinder::getHeight() const { return height; }

// Calculate area of Cylinder (i.e., surface area)
double Cylinder::area() const
{
   return 2 * Circle::area() +
          2 * 3.14159 * radius * height;
}

// Calculate volume of Cylinder
double Cylinder::volume() const
   { return Circle::area() * height; }

// Output Cylinder dimensions
ostream &operator<<( ostream &output, const Cylinder &c )
```

```
{
   output << static_cast< Circle >( c )
          << "; Height = " << c.height;

   return output;   // enables cascaded calls
}
```

Fig. 9.10 Demonstrating class `Cylinder` (part 3 of 5).

```
// Fig. 9.10: fig09_10.cpp
// Driver for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "point2.h"
#include "circle2.h"
#include "cylindr2.h"

int main()
{
   // create Cylinder object
   Cylinder cyl( 5.7, 2.5, 12, 23 );

```

Fig. 9.10 Demonstrating class `Cylinder` (part 4 of 5).

```
   // use get functions to display the Cylinder
   cout << "X coordinate is " << cyl.getX()
        << "\nY coordinate is " << cyl.getY()
        << "\nRadius is " << cyl.getRadius()
        << "\nHeight is " << cyl.getHeight() << "\n\n";

   // use set functions to change the Cylinder's attributes
   cyl.setHeight( 10 );
   cyl.setRadius( 4.25 );
   cyl.setPoint( 2, 2 );
   cout << "The new location, radius, and height of cyl are:\n"
        << cyl << '\n';

   // display the Cylinder as a Point
   Point &pRef = cyl;   // pRef "thinks" it is a Point
   cout << "\nCylinder printed as a Point is: "
        << pRef << "\n\n";

   // display the Cylinder as a Circle
   Circle &circleRef = cyl;  // circleRef thinks it is a Circle
   cout << "Cylinder printed as a Circle is:\n" << circleRef
        << "\nArea: " << circleRef.area() << endl;

   return 0;
}
```

```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.00

Cylinder printed as a Point is: [2, 2]

Cylinder printed as a Circle is:
Center = [2, 2]; Radius = 4.25
Area: 56.74
```

Fig. 9.10 Demonstrating class `Cylinder` (part 5 of 5).

```
// Fig. 9.11: base1.h
// Definition of class Base1
#ifndef BASE1_H
#define BASE1_H

class Base1 {
public:
   Base1( int x ) { value = x; }
   int getData() const { return value; }
protected:      // accessible to derived classes
   int value;   // inherited by derived class
};

#endif
```

Fig. 9.11 Demonstrating multiple inheritance (part 1 of 6).

```
// Fig. 9.11: base2.h
// Definition of class Base2
#ifndef BASE2_H
#define BASE2_H

class Base2 {
public:
   Base2( char c ) { letter = c; }
   char getData() const { return letter; }
protected:        // accessible to derived classes
   char letter;   // inherited by derived class
};

#endif
```

Fig. 9.11 Demonstrating multiple inheritance (part 2 of 6).

```
// Fig. 9.11: derived.h
// Definition of class Derived which inherits
// multiple base classes (Base1 and Base2).
#ifndef DERIVED_H
#define DERIVED_H

#include "base1.h"
#include "base2.h"

// multiple inheritance
class Derived : public Base1, public Base2 {
   friend ostream &operator<<( ostream &, const Derived & );

public:
   Derived( int, char, double );
   double getReal() const;

private:
   double real;   // derived class's private data
};

#endif
```

Fig. 9.11 Demonstrating multiple inheritance (part 3 of 6).

```
// Fig. 9.11: derived.cpp
// Member function definitions for class Derived
#include <iostream.h>
#include "derived.h"

// Constructor for Derived calls constructors for
// class Base1 and class Base2.
// Use member initializers to call base-class constructors
Derived::Derived( int i, char c, double f )
   : Base1( i ), Base2( c ), real ( f ) { }

// Return the value of real
double Derived::getReal() const { return real; }

// Display all the data members of Derived
ostream &operator<<( ostream &output, const Derived &d )
{
   output << "    Integer: " << d.value
          << "\n  Character: " << d.letter
          << "\nReal number: " << d.real;

   return output;   // enables cascaded calls
}
```

Fig. 9.11 Demonstrating multiple inheritance (part 4 of 6).

```
// Fig. 9.11: fig09_11.cpp
// Driver for multiple inheritance example
#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

int main()
{
   Base1 b1( 10 ), *base1Ptr = 0;  // create Base1 object
   Base2 b2( 'Z' ), *base2Ptr = 0; // create Base2 object
   Derived d( 7, 'A', 3.5 );       // create Derived object

   // print data members of base class objects
   cout << "Object b1 contains integer " << b1.getData()
        << "\nObject b2 contains character " << b2.getData()
        << "\nObject d contains:\n" << d << "\n\n";

   // print data members of derived class object
   // scope resolution operator resolves getData ambiguity
   cout << "Data members of Derived can be"
        << " accessed individually:"
        << "\n    Integer: " << d.Base1::getData()
        << "\n  Character: " << d.Base2::getData()
        << "\nReal number: " << d.getReal() << "\n\n";

   cout << "Derived can be treated as an "
        << "object of either base class:\n";

   // treat Derived as a Base1 object
   base1Ptr = &d;
   cout << "base1Ptr->getData() yields "
        << base1Ptr->getData() << '\n';

   // treat Derived as a Base2 object
   base2Ptr = &d;
   cout << "base2Ptr->getData() yields "
        << base2Ptr->getData() << endl;

   return 0;
}
```

Fig. 9.11 Demonstrating multiple inheritance (part 5 of 6).

```
Object b1 contains integer 10
Object b2 contains character Z
Object d contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 9.11 Demonstrating multiple inheritance (part 6 of 6).