Chapter 2
Object-Oriented Design (OOD) and C++
Objectives (cont’d.)

• Explore three types of inheritance: public, protected, and private
• Learn about composition
• Become familiar with the three basic principles of object-oriented design
• Learn about overloading
• Become aware of the restrictions on operator overloading
Objectives (cont’d.)

• Examine the pointer this
• Learn about friend functions
• Explore the members and nonmembers of a class
• Discover how to overload various operators
• Learn about templates
• Explore how to construct function templates and class templates
Inheritance

• An “is-a” relationship
  – Example: “every employee is a person”

• Allows new class creation from existing classes
  – Base class: the existing class
  – Derived class: new class created from existing classes
    • Inherits base classes’ properties
    • Reduces software complexity
    • Becomes base class for future derived class

• Inheritance types
  – Single inheritance and multiple inheritance
Inheritance (cont’d.)

- Viewed as treelike or hierarchical
  - Base class shown with its derived classes
- Derived class general syntax
  - No `memberAccessSpecifier` specified
  - Assume `private` inheritance

```cpp
class className: memberAccessSpecifier baseClassName {
  member list
};
```

FIGURE 2-1
Inheritance hierarchy
Inheritance (cont’d.)

• Facts to keep in mind
  – private base class members
    • private to the base class
  – public base class member inheritance
    • public members or private members

– Derived class
  • Can include additional members
  • Can redefine public member base class functions

– All base class member variables
  • Derived class member variables
Redefining (Overriding) Member Functions of the Base Class

• Base class public member function included in a derived class
  – Same name, number, and types of parameters as base class member function

• Function overloading
  – Same name for base class functions and derived class functions
  – Different sets of parameters
Constructors of Derived and Base Classes

• Derived class with own private member variables
  – Explicitly includes its own constructors

• Constructors
  – Initialize member variables

• Declared derived class object inherits base class members
  – Cannot directly access private base class data
  – Same is true for derived class member functions
Constructors of Derived and Base Classes (cont’d.)

• Derived class constructors can only directly initialize inherited members (public data)
• Derived class object must automatically execute base class constructor
  – Triggers base class constructor execution
  – Call to base class constructor specified in heading of derived class constructor definition
Constructors of Derived and Base Classes (cont’d.)

- **Example:** 
  ```
  class rectangleType 
  
  contains default constructor
  
  Does not specify any constructor of the class
  ```

- **boxType**
  ```
  boxType::boxType ()
  
  { height = 0.0; }
  ```

```cpp
boxType::boxType (double l, double w, double h) : rectangleType (l, w) {
  if (h >= 0)
    height = h;
  else
    height = 0;
}
```
Constructors of Derived and Base Classes (cont’d.)

- Consider the following statements

```cpp
rectangleType myRectangle(5.0, 3.0);  //Line 1
boxType myBox(6.0, 5.0, 4.0);          //Line 2
myRectangle.print();                   //Line 3
cout << endl;                          //Line 4
myBox.print();                         //Line 5
cout << endl;                          //Line 6
```

![Diagram showing objects myRectangle and myBox with their properties]
Header File of a Derived Class

• Required to define new classes
• Base class already defined
  – Header files contain base class definitions
• New class header files contain commands
  – Tell computer where to look for base classes’ definitions
Multiple Inclusions of a Header File

- Preprocessor command `include`  
  - Used to include header file in a program
- Preprocessor processes the program  
  - Before program compiled
- Avoid multiple inclusions of a file in a program  
  - Use preprocessor commands in the header file
Multiple Inclusions of a Header File (cont’d.)

- Preprocessor commands and meaning

```c++
//Header file test.h

#define H_test

const int ONE = 1;
const int TWO = 2;

#ifndef H_test
#define H_test
#endif

  a.  #ifndef H_test means “if not defined H_test”
  b.  #define H_test means “define H_test”
  c.  #endif means “end if”

Here H_test is a preprocessor identifier.
```
Protected Members of a Class

- **private class members**
  - private to the class
  - Cannot be directly accessed outside the class
  - Derived class cannot access private members

- **Solution: make private member public**
  - Problem: anyone can access that member

- **Solution: declare member as protected**
  - Derived class member allowed access
  - Prevents direct access outside the class
Inheritance as *public*, *protected*, or *private*

- Consider the following statement
  - MemberAccessSpecifier: *public*, *protected*, or *private*

```cpp
class B: memberAccessSpecifier A {
    
    
};
```
Inheritance as `public`, `protected`, or `private` (cont’d.)

- **public** `MemberAccessSpecifier`
  - `public` members of A, `public` members of B: directly accessed in class B
  - `protected` members of A, `protected` members of B: can be directly accessed by B member functions and friend functions
  - `private` members of A, hidden to B: can be accessed by B member functions and friend functions through `public` or `protected` members of A
Inheritance as **public**, **protected**, or **private** (cont’d.)

- **protected** MemberAccessSpecifier
  - **public** members of A, **protected** members of B: can be accessed by B member functions and **friend** functions
  - **protected** members of A, **protected** members of B: can be accessed by B member functions and **friend** functions
  - **private** members of A hidden to B: can be accessed by B member functions and **friend** functions through the **public** or **protected** members of A
Inheritance as public, protected, or private (cont’d.)

- **private** MemberAccessSpecifier
  - public members of A, private members of B: can be accessed by B member functions and friend functions
  - protected members of A, private members of B: can be accessed by B member functions and friend functions
  - private members of A, hidden to B: can be accessed by B member functions and friend functions through the public or protected members of A
Composition

• Another way to relate two classes
• One or more class members
  – Another class type object
• Is a “has-a” relationship
  – Example: “every person has a date of birth”
Composition (cont’d.)

**FIGURE 2-6** UML class diagram of the class `dateType`

**FIGURE 2-7** UML class diagram of the class `personalInfoType` and composition (aggregation)
Polymorphism: Operator and Function Overloading

• Encapsulation
  – Ability to combine data and operations
  – Object-oriented design (OOD) first principle

• Inheritance
  – OOD second principle
  – Encourages code reuse

• Polymorphism
  – OOD third principle
  – Occurs through operator overloading and templates
    • Function templates simplify template function overloading
Operator Overloading

• Why operator overloading is needed
  – Built-in operations on classes
    • Assignment operator and member selection operator
    • Other operators cannot be directly applied to class objects
  – Operator overloading
    • Programmer extends most operation definitions
    • Relational operators, arithmetic operators, insertion operators for data output, and extraction operators for data input applied to classes
Operator Overloading (cont’d.)

- **Examples**
  - Stream insertion operator (<<), stream extraction operator(>>, +, and –

- **Advantage**
  - Operators work effectively in specific applications

- **C++ does not allow user to create new operators**

- **Overload an operator**
  - Write functions (header and body)
  - Function name overloading an operator: reserved word operator followed by operator to be overloaded
Operator Overloading (cont’d.)

• Overload an operator
  – Write functions (header and body)
  – Function name overloading an operator: reserved word operator followed by operator to be overloaded

• Example: operator >=
  – Function name: operator>=

• Operator function
  – Function overloading an operator
Syntax for Operator Functions

- Result of an operation: value
  - Operator function: value-returning function
- Operator: reserved word
- Overloading an operator for a class
  - Include statement to declare the function to overload the operator in class definition
  - Write operator function definition
- Operator function heading syntax

```
returnType operator operatorSymbol(arguments)
```
Overloading an Operator: Some Restrictions

• Cannot change operator precedence
• Cannot change associativity
  – Example: arithmetic operator + goes from left to right and cannot be changed
• Cannot use default arguments with an overloaded operator
• Cannot change number of arguments an operator takes
Overloading an Operator: Some Restrictions (cont’d.)

• Cannot create new operators
• Some operators cannot be overloaded
  .  .*  ::  ?:  sizeof
• How an operator works with built-in types remains the same
• Operators can be overloaded
  – For objects of the user-defined type
  – For combination of objects of the user-defined type and objects of the built-in type
The Pointer \textit{this}

- Sometimes necessary to refer to object as a whole
  - Rather than object’s individual data members
- Object's hidden pointer to itself
- C++ reserved word
- Available for use
- When object invokes member function
  - Member function references object’s pointer \textit{this}
Friend Functions of Classes

• A nonmember function of a class
  – Has access to all class members (public or non-public)

• Making function as a friend of a class
  – Reserved word friend precedes function prototype (in the class definition)

• Word friend appears only in function prototype in the class definition
  • Not in friend function definition
Friend Functions of Classes (cont’d.)

- Definition of a friend function
  - Class name, scope resolution operator do not precede name of friend function in the function heading
  - Word friend does not appear in friend function’s definition heading
Operator Functions as Member Functions and Nonmember Functions

• Two rules when including operator function in a class definition
  – Function overloading operators ( ), [ ], –>, or = for a class
    • Must be declared as a class member
Member Functions and Nonmember Functions (cont’d.)

- Two rules when including operator function in a class definition (cont’d.)
  - Suppose operator \( \text{op} \) overloaded for class \( \text{opOverClass} \)
    - If leftmost operand of \( \text{op} \) is an object of a different type:
      - Function overloading operator \( \text{op} \) for \( \text{opOverClass} \) must be a nonmember (friend of class \( \text{opOverClass} \) )
    - If operator function overloading operator \( \text{op} \) for class \( \text{opOverClass} \) is a member of the class \( \text{opOverClass} \):
      - When applying \( \text{op} \) on objects of type \( \text{opOverClass} \), leftmost operand of \( \text{op} \) must be of type \( \text{opOverClass} \)
Member Functions and Nonmember Functions (cont’d.)

• Functions overloading insertion operator (<<) and extraction operator (>>) for a class
  – Must be nonmembers
• Operators can be overloaded as
  – Member functions or nonmember functions
  – Except for exceptions noted earlier
• C++ consists of binary and unary operators
• C++ contains a ternary operator
  – Cannot be overloaded
Overloading Binary Operators

• Two ways to overload
  – As a member function of a class
  – As a friend function

• As member functions
  – General syntax

Function Prototype (to be included in the definition of the class):
returnType operator#(const className&) const;
Overloading Binary Operators (cont’d.)

• As member functions (cont’d.)
  – Function definition

```cpp
returnType className::operator# 
      (const className& otherObject) const 
{
    //algorithm to perform the operation

    return value;
}
```
Overloading Binary Operators (cont’d.)

• As nonmember functions
  – General syntax

Function Prototype (to be included in the definition of the class):

friend returnType operator#(const className&, const className&);
Overloading Binary Operators (cont’d.)

• As nonmember functions (cont’d.)
  – Function definition

```cpp
returnType operator#(const className& firstObject,
                     const className& secondObject)
{
    //algorithm to perform the operation

    return value;
}
```
Overloading the Stream Insertion (<<) and Extraction (>>) Operators

• Operator function overloading insertion operator and extraction operator for a class
  – Must be nonmember function of that class

• Overloading the stream extraction operator (>>)
  – General syntax
Overloading the Stream Insertion (<<) and Extraction (>>) Operators (cont’d.)

- Overloading the stream extraction operator (>>)
  - General syntax and function definition

```cpp
Function Prototype (to be included in the definition of the class):
friend returnType operator>>(const className&, const className&);

Function Definition:
returnType operator>>(const className& firstObject,
const className& secondObject)
{
    //algorithm to perform the operation
    return value;
}
```
Overloading the Stream Insertion (<<) and Extraction (>>) Operators (cont’d.)

• Overloading unary operations
  – Similar to process for overloading binary operators
  – Difference: unary operator has only one argument

• Process for overloading unary operators
  – If operator function is a member of the class: it has no parameters
  – If operator function is a nonmember (friend function of the class): it has one parameter
Operator Overloading: Member Versus Nonmember

- Certain operators can be overloaded as
  - Member functions or nonmember functions
- Example: binary arithmetic operator +
  - As a member function
    - Operator + has direct access to data members
    - Need to pass only one object as a parameter
  - As a nonmember function
    - Must pass both objects as parameters
    - Could require additional memory and computer time
- Recommendation for efficiency
  - Overload operators as member functions
Function Overloading

• Creation of several functions with the same name
  – All must have different parameter set
    • Parameter types determine which function to execute
  – Must give the definition of each function
  – Example: original code and modified code with function overloading

```c++
int largerInt(int x, int y);
char largerChar(char first, char second);
double largerDouble(double u, double v);
string largerString(string first, string second);

int larger(int x, int y);
char larger(char first, char second);
double larger(double u, double v);
string larger(string first, string second);
```
Templates

• Function template
  – Writing a single code segment for a set of related functions

• Class template
  – Writing a single code segment for a set of related classes

• Syntax
  – Data types: parameters to templates

\[
\text{template <class Type> declaration;}
\]
Function Templates

• Simplifies process of overloading functions
• Syntax and example

```cpp
template <class Type>
function definition;

template <class Type>
Type larger(Type x, Type y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```
Class Templates

• Used to write a single code segment for a set of related classes
• Called parameterized types
  – Specific class generated based on parameter type
• Syntax and example

```cpp
template <class elemType>
class listType
{
public:
    bool isEmpty();
    bool isFull();
    void search(const elemType& searchItem, bool& found);
    void insert(const elemType& newElement);
    void remove(const elemType& removeElement);
    void destroyList();
    void printList();
    listType();
}

private:
    elemType list[100]; //array to hold the list elements
    int length; //variable to store the number
    //of elements in the list
};
```
Header File and Implementation File of a Class Template

• Not possible to compile implementation file independently of client code

• Solution
  – Put class definition and definitions of the function templates directly in client code
  – Put class definition and definitions of the function templates together in same header file
  – Put class definition and definitions of the functions in separate files (as usual): include directive to implementation file at end of header file
Summary

- Inheritance and composition
  - Ways to relate two or more classes
  - Single and multiple inheritance
  - Inheritance: an “is a” relationship
  - Composition: a “has a” relationship

- Private members of a base class are private to the base class
  - Derived class cannot directly access them

- Public members of a base class can be inherited either as public, protected, and private by the derived class
Summary (cont’d.)

• Three basic principles of OOD
  – Encapsulation, inheritance, and polymorphism
• Operator overloading
  – Operator has different meanings with different data types
  – Operator function: function overloading an operator
• friend function: nonmember of a class
• Function name can be overloaded
• Templates
  – Write a single code segment for a set of related functions or classes