UCLA Stat 130D
Statistical Computing and Visualization in C++

Instructor: Ivo Dinov, Asst. Prof. in Statistics / Neurology

University of California, Los Angeles, Winter 2007

http://www.stat.ucla.edu/~dinov/courses_students.html

Inheritance in C++

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

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

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`

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

`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

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;
```


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


```
// 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. 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;
```

1. `Point` class definition

---

1. Load header

1.1 Function definitions


```
   y = b;
}

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

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

1.1 Function definitions

---

1. `Circle` class definition


```
   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
// 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 ); }
```

1. Circle definition

---

1. Load header

1.1 Function Definitions

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

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

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

   return output;   // enables cascaded calls
}
// Fig. 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 );
```

1. 1 Function Definitions

Driver

1. Load headers

1.1 Initialize objects


```
   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)
   // 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;
}
```

1.1 Initialize objects

1.2 Assign objects

2. Function calls

Program Output

```
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
  - Cannot directly access **`private`** members of its base class
  - 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 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


```
// 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
// 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 ];
```

1. `Employee` class definition

---

1. Load header

1.1 Function definitions

```
33    assert( firstName != 0 ); // terminate if not allocated
34    strcpy( firstName, first );
35
36    lastName = new char[ strlen( last ) + 1 ];
37    assert( lastName != 0 );  // terminate if not allocated
38    strcpy( lastName, last );
39 }
40
41 // Output employee name
42 void Employee::print() const
43    { cout << firstName << ' ' << lastName; }
44
45 // Destructor deallocates dynamically allocated memory
46 Employee::~Employee()
47 {
48    delete [] firstName;   // reclaim dynamic memory
49    delete [] lastName;    // reclaim dynamic memory
50 }
51 // hourly.h
52 // Definition of class HourlyWorker
53 #ifndef HOURLY_H
54 #define HOURLY_H
55
56 #include "employ.h"
57
58 class HourlyWorker : public Employee {
59 public:
60    HourlyWorker( const char*, const char*, double, double );
61    double getPay() const;  // calculate and return salary
62    void print() const;     // overridden base-class print
63 private:
```

1.1 Function definitions

1. HourlyWorker class definition


```
64    double wage;            // wage per hour
65    double hours;           // hours worked for week
66 };
67
68 #endif
69 // hourly.cpp
70 // Member function definitions for class HourlyWorker
71 #include <iostream>
72
73 using std::cout;
74 using std::endl;
75
76 #include <iomanip>
77
78 using std::ios;
79 using std::setiosflags;
80 using std::setprecision;
81
82 #include "hourly.h"
83
84 // Constructor for class HourlyWorker
85 HourlyWorker::HourlyWorker( const char *first,
86                             const char *last,
87                             double initHours, double initWage )
88    : Employee( first, last )   // call base-class constructor
89 {
90    hours = initHours;  // should validate
91    wage = initWage;    // should validate
92 }
93
94 // Get the HourlyWorker's pay
95 double HourlyWorker::getPay() const { return wage * hours; }
```

1. Load header

1.1 Function definitions


```
96
97 // Print the HourlyWorker's name and pay
98 void HourlyWorker::print() const
99 {
100   cout << "HourlyWorker::print() is executing\n\n";
101   Employee::print();   // call base-class print function
102
103   cout << " is an hourly worker with pay of $"
104        << setiosflags( ios::fixed | ios::showpoint )
105        << setprecision( 2 ) << getPay() << endl;
106}
107// Example.cpp
108// Overriding a base-class member function in a
109// derived class.
110#include "hourly.h"
111
112int main()
113{
114   HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
115   h.print();
116   return 0;
117}
HourlyWorker::print() is executing

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

1.1 Function Definitions

1. Load header

1.1 Initialize object

2. Function call

`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.

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`**

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

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

1. Point definition

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

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

#endif
```

---

1. Load header

1.1 Function definitions

```
// 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;
}
// circle2.h
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

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

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

#endif
```

1.1 Function definitions

---

1. Load header

1.1 Circle Definition


```
// 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;
}

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

1. Load header

1.1 Function Definitions


```
// Example.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 );
   }

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

1. Load headers

1.1 Initialize objects

2. Objects enter and leave scope

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]
```

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

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

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

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`**

"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 or handle to another object
  - Also called an association

Case Study: Point, Circle, Cylinder

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

1. Point definition

1.1 Function definitions

```
1  // 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 // 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;
```


1.1 Function definitions

```
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 }
```


1. Circle definition

1.1 Function definitions

```
1  // circle2.h
2  // Definition of class Circle
3  #ifndef CIRCLE2_H
4  #define CIRCLE2_H
5
6  #include <iostream>
7
8  using std::ostream;
9
10 #include "point2.h"
11
12 class Circle : public Point {
13    friend ostream &operator<<( ostream &, const Circle & );
14 public:
15    // default constructor
16    Circle( double r = 0.0, int x = 0, int y = 0 );
17    void setRadius( double );    // set radius
18    double getRadius() const;    // return radius
19    double area() const;         // calculate area
20 protected:          // accessible to derived classes
21    double radius;   // radius of the Circle
22 };
23
24 #endif
25 // circle2.cpp
26 // Member function definitions for class Circle
27 #include <iomanip>
28
29 using std::ios;
30 using std::setiosflags;
31 using std::setprecision;
32
33 #include "circle2.h"
```


1.1 Function definitions

```
34
35 // Constructor for Circle calls constructor for Point
36 // with a member initializer and initializes radius
37 Circle::Circle( double r, int a, int b )
38    : Point( a, b )       // call base-class constructor
39 { setRadius( r ); }
40
41 // Set radius
42 void Circle::setRadius( double r )
43    { radius = ( r >= 0 ? r : 0 ); }
44
45 // Get radius
46 double Circle::getRadius() const { return radius; }
47
48 // Calculate area of Circle
49 double Circle::area() const
50    { return 3.14159 * radius * radius; }
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. Cylinder definition

1.1 Function definitions

```
1  // 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
29 #endif
```


1.1 Function definitions

```
30 // cylindr2.cpp
31 // Member and friend function definitions
32 // for class Cylinder.
33 #include "cylindr2.h"
34
35 // Cylinder constructor calls Circle constructor
36 Cylinder::Cylinder( double h, double r, int x, int y )
37    : Circle( r, x, y )   // call base-class constructor
38 { setHeight( h ); }
39
40 // Set height of Cylinder
41 void Cylinder::setHeight( double h )
42    { height = ( h >= 0 ? h : 0 ); }
43
44 // Get height of Cylinder
45 double Cylinder::getHeight() const { return height; }
46
47 // Calculate area of Cylinder (i.e., surface area)
48 double Cylinder::area() const
49 {
50    return 2 * Circle::area() +
51           2 * 3.14159 * radius * height;
52 }
53
54 // Calculate volume of Cylinder
55 double Cylinder::volume() const
56    { return Circle::area() * height; }
57
58 // Output Cylinder dimensions
59 ostream &operator<<( ostream &output, const Cylinder &c )
60 {
```

```
61    output << static_cast< Circle >( c )
62           << "; Height = " << c.height;
63
64    return output;   // enables cascaded calls
65 }
66 // Example.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 );
```

initions

Driver

1. Load headers

1.1 Initialize object

2. Function calls

2.1 Change attributes

3. Output


```
92    cout << "The new location, radius, and height of cyl are:\n"
93         << cyl << '\n';
94
95    cout << "The area of cyl is:\n"
96         << cyl.area() << '\n';
97
98    // display the Cylinder as a Point
99    Point &pRef = cyl;   // pRef "thinks" it is a Point
100   cout << "\nCylinder printed as a Point is: "
101        << pRef << "\n\n";
102
103   // display the Cylinder as a Circle
104   Circle &circleRef = cyl;  // circleRef thinks it is a Circle
105   cout << "Cylinder printed as a Circle is:\n" << circleRef
106        << "\nArea: " << circleRef.area() << endl;
107
108   return 0;
109}
```

3. Output

Program Output

```
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
```