Chapter 2

Object-Oriented Design (OOD) and C++

Chapter 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

Chapter Objectives

- Explore three types of inheritance: public, private, and protected
- Learn about composition
- Become familiar with the three basic principles of object-oriented design
- Learn about overloading
- Learn the restrictions on operator overloading

Chapter Objectives

- 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

- “is-a” relationship
- Single inheritance
  - New class derived from one existing class (base class)
- Multiple inheritance
  - New class derived from more than one base class

Inheritance

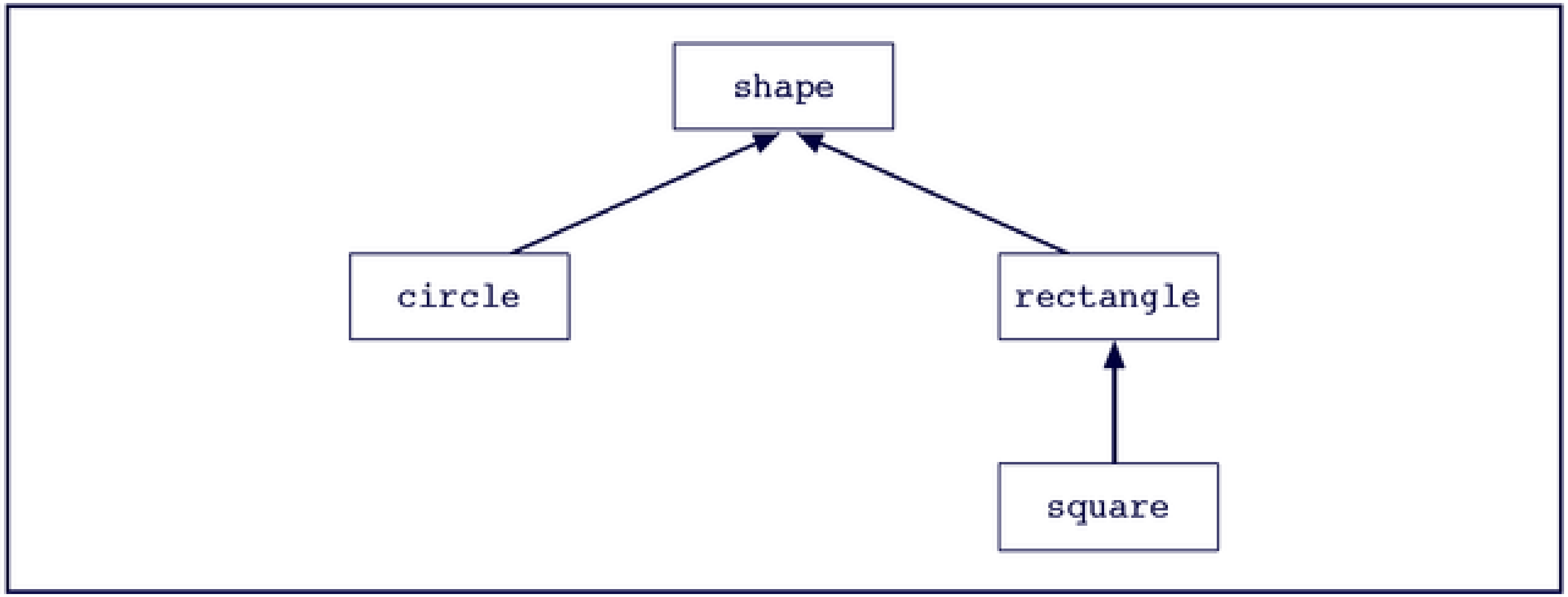

Figure 2-1 Inheritance hierarchy

General syntax to define a derived class:

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

Inheritance

- Private members of base class
  - private to base class; cannot be accessed directly by derived class
- Derived class can redefine member functions of base class; redefinition applies only to object of subclass
- Redefine: function member in derived class has same name, number, and type of parameters as function in base class

Redefining Member Functions

```
class baseclass {
 public: void print() const;
 private: int u,v;  };
```

[`print` prints `u` and `v`]

```
class derivedclass: public baseclass {
 public: void print() const;
 private: int x,y; };
```

[`print` prints `x` and `y`, then calls `baseclass::print()`]

Constructors of Derived and Base Classes

- Derived class can have its own private data members
- Constructors of derived class can (directly) initialize only private data members of the derived class
- Execution of derived class’s constructor triggers execution of one of base class’s constructors
- Call to base class’s constructor specified in heading of derived class constructor’s definition

Constructors of Derived and Base Classes (Example)

```
class baseclass {
 public: baseclass();
        baseclass(int x,int y);
 private: …; };


class derivedclass: public baseclass {
 public: derivedclass();
         derivedclass(int x,int y, int z);
 private: … ; };


derivedclass::derivedclass(int x,int y,int z) :
  baseclass(x,y) {
  // set derivedclass's private vars here
  }
```

Header Files

- Created to define new classes
- Definitions of base classes contained in header files
- Header files of new classes contain commands telling computer where to look for definitions of base classes

Preprocessor Commands

```
//Header file test.h

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

```
//Header file testA.h

#include "test.h"
```

```
//Program headerTest.cpp

#include "test.h"
#include "testA.h"
```

- `#ifndef H_test` means "if not defined H_test"
- `#define H_test` means "define H_test"
- `#endif` means "end if"

Protected Members of a Class

- If derived class needs
  - access to member of base class AND
  - needs to prevent direct access of member outside base class
- Then member of base class declared with member access specifier “protected”
- See pages 79-80 for inheritance types

Composition

- One or more members of a class are objects of another class type
- "has-a" relation between classes
- E.g. class `personalInfo` has as members `personType` and `dateType`:

```
class personalInfo { …
   private:
       personType name;
       dateType bDay;
 }
```

- Call to constructor of member objects specified in heading of definition of class's constructor

Basic Principles of OOD

- ✓ Abstraction
  - Separating logical properties of data from the implementation (did this in Chapter 1)
- ✓ Encapsulation
  - combining data and operations into a single unit (did this in Chapter 1)
- ✓ Inheritance
  - ability to create new data types from existing ones
- ➢ Polymorphism
  - using the same expression to denote different meanings, e.g., overloading, overriding

Operator Overloading

- Operator function: function that overloads an operator

```
cube s(5), t(7);
if (s < t) {
   ++s;
  }
```

- To overload an operator
  - Write function definition (header and body)
  - Name of function that overloads an operator is reserved word `operator` followed by operator overloaded

Operator Overloading

- The syntax of the heading of an operator function:

```
returnType  operator  operatorSymbol (arguments)
```

- Cube example:

```
class cube {
private:  int size;
public:
bool operator<(const cube &) const;
cube &operator++();
cube operator++(int);     . . . };

bool cube::operator<(const cube &right) const {
  return (size <  right.size);  }
cube &cube::operator++() {   // prefix ++
  size++;
  return (*this);  }
cube cube::operator++(int) {  // postfix ++
  cube x(*this);
  size++;
  return (x);  }
```

Operator Overloading Restrictions

- You cannot change the precedence of an operator
- The associativity cannot be changed
- You cannot use default arguments
- You cannot change the number of arguments
- You cannot create new operators
- The meaning of how an operator works with built-in types, such as int, remains the same
- Operators can be overloaded either for objects of the user-defined type, or for a combination of objects of the user-defined type and objects of the built-in type

Pointer this

- Every object of a class maintains a (hidden) pointer to itself
- Name of pointer is `this`
- In C++, `this` is a reserved word
- Available for you to use
- When object invokes member function, the member function references the pointer this of the object

friend Functions of Classes

- friend function: nonmember function with access to private data members
- To make function a friend to a class, reserved word friend precedes prototype
- Word friend appears only in prototype, not in definition of the friend function

Friend Example

```
class exampleclass {
 friend void friendfunc(exampleclass arg);
 public: ...
 private: int x;
}

void friendfunc(exampleclass arg) {
exampleclass local;
arg.x = 7;
local.y=9;
}
```

Operator Functions as Member and Nonmember Functions

- A function that overloads any of the operators `()`, `[]`, `->`, or = for a class must be declared as a member of the class
- Assume `opOverClass` has overloaded `op`
  - If the leftmost operand of `op` is not an object of `opOverClass`, the overloaded function must be a nonmember (friend of `opOverClass`)
  - If the `op` function is a member of `opOverClass`, then when applying `op` to `opOverClass` objects, the leftmost operand of op must be of type `opOverClass`

Overloading Binary Operators as Member Functions

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

```
returnType operator op(const className&) const;
```

Function Definition:

```
returnType className::operator op
                      (const className& otherObject) const
{
      //algorithm to perform the operation
      return (value);
}
```

Overloading Binary Operators as Nonmember Functions

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

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

Function Definition:

```
returnType operator op(const className& firstObject,
                       const className& secondObject)
{
      //algorithm to perform the operation
      return (value);
}
```

Overloading the Stream Insertion Operator (<<)

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

```
friend ostream& operator<<(ostream&, const className&);
```

Function Definition:

```
ostream& operator<<(ostream& osObject, const className& object)
{
      //local declaration if any
      //Output the members of the object
      //osObject<<. . .
      //Return the ostream object
      return osObject;
}
```

Overloading the Stream Extraction Operator (>>)

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

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

Function Definition:

```
istream& operator>>(istream& isObject, className& object)
{
      //local declaration if any
      //Read the data into the object
      //isObject>>. . .
      //Return the istream object
      return isObject;
}
```

Overloading Unary Operators

- If the operator function is a member of the class, it has no parameters
  - Exception (sort of): dummy “int” argument for postfix ++, −− operators
- If the operator function is a non-member (a friend), it has one parameter

Programming Example: Complex Numbers

```
complexType
------------------------------------------------------------
-realPart: double
-imaginaryPart: double
------------------------------------------------------------
+operator<<(ostream&, const complexType&): ostream&
+operator>>(istream&, complexType&): istream&
+setComplex(const double&, const double&): void
+complexType(double = 0, double = 0)
+operator+(const complexType&) const: complexType
+operator*(const complexType&) const: complexType
+operator==(const complexType&) const: bool
```

Figure 2-7 UML diagram of the class `complexType`

Overloading Functions

- Example: constructors

```
cube::cube() {size = 0;}
cube::cube(int s) {size = s;}
```

- Can do this with any function
- How does the complier know which one to use?
  - Correct function chosen by formal parameter list

Templates

- Powerful C++ tools
- Use a function template to write a single code segment for a set of related (overloaded) functions
- Use a class template for related classes

Syntax

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

Syntax of the function template:

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

Syntax for a class template:

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

Template Example

- Returns the larger of `x` and `y` of whatever type

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

- What `Type` will be used in the following?

```
  cout << larger(5,6) << endl;
```

Class Templates

- E.g. the elements in a list ADT could be `char`s, `int`s, `double`s, etc.
  - Don't want to have to define same class for each type!

```
template<class elemType>
class listType {. . .
bool isEmpty();
void insert(const elemType & newElement);  . . .
elemType list[100]; . . . }
```

- To instantiate a list of ints:

```
listType<int> intList;
```