C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 11: Inheritance and Composition

Objectives

In this chapter, you will:

- 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

Objectives

In this chapter, you will:

- Explore three types of inheritance: public, protected, and private
- Learn about composition (aggregation)
- Become familiar with the three basic principles of object-oriented design

Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to <u>reuse the code functionality and fast implementation time.</u>

Inheritance can be defined as the <u>process where one object acquires the properties of another</u>.

Inheritance

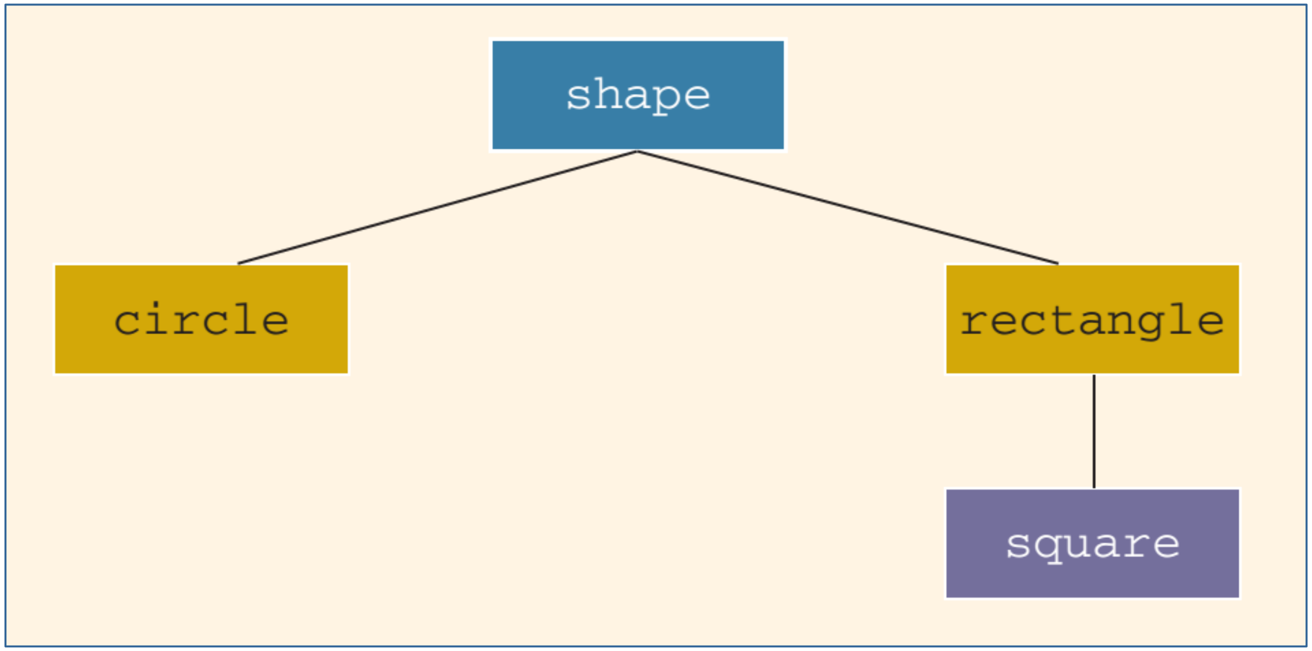
When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The new classes that we create from the existing classes are called the **derived** classes; the existing classes are called the **base** classes.

The derived classes inherit the properties of the base classes. So rather than create completely new classes from scratch, we can take advantage of inheritance and reduce software complexity.

Inheritance

Inheritance can be either single inheritance or multiple inheritance. In **single inheritance**, the derived class is derived from a single base class. In **multiple inheritance**, the derived class is derived from more than one base class.



Md. Sharif Hossen
Lecturer, Dept. of ICT, Comilla University, Bangladesh

Inheritance

The general syntax of a derived class is:

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

Where access-specifier is one of **public**, **protected**, or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Inheritance

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Md. Sharif Hossen
Lecturer, Dept. of ICT, Comilla University, Bangladesh

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public**, **protected** or **private** inheritance. The type of inheritance is specified by the access-specifier

We hardly use **protected** or **private** inheritance but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

1. **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become public members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are **never accessible** directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

Md. Sharif Hossen
Lecturer, Dept. of ICT, Comilla University, Bangladesh

Type of Inheritance

2. **Protected Inheritance:** When deriving from a **protected** base class, **public** **and** **protected** members of the base class become protected members of the derived class.

3. **Private Inheritance:** When deriving from a private base class, **public** **and** **protected** members of the base class become private members of the derived class.

Multiple Inheritances

C++ class can inherit members from more than one class and here is the extended syntax:

```
class derived-class: access baseA, access baseB....
```

Example:

```
class D: public A, private B, protected C
```

Multiple Inclusions of a Header File

What is inherited from the base class?

A derived class inherits all base class methods with the following exceptions:

i. Constructors, destructors and copy constructors of the base class.

ii. Overloaded operators of the base class.

iii. The friend functions of the base class.

Md. Sharif Hossen
Lecturer, Dept. of ICT, Comilla University, Bangladesh

Polymorphism

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be overridden, and this mechanism is called Function Overriding

Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.

2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Example of Function Overriding

```
class Base
{
        public:
                void show()
                {
                        cout << "Base class";
                }
};
class Derived: public Base
{
        public:
                void show()
                {
                        cout << "Derived Class";
                }
}
```

In this example, function **show()** is overridden in the derived class.

Md. Sharif Hossen
Lecturer, Dept. of ICT, Comilla University, Bangladesh

Function Call Binding with class Objects

Connecting the function call to the function body is called **Binding**. When it is done before the program is run, its called Early Binding or **Static Binding** or Compile-time Binding.

```
class Base
{
                public:
                                void show() {
                                                cout << "Base class";
                                }
};
class Derived: public Base
{
                 public:
                                void show() {
                                                cout << "Derived Class";
                                }
}
```

```
int main()
{
 Base b;       //Base class object
 Derived d;     //Derived class object
 b.show();     //Early Binding Ocuurs
 d.show();
}
```

In the above example, we are calling the overrided function using Base class and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.

Md. Sharif Hossen
Lecturer, Dept. of ICT, Comilla University, Bangladesh

Friend Functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.

Friends are functions or classes declared as such.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword `friend`:

```
// friend functions
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area () {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set values (int a, int b) {
  width = a;
  height = b;
}
```

```
CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
  return 0;
}
```

Friend Class

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
// friend class
#include <iostream>
using namespace std;

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (CSquare a);
};
```

```
class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

In this example, we have declared `CRectangle` as a friend of `CSquare` so that `CRectangle` member functions could have access to the protected and private members of `CSquare`, more concretely to `CSquare::side`, which describes the side width of the square.

Important points for friend classes and functions

Following are some important points about friend functions and classes:

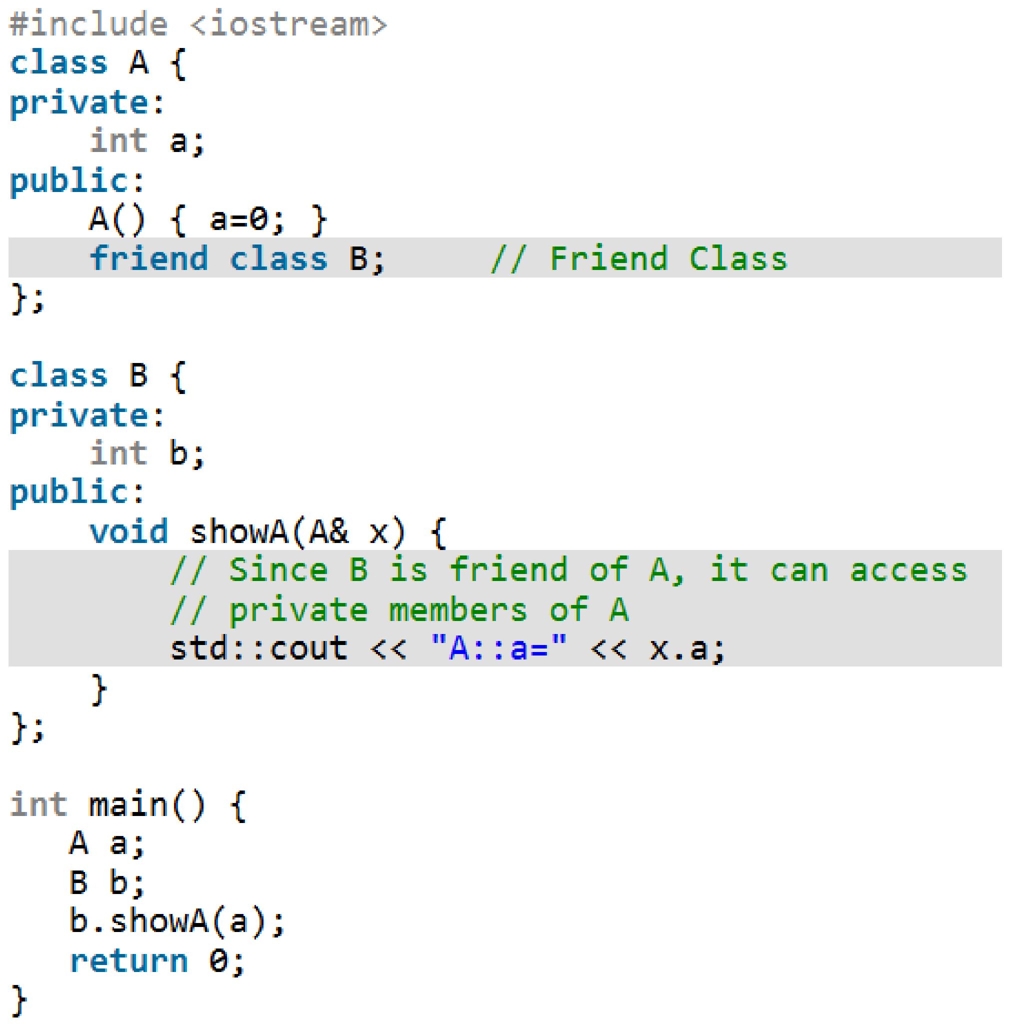
1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.

3) Friendship is not inherited

4) The concept of friends is not there in Java.

A simple and complete C++ program to demonstrate friend Class



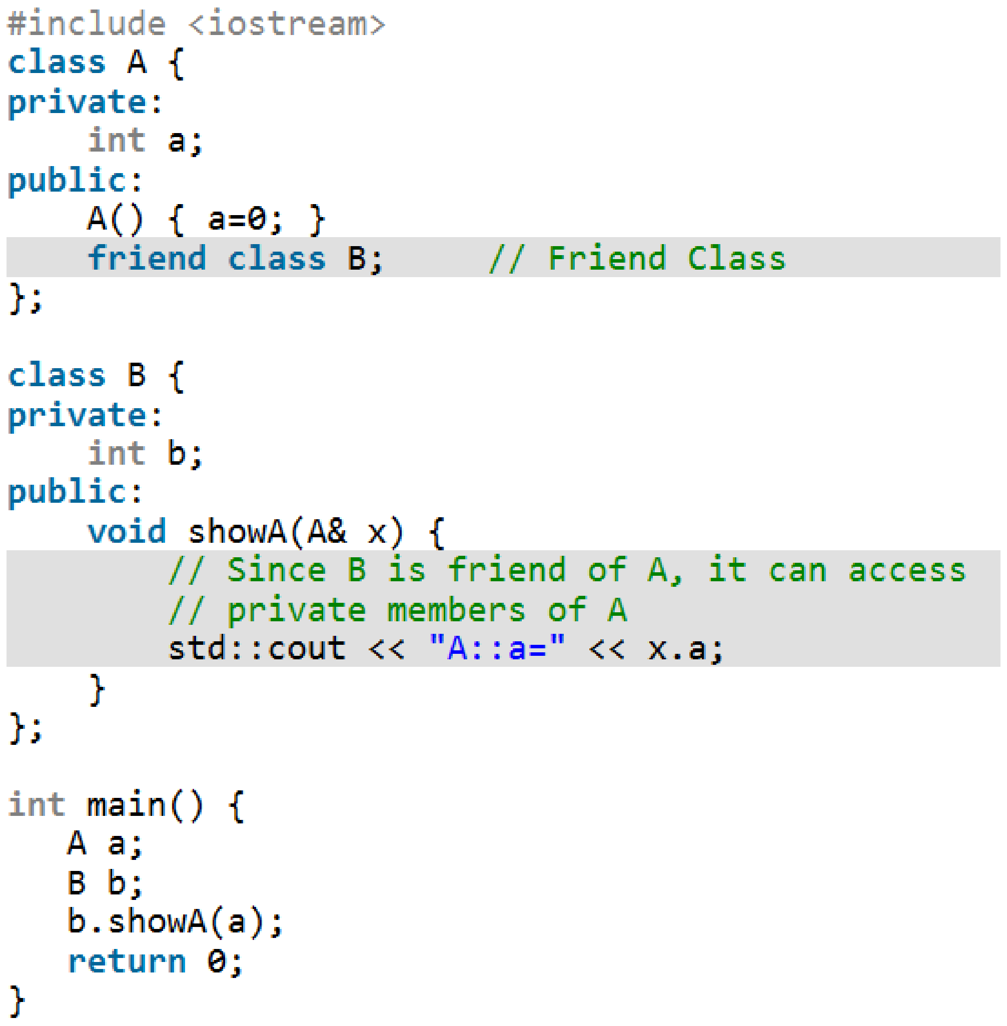
```cpp
#include <iostream>
class A {
private:
    int a;
public:
    A() { a=0; }
    friend class B;     // Friend Class
};

class B {
private:
    int b;
public:
    void showA(A& x) {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};

int main() {
   A a;
   B b;
   b.showA(a);
   return 0;
}
```

Md. Sharif Hossen
Lecturer, Dept. of ICT, Comilla University, Bangladesh

A simple and complete C++ program to demonstrate friend function of another class

```
#include <iostream>

class B;

class A
{
public:
    void showB(B& );
};

class B
{
private:
    int b;
public:
    B()  {  b = 0; }
    friend void A::showB(B& x); // Friend function
};

void A::showB(B &x)
{
    // Since show() is friend of B, it can
    // access private members of B
    std::cout << "B::b = " << x.b;
}

int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

A simple and complete C++ program to demonstrate global friend



```
#include <iostream>
class A {
private:
    int a;
public:
    A() { a=0; }
    friend class B;     // Friend Class
};

class B {
private:
    int b;
public:
    void showA(A& x) {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};

int main() {
   A a;
   B b;
   b.showA(a);
   return 0;
}
```