Derived Classes and Inheritance

Chapter 9 D&D

Derived Classes

- It is sometimes the case that we have a class is nearly what we need.
- Derived classes acquire the properties of an existing class.
- The original class is called the base class.
Inheritance

- Inheritance
  - New classes created from existing classes
  - 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

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)

<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>
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

---

**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
Derived Classes

- A derived class inherits member functions of base class.
- A derived class can be used anywhere the base class is expected.

- However, a base class CANNOT be used anywhere the derived class is expected.
Casting Base-Class Pointers to Derived Class Pointers

• 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:

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

An Example

• 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 & output, const Point & p )
{
    output << '(' << p.x << " , " << p.y << ')';
    return output; // enables cascaded calls
}

friend Point operator=( Point & p )
{
    p.x = a;
    p.y = b;
}

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,
}

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

#include <iostream>

using std::ostream;

#include "point.h"

class Circle : public Point { // Circle inherits from Point

friend ostream &operator<<( ostream & output, const Circle & c )
{
    return output;
}

public:
    Circle( int = 0, int = 0 ); // default constructor
};

#endif

// Fig. 9.4: circle.cpp
// Member functions for class Circle
#include <iostream>

using std::ostream;

#include "point.h"

class Circle : public Point { // Circle inherits from Point

public:
    Circle( int = 0, int = 0 ); // 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: 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 ); }

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

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

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: fig09_04.cpp

// Casting base-class pointers to derived-class pointers

#include <iostream>

Point * pointPtr = 0, p( 30, 50 );

int main()
{
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);

// cast base-class pointer to derived-class pointer
pointPtr = &p; // assign address of Point to pointPtr

circlePtr = static_cast<Circle*>(pointPtr);

return 0;

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
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

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
```cpp
// Fig. 9.5: employ.h
// Definition of class Employee
#if !defined 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

// Member function definitions for class Employee
#include <iostream>

using std::cout;
#include <string>
#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
#if !defined 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 hourlyRate; // rate of pay
};
#endif
```

**HourlyWorker** inherits from **Employee**.

**HourlyWorker** will override the **print** function.
// Member function definitions for class HourlyWorker
#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::ios;
using std::setw;
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; }

// 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 $";
    << setwflags( ios::fixed | ios::showpoint )
    << setprecision(2) << getPay() << endl;
}

// 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;
}

HourlyWorker::print() is executing

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

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

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

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

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

// Fig. 9.7: circle2.h
// Definition of class Circle
#ifdef 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
```

Circle inherits from Point.

---

14
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
...

// Destructor for class Circle
Circle::~Circle()
{
...
Program Output

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
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
  – 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

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
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

```cpp
1 // Fig. 9.8: point2.h
2 // Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 #include <iostream>
7 using std::ostream;
8
9 class Point {
10   friend ostream operator<<( ostream & o, const Point & );
11 public:
12   Point( int = 0, int = 0 ); // default constructor
13   void setPoint( int, int ); // set coordinates
14   int getX() const { return x; } // get x coordinate
15   int getY() const { return y; } // get y coordinate
16 protected: // accessible to derived classes
17     int x, y; // coordinates of the point
18};
19
20 #endif
21 // Fig. 9.8: point2.cpp
22 // Member functions for class Point
23 #include "point2.h"
24
25 // Constructor for class Point
26 Point::Point( int a, int b ) { setPoint( a, b ); }
27
28 // Set the x and y coordinates
29 void Point::setPoint( int a, int b )
30 {
31     x = a;
32
33     y = b;
34 }
```
\begin{verbatim}
"Y=\theta;

// Output the Point
ostream operator<<(ostream &output, const Point &p)
{
    output << '[' << p.x << "\," << p.y << ']';
    return output;  // enables cascading
}
\end{verbatim}
34 // Constructor for Circle calls constructor for Point
35 // with a member initializer and initializes radius
36 Circle::Circle( double r, int a, int b )
37 : Point( a, b ) // call base-class constructor
38 { setRadius( r ); } // setRadius
40
41 // Set radius
42 void Circle::setRadius( double r )
43 { radius = ( r >= 0 ? r : 0 ); } // set radius
44
45 // Get radius
46 double Circle::getRadius() const { return radius; } // get radius
47
48 // Calculate area of Circle
49 double Circle::area() const
50 { return 3.14159 * radius * radius; } // calculate area
51
52 // Output a circle in the form:
53 // Center = [ x, y ]; Radius = #.##
54 ostream & operator<<( ostream & output, const Circle & c )
55 {
56 output << "Center = " << static_cast< Point > ( c )
57 << "; Radius = "
58 << setiosflags( ios::fixed | ios::showpoint )
59 << setprecision( 2 ) << c.radius;
60
61 return output; // enables cascaded calls
62 }

1 // Fig. 9.10: cylind2.h
2 // Definition of class Cylinder
3 #ifdef CYLIND2_R
4 #define CYLIND2_R
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 & output, const Cylinder & c );
14
15 public:
16 // default constructor
17 Cylinder( double h = 0.0, double r = 0.0, int x = 0, int y = 0 );
18
19 void setHeight( double ); // set height
20 double getHeight() const; // return height
21 double area() const; // calculate and return area
22 double volume() const; // calculate and return volume
23
24 protected:
25 double height; // height of the Cylinder
26
27 
28 #endif
// Fig. 9.10: cylindr2.cpp
// Member and friend function definitions
#include "cylindr2.h"

// Cylinder constructor calls Circle constructor
Cylinder::Cylinder( double h, double r, int x, int y )
{ Circle( x, 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: fig09_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 );
  cout << "X coordinate is 12
        Y coordinate is 23
        Radius is 2.5
        Height is 5.7
        ");
  cyl.setHeight( 10 );
cyl.setRadius( 4.25 );
cyl.setPoint( 2, 2 );
  return 0;
}
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:
Center = [2, 2]; Radius = 4.25
Area: 56.74

Multiple Inheritance

- Multiple Inheritance
  - Derived-class inherits from multiple base-classes
  - Encourages software reuse, but can create ambiguities
// 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: 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: 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 &s, const Derived & );

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

private:
    double real;  // derived class's private data
};
#endif
#include "derived.h"

// Fig. 9.11: derived.cpp
// Member function definitions for class Derived

// 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
}

// Driver for multiple inheritance example
#include <iostream>

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

#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
    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"
        "BASE1PTR = id;
    base1Ptr = &d;
    cout << "BASE1PTR->getData() yields "
        << base1Ptr->getData() << "\n";

    cout << "BASE2PTR = id;
    base2Ptr = &d;

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

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:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A