IS 0020
Program Design and Software Tools
Introduction to C++ Programming

Operator Overloading, Inheritance
Lecture 6

February 10, 2005

Fundamentals of Operator Overloading

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
  - Operator sensitive to context
- Types
  - Built in (**`int`**, **`char`**) or user-defined
  - Can use existing operators with user-defined types
    - Cannot create new operators
- Overloading operators
  - Create a function for the class
  - Name function **`operator`** followed by symbol
    - **`Operator+`** for the addition operator **`+`**

Fundamentals of Operator Overloading

- Using operators on a class object
  - It must be overloaded for that class
  - Exceptions:
    - Assignment operator, **=**
      - May be used without explicit overloading
      - Memberwise assignment between objects
    - Address operator, **&**
      - May be used on any class without overloading
      - Returns address of object
    - Both can be overloaded

Restrictions on Operator Overloading

- Cannot change
  - How operators act on built-in data types
    - I.e., cannot change integer addition
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order-of-operations
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - **&** is unitary, only acts on one operand
- Cannot create new operators
- Operators must be overloaded explicitly
  - Overloading **+** does not overload **+=**

Restrictions on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
  - Member functions
    - Use **`this`** keyword to implicitly get argument
    - Gets left operand for binary operators (like **+**)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **`friend`** to access **`private`** or **`protected`** data
- Example Overloaded **<<** operator
  - Left operand of type **`ostream &`**
    - Such as **`cout`** object in **`cout << classObject`**
  - Similarly, overloaded **>>** needs **`istream &`**
  - Thus, both must be non-member functions

Operator Functions As Class Members Vs. As Friend Functions

- Commutative operators
  - May want **+** to be commutative
    - So both "**a + b**" and "**b + a**" work
  - Suppose we have two different classes
  - Overloaded operator can only be member function when its class is on left
    - **HugeIntClass + Long int**
    - Can be member function
  - When other way, need a non-member overload function
    - **Long int + HugeIntClass**

Overloading Stream-Insertion and Stream-Extraction Operators

- **<<** and **>>**
  - Already overloaded to process each built-in type
  - Can also process a user-defined class
- Example program
  - Class **PhoneNumber**
    - Holds a telephone number
  - Print out formatted number automatically
    - **(123) 456-7890**

Outline

fig08_03.cpp
(1 of 3)

```
1  // Fig. 8.3: fig08_03.cpp
2  // Overloading the stream-insertion and
3  // stream-extraction operators.
4  #include <iostream>
5
6  using std::cout;
7  using std::cin;
8  using std::endl;
9  using std::ostream;
10 using std::istream;
11
12 #include <iomanip>
13
14 using std::setw;
15
16 // PhoneNumber class definitio
17 class PhoneNumber {
18    friend ostream &operator<<(
19    friend istream &operator>>(
20
21 private:
22    char areaCode[ 4 ];  // 3-d
23    char exchange[ 4 ];  // 3-digit exchange and null
24    char line[ 5 ];      // 4-digit line and null
25
26 }; // end class PhoneNumber
```

Notice function prototypes for overloaded operators **>>** and **<<**

They must be non-member **friend** functions, since the object of class **Phonenumber** appears on the right of the operator.

```
cin << object
cout >> object
```


Outline

fig08_03.cpp
(2 of 3)

```
27
28 // overloaded stream-insertion operator; cannot be
29 // a member function if we would like to invoke it with
30 // cout << somePhoneNumber;
31 ostream &operator<<( ostream &output, const PhoneNumber &num )
32 {
33    output << "(" << num.areaCode << ") "
34           << num.exchange << "-" << num.line;
35
36    return output;     // enables cout << a << b << c;
37
38 } // end function operator<<
39
40 // overloaded stream-extraction operator; cannot b
41 // a member function if we would like to invoke it
42 // cin >> somePhoneNumber;
43 istream &operator>>( istream &input, PhoneNumber &
44 {
45    input.ignore();                    // skip (
46    input >> setw( 4 ) >> num.areaCode; // input ar
47    input.ignore( 2 );                 // skip ) a
48    input >> setw( 4 ) >> num.exchange;
49    input.ignore();
50    input >> setw( 5 ) >> num.line;
51
52    return input;      // enables cin
```

The expression:
cout << phone;
is interpreted as the function call:
operator<<(cout, phone);

output is an alias for **cout.**

This allows objects to be cascaded.
cout << phone1 << phone2;
first calls
operator<<(cout, phone1), and returns **cout**.

Next, **cout << phone2** executes.

Stream manipulator **setw** restricts number of characters read. **setw(4)** allows 3 characters to be read, leaving room for the null character.

Outline

fig08_03.cpp
(3 of 3)

fig08_03.cpp
output (1 of 1)

```
53
54 } // end function operator>>
55
56 int main()
57 {
58    PhoneNumber phone; // create object phone
59
60    cout << "Enter phone number in the form (123) 456-7890:\n";
61
62    // cin >> phone invokes operator>> by implicitly issuing
63    // the non-member function call operator>>( cin, phone )
64    cin >> phone;
65
66    cout << "The phone number entered was: " ;
67
68    // cout << phone invokes operator<< by implicitly issuing
69    // the non-member function call operator<<( cout, phone )
70    cout << phone << endl;
71
72    return 0;
73
74 } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```


Overloading Unary Operators

- Overloading unary operators
  - Non-**`static`** member function, no arguments
  - Non-member function, one argument
    - Argument must be class object or reference to class object
  - Remember, **`static`** functions only access **`static`** data

Overloading Operators

- Overloading unary operators (**!** to test for empty string)
  - Non-**static** member function: **!s** becomes **s.operator!()**

    ```
    bool operator!() const;
    ```
  - Non-member function:**!s** becomes **operator!(s)**

    ```
    friend bool operator!( const String & )
    ```
- Overloading binary operators
  - Non-member function (arg. must be class object or reference)

    ```
    friend const String &operator+=(String &, const String & );
    ```
  - Non-**static** member function:

    ```
    const String &operator+=( const String & );
    ```
  - **y += z** equivalent to **y.operator+=(z)**

Case Study: Array class

- Arrays in C++
  - No range checking
  - Cannot be compared meaningfully with **==**
  - No array assignment (array names **const** pointers)
  - Cannot input/output entire arrays at once
- Example:Implement an **Array** class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with **<<** and **>>**
  - Array comparisons with **==** and **!=**

Case Study: Array class

- Copy constructor
  - Used whenever copy of object needed
    - Passing by value (return value or parameter)
    - Initializing an object with a copy of another
      - **`Array newArray( oldArray );`**
      - **`newArray`** copy of **`oldArray`**
  - Prototype for class **`Array`**
    - **`Array( const Array & );`**
    - *Must* take reference
      - Otherwise, pass by value
      - Tries to make copy by calling copy constructor…
      - Infinite loop

Outline

array1.h (1 of 2)

```
1  // Fig. 8.4: array1.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY1_H
4  #define ARRAY1_H
5
6  #include <iostream>
7
8  using std::ostream;
9  using std::istream;
10
11 class Array {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14
15 public:
16    Array( int = 10 );       // default con
17    Array( const Array & );  // copy constr
18    ~Array();                // destructor
19    int getSize() const;     // return size
20
21    // assignment operator
22    const Array &operator=( const Array & );
23
24    // equality operator
25    bool operator==( const Array & ) const;
26
```

Most operators overloaded as member functions (except **<<** and **>>**, which must be non-member functions).

Prototype for copy constructor.

Outline

array1.h (2 of 2)

```
27    // inequality operator; returns opposite of == operator
28    bool operator!=( const Array &right ) const
29    {
30       return ! ( *this == right ); // invokes Array::operator==
31
32    } // end function operator!=
33
34    // subscript operator for non-cor
35    int &operator[]( int );
36
37    // subscript operator for const objects returns rvalue
38    const int &operator[]( int ) const;
39
40 private:
41    int size; // array size
42    int *ptr; // pointer to first element of array
43
44 }; // end class Array
45
46 #endif
```

!= operator simply returns opposite of **==** operator. Thus, only need to define the **==** operator.

Outline

array1.cpp (1 of 7)

```
1  // Fig 8.5: array1.cpp
2  // Member function definitions for class Array
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 #include <new>       // C++ standard "new" operator
14
15 #include <cstdlib>   // exit function prototype
16
17 #include "array1.h"  // Array class definition
18
19 // default constructor for class Array (default size 10)
20 Array::Array( int arraySize )
21 {
22    // validate arraySize
23    size = ( arraySize > 0 ? arraySize : 10 );
24
25    ptr = new int[ size ]; // create space for array
26
```

Outline

array1.cpp (2 of 7)

```
27    for ( int i = 0; i < size; i++ )
28       ptr[ i ] = 0;          // initialize array
29
30 } // end Array default constructor
31
32 // copy constructor for class Array;
33 // must receive a reference to pre
34 Array::Array( const Array &arrayTo
35    : size( arrayToCopy.size )
36 {
37    ptr = new int[ size ]; // create space for array
38
39    for ( int i = 0; i < size; i++ )
40       ptr[ i ] = arrayToCopy.ptr[ i ];  // copy into object
41
42 } // end Array copy constructor
43
44 // destructor for class Array
45 Array::~Array()
46 {
47    delete [] ptr;  // reclaim array space
48
49 } // end destructor
50
```

We must declare a new integer array so the objects do not point to the same memory.

Outline

array1.cpp (3 of 7)

```
51 // return size of array
52 int Array::getSize() const
53 {
54    return size;
55
56 } // end function getSize
57
58 // overloaded assignment operator
59 // const return avoids: ( a1 = a2 ) = a3
60 const Array &Array::operator=( const Array &right )
61 {
62    if ( &right != this ) {  // check for self-assignment
63
64       // for arrays of different sizes, deallocate original
65       // left-side array, then allocate new left-side array
66       if ( size != right.size ) {
67          delete [] ptr;         // reclaim space
68          size = right.size;     // resize this object
69          ptr = new int[ size ]; // create space for array copy
70
71       } // end inner if
72
73       for ( int i = 0; i < size; i++ )
74          ptr[ i ] = right.ptr[ i ];  // copy array into object
75
76    } // end outer if
```

Want to avoid self-assignment.

Outline

array1.cpp (4 of 7)

```
77
78    return *this;   // enables x = y = z, for example
79
80 } // end function operator=
81
82 // determine if two arrays are equal and
83 // return true, otherwise return false
84 bool Array::operator==( const Array &right ) const
85 {
86    if ( size != right.size )
87       return false;    // arrays of different sizes
88
89    for ( int i = 0; i < size; i++ )
90
91       if ( ptr[ i ] != right.ptr[ i ] )
92          return false; // arrays are not equal
93
94    return true;        // arrays are equal
95
96 } // end function operator==
97
```


Outline

array1.cpp (5 of 7)

```
98 // overloaded subscript operator for non-const Arrays
99 // reference return creates an lvalue
100 int &Array::operator[]( int subscript )
101 {
102    // check for subscript out of range error
103    if ( subscript < 0 || subscript >= size
104       cout << "\nError: Subscript " << sub
105            << " out of range" << endl;
106
107       exit( 1 );  // terminate program; subscript out of range
108
109    } // end if
110
111    return ptr[ subscript ]; // reference return
112
113 } // end function operator[]
114
```

integers1[5] calls **integers1.operator[](5)**

exit() (header **<cstdlib>**) ends the program.

Outline

array1.cpp (6 of 7)

```
115 // overloaded subscript operator for const Arrays
116 // const reference return creates an rvalue
117 const int &Array::operator[]( int subscript ) const
118 {
119    // check for subscript out of range error
120    if ( subscript < 0 || subscript >= size ) {
121       cout << "\nError: Subscript " << subscript
122            << " out of range" << endl;
123
124       exit( 1 );  // terminate program; subscript out of range
125
126    } // end if
127
128    return ptr[ subscript ]; // const reference return
129
130 } // end function operator[]
131
132 // overloaded input operator for class Array;
133 // inputs values for entire array
134 istream &operator>>( istream &input, Array &a )
135 {
136    for ( int i = 0; i < a.size; i++ )
137       input >> a.ptr[ i ];
138
139    return input;   // enables cin >> x >> y;
140
141 } // end function
```


Outline

array1.cpp (7 of 7)

```
142
143 // overloaded output operator for class Array
144 ostream &operator<<( ostream &output, const Array &a )
145 {
146    int i;
147
148    // output private ptr-based array
149    for ( i = 0; i < a.size; i++ ) {
150       output << setw( 12 ) << a.ptr[ i ];
151
152       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
153          output << endl;
154
155    } // end for
156
157    if ( i % 4 != 0 )  // end last line of output
158       output << endl;
159
160    return output;   // enables cout << x << y;
161
162 } // end function operator<<
```

Converting between Types

- Cast operator (conversion operator)
  - Convert from One class to another built-in type
  - Must be non-**`static`** member function -
    - Cannot be **`friend`**
  - Do not specify return type
    - Implicitly returns type to which you are converting
  - Example: **`A::operator char *() const;`**
    - Casts class **`A`** to a temporary **`char *`**
    - **`(char *)s`** calls **`s.operator char*()`**

    ```
    A::operator int() const;
    A::operator OtherClass() const;
    ```
- Casting can prevent need for overloading
  - Suppose class **`String`** can be cast to **`char *`**
  - **`cout << s;`** // **`cout`** expects **`char *`**; **`s`** is a **`String`**
    - Compiler implicitly calls the function to convert **`s`** to **`char *`**
    - Do not have to overload **`<< for String`**

Case Study: A `String` Class

- Build class **`String`**
  - String creation, manipulation
  - Class **`string`** in standard library (more Chapter 15)
- Conversion constructor
  - Single-argument constructor
  - Turns objects of other types into class objects
    - **`String s1("hi");`**
    - Creates a **`String`** from a **`char *`**
  - Any single-argument constructor is a conversion constructor

Overloading ++ and - -

- Increment/decrement operators can be overloaded
  - Add 1 to a **`Date`** object, **`d1`**
  - Prototype (member function)
    - **`Date &operator++();`**
    - **`++d1`** same as **`d1.operator++()`**
  - Prototype (non-member)
    - **`Friend Date &operator++( Date &);`**
    - **`++d1`** same as **`operator++( d1 )`**

Overloading ++ and - -

- To distinguish pre/post increment
  - Post increment has a dummy parameter
    - **`int`** of **`0`**
  - Prototype (member function)
    - **`Date operator++( int );`**
    - **`d1++`** same as **`d1.operator++( 0 )`**
  - Prototype (non-member)
    - **`friend Date operator++( Data &, int );`**
    - **`d1++`** same as **`operator++( d1, 0 )`**
  - Integer parameter does not have a name
    - Not even in function definition

Overloading ++ and - -

- Return values
  - Preincrement
    - Returns by reference (**`Date &`**)
    - lvalue (can be assigned)
  - Postincrement
    - Returns by value
    - Returns temporary object with old value
    - rvalue (cannot be on left side of assignment)
- Example **`Date`** class
  - Overloaded increment operator
    - Change day, month and year
  - Overloaded **`+=`** operator
  - Function to test for leap years
  - Function to determine if day is last of month

Outline

date1.h (1 of 2)

```
1  // Fig. 8.10: date1.h
2  // Date class definition.
3  #ifndef DATE1_H
4  #define DATE1_H
5  #include <iostream>
6
7  using std::ostream;
8
9  class Date {
10    friend ostream &operator<<( ostream &, const Date & );
11
12 public:
13    Date( int m = 1, int d = 1, int y =
14    void setDate( int, int, int ); // s
15
16    Date &operator++();            // preincrement operator
17    Date operator++( int );        // postincrement operator
18
19    const Date &operator+=( int ); // add days, modify object
20
21    bool leapYear( int ) const;    // is this a leap year?
22    bool endOfMonth( int ) const;  // is this end of month?
```

Note difference between pre and post increment.

Outline

date1.h (2 of 2)

```
23
24 private:
25    int month;
26    int day;
27    int year;
28
29    static const int days[];       // array of days per month
30    void helpIncrement();          // utility function
31
32 }; // end class Date
33
34 #endif
```

```
35 Date &Date::operator++()
36  {
37     helpIncrement();
37    return *this;  // reference return to create an lvalue
39 } // end function operator++
40
41 // overloaded postincrement operator; note that the dummy
42 // integer parameter does not have a parameter name
43 Date Date::operator++( int )
44 {
45    Date temp = *this;  // hold current state of object
46    helpIncrement();
48    // return unincremented, saved, temporary object
49    return temp;   // value return; not a reference return
51 } // end function operator++
```


Inheritance

- Inheritance
  - Software reusability
  - Create new class from existing class
    - Absorb existing class's data and behaviors
    - Enhance with new capabilities
  - Derived class inherits from base class
    - Derived class
      - More specialized group of objects
      - Behaviors inherited from base class
        - Can customize
      - Additional behaviors

Inheritance

- Class hierarchy
  - Direct base class
    - Inherited explicitly (one level up hierarchy)
  - Indirect base class
    - Inherited two or more levels up hierarchy
  - Single inheritance
    - Inherits from one base class
  - Multiple inheritance
    - Inherits from multiple base classes
      - Base classes possibly unrelated
    - Chapter 22

Inheritance

- Three types of inheritance
  - **`public`**
    - Every object of derived class also object of base class
      - Base-class objects not objects of derived classes
      - Example: All cars vehicles, but not all vehicles cars
    - Can access non-**`private`** members of base class
      - Derived class can effect change to **`private`** base-class members
        - Through inherited non-**`private`** member functions
  - **`private`**
    - Alternative to composition
    - Chapter 17
  - **`protected`**
    - Rarely used

35

Inheritance

- Abstraction
  - Focus on commonalities among objects in system
- "is-a" vs. "has-a"
  - "is-a"
    - Inheritance
    - Derived class object treated as base class object
    - Example: Car *is a* vehicle
      - Vehicle properties/behaviors also car properties/behaviors
  - "has-a"
    - Composition
    - Object contains one or more objects of other classes as members
    - Example: Car *has a* steering wheel

36

Base Classes and Derived Classes

- Base classes and derived classes
  - Object of one class "is an" object of another class
    - Example: Rectangle is quadrilateral.
  - Base class typically represents larger set of objects than derived classes
    - Example:
      - Base class: **`Vehicle`**
        - Cars, trucks, boats, bicycles, …
      - Derived class: **`Car`**
        - Smaller, more-specific subset of vehicles

Base Classes and Derived Classes

- Inheritance examples

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

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

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

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

Base Classes and Derived Classes

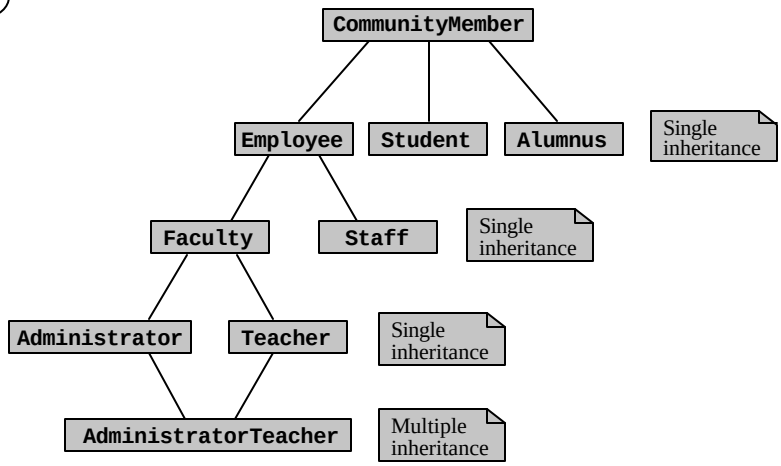
- Inheritance hierarchy
  - Inheritance relationships: tree-like hierarchy structure
  - Each class becomes
    - Base class
      - Supply data/behaviors to other classes

    OR
    - Derived class
      - Inherit data/behaviors from other classes

39

Inheritance hierarchy

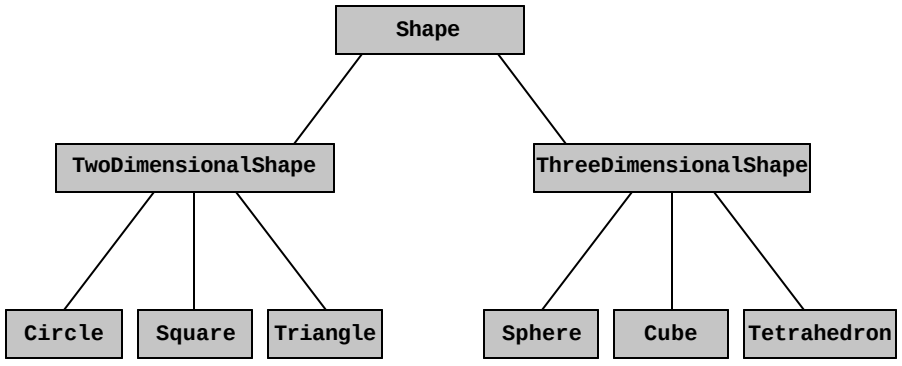
Fig. 9.2 Inheritance hierarchy for university **`CommunityMember`**s.



40

Inheritance hierarchy

Fig. 9.3 Inheritance hierarchy for Shapes.



Base Classes and Derived Classes

- **`public`** inheritance
  - Specify with:
    `Class TwoDimensionalShape : public Shape`
    - Class **`TwoDimensionalShape`** inherits from class **`Shape`**
  - Base class **`private`** members
    - Not accessible directly
    - Still inherited - manipulate through inherited member functions
  - Base class **`public`** and **`protected`** members
    - Inherited with original member access
  - **`friend`** functions
    - Not inherited

`protected` Members

- **`protected`** access
  - Intermediate level of protection between **`public`** and **`private`**
  - **`protected`** members accessible to
    - Base class members
    - Base class **`friend`**s
    - Derived class members
    - Derived class **`friend`**s
  - Derived-class members
    - Refer to **`public`** and **`protected`** members of base class
      - Simply use member names

Relationship between Base Classes and Derived Classes

- Base class and derived class relationship
  - Example: Point/circle inheritance hierarchy
    - Point
      - x-y coordinate pair
    - Circle
      - x-y coordinate pair
      - Radius

Relationship between Base Classes and Derived Classes

- Using **`protected`** data members
  - Advantages
    - Derived classes can modify values directly
    - Slight increase in performance
      - Avoid set/get function call overhead
  - Disadvantages
    - No validity checking
      - Derived class can assign illegal value
    - Implementation dependent
      - Derived class member functions more likely dependent on base class implementation
      - Base class implementation changes may result in derived class modifications
        - Fragile (brittle) software

45

Case Study: Three-Level Inheritance Hierarchy

- Three level point/circle/cylinder hierarchy
  - Point
    - x-y coordinate pair
  - Circle
    - x-y coordinate pair
    - Radius
  - Cylinder
    - x-y coordinate pair
    - Radius
    - Height

46

Constructors and Destructors in Derived Classes

- Instantiating derived-class object
  - Chain of constructor calls
    - Derived-class constructor invokes base class constructor
      - Implicitly or explicitly
    - Base of inheritance hierarchy
      - Last constructor called in chain
      - First constructor body to finish executing
      - Example: **`Point3`**/**`Circle4`**/**`Cylinder`** hierarchy
        - **`Point3`** constructor called last
        - **`Point3`** constructor body finishes execution first
    - Initializing data members
      - Each base-class constructor initializes data members inherited by derived class

Constructors and Destructors in Derived Classes

- Destroying derived-class object
  - Chain of destructor calls
    - Reverse order of constructor chain
    - Destructor of derived-class called first
    - Destructor of next base class up hierarchy next
      - Continue up hierarchy until final base reached
        - After final base-class destructor, object removed from memory
- Base-class constructors, destructors, assignment operators
  - Not inherited by derived classes
  - Derived class constructors, assignment operators can call
    - Constructors
    - Assignment operators

`public`, `protected` and `private` 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.

Outline

point.h (1 of 1)

```
1  // Fig. 9.4: point.h
2  // Point class definition represents an x-y coordinate pair.
3  #ifndef POINT_H
4  #define POINT_H
5
6  class Point {
7
8  public:
9     Point( int = 0, int = 0 ); // default constructor
10
11    void setX( int );          // set x in coordinate pair
12    int getX() const;          // return x from coordinate pair
13
14    void setY( int );          // set y in coordinate pair
15    int getY() const;          // return y from coordinate pair
16
17    void print() const;        // output Point object
18
19 private:
20    int x;  // x part of coordinate pair
21    int y;  // y part of coordinate pair
22
23 }; // end class Point
24
25 #endif
```

Maintain **x**- and **y**-coordinates as **private** data members.

Outline

circle2.h (1 of 2)

```
1  // Fig. 9.10: circle2.h
2  // Circle2 class contains x-y coordinate pair and radius.
3  #ifndef CIRCLE2_H
4  #define CIRCLE2_H
5
6  #include "point.h"  // Point class
7
8  class Circle2 : public Point {
9
10 public:
11
12    // default constructor
13    Circle2( int = 0, int = 0, do
14
15    void setRadius( double );   // set radius
16    double getRadius() const;   // return radius
17
18    double getDiameter() const;       // return diameter
19    double getCircumference() const;  // return circumference
20    double getArea() const;           // return area
21
22    void print() const;
23
24 private:
25    double radius;  // Circle2's radius
```

Class **Circle2** inherits from class **Point**.

Keyword **public** indicates type of inheritance.

Maintain **private** data member **radius**.

Outline

```
26
27 }; // end class Circle2
28
29 #endif
```

circle2.h (2 of 2)

circle2.cpp (1 of 3)

```
1  // Fig. 9.11: circle2.cpp
2  // Circle2 class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "circle2.h"   // Circle2 class definition
8
9  // default constructor
10 Circle2::Circle2( int xValue, i                    e )
11 {
12    x = xValue;
13    y = yValue;
14    setRadius( radiusValue );
15
16 } // end Circle2 constructor
17
```

Attempting to access base class **Point**'s **private** data members **x** and **y** results in syntax errors.

Outline

circle2.cpp (2 of 3)

```
18 // set radius
19 void Circle2::setRadius( double radiusValue )
20 {
21    radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
22
23 } // end function setRadius
24
25 // return radius
26 double Circle2::getRadius() const
27 {
28    return radius;
29
30 } // end function getRadius
31
32 // calculate and return diameter
33 double Circle2::getDiameter() const
34 {
35    return 2 * radius;
36
37 } // end function getDiameter
38
```

Outline

circle2.cpp (3 of 3)

```
39 // calculate and return circumference
40 double Circle2::getCircumference() const
41 {
42    return 3.14159 * getDiameter();
43
44 } // end function getCircumference
45
46 // calculate and return area
47 double Circle2::getArea() const
48 {
49    return 3.14159 * radius * radius;
50
51 } // end function getArea
52
53 // output Circle2 object
54 void Circle2::print() const
55 {
56    cout << "Center = [" << x << ", " << y << ']'
57         << "; Radius = " << radius;
58
59 } // end function print
```

Attempting to access base class **Point**'s **private** data members **x** and **y** results in syntax errors.

Outline

point.h (1 of 1)

```
1  // Fig. 9.4: point.h
2  // Point class definition represents an x-y coordinate pair.
3  #ifndef POINT_H
4  #define POINT_H
5
6  class Point {
7
8  public:
9     Point( int = 0, int = 0 ); // default constructor
10
11    void setX( int );          // set x in coordinate pair
12    int getX() const;          // return x from coordinate pair
13
14    void setY( int );          // set y in coordinate pair
15    int getY() const;          // return y from coordinate pair
16
17    void print() const;        // output Point object
18
19 private:
20    int x;  // x part of coordinate pair
21    int y;  // y part of coordinate pair
22
23 }; // end class Point
24
25 #endif
```

Maintain **x**- and **y**-coordinates as **private** data members.

Outline

point.cpp (1 of 3)

```
1  // Fig. 9.5: point.cpp
2  // Point class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "point.h"   // Point class definition
8
9  // default constructor
10 Point::Point( int xValue, int yValue )
11 {
12    x = xValue;
13    y = yValue;
14
15 } // end Point constructor
16
17 // set x in coordinate pair
18 void Point::setX( int xValue )
19 {
20    x = xValue; // no need for validation
21
22 } // end function setX
23
```


Outline

point.cpp (2 of 3)

```
24 // return x from coordinate pair
25 int Point::getX() const
26 {
27    return x;
28
29 } // end function getX
30
31 // set y in coordinate pair
32 void Point::setY( int yValue )
33 {
34    y = yValue; // no need for validation
35
36 } // end function setY
37
38 // return y from coordinate pair
39 int Point::getY() const
40 {
41    return y;
42
43 } // end function getY
44
```

```
45  // output Point object
46  void Point::print() const
47  {
48     cout << '[' << x << ", " << y << ']';
49
50  } // end function print
```

Outline

point.cpp (3 of 3)


```
1   // Fig. 9.6: pointtest.cpp
2   // Testing class Point.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include "point.h"  // Point class definition
9
10  int main()
11  {
12     Point point( 72, 115 );      // instantiate Point object
13
14     // display point coordinates
15     cout << "X coordinate is " << point.getX()
16          << "\nY coordinate is " << point.getY();
17
18     point.setX( 10 ); // set x-coordinate
19     point.setY( 10 ); // set y-coordinate
20
21     // display new point value
22     cout << "\n\nThe new location is ";
23     point.print();
24     cout << endl;
25
```

Create a **`Point`** object.

Invoke set functions to modify **`private`** data.

Invoke **`public`** function **`print`** to display new coordinates.

Outline

pointtest.cpp
(1 of 2)

```
26     return 0;  // indicates successful termination
27
28  } // end main
```

```
X coordinate is 72
Y coordinate is 115

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

Outline

pointtest.cpp
(2 of 2)

pointtest.cpp
output (1 of 1)


```
1   // Fig. 9.7: circle.h
2   // Circle class contains x-y coordinate pair and radius.
3   #ifndef CIRCLE_H
4   #define CIRCLE_H
5
6   class Circle {
7
8   public:
9
10     // default constructor
11     Circle( int = 0, int = 0, double = 0.0 );
12
13     void setX( int );              // set
14     int getX() const;              // return x from coordinate pair
15
16     void setY( int );              // set y in coordinate pair
17     int getY() const;              // return y from coordinate pair
18
19     void setRadius( double );      // set radius
20     double getRadius() const;      // return radius
21
22     double getDiameter() const;       // return diameter
23     double getCircumference() const;  // return circumference
24     double getArea() const;           // return area
25
```

Note code similar to **`Point`** code.

Outline

circle.h (1 of 2)

Outline

circle.h (2 of 2)

```
26    void print() const;          // output Circle object
27
28 private:
29    int x;          // x-coordina
30    int y;          // y-coordinate of circle's center
31    double radius;  // Circle's radius
32
33 }; // end class Circle
34
35 #endif
```

Maintain **x**-**y** coordinates and **radius** as **private** data members.

Note code similar to **Point** code.

Outline

circle.cpp (1 of 4)

```
1  // Fig. 9.8: circle.cpp
2  // Circle class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "circle.h"   // Circle class definition
8
9  // default constructor
10 Circle::Circle( int xValue, int yValue, double radiusValue )
11 {
12    x = xValue;
13    y = yValue;
14    setRadius( radiusValue );
15
16 } // end Circle constructor
17
18 // set x in coordinate pair
19 void Circle::setX( int xValue )
20 {
21    x = xValue; // no need for validation
22
23 } // end function setX
24
```

Outline

circle.cpp (2 of 4)

```
25 // return x from coordinate pair
26 int Circle::getX() const
27 {
28    return x;
29
30 } // end function getX
31
32 // set y in coordinate pair
33 void Circle::setY( int yValue )
34 {
35    y = yValue; // no need for validation
36
37 } // end function setY
38
39 // return y from coordinate pair
40 int Circle::getY() const
41 {
42    return y;
43
44 } // end function getY
45
```


Outline

circle.cpp (3 of 4)

```
46 // set radius
47 void Circle::setRadius( double radiusValue )
48 {
49    radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
50
51 } // end function setRadius
52
53 // return radius
54 double Circle::getRadius() const
55 {
56    return radius;
57
58 } // end function getRadius
59
60 // calculate and return diameter
61 double Circle::getDiameter() const
62 {
63    return 2 * radius;
64
65 } // end function getDiameter
66
```

Ensure non-negative value for **`radius`**.

Outline

circle.cpp (4 of 4)

```
67  // calculate and return circumference
68  double Circle::getCircumference() const
69  {
70     return 3.14159 * getDiameter();
71
72  } // end function getCircumference
73
74  // calculate and return area
75  double Circle::getArea() const
76  {
77     return 3.14159 * radius * radius;
78
79  } // end function getArea
80
81  // output Circle object
82  void Circle::print() const
83  {
84     cout << "Center = [" << x << ", " << y << ']'
85          << "; Radius = " << radius;
86
87  } // end function print
```


Outline

circletest.cpp
(1 of 2)

```
1   // Fig. 9.9: circletest.cpp
2   // Testing class Circle.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7   using std::fixed;
8
9   #include <iomanip>
10
11  using std::setprecision;
12
13  #include "circle.h"  // Circle class defin
14
15  int main()
16  {
17     Circle circle( 37, 43, 2.5 );  // instantiate Circle object
18
19     // display point coordinates
20     cout << "X coordinate is " << circle.getX()
21          << "\nY coordinate is " << circle.getY()
22          << "\nRadius is " << circle.getRadius();
23
```

Create **`Circle`** object.

Outline

circletest.cpp
(2 of 2)

```
24     circle.setX( 2 );          // set new x-coordinate
25     circle.setY( 2 );          // set new y-coordinate
26     circle.setRadius( 4.25 );  // set new radius
27
28     // display new point value
29     cout << "\n\nThe new location and
30     circle.print();
31
32     // display floating-point values
33     cout << fixed << setprecision( 2
34
35     // display Circle's diameter
36     cout << "\nDiameter is " << circle.getDiameter();
37
38     // display Circle's circumference
39     cout << "\nCircumference is " << circle.getCircumference();
40
41     // display Circle's area
42     cout << "\nArea is " << circle.getArea();
43
44     cout << endl;
45
46     return 0;  // indicates successful termination
47
48  } // end main
```

Use set functions to modify **`private`** data.

Invoke **`public`** function **`print`** to display new coordinates.

Outline

circletest.cpp
output (1 of 1)

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```

Outline

circle2.h (1 of 2)

```
1  // Fig. 9.10: circle2.h
2  // Circle2 class contains x-y coordinate pair and radius.
3  #ifndef CIRCLE2_H
4  #define CIRCLE2_H
5
6  #include "point.h"  // Point class
7
8  class Circle2 : public Point {
9
10 public:
11
12    // default constructor
13    Circle2( int = 0, int = 0, do
14
15    void setRadius( double );   // set radius
16    double getRadius() const;   // return radius
17
18    double getDiameter() const;       // return diameter
19    double getCircumference() const;  // return circumference
20    double getArea() const;           // return area
21
22    void print() const;          //
23
24 private:
25    double radius; // Circle2's radius
```

Class **Circle2** inherits from class **Point**.

Keyword **public** indicates type of inheritance.

Maintain **private** data member **radius**.

Outline

circle2.h (2 of 2)

```
26
27 }; // end class Circle2
28
29 #endif
```

circle2.cpp (1 of 3)

```
1  // Fig. 9.11: circle2.cpp
2  // Circle2 class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "circle2.h"   // Circle2 class definition
8
9  // default constructor
10 Circle2::Circle2( int xValue, i                    e )
11 {
12    x = xValue;
13    y = yValue;
14    setRadius( radiusValue );
15
16 } // end Circle2 constructor
17
```

Attempting to access base class **Point**'s **private** data members **x** and **y** results in syntax errors.

Outline

circle2.cpp (2 of 3)

```
18 // set radius
19 void Circle2::setRadius( double radiusValue )
20 {
21    radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
22
23 } // end function setRadius
24
25 // return radius
26 double Circle2::getRadius() const
27 {
28    return radius;
29
30 } // end function getRadius
31
32 // calculate and return diameter
33 double Circle2::getDiameter() const
34 {
35    return 2 * radius;
36
37 } // end function getDiameter
38
```


Outline

circle2.cpp (3 of 3)

```
39 // calculate and return circumference
40 double Circle2::getCircumference() const
41 {
42    return 3.14159 * getDiameter();
43
44 } // end function getCircumference
45
46 // calculate and return area
47 double Circle2::getArea() const
48 {
49    return 3.14159 * radius * radius;
50
51 } // end function getArea
52
53 // output Circle2 object
54 void Circle2::print() const
55 {
56    cout << "Center = [" << x << ", " << y << ']'
57         << "; Radius = " << radius;
58
59 } // end function print
```

Attempting to access base class **`Point`**'s **`private`** data members **`x`** and **`y`** results in syntax errors.

```
C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(12) : error C2248: 'x' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(20) :
        see declaration of 'x'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(13) : error C2248: 'y' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(21) :
        see declaration of 'y'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(56) : error C2248: 'x' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(20) :
        see declaration of 'x'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(56) : error C2248: 'y' :
cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(21) :
        see declaration of 'y'
```

Attempting to access base class **Point**'s **private** data members **x** and **y** results in syntax errors.

Outline

circle2.cpp output (1 of 1)


```
1  // Fig. 9.12: point2.h
2  // Point2 class definition represents an x-y coordinate pair.
3  #ifndef POINT2_H
4  #define POINT2_H
5
6  class Point2 {
7
8  public:
9     Point2( int = 0, int = 0 ); // default constructor
10
11    void setX( int );    // set x in coordinate pair
12    int getX() const;    // return x from coordinate pair
13
14    void setY( int );    // set y in coordinate pair
15    int getY() const;    // return y from coordinate pair
16
17    void print() const;  //
18
19 protected:
20    int x;  // x part of coor
21    int y;  // y part of coordinate pair
22
23 }; // end class Point2
24
25 #endif
```

Maintain **x**- and **y**-coordinates as **protected** data, accessible to derived classes.

Outline

point2.h (1 of 1)

Outline

point2.cpp (1 of 3)

```
1  // Fig. 9.13: point2.cpp
2  // Point2 class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "point2.h"   // Point2 class definition
8
9  // default constructor
10 Point2::Point2( int xValue, int yValue )
11 {
12    x = xValue;
13    y = yValue;
14
15 } // end Point2 constructor
16
17 // set x in coordinate pair
18 void Point2::setX( int xValue )
19 {
20    x = xValue; // no need for validation
21
22 } // end function setX
23
```


Outline

point2.cpp (2 of 3)

```
24 // return x from coordinate pair
25 int Point2::getX() const
26 {
27    return x;
28
29 } // end function getX
30
31 // set y in coordinate pair
32 void Point2::setY( int yValue )
33 {
34    y = yValue; // no need for validation
35
36 } // end function setY
37
38 // return y from coordinate pair
39 int Point2::getY() const
40 {
41    return y;
42
43 } // end function getY
44
```

```
45 // output Point2 object
46 void Point2::print() const
47 {
48    cout << '[' << x << ", " << y << ']';
49
50 } // end function print
```


Outline

point2.cpp (3 of 3)


```
1  // Fig. 9.14: circle3.h
2  // Circle3 class contains x-y coordinate pair and radius.
3  #ifndef CIRCLE3_H
4  #define CIRCLE3_H
5
6  #include "point2.h"  // Point2 d
7
8  class Circle3 : public Point2 {
9
10 public:
11
12    // default constructor
13    Circle3( int = 0, int = 0, double = 0.0 );
14
15    void setRadius( double );   // set radius
16    double getRadius() const;   // return radius
17
18    double getDiameter() const;       // return diameter
19    double getCircumference() const;  // return circumference
20    double getArea() const;           // return area
21
22    void print() const;
23
24 private:
25    double radius;  // Circle3's radius
```

Class **`Circle3`** inherits from class **`Point2`**.

Maintain **`private`** data member **`radius`**.

Outline

circle3.h (1 of 2)

Outline

circle3.h (2 of 2)

```
26
27 }; // end class Circle3
28
29 #endif
```


Outline

circle3.cpp (1 of 3)

```
1  // Fig. 9.15: circle3.cpp
2  // Circle3 class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "circle3.h"   // Circle3 class definition
8
9  // default constructor
10 Circle3::Circle3( int xValue,                    )
11 {
12    x = xValue;
13    y = yValue;
14    setRadius( radiusValue );
15
16 } // end Circle3 constructor
17
18 // set radius
19 void Circle3::setRadius( double radiusValue )
20 {
21    radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
22
23 } // end function setRadius
24
```

Constructor first implicitly calls base class's default constructor.

protected in base class **Point2**.

Outline

circle3.cpp (2 of 3)

```
25 // return radius
26 double Circle3::getRadius() const
27 {
28    return radius;
29
30 } // end function getRadius
31
32 // calculate and return diameter
33 double Circle3::getDiameter() const
34 {
35    return 2 * radius;
36
37 } // end function getDiameter
38
39 // calculate and return circumference
40 double Circle3::getCircumference() const
41 {
42    return 3.14159 * getDiameter();
43
44 } // end function getCircumference
45
```


Outline

circle3.cpp (3 of 3)

```
46 // calculate and return area
47 double Circle3::getArea() const
48 {
49    return 3.14159 * radius * radius;
50
51 } // end function getArea
52
53 // output Circle3 object
54 void Circle3::print() const
55 {
56    cout << "Center = [" << x << ", " << y << ']'
57         << "; Radius = " << radius;
58
59 } // end function print
```

Access inherited data members **x** and **y**, declared **protected** in base class **Point2**.

Outline

circletest3.cpp
(1 of 2)

```
1  // Fig. 9.16: circletest3.cpp
2  // Testing class Circle3.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10
11 using std::setprecision;
12
13 #include "circle3.h"  // Circle3 class def
14
15 int main()
16 {
17    Circle3 circle( 37, 43, 2.5 ); // instantiate Circle3 objec
18
19    // display point coordinates
20    cout << "X coordinate is " << circle.getX()
21         << "\nY coordinate is " << circle.getY()
22         << "\nRadius is " << circle.getRadius();
23
```

Create **`Circle3`** object.

Use inherited get functions to access inherited **`protected`** data.

Use **`Circle3`** get function to access **`private`** data **`radius`**.

Outline

circletest3.cpp
(2 of 2)

```
24    circle.setX( 2 );          // set new x-coordinate
25    circle.setY( 2 );          // set new y-coordinate
26    circle.setRadius( 4.25 );  // set new ra
27
28    // display new point value
29    cout << "\n\nThe new location and radius
30    circle.print();
31
32    // display floating-point values with 2 digits of precision
33    cout << fixed << setprecision( 2 );
34
35    // display Circle3's diameter
36    cout << "\nDiameter is " << circle.getDiameter();
37
38    // display Circle3's circumference
39    cout << "\nCircumference is " << circle.getCircumference();
40
41    // display Circle3's area
42    cout << "\nArea is " << circle.getArea();
43
44    cout << endl;
45
46    return 0;  // indicates successful termination
47
48 } // end main
```

Use inherited set functions to modify inherited

Use **`Circle3`** set function to modify **`private`** data **`radius`**.

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```


Outline

circletest3.cpp
output (1 of 1)

Relationship between Base Classes and Derived Classes

- Using **`protected`** data members
  - Advantages
    - Derived classes can modify values directly
    - Slight increase in performance
      - Avoid set/get function call overhead
  - Disadvantages
    - No validity checking
      - Derived class can assign illegal value
    - Implementation dependent
      - Derived class member functions more likely dependent on base class implementation
      - Base class implementation changes may result in derived class modifications
        - Fragile (brittle) software

Outline

point3.h (1 of 1)

```
1  // Fig. 9.17: point3.h
2  // Point3 class definition represents an x-y coordinate pair.
3  #ifndef POINT3_H
4  #define POINT3_H
5
6  class Point3 {
7
8  public:
9     Point3( int = 0, int = 0 ); // default constructor
10
11    void setX( int );    // set x in coordinate pair
12    int getX() const;    // return x from coordinate pair
13
14    void setY( int );    // set y in coordinate pair
15    int getY() const;    // return y from coordinate pair
16
17    void print() const;  //
18
19 private:
20    int x;  // x part of coordinate pair
21    int y;  // y part of coordinate pair
22
23 }; // end class Point3
24
25 #endif
```

Better software-engineering practice: **`private`** over **`protected`** when possible.

Outline

point3.cpp (1 of 3)

```
1  // Fig. 9.18: point3.cpp
2  // Point3 class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "point3.h"   // Point3 class definition
8
9  // default constructor
10 Point3::Point3( int xValue, int yValue )
11    : x( xValue ), y( yValue )
12 {
13    // empty body
14
15 } // end Point3 constructor
16
17 // set x in coordinate pair
18 void Point3::setX( int xValue )
19 {
20    x = xValue; // no need for validation
21
22 } // end function setX
23
```

Member initializers specify values of **`x`** and **`y`**.

Outline

point3.cpp (2 of 3)

```
24 // return x from coordinate pair
25 int Point3::getX() const
26 {
27    return x;
28
29 } // end function getX
30
31 // set y in coordinate pair
32 void Point3::setY( int yValue )
33 {
34    y = yValue; // no need for validation
35
36 } // end function setY
37
38 // return y from coordinate pair
39 int Point3::getY() const
40 {
41    return y;
42
43 } // end function getY
44
```


Outline

point3.cpp (3 of 3)

```
45 // output Point3 object
46 void Point3::print() const
47 {
48    cout << '[' << getX() << ", " << getY() << ']';
49
50 } // end function print
```

Invoke non-**private** member functions to access **private** data.

Outline

circle4.h (1 of 2)

```
1  // Fig. 9.19: circle4.h
2  // Circle4 class contains x-y coordinate pair and radius.
3  #ifndef CIRCLE4_H
4  #define CIRCLE4_H
5
6  #include "point3.h"  // Point3 class definition
7
8  class Circle4 : public Point3 {
9
10 public:
11
12    // default constructor
13    Circle4( int = 0, int = 0, double = 0.0 );
14
15    void setRadius( double );   // set radius
16    double getRadius() const;   // return radius
17
18    double getDiameter() const;       // return diameter
19    double getCircumference() const;  // return circumference
20    double getArea() const;           // return area
21
22    void print() const;               // output
23
24 private:
25    double radius; // Circle4's radius
```

Class **Circle4** inherits from class **Point3**.

Maintain **private** data member **radius**.

Outline

circle4.h (2 of 2)

```
26
27 }; // end class Circle4
28
29 #endif
```

Outline

circle4.cpp (1 of 3)

```
1  // Fig. 9.20: circle4.cpp
2  // Circle4 class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "circle4.h"   // Circle4 cl
8
9  // default constructor
10 Circle4::Circle4( int xValue, int yV
11    : Point3( xValue, yValue )  // call base-class constructor
12 {
13    setRadius( radiusValue );
14
15 } // end Circle4 constructor
16
17 // set radius
18 void Circle4::setRadius( double radiusValue )
19 {
20    radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
21
22 } // end function setRadius
23
```

Base-class initializer syntax passes arguments to base class **`Point3`**.

Outline

circle4.cpp (2 of 3)

```
24 // return radius
25 double Circle4::getRadius() const
26 {
27    return radius;
28
29 } // end function getRadius
30
31 // calculate and return diameter
32 double Circle4::getDiameter() const
33 {
34    return 2 * getRadius();
35
36 } // end function getDiameter
37
38 // calculate and return circumference
39 double Circle4::getCircumference() const
40 {
41    return 3.14159 * getDiameter();
42
43 } // end function getCircumference
44
```

Invoke function **`getRadius`** rather than directly accessing data member **`radius`**.

Outline

circle4.cpp (3 of 3)

```
45 // calculate and return area
46 double Circle4::getArea() const
47 {
48    return 3.14159 * getRadius() * getRadius();
49
50 } // end function getArea
51
52 // output Circle4 object
53 void Circle4::print() const
54 {
55    cout << "Center = ";
56    Point3::print();       // invoke P
57    cout << "; Radius = " << getRadius();
58
59 } // end function print
```

Redefine class **Point3**'s member function **print**.

Invoke function **getRadius**

Invoke base-class **Point3**'s **print** function using binary scope-resolution operator (**::**).

Outline

circletest4.cpp
(1 of 2)

```
1  // Fig. 9.21: circletest4.cpp
2  // Testing class Circle4.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10
11 using std::setprecision;
12
13 #include "circle4.h"  // Circle4 class def
14
15 int main()
16 {
17    Circle4 circle( 37, 43, 2.5 ); // instantiate Circle4 objec
18
19    // display point coordinates
20    cout << "X coordinate is " << circle.getX()
21         << "\nY coordinate is " << circle.getY()
22         << "\nRadius is " << circle.getRadius();
23
```

Create **Circle4** object.

Use inherited get functions to access inherited **protected**

Use **Circle3** get function to access **private** data **radius**.

```
24     circle.setX( 2 );              // set new x-coordinate
25     circle.setY( 2 );              // set new y-coordinate
26     circle.setRadius( 4.25 );      // set new ra
27
28     // display new circle value
29     cout << "\n\nThe new location and radius
30     circle.print();
31
32     // display floating-point values with 2 digits of precision
33     cout << fixed << setprecision( 2 );
34
35     // display Circle4's diameter
36     cout << "\nDiameter is " << circle.getDiameter();
37
38     // display Circle4's circumference
39     cout << "\nCircumference is " << circle.getCircumference();
40
41     // display Circle4's area
42     cout << "\nArea is " << circle.getArea();
43
44     cout << endl;
45
46     return 0;  // indicates successful termination
47
48  } // end main
```

Use inherited set functions to modify inherited

Use **`Circle3`** set function to modify **`private`** data **`radius`**.


Outline

circletest4.cpp
(2 of 2)


```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```


Outline

circletest4.cpp
output (1 of 1)

Case Study: Three-Level Inheritance Hierarchy

- Three level point/circle/cylinder hierarchy
  - Point
    - x-y coordinate pair
  - Circle
    - x-y coordinate pair
    - Radius
  - Cylinder
    - x-y coordinate pair
    - Radius
    - Height

Outline

cylinder.h (1 of 2)

```
1  // Fig. 9.22: cylinder.h
2  // Cylinder class inherits from class Circle4.
3  #ifndef CYLINDER_H
4  #define CYLINDER_H
5
6  #include "circle4.h"  // Circle4
7
8  class Cylinder : public Circle4 {
9
10 public:
11
12    // default constructor
13    Cylinder( int = 0, int = 0, double = 0.0, double = 0.0 );
14
15    void setHeight( double );  // set Cylinder's height
16    double getHeight() const;  // return Cylinder's height
17
18    double getArea() const;    // return Cylinder's area
19    double getVolume() const;  // return Cylinder's volume
20    void print() const;        // o
21
22 private:
23    double height; // Cylinder's height
24
25 }; // end class Cylinder
```

Class **Cylinder** inherits from class **Circle4**.

Maintain **private** data member **height**.

Outline

cylinder.h (2 of 2)

cylinder.cpp
(1 of 3)

```
26
27 #endif
```

```
1  // Fig. 9.23: cylinder.cpp
2  // Cylinder class inherits from class Circle4.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "cylinder.h"   // Cylinder class definition
8
9  // default constructor
10 Cylinder::Cylinder( int xValue, int yValue
11   double heightValue )
12   : Circle4( xValue, yValue, radiusValue )
13 {
14    setHeight( heightValue );
15
16 } // end Cylinder constructor
17
```

Base-class initializer syntax passes arguments to base class **`Circle4`**.

Outline

cylinder.cpp
(2 of 3)

```
18 // set Cylinder's height
19 void Cylinder::setHeight( double heightValue )
20 {
21    height = ( heightValue < 0.0 ? 0.0 : heightValue );
22
23 } // end function setHeight
24
25 // get Cylinder's height
26 double Cylinder::getHeight() const
27 {
28    return height;
29
30 } // end function getHeight
31
32 // redefine Circle4 function getArea to
33 double Cylinder::getArea() const
34 {
35    return 2 * Circle4::getArea() +
36       getCircumference() * getHeight();
37
38 } // end function getArea
39
```

Redefine base class
er function
e area.

Invoke base-class **`Circle4`**'s **`getArea`** function using binary scope-resolution operator (**`::`**).

Outline

cylinder.cpp
(3 of 3)

```
40 // calculate Cylinder volume
41 double Cylinder::getVolume() const
42 {
43    return Circle4::getArea() * getHeight()
44
45 } // end function getVolume
46
47 // output Cylinder object
48 void Cylinder::print() const
49 {
50    Circle4::print();
51    cout << "; Height = " << getHeight();
52
53 } // end function print
```

Invoke base-class **`Circle4`**'s **`getArea`** function using binary scope-resolution operator (**`::`**).

Redefine class **`Circle4`**'s **`int`**.

Invoke base-class **`Circle4`**'s **`print`** function using binary scope-resolution operator (**`::`**).

Outline

cylindertest.cpp
(1 of 3)

```
1  // Fig. 9.24: cylindertest.cpp
2  // Testing class Cylinder.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10
11 using std::setprecision;
12
13 #include "cylinder.h"  // Cylinder class definition
14
15 int main()
16 {
17    // instantiate Cylinder object
18    Cylinder cylinder( 12, 23, 2.5, 5.7 );
19
20    // display point coordinates
21    cout << "X coordinate is " << cylinder.getX()
22         << "\nY coordinate is " << cylinder.getY()
23         << "\nRadius is " << cylinder.getRadius()
24         << "\nHeight is " << cylinder.getHeight();
25
```

Invoke indirectly inherited **`Point3`** member functions

Invoke **`Cylinder`** member function.

Outline

cylindertest.cpp
(2 of 3)

```
26     cylinder.setX( 2 );          // set new x-coordinate
27     cylinder.setY( 2 );          // set new
28     cylinder.setRadius( 4.25 ); // set new
29     cylinder.setHeight( 10 );    // set new
30
31     // display new cylinder value
32     cout << "\n\nThe new location and radius
33     cylinder.print();
34
35     // display floating-point values
36     cout << fixed << setprecision( 2
37
38     // display cylinder's diameter
39     cout << "\n\nDiameter is " << cylinder.getDiameter();
40
41     // display cylinder's circumference
42     cout << "\nCircumference is "
43          << cylinder.getCircumference();
44
45     // display cylinder's area
46     cout << "\nArea is " << cylinder.getArea();
47
48     // display cylinder's volume
49     cout << "\nVolume is " << cylinder.getVolume();
50
```

Invoke indirectly inherited Point3 member functions

Invoke directly inherited

Invoke **Cylinder** member function.

Invoke redefined **print** function.

Invoke redefined **getArea** function.

Outline

cylindertest.cpp
(3 of 3)

```
51     cout << endl;
52
53     return 0;  // indicates successful termination
54
55  } // end main
```

cylindertest.cpp
output (1 of 1)

```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25; Height = 10

Diameter is 8.50
Circumference is 26.70
Area is 380.53
Volume is 567.45
```

Software Engineering with Inheritance

- Customizing existing software
  - Inherit from existing classes
    - Include additional members
    - Redefine base-class members
    - No direct access to base class's source code
      - Link to object code
  - Independent software vendors (ISVs)
    - Develop proprietary code for sale/license
      - Available in object-code format
    - Users derive new classes
      - Without accessing ISV proprietary source code