# W 9.1 - Inheritance

## Outline

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>9.2</td>
<td>Inheritance: Base Classes and Derived Classes</td>
</tr>
<tr>
<td>9.3</td>
<td>Protected Members</td>
</tr>
<tr>
<td>9.4</td>
<td>Casting Base-Class Pointers to Derived-Class Pointers</td>
</tr>
<tr>
<td>9.5</td>
<td>Using Member Functions</td>
</tr>
<tr>
<td>9.6</td>
<td>Overriding Base-Class Members in a Derived Class</td>
</tr>
<tr>
<td>9.7</td>
<td>Public, Protected and Private Inheritance</td>
</tr>
<tr>
<td>9.8</td>
<td>Direct Base Classes and Indirect Base Classes</td>
</tr>
<tr>
<td>9.9</td>
<td>Using Constructors and Destructors in Derived Classes</td>
</tr>
<tr>
<td>9.10</td>
<td>Implicit Derived-Class Object to Base-Class Object Conversion</td>
</tr>
<tr>
<td>9.11</td>
<td>Software Engineering with Inheritance</td>
</tr>
<tr>
<td>9.12</td>
<td>Composition vs. Inheritance</td>
</tr>
<tr>
<td>9.13</td>
<td>“Uses A” and “Knows A” Relationships</td>
</tr>
<tr>
<td>9.14</td>
<td>Case Study: Point, Circle, Cylinder</td>
</tr>
<tr>
<td>9.15</td>
<td>Multiple Inheritance</td>
</tr>
</tbody>
</table>
9.1 Introduction

• Inheritance
  – New classes created from existing classes
  – Absorb attributes and behaviors
  – Derived class
    • Class that inherits data members and member functions from a previously defined base class
  – Single inheritance
    • Class inherits from one base class
  – Multiple inheritance
    • Class inherits from multiple base classes
  – Types of inheritance
    • public: Derived objects are accessible by the base class objects
    • private: Derived objects are inaccessible by the base class
    • protected: Derived classes and friends can access protected members of the base class
9.1 Introduction

- Polymorphism
  - Write programs in a general fashion
  - Handle a wide variety of existing (and unspecified) related classes
9.2 Inheritance: Base and Derived Classes

• Base and derived classes
  – Often an object from a derived class (subclass) is also an object of a base class (superclass)
    • A rectangle is a derived class in reference to a quadrilateral and a base class in reference to a square

• Inheritance examples

<table>
<thead>
<tr>
<th>Base class</th>
<th>Derived classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>GraduateStudent</td>
</tr>
<tr>
<td></td>
<td>UndergraduateStudent</td>
</tr>
<tr>
<td>Shape</td>
<td>Circle</td>
</tr>
<tr>
<td></td>
<td>Triangle</td>
</tr>
<tr>
<td></td>
<td>Rectangle</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan</td>
</tr>
<tr>
<td></td>
<td>HomeImprovementLoan</td>
</tr>
<tr>
<td></td>
<td>MortgageLoan</td>
</tr>
<tr>
<td>Employee</td>
<td>FacultyMember</td>
</tr>
<tr>
<td></td>
<td>StaffMember</td>
</tr>
<tr>
<td>Account</td>
<td>CheckingAccount</td>
</tr>
<tr>
<td></td>
<td>SavingsAccount</td>
</tr>
</tbody>
</table>
9.2 Inheritance: Base and Derived Classes

• Implementation of **public** inheritance

```cpp
class CommissionWorker : public Employee {
    ...
};
```

– Class `CommissionWorker` inherits from class `Employee`

– `friend` functions not inherited

– `private` members of base class not accessible from derived class
9.3 protected Members

• **protected** access
  - Intermediate level of protection between **public** and **private** inheritance
  - Derived-class members can refer to **public** and **protected** members of the base class simply by using the member names
  - Note that **protected** data “breaks” encapsulation
9.4 Casting Base-Class Pointers to Derived Class Pointers

• Derived classes relationships to base classes
  – Objects of a derived class can be treated as objects of the base class
    • Reverse not true — base class objects cannot be derived-class objects

• Downcasting a pointer
  – Use an explicit cast to convert a base-class pointer to a derived-class pointer
  – If pointer is going to be dereferenced, the type of the pointer must match the type of object to which the pointer points
  – Format:

    derivedPtr = static_cast< DerivedClass * > basePtr;
9.4 Casting Base-Class Pointers to Derived-Class Pointers

• The following example:
  – Demonstrates the casting of base class pointers to derived class pointers
  – Class Circle is derived from class Point
  – A pointer of type Point is used to reference a Circle object, and a pointer to type Circle is used to reference a Point object
// Fig. 9.4: point.h
// Definition of class Point
#ifndef POINT_H
#define POINT_H

#include <iostream>

using std::ostream;

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: point.cpp
// Member functions for class Point
#include <iostream>
#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;
}
1. Circle class definition

Class Circle publicly inherits from class Point, so it will have class Point's public and protected member functions and data.
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;

};

#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 ); }

Circle inherits from Point, and has Point's data members (which are set by calling Point's constructor).
89 // Get radius of Circle
90 double Circle::getRadius() const { return radius; }
91
92 // Calculate area of Circle
93 double Circle::area() const
94    { return 3.14159 * radius * radius; }
95
96 // Output a Circle in the form:
97 // Center = [x, y]; Radius = #.##
98 ostream &operator<<( ostream &output, const Circle &c )
99 {
100   output << "Center = " << static_cast< Point >( c )
101     << "; Radius = "
102     << setiosflags( ios::fixed | ios::showpoint )
103     << setprecision( 2 ) << c.radius;
104
105   return output; // enables cascaded calls
106 }
107// Fig. 9.4: fig09_04.cpp
108// Casting base-class pointers to derived-class pointers
109#include <iostream>
110
111using std::cout;
112using std::endl;
113
114#include <iomanip>
115
116#include "point.h"
117#include "circle.h"
118
119int main()
120{
121   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
pointPtr = &c; // assign address of Circle to pointPtr

cout << "\nCircle c (via *pointPtr): "
    << *pointPtr << '\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): "
    << *pointPtr << '\n';

// Treat a Circle as a Circle (with some casting)
circlePtr = static_cast< Circle * >( pointPtr );

cout << "\nCircle c (via *circlePtr): "
    << *circlePtr << '\n';

// Assign derived-class pointer ( &c ) to base class pointer pointPtr.
The base class pointer only "sees" the base-class part of the object it points to.

return 0;

// Assign derived-class pointer ( &c ) to base class pointer pointPtr.
The base class pointer only "sees" the base-class part of the object it points to.

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

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

// 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): "
    << *circlePtr << '\n';

// Treat a Circle as a Circle (with some casting)
circlePtr = static_cast< Circle * >( pointPtr );

cout << "\nCircle c (via *circlePtr): "
    << *circlePtr << '\n';

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

Area of object circlePtr points to: 22.90

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

• Derived class member functions
  – Cannot directly access private members of their base class
    • Maintains encapsulation
  – Hiding private members is a huge help in testing, debugging and correctly modifying systems
9.6 Overriding Base-Class Members in a Derived Class

• To override a base-class member function
  – In the derived class, supply a new version of that function with the same signature
    • same function name, different definition
  – When the function is then mentioned by name in the derived class, the derived version is automatically called
  – The scope-resolution operator may be used to access the base class version from the derived class
// 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: employ.cpp
// Member function definitions for class Employee
#include <iostream>

using std::cout;

#include <cstring>
#include <cassert>
#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: 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:

HourlyWorker inherits from Employee.
HourlyWorker will override the print function.
double wage; // wage per hour
double hours; // hours worked for week

};
#endif
// Fig. 9.5: hourly.cpp
// Member function definitions for class HourlyWorker
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#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; }
The `print` function is overridden in `HourlyWorker`. However, the new function still can call the original `print` function using `::`
## 9.7 public, private, and protected Inheritance

<table>
<thead>
<tr>
<th>Base class member access specifier</th>
<th>Type of inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public inheritance</td>
</tr>
<tr>
<td><strong>Public</strong></td>
<td>public in derived class. Can be accessed directly by any non-static member functions, friend functions and non-member functions.</td>
</tr>
<tr>
<td><strong>Protected</strong></td>
<td>protected in derived class. Can be accessed directly by all non-static member functions and friend functions.</td>
</tr>
<tr>
<td><strong>Private</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
9.8 Direct and Indirect Base Classes

- Direct base class
  - Explicitly listed derived class’s header with the colon (:) notation when that derived class is declared
    ```cpp
    class HourlyWorker : public Employee
    ```
  - **Employee** is a direct base class of **HourlyWorker**

- Indirect base class
  - Not listed in derived class’s header
  - Inherited from two or more levels up the class hierarchy
    ```cpp
    class MinuteWorker : public HourlyWorker
    ```
  - **Employee** is an indirect base class of **MinuteWorker**
9.9 Using Constructors and Destructors in Derived Classes

• Base class initializer
  – Uses member-initializer syntax
  – Can be provided in the derived class constructor to call the base-class constructor explicitly
    • Otherwise base class’s default constructor called implicitly
  – Base-class constructors and base-class assignment operators are not inherited by derived classes
    • Derived-class constructors and assignment operators, however, can call base-class constructors and assignment operators
9.9 Using Constructors and Destructors in Derived Classes

- A derived-class constructor
  - Calls the constructor for its base class first to initialize its base-class members
  - If the derived-class constructor is omitted, its default constructor calls the base-class’ default constructor

- Destructors are called in the reverse order of constructor calls
  - So a derived-class destructor is called before its base-class destructor

- Destructor of Class Employee frees up the dynamically assigned arrays [ ] firstname and [ ] lastname, avoiding memory leaks (lines 48, 49).
// 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:
    int x, y; // x and y coordinates of Point
};

#endif

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

using std::cout;
using std::endl;

#include "point2.h"

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

    cout << "Point constructor: [" << x << ", " << y << "]" << endl;
}
1.1 Function definitions
----------------------
1. Load header

1.1 Circle Definition

Circle inherits from Point.
58 // Fig. 9.7: circle2.cpp
59 // Member function definitions for class Circle
60 #include <iostream>
61
62 using std::cout;
63 using std::endl;
64
65 #include "circle2.h"
66
67 // Constructor for Circle calls constructor for Point
68 Circle::Circle( double r, int a, int b )
69    : Point( a, b ) // call base-class constructor
70 {
71    radius = r; // should validate
72    cout << "Circle constructor: radius is "
73         << radius << " [" << x << ", " << y << "]' " << endl;
74 }
75
76 // Destructor for class Circle
77 Circle::~Circle()
78 {
79    cout << "Circle destructor: radius is "
80         << radius << " [" << x << ", " << y << "]' " << endl;
81 }
```cpp
#include <iostream>

using std::cout;
using std::endl;

#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;
}
```

Object created inside a block destroyed once it leaves scope.

Remember that the `Point` constructor is called for `Circle` objects before the `Circle` constructor (inside to out).

- Point constructor: [11, 22]
- Circle constructor: radius is 4.5 [72, 29]
- Point constructor: [72, 29]
- Circle constructor: radius is 10 [5, 5]
- Point destructor: [5, 5]
- Circle destructor: radius is 4.5 [72, 29]
- Point destructor: [72, 29]
Program Output

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]
9.10 Implicit Derived-Class Object to Base-Class Object Conversion

- Assignment of derived and base classes
  - Derived-class type and base-class type are different
  - Derived-class object can be treated as a base-class object
    - Derived class has members corresponding to all of the base class’s members
    - Derived-class has more members than the base-class object
    - Base-class can be assigned a derived-class
  - Base-class object cannot be treated as a derived-class object
    - Would leave additional derived class members undefined
    - Derived-class cannot be assigned a base-class
    - Assignment operator can be overloaded to allow such an assignment
9.10 Implicit Derived-Class Object to Base-Class Object Conversion

• Mixing base and derived class pointers and objects
  – Referring to a base-class object with a base-class pointer
    • Allowed (straightforward)
  – Referring to a derived-class object with a derived-class pointer
    • Allowed (straightforward)
  – Referring to a derived-class object with a base-class pointer
    • Possible syntax error
    • Code can only refer to base-class members, or syntax error
  – Referring to a base-class object with a derived-class pointer
    • Syntax error
    • The derived-class pointer must first be cast to a base-class pointer

• Need way to resolve base-class Vs derived-class routines using base-class pointers (Virtual functions)
9.11 Software Engineering With Inheritance

• Classes are often closely related
  – “Factor out” common attributes and behaviors and place these in a base class
  – Use inheritance to form derived classes

• Modifications to a base class
  – Derived classes do not change as long as the public and protected interfaces are the same
  – Derived classes may need to be recompiled

• Use Inheritance sparingly, often times complexity is introduced needlessly. Can make for bad engineering thus hard to understand systems.
9.12 Composition vs. Inheritance

• “Is a” relationships
  – Inheritance
    • Relationship in which a class is derived from another class

• “Has a” relationships
  – Composition
    • Relationship in which a class contains other classes as members

• **Has a**, is a composition. **Is a kind of**, is inheritance.

• Interchangeable! Careful.
9.13 “Uses A” And “Knows A” Relationships

- **“Uses a”**
  - One object issues a function call to a member function of another object. Limited!

- **“Knows a”**
  - One object is aware of another
    - Contains a pointer or handle to another object
    - Has access to all public stuff.
  - Also called an association
9.14 Case Study: Point, Circle, Cylinder

- Point, circle, cylinder hierarchy
  - **Point** class is base class
  - **Circle** class is derived from **Point** class
  - **Cylinder** class is derived from **Circle** class
1 // Fig. 9.8: point2.h
2 // Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 #include <iostream>
7
8 using std::ostream;
9
10 class Point {
11    friend ostream &operator<<( ostream &, const Point & );
12 public:
13    Point( int = 0, int = 0 );      // default constructor
14    void setPoint( int, int );      // set coordinates
15    int getX() const { return x; }  // get x coordinate
16    int getY() const { return y; }  // get y coordinate
17 protected:        // accessible to derived classes
18    int x, y;      // coordinates of the point
19  };
20
21 #endif
22 // Fig. 9.8: point2.cpp
23 // Member functions for class Point
24 #include "point2.h"
25
26 // Constructor for class Point
27 Point::Point( int a, int b ) { setPoint( a, b ); }  
28
29 // Set the x and y coordinates
30 void Point::setPoint( int a, int b )
31 {
32    x = a;
```cpp
33    y = b;
34 }
35
36    // Output the Point
37    ostream &operator<<( ostream &output, const Point &p )
38    {
39        output << '[' << p.x << ', ' << p.y << ']';
40    
41    return output;          // enables cascading
42 }
```
circle definition

1.1 Function definitions

```cpp
#include <iostream>

using std::ostream;

#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
```

Circle data members are protected to be made accessible by Cylinder.
// Constructor for Circle calls constructor for Point
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; }

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

// Output a circle in the form:
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
1 // Fig. 9.10: cylindr2.h
2 // Definition of class Cylinder
3 #ifndef CYLINDR2_H
4 #define CYLINDR2_H
5
6 #include <iostream>
7
8 using std::ostream;
9
10 #include "circle2.h"
11
12 class Cylinder : public Circle {
13    friend ostream &operator<<( ostream &, const Cylinder & );
14
15 public:
16    // default constructor
17    Cylinder( double h = 0.0, double r = 0.0, 
18               int x = 0, int y = 0 );
19
20    void setHeight( double );   // set height
21    double getHeight() const;   // return height
22    double area() const;        // calculate and return area
23    double volume() const;      // calculate and return volume
24
25 protected:
26    double height;              // height of the Cylinder
27
28 #endif
1.1 Function definitions

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 )
{
```cpp
61    output << static_cast< Circle >( c )
62           << "; Height = " << c.height;
63
64    return output; // enables cascaded calls
65 }
66 // Fig. 9.10: fig09_10.cpp
67 // Driver for class Cylinder
68 #include <iostream>
69
70 using std::cout;
71 using std::endl;
72
73 #include "point2.h"
74 #include "circle2.h"
75 #include "cylindr2.h"
76
77 int main()
78 {
79    // create Cylinder object
80    Cylinder cyl( 5.7, 2.5, 12, 23 );
81
82    // use get functions to display the Cylinder
83    cout << "X coordinate is " << cyl.getX()
84           << "\nY coordinate is " << cyl.getY()
85           << "\nRadius is " << cyl.getRadius()
86           << "\nHeight is " << cyl.getHeight() << "\n\n";
87
88    // use set functions to change the Cylinder's attributes
89    cyl.setHeight( 10 );
90    cyl.setRadius( 4.25 );
91    cyl.setPoint( 2, 2 );
```
cout << "The new location, radius, and height of cyl are:\n"
    << cyl << '\n';

cout << "The area of cyl is:\n"
    << cyl.area() << '\n';

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

// display the Cylinder as a Circle
Circle &circleRef = cyl; // circleRef "thinks" cyl 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
The area of cyl is:
380.53
Cylinder printed as a Point is: [2, 2]
Cylinder printed as a Circle is:
Center = [2, 2]; Radius = 4.25
Area: 56.74
9.15 Multiple Inheritance

- Multiple Inheritance
  - Derived-class inherits from multiple base-classes
  - Encourages software reuse, but can create ambiguities
```cpp
1 // Fig. 9.11: base1.h
2 // Definition of class Base1
3 #ifndef BASE1_H
4 #define BASE1_H
5
6 class Base1 {
7   public:
8      Base1( int x ) { value = x; }
9      int getData() const { return value; }
10 protected:  // accessible to derived classes
11      int value;  // inherited by derived class
12   }
13}
14
15 #endif
16 // Fig. 9.11: base2.h
17 // Definition of class Base2
18 #ifndef BASE2_H
19 #define BASE2_H
20
21 class Base2 {
22   public:
23      Base2( char c ) { letter = c; }
24      char getData() const { return letter; }
25 protected:  // accessible to derived classes
26      char letter;  // inherited by derived class
27   }
28
29 #endif
```
// Fig. 9.11: derived.h
// Definition of class Derived which inherits
// multiple base classes (Base1 and Base2).
#ifndef DERIVED_H
#define DERIVED_H

#include <iostream>

using std::ostream;

#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
55 // Fig. 9.11: derived.cpp
56 // Member function definitions for class Derived
57 #include "derived.h"
58
59 // Constructor for Derived calls constructors for
60 // class Base1 and class Base2.
61 // Use member initializers to call base-class constructors
62 Derived::Derived( int i, char c, double f )
63    : Base1( i ), Base2( c ), real ( f ) { }
64
65 // Return the value of real
66 double Derived::getReal() const { return real; }  
67
68 // Display all the data members of Derived
69 ostream &operator<<( ostream &output, const Derived &d )
70 {
71    output << "    Integer: " << d.value
72        << "\n  Character: " << d.letter
73        << "\nReal number: " << d.real;
74
75    return output;  // enables cascaded calls
76 }
77
78 // Fig. 9.11: fig09_11.cpp
79 // Driver for multiple inheritance example
80 #include <iostream>
81 using std::cout;
82 using std::endl;
83
84 #include "base1.h"
85 #include "base2.h"
```cpp
#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:
         Integer: " << d.Base1::getData()
         << "\nCharacter: " << 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;
```

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:

Treat d as a **Base1** object.

Treat d as a **Base2** object.

base1Ptr->getData() yields 7
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

cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
return 0;
Graded Exercises

• Read the summary of Ch 9 (pp 618..622)
• Do Self-Review exercises Ch 9 Deitel & Deitel
• Do following Exercises
  – 9.2 (make a diagram like on Fig 9.2, put in folder),
  – 9.12 (on paper & put in folder)