OBJECT ORIENTED PROGRAMMING USING C++

Chapter 19 - C++ Inheritance

Outline

19.1 Introduction
19.2 Inheritance: Base Classes and Derived Classes
19.3 Protected Members
19.4 Casting Base-Class Pointers to Derived-Class Pointers
19.5 Using Member Functions
19.6 Overriding Base-Class Members in a Derived Class
19.7 Public, Protected and Private Inheritance
19.8 Direct Base Classes and Indirect Base Classes
19.9 Using Constructors and Destructors in Derived Classes
19.10 Implicit Derived-Class Object to Base-Class Object Conversion
19.11 Software Engineering with Inheritance
19.12 Composition vs. Inheritance
19.13 *Uses A* and *Knows A* Relationships
19.14 Case Study: Point, Circle, Cylinder

◀ ▶

Objectives

- In this chapter, you will learn:
  - To be able to create new classes by inheriting from existing classes.
  - To understand how inheritance promotes software reusability.
  - To understand the notions of base classes and derived classes.

19.1 Introduction

- Inheritance
  - New classes created from existing classes
  - Absorb attributes and behaviors.
- Polymorphism
  - Write programs in a general fashion
  - Handle a wide variety of existing (and unspecified) related classes
- Derived class
  - Class that inherits data members and member functions from a previously defined base class

◀ ▶

19.1 Introduction

- Inheritance
  - Single Inheritance
    - Class inherits from one base class
  - Multiple Inheritance
    - Class inherits from multiple base classes
  - Three types of inheritance:
    - `public`: Derived objects are accessible by the base class objects (focus of this chapter)
    - `private`: Derived objects are inaccessible by the base class
    - `protected`: Derived classes and `friends` can access protected members of the base class

19.2 Base and Derived Classes

- Often an object from a derived class (subclass) “is an” object of a base class (superclass)

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

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

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

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

Fig. 19.1 Some simple inheritance examples.

19.2 Base and Derived Classes

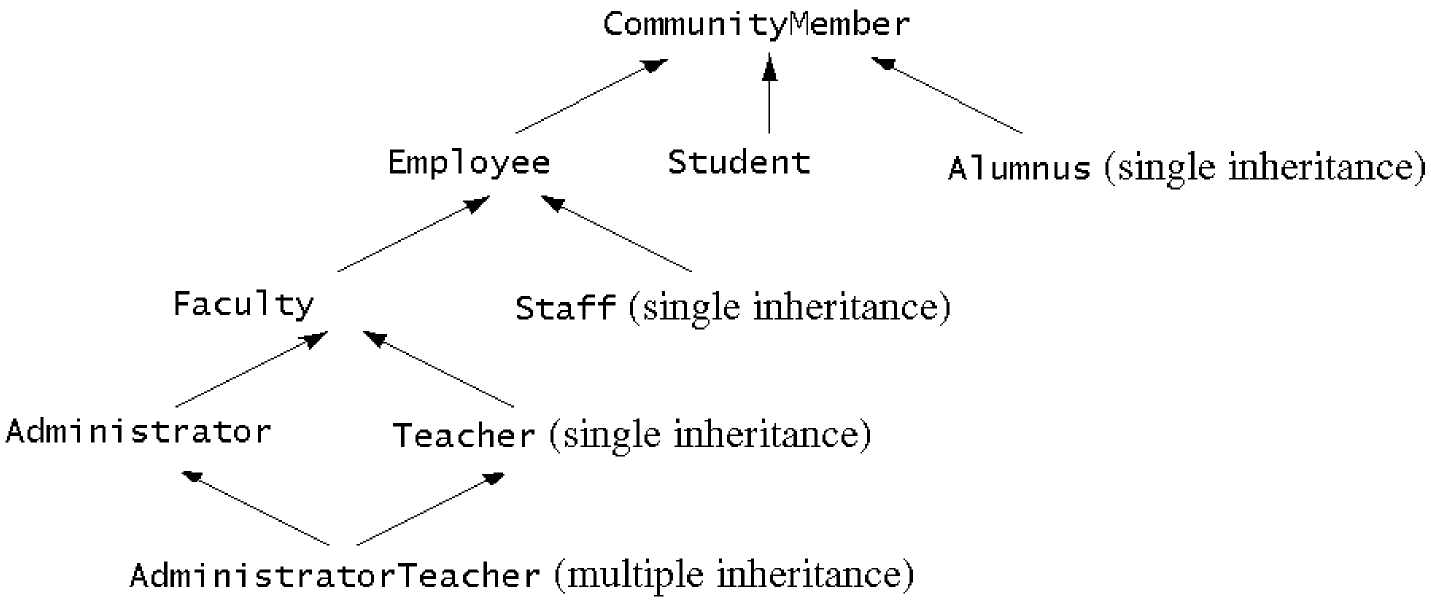

CommunityMember
Employee
Student
Alumnus (single inheritance)
Faculty
Staff (single inheritance)
Administrator
Teacher (single inheritance)
AdministratorTeacher (multiple inheritance)

Fig. 19.2 An inheritance hierarchy for university community members.

19.2 Base and Derived Classes

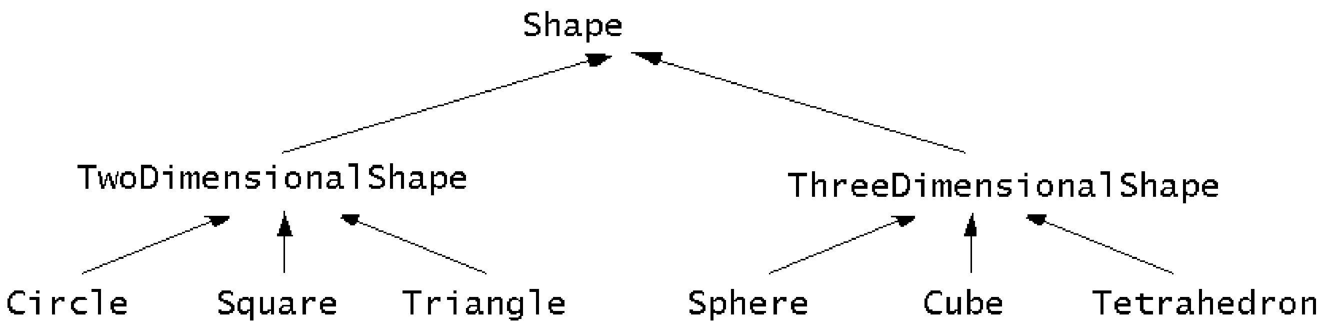


Fig. 19.3 A portion of a Shape class hierarchy.

19.2 Base and Derived Classes

- Implementation of `public` inheritance

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

Class `CommissionWorker` inherits from class `Employee`

- `friend` functions not inherited
- `private` members of base class not accessible from derived class

◀ ▶

19.3 Protected Members

- `protected` inheritance
  - 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

19.4 Casting Base Class Pointers to Derived Class Pointers

- Object of a derived class
  - Can be treated as an object of the base class
  - Reverse not true - base class objects not a derived-class object
- Downcasting a pointer
  - Use an explicit cast to convert a base-class pointer to a derived-class pointer
  - Be sure that the type of the pointer matches the type of object to which the pointer points

```
derivedPtr = static_cast< DerivedClass * > basePtr;
```

19.4 Casting Base-Class Pointers to Derived-Class Pointers

- Example
  - `Circle class` derived from the `Point` base `class`
  - We use pointer of type `Point` to reference a `Circle` object, and vice-versa

```
// Fig. 19.4: 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
}; // end class Point

#endif
```

Outline

point.h

```
// Fig. 19.4: 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;
   y = b;
} // end function setPoint

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

   return output;   // enables cascaded calls
} // end operator<< function
```

Outline

point.cpp

```
// Fig. 19.4: circle.h
// Definition of class Circle
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

using std::ostream;

#include <iomanip>

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

#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;
}; // end class Circle

#endif
```

circle.h

```
// Fig. 19.4: 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
} // end operator<< function
```

Outline

circle.cpp

```cpp
// Fig. 19.4: fig19_04.cpp
// Casting base-class pointers to derived-class pointers
#include <iostream>

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

#include <iomanip>

#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';

```

Outline

fig19_04.cpp (1 of 2)

fig19_04.cpp (2 of 2)

```cpp
   // Treat a Circle as a Circle (with some casting)
   // 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;
} // end function main
```

```
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
```

19.5 Using Member Functions

- Derived class
  - Cannot directly access `private` members of its base class
  - Hiding `private` members is a huge help in testing, debugging and correctly modifying systems

◀ ▶

19.6 Overriding Base-Class Members in a Derived Class

- To override a base-class member function
  - In derived class, supply new version of that function
    - Same function name, different definition
  - The scope-resolution operator may be used to access the base class version from the derived class

Outline

employ.h

```
// Fig. 19.5: 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
}; // end class Employee

#endif
```

employ.cpp (1 of 2)

```
// Fig. 19.5: employ.cpp
// Member function definitions for class Employee
#include <iostream>

using std::cout;

#include <cstring>
#include <cassert>
#include "employ.h"

```

employ.cpp (2 of 2)

```
// 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 );
} // end Employee constructor

// 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
} // end Employee destructor
```

Outline

hourly.h

```cpp
// Fig. 19.5: 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
}; // end class HourlyWorker

#endif
```

hourly.cpp (1 of 2)

```cpp
// Fig. 19.5: hourly.cpp
// Member function definitions for class HourlyWorker
#include <iostream>

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

#include <iomanip>

```

hourly.cpp (2 of 2)

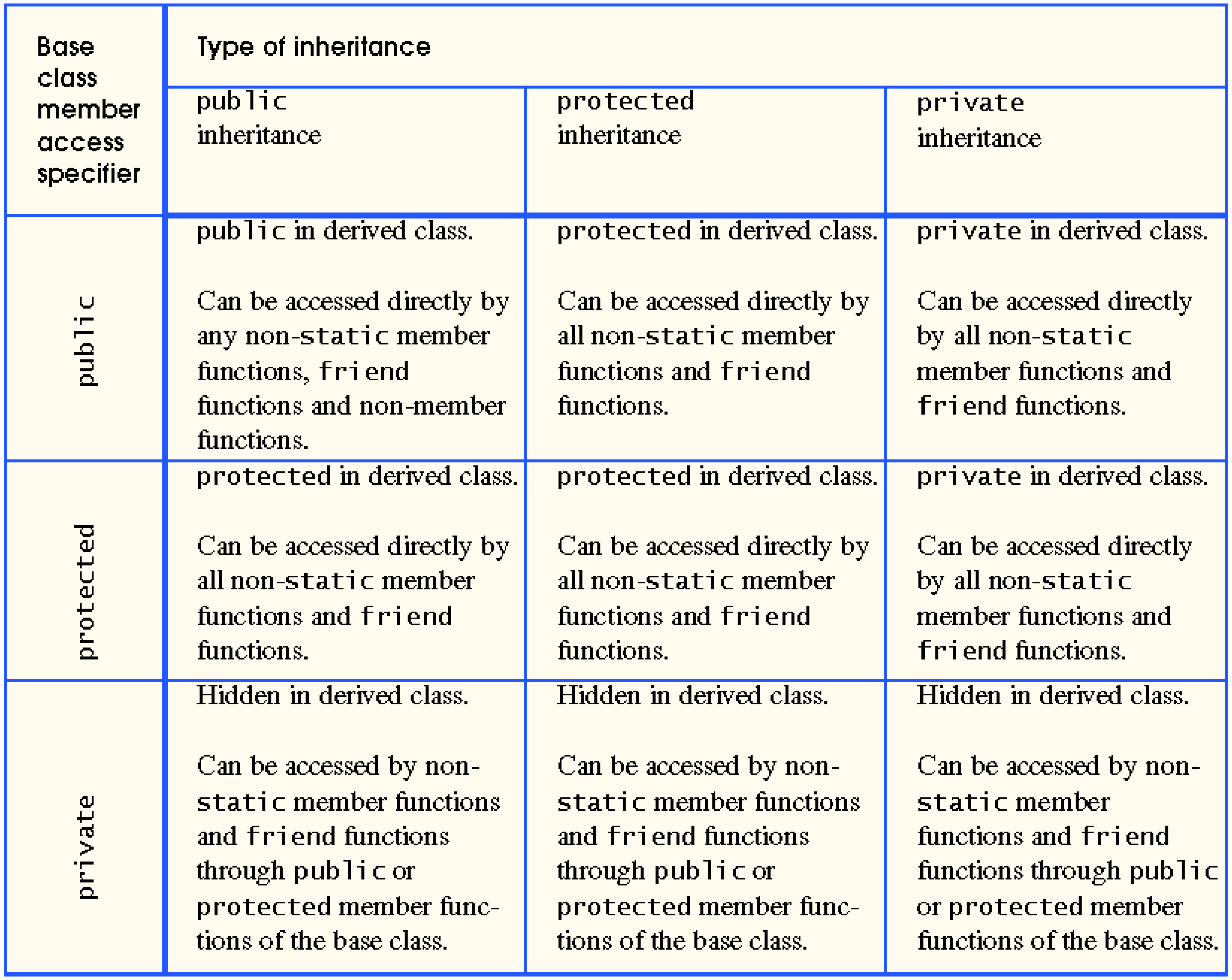
```
using std::ios;
using std::setiosflags;
using std::setprecision;

#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
} // end HourlyWorker constructor

// 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;
} // end function print
```

```
// Fig. 19.5: fig19_05.cpp
// Overriding a base-class member function in a
// derived class.
#include "hourly.h"

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

```
HourlyWorker::print() is executing

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

fig19_05.cpp

19.7 Public, Private, and Protected Inheritance



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. 19.6 Summary of base-class member accessibility in a derived class.

19.8 Direct and Indirect Base Classes

- Direct base class
  - Explicitly listed derived class' header with the colon (**`:`**) notation when that derived class is declared.
  - **`class HourlyWorker : public Employee`**
    - **`Employee`** is a direct base class of **`HourlyWorker`**
- Indirect base class
  - Inherited from two or more levels up the class hierarchy
  - **`class MinuteWorker : public HourlyWorker`**
    - **`Employee`** is an indirect base class of **`MinuteWorker`**

19.9 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' default constructor called implicitly
  - Base-class constructors and base-class assignment operators are not inherited by derived classes
    - However, derived-class constructors and assignment operators can call still them

19.9 Using Constructors and Destructors in Derived Classes

- 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.
  - Derived-class destructor is called before its base-class destructor

◀ ▶

```cpp
// Fig. 19.7: 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
}; // end class Point

#endif
```

point2.h

point2.cpp

```
// Fig. 19.7: 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;
} // end Point constructor

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

```
// Fig. 19.7: 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;
}; // end class Circle

#endif
```

Outline

circle2.h

```cpp
// Fig. 19.7: circle2.cpp
// Member function definitions for class Circle
#include <iostream>

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

#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;
} // end Circle constructor

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

Outline

circle2.cpp

```cpp
// Fig. 19.7: fig19_07.cpp
// Demonstrate when base-class and derived-class
// constructors and destructors are called.
#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 );
   } // end block

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

Outline

fig19_07.cpp (1 of 2)

fig19_07.cpp (2 of 2)

```
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]
```

19.10 Implicit Derived-Class Object to Base-Class Object Conversion

- `baseClassObject = derivedClassObject;`
  - This will work
    - Remember, the derived class object has more members than the base class object
  - Extra data is not given to the base class
- `derivedClassObject = baseClassObject;`
  - May not work properly
    - Unless an assignment operator is overloaded in the derived class, data members exclusive to the derived class will be unassigned
  - Base class has less data members than the derived class
    - Some data members missing in the derived class object

19.10 Implicit Derived-Class Object to Base-Class Object Conversion

- Four ways to mix base and derived class pointers and objects:
  - Referring to a base-class object with a base-class pointer
    - Allowed
  - Referring to a derived-class object with a derived-class pointer
    - Allowed
  - 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

◀ ▶

19.11 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

19.12 Composition vs. Inheritance

- "is a" relationship
  - Inheritance
- "has a" relationship
  - Composition - class has an object from another class as a data member

```
Employee “is a” BirthDate;   //Wrong!
Employee “has a” BirthDate;//Composition
```

19.13 *Uses A* And *Knows A* Relationships

- “uses a” relationship
  - One object issues a function call to a member function of another object
- “knows a” relationship
  - One object is aware of another
    - Contains a pointer handle or reference handle to another object
  - Also called an association

◀ ▶

19.14 Case Study: Point, Circle, Cylinder

- Define class `Point`
  - Derive `Circle`
    - Derive `Cylinder`

```
// Fig. 19.8: point2.h
// Definition of class Point
#ifndef POINT2_H
#define POINT2_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 to derived classes
   int x, y;      // coordinates of the point
}; // end class Point

#endif
```

Outline

point2.h

```cpp
// Fig. 19.8: point2.cpp
// Member functions for class Point
#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;
} // end function setPoint

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

   return output;          // enables cascading
} // end operator<< function
```

Outline

point2.cpp

```cpp
// Fig. 19.8: fig19_08.cpp
// Driver for class Point
#include <iostream>

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

#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;
} // end function main
```

```
X coordinate is 72
Y coordinate is 115

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

Outline

fig19_08.cpp

```cpp
// Fig. 19.9: circle2.h
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#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
}; // end class Circle

#endif
```

circle2.h

```cpp
// Fig. 19.9: circle2.cpp
// Member function definitions for class Circle
#include <iomanip>

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

#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; }

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

```

Outline

circle2.cpp (1 of 2)

```cpp
// 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
} // end operator<< function
```

Outline

circle2.cpp (2 of 2)

```cpp
// Fig. 19.9: fig19_09.cpp
// Driver for class Circle
#include <iostream>

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

#include "point2.h"
#include "circle2.h"

```

fig19_09.cpp (1 of 2)

```cpp
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;
} // end function main
```

```
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]
```

Outline

fig19_09.cpp (2 of 2)

```cpp
// Fig. 19.10: cylindr2.h
// Definition of class Cylinder
#ifndef CYLINDR2_H
#define CYLINDR2_H

#include <iostream>

using std::ostream;

#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
}; // end class Cylinder

#endif
```

Outline

cylindr2.h

```
// Fig. 19.10: cylindr2.cpp
// Member and friend function definitions
// for class Cylinder.
#include "cylindr2.h"

// 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;
} // end function area

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

```

Outline

cylindr2.cpp (1 of 2)

Outline

cylindr2.cpp (2 of 2)

```
// Output Cylinder dimensions
ostream &operator<<( ostream &output, const Cylinder &c )
{
   output << static_cast< Circle >( c )
          << "; Height = " << c.height;

   return output;   // enables cascaded calls
} // end operator<< function
```

fig19_10.cpp (1 of 3)

```
// Fig. 19.10: fig19_10.cpp
// Driver for class Cylinder
#include <iostream>

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

#include "point2.h"
#include "circle2.h"
#include "cylindr2.h"

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

```

```cpp
   // 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';

   cout << "The area of cyl is:\n"
        << cyl.area() << '\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;
} // end function main
```

fig19_10.cpp (2 of 3)

```
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
```

Outline

fig19_10.cpp (3 of 3)