# Data Structures Using C++ 2E

*Chapter 2*

*Object-Oriented Design (OOD) and C++*

# Objectives

- Learn about inheritance
- Learn about derived and base classes
- Explore how to redefine the member functions of a base class
- Examine how the constructors of base and derived classes work
- Learn how to construct the header file of a derived class
- Explore three types of inheritance: `public`, `protected`, and `private`
- Learn about composition

# Objectives (cont'd.)

- Become familiar with the three basic principles of object-oriented design
- Learn about overloading
- Become aware of the restrictions on operator overloading
- 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.)

- Inheritance types
  - Single inheritance and multiple inheritance
- Viewed as treelike or hierarchical
  - Base class shown with its derived classes
- Derived class general syntax
  - No memberAccessSpecifier specified ➔ private

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

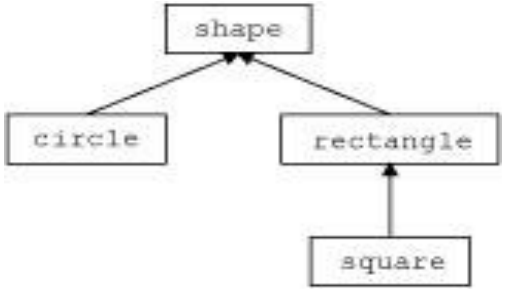



**FIGURE 2-1**
Inheritance hierarchy

# Inheritance (cont'd.)

- Facts to keep in mind
  - `private` base class members: private to base class
    - Inaccessible to derived class
  - `public` base class member inheritance
    - `public` or `private` members of derived class
  - Derived class
    - Can include additional members
    - Can redefine (override) `public` member base class functions
  - All base class member variables/functions
    - Derived class member variables/functions

# Redefining (Overriding) Member Functions of the Base Class

- Override public member function of base class in a derived class
  - Same name, number, and types of parameters as base class member function
  - E.g., `print()` in the class `BoxType`

vs

- Function overloading
  - Same name for base class functions and derived class functions
  - Different sets of parameters
  - Also allowed in class

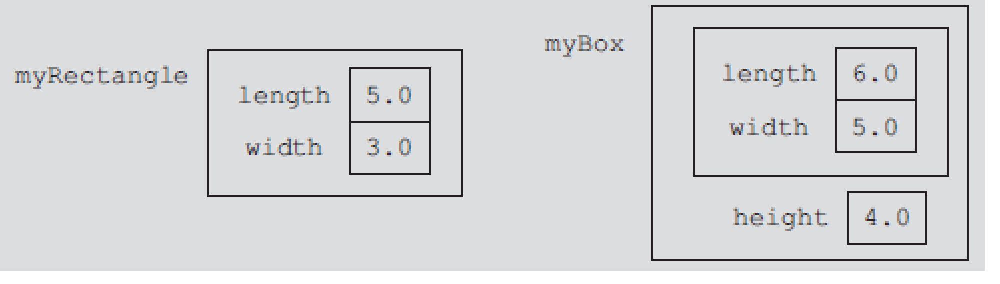
It is legal to declare a data member of the same name in a derived class as the base class; Only derive class will be able to use it.  Note there is only virtual function, no virtual variable.

```
class rectangleType
{
public:
    void setDimension(double l, double w);
      //Function to set the length and width of the rectangle.
      //Postcondition: length = l; width = w;

    double getLength() const;
      //Function to return the length of the rectangle.
      //Postcondition: The value of length is returned.

    double getWidth() const;
      //Function to return the width of the rectangle.
      //Postcondition: The value of width is returned.

    double area() const;
      //Function to return the area of the rectangle.
      //Postcondition: The area of the rectangle is calculated
      //    and returned.

    double perimeter() const;
      //Function to return the perimeter of the rectangle.
      //Postcondition: The perimeter of the rectangle is
      //    calculated and returned.

    void print() const;
      //Function to output the length and width of the rectangle.

    rectangleType();
      //default constructor
      //Postcondition: length = 0; width = 0;

    rectangleType(double l, double w);
      //constructor with parameters
      //Postcondition: length = l; width = w;

private:
    double length;
    double width;
};
```

```
class boxType: public rectangleType
{
public:
    void setDimension(double l, double w, double h);
      //Function to set the length, width, and height of the box.
      //Postcondition: length = l; width = w; height = h;

    double getHeight() const;
      //Function to return the height of the box.
      //Postcondition: The value of height is returned.

    double area() const;
      //Function to return the surface area of the box.
      //Postcondition: The surface area of the box is
      //    calculated and returned.

    double volume() const;
      //Function to return the volume of the box.
      //Postcondition: The volume of the box is calculated and
      //    returned.

    void print() const;
      //Function to output the length, width, and height of a box.

    boxType();
      //Default constructor
      //Postcondition: length = 0; width = 0; height = 0;

    boxType(double l, double w, double h);
      //Constructor with parameters
      //Postcondition: length = l; width = w; height = h;

private:
    double height;
};
```

# 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`
  - Constructor of `class boxType` will be called first
- Write the definitions of constructors with parameters

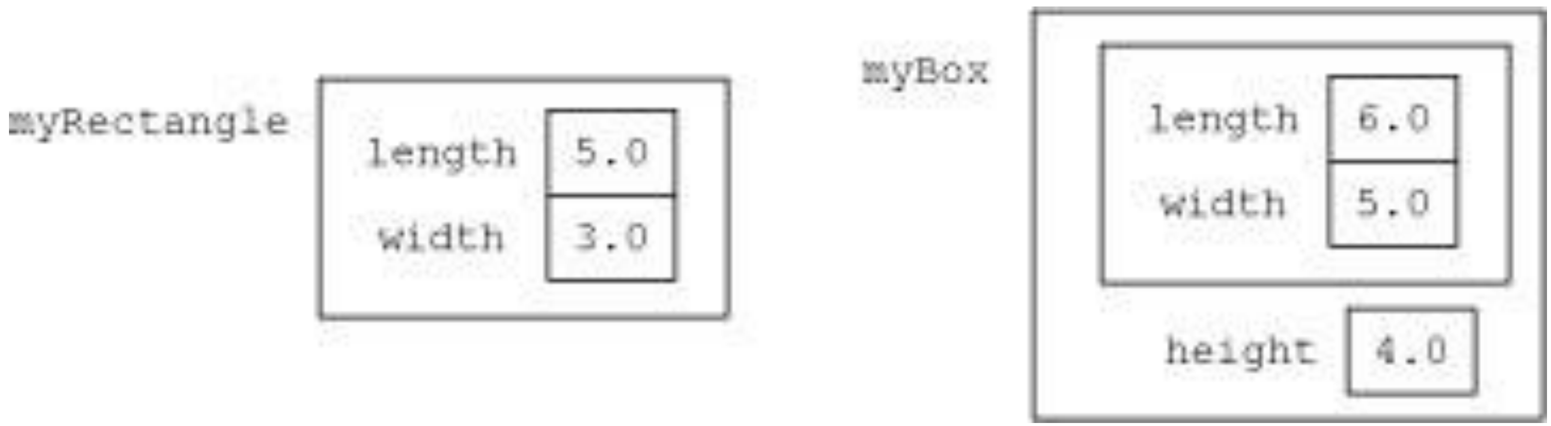
```
boxType::boxType()
{
    height = 0.0;
}
```

```
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

```
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
```



```
void rectangleType::print() const
{
    cout << "Length = "  << length
         << "; Width = " << width;
}

 void boxType::print() const
 {
     rectangleType::print();
     cout << "; Height = " << height;
 }
```

# 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
- baseClass.h, baseClass.cpp
- derivedClass.h, derivedClass.cpp
  – derivedClass.h: include baseClass.h
- main/driver program: include derivedClass.h

# 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

```
//Header file test.h

#ifndef H_test
#define H_test
const int ONE = 1;
const int TWO = 2;
#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

# `public` Inheritance

```
class B: public A
{
…
};
```

- `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 *indirectly* accessed by B member functions and `friend` functions through `public` or `protected` members of A

# `protected` Inheritance

```
class B: protected A
{
…
};
```

- `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 *indirectly* accessed by B member functions and `friend` functions through the `public` or `protected` members of A

# `private` Inheritance

```
class B: private A
{
…
};
```

- `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 *indirectly* 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 of class A is another class type B
- Is a ‘‘has-a’’ relationship
  – Example: ‘‘every person has a date of birth’’

# Composition (cont'd.)

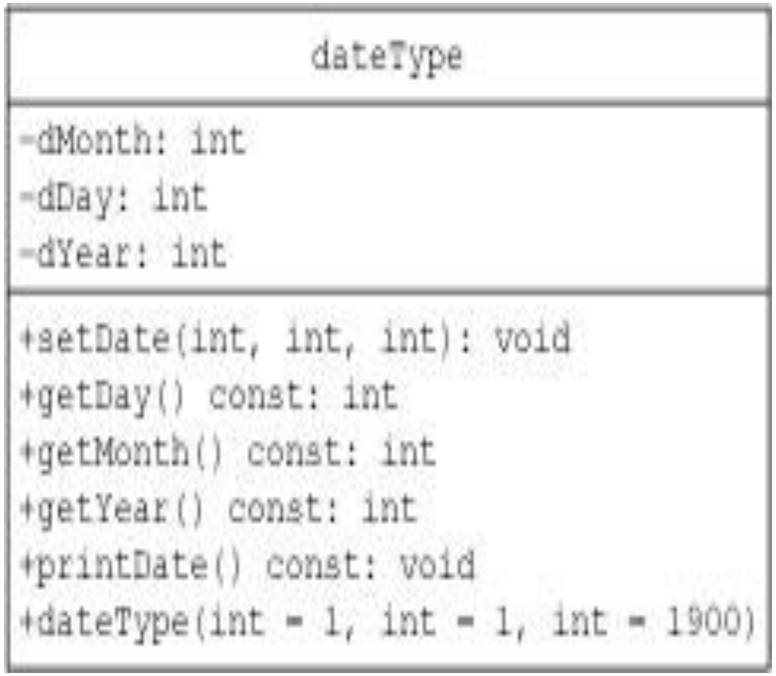



**FIGURE 2-6** UML class diagram of the `class dateType`

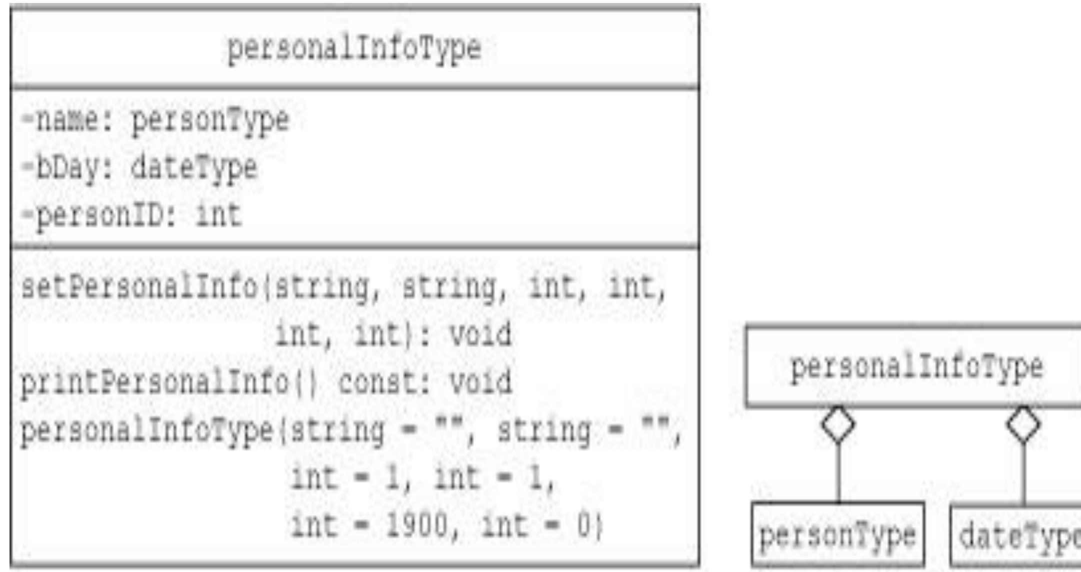



**FIGURE 2-7** UML class diagram of the `class personalInfoType` and composition (aggregation)

# Composition vs Aggregation

- Composition
  - One obj controls the lifetime of the other obj
  - Implies ownership
  - when the owning object is destroyed, so are the contained objects
  - E.g. School and departments

- Aggregation
  - One obj not necessarily control the lifetime of the other obj
  - Does not imply ownership
  - when the owning object is destroyed, the other object is not necessarily destroyed
  - E.g. Department and professors

# Composition vs Aggregation (cont'd)

- Composition
  - Typically use normal member variables
  - Can use pointer values if the composition class automatically handles allocation/deallocation
  - Responsible for creation/destruction of subclasses
- Aggregation
  - Typically use pointer variables pointing to an object that lives outside the scope of the aggregate class
  - Can use reference values that point to an object that lives outside the scope of the aggregate class
  - Not responsible for creating/destroying subclasses

# Constructor & destructor

```
#include <iostream>
class Base
{public:
  Base() {cout<<"Constructing Base\n";}
  ~Base(){cout<<"Destroying Base\n";}
};
class Derive: public Base
{public:
 Derive(){cout<<"Constructing Derive\n";}
 ~Derive(){cout<<"Destroying Derive\n";}
};
void main() {
   Derive a;
}
```

Constructing Base
Constructing Derive
Destroying Derive
Destroying Base

```
#include <iostream>
class X
{public:
  X() {cout<<"Constructing X\n";}
  ~X(){cout<<"Destroying X\n";}
};
class Base
{public:
  Base() {cout<<"Constructing Base\n";}
  ~Base(){cout<<"Destroying Base\n";}
  X objX;
};
class Derive: public Base
{public:
 Derive(){cout<<"Constructing Derive\n";}
 ~Derive(){cout<<"Destroying Derive\n";}
};
void main() {
   Derive a;
}
```

Constructing X
Constructing Base
Constructing Derive
Destroying Derive
Destroying Base
Destroying X

# Polymorphism: Operator and Function Overloading

- Encapsulation
  - Ability to combine data and operations
  - Object-oriented design (OOD) first principle
- Inheritance
  - Encourages code reuse
- Polymorphism
  - 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
- Examples
  - Stream insertion operator (<<), stream extraction operator(>>), +, and –

# Operator Overloading (cont'd.)

- 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
  - E.g., Function name: `operator>=`

# Syntax for Operator Functions

- Operator function
  - Function overloading an operator
  - Result of an operation: value
  - Operator function: value-returning function
- 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
```

> **a.*b**
> Dereference a pointer to class member
> a is of class T; b is a pointer to a member in class T

- 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 `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 `this`

# Friend Functions of Classes

- A nonmember function of a class
  - Has access to *all* class members (`public` , `protected`, `private`)
- Making function as a `friend` of a class
  - Reserved word `friend` precedes function prototype (in the class definition)
  - Reserved word not in `friend` function definition/implementation
  - Class name, scope resolution operator do not precede name of friend function in the function heading

# Friend Functions of Classes (cont'd.)

```
class a
{
  friend void friendFunc(/*parameter*/);
}
void friendFunc(/*parameter*/)
{ /* can access private members of class a */
  …
}
```

# Member Functions and Nonmember Functions (cont'd.)

- Rules for operator function
  1. Function overloading operators `()`, `[]`, `->`, or `=` for a class must be class member function
  2. Suppose operator `op` (such as +, >>) is overloaded for `class opOverClass`
- If leftmost operand of `op` is of different type ➔

  Function overloading operator `op` for `opOverClass` must be a nonmember (friend of `class opOverClass`)

- If operator function overloading operator `op` for class `opOverClass` is a member of the class `opOverClass` ➔

  When applying `op` on objects of type `opOverClass`, leftmost operand of `op` must be of type `opOverClass`

# Member Functions and Nonmember Functions (cont'd.)

- Functions overloading insertion operator (<<) and extraction operator (>>) for a class
  - Must be nonmembers
    - `cin` is an `istream` obj, `cout` is an `ostream` obj
- 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, (`condition? a : b`)
  - 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;
```

  - Function definition – one formal parameter

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

    return value;
}
```

# Overloading Binary Operators (cont'd.)

- As friend/nonmember functions
  - General syntax

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

```
friend returnType operator#(const className&, const className&);
```

  - Function definition – two formal parameters

```
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 Insertion (<<) and Extraction (>>) Operators (cont'd.)

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

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

```
friend istream& operator>>(istream&, className&);
```

**Function Definition:**

```
istream& operator>>(istream& isObject, className& cObject)
{
      //local declaration, if any
      //Read the data into cObject.
      //isObject >> . . .

      //Return the stream object.
    return isObject;
}
```

```cpp
#include <iostream>

using namespace std;

class rectangleType
{
    //Overload the stream insertion and extraction operators
    friend ostream& operator<< (ostream&, const rectangleType &);
    friend istream& operator>> (istream&, rectangleType &);

public:
    void setDimension(double l, double w);
    double getLength() const;
    double getWidth() const;
    double area() const;
    double perimeter() const;
    void print() const;

    rectangleType operator+(const rectangleType&) const;
      //Overload the operator +
    rectangleType operator*(const rectangleType&) const;
      //Overload the operator *

    bool operator==(const rectangleType&) const;
      //Overload the operator ==
    bool operator!=(const rectangleType&) const;
      //Overload the operator !=

    rectangleType();
    rectangleType(double l, double w);

private:
    double length;
    double width;
};
```

```
ostream& operator<< (ostream& osObject,
                     const rectangleType& rectangle)
{
    osObject << "Length = "  << rectangle.length
             << "; Width = " << rectangle.width;

    return osObject;
}

istream& operator>> (istream& isObject,
                     rectangleType& rectangle)
{
    isObject >> rectangle.length >> rectangle.width;

    return isObject;
}
```

```
rectangleType rectangleType::operator+
                          (const rectangleType& rectangle) const
{
    rectangleType tempRect;

    tempRect.length = length + rectangle.length;
    tempRect.width = width + rectangle.width;

    return tempRect;
}
```

RectangleA + RectangleB → RectangleA.operator+(RectangleB)

```
rectangleType rectangleType::operator*
                          (const rectangleType& rectangle) const
{
    rectangleType tempRect;

    tempRect.length = length * rectangle.length;
    tempRect.width = width * rectangle.width;

    return tempRect;
}
```

RectangleA * RectangleB → RectangleA.operator*(RectangleB)

# Overloading Unary Operations

- 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 (right operand) 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

```
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

```
template <class Type>
declaration;
```

# Function Templates

- Writing a single code segment for a set of related functions
- Simplifies process of overloading functions
- Syntax and example

```
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

```
template <class Type>
class declaration
```

```
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 (recommended)
  - 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
- Inheritance
  - Public
  - Protected
  - Private
  - How this affect private/protected/public member variables in base class?

# Summary (cont'd.)

- Three basic principles of OOD
  - Encapsulation, inheritance, and polymorphism
- Operator overloading
  - Either member function or non-member function
  - Some have to be overloaded as member functions
  - Some have to be overloaded as friend/nonmember functions
- `friend` function: nonmember of a class
- Function overloading
- Templates
  - Write a single code segment for a set of related functions or classes
  - In .h file; no .cpp file (In contract, both .h and .cpp for a class)
  - Template parameter(s)

# Self Exercises

- Programming Exercises: 5, 6, 7, 12, 18, 19