W 9.1 - Inheritance

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

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

• Implementation of \texttt{public} inheritance

  
  \begin{verbatim}
  class CommissionWorker : public Employee {
    ...
  }
  \end{verbatim}

  – Class \texttt{CommissionWorker} inherits from class \texttt{Employee}
  – \texttt{friend} functions not inherited
  – \texttt{private} members of base class not accessible from derived class

9.3 \texttt{protected} Members

• \texttt{protected} access

  – Intermediate level of protection between \texttt{public} and \texttt{private} inheritance
  – Derived-class members can refer to \texttt{public} and \texttt{protected} members of the base class simply by using the member names
  – Note that \texttt{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
1. **Point class definition**

---

1. **Load header**

1.1 **Function definitions**

---

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

1. Load header

1.1 Function Definitions

Circle inherits from Point. and has Point's data members (which are set by calling Point's constructor).

...
Assign `pointPtr` to a `Point` object. It has no derived-class information. When it is cast to a `Circle *`, `circlePtr` is really assigned to a base-class object with no derived-class information. This is dangerous.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
<td><code>Circle *circlePtr = 0, c(2.7, 120, 89);</code></td>
</tr>
<tr>
<td>124</td>
<td><code>cout &lt;&lt; &quot;Point p: &quot; &lt;&lt; p &lt;&lt; &quot;\nCircle c: &quot; &lt;&lt; c &lt;&lt; \&quot;\n\&quot;;</code></td>
</tr>
<tr>
<td>126</td>
<td><code>// Treat a Circle as a Point</code></td>
</tr>
<tr>
<td>127</td>
<td><code>pointPtr = &amp;c;</code></td>
</tr>
<tr>
<td>128</td>
<td><code>cout &lt;&lt; \&quot;\\nCircle c (via *pointPtr): \&quot;</code></td>
</tr>
<tr>
<td>129</td>
<td><code>&lt;&lt; *pointPtr &lt;&lt; \&quot;\n\&quot;</code></td>
</tr>
<tr>
<td>132</td>
<td><code>// Treat a Circle as a Circle (with some casting)</code></td>
</tr>
<tr>
<td>133</td>
<td><code>circlePtr = static_cast&lt; Circle * &gt;( pointPtr );</code></td>
</tr>
<tr>
<td>135</td>
<td><code>cout &lt;&lt; \&quot;\\nCircle c (via *circlePtr): \&quot;</code></td>
</tr>
<tr>
<td>136</td>
<td><code>&lt;&lt; circlePtr-&gt;area() &lt;&lt; endl;</code></td>
</tr>
</tbody>
</table>

Point p: [30, 50]
Circle c: Center = [120, 89]; Radius = 2.70
Circle c (via *pointPtr): [120, 89]
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
1. Employee class definition

```
#include "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
};
```

```
1 // Fig. 9.5: employ.h
2 // Definition of class Employee
3 #ifndef EMPLOY_H
4 #define EMPLOY_H
5
6 class Employee {
7    public:
8        Employee(const char *, const char *); // constructor
9        void print() const; // output first and last name
10    ~Employee(); // destructor
11    private:
12        char *firstName; // dynamically allocated string
13        char *lastName; // dynamically allocated string
14    };
15
16 #endif
```

18 // 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
}
```

52 // Fig. 9.5: hourly.h
53 // Definition of class HourlyWorker
54 #ifndef HOURLY_H
55 #define HOURLY_H
56
57 #include "employ.h"
58
59 class HourlyWorker : public Employee {
60    public:
61        HourlyWorker(const char *, const char *, double, double);
62        double getPay() const; // calculate and return salary
63        void print() const; // overridden base-class print
64    private:
65
66        HourlyWorker inherits from Employee.
67
68        HourlyWorker will override the print function.
```

Outline

1. Employee class definition

---------------------
1.1 Function definitions

1. Load header
Outline

1. Load header

1.1 Function definitions

Program Output

The `print` function is overridden in `HourlyWorker`. However, the new function still can call the original `print` function using `::`.

Bob Smith is an hourly worker with pay of $400.00
9.7  public, private, and protected Inheritance

<table>
<thead>
<tr>
<th>Base class member access specifier</th>
<th>public inheritance</th>
<th>protected inheritance</th>
<th>private inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Public</strong></td>
<td>public in derived class. Can be accessed directly by any non-static member functions and friend functions.</td>
<td>protected in derived class. Can be accessed directly by all non-static member functions and friend functions.</td>
<td>private in derived class. Can be accessed directly by all non-static member functions and friend 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>
<td>protected in derived class. Can be accessed directly by all non-static member functions and friend functions.</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>
<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>
<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
    
    ```
    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
    
    ```
    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

• 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).
1 // Fig. 9.7: point2.h
2 // Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 class Point{
7   public:
8       Point(int = 0, int = 0); // default constructor
9       ~Point(); // destructor
10 protected: // accessible by derived classes
11       int x, y; // x and y coordinates of Point
12   }
13
14 #endif
15 // Fig. 9.7: point2.cpp
16 // Member function definitions for class Point
17 #include <iostream>
18
19 using std::cout;
20 using std::endl;
21
22 #include "point2.h"
23
24 // Constructor for class Point
25 Point::Point(int a, int b)
26 {
27     x = a;
28     y = b;
29     cout << "Point constructor: [" << x << " , " << y << "]" << endl;
30 }
31
32 // Destructor for class Point
33 Point::~Point()
34 {
35     cout << "Point destructor: [" << x << " , " << y << "]" << endl;
36 }
37
38 // Definition of class Circle
39 #ifndef CIRCLE2_H
40 #define CIRCLE2_H
41
42 #include "point2.h"
43
44 class Circle : public Point {
45   public:
46       // default constructor
47       Circle(double r = 0.0, int x = 0, int y = 0);
48       ~Circle();
49   private:
50       double radius;
51   }
52 #endif
53
54 // Fig. 9.7: circle2.h
55 // Definition of class Circle
56 #ifndef CIRCLE2_H
57 #define CIRCLE2_H
58 #include "point2.h"
59
60 class Circle : public Point {
61   public:
62       // default constructor
63       Circle(double r = 0.0, int x = 0, int y = 0);
64       ~Circle();
65   private:
66       double radius;
67   }
68 #endif
69
70 #include "point2.h"
71
72 class Circle : public Point {
73   public:
74       // default constructor
75       Circle(double r = 0.0, int x = 0, int y = 0);
76       ~Circle();
77   private:
78       double radius;
79   }
80 #endif
81
82 // Fig. 9.7: circle2.cpp
83 // Member function definitions for class Circle
84 #include <iostream>
85
86 using std::cout;
87 using std::endl;
88
89 #include "circle2.h"
90
91 // Constructor for class Circle
92 Circle::Circle(double r, int x, int y)
93 {
94     //...
// Fig. 9.7: 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
{   // should validate
    radius = r;  // should validate
    cout << "Circle constructor: radius is "
        << radius << " \[" << x <<", " << y << "]' " << endl;
}

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

// Figure 9.7: fig09_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
    {   // should validate
        Point p( 11, 22 );
    }

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

Outline

1. Point definition
   1.1 Function definitions

```
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    
13 public:
14    Point( int = 0, int = 0 );      // default constructor
15    void setPoint( int, int );      // set coordinates
16    int getX() const { return x; }  // get x coordinate
17    int getY() const { return y; }  // get y coordinate
18 
19 protected:        // accessible to derived classes
20    int x, y;      // coordinates of the point
21      
22 #endif
23 // Fig. 9.8: point2.cpp
24 // Member functions for class Point
25 #include "point2.h"
26 
27 // Constructor for class Point
28 Point::Point( int a, int b ) { setPoint( a, b ); }
29 
30 // Set the x and y coordinates
31 void Point::setPoint( int a, int b )
32 {
33    x = a;
```

Point data members are protected to be made accessible by Circle.
```cpp
// Fig. 9.9: 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:
    double radius;   // radius of the Circle
};

#endif

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

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

#include "circle2.h"
```

1. `circle` definition

Circle data members are protected to be made accessible by Cylinder.
1.1 Function definitions

```cpp
// 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; }

// 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
}
```

---

```cpp
// Fig. 9.10: 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 &output, const Cylinder &c );

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
```
1.1 Function definitions

---

1. Load headers

2. Function calls

3. Output data

---

1. Initialize object

2. Change attributes

3. Output data
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
1. Base1 definition

--------

1. Base2 definition

--------

1. Derived Definition

---

Derived inherits from Base1 and Base2.
1. Load header

1.1 Function Definitions

// Fig. 9.11: derived.cpp
// Member function definitions for class Derived
#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 ) { }

// Display all the data members of Derived
ostream operator<<( ostream &output, const Derived &d )
{
    output << "    Integer: " << d.value
    << "  Character: " << d.letter
    << "  Real number: " << d.real;
    return output;   // enables cascaded calls
}

// Fig. 9.11: fig09_11.cpp
// Driver for multiple inheritance example
#include <iostream>
#include "base1.h"
#include "base2.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() 
    << "  Character: " << b1.letter 
    << "  Real number: " << b1.real; 

    // print data members of derived class objects
    cout << "Object d contains:
" << d << "
    Integer: " << d.Base1::getData() 
        << "  Character: " << d.Base2::getData() 
        << "  Real number: " << d.getReal() << "\n";

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

    return 0;
}
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:
basedPtr->getData() yields 7
base2Ptr->getData() yields A

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)