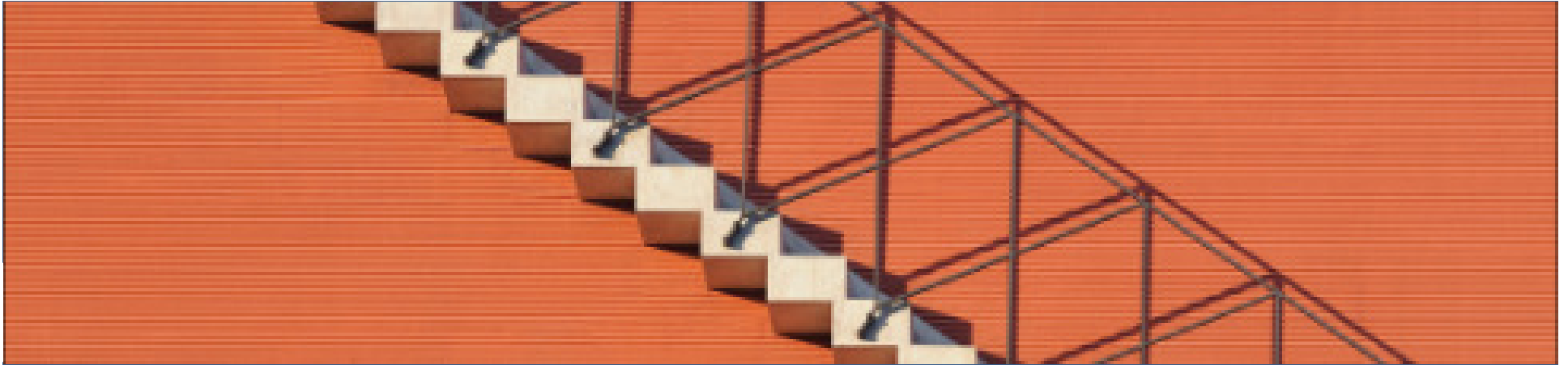




Module 8

Inheritance

Objectives

- In this topic, you will:
 - Learn about inheritance
 - Learn about derived and base classes
 - 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

Introduction

- Two common ways to relate two classes in a meaningful way are:
 - Inheritance (“is-a” relationship)
 - Composition, or aggregation: (“has-a” relationship)

Inheritance

- Inheritance: “is-a” relationship
 - Example: “every employee is a person”
- Inheritance allows creation of new classes from existing classes
 - Derived classes: new classes created from the existing class
 - Base class: the original class
- Derived class inherits the properties of its base classes

Inheritance (cont'd.)

- Inheritance helps reduce software complexity
- Single inheritance: derived class has a single base class
- Multiple inheritance: derived class has more than one base class
- Public inheritance: all public members of base class are inherited as public members by derived class

Inheritance (cont'd.)

- Inheritance can be viewed as a tree-like, or hierarchical, structure between the base class and its derived classes

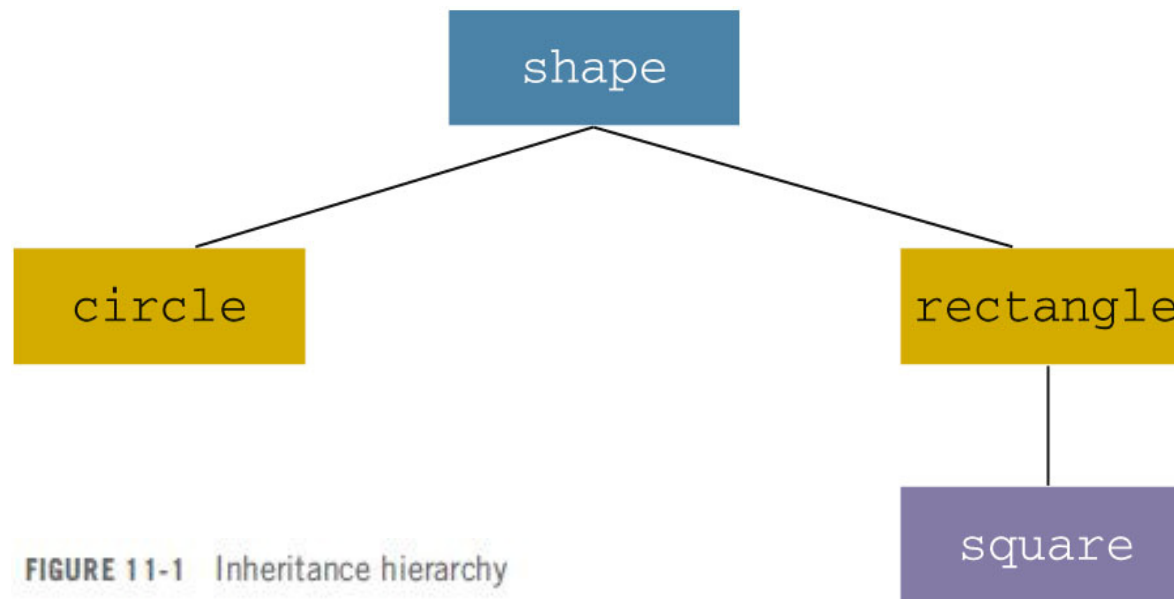


FIGURE 11-1 Inheritance hierarchy

Inheritance (cont'd.)

- Syntax of a derived class:

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

- **memberAccessSpecifier** is **public**, **protected**, or **private** (default)
- `private` members of a base class are `private` to the base class
 - Derived class cannot directly access them

Inheritance (cont'd.)

- `public` members of base class can be inherited as `public` or `private` members
- Derived class can include additional members (data and/or functions)
- Derived class can redefine `public` member functions of the base class
 - Applies only to the objects of the derived class
- All members variables of the base class are also member variables of the derived class

Redefining (Overriding) Member Functions of the Base Class

- To redefine a `public` member function:
 - Corresponding function in derived class must have same name/number/types of parameters
- If derived class overrides a `public` member function of the base class, then to call the base class function, specify:
 - Name of the base class
 - Scope resolution operator (`::`)
 - Function name with appropriate parameter list

Redefining Member Functions of the Base Class (cont'd.)

- boxType is derived from rectangleType, and it is a public inheritance
 - Also overrides print and area

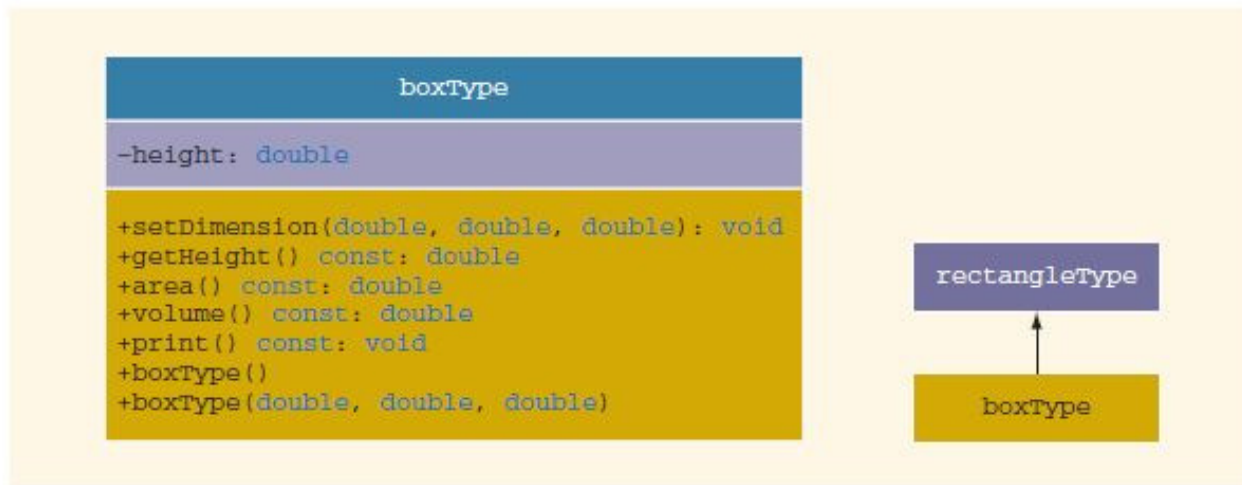


FIGURE 11-3 UML class diagram of the **class** boxType and the inheritance hierarchy

Constructors of Derived and Base Classes

- Derived class constructor cannot directly access `private` members of the base class
 - It can directly initialize only `public` member variables of the base class
- When a derived object is declared, it must execute one of the base class constructors
- Call to base class constructor is specified in heading of derived class constructor definition

Destructors in a Derived Class

- Destructors: used to deallocate dynamic memory allocated by the objects of a class
- When a derived class object goes out of scope
 - Automatically invokes its destructor
- When the destructor of the derived class executes
 - Automatically invokes the destructor of the base class

Header File of a Derived Class

- To define new classes, create new header files
- To create new derived classes, include commands that specify where the base class definitions can be found
- Definitions of the member functions can be placed in a separate file

Multiple Inclusions of a Header File

- Use the preprocessor command (`#include`) to include a header file in a program
 - Preprocessor processes the program before it is compiled
- To avoid multiple inclusion of a file in a program, use certain preprocessor commands in the header file

C++ Stream Classes

- `ios` is the base class for all stream classes
 - Contains formatting flags and member functions to access/modify the flag settings

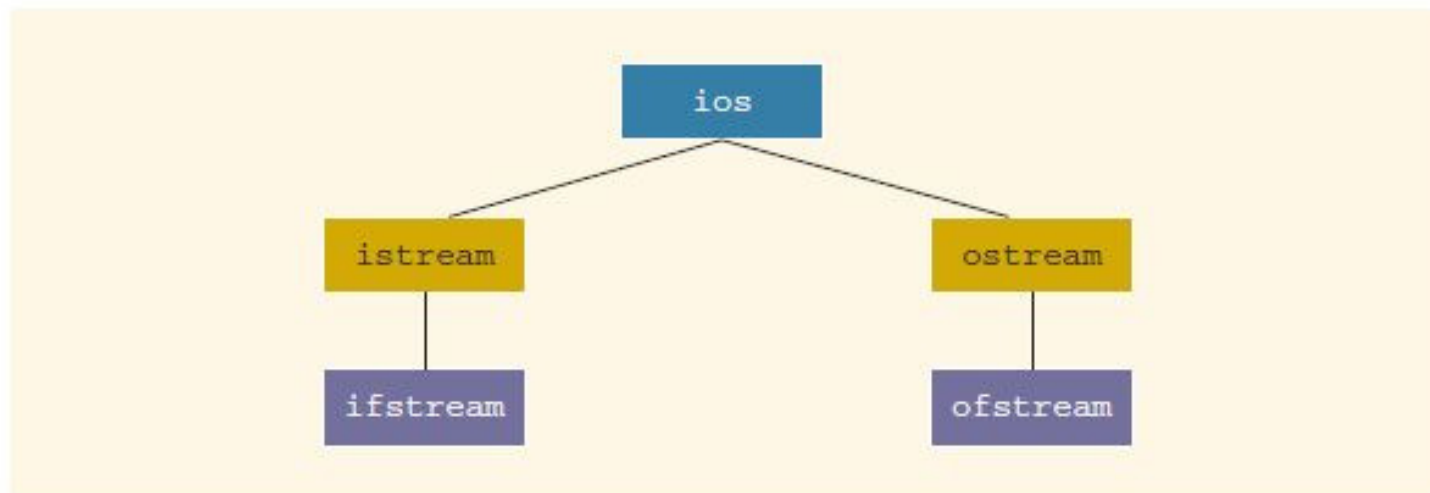


FIGURE 11-6 C++ stream classes hierarchy

C++ Stream Classes (cont'd.)

- `istream` and `ostream` provide operations for data transfer between memory and devices
 - `istream` defines the extraction operator (`>>`) and functions `get` and `ignore`
 - `ostream` defines the insertion operator (`<<`) which is used by `cout`
- `ifstream/ofstream` objects are for file I/O
 - Header file `fstream` contains the definitions for these

Protected Members of a Class

- Derived class cannot directly access `private` members of its base class
 - To give it direct access, declare that member as `protected`

Object-Oriented Design (OOD) and Object-Oriented Programming (OOP)

- The fundamental principles of object-oriented design (OOD) are:
 - Encapsulation: combines data and operations on data in a single unit
 - Inheritance: creates new objects (classes) from existing objects (classes)
 - Polymorphism: the ability to use the same expression to denote different operations

OOD and OOP (cont'd.)

- In OOD:
 - Object is a fundamental entity
 - Debug at the class level
 - A program is a collection of interacting objects
- OOD encourages code reuse
- Object-oriented programming (OOP) implements OOD

OOD and OOP (cont'd.)

- C++ supports OOP through the use of classes
- Function name and operators can be overloaded
- Polymorphic function or operator: has many forms
 - Example: division with floating point and division with integer operands

OOD and OOP (cont'd.)

- Templates provide parametric polymorphism
- C++ provides virtual functions to implement polymorphism in an inheritance hierarchy
 - Allows run-time selection of appropriate member functions
- Objects are created when class variables are declared
- Objects interact with each other via function calls

Virtual function

- In object-oriented programming, a virtual function or virtual method is a function or method whose behaviour can be overridden within an inheriting class by a function with the same signature.
- This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).

```

#include <iostream>
using namespace std;

class Fruit
{
    public:
    virtual void eat()
    {
        cout << "I eat generic fruit.\n";
    }
};

class Durian : public Fruit
{
    public:
    void eat()
    {
        cout << "I eat durian\n";
    }
};

class Apple : public Fruit
{
    public:
    void eat()
    {
        cout << "I eat apple\n";
    }
};

class OtherFruit : public Fruit
{
    // empty
};

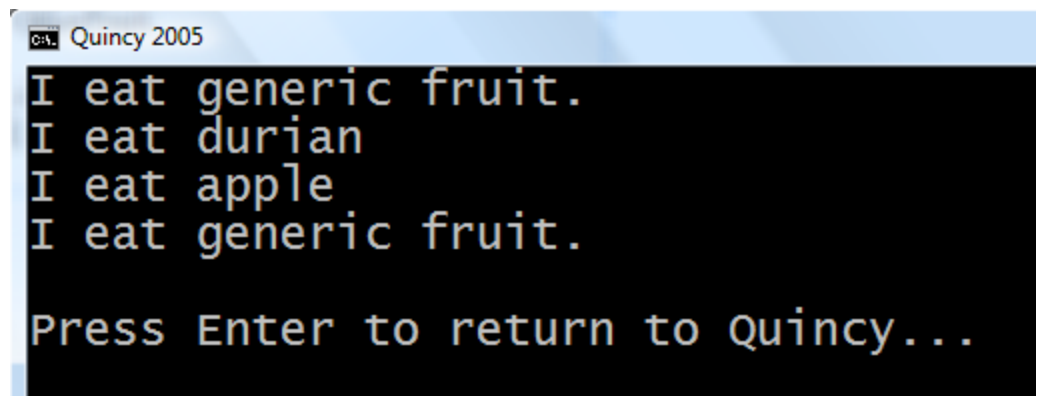
```

```

int main() {
    Fruit* fruits [4];
    fruits [0] = new Fruit;
    fruits [1] = new Durian;
    fruits [2] = new Apple;
    fruits [3] = new OtherFruit;

    for (int i = 0; i < 4; i++)
        fruits [i] -> eat ();
}

```



```

C:\> Quincy 2005
I eat generic fruit.
I eat durian
I eat apple
I eat generic fruit.

Press Enter to return to Quincy...

```