University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Object Oriented Programming
البرمجة الشيئية

Assist. Prof. Dr. Ekhlas Falih Naser
أ.م.د. أخلاص فالح ناصر

cs.uotechnology.edu.iq

Overview for functions

A function is a set of statements designed to accomplish a particular task. The advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program. C++ has added many new features to functions to make them more reliable and flexible.

General Format of a Function Definition:

Functions can be define before the definition of the main() function, or they can be declared before it and define after it.

Declaring a function means listing its return type, name, and arguments. This line is called the function prototype. A function prototype tells the compiler the type of data returned by the function. It is usually defined after the preprocessing statements at the beginning of the program.

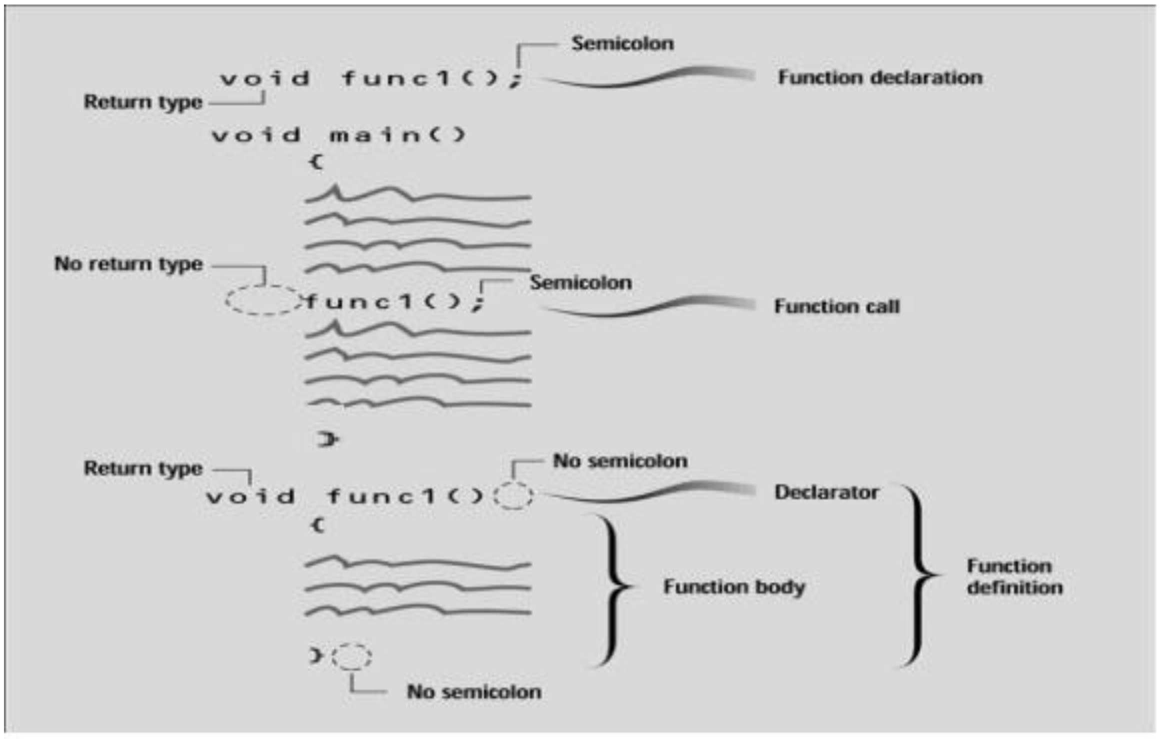


Figure 1: ***Function syntax.*:**

Example 1: Write a C++ program to find the maximum value between two values using function

```
#include <iostream.h>
float max ()
{
   float  a,  b;        a=3.5;         b=10.25;
    float  c;
    if (a > b)
            c = a;
      else
            c = b;
    return (c);
}
void main (  )
{    float k;
          k=max();
     cout <<k;
}
```

```
Output:-
10.25
```

Local and Global Variables:

The variables in general bay be classified as local or global variables.

(a) **Local variables:** Identifiers, variables and functions in a block are said to belong to a particular block or function and these identifiers are known as the local parameters or variables. Local variables are defined inside a function block or a compound statement. For example,

```
#include <iostream.h>
void sum ()
{        int  a,b,s;       // a,b,s  are local variables in a function sum()
         a=3;
         b=5;
          s=a+b;
          cout<<s;
}
void main( )
{
          sum();
}
```

(b) Global variables: these are variables defined outside the main function block. These variables are referred by the same data type and by the same name through out the program in both the calling portion of the program and in the function block.

```
#include <iostream.h>
int  a,b,s;          // a,b,s  are Global variables
void sum ()
{       s=a+b;
cout<<s;
}
void main( )
{       a=3;
        b=5;
        sum();
}
```

Inline Function:

C++ suppliers' programmers with the inline keyword, which can speed up programs by making very short functions execute more efficiently. Normally a function resides in a separate part of memory, and is referred to by a running program in which it is called. Inline functions save the step of retrieving the function during execution time, at the cost of a larger compiled program.

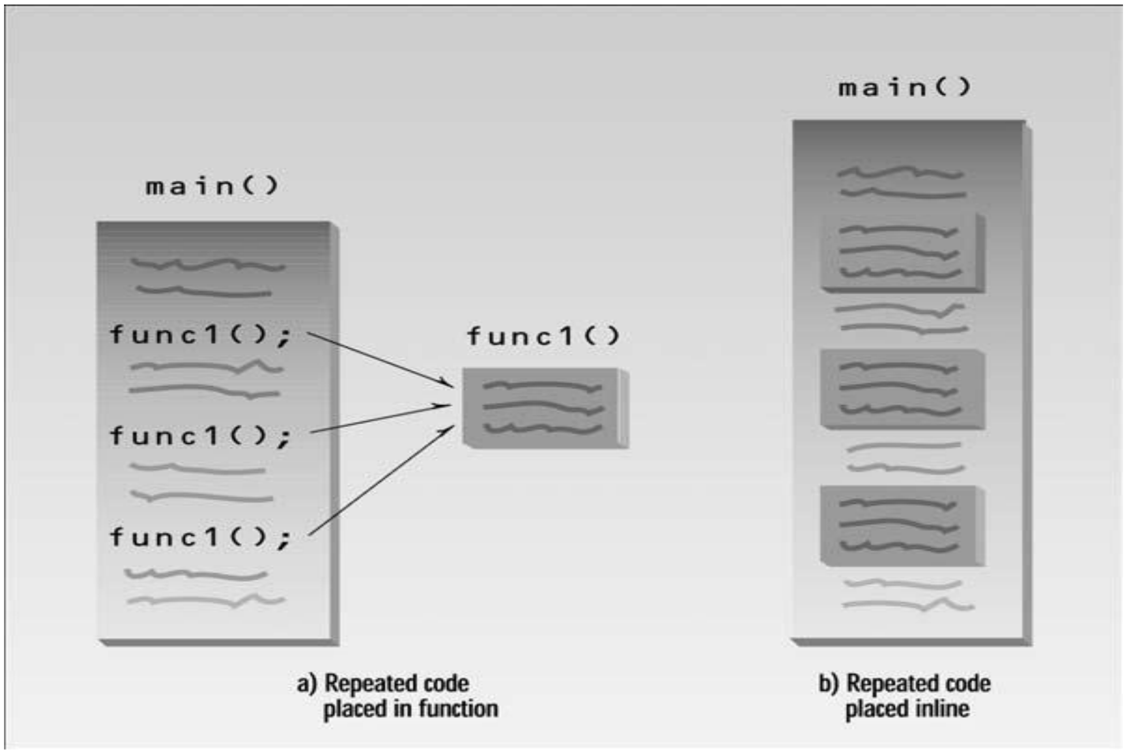

main()
func1();
func1();
func1();
func1()
main()
a) Repeated code placed in function
b) Repeated code placed inline

Figure: ***Functions versus inline code***

Example 2: Write a C++ program to find the square of a number using inline function

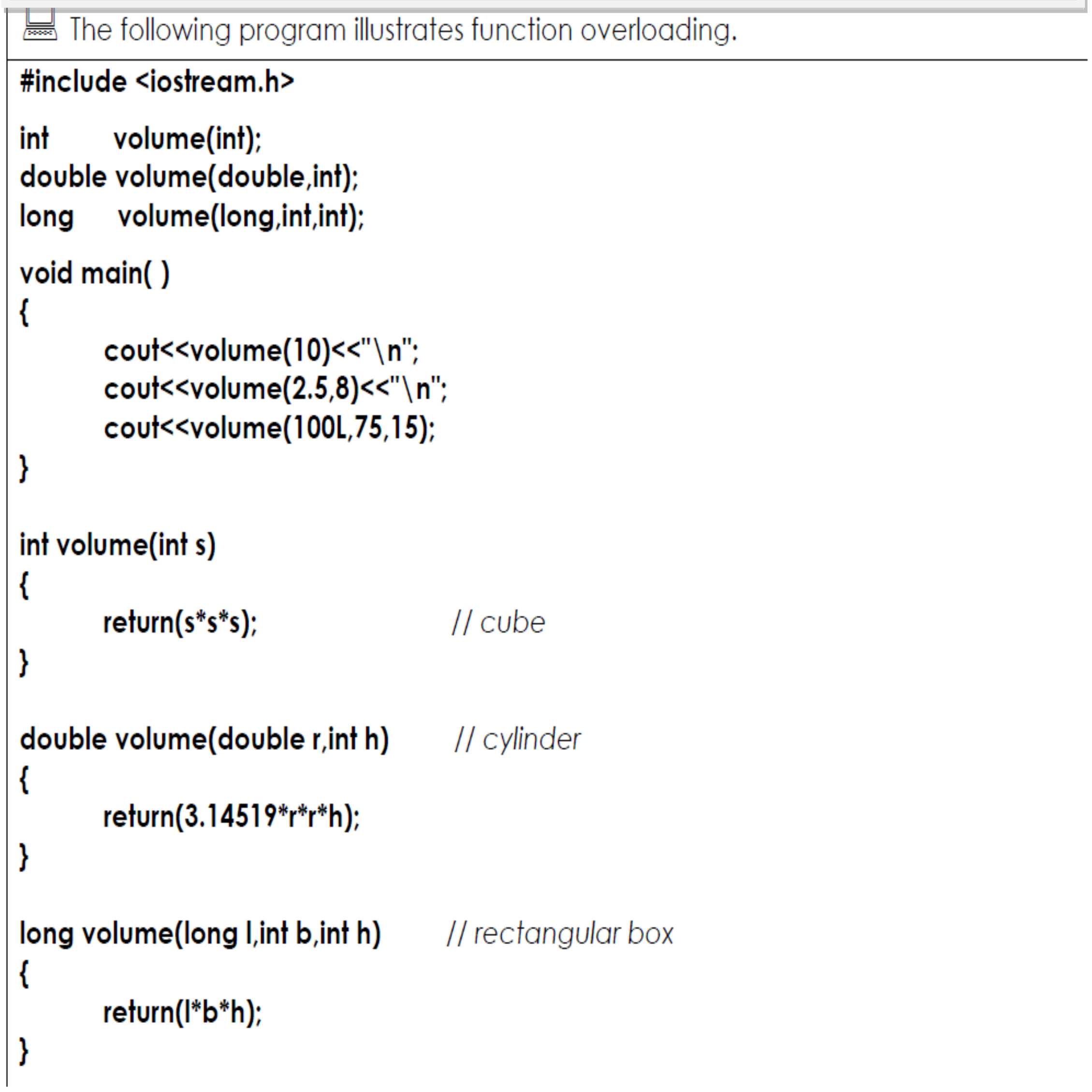
```
#include<iostream.h>
inline int square ( int y )
   {
        return ( y*y );
      }
void main( )
 {
          int m;
                m=5;
                cout<< square ( m ) ;
}
```

Function Overloading:

Overloading refers to the use the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks.

We can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call.

Example3:



The following program illustrates function overloading.

```
#include <iostream.h>

int       volume(int);
double volume(double,int);
long     volume(long,int,int);

void main( )
{
        cout<<volume(10)<<"\n";
        cout<<volume(2.5,8)<<"\n";
        cout<<volume(100L,75,15);
}

int volume(int s)
{
        return(s*s*s);                    // cube
}

double volume(double r,int h)       // cylinder
{
        return(3.14519*r*r*h);
}

long volume(long l,int b,int h)       // rectangular box
{
        return(l*b*h);
}
```

Passing Parameters:

There are two main methods for passing parameters to a program:

1- Passing by Value:

When parameters are passed by value, a copy of the parameters value is taken from the calling function and passed to the called function. The original variables inside the calling function, regardless of changes made by the function to it are parameters will not change. All the pervious examples used this method.

Example 4

💻 Write C++ program using function to calculate the average of two numbers entered by the user in the main program:

```
#include<iostream.h>

void aver (int x1, int x2)
{
        float z;
        z = ( x1 + x2) / 2.0;
       cout<<z;
}
void main( )
{
        int   num1,num2;
        cout << "Enter 2 numbers \n";
        cin >> num1 >> num2;
        aver (num1, num2);

}
```

```
Output:-
4.5
```

```
Input:-
Enter 2 numbers
   6    3
```

2- Passing by Reference:

When parameters are passed by reference their addresses are copied to the corresponding arguments in the called function, instead of copying their values. Thus pointers are usually used in function arguments list to receive passed references.

Example 5:

The following program illustrates passing parameter by reference.

```
#include <iostream.h>
void swap(int *a,int *b);
void main( )
{
        int x=10;
        int y=15;
        cout<<"x before swapping is:"<<x<<"\n";
        cout<<"y before swapping is:"<<y<<"\n";

        swap(&x,&y);
        cout<<"x after swapping is:"<<x<<"\n";
        cout<<"y after swapping is:"<<y<<"\n";
}
void swap(int *a,int *b)
{
        int c;
        c=*a;
        *a=*b;
        *b=c;
}
```

Example 6:

Write a C++ program using functions to print the contents of an integer array of 10 elements.

```
#include <iostream.h>
int const size=10;
void pary(int y[]);
void main( )
{
        int s[size]={2,4,6,8,10,12,14,16,18,20};
        pary(s);
}
void pary(int x[])
{
        int i;
        for (i=0;i<size;i++)
            cout<<x[i]<<endl;
}
```

Example 7:

Design a function that takes the third element of an integer array defined in the main program. The function must multiply the element by 2 and return the result to the main program.

```
#include<iostream.h>
int elementedit(int);

void main( )
{
        int a[4]={2,5,3,7};
        int k;
        k=elementedit(a[2]);
        cout<<"k after change is: "<<k;
}

int elementedit(int m)
{
        m=m*2;
        return(m);
}
```

Default Arguments

As with global functions, a member function of a class may have default arguments. The same rules apply: all default arguments should be trailing arguments, and the argument should be an expression consisting of objects defined within the scope in which the class appears.

Surprisingly, a function can be called without specifying all its arguments. This won’t work on just any function: The function declaration must provide default values for those arguments that are not specified.

Example 8:Write a simple program to represent a default argument

```
// missarg.cpp
// demonstrates missing and default arguments
#include <iostream>
using namespace std;

void repchar(char='*', int=45);    //declaration with
                                   //default arguments
int main()
   {
   repchar();                      //prints 45 asterisks
   repchar('=');                   //prints 45 equal signs
   repchar('+', 30);               //prints 30 plus signs
   return 0;
   }
//--------------------------------------------------------------
// repchar()
// displays line of characters
void repchar(char ch, int n)       //defaults supplied
   {                               //   if necessary
   for(int j=0; j<n; j++)          //loops n times
      cout << ch;                  //prints ch
   cout << endl;
   }
```

In this program the function repchar() takes two arguments. The first time it's called with no arguments, the second time with one, and the third time with two. Why do the first two calls work? Because the called function provides default arguments, which will be used if the calling program doesn't supply them. The default arguments are specified in the declaration for repchar(): **void repchar(char='*', int=45);** The default argument follows an equal sign, which is placed directly after the type name. You can also use variable names, as in **void repchar(char reptChar='*', int numberReps=45);** If one argument is missing when the function is called, it is assumed to be the last argument. The repchar() function assigns the value of the single argument to the **ch** parameter and uses the default value 45 for the n parameter. If both arguments are missing, the function assigns the default value '*' to **ch** and the default value 45 to n.

Overview of OOP:

When working with computers, it is very important to be fashionable. In the **1960s**, the new fashion was what was called *high-level languages* (H.L.L.) such as ***FORTRAN*** and ***COBOL***, in which the programmer did not have to understand the ***machine instructions***.

In the **1970s**, people realized that there were better ways to program than with a jumble of GOTO statements, and the *structured programming languages* such as ***PASCAL*** were invented.

In the **1980s**, much time was invested in trying to get good results out of *fourth-generation languages* (4GLs), in which complicated programming structures could be coded in a few words. There were also schemes such as Analyst Workbenches, which made systems analysts into highly paid and overqualified programmers.

Bjarne Stroustrup at Bell Labs developed C++ during 198-1985. The term C++ was first used in 1983. Prior to 1983, Stroustrup added features to C programming language and formed what he called “C with Classes”. In addition to the efficiency and portability of C, C++ provides number of new features. C++ programming language is basically an extension of C programming language.

The fashion of the **1990s** is most definitely *object-oriented programming*.

Read any book on object-oriented programming, and the first things you will read about are three importance OOP features:

• Encapsulation *and Data Hiding*.

• Inheritance *and Reuse*.

• Polymorphism.

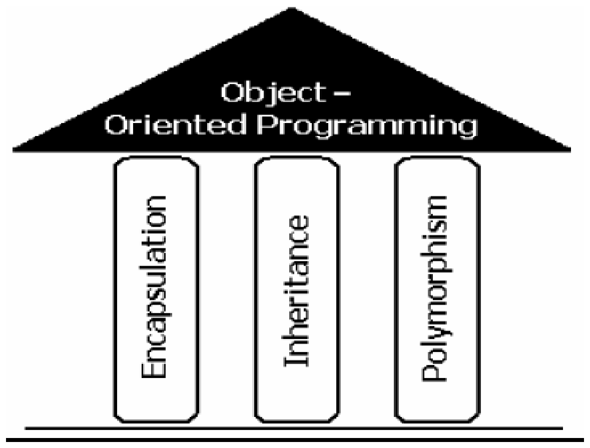


Figure 1: The three pillars of OOP.

Encapsulation and Data Hiding:

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes.

Once created, class acts as a fully encapsulated entity, it is used as a whole unit. The actual inner workings of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it.

Inheritance and Reuse:

C++ supports the idea of ***reuse*** through ***inheritance***. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The Quasar model is derived from the Star model and thus inherits all its qualities, but can add to them as needed.

Polymorphism:

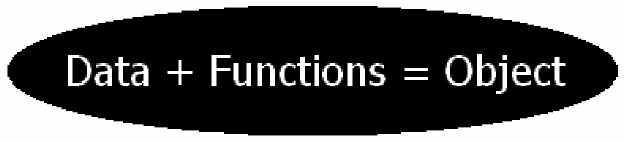
C++ supports the idea that different objects do "the right thing" through what is called function ***polymorphism*** and class polymorphism.

Poly means many, and morph means form. Polymorphism refers to the same name taking *many forms*.

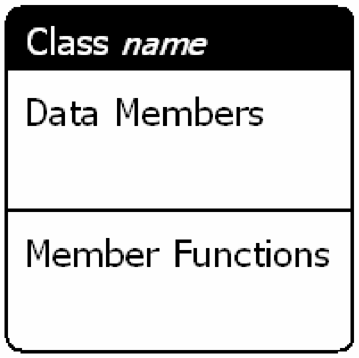
Class Definition:

Class is a keyword, whose functionality is similar to that of the ***struct*** keyword, but with the possibility of including functions as members, instead of only data.

Classes are collections of variables and functions that operate on those variables. The variables in a class definition are called data members, and the functions are called member functions.



Note: Class is a specification for number of objects.



A **class definition** consists of two parts: header and body. The class **header** specifies the class **name** and its **base classes**. The class **body** defines the class **members**. Two types of members are supported:

- **Data members** have the syntax of variable definitions and specify the representation of class objects.
- **Member functions** have the syntax of function prototypes and specify the class operations, also called the class **interface**.

General form of class declaration:

```
class class-name
{
        public:
          public-data-members;
          public-functions;

        private:
          private-data-members;
          private -functions;

        protected:
          protected-data-members;
          protected -functions;
};
```

Class members fall under one of three different access permission categories:

❖ **Public** members are accessible by all class users.

❖ **Private** members are only accessible by the class members.

❖ ·**Protected** members are only accessible by the class members and the members of a derived class.

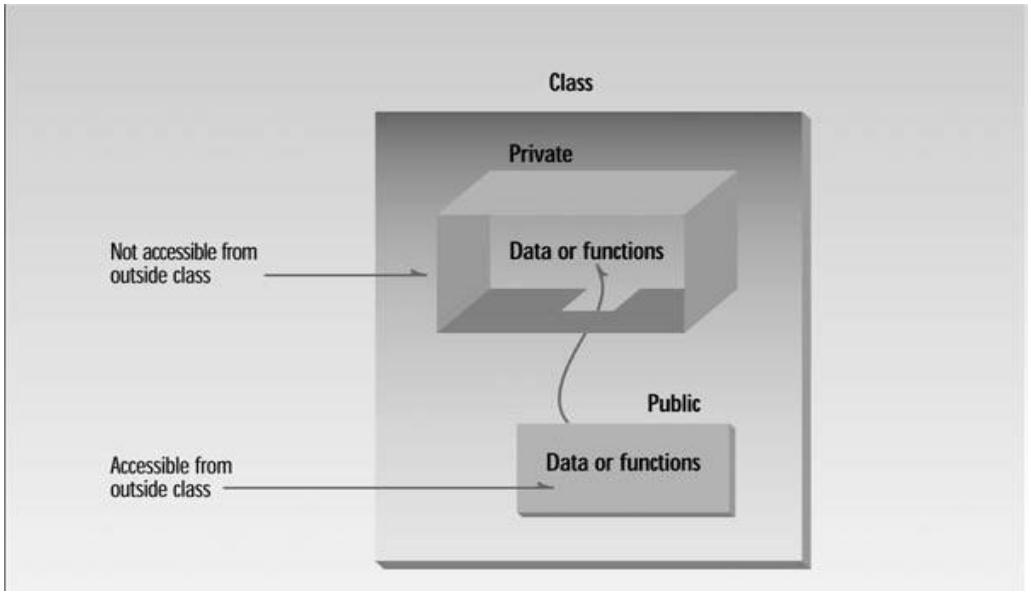
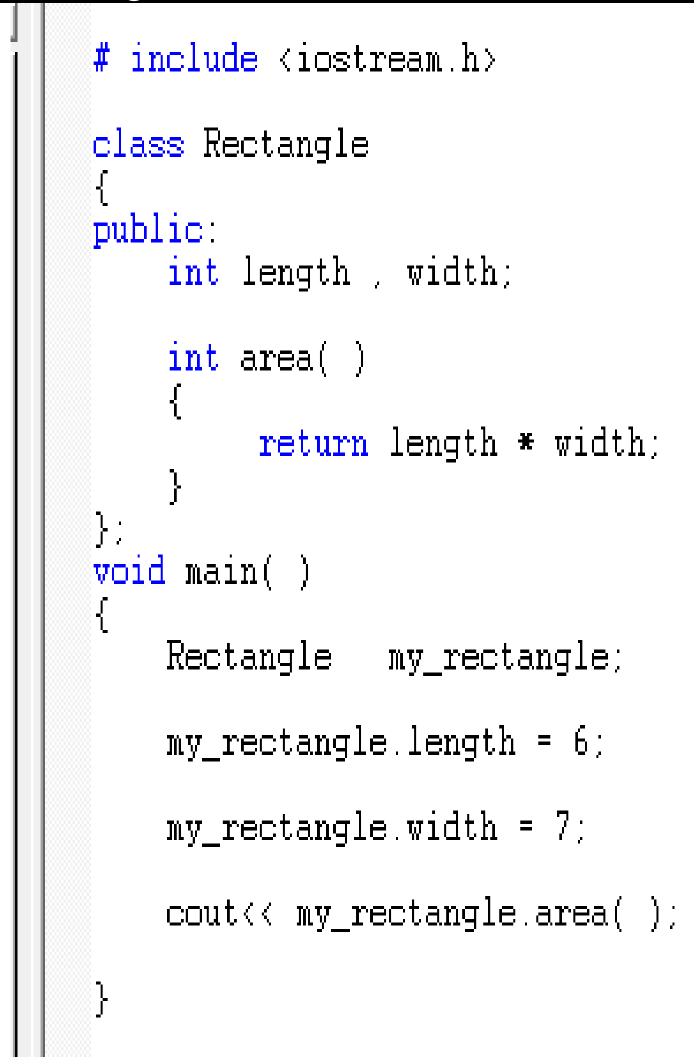

Class
Private
Data or functions
Not accessible from outside class
Public
Data or functions
Accessible from outside class

Figure 2: Public and Private Definition

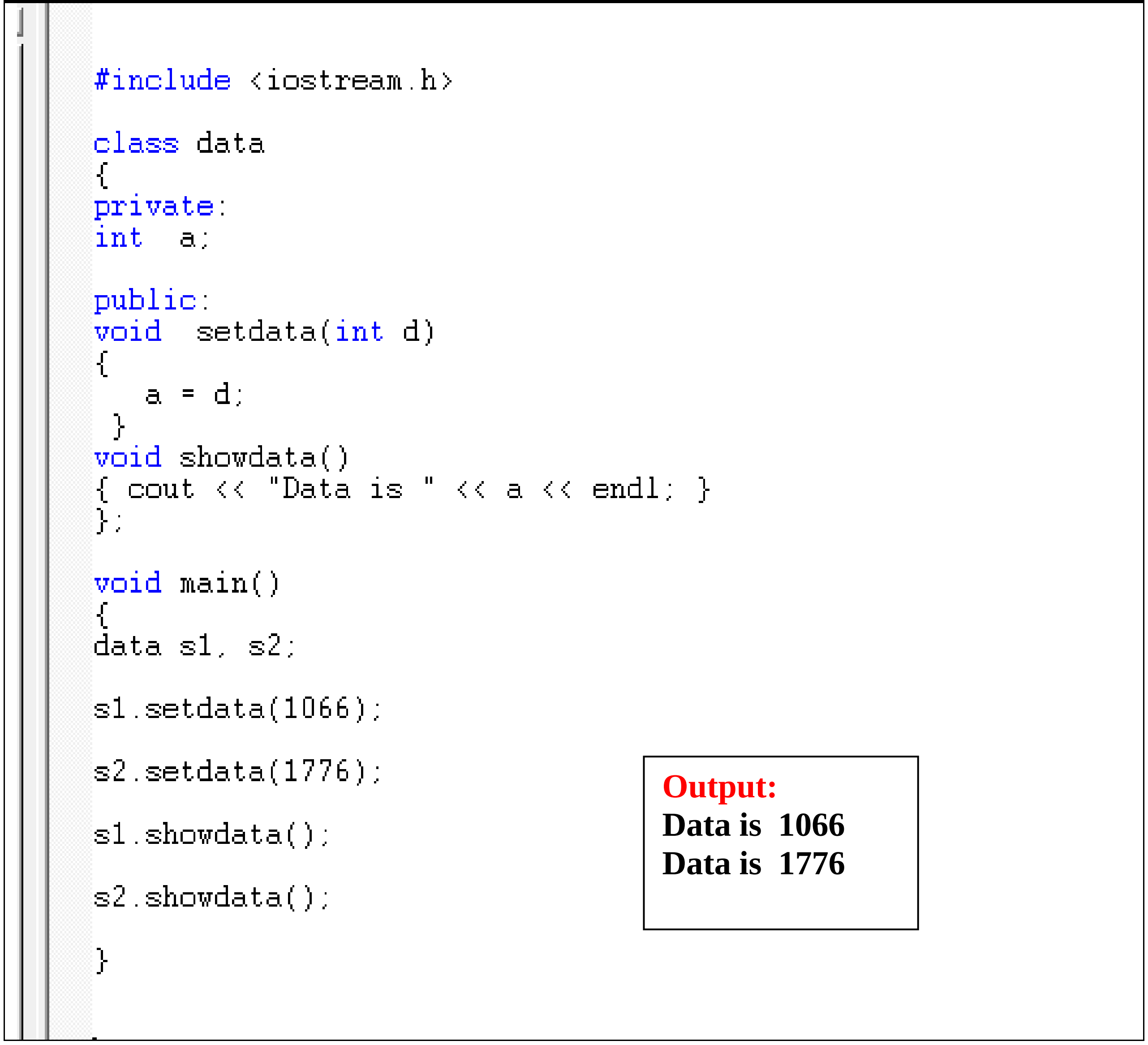
Example 1: write an OOP to find the area of rectangle using a class called rectangle



```
# include <iostream.h>

class Rectangle
{
public:
    int length , width;

    int area( )
    {
         return length * width;
    }
};
void main( )
{
    Rectangle   my_rectangle;

    my_rectangle.length = 6;

    my_rectangle.width = 7;

    cout<< my_rectangle.area( );

}
```

Output
42

Example 2: write an OOP to read a data and print it. Create a class called data with one data member called (a).

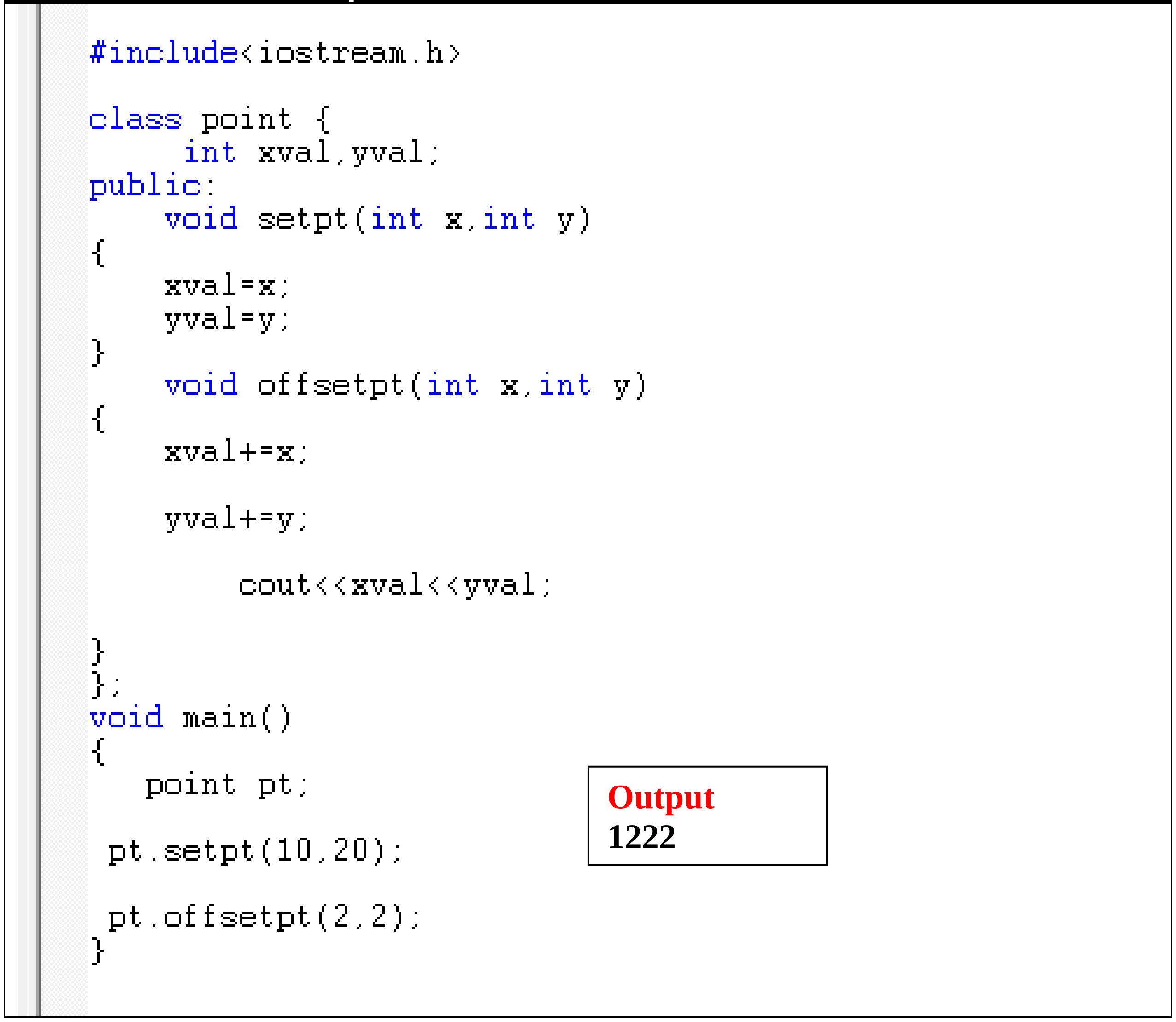


```
#include <iostream.h>

class data
{
private:
int   a;

public:
void  setdata(int d)
{
   a = d;
 }
void showdata()
{ cout << "Data is " << a << endl; }
};

void main()
{
data s1, s2;

s1.setdata(1066);

s2.setdata(1776);

s1.showdata();

s2.showdata();

}
```

Output:

```
Data is  1066
Data is  1776
```

Example 3: write an OOP to define the coordinate of point and shift these values of this point.



```
#include<iostream.h>

class point {
     int xval,yval;
public:
    void setpt(int x,int y)
{
    xval=x;
    yval=y;
}
    void offsetpt(int x,int y)
{
    xval+=x;

    yval+=y;

        cout<<xval<<yval;

}
};
void main()
{
   point pt;

 pt.setpt(10,20);

 pt.offsetpt(2,2);
}
```

Output
1222

Class Constructors and Destructors:

A **class constructor** is a function that is executed automatically whenever a new instance of a given class is declared.

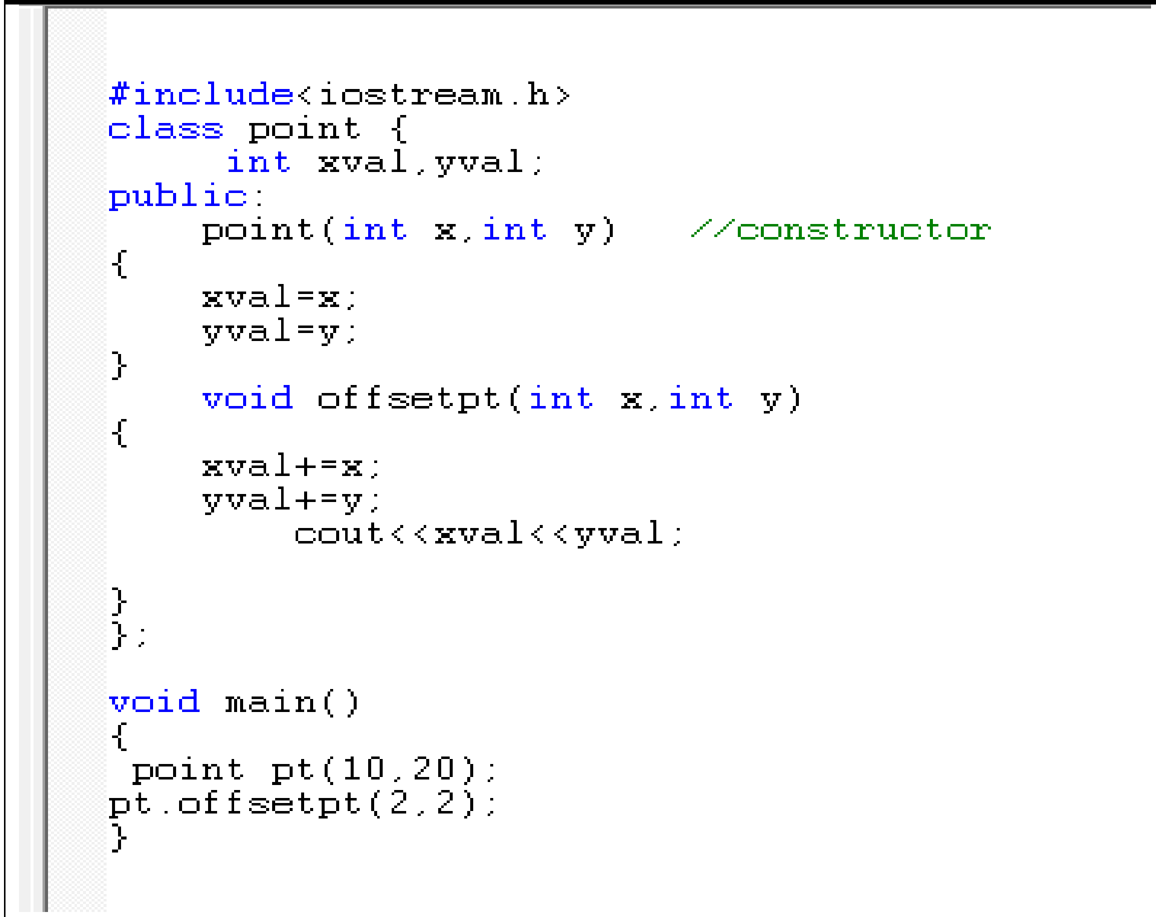
The main purpose of a class constructor is to perform any initializations related to the class instances via passing of some parameter values as initial values and allocate proper memory locations for that object.

Note1: A class constructor must have the same name as that of the associated class.

Note2: A class constructor has not return type, not even void.

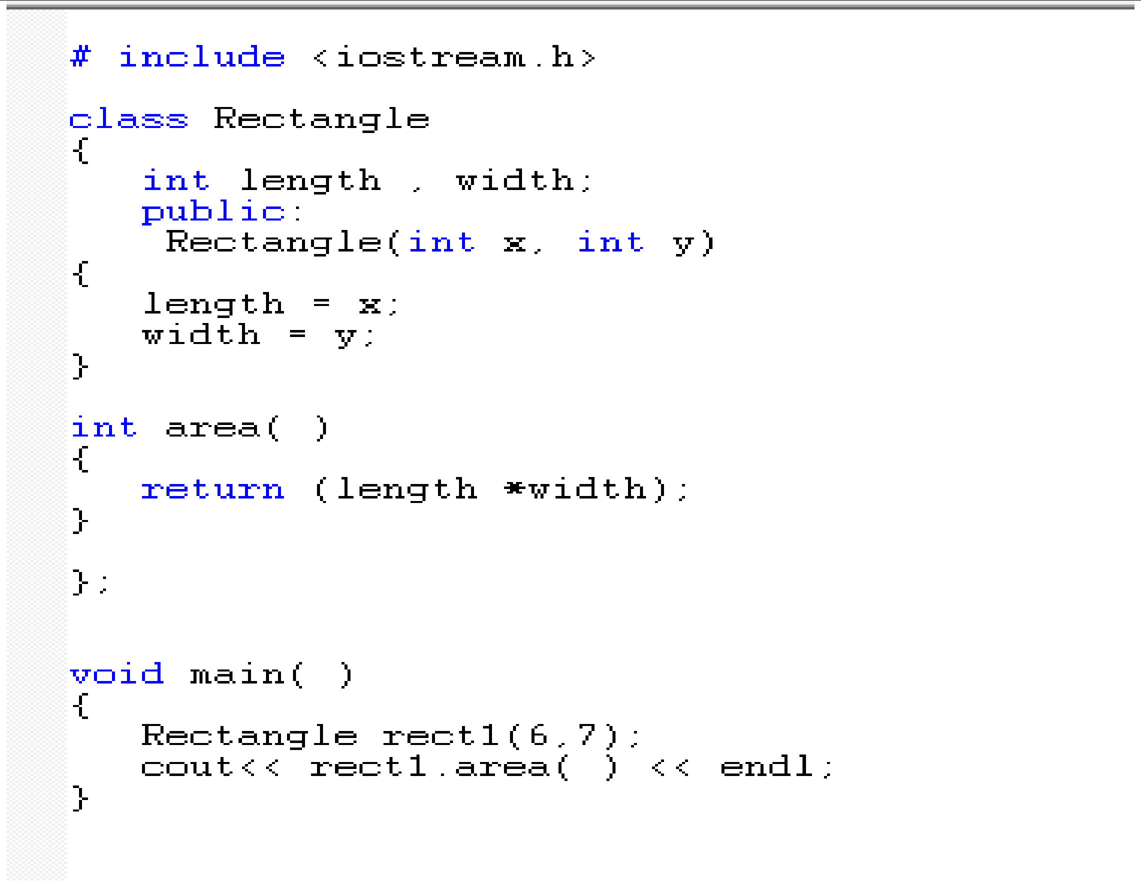
Note3: A class constructor can be overloaded

Example 1: Write an OOP to represent simple constructor with class point which contains two data members vxal and yval.



```
#include<iostream.h>
class point {
     int xval,yval;
public:
    point(int x,int y)   //constructor
{
    xval=x;
    yval=y;
}
    void offsetpt(int x,int y)
{
    xval+=x;
    yval+=y;
        cout<<xval<<yval;

}
};

void main()
{
 point pt(10,20);
pt.offsetpt(2,2);
}
```

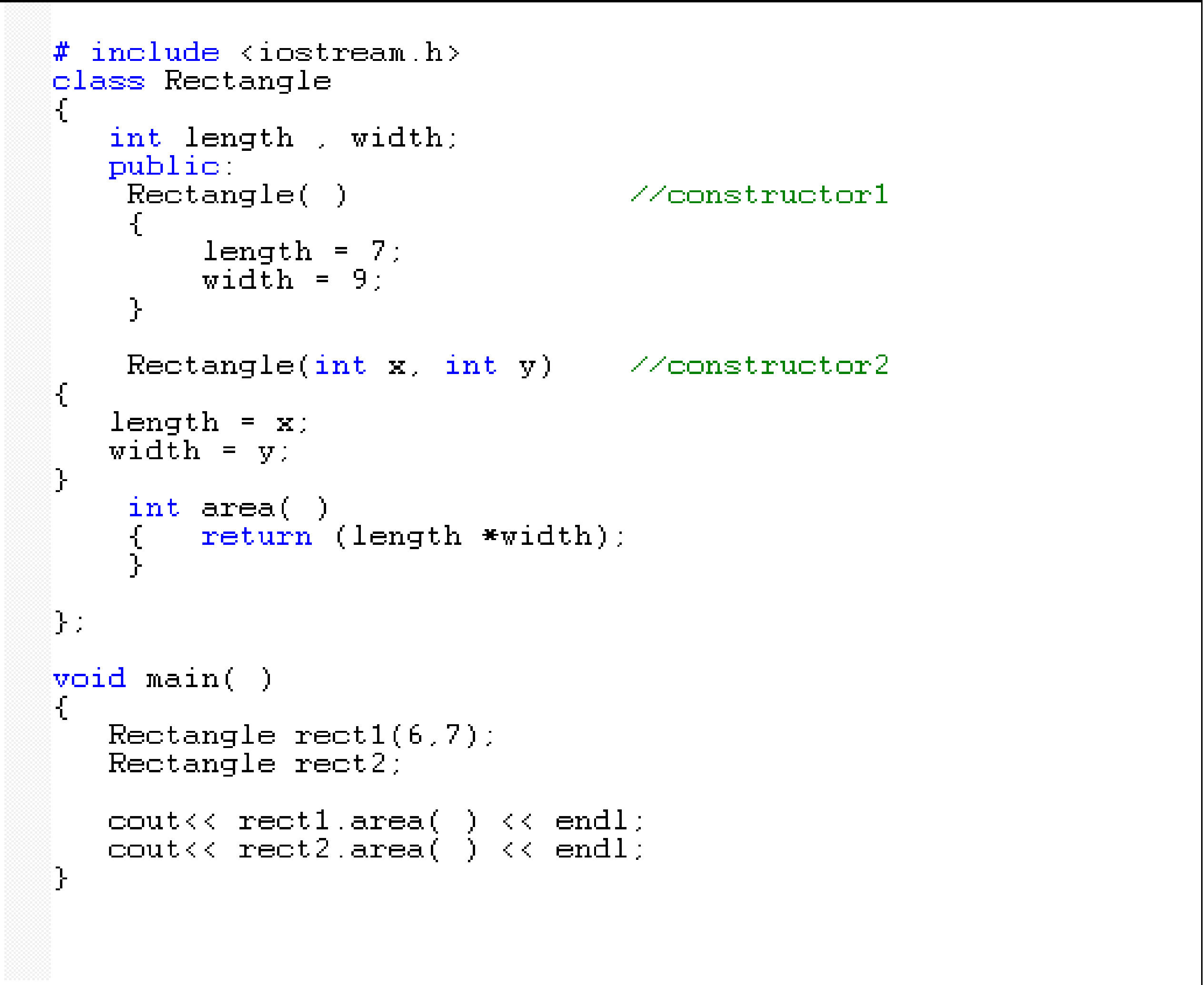
Example 2: Write an OOP to represent a rectangle constructor.



```
# include <iostream.h>

class Rectangle
{
   int length , width;
   public:
    Rectangle(int x, int y)
{
   length = x;
   width = y;
}

int area( )
{
   return (length *width);
}

};


void main( )
{
   Rectangle rect1(6,7);
   cout<< rect1.area( ) << endl;
}
```

A class may have more than one constructor. To avoid ambiguity, however, each of these must have a unique signature.

Example 3: Write an OOP to represent multiple constructors in class rectangle.

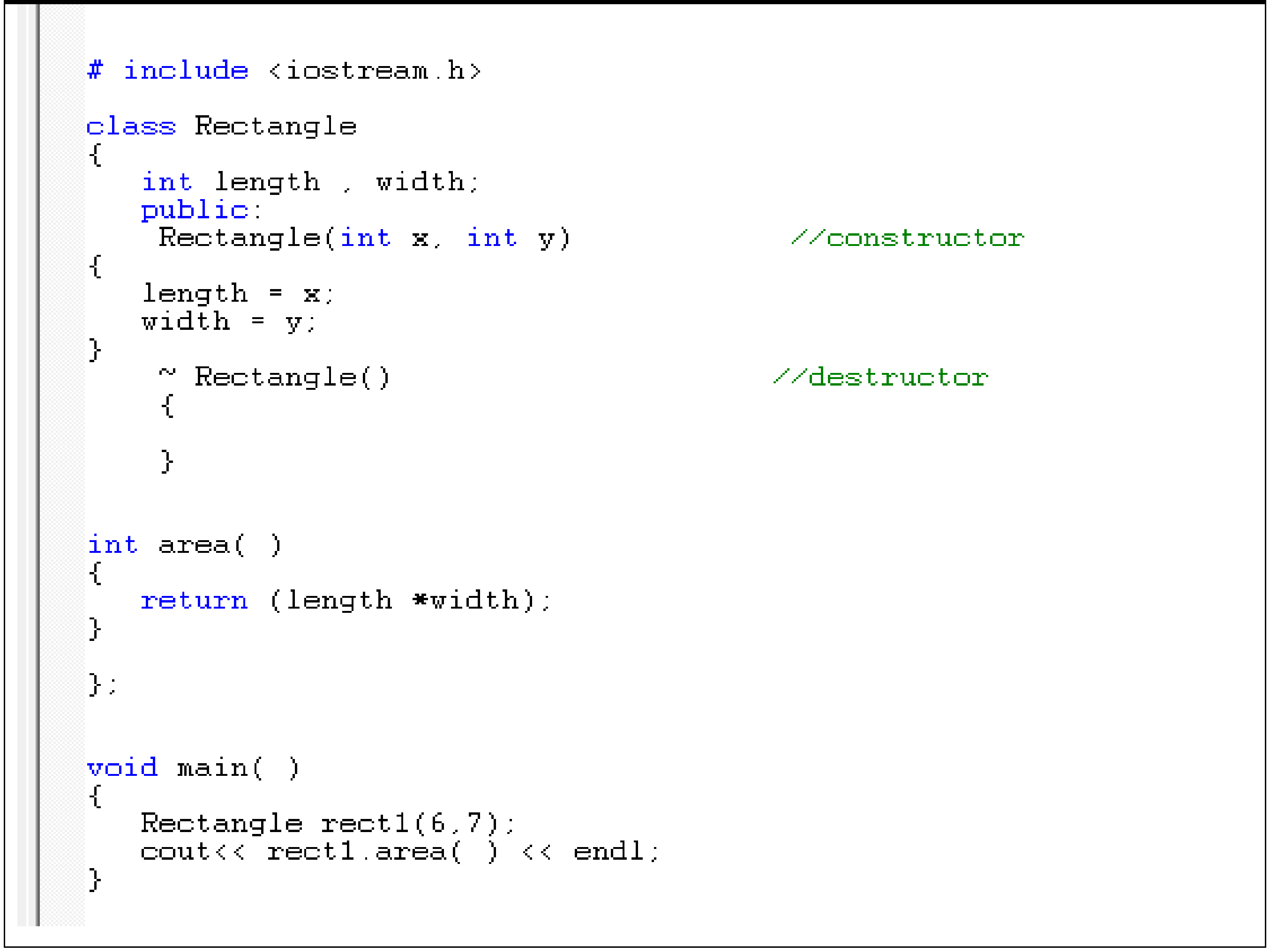


```
# include <iostream.h>
class Rectangle
{
   int length , width;
   public:
    Rectangle( )                    //constructor1
    {
        length = 7;
        width = 9;
    }

    Rectangle(int x, int y)     //constructor2
{
   length = x;
   width = y;
}
    int area( )
    {   return (length *width);
    }

};

void main( )
{
   Rectangle rect1(6,7);
   Rectangle rect2;

   cout<< rect1.area( ) << endl;
   cout<< rect2.area( ) << endl;
}
```

Destructor

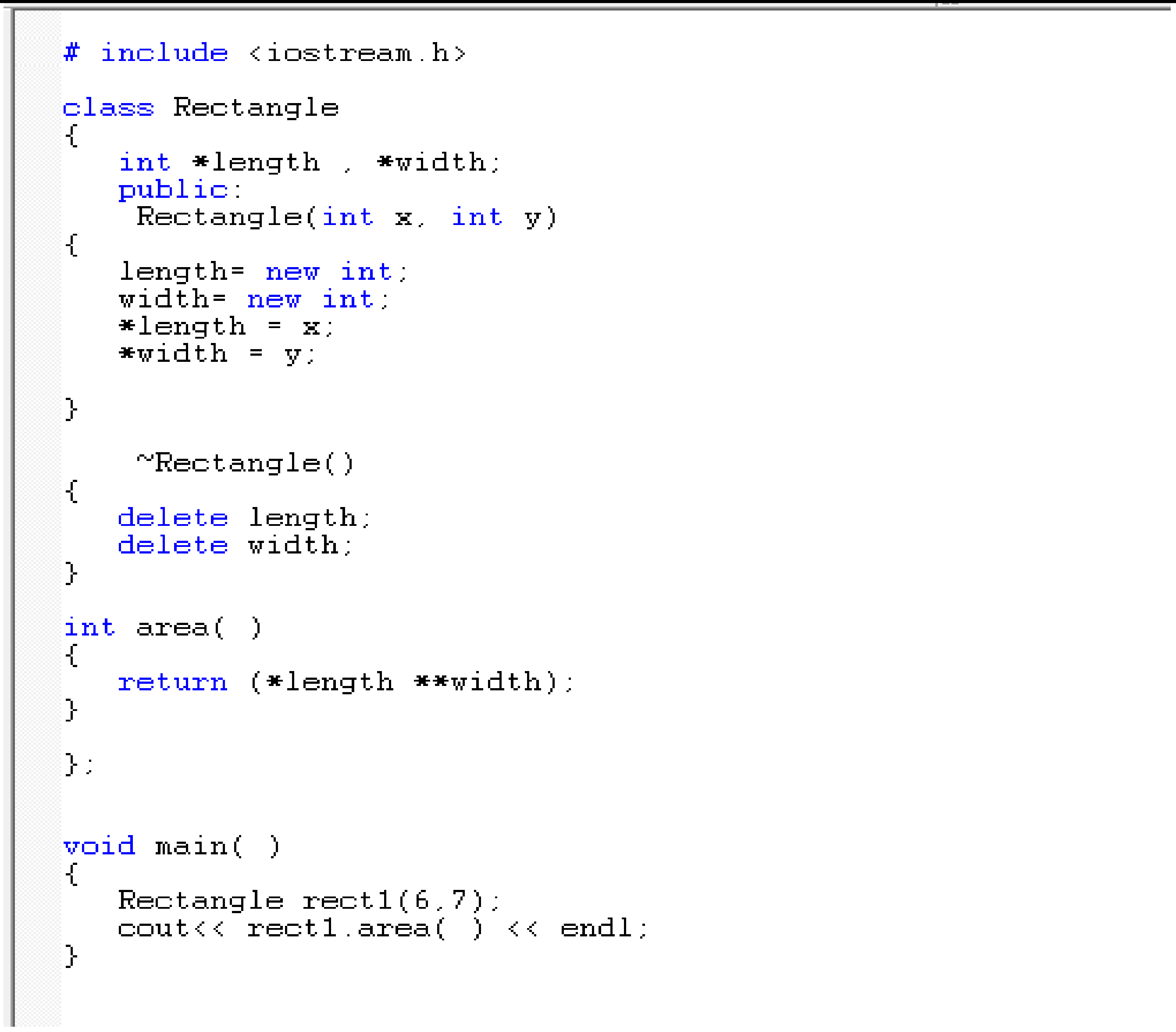
Just as a constructor is used to initialize an object when it is created, a destructor is used to clean up the object just before it is destroyed. A destructor always has the same name as the class itself, but is preceded with a ~ symbol. Unlike constructors, a class may have at most one destructor. A destructor never takes any arguments and has no explicit return type.Destructors are generally useful for classes which have pointer data members which point to memory blocks allocated by the class itself. In such cases it is important to release member-allocated memory before the object is destroyed. A destructor can do just that.

Example 1: Write an OOP to represent a simple destructor with class rectangle.



```
# include <iostream.h>

class Rectangle
{
   int length , width;
   public:
    Rectangle(int x, int y)             //constructor
{
   length = x;
   width = y;
}
    ~ Rectangle()                       //destructor
    {

    }


int area( )
{
   return (length *width);
}

};


void main( )
{
   Rectangle rect1(6,7);
   cout<< rect1.area( ) << endl;
}
```

Example2: Write an OOP to represent destructor of pointer members in class rectangle.



```
# include <iostream.h>

class Rectangle
{
   int *length , *width;
   public:
    Rectangle(int x, int y)
{
   length= new int;
   width= new int;
   *length = x;
   *width = y;

}

    ~Rectangle()
{
   delete length;
   delete width;
}

int area( )
{
   return (*length **width);
}

};


void main( )
{
   Rectangle rect1(6,7);
   cout<< rect1.area( ) << endl;
}
```

Note: A class destructor proceeded with a tilde (~).

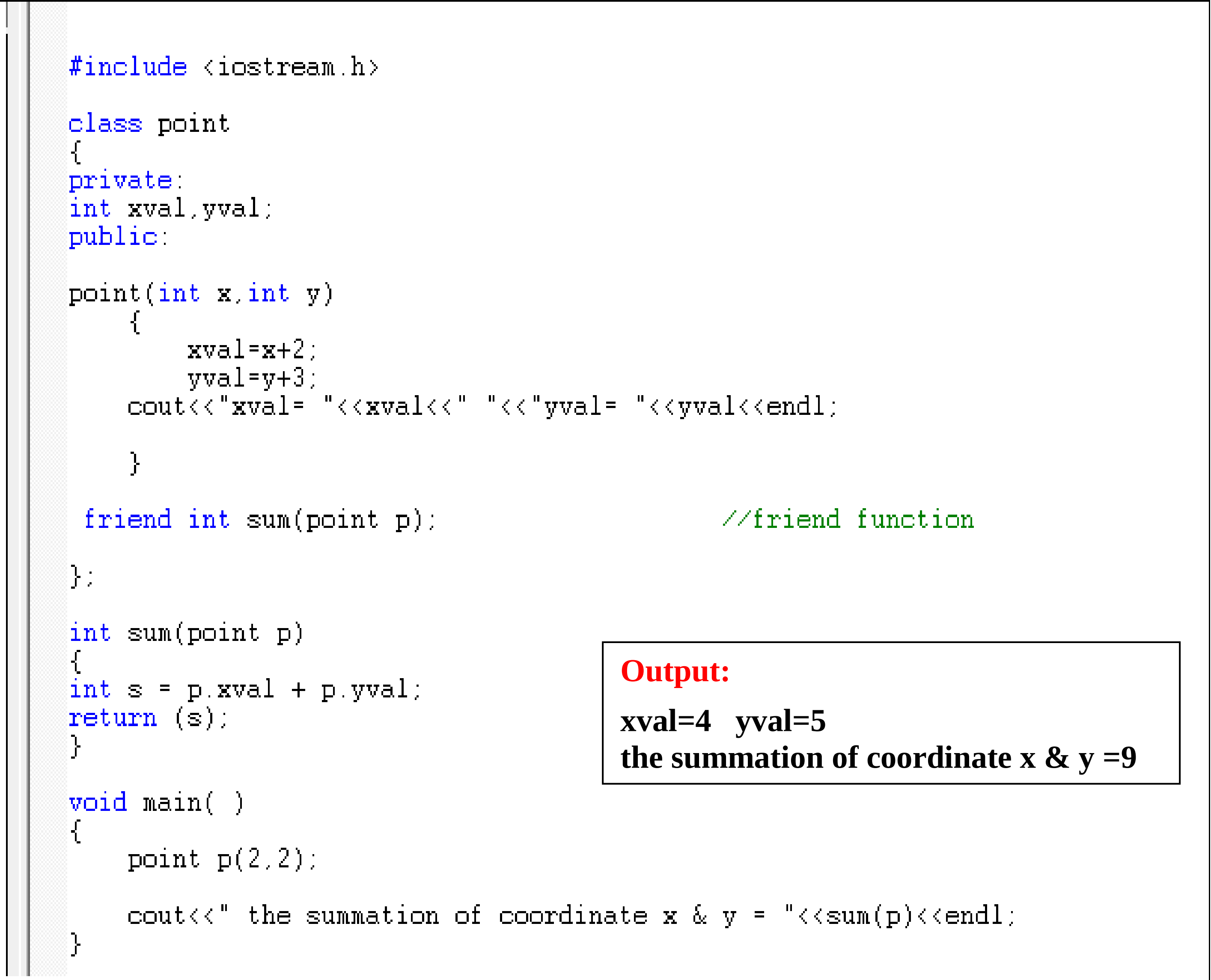
Friend function

Occasionally we may need to grant a function access to the nonpublic members of a class. Such an access is obtained by declaring the function a friend of the class. There are two possible reasons for requiring this access:

• It may be the only correct way of defining the function.

• It may be necessary if the function is to be implemented efficiently.

Example1: Write an OOP to find the summation of point coordinates using friend function.

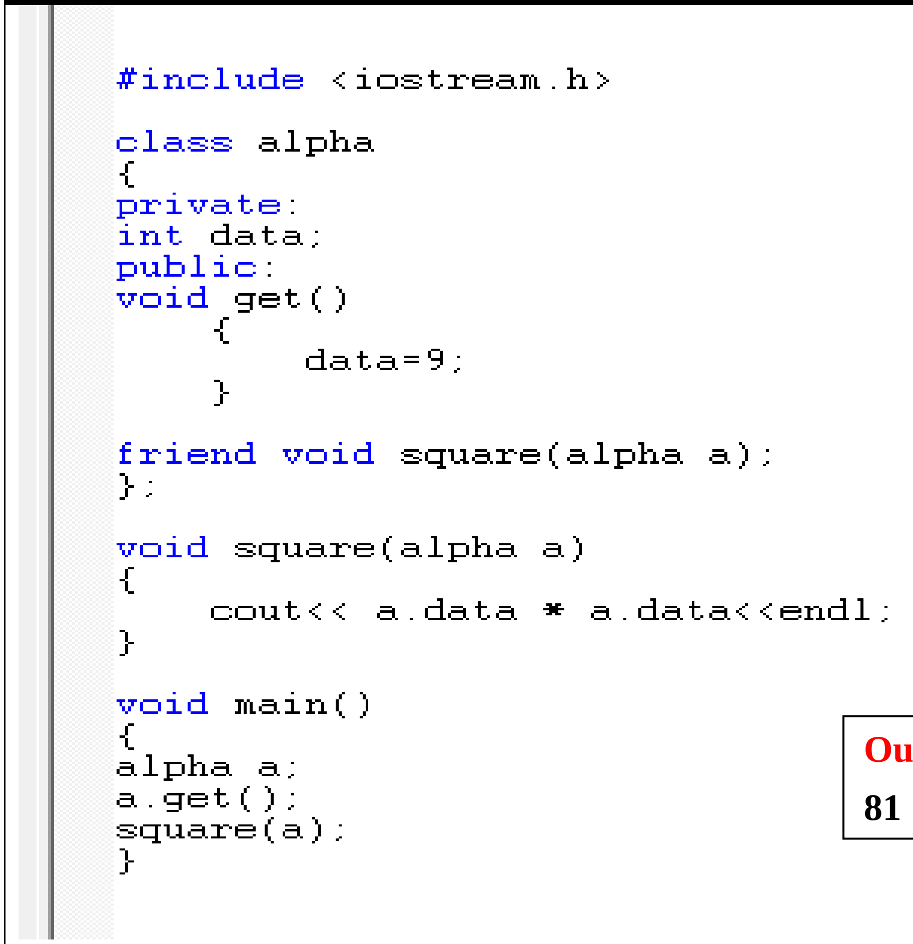


```
#include <iostream.h>

class point
{
private:
int xval,yval;
public:

point(int x,int y)
    {
        xval=x+2;
        yval=y+3;
    cout<<"xval= "<<xval<<" "<<"yval= "<<yval<<endl;

    }

 friend int sum(point p);                    //friend function

};

int sum(point p)
{
int s = p.xval + p.yval;
return (s);
}

void main( )
{
    point p(2,2);

    cout<<" the summation of coordinate x & y = "<<sum(p)<<endl;
}
```

Output:

```
xval=4   yval=5
the summation of coordinate x & y =9
```

Example2: Write an OOP to print the square value of the data in class alpha using friend function .

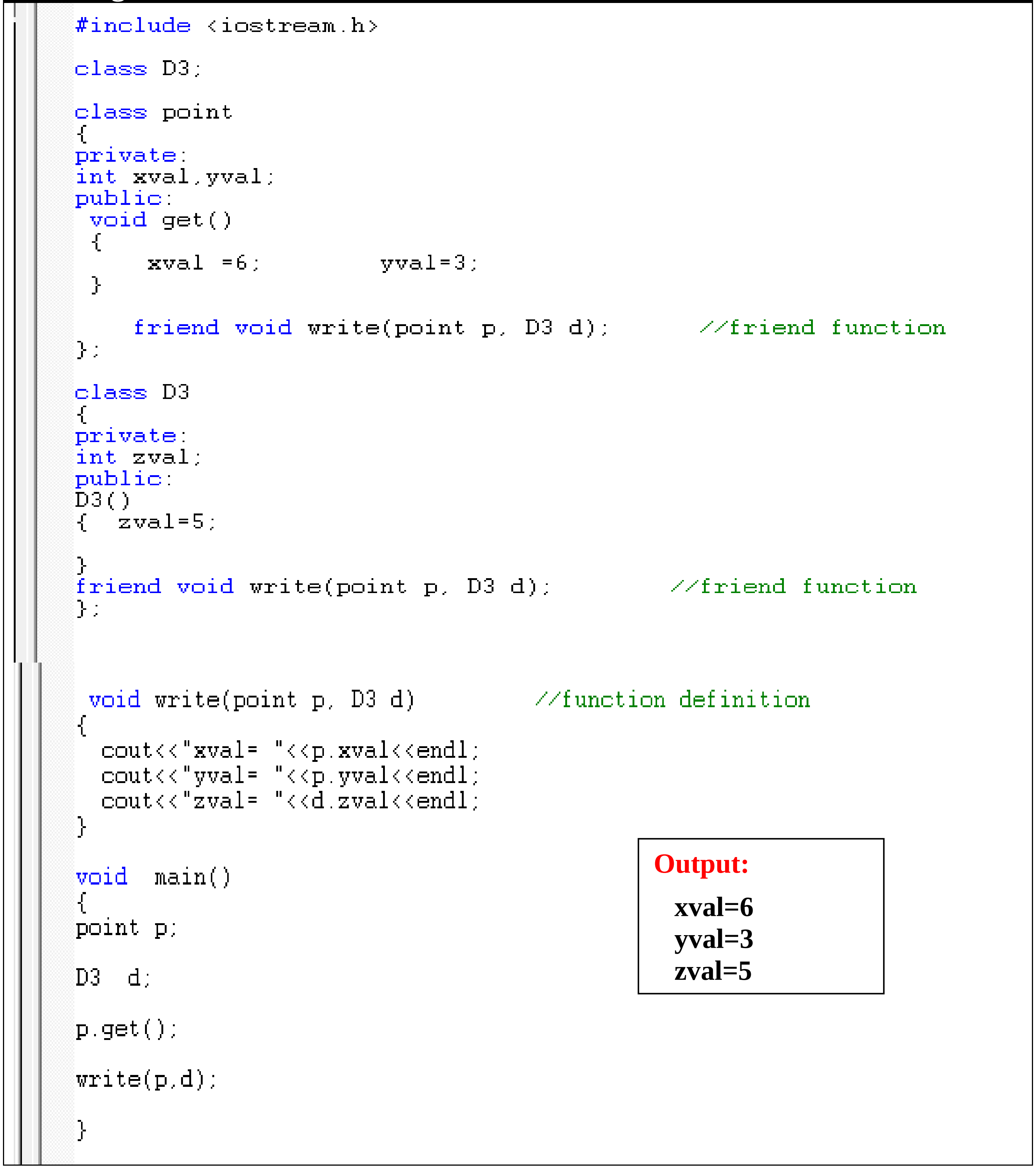


```
#include <iostream.h>

class alpha
{
private:
int data;
public:
void get()
    {
        data=9;
    }

friend void square(alpha a);
};

void square(alpha a)
{
    cout<< a.data * a.data<<endl;
}

void main()
{
alpha a;
a.get();
square(a);
}
```

Output:

81

Example 3: Write an OOP to print the coordinates of a class point and a class D3 using friend function.



```
#include <iostream.h>

class D3;

class point
{
private:
int xval,yval;
public:
 void get()
 {
     xval =6;          yval=3;
 }

    friend void write(point p, D3 d);        //friend function
};

class D3
{
private:
int zval;
public:
D3()
{  zval=5;

}
friend void write(point p, D3 d);        //friend function
};


 void write(point p, D3 d)         //function definition
{
  cout<<"xval= "<<p.xval<<endl;
  cout<<"yval= "<<p.yval<<endl;
  cout<<"zval= "<<d.zval<<endl;
}

void  main()
{
point p;

D3  d;

p.get();

write(p,d);

}
```

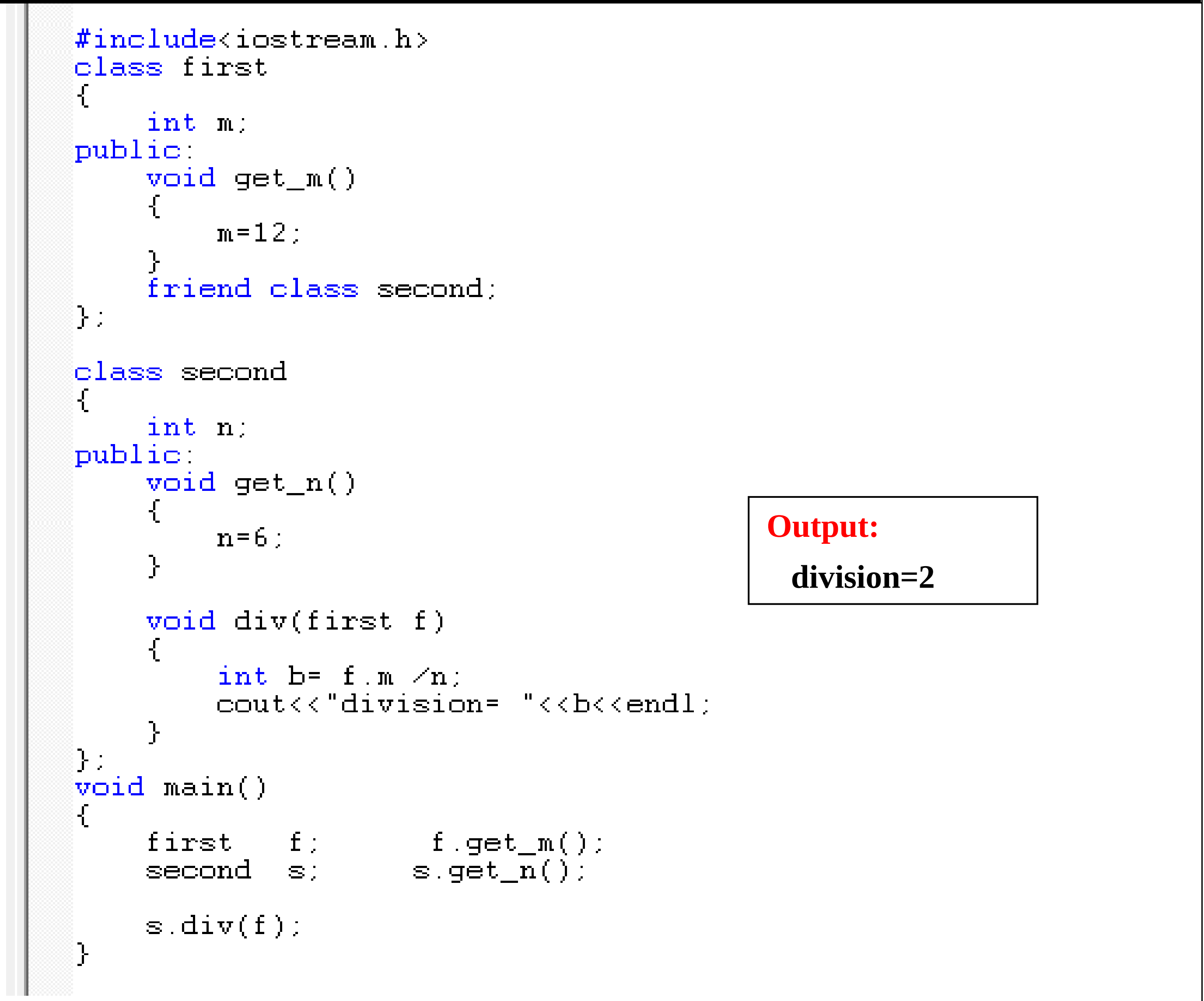
Output:

```
xval=6
yval=3
zval=5
```

friend class

The member functions of a class can all be made friends at the same time when you make the entire class a friend.

Example 1:Write an OOP to divide the value m in a class first on the value n in the class second using a friend class.

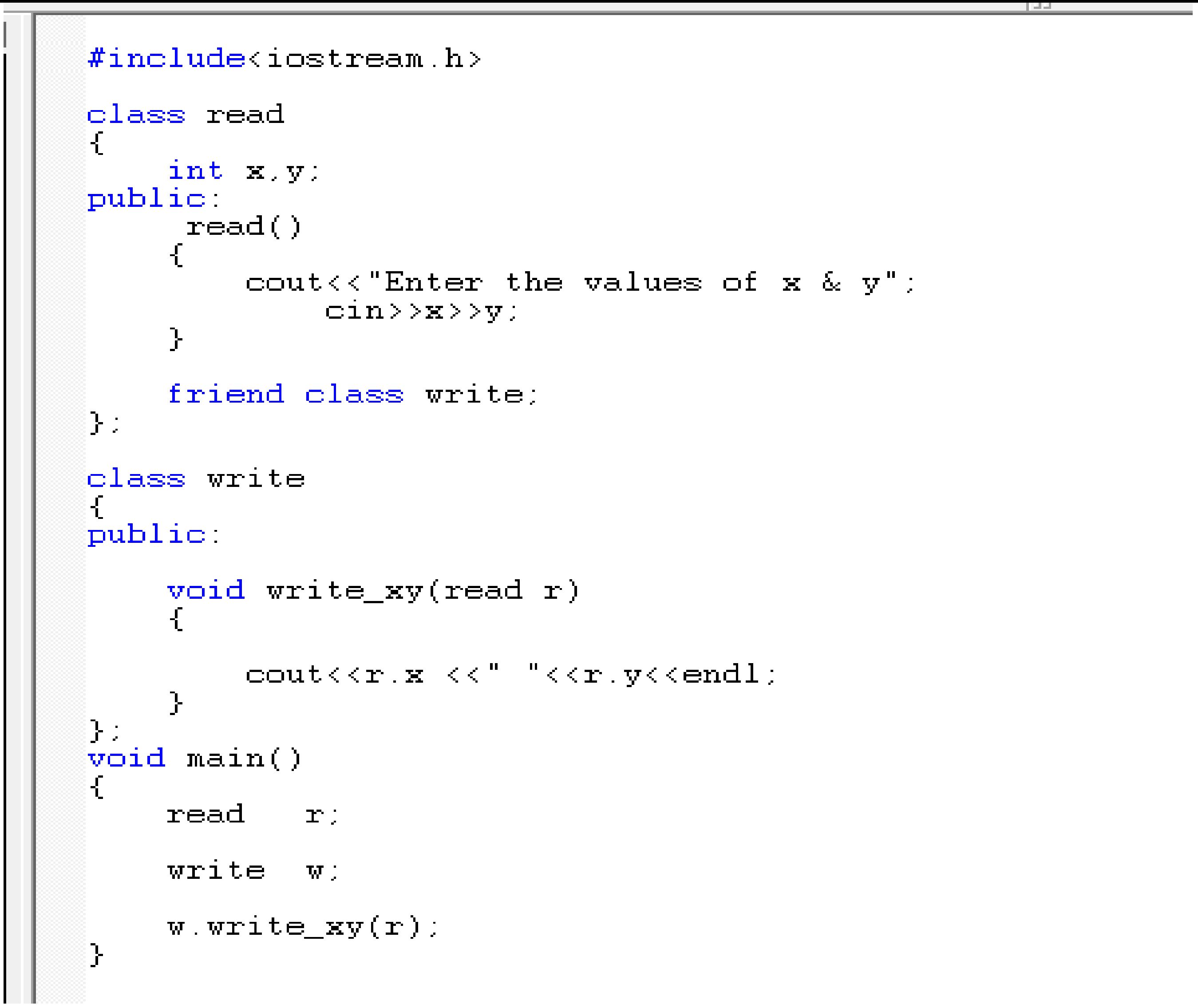


```
#include<iostream.h>
class first
{
	int m;
public:
	void get_m()
	{
		m=12;
	}
	friend class second;
};

class second
{
	int n;
public:
	void get_n()
	{
		n=6;
	}

	void div(first f)
	{
		int b= f.m /n;
		cout<<"division= "<<b<<endl;
	}
};
void main()
{
	first   f;       f.get_m();
	second  s;      s.get_n();

	s.div(f);
}
```

Output:

```
division=2
```

Example 2:Write an OOP to read the values x and y in a class read and write the values x and y in a class write using friend class



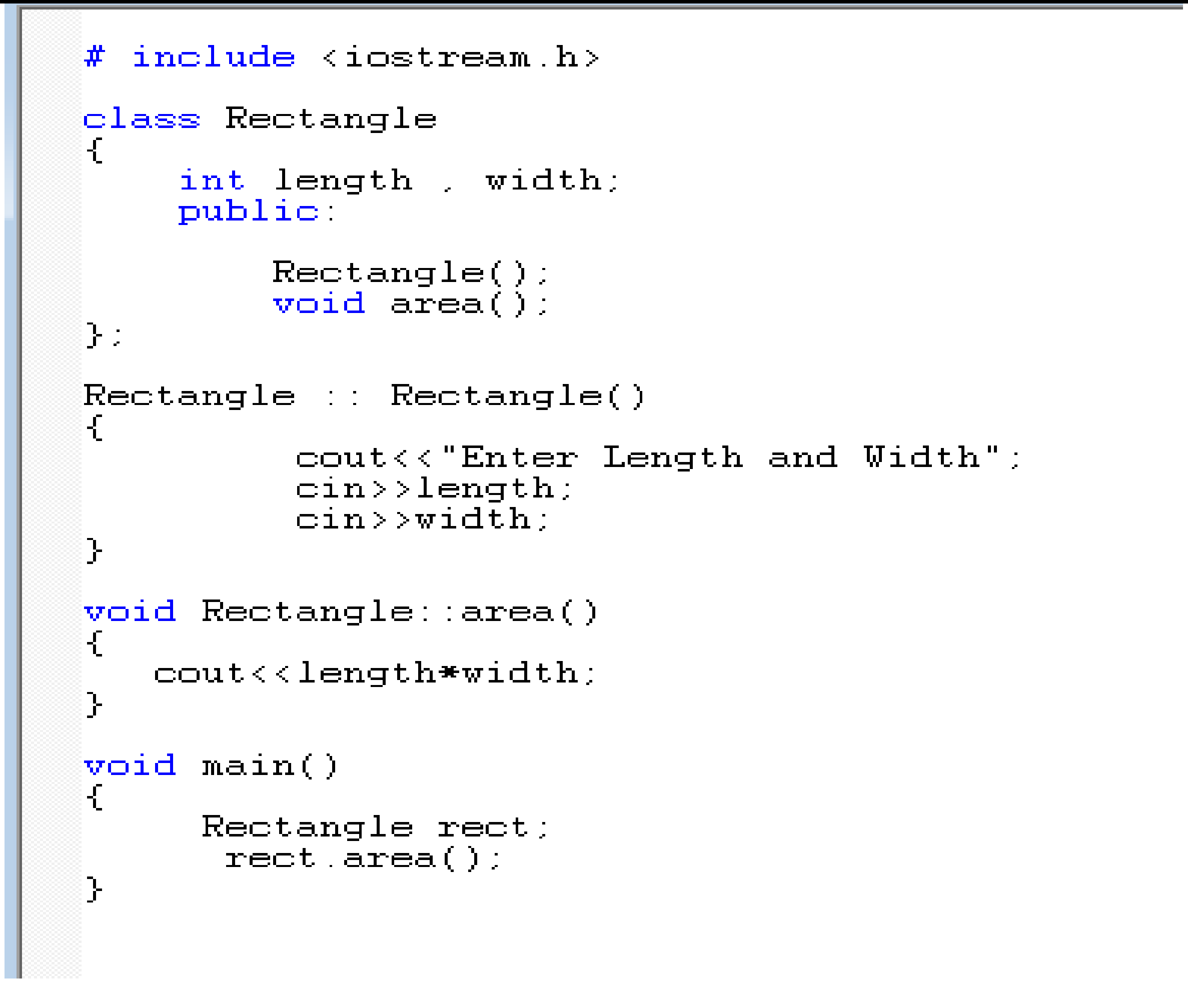
```
#include<iostream.h>

class read
{
    int x,y;
public:
     read()
    {
        cout<<"Enter the values of x & y";
            cin>>x>>y;
    }

    friend class write;
};

class write
{
public:

    void write_xy(read r)
    {

        cout<<r.x <<"  "<<r.y<<endl;
    }
};
void main()
{
    read   r;

    write  w;

    w.write_xy(r);
}
```

Scope Operator Resolution

When calling a member function, we usually use an abbreviated syntax. For example: **pt.OffsetPt(2,2); // abbreviated form**

This is equivalent to the full form: pt.Point::OffsetPt(2,2); // full form

Example 1: write an OOP to find the area of rectangle using the concept of Scope Operator Resolution



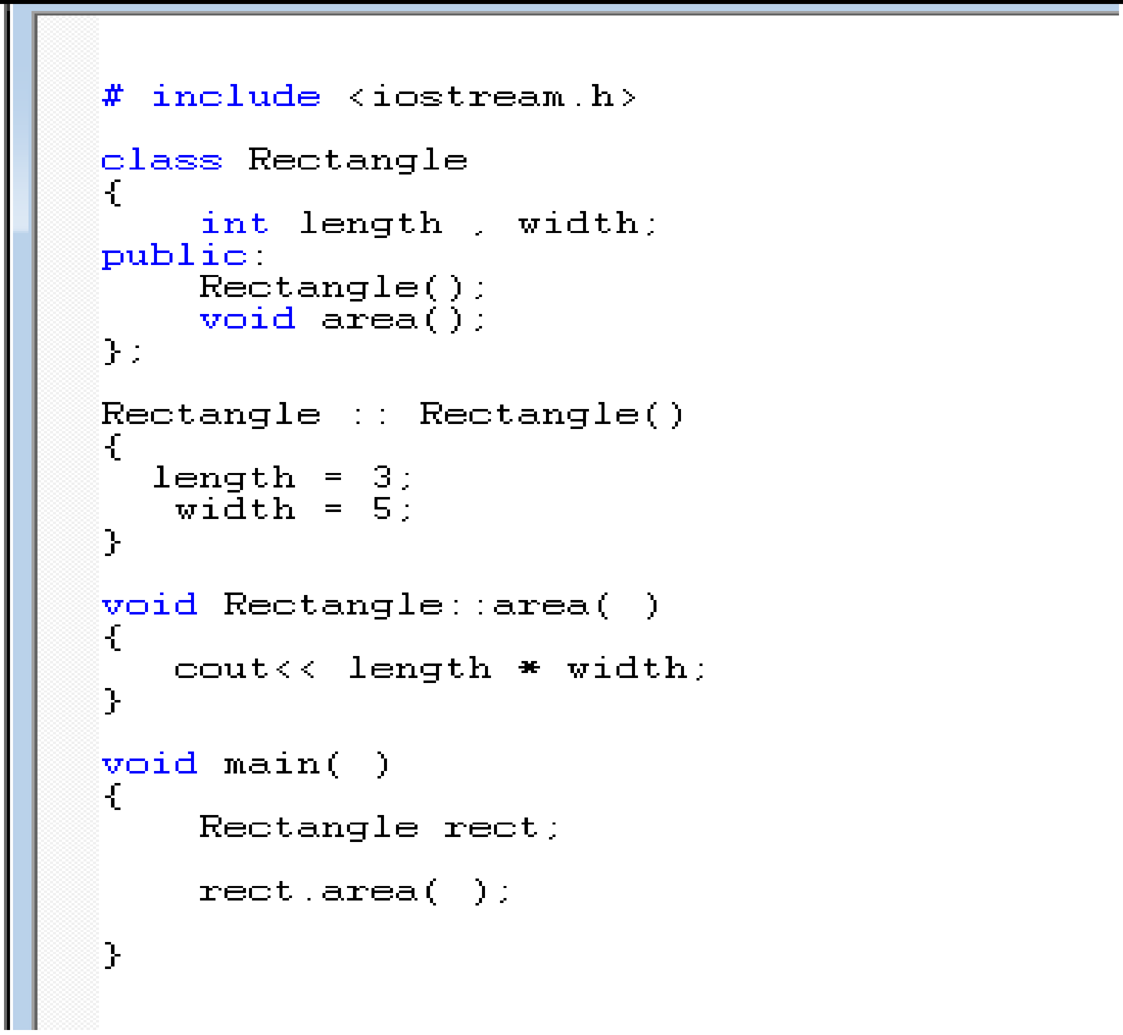
```
# include <iostream.h>

class Rectangle
{
    int length , width;
    public:

        Rectangle();
        void area();
};

Rectangle :: Rectangle()
{
          cout<<"Enter Length and Width";
          cin>>length;
          cin>>width;
}

void Rectangle::area()
{
   cout<<length*width;
}

void main()
{
     Rectangle rect;
      rect.area();
}
```

Member Initialization List

There are two ways of initializing the data members of a class.

1) The first approach involves initializing the data members using assignments in the body of a constructor. For example:

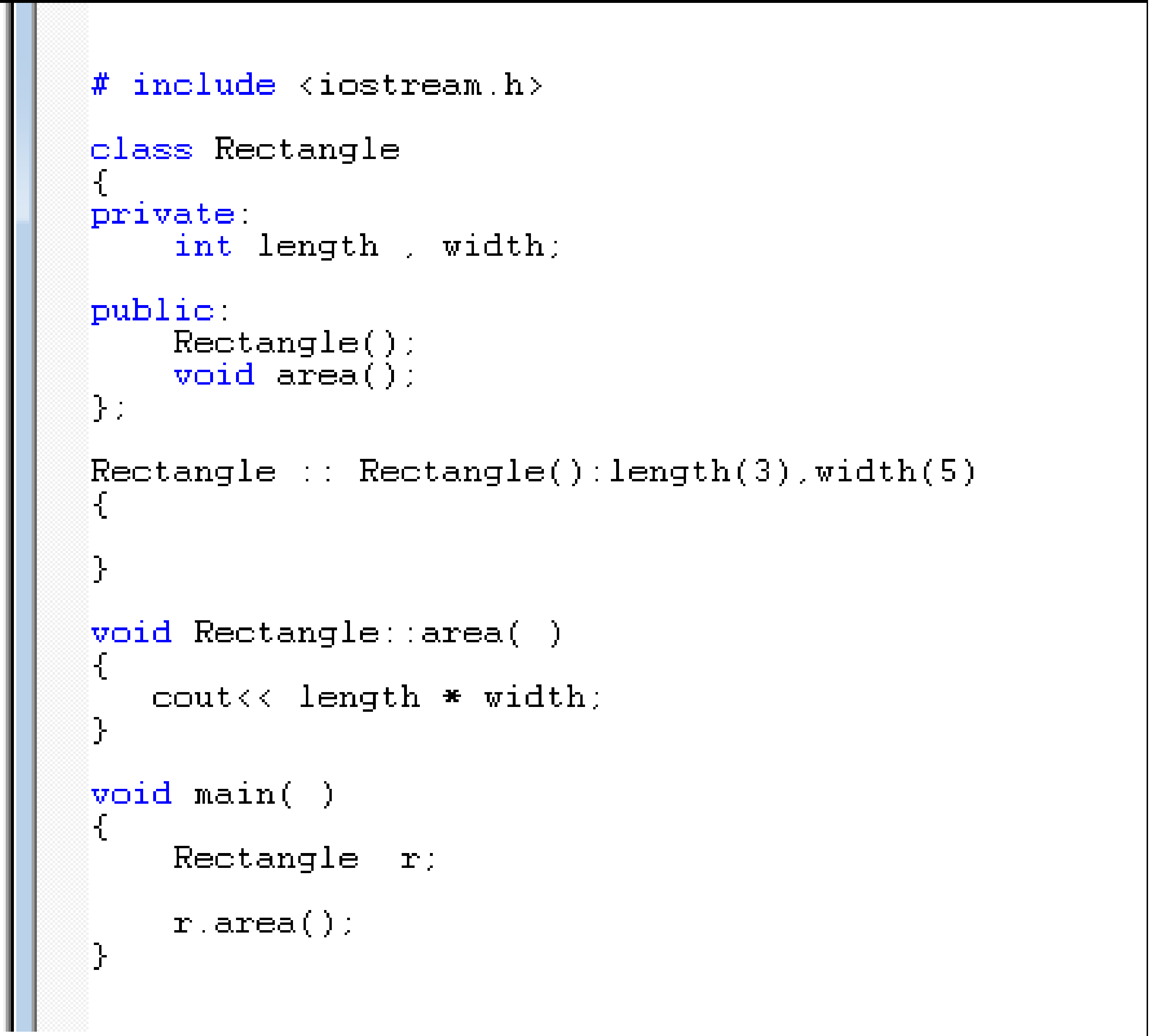
Example 1: Write an OOP to initialize the data member of a class using assignments in the body of constructor.



```
# include <iostream.h>

class Rectangle
{
     int length , width;
public:
     Rectangle();
     void area();
};

Rectangle :: Rectangle()
{
   length = 3;
    width = 5;
}

void Rectangle::area( )
{
    cout<< length * width;
}

void main( )
{
     Rectangle rect;

     rect.area( );

}
```

2) The second approach uses a member initialization list in the definition of a constructor. For example:

Example 2: Write an OOP to initialize the data member of a class in the definition of constructor.



```
# include <iostream.h>

class Rectangle
{
private:
     int length , width;

public:
     Rectangle();
     void area();
};

Rectangle :: Rectangle():length(3),width(5)
{

}

void Rectangle::area( )
{
   cout<< length * width;
}

void main( )
{
     Rectangle  r;

     r.area();
}
```

Constant member

A class data member may define as constant.

Constant Function Argument

Suppose you want to pass an argument by reference for efficiency, but not only do you want the function not to modify it, you want a guarantee that the function *cannot* modify it. To obtain such a guarantee, you can apply the const modifier to the variable in the function declaration.

Example 1:Write a C++ simple program to represent a constant argument

```
//constarg.cpp
//demonstrates constant function arguments

void aFunc(int& a, const int& b);  //declaration

int main()
   {
   int alpha = 7;
   int beta = 11;
   aFunc(alpha, beta);
   return 0;
   }
//------------------------------------------------------------
void aFunc(int& a, const int& b)   //definition
   {
   a = 107;   //OK
   b = 111;   //error: can't modify constant argument
   }
```

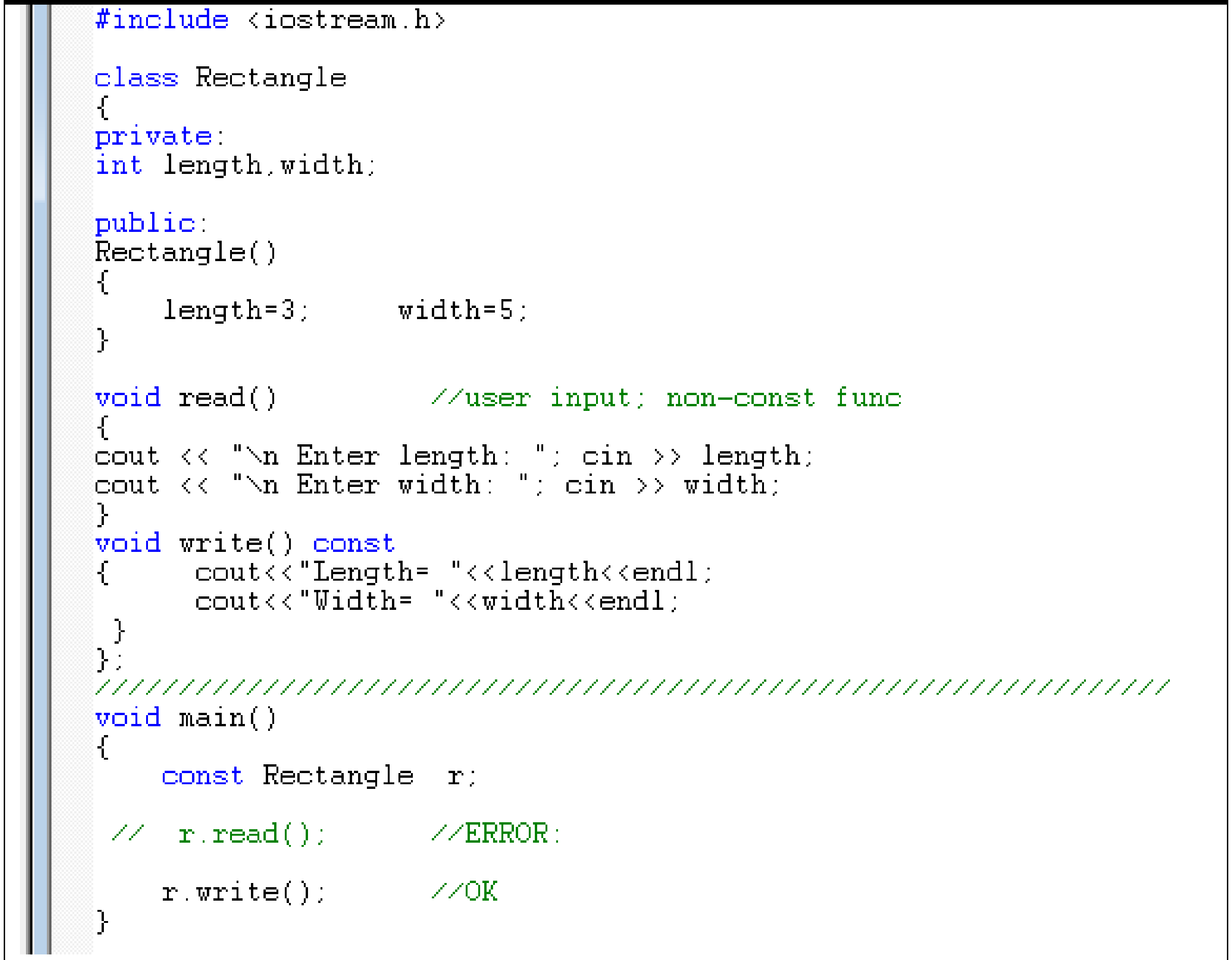
Here we want to be sure that aFunc() can’t modify the variable beta. (We don’t care if it modifies alpha.) So we use the const modifier with beta in the function declaration (and definition):

void aFunc(int& alpha, const int& beta);

Constant Member Functions

We can apply **const** to variables of basic types such as int to keep them from being modified. In a similar way, we can apply const to objects of classes. When an object is declared as const, you can't modify it.

Example 2:Write an OOP to represent a constant member function

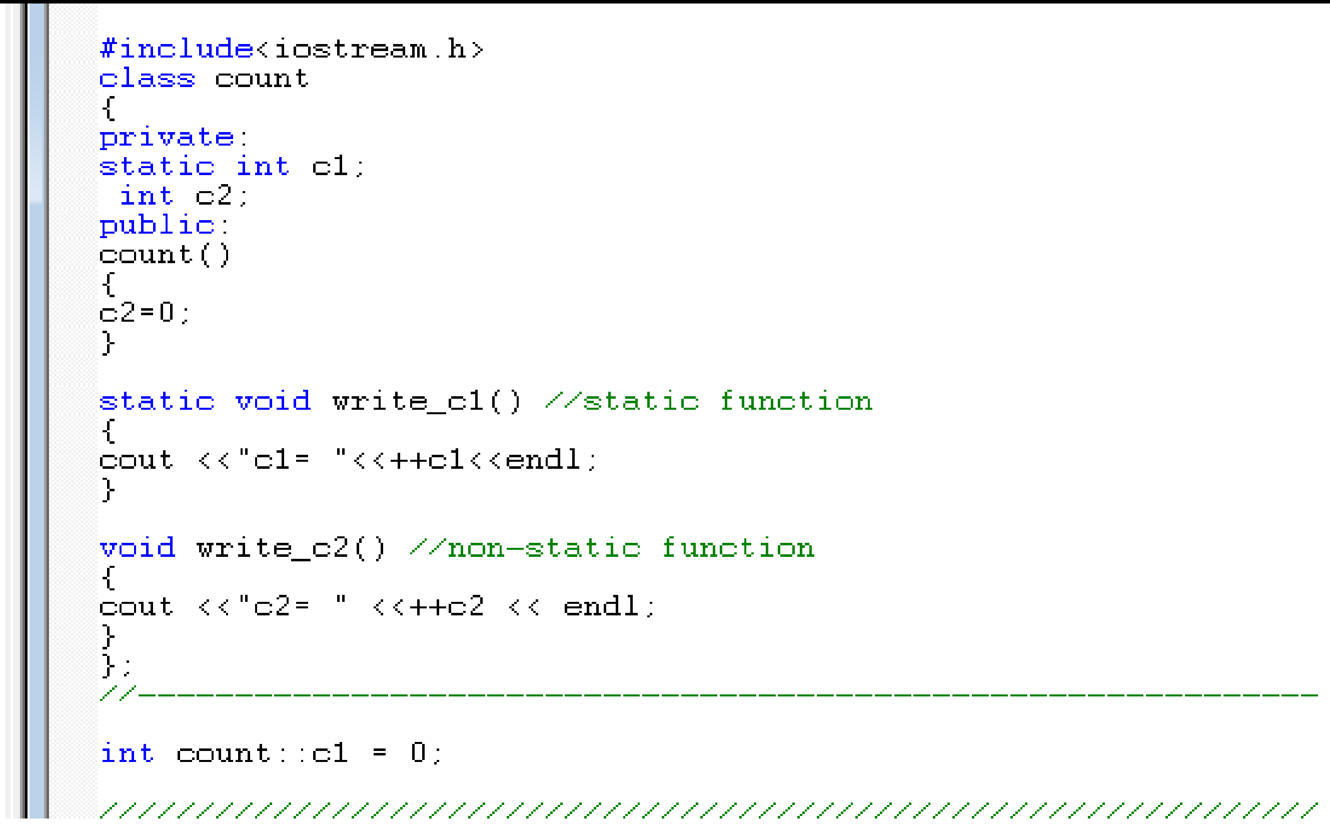


```
#include <iostream.h>

class Rectangle
{
private:
int length,width;

public:
Rectangle()
{
    length=3;      width=5;
}

void read()          //user input; non-const func
{
cout << "\n Enter length: "; cin >> length;
cout << "\n Enter width: "; cin >> width;
}
void write() const
{     cout<<"Length= "<<length<<endl;
      cout<<"Width= "<<width<<endl;
 }
};
////////////////////////////////////////////////////////////////
void main()
{
    const Rectangle  r;

 //   r.read();      //ERROR:

    r.write();      //OK
}
```

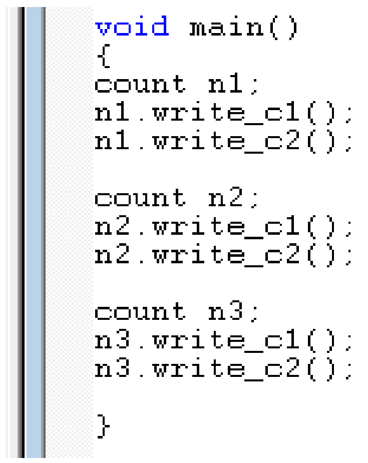
Static Members

A data member of a class can be defined to be static. This ensures that there will be exactly one copy of the member, shared by all objects of the class.

Example 1: Write an OOP to represent a static members for count class.



```
#include<iostream.h>
class count
{
private:
static int c1;
 int c2;
public:
count()
{
c2=0;
}

static void write_c1() //static function
{
cout <<"c1= "<<++c1<<endl;
}

void write_c2() //non-static function
{
cout <<"c2= " <<++c2 << endl;
}
};
//---------------------------------------------------------------

int count::c1 = 0;

///////////////////////////////////////////////////////////////
```

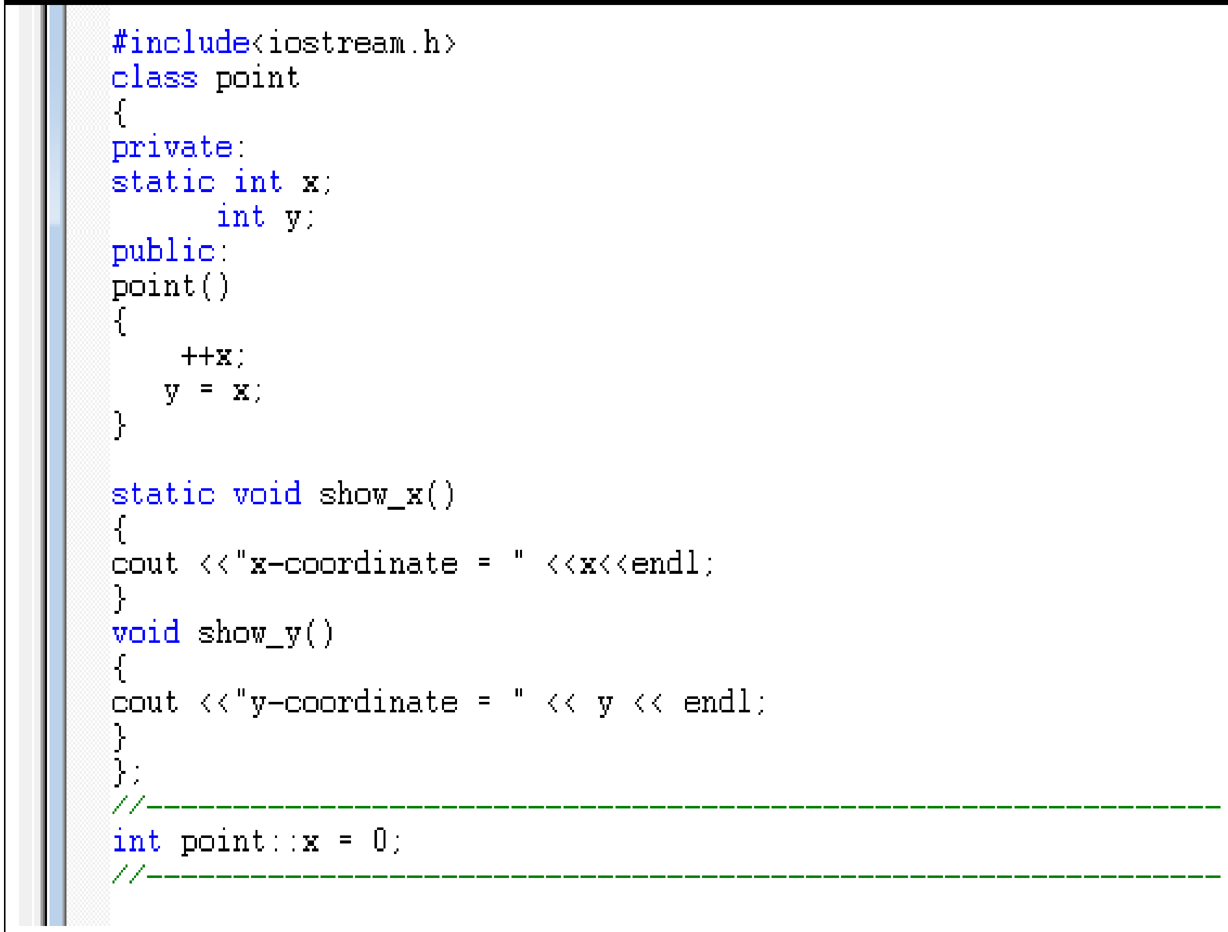


```
void main()
{
count n1;
n1.write_c1();
n1.write_c2();

count n2;
n2.write_c1();
n2.write_c2();

count n3;
n3.write_c1();
n3.write_c2();

}
```

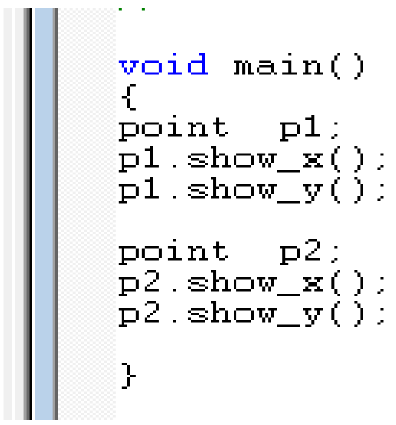
Output:

```
c1=1
c2=1
c1=2
c2=1
c1=3
c2=1
```

Example 2: Write an OOP to represent a static members for point class.



```
#include<iostream.h>
class point
{
private:
static int x;
       int y;
public:
point()
{
    ++x;
   y = x;
}

static void show_x()
{
cout <<"x-coordinate = " <<x<<endl;
}
void show_y()
{
cout <<"y-coordinate = " << y << endl;
}
};
//----------------------------------------------------------------
int point::x = 0;
//----------------------------------------------------------------
```



```
void main()
{
point  p1;
p1.show_x();
p1.show_y();

point  p2;
p2.show_x();
p2.show_y();

}
```

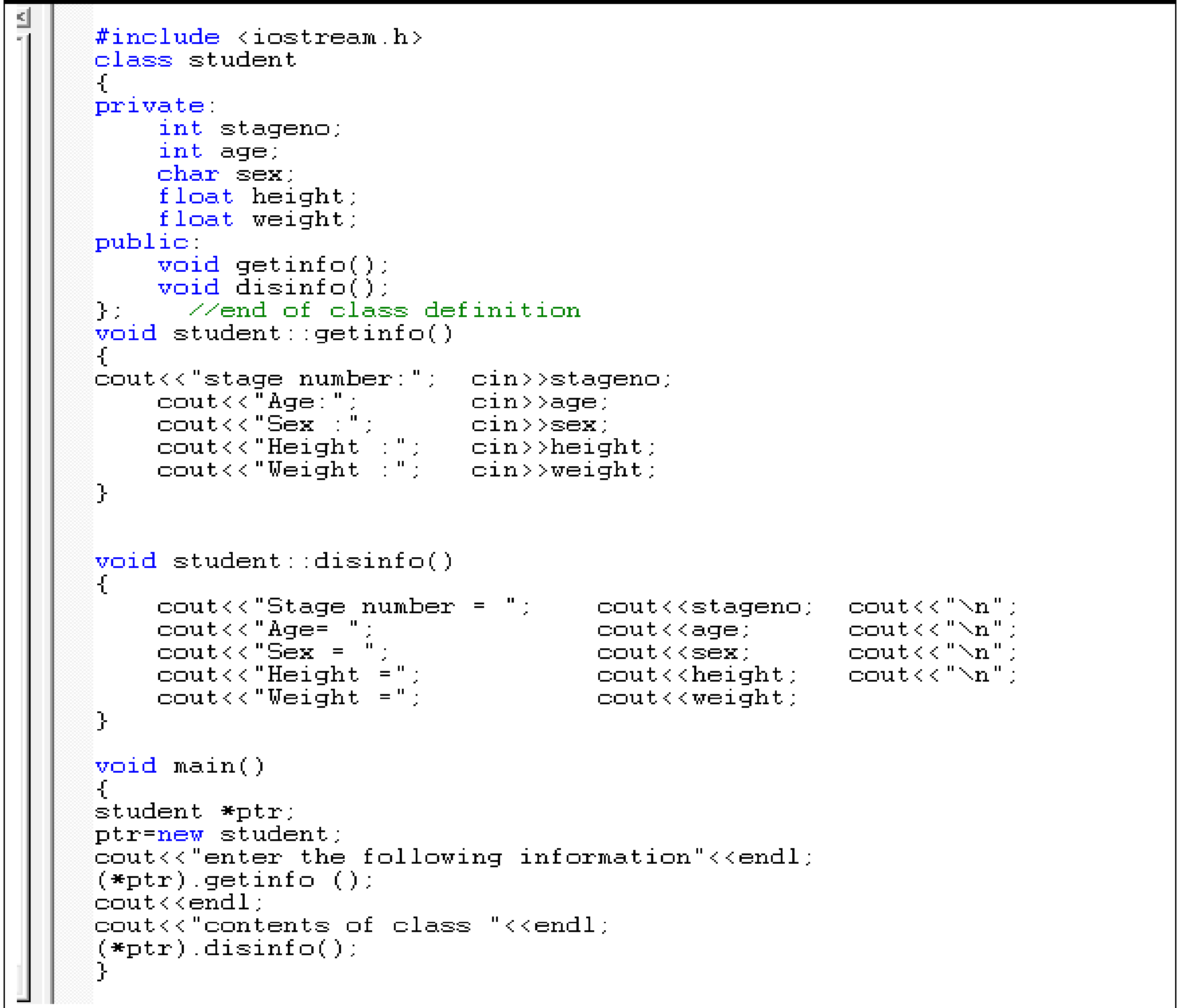
Output:
x-coordinate= 1
y-coordinate= 1
x-coordinate= 2
y-coordinate=2

Objects Pointers

It has already been stated that a pointer is a variable which holds the memory address of another variable of any basic data type. It has been shown that how a pointer variable can be declared with a class.

First way :- ***(*object name).member name=variable;***

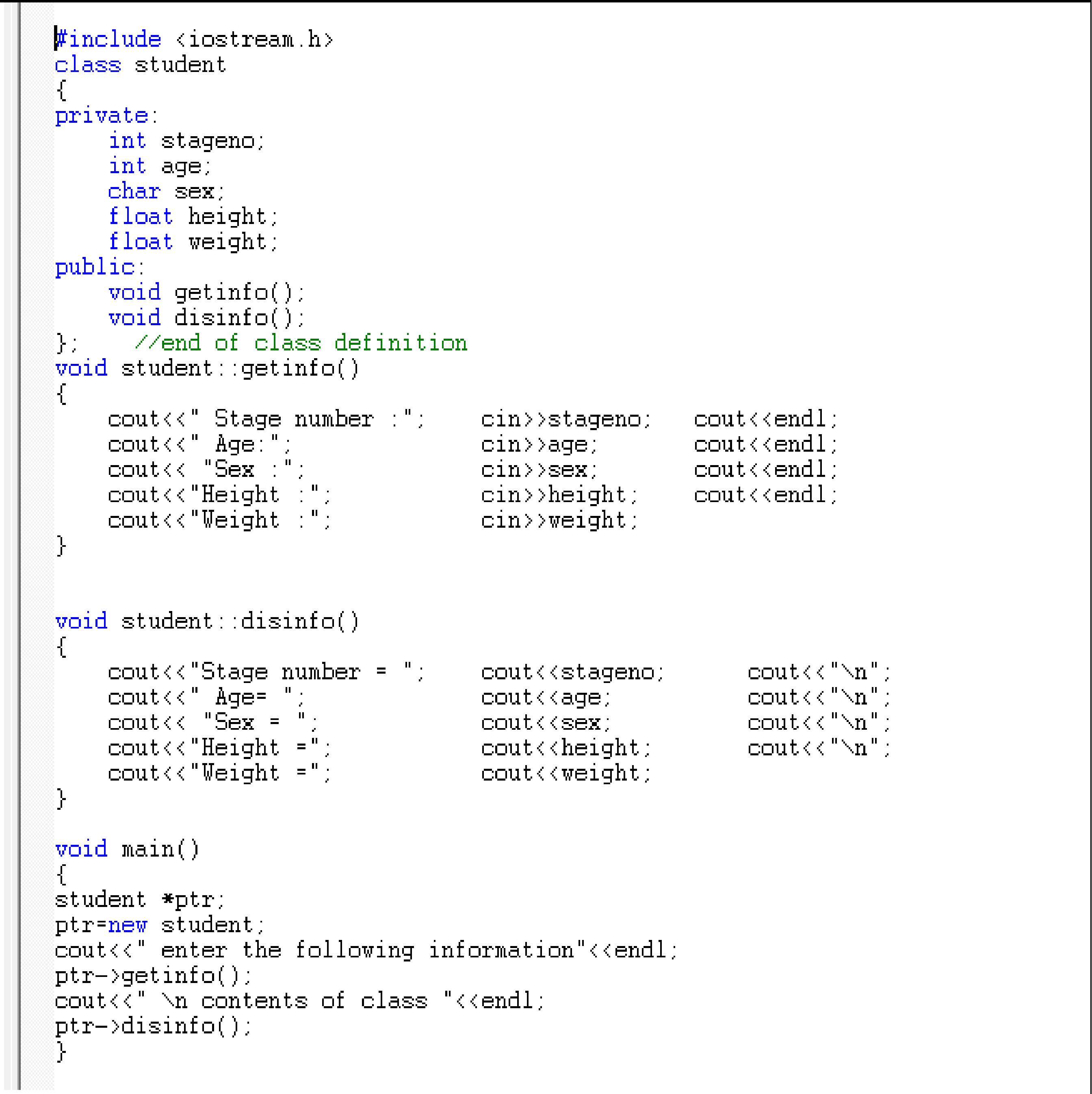
Example 1: Write an OOP to read and display student information using pointer.



```
#include <iostream.h>
class student
{
private:
     int stageno;
     int age;
     char sex;
     float height;
     float weight;
public:
     void getinfo();
     void disinfo();
};     //end of class definition
void student::getinfo()
{
cout<<"stage number:";   cin>>stageno;
     cout<<"Age:";        cin>>age;
     cout<<"Sex  :";      cin>>sex;
     cout<<"Height  :";   cin>>height;
     cout<<"Weight  :";   cin>>weight;
}


void student::disinfo()
{
     cout<<"Stage number =  ";    cout<<stageno;   cout<<"\n";
     cout<<"Age=  ";              cout<<age;       cout<<"\n";
     cout<<"Sex  =  ";            cout<<sex;       cout<<"\n";
     cout<<"Height  =";           cout<<height;    cout<<"\n";
     cout<<"Weight  =";           cout<<weight;
}

void main()
{
student *ptr;
ptr=new student;
cout<<"enter the following information"<<endl;
(*ptr).getinfo ();
cout<<endl;
cout<<"contents of class "<<endl;
(*ptr).disinfo();
}
```

<u>Second way:-</u> ***object name ->member name=variable;***

Example 2: Write an OOP to read and display student information using pointer.

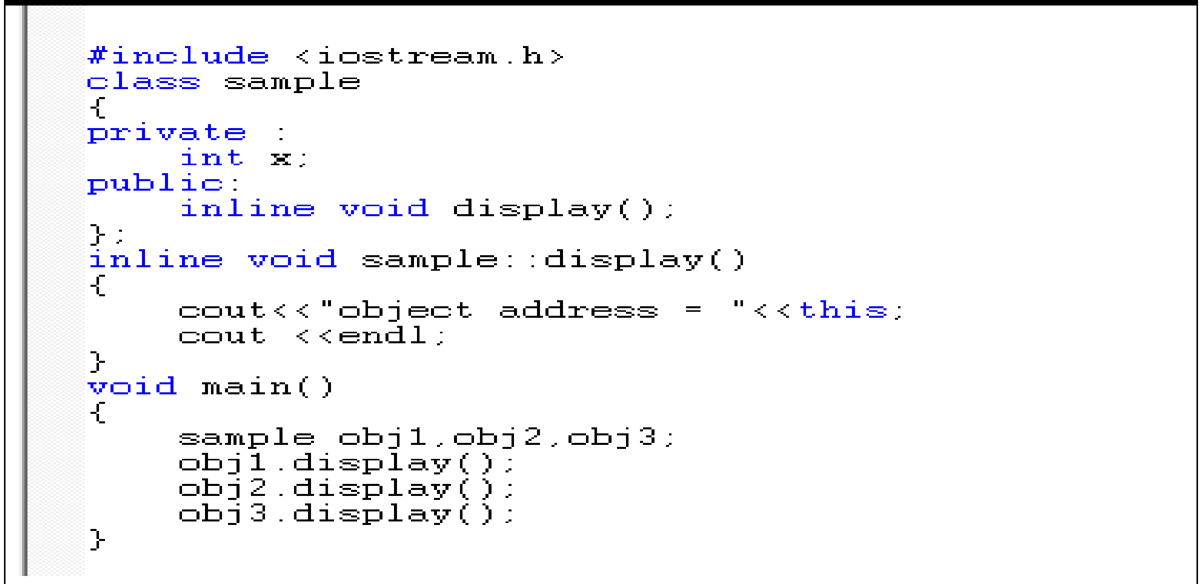


```
#include <iostream.h>
class student
{
private:
    int stageno;
    int age;
    char sex;
    float height;
    float weight;
public:
    void getinfo();
    void disinfo();
};    //end of class definition
void student::getinfo()
{
    cout<<" Stage number :";    cin>>stageno;   cout<<endl;
    cout<<" Age:";              cin>>age;       cout<<endl;
    cout<< "Sex :";             cin>>sex;       cout<<endl;
    cout<<"Height :";           cin>>height;    cout<<endl;
    cout<<"Weight :";           cin>>weight;
}


void student::disinfo()
{
    cout<<"Stage number = ";    cout<<stageno;      cout<<"\n";
    cout<<" Age= ";             cout<<age;          cout<<"\n";
    cout<< "Sex = ";            cout<<sex;          cout<<"\n";
    cout<<"Height =";           cout<<height;       cout<<"\n";
    cout<<"Weight =";           cout<<weight;
}

void main()
{
student *ptr;
ptr=new student;
cout<<" enter the following information"<<endl;
ptr->getinfo();
cout<<" \n contents of class "<<endl;
ptr->disinfo();
}
```

This pointer

This pointer is a variable which is used to access the address of the class itself.

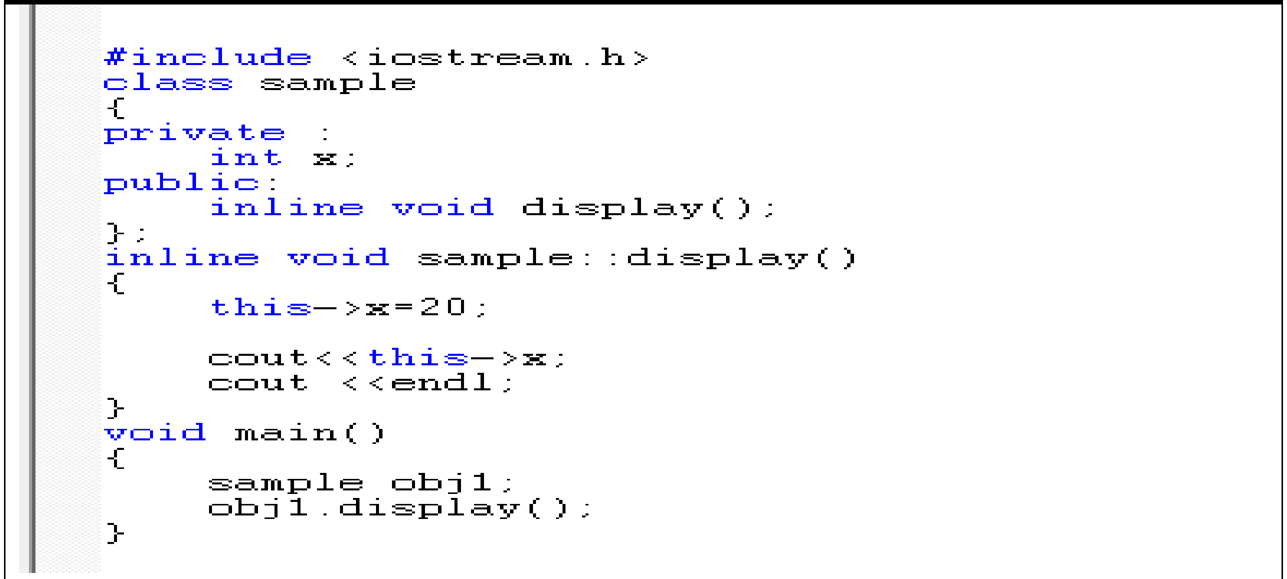
Example 1:Write an OOP to display the address of class using *this* pointer



```
#include <iostream.h>
class sample
{
private :
        int x;
public:
        inline void display();
};
inline void sample::display()
{
        cout<<"object address = "<<this;
        cout <<endl;
}
void main()
{
        sample obj1,obj2,obj3;
        obj1.display();
        obj2.display();
        obj3.display();
}
```

The above program create three objects,**obj1, obj2, obj3** and displays each object’s address using ***this*** pointer.

Example 2:Write an OOP to display the content of class member using *this* pointer

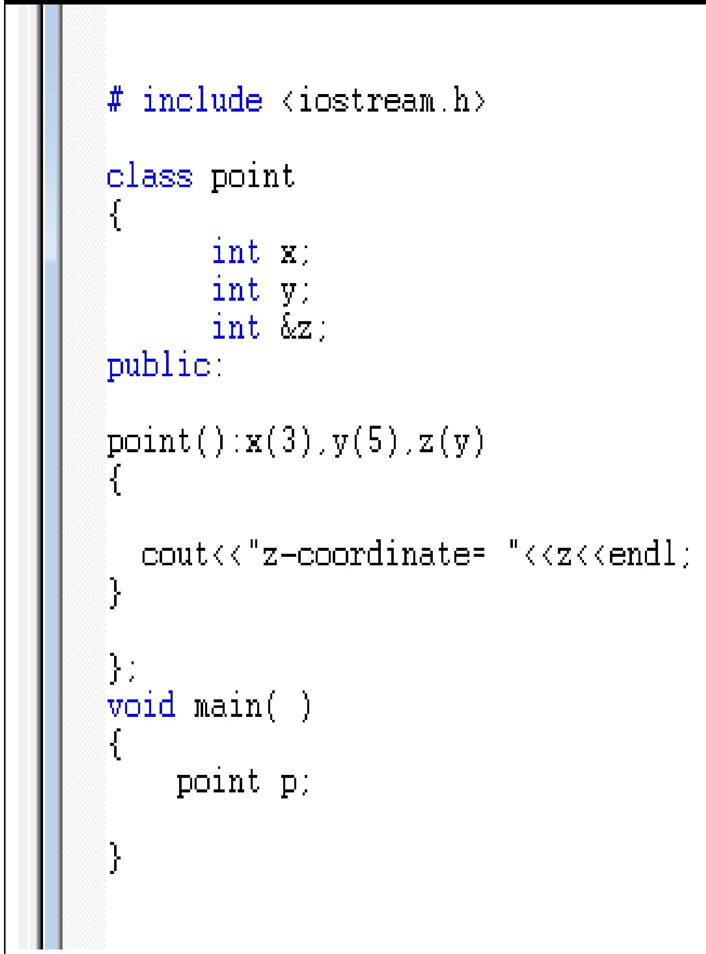


```
#include <iostream.h>
class sample
{
private :
        int x;
public:
        inline void display();
};
inline void sample::display()
{
        this->x=20;

        cout<<this->x;
        cout <<endl;
}
void main()
{
        sample obj1;
        obj1.display();
}
```

References Members

A class data member may define as reference. For example:

Example 1:Write an OOP to find the value of third coordinate z using reference member.



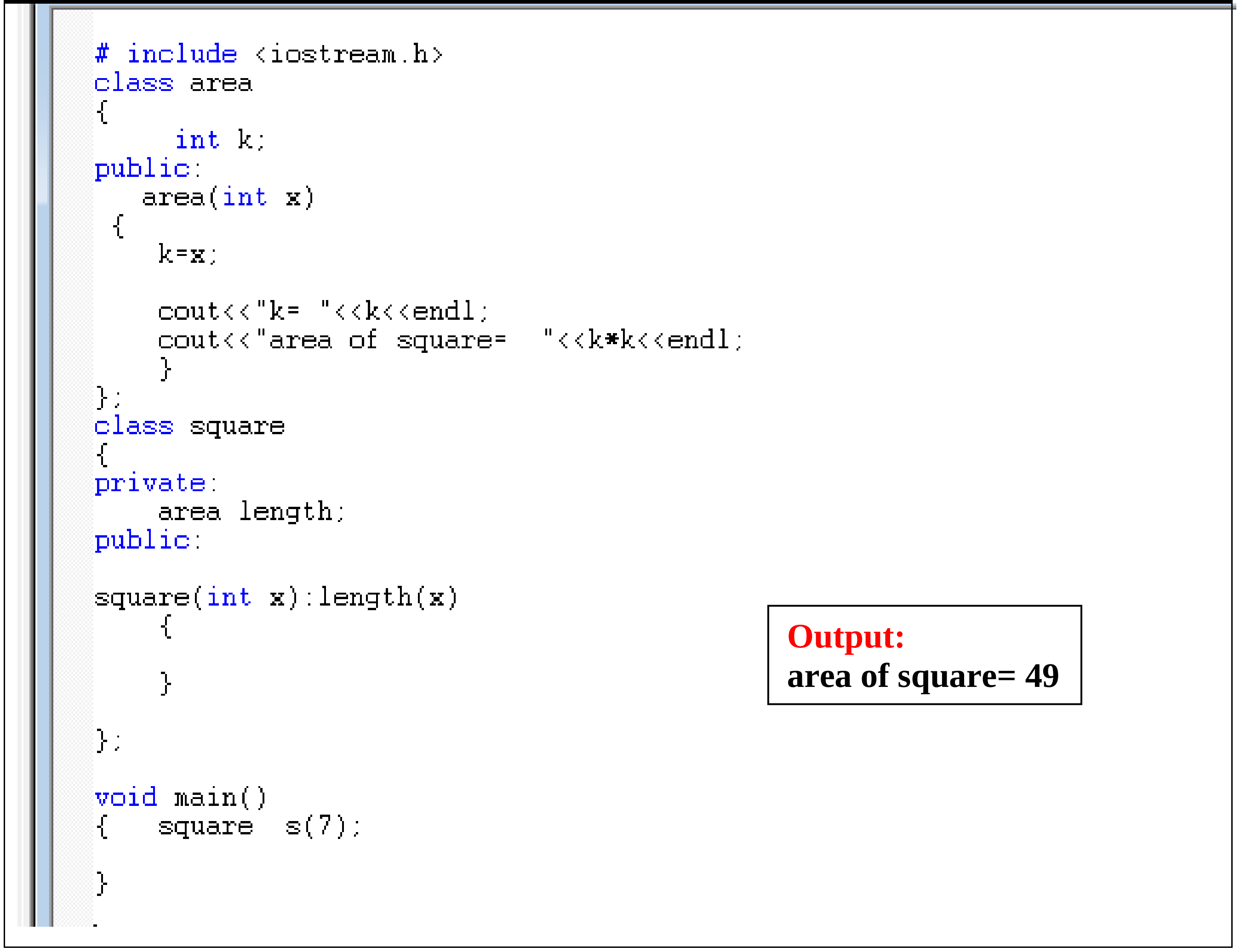
```
# include <iostream.h>

class point
{
      int x;
      int y;
      int &z;
public:

point():x(3),y(5),z(y)
{

  cout<<"z-coordinate= "<<z<<endl;
}

};
void main( )
{
    point p;

}
```

Output:
z-coordinate= 5

Class Object Member

A data member of a class may be of a user-defined type, that is, an object of another class.

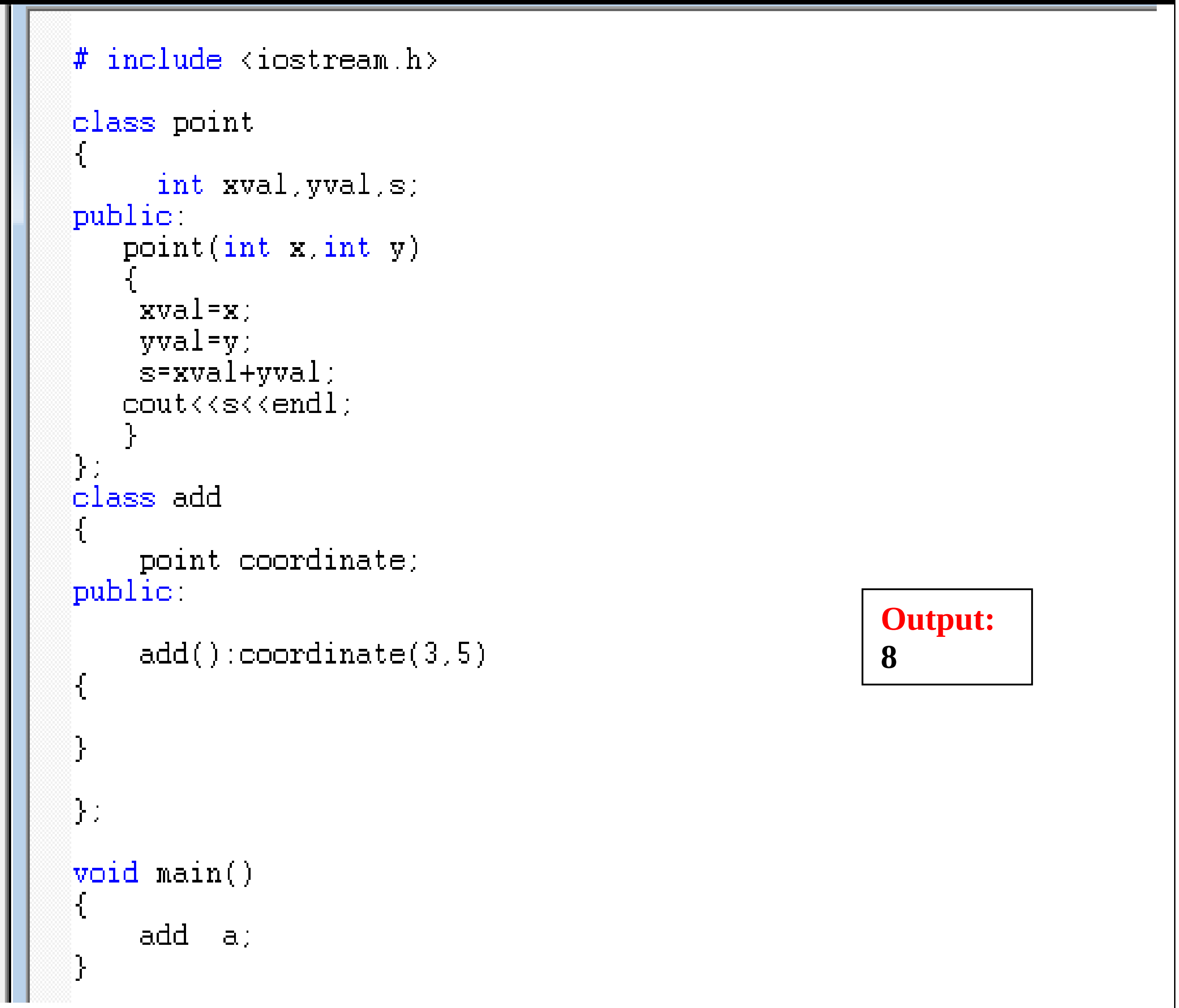
Example 1:Write an OOP to find the area of square using class object member



```
# include <iostream.h>
class area
{
     int k;
public:
   area(int x)
 {
    k=x;

    cout<<"k= "<<k<<endl;
    cout<<"area of square=  "<<k*k<<endl;
    }
};
class square
{
private:
    area length;
public:

square(int x):length(x)
    {

    }

};

void main()
{   square  s(7);

}
```

Output:
area of square= 49

Example 2: Write an OOP to find the summation of point coordinates using class object member



```
# include <iostream.h>

class point
{
     int xval,yval,s;
public:
   point(int x,int y)
   {
    xval=x;
    yval=y;
    s=xval+yval;
   cout<<s<<endl;
   }
};
class add
{
    point coordinate;
public:

    add():coordinate(3,5)
{

}

};

void main()
{
    add  a;
}
```

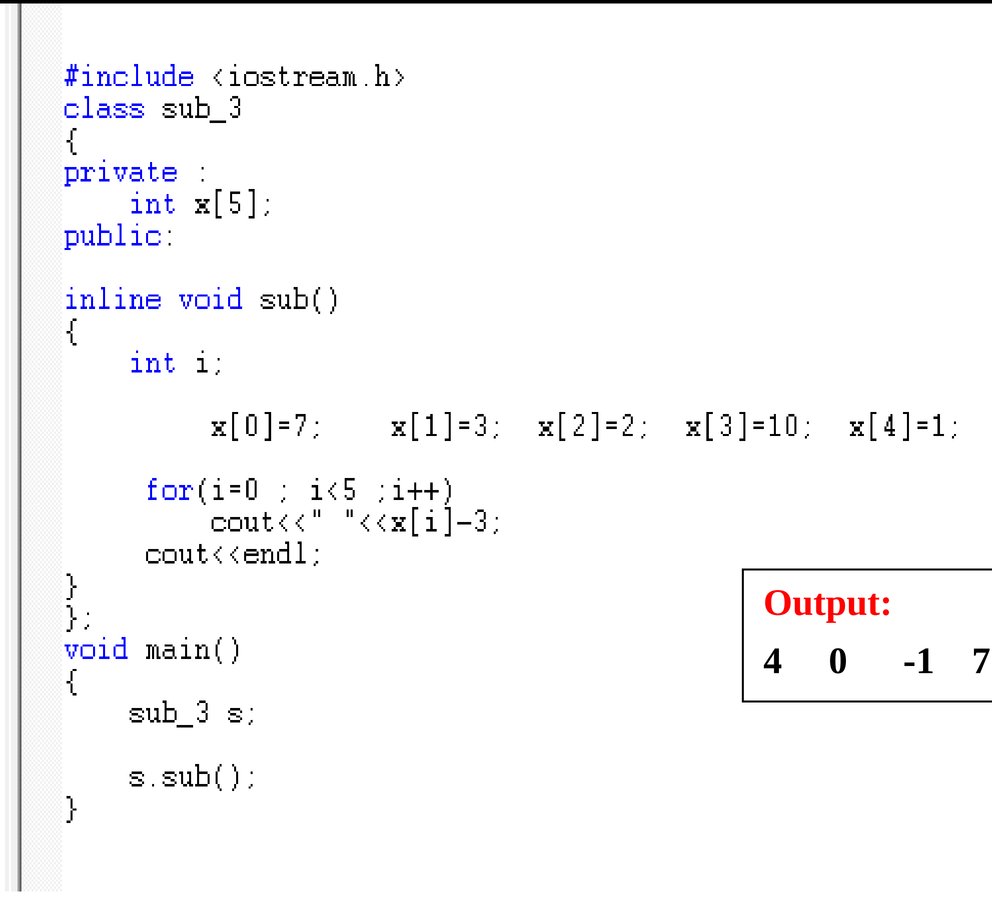
Output:
8

Arrays as Class Data Member

In C++, the definition of an array specifies a variable type and a name. But it includes another feature: a size. The size specifies how many data items the array will contain.

The items in an array are called *elements* .All elements in an array are of the same type; only the values vary. As specified in the definition, the array has exactly four elements. The **first** array element is numbered **0**. Thus, since there are four elements, the last one is number **3**.

Example 1: Write an OOP to subtract a value 3 from data member with 5 elements.



```
#include <iostream.h>
class sub_3
{
private :
    int x[5];
public:

inline void sub()
{
    int i;

         x[0]=7;    x[1]=3;  x[2]=2;  x[3]=10;  x[4]=1;

     for(i=0 ; i<5 ;i++)
         cout<<" "<<x[i]-3;
     cout<<endl;
}
};
void main()
{
    sub_3 s;

    s.sub();
}
```

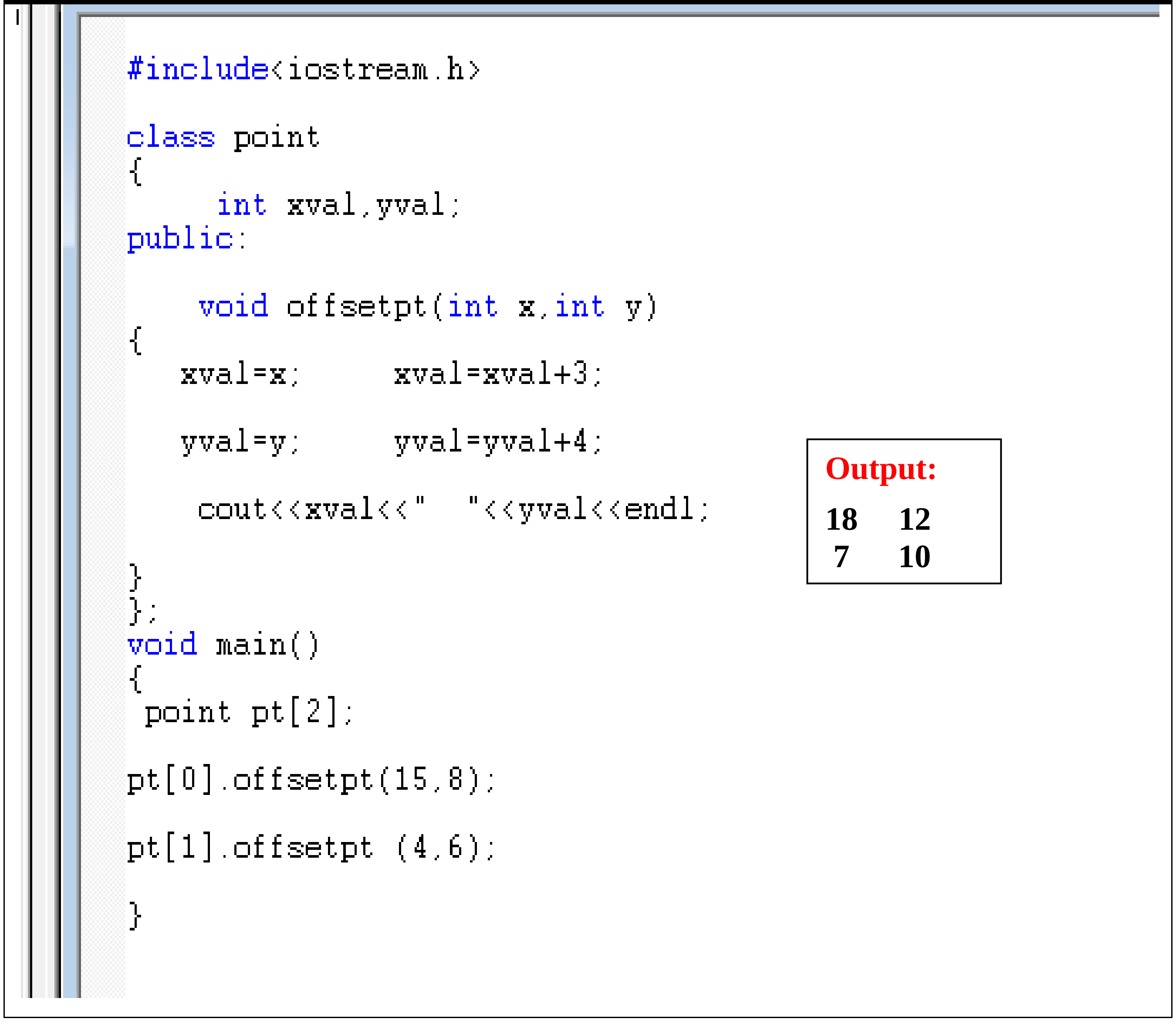
Output:

```
4    0    -1    7   -2
```

Object Arrays

- An array of a user-defined type is defined and used much in the same way as an array of a built-in type.

Example 1: Write an OOP to represent object array of class point for 2 points objects .

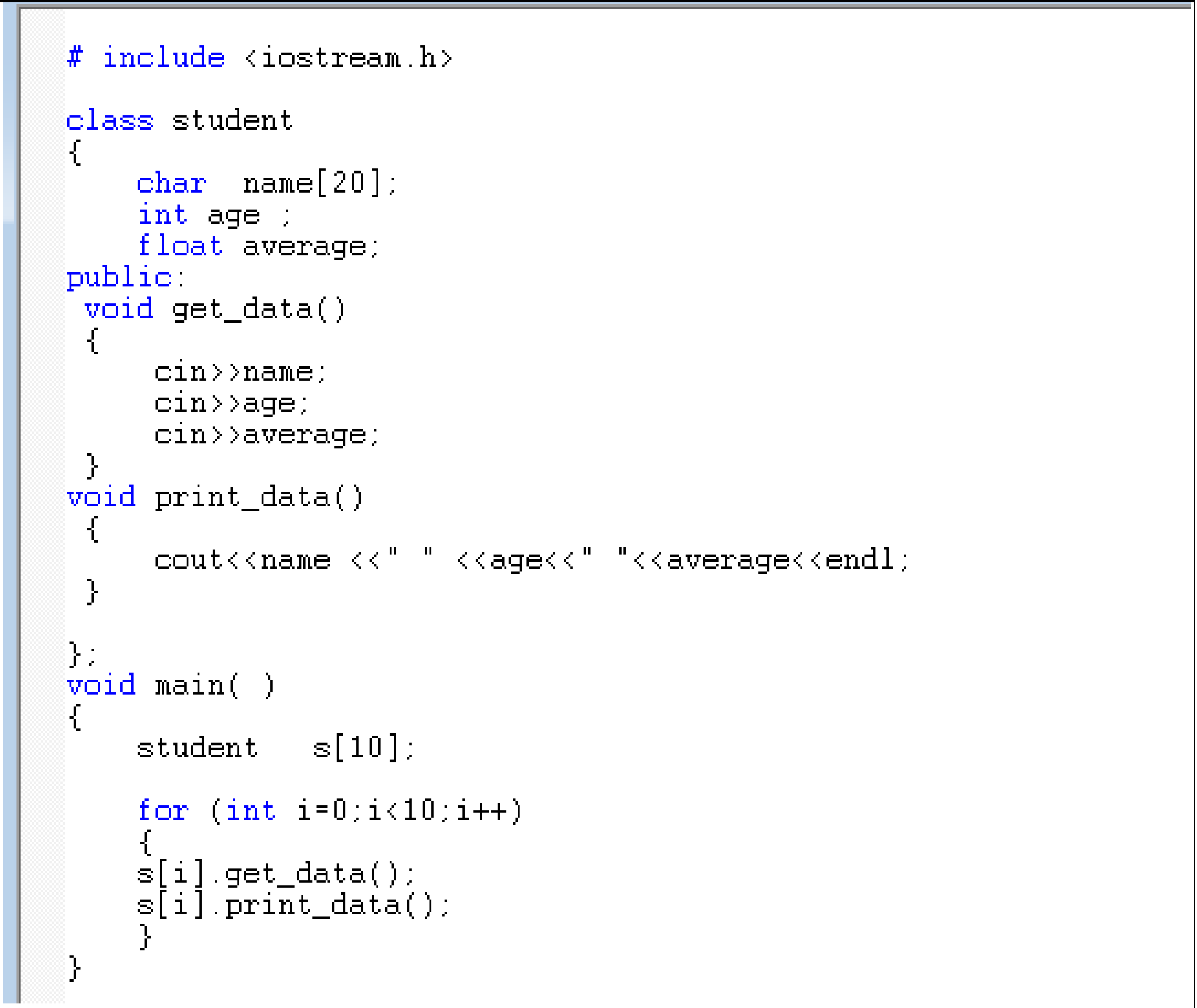


```
#include<iostream.h>

class point
{
     int xval,yval;
public:

    void offsetpt(int x,int y)
{
   xval=x;      xval=xval+3;

   yval=y;      yval=yval+4;

    cout<<xval<<"  "<<yval<<endl;

}
};
void main()
{
 point pt[2];

pt[0].offsetpt(15,8);

pt[1].offsetpt (4,6);

}
```

Output:

```
18    12
 7    10
```

Example 2: Write an OOP to read and write student information for 10 students objects.

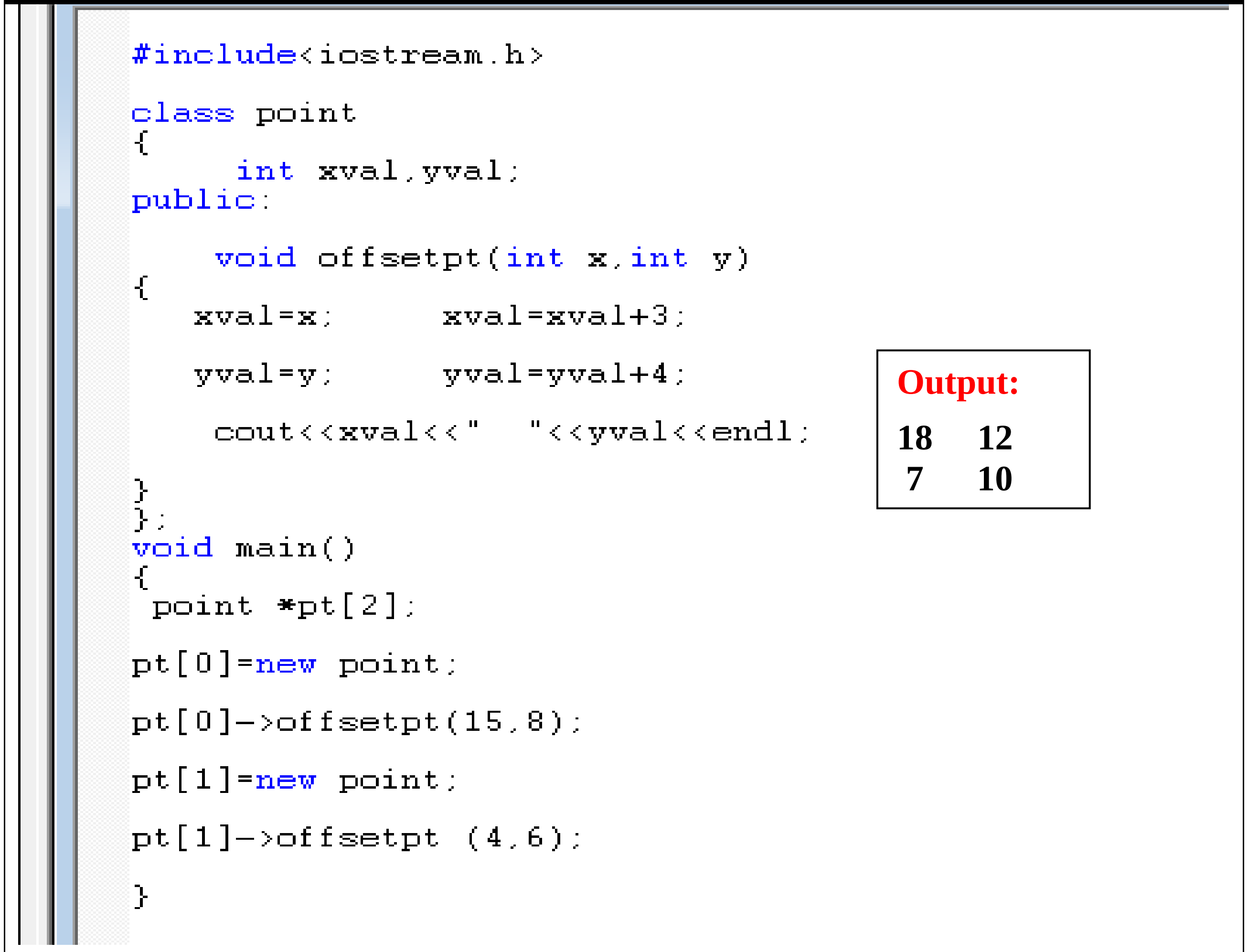


```
# include <iostream.h>

class student
{
    char  name[20];
    int age ;
    float average;
public:
 void get_data()
 {
     cin>>name;
     cin>>age;
     cin>>average;
 }
void print_data()
 {
     cout<<name <<" " <<age<<" "<<average<<endl;
 }

};
void main( )
{
    student   s[10];

    for (int i=0;i<10;i++)
    {
    s[i].get_data();
    s[i].print_data();
    }
}
```

An Array of Pointers to Objects

A common programming construction is an array of pointers to objects. This arrangement allows easy access to a group of objects, and is more flexible than placing the objects themselves in an array.

Example 1: Write an OOP to represent array of pointer to object for class point with 2 points objects .

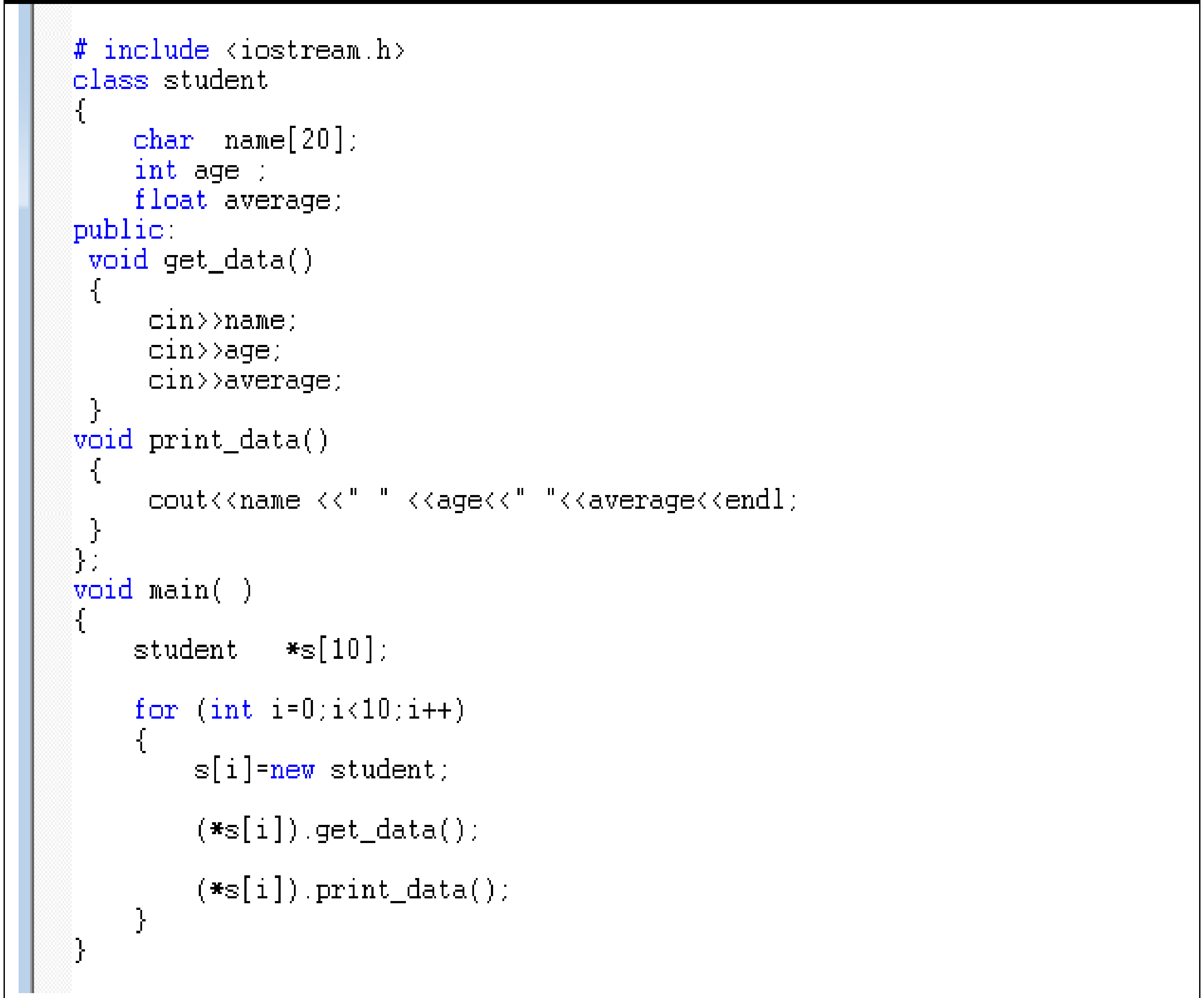


```
#include<iostream.h>

class point
{
     int xval,yval;
public:

    void offsetpt(int x,int y)
{
   xval=x;      xval=xval+3;

   yval=y;      yval=yval+4;

    cout<<xval<<"  "<<yval<<endl;

}
};
void main()
{
 point *pt[2];

pt[0]=new point;

pt[0]->offsetpt(15,8);

pt[1]=new point;

pt[1]->offsetpt (4,6);

}
```

Output:

```
18    12
 7    10
```

Example 2: Write an OOP to read and write student information for 10 students using array of pointer to objects.



```
# include <iostream.h>
class student
{
    char  name[20];
    int age ;
    float average;
public:
 void get_data()
 {
     cin>>name;
     cin>>age;
     cin>>average;
 }
void print_data()
 {
     cout<<name <<" " <<age<<" "<<average<<endl;
 }
};
void main( )
{
    student   *s[10];

    for (int i=0;i<10;i++)
    {
        s[i]=new student;

        (*s[i]).get_data();

        (*s[i]).print_data();
    }
}
```

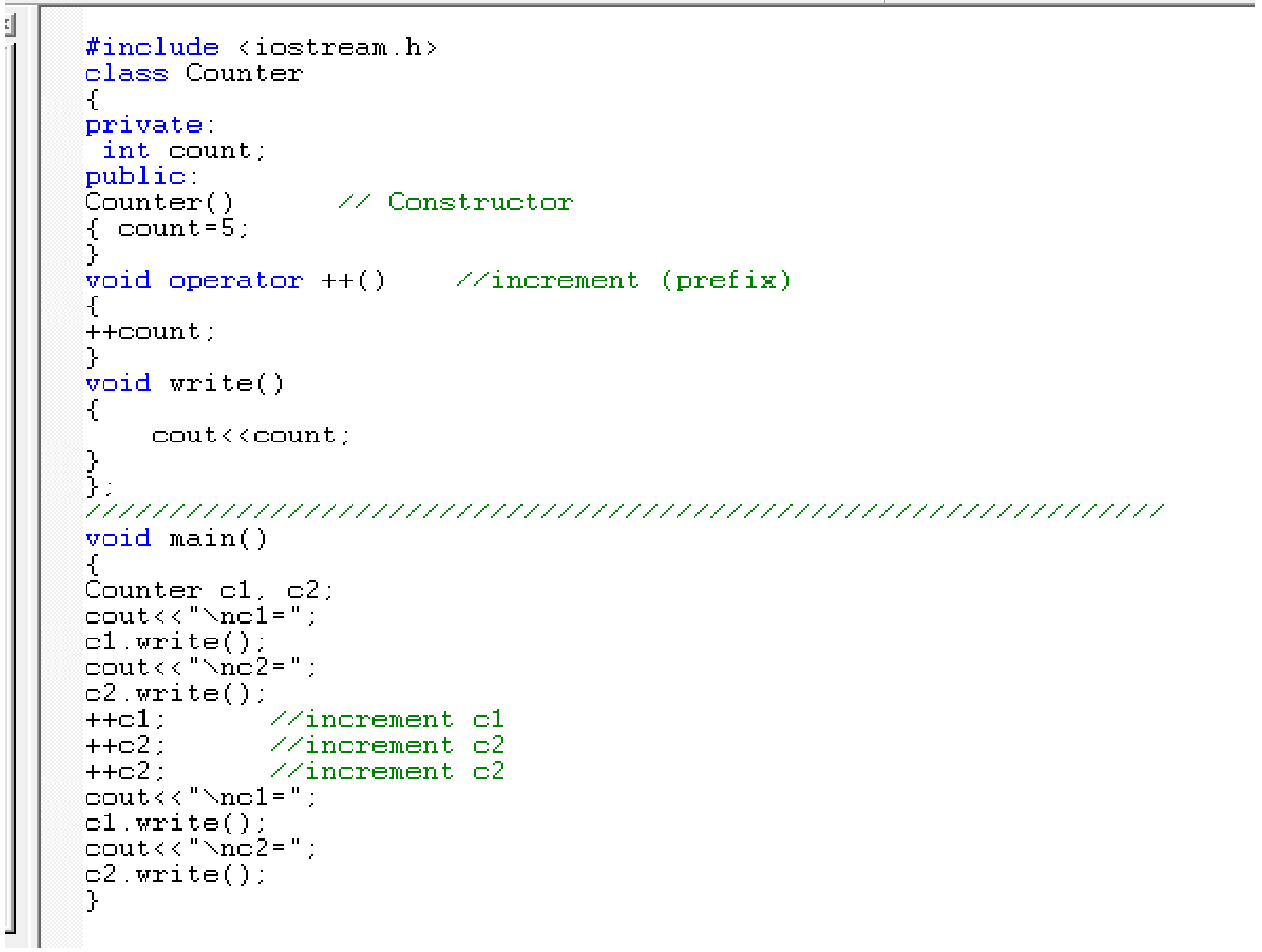
Operator overloading

Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure program into intuitively obvious ones. Operators overloading was applied for **unary** and **binary**

1- Overloading Unary Operators

Unary operators act on only one operand. Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -33.

Example 1:-Write an oo program to increment the counter variable using ++ operator.



```
#include <iostream.h>
class Counter
{
private:
 int count;
public:
Counter()       // Constructor
{ count=5;
}
void operator ++()    //increment (prefix)
{
++count;
}
void write()
{
    cout<<count;
}
};
/////////////////////////////////////////////////////////////////
void main()
{
Counter c1, c2;
cout<<"\nc1=";
c1.write();
cout<<"\nc2=";
c2.write();
++c1;      //increment c1
++c2;      //increment c2
++c2;      //increment c2
cout<<"\nc1=";
c1.write();
cout<<"\nc2=";
c2.write();
}
```

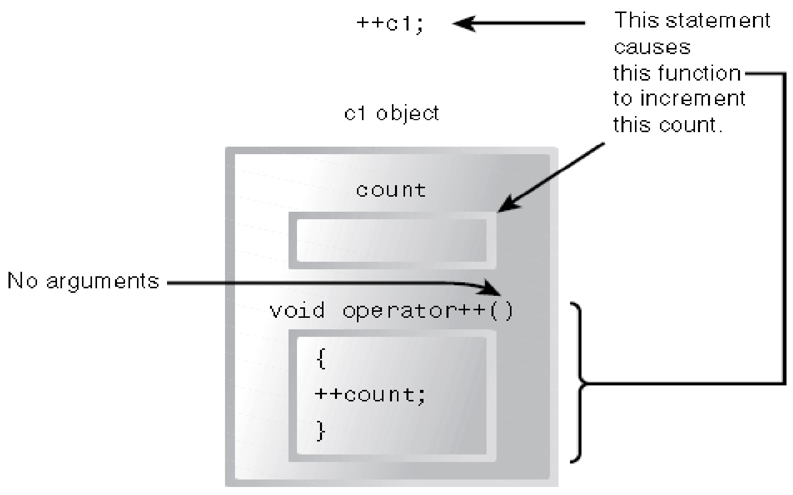
Here’s the program’s output:

```
c1=5 ←——— counts are initially 5
c2=5 ←———
c1=6 ←——— incremented once
c2=7 ←——— incremented twice
```

The ++ operator is applied once to c1 and twice to c2. We use prefix notation in this example.

<u>Operator Arguments</u>

In main () the ++ operator is applied to a specific object, as in the expression ++c1. Yet operator++() takes no arguments. What does this operator increment? It increment the count data in the object of which it is a member. Since member functions can always access the particular object for which they’ve been invoked, this operator requires no arguments. This is shown in Figure 1.



```
void operator++()
{
++count;
}
```

++c1;
This statement causes this function to increment this count.
c1 object
count
No arguments

Figure 1: ***Overloaded unary operator: no arguments.***

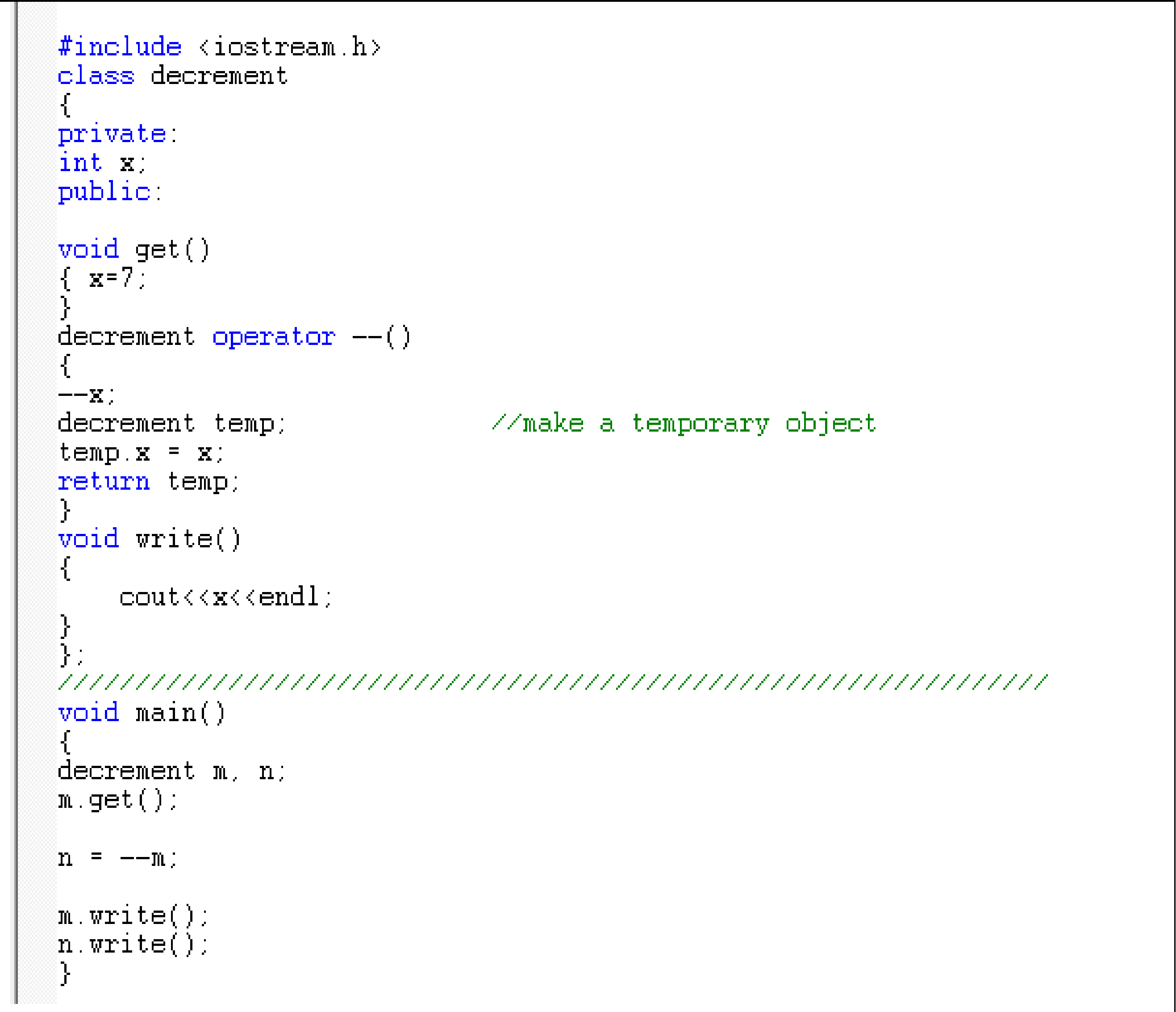
Operator Return Values

You will discover it if you use a statement like this in main (): **c1 = --c2;**

There are two types to return object in the operator overloading:-

a-Using Temporary Object.

Example 2:-Write an OOP to decrement a variable in class decrement using (--) operator and return the value to other object using temporary object .

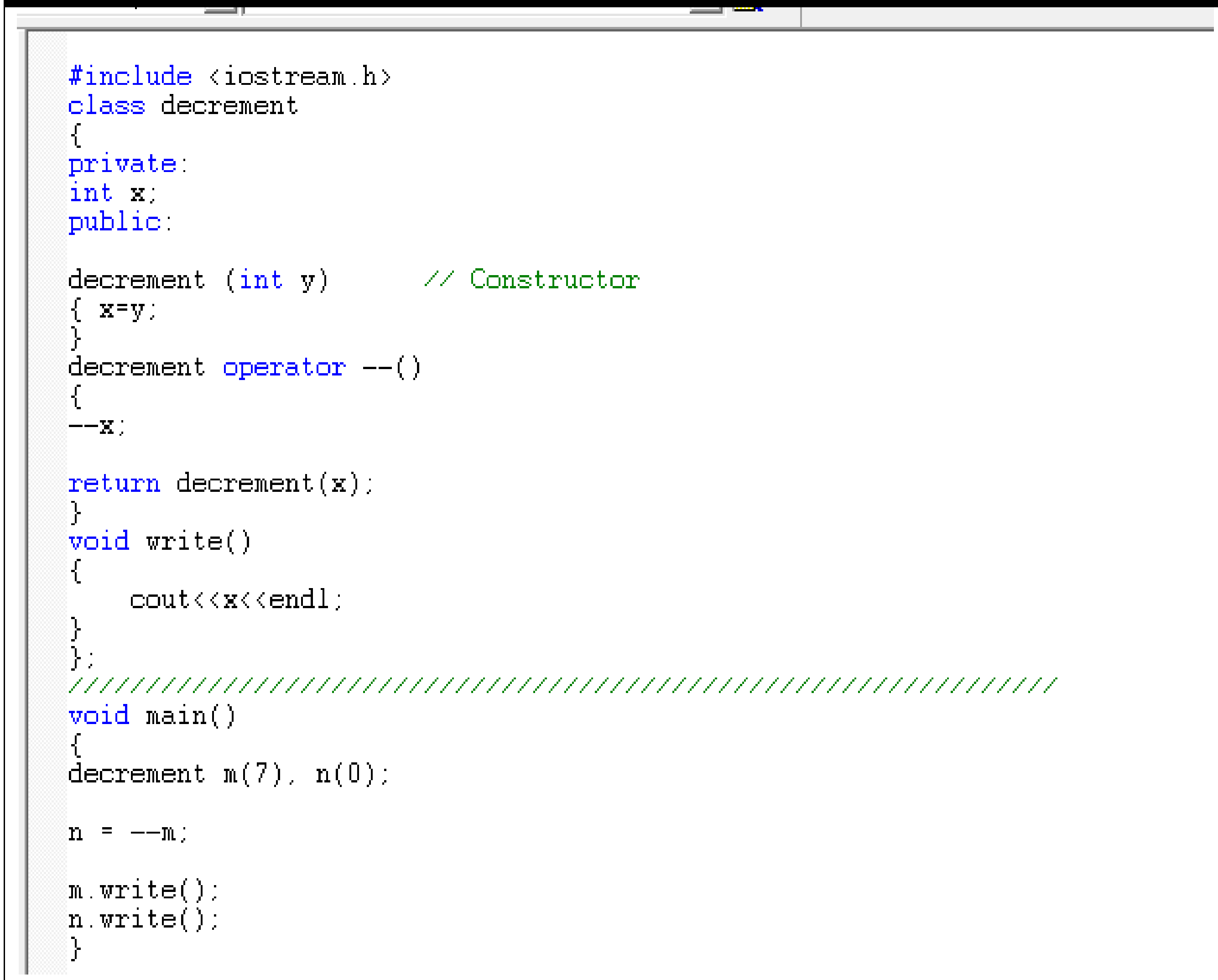


```
#include <iostream.h>
class decrement
{
private:
int x;
public:

void get()
{ x=7;
}
decrement operator --()
{
--x;
decrement temp;                //make a temporary object
temp.x = x;
return temp;
}
void write()
{
    cout<<x<<endl;
}
};
////////////////////////////////////////////////////////////////
void main()
{
decrement m, n;
m.get();

n = --m;

m.write();
n.write();
}
```

b- Nameless Temporary Objects

In Example 2 we created a temporary object of type decrement, named temp, whose sole purpose was to provide a return value for the -- operator.

Example 3:- Write an OOP to decrement a variable in class decrement using (--) operator and return the value to other object using nameless temporary object.



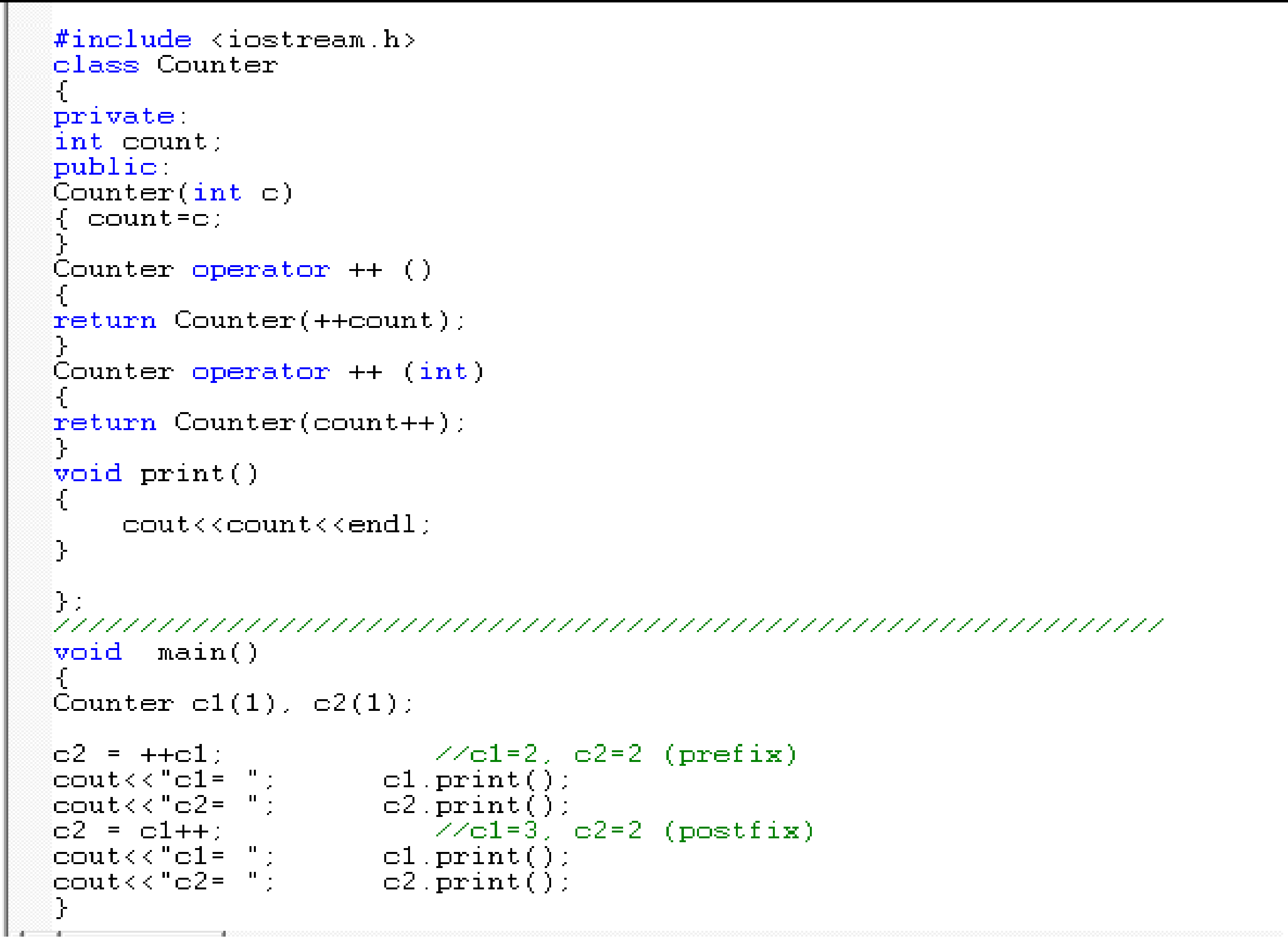
```
#include <iostream.h>
class decrement
{
private:
int x;
public:

decrement (int y)       // Constructor
{ x=y;
}
decrement operator --()
{
--x;

return decrement(x);
}
void write()
{
    cout<<x<<endl;
}
};
////////////////////////////////////////////////////////////////////////
void main()
{
decrement m(7), n(0);

n = --m;

m.write();
n.write();
}
```

In this program a single statement **return decrement(x);** this statement creates an object of type decrement **decrement (int y)** //constructor, one arg

Postfix Notation

We define two overloaded ++ operators, as shown in the Example4:

Example 4:-Write an oo program to increment the counter variable with ++ operator using both prefix and postfix.



```
#include <iostream.h>
class Counter
{
private:
int count;
public:
Counter(int c)
{ count=c;
}
Counter operator ++ ()
{
return Counter(++count);
}
Counter operator ++ (int)
{
return Counter(count++);
}
void print()
{
    cout<<count<<endl;
}

};
////////////////////////////////////////////////////////////////
void  main()
{
Counter c1(1), c2(1);

c2 = ++c1;              //c1=2, c2=2 (prefix)
cout<<"c1= ";      c1.print();
cout<<"c2= ";      c2.print();
c2 = c1++;              //c1=3, c2=2 (postfix)
cout<<"c1= ";      c1.print();
cout<<"c2= ";      c2.print();
}
```

Now there are two different decelerator for overloading the ++ operator. The one we've seen before, for prefix notation, is **Counter operator ++ ().** The new one, for postfix notation, is **Counter operator ++ (int)**. The only difference is the (int) in the parentheses. This (int) isn't really an argument, and it doesn't mean integer. It's simply a signal to the compiler to create the postfix version of the operator.

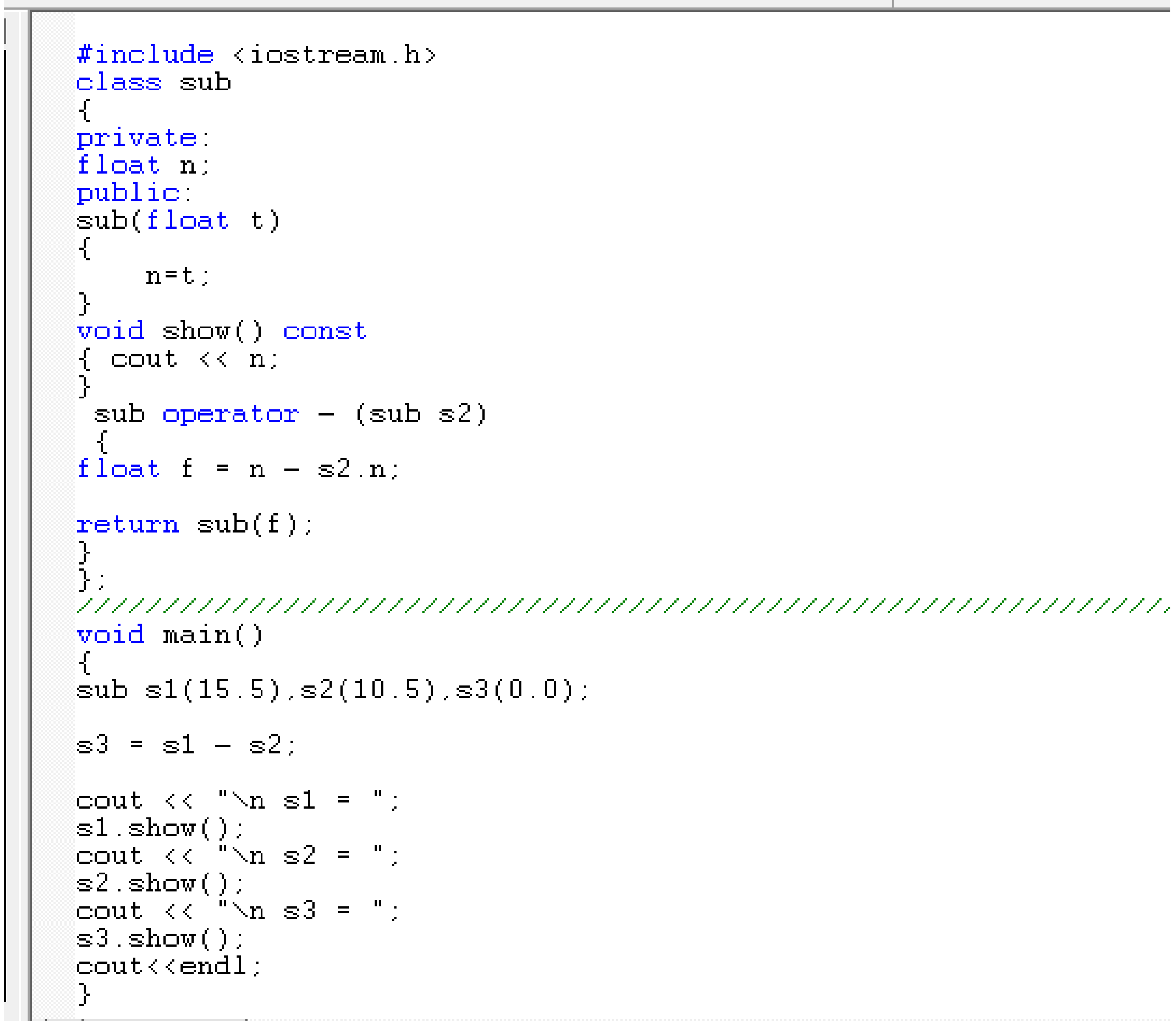
2- <u>Overloading Binary Operators</u>

Binary operators can be overloaded just as easily as unary operators. There are two types of overloading binary operators:

a) <u>Arithmetic Operators</u>

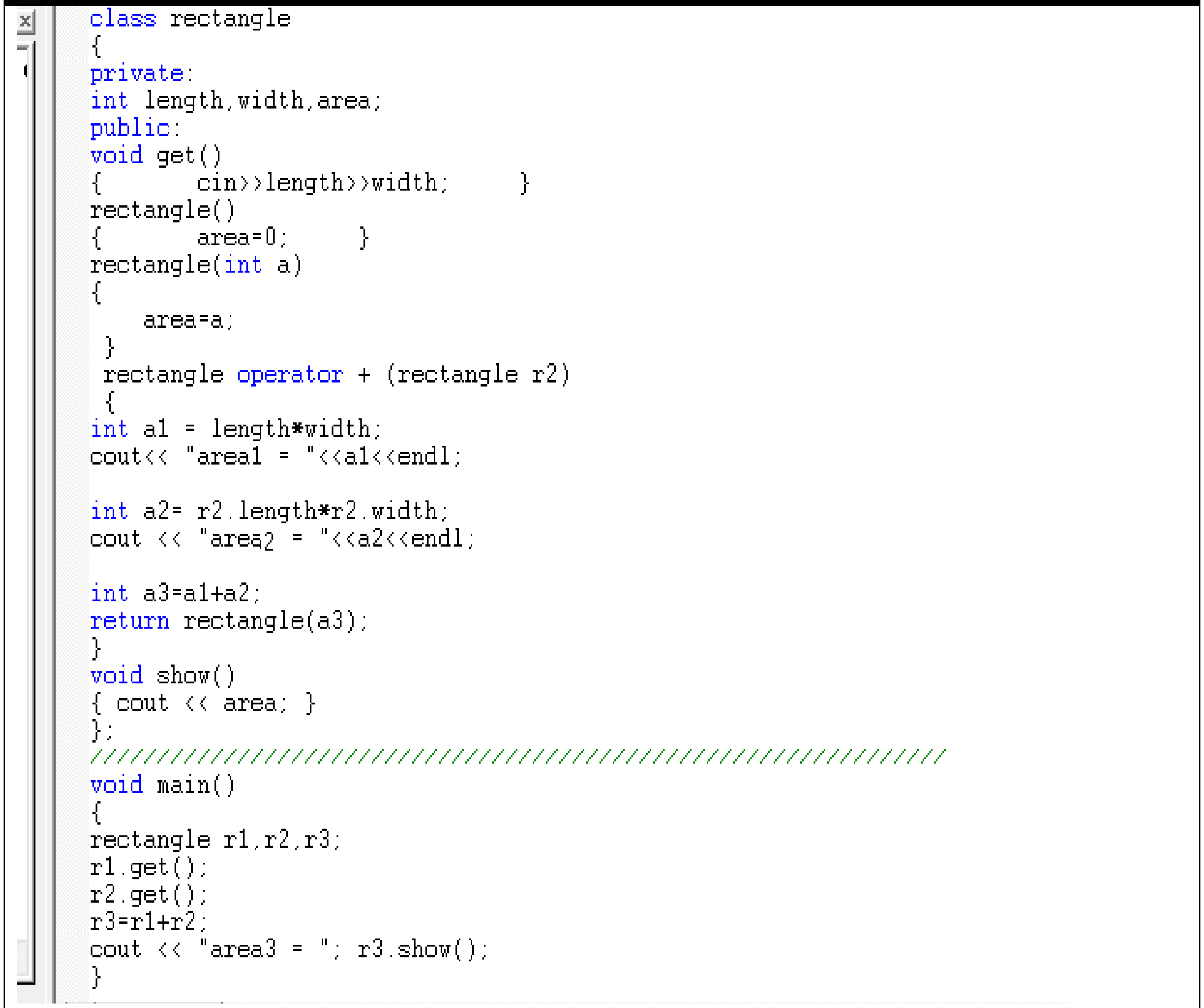
Arithmetic operators consists of (+,-,*,/) as illustrated in examples bellow

Example 5:-Write an OOP to subtract two numbers objects using - operator.



```
#include <iostream.h>
class sub
{
private:
float n;
public:
sub(float t)
{
    n=t;
}
void show() const
{ cout << n;
}
 sub operator - (sub s2)
 {
float f = n - s2.n;

return sub(f);
}
};
////////////////////////////////////////////////////////////////
void main()
{
sub s1(15.5),s2(10.5),s3(0.0);

s3 = s1 - s2;

cout << "\n s1 = ";
s1.show();
cout << "\n s2 = ";
s2.show();
cout << "\n s3 = ";
s3.show();
cout<<endl;
}
```

Example 6:-Write an OOP to add two area of rectangle using+ operator.



```
class rectangle
{
private:
int length,width,area;
public:
void get()
{       cin>>length>>width;     }
rectangle()
{       area=0;     }
rectangle(int a)
{
    area=a;
 }
 rectangle operator + (rectangle r2)
 {
int a1 = length*width;
cout<< "area1 = "<<a1<<endl;

int a2= r2.length*r2.width;
cout << "area2 = "<<a2<<endl;

int a3=a1+a2;
return rectangle(a3);
}
void show()
{ cout << area; }
};
//////////////////////////////////////////////////////////////////
void main()
{
rectangle r1,r2,r3;
r1.get();
r2.get();
r3=r1+r2;
cout << "area3 = "; r3.show();
}
```

Input for r1 : **3 4**

Input for r2 : **9 2**

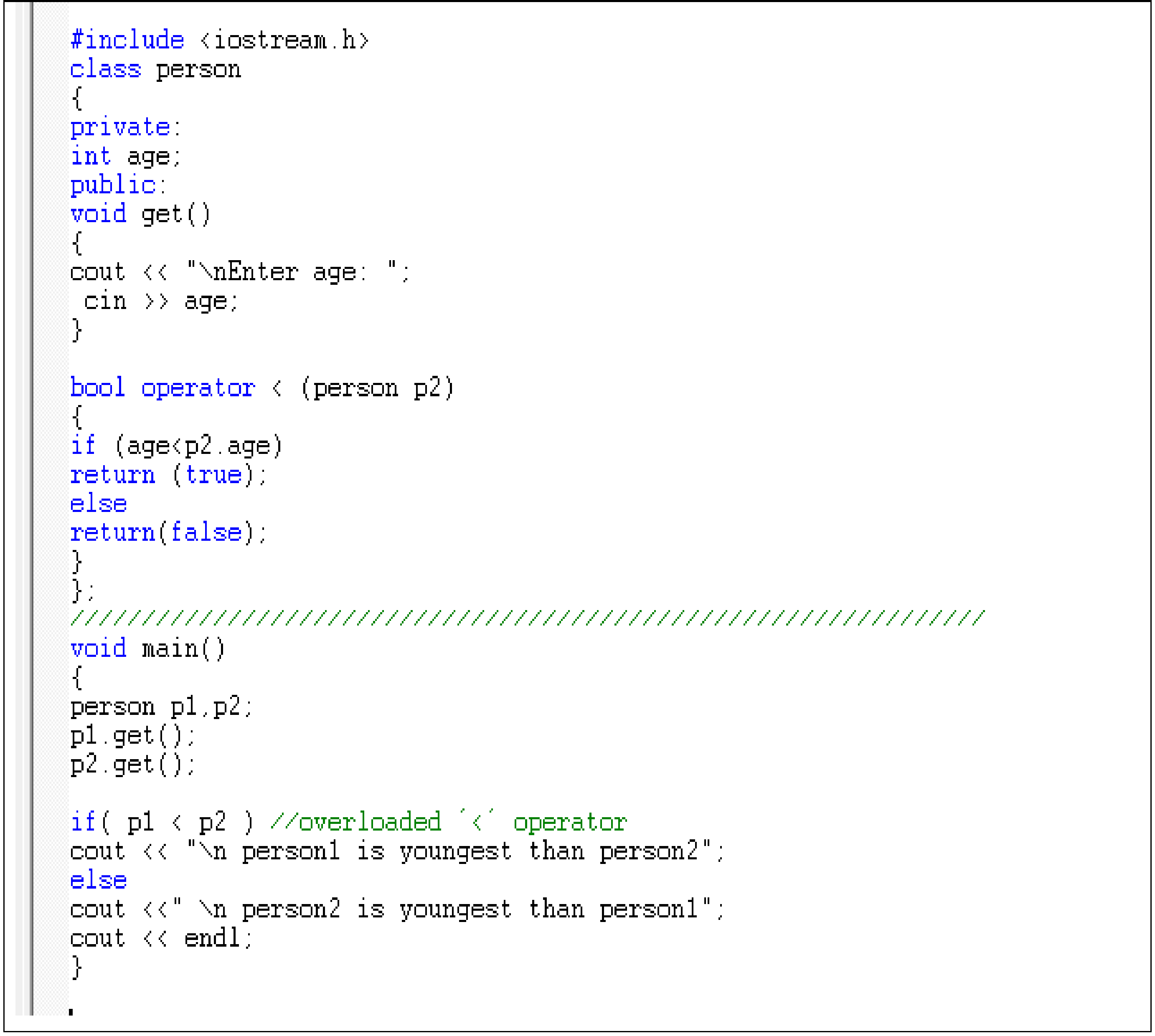
Outputs:

```
area1 = 12
area2=18
area3=30
```

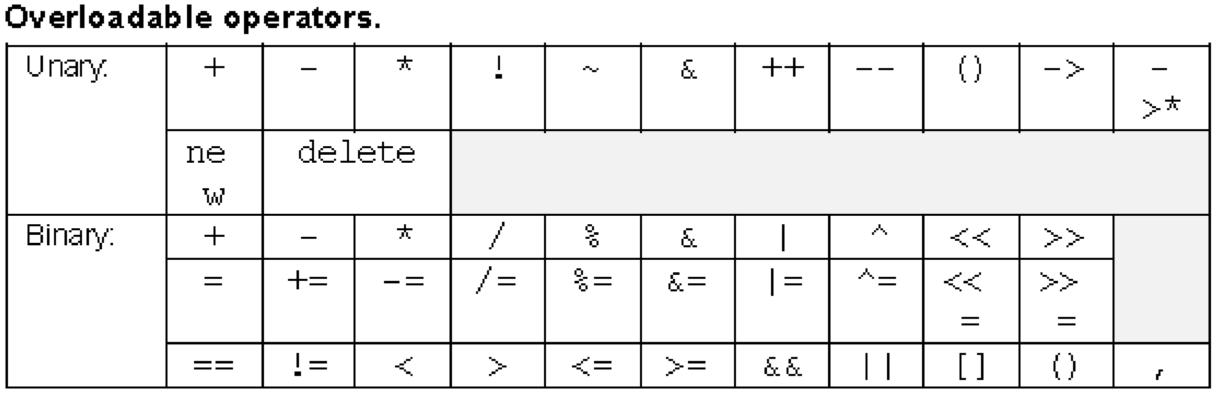
b) <u>Comparison Operators</u>

Comparing operators consists of (>, <, >= ,<= ,==, !=).Examples bellow show some of these operator.

Example 7:-Write an OOP to compare two ages using operator < in class person.



```
#include <iostream.h>
class person
{
private:
int age;
public:
void get()
{
cout << "\nEnter age: ";
 cin >> age;
}

bool operator < (person p2)
{
if (age<p2.age)
return (true);
else
return(false);
}
};
///////////////////////////////////////////////////////////////////////////
void main()
{
person p1,p2;
p1.get();
p2.get();

if( p1 < p2 ) //overloaded '<' operator
cout << "\n person1 is youngest than person2";
else
cout <<" \n person2 is youngest than person1";
cout << endl;
}
```

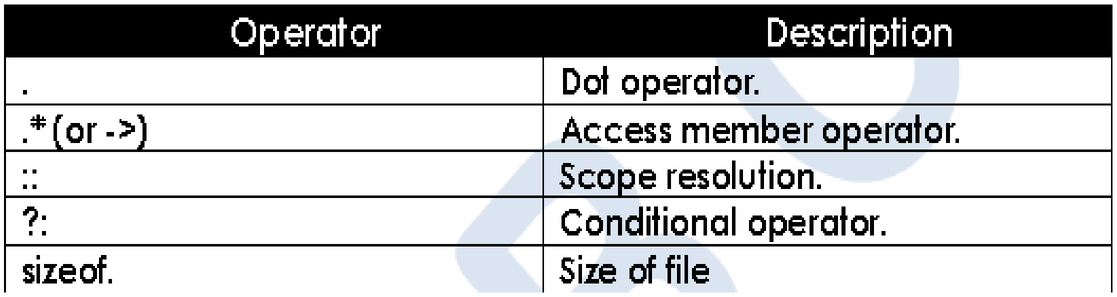


Overloadable operators.

Unary:	+	-	*	!	~	&	++	--	()	->	->*
	new	delete									
Binary:	+	-	*	/	%	&	|	^	<<	>>	
	=	+=	-=	/=	%=	&=	|=	^=	<<=	>>=	
	==	!=	<	>	<=	>=	&&	||	[]	()	,

Figure 3: Overloadable Operators

Operators that can't be overloaded are listed bellow:



Operator	Description
.	Dot operator.
.* (or ->)	Access member operator.
::	Scope resolution.
?:	Conditional operator.
sizeof.	Size of file

Figure 4: Operators can't be overloaded

Inheritance

Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves. Inheritance is the process of creating new classes, called ***derived classes***, from existing or ***base classes***. The derived class **inherits** all the capabilities of the base class but can add embellishments and refinements of its own. The base class is unchanged by this process. The inheritance relationship is shown in Figure 1.

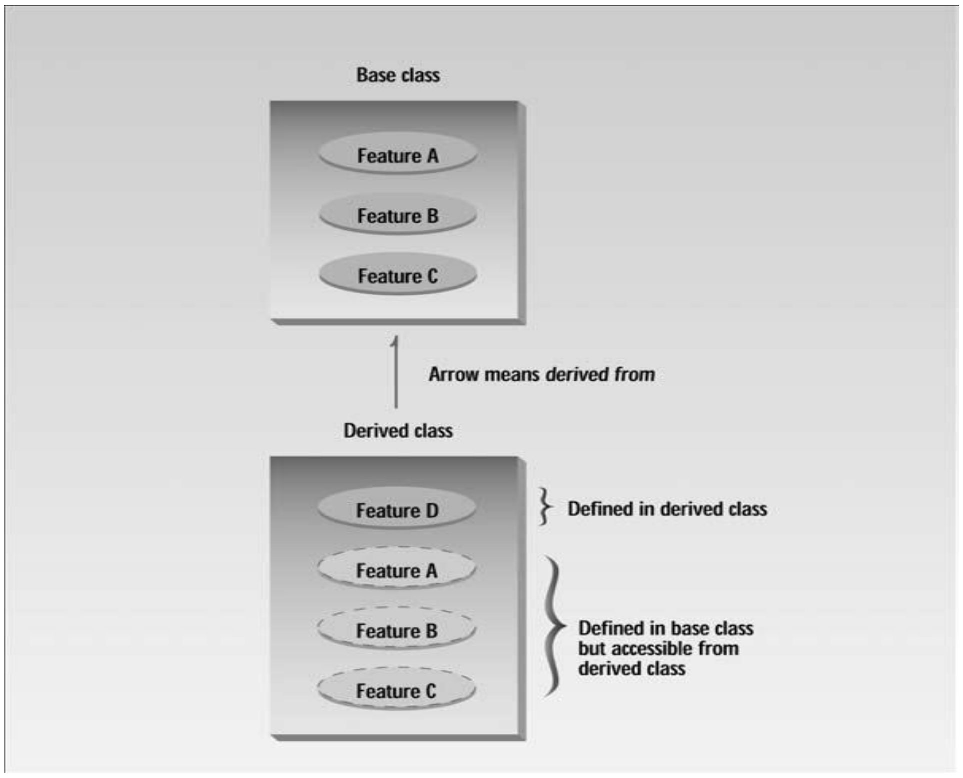


Figure 1: *Inheritance.*

An important result of reusability is the ease of distributing class libraries. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

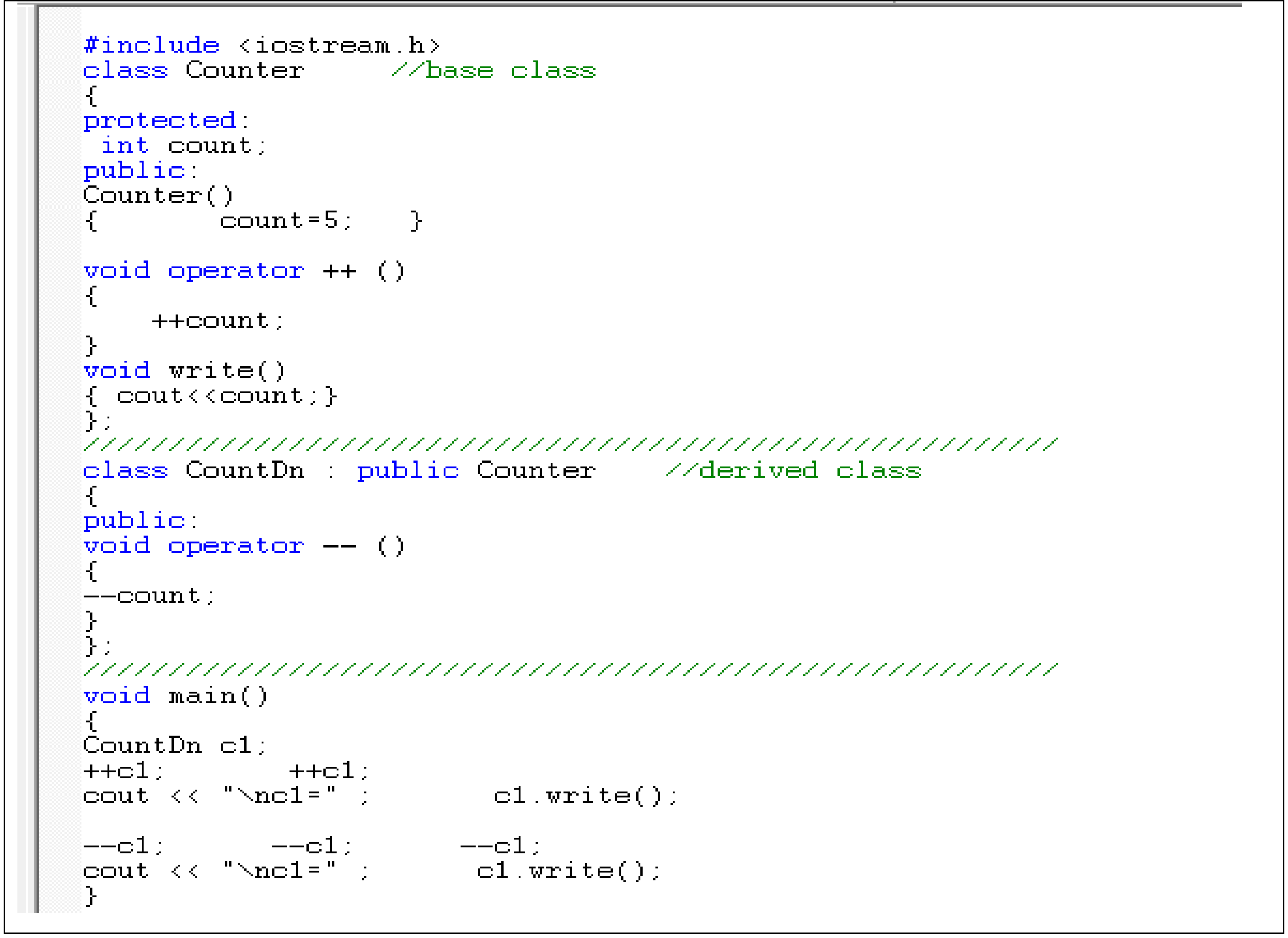
Derived Class and Base Class

There are two main **reasons** that we might not want to modifying the base class.

1) First, the base class works very well and has undergone many hours of testing and debugging.

2) Second reason for not modifying the base class: We might not have access to its source code, especially if it was distributed as part of a class library.

Example 1:-Write an OOP to decrement the count variable in class counter using – operator and inheritance with ++ operator. The program includes a class called Counter which contains a private count and a class CountDn was drived from Counter class.



```
#include <iostream.h>
class Counter      //base class
{
protected:
 int count;
public:
Counter()
{        count=5;    }

void operator ++ ()
{
    ++count;
}
void write()
{ cout<<count;}
};
/////////////////////////////////////////////////////////
class CountDn : public Counter      //derived class
{
public:
void operator -- ()
{
--count;
}
};
/////////////////////////////////////////////////////////
void main()
{
CountDn c1;
++c1;       ++c1;
cout << "\nc1=" ;       c1.write();

--c1;       --c1;       --c1;
cout << "\nc1=" ;      c1.write();
}
```

Output from example 1:-

c1=7 ← after ++c1, ++c1

c1=4 ← after --c1, --c1.—c1

Accessing Base Class Members

An important topic in inheritance knows when a member function in the base class can be used by objects of the derived class. This is called **accessibility**.

The protected Access Specifier

We have increased the functionality of a class without modifying it. Let's first review what we know about the access specifies private and public. A member function of a class can always access class members, whether they are public or private. But an object declared externally can only invoke public members of the class. Private members are, well, *private*. This is shown in Figure 2.

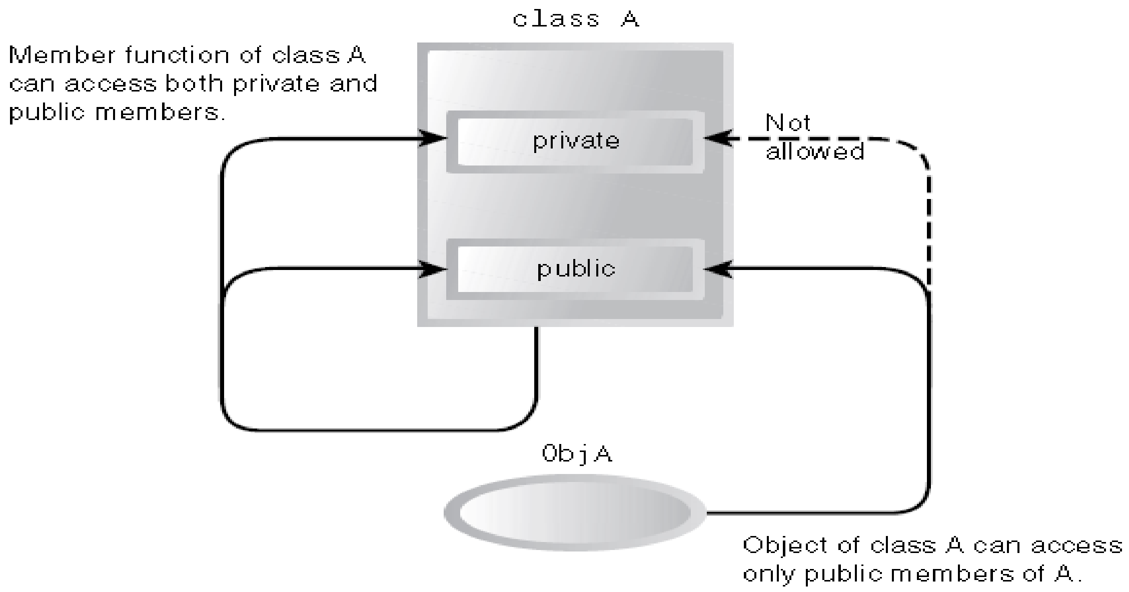

class A
Member function of class A can access both private and public members.
private
Not allowed
public
ObjA
Object of class A can access only public members of A.

Figure 2: Access *specifiers without inheritance*

A protected member, on the other hand, can be accessed by member functions in its own class and in any class derived from its own class. It can't be accessed from functions outside these classes, such as main(). The situation is shown in Figure 3.

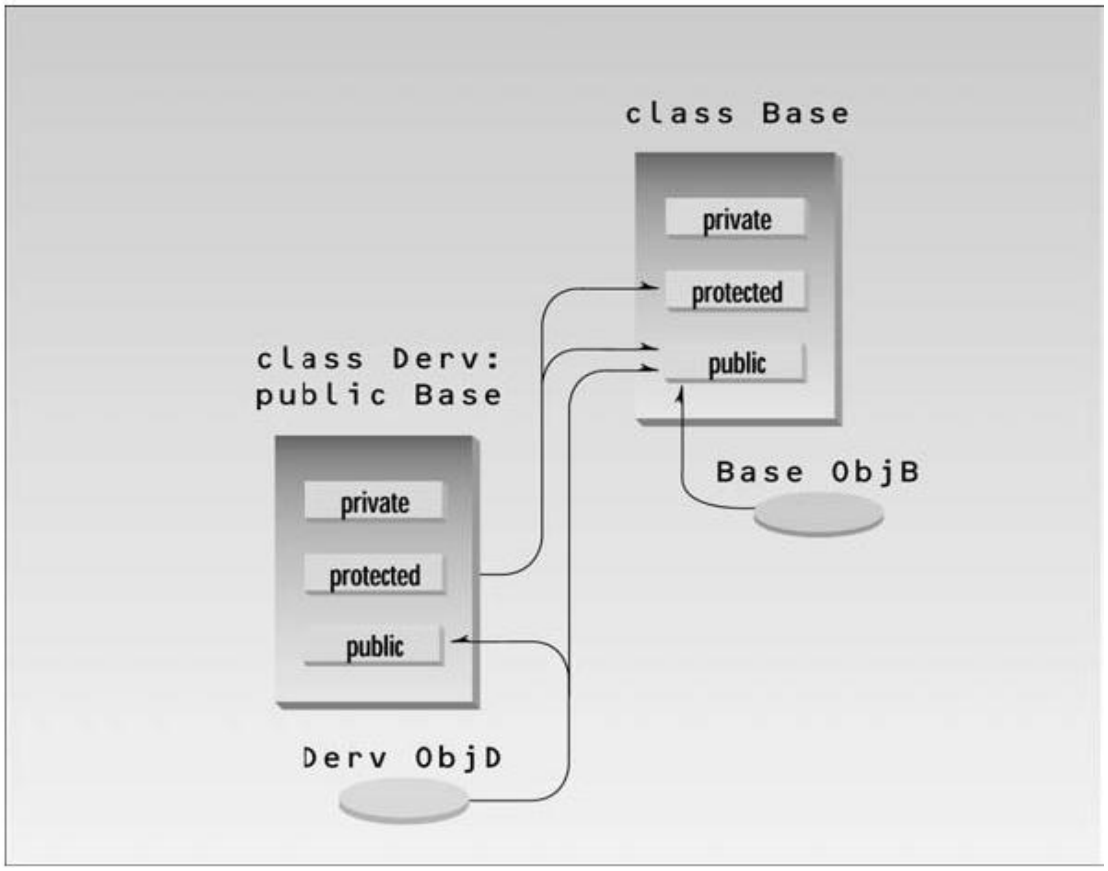
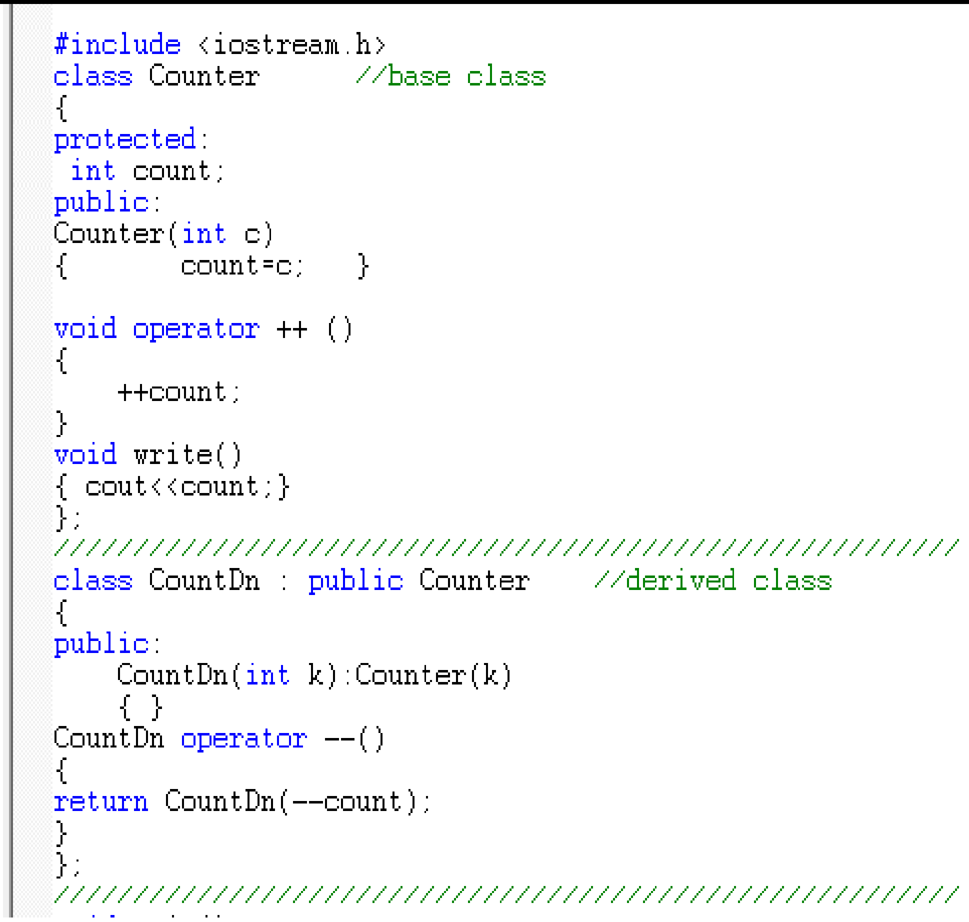


Figure 3: ***Access specifiers with inheritance.***

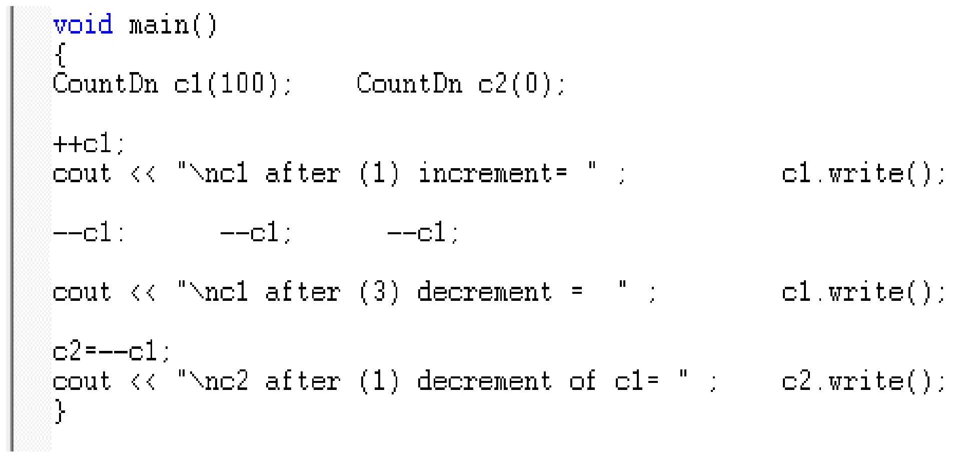
Table 1: Inheritance and Accessibility

Access Specifier	*Accessible from Own Class*	*Accessible from Derived Class*	*Accessible from Objects Outside Class*
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

Example 2:- Write an OOP to decrement the count variable in class counter using – operator and inheritance with ++ operator and the main contains the statement c2=--c1



```
#include <iostream.h>
class Counter      //base class
{
protected:
 int count;
public:
Counter(int c)
{        count=c;    }

void operator ++ ()
{
    ++count;
}
void write()
{ cout<<count;}
};
//////////////////////////////////////////////////////////
class CountDn : public Counter    //derived class
{
public:
    CountDn(int k):Counter(k)
    { }
CountDn operator --()
{
return CountDn(--count);
}
};
//////////////////////////////////////////////////////////
```



```
void main()
{
CountDn c1(100);    CountDn c2(0);

++c1;
cout << "\nc1 after (1) increment= " ;          c1.write();

--c1:       --c1;       --c1;

cout << "\nc1 after (3) decrement =  " ;        c1.write();

c2=--c1;
cout << "\nc2 after (1) decrement of c1= " ;    c2.write();
}
```

Output from example 2:-

```
c1 after (1) increment =101
c1 after (3) decrement =98
c2 after (1) decrement of c1 =97
```

Dangers of protected

You should know that there’s a disadvantage to making class members protected. Say you’ve written a class library, which you’re distributing to the public. Any programmer who buys this library can access protected members of your classes simply by deriving other classes from them. This makes protected members considerably less secure than private members. To avoid corrupted data, it’s often safer to force derived classes to access data in the base class using only public functions.

Overriding Member Functions

You can use member functions in a derived class that override—that is, have the same name as those in the base class. You might want to do this so that calls in your program work the same way for objects of both base and derived classes.

Example 3:-Write an OOP to read and write the information of employee. Create a class called employee which contains employee's name and number as private data items. Derived from class employee a class called manger which contains a salary of type float as private item.

```
#include <iostream.h>
class employee
{     private:
          char name[100];          long number;
public:
 void getdata()
{       cin >> name;         cin >> number;     }

void putdata()
{        cout << name;         cout << number;   }
};

class  manager : public  employee
{  private:
         float    salary;
public:
void  getdata()
{
       employee :: getdata();
            cin >> salary;
}
void putdata()
{
      employee :: putdata();
          cout << salary;
}
};
void  main()
{         manager  m1;
            m1.getdata();
            m1.putdata();
}
```

Class Hierarchies

Inheritance has been used to add functionality to an existing class. Now let's look at an example where inheritance is used for a different purpose: as part of the original design of a program.

The database stores a name and an employee identification number for all employees. However, for managers, it also stores their titles. For scientists, it stores the number of scholarly articles they have published. Laborers need no additional data beyond their names and numbers.

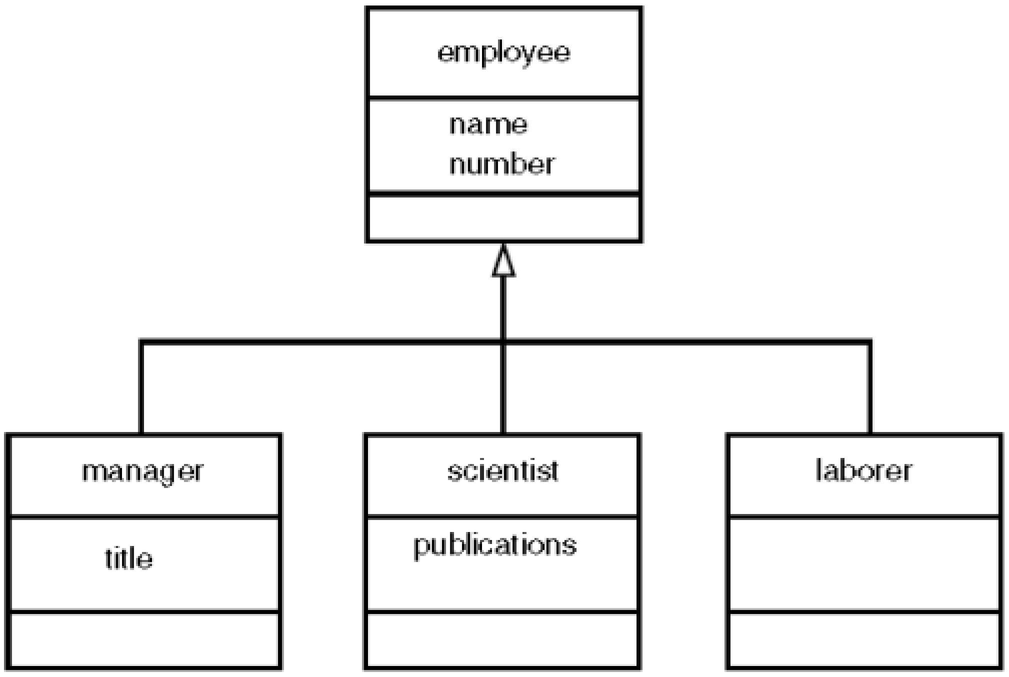
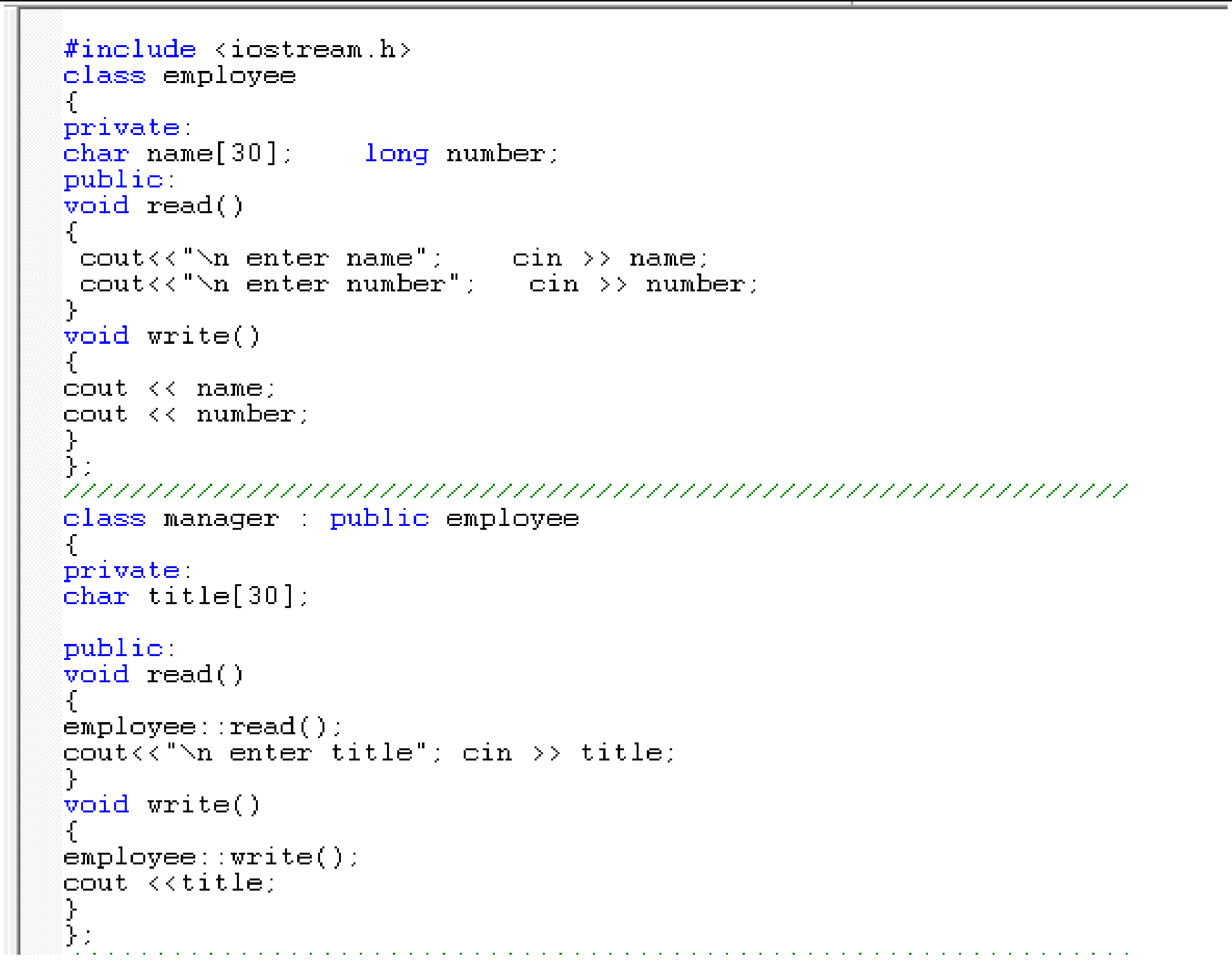
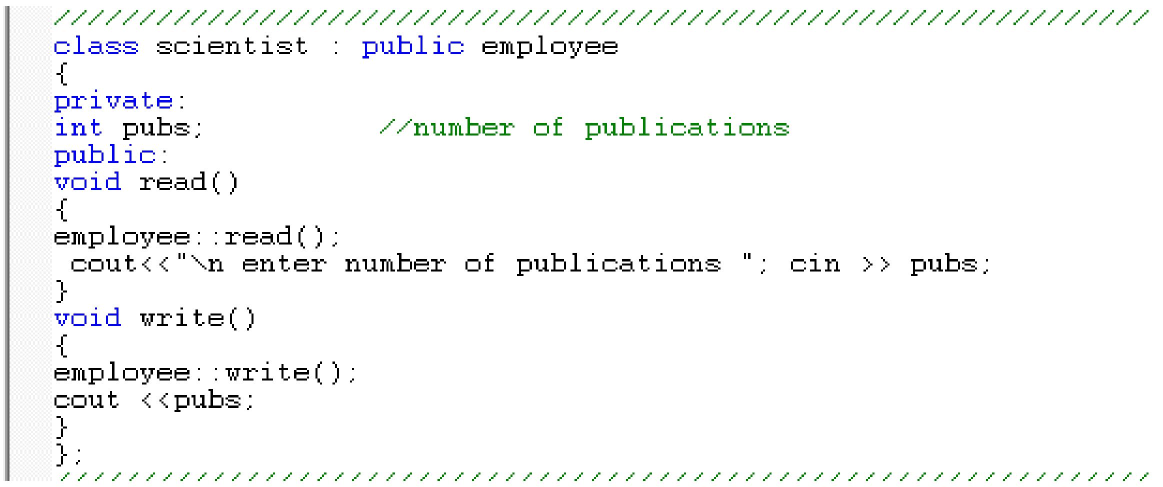


Figure 1: ***class diagram for EMPLOY***

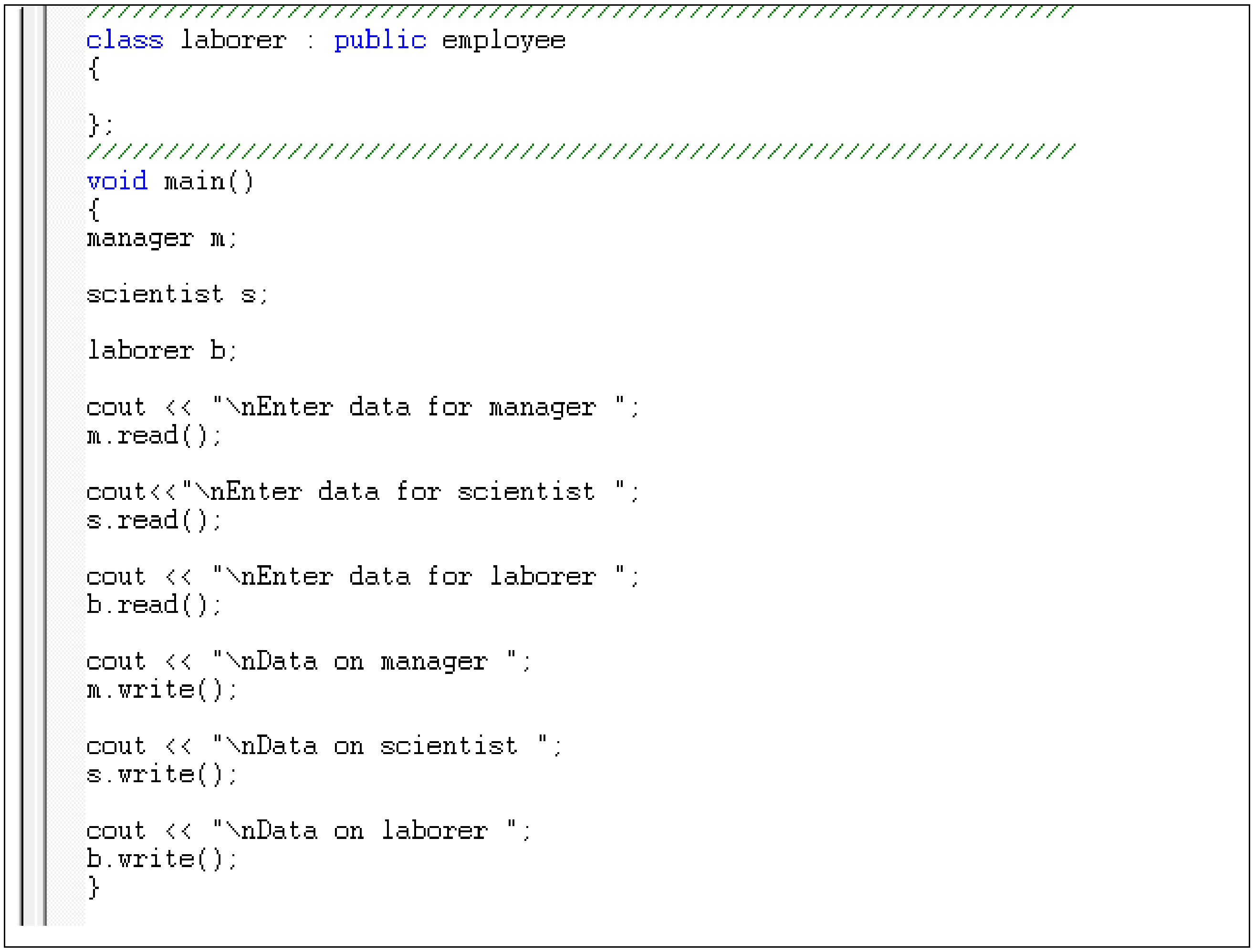
Example 1:-Write an oo program to model employ in figure1 using inheritance.



```
#include <iostream.h>
class employee
{
private:
char name[30];    long number;
public:
void read()
{
 cout<<"\n enter name";    cin >> name;
 cout<<"\n enter number";    cin >> number;
}
void write()
{
cout << name;
cout << number;
}
};
/////////////////////////////////////////////////////////////////
class manager : public employee
{
private:
char title[30];

public:
void read()
{
employee::read();
cout<<"\n enter title"; cin >> title;
}
void write()
{
employee::write();
cout <<title;
}
};
```



```
/////////////////////////////////////////////////////////////////
class scientist : public employee
{
private:
int pubs;            //number of publications
public:
void read()
{
employee::read();
 cout<<"\n enter number of publications "; cin >> pubs;
}
void write()
{
employee::write();
cout <<pubs;
}
};
/////////////////////////////////////////////////////////////////
```



```
//////////////////////////////////////////////////////////////////////
class laborer : public employee
{

};
//////////////////////////////////////////////////////////////////////
void main()
{
manager m;

scientist s;

laborer b;

cout << "\nEnter data for manager ";
m.read();

cout<<"\nEnter data for scientist ";
s.read();

cout << "\nEnter data for laborer ";
b.read();

cout << "\nData on manager ";
m.write();

cout << "\nData on scientist ";
s.write();

cout << "\nData on laborer ";
b.write();
}
```

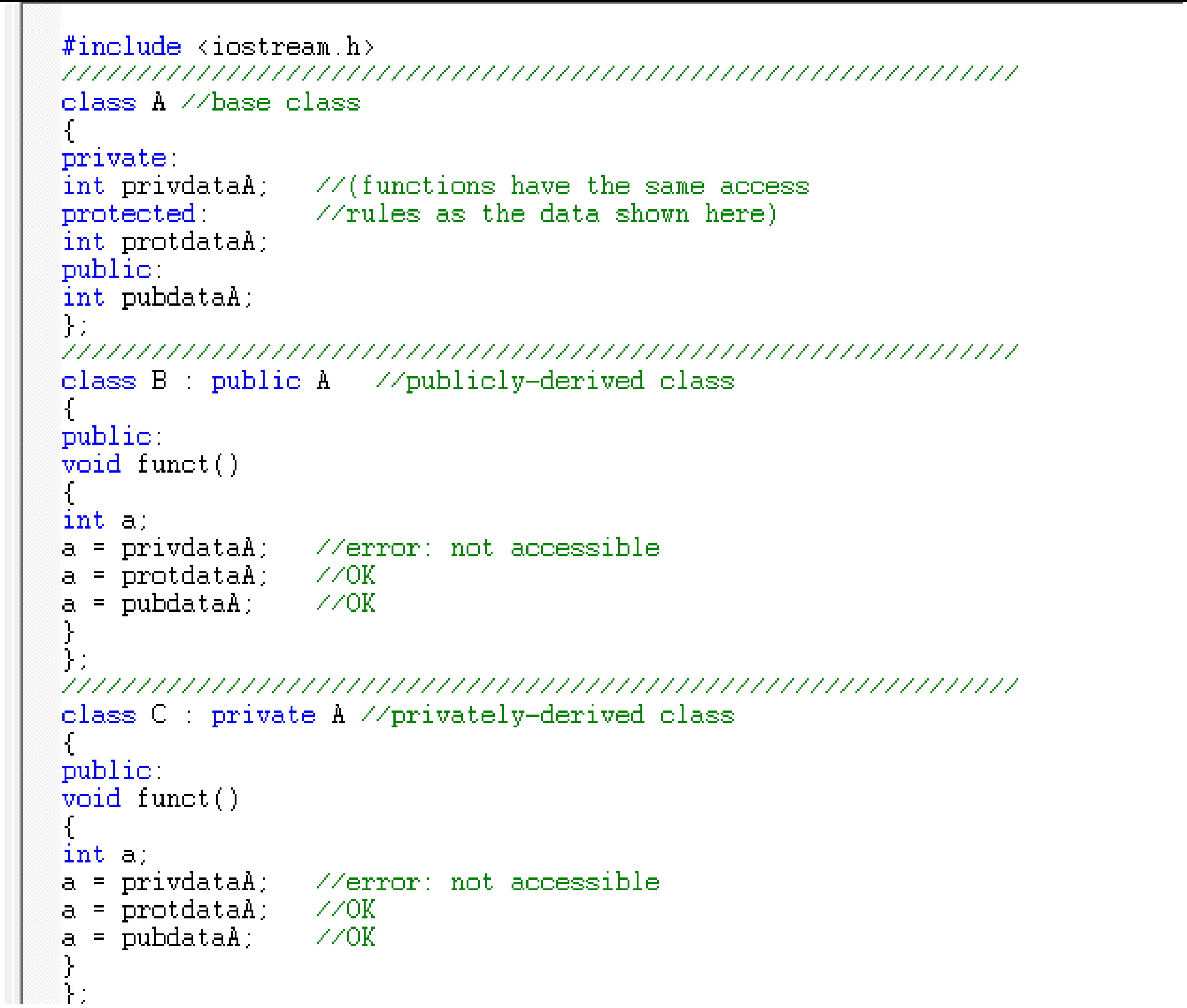
<u>“Abstract” Base Class</u>

It may seem that the laborer class is unnecessary, but by making it a separate class we emphasize that all classes are descended from the same source, employee. Also, if in the future we decided to modify the laborer class, we would not need to change the declaration for employee.

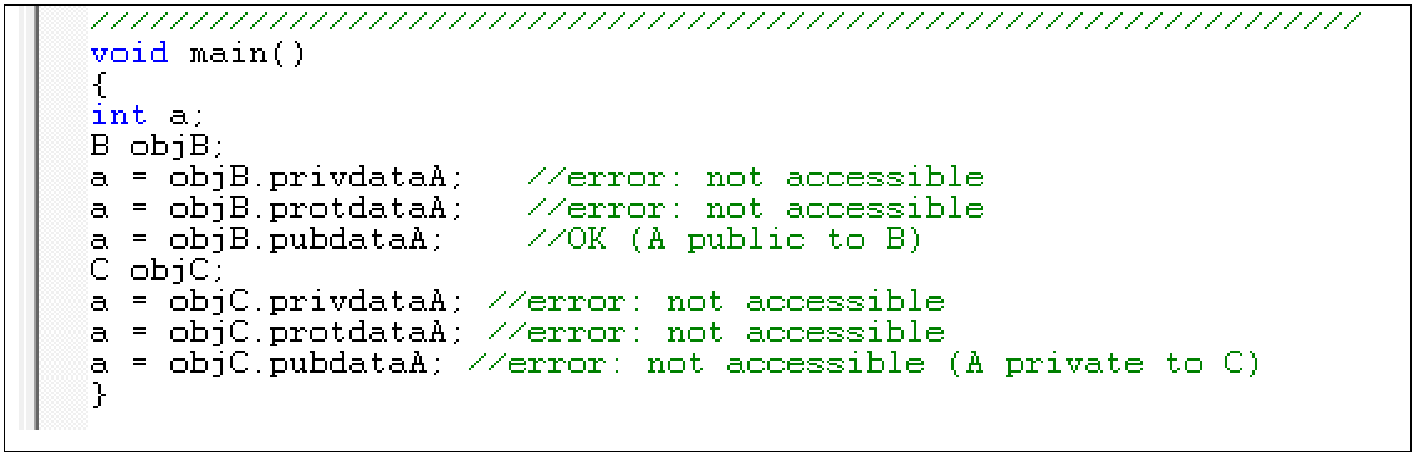
Access Combinations

There are so many possibilities for access that it’s instructive to look at Example 2:

Example 2:-



```
#include <iostream.h>
////////////////////////////////////////////////////////////////
class A //base class
{
private:
int privdataA;   //(functions have the same access
protected:       //rules as the data shown here)
int protdataA;
public:
int pubdataA;
};
////////////////////////////////////////////////////////////////
class B : public A   //publicly-derived class
{
public:
void funct()
{
int a;
a = privdataA;   //error: not accessible
a = protdataA;   //OK
a = pubdataA;    //OK
}
};
////////////////////////////////////////////////////////////////
class C : private A //privately-derived class
{
public:
void funct()
{
int a;
a = privdataA;   //error: not accessible
a = protdataA;   //OK
a = pubdataA;    //OK
}
};
```



```
/////////////////////////////////////////////////////////////////////////
void main()
{
int a;
B objB;
a = objB.privdataA;    //error: not accessible
a = objB.protdataA;    //error: not accessible
a = objB.pubdataA;     //OK (A public to B)
C objC;
a = objC.privdataA; //error: not accessible
a = objC.protdataA; //error: not accessible
a = objC.pubdataA; //error: not accessible (A private to C)
}
```

The program specifies a base class, **A**, with private, protected, and public data items. Two classes, **B** and **C**, are derived from **A**. **B** is publicly derived and **C** is privately derived. As we've seen before, functions in the derived classes can access protected and public data in the base class. Objects of the derived classes cannot access private or protected members of the base class. What's new is the difference between publicly derived and privately derived classes. Objects of the publicly derived class B can access public members of the base class **A**, while objects of the privately derived class **C** cannot; they can only access the public members of their own derived class. This is shown in Figure 2.

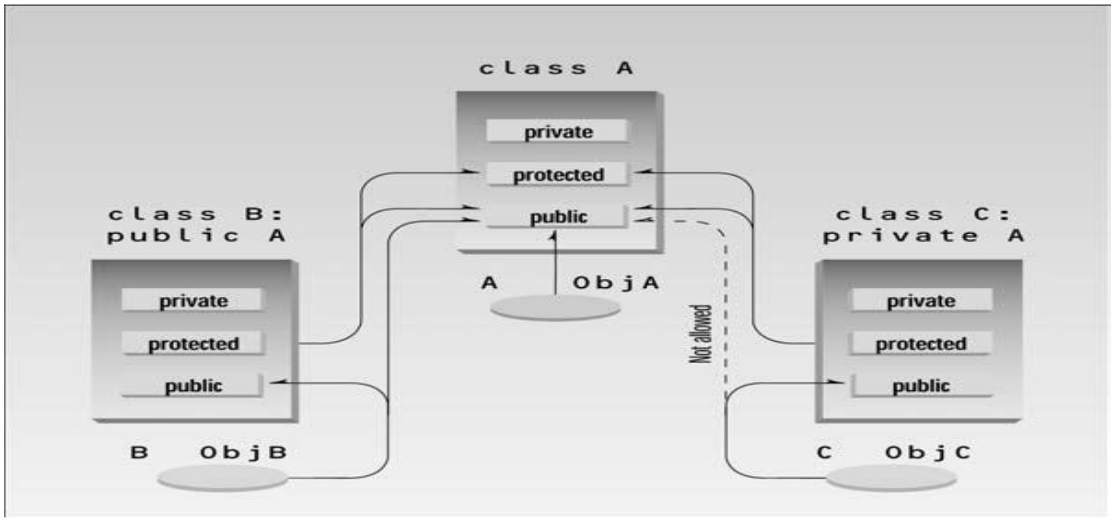


Figure2: ***Public and private derivation.***

<u>Levels of Inheritance</u>

Classes can be derived from classes that are themselves derived. Here's a miniprogram that shows the idea:

```
class A
{ };
class B : public A
{ };
class C : public B
{ };
```

Here B is derived from A, and C is derived from B. The process can be extended to an arbitrary number of levels D could be derived from C, and so on. Suppose that we decided to add a special kind of laborer called a *foreman* to the EMPLOY program. Since a foreman is a kind of laborer, the foreman class is derived from the laborer class, as shown in Figure 3.

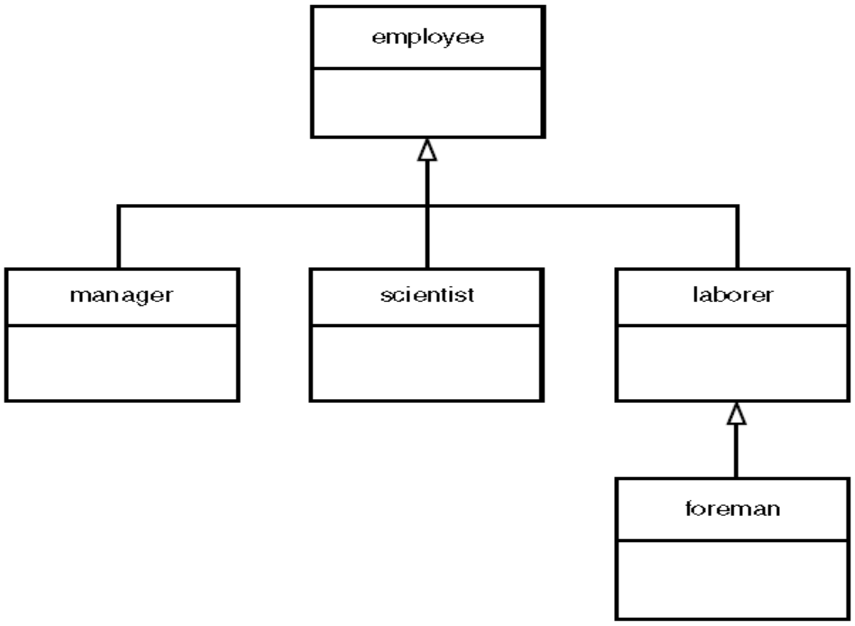
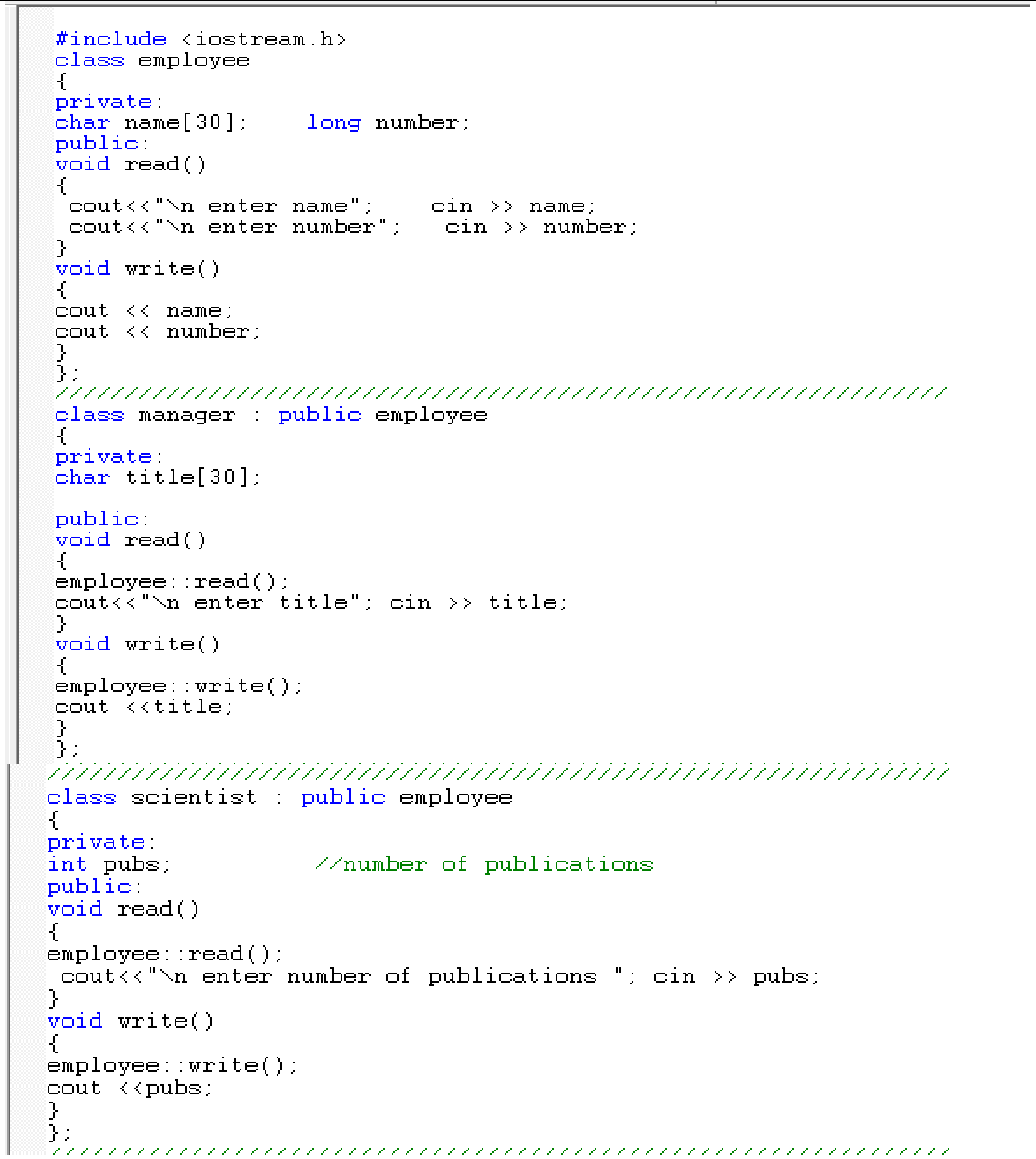
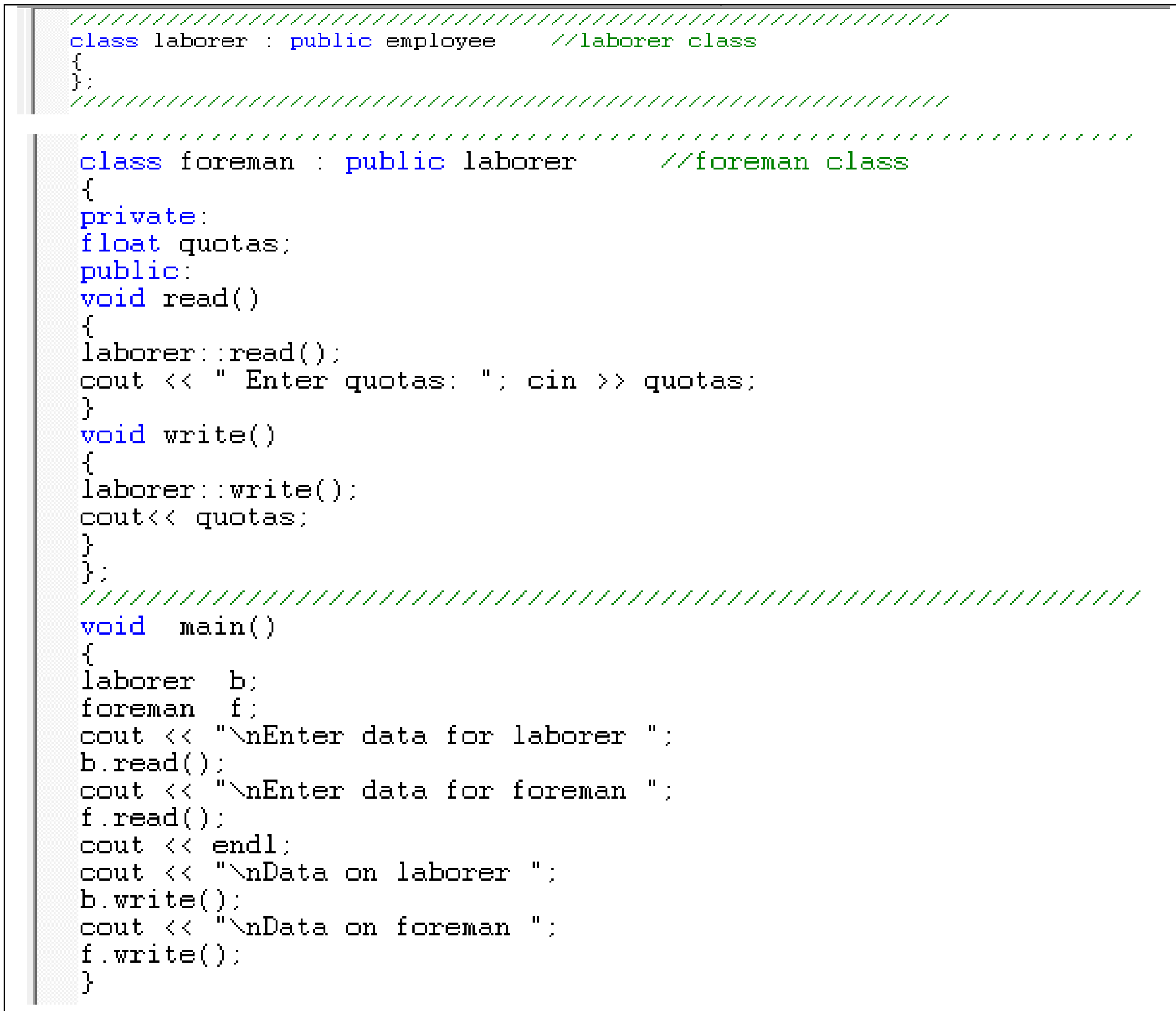


Figure 3: ***class diagram for EMPLOY2.***

Example 3:-Write an oo program to model employ database in figure3.



```
#include <iostream.h>
class employee
{
private:
char name[30];     long number;
public:
void read()
{
 cout<<"\n enter name";     cin >> name;
 cout<<"\n enter number";    cin >> number;
}
void write()
{
cout << name;
cout << number;
}
};
/////////////////////////////////////////////////////////////////
class manager : public employee
{
private:
char title[30];

public:
void read()
{
employee::read();
cout<<"\n enter title"; cin >> title;
}
void write()
{
employee::write();
cout <<title;
}
};
/////////////////////////////////////////////////////////////////
class scientist : public employee
{
private:
int pubs;           //number of publications
public:
void read()
{
employee::read();
 cout<<"\n enter number of publications "; cin >> pubs;
}
void write()
{
employee::write();
cout <<pubs;
}
};
/////////////////////////////////////////////////////////////////
```



```
/////////////////////////////////////////////////////////////////////
class laborer : public employee    //laborer class
{
};
/////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
class foreman : public laborer     //foreman class
{
private:
float quotas;
public:
void read()
{
laborer::read();
cout << " Enter quotas: "; cin >> quotas;
}
void write()
{
laborer::write();
cout<< quotas;
}
};
/////////////////////////////////////////////////////////////////////
void  main()
{
laborer  b;
foreman  f;
cout << "\nEnter data for laborer ";
b.read();
cout << "\nEnter data for foreman ";
f.read();
cout << endl;
cout << "\nData on laborer ";
b.write();
cout << "\nData on foreman ";
f.write();
}
```

Multiple Inheritances

A class can be derived from more than one base class. This is called **multiple inheritances**. Figure 4 shows how this looks when a class C is derived from base classes A and B.

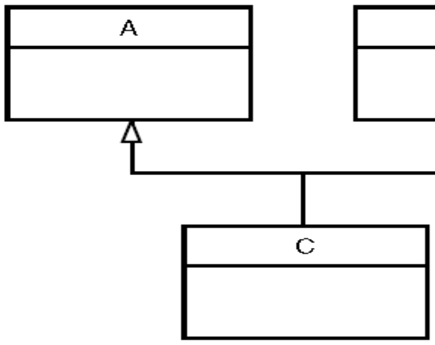


Figure 4: ***class diagram for multiple inheritances.***

The syntax for multiple inheritances is similar to that for single inheritance. In the situation shown in Figure 4, the relationship is expressed like this:

```
class A // base class A
{
};
class B // base class B
{
};
class C : public A, public B // C is derived from A and B
{
};
```

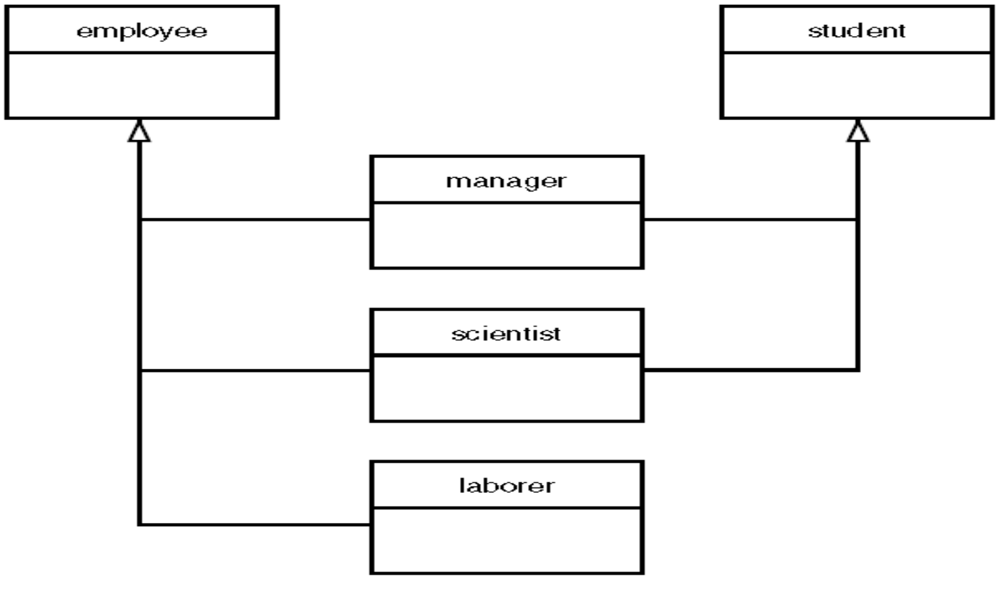
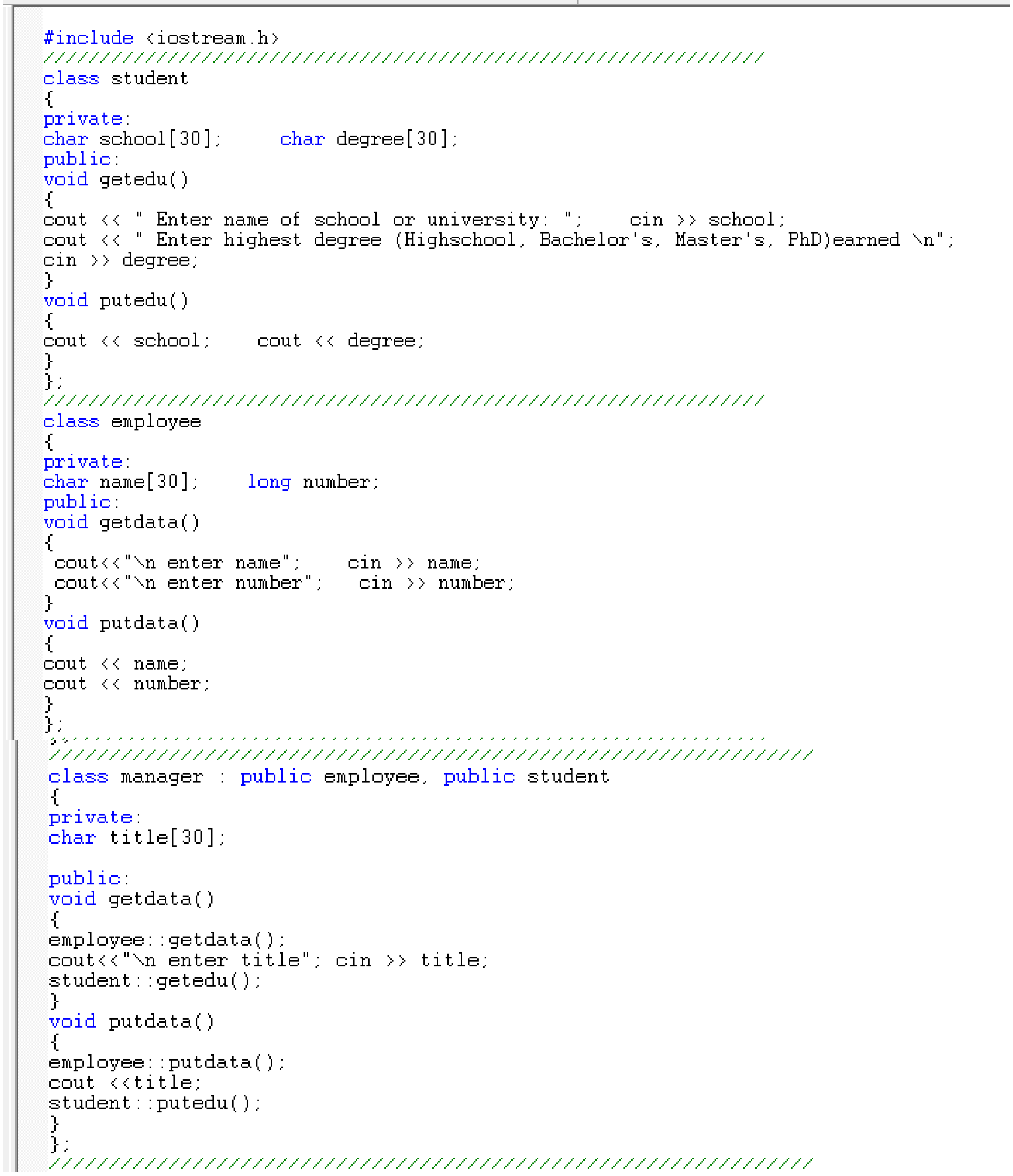
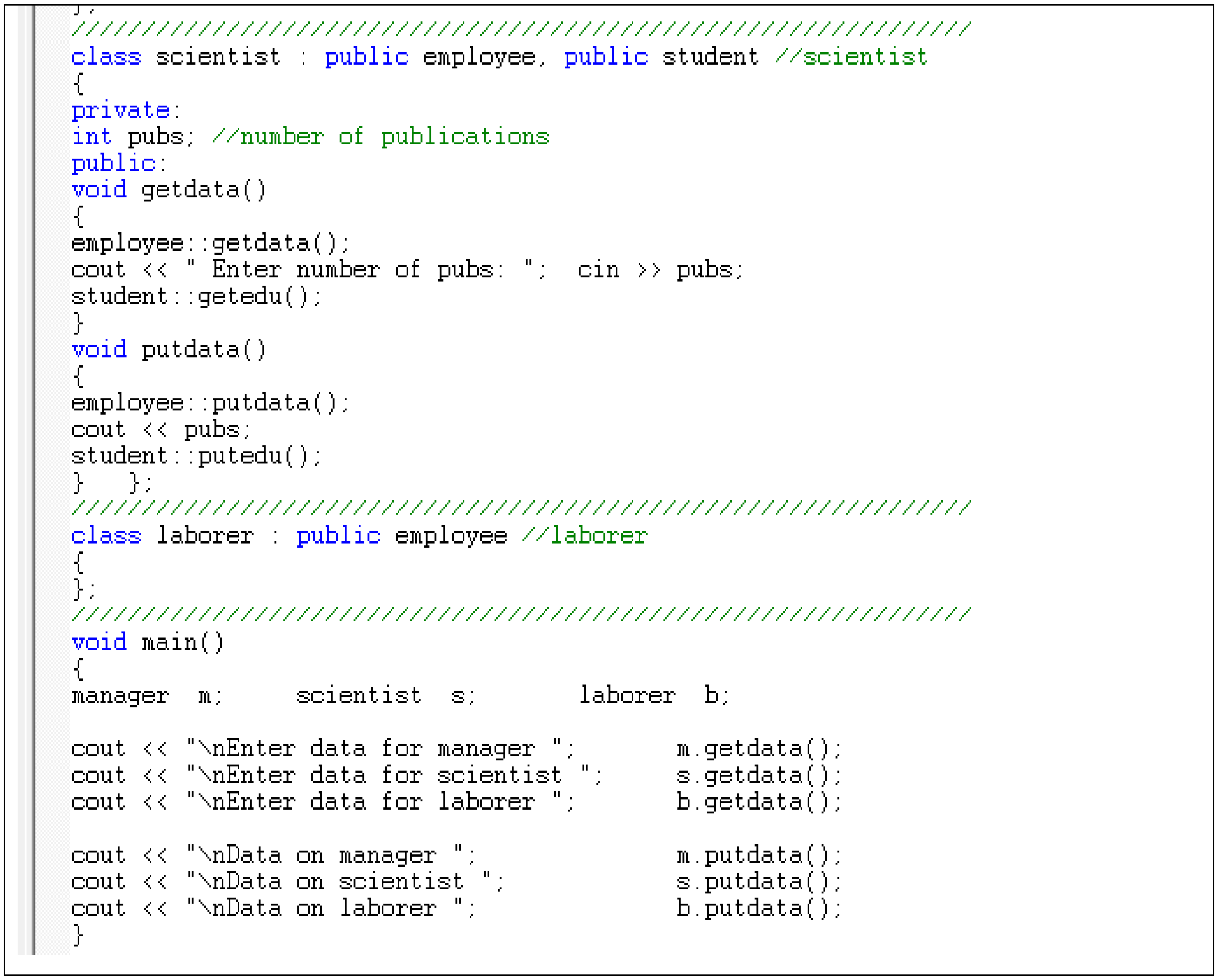


Figure5 : Multiple Inheritances with Employee .

Example 4:-Write an OOP to model employee database in figure5 .



```
#include <iostream.h>
/////////////////////////////////////////////////////////////////
class student
{
private:
char school[30];      char degree[30];
public:
void getedu()
{
cout << " Enter name of school or university: ";    cin >> school;
cout << " Enter highest degree (Highschool, Bachelor's, Master's, PhD)earned \n";
cin >> degree;
}
void putedu()
{
cout << school;    cout << degree;
}
};
/////////////////////////////////////////////////////////////////
class employee
{
private:
char name[30];    long number;
public:
void getdata()
{
 cout<<"\n enter name";    cin >> name;
 cout<<"\n enter number";   cin >> number;
}
void putdata()
{
cout << name;
cout << number;
}
};
/////////////////////////////////////////////////////////////////
class manager : public employee, public student
{
private:
char title[30];

public:
void getdata()
{
employee::getdata();
cout<<"\n enter title"; cin >> title;
student::getedu();
}
void putdata()
{
employee::putdata();
cout <<title;
student::putedu();
}
};
/////////////////////////////////////////////////////////////////
```



```
};
//////////////////////////////////////////////////////////////////////
class scientist : public employee, public student //scientist
{
private:
int pubs; //number of publications
public:
void getdata()
{
employee::getdata();
cout << " Enter number of pubs: ";  cin >> pubs;
student::getedu();
}
void putdata()
{
employee::putdata();
cout << pubs;
student::putedu();
}   };
//////////////////////////////////////////////////////////////////////
class laborer : public employee //laborer
{
};
//////////////////////////////////////////////////////////////////////
void main()
{
manager  m;     scientist  s;       laborer  b;

cout << "\nEnter data for manager ";      m.getdata();
cout << "\nEnter data for scientist ";    s.getdata();
cout << "\nEnter data for laborer ";      b.getdata();

cout << "\nData on manager ";             m.putdata();
cout << "\nData on scientist ";           s.putdata();
cout << "\nData on laborer ";             b.putdata();
}
```

Ambiguity in Multiple Inheritances

There are **two types ambiguity** in Multiple Inheritances

1. **Two base classes have functions** with the **same name**, while a class derived from both base classes has no function with this name. How do objects of the derived class access the correct base class function? The name of the function alone is insufficient, since the compiler can’t figure out which of the two functions is meant.

Example: demonstrates ambiguity in multiple inheritance

```
#include <iostream.h>
class A
{
public:
void show() {   cout << " A  \n”; }
};
class B
{
public:
void show() { cout << "  B  \n”; }
};
class C : public A, public B
{
};
/////////////////////////////////////////////////////////////////
void  main()
{
C     C1;    //object of class C
// C1. show();             //ambiguous--will not compile
C1.A :: show();            //OK
C1.B :: show();           //OK
}
```

The problem is resolved using the scope-resolution operator to specify the class in which the function lies. Thus **C1.A::show();** refers to the version of show() that’s in the A class, while **C1.B::show();** refers to the function in the B class.

2. Another kind of ambiguity arises if you derive a class from two classes that are each derived from the same class. This creates a **diamond-shaped** inheritance tree.

Example: investigates diamond-shaped multiple inheritance

```
#include <iostream.h>
class A
{
public:
void   print();
};
class B : public A
{ };
class C : public A
{ };
class D : public B, public C
{ };
////////////////////////////////////////////////////////////////
void   main()
{
D   D1;
D1.print();       //ambiguous: won't compile
}
```

Classes **B** and **C** are both derived from class **A**, and class **D** is derived by multiple inheritance from both **B** and **C**. **Trouble** starts if you try to access a member function in class **A** from an object of class **D**. In this example **D1** tries to access **print**(). However, both **B** and **C** contain a copy of **print(),** inherited from **A**. The **compiler can't decide which copy to use**, and signals an error.

Virtual Functions

Virtual means ***existing*** *in appearance but* ***not*** *in* ***reality***. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Why are virtual functions needed? Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call. For example, in suppose a graphics program in figure1 includes several different shapes: a **triangle**, a **ball**, a **square**. Each of these classes has a member function **draw ()** that causes the object to be drawn on the screen.

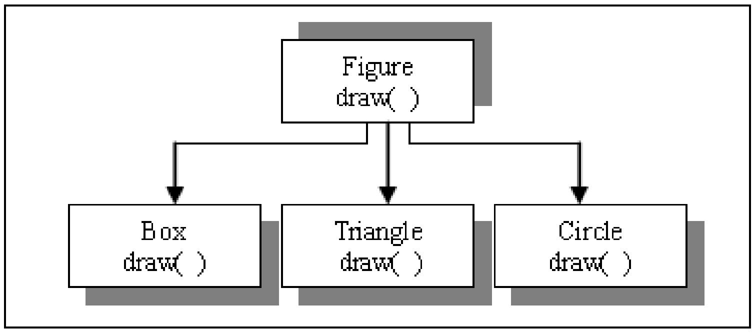


Figure 1: The class hierarchy for the Figures example.

Now suppose you plan to make a picture by grouping a number of these elements together and you want to draw the picture in a convenient way. One approach is to create an array that holds pointers to all the different objects in the picture. The array might be defined like this:

Figure *ptrarr[100]; // array of 100 pointers to Figures .If you insert pointers to all the shapes into this array, you can then draw an entire picture using loop:

For (int j=0; j<N; j++)   ptrarr[j]->draw();

This is an amazing capability: Completely different functions are executed by the same function call. If the pointer in ptrarr points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle-drawing function is called. This is called ***polymorphism***, which means ***different forms***.

Polymorphism

Polymorphism is one of the crucial features of object oriented programming. It simply means "***one name, multiple forms***". However, polymorphism allows an entity (variable, function or object) to take a variety of representations (take a multiple forms).

In C++ Polymorphism is implemented via ***virtual functions.***

Therefore we have to distinguish different three types of polymorphism:

A. Polymorphism of Variables:

The first type of polymorphism is similar to the concept of dynamic binding. Here, the type of a variable depends on its content. Thus, its type depends on the content at a specific time:

```
int     a=5;    //use a as integer
…..
char    a='g';  //use a as character
……
```

B. Polymorphism of Functions:

Another type of polymorphism can be defined for functions. For example, suppose you want to define a function *isNull( )* which returns TRUE if its argument is zero and FALSE otherwise. For integer numbers this is easy:

```
Bool  isNull(int  r)
{
If (r==0)     Return(true)
Else   Return(false)   }
```

However, if we want to check this for float numbers, we should use another comparison due to the precision problem:

```
Bool   isNull(float  k)
{
If (k<=0.01)&&(k>-0.99)
Return(true)
Else
Return(false)
}
```

Since the parameter list of both *isNull* functions differs, the compiler is able to figure-out the correct function call by using the actual types of the arguments.

```
int  r;
float   k;
r=0;
k=0.0;
If ( isNull(r))   //use isNull integer
If ( isNull(k))   //use isNull float
```

This type of polymorphism allows us to reuse the same name for functions (or methods) as long as the parameter list differs. Sometimes this type of polymorphism is called **overloading**.

C. Polymorphism of Objects:

The last type of polymorphism allows an object to choose correct methods. In this type, polymorphism refers to situation in which objects belong to different classes can be respond to the same message, usually in different ways. For example, suppose we have classes box, triangle, and circle, whose objects represent the corresponding geometrical figures, as shown in figure (1).

For the polymorphic approach to work, several conditions must be met.

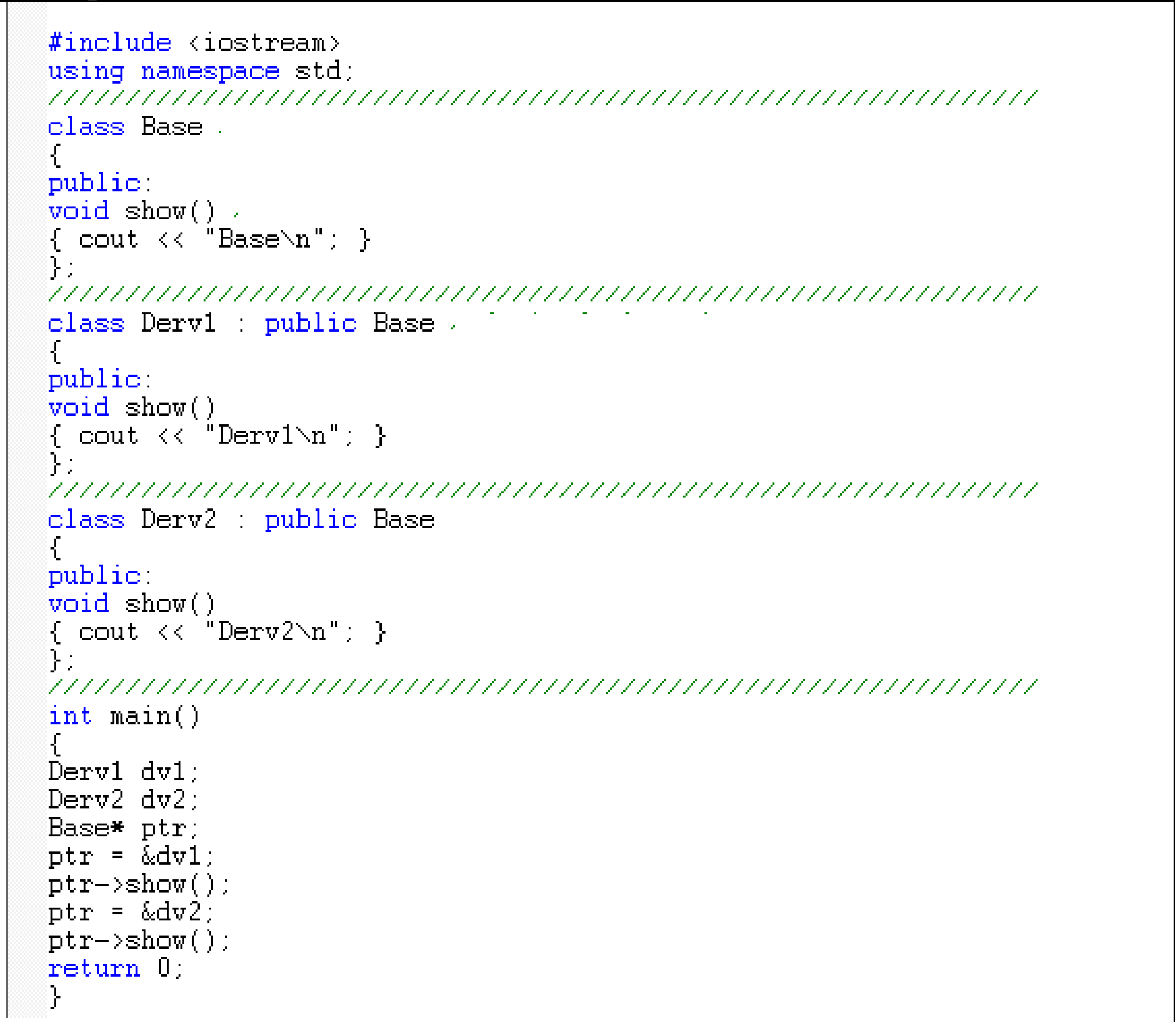
1) First, all the different classes of shapes, such as balls and triangles, must be descended from a single base class.

2) Second, the draw() function must be declared to be virtual in the base class.

<u>Normal Member Functions Accessed with Pointers</u>

Example 1 shows what happens when a base class and derived classes all have functions with the same name.

Example 1:



```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base
{
public:
void show()
{ cout << "Base\n"; }
};
////////////////////////////////////////////////////////////////
class Derv1 : public Base
{
public:
void show()
{ cout << "Derv1\n"; }
};
////////////////////////////////////////////////////////////////
class Derv2 : public Base
{
public:
void show()
{ cout << "Derv2\n"; }
};
////////////////////////////////////////////////////////////////
int main()
{
Derv1 dv1;
Derv2 dv2;
Base* ptr;
ptr = &dv1;
ptr->show();
ptr = &dv2;
ptr->show();
return 0;
}
```

The **Derv1** and **Derv2** classes are derived from class **Base**. Each of these three classes has a member function **show()**. In main () we create objects of class **Derv1** and **Derv2**, and a pointer to class Base. Then we put the address of a derived class object in the base class pointer in the line

ptr = &dv1; // derived class address in base class pointer .

Now the question is, when you execute the line :-

ptr->show(); what function is called? Is it **show() of Base or show()** of **Derv1**? Again, in the last two lines of not virtual we put the address of an object of class **Derv2** in the pointer, and again execute

ptr->show(); Which of the **show()** functions is called here?

The output from the program:

Base

Base

As you can see, the function in the base class is always executed. The compiler **ignores** the ***contents*** of the pointer **ptr** and chooses the member function that matches the *type* of the pointer, as shown in figure 2

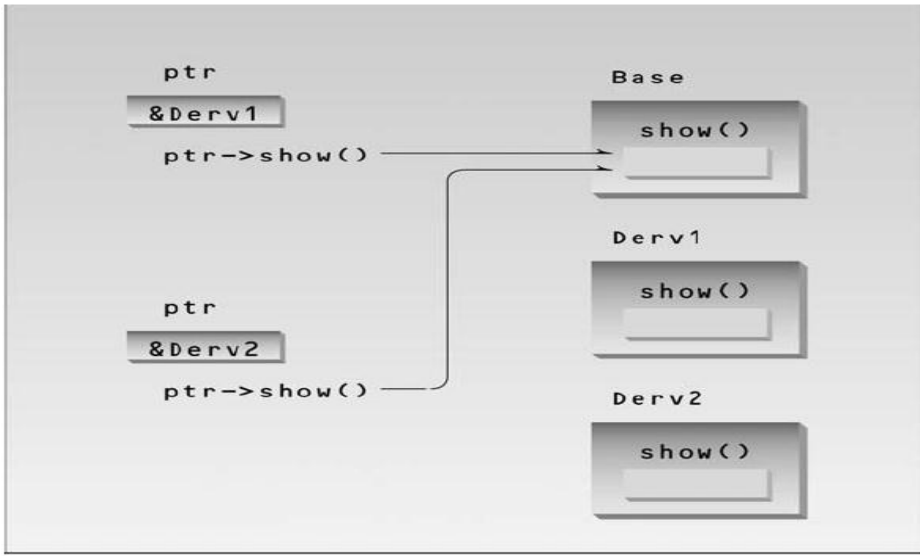
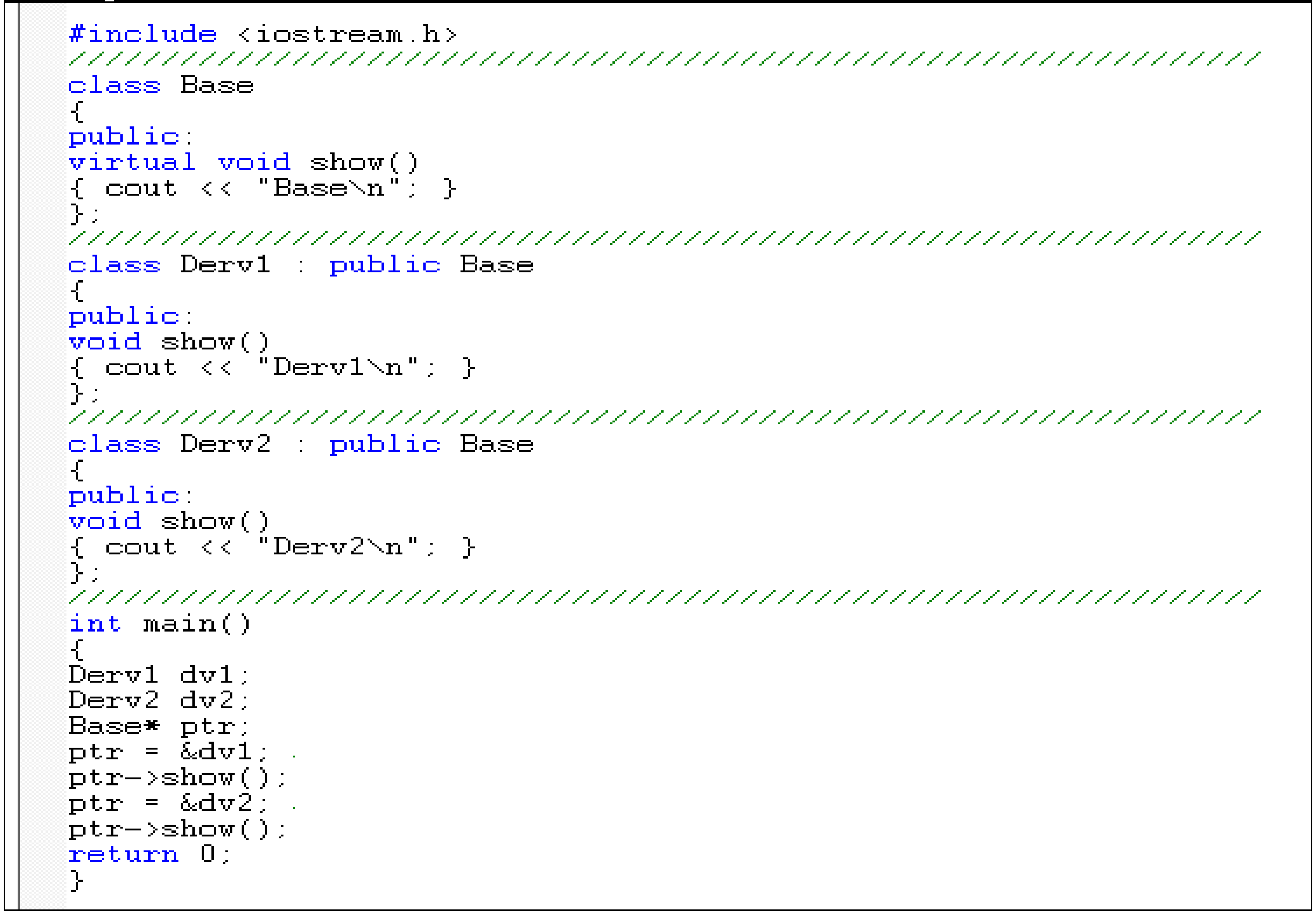


Figure 2 ***Nonvirtual pointer accesses.***

Virtual Member Functions Accessed with Pointers

We’ll place the keyword *virtual* in front of the declarator for the show() function in the base class. Here’s the listing for the resulting program.

Example 2:



```
#include <iostream.h>
//////////////////////////////////////////////////////////////////////
class Base
{
public:
virtual void show()
{ cout << "Base\n"; }
};
//////////////////////////////////////////////////////////////////////
class Derv1 : public Base
{
public:
void show()
{ cout << "Derv1\n"; }
};
//////////////////////////////////////////////////////////////////////
class Derv2 : public Base
{
public:
void show()
{ cout << "Derv2\n"; }
};
//////////////////////////////////////////////////////////////////////
int main()
{
Derv1 dv1;
Derv2 dv2;
Base* ptr;
ptr = &dv1; .
ptr->show();
ptr = &dv2; .
ptr->show();
return 0;
}
```

The output of this program is

Derv1

Derv2

The member functions of the derived classes, not the base class, are executed. We change the contents of ptr from the address of Derv1 to that of Derv2, and the instance of show() that is executed also changes. So the same function call **ptr->show();** executes different functions, depending on the contents of ptr. The rule is that the **compiler** selects the function based on the *contents* of the pointer ptr, not on the *type* of the pointer, as in not virtual. This is shown in figure 3

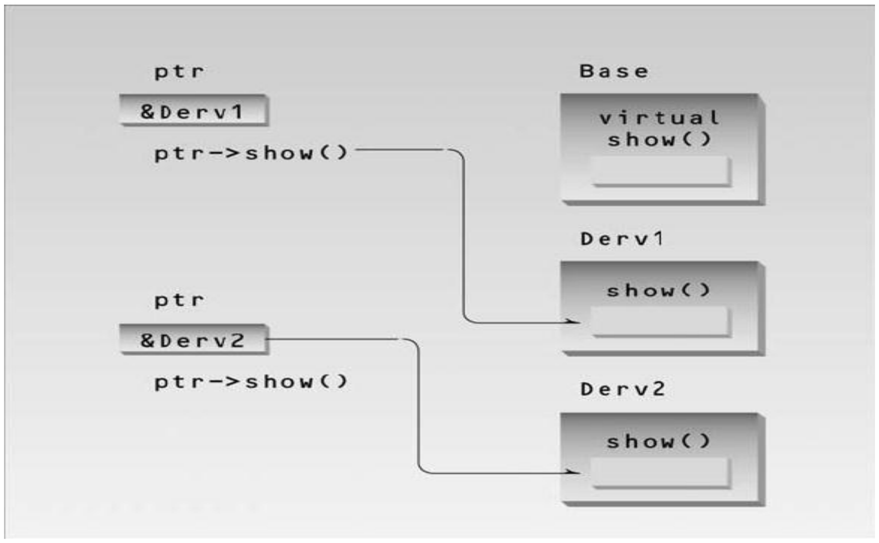


Figure 3 ***Virtual pointer access.***

Late Binding

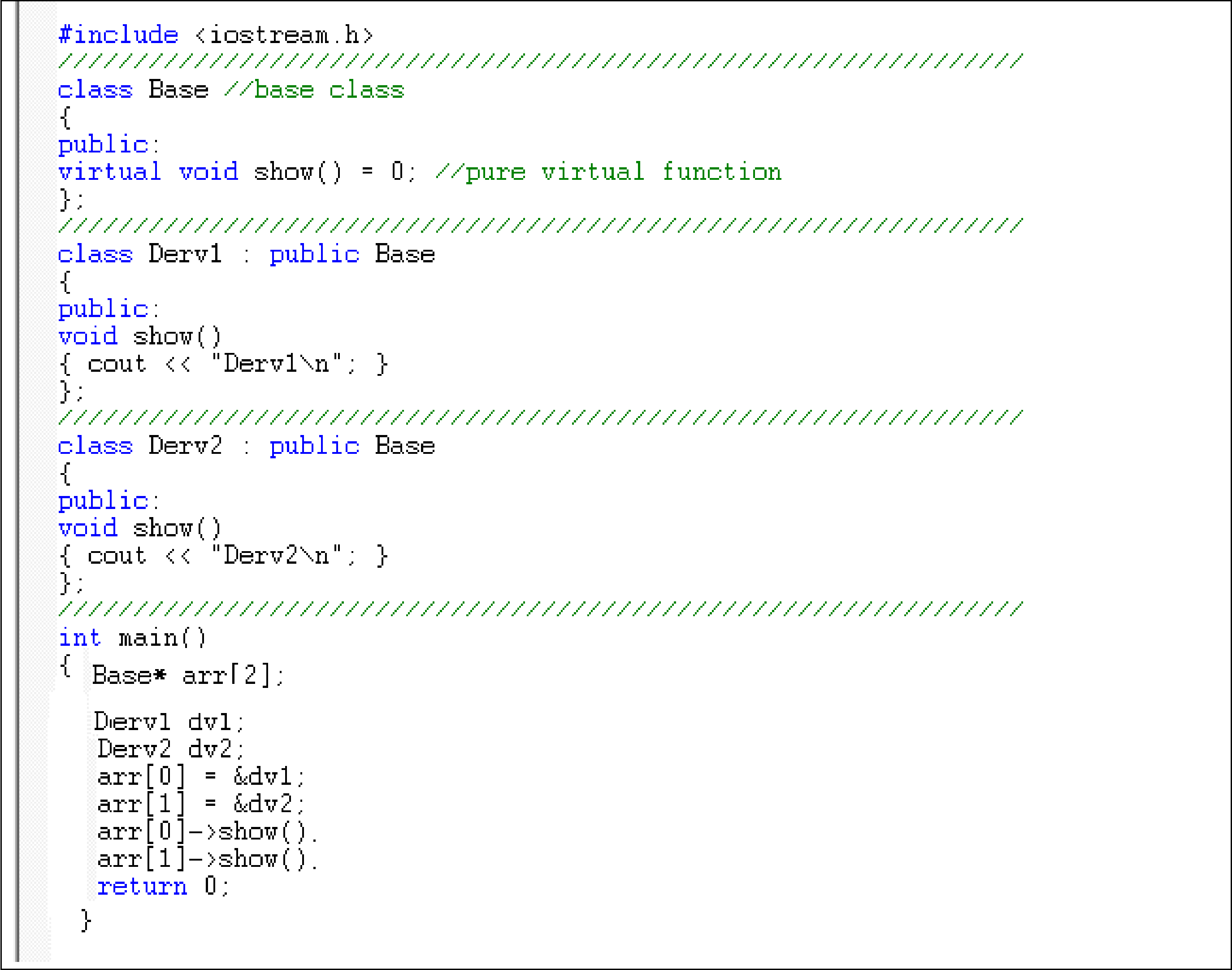
In not virtual the compiler has no problem with the expression **ptr->show();** It always compiles a call to the show() function in the **base** class. But in virtual the compiler doesn't know what class the contents of ptr may contain. It could be the address of an object of the **Derv1** class or of the **Derv2** class. Which version of draw() does the compiler call? In fact the compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what class is pointed to by ptr, the appropriate version of draw will be called. This is called ***late binding*** or ***dynamic binding***.(Choosing functions in the normal way, during compilation, is called ***early binding*** or ***static binding***.) Late binding requires some overhead but provides increased power and flexibility.

Abstract Classes and Pure Virtual Functions

When we will never want to instantiate objects of a base class, we call it an ***abstract class***. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. It may also provide an interface for the class hierarchy. By placing at least one ***pure virtual function*** in the base class.

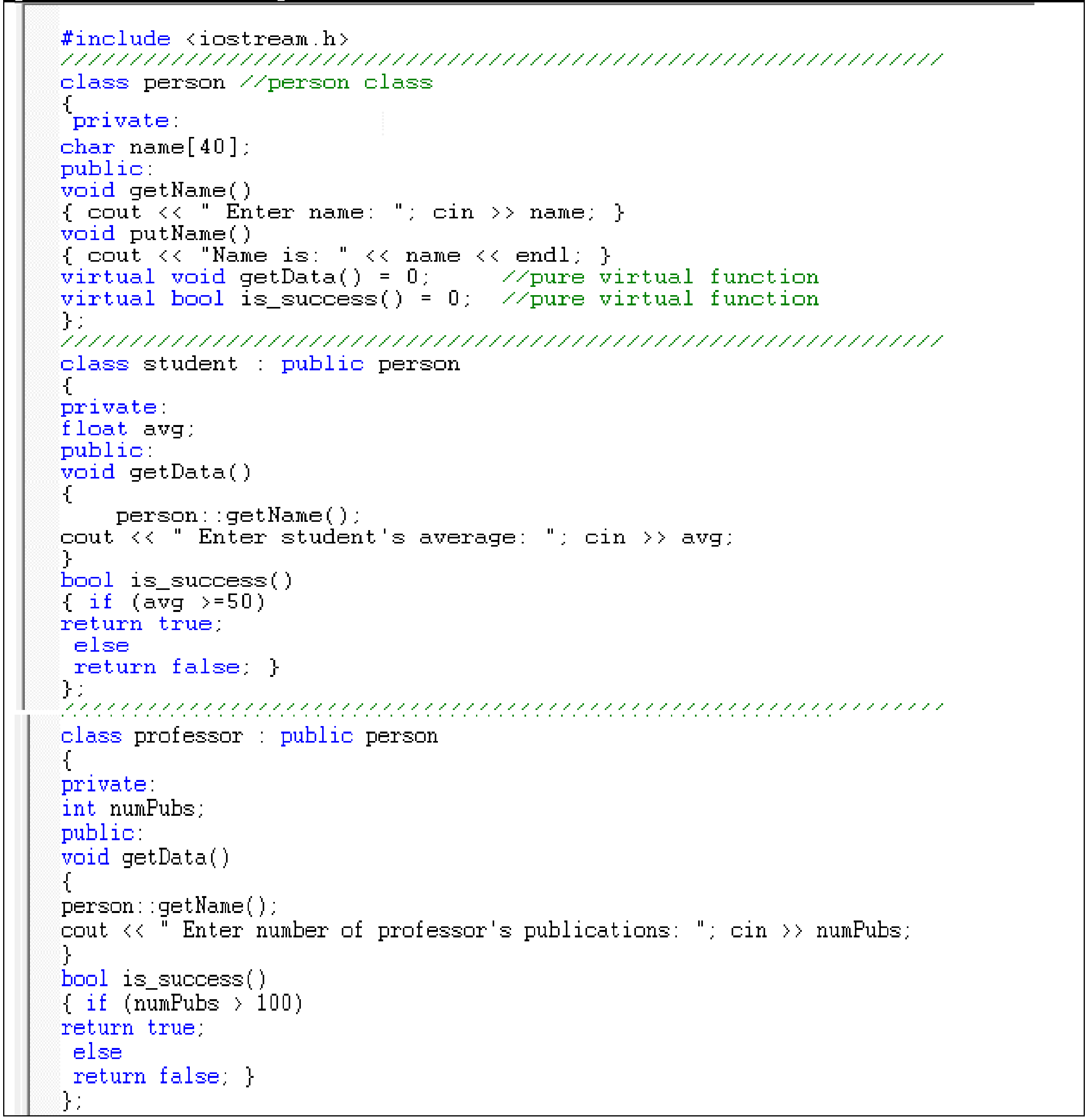
A pure virtual function is one with the expression =0 added to the declaration. This is shown in the example3.

Example 3:

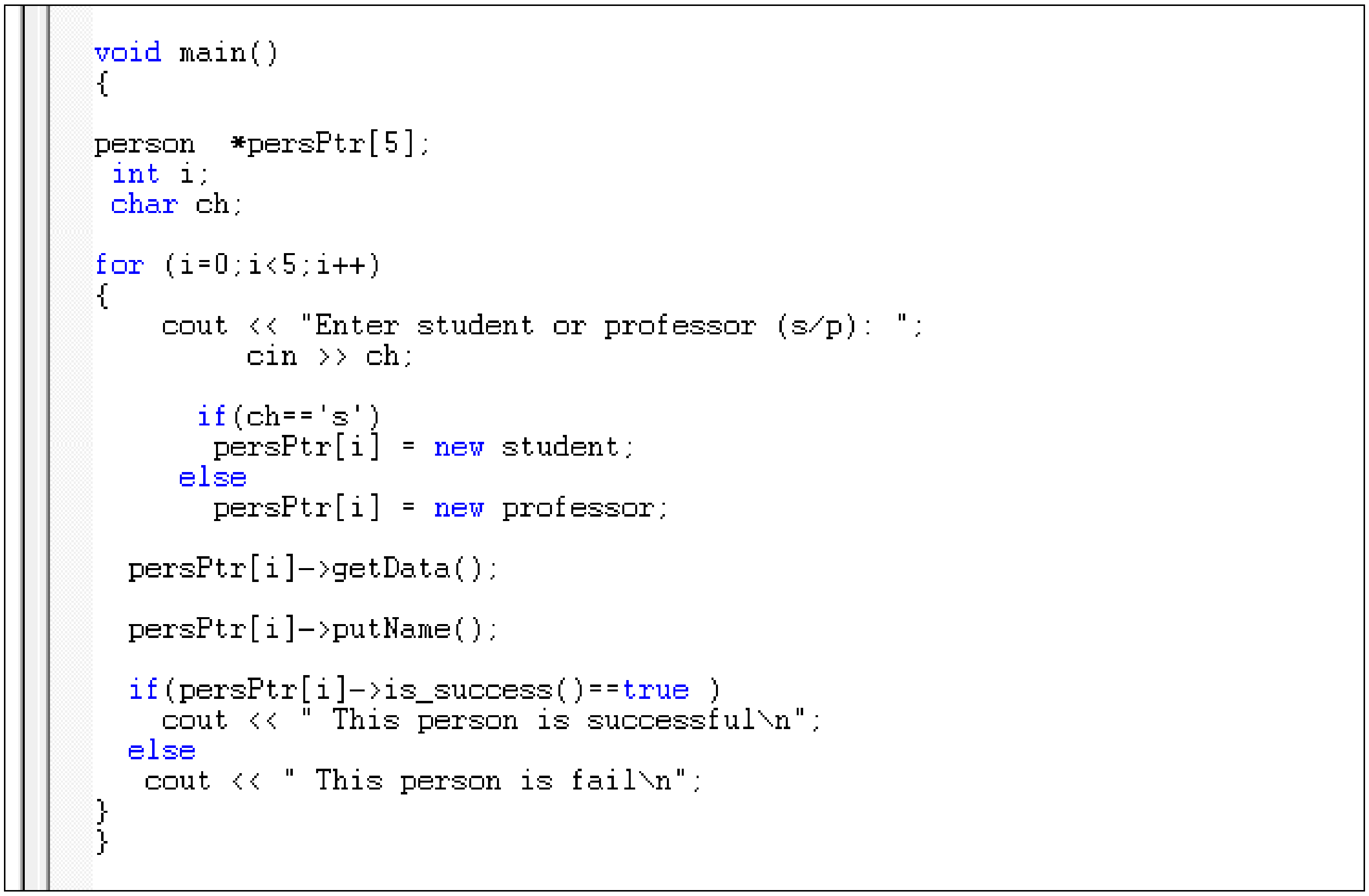


```
#include <iostream.h>
//////////////////////////////////////////////////////////////////////
class Base //base class
{
public:
virtual void show() = 0; //pure virtual function
};
//////////////////////////////////////////////////////////////////////
class Derv1 : public Base
{
public:
void show()
{ cout << "Derv1\n"; }
};
//////////////////////////////////////////////////////////////////////
class Derv2 : public Base
{
public:
void show()
{ cout << "Derv2\n"; }
};
//////////////////////////////////////////////////////////////////////
int main()
{  Base* arr[2];

   Derv1 dv1;
   Derv2 dv2;
   arr[0] = &dv1;
   arr[1] = &dv2;
   arr[0]->show().
   arr[1]->show().
   return 0;
  }
```

Example 4: Write an OOP to read and chek a person if successful or fail. Create a class called person which contains name of type string and derived a class student and a class professor from a class person



```
#include <iostream.h>
//////////////////////////////////////////////////////////////////////////
class person //person class
{
 private:
char name[40];
public:
void getName()
{ cout << " Enter name: "; cin >> name; }
void putName()
{ cout << "Name is: " << name << endl; }
virtual void getData() = 0;      //pure virtual function
virtual bool is_success() = 0;  //pure virtual function
};
//////////////////////////////////////////////////////////////////////////
class student : public person
{
private:
float avg;
public:
void getData()
{
    person::getName();
cout << " Enter student's average: "; cin >> avg;
}
bool is_success()
{ if (avg >=50)
return true;
 else
 return false; }
};
//////////////////////////////////////////////////////////////////////////
class professor : public person
{
private:
int numPubs;
public:
void getData()
{
person::getName();
cout << " Enter number of professor's publications: "; cin >> numPubs;
}
bool is_success()
{ if (numPubs > 100)
return true;
 else
 return false; }
};
```



```
void main()
{

person  *persPtr[5];
 int i;
 char ch;

for (i=0;i<5;i++)
{
    cout << "Enter student or professor (s/p): ";
         cin >> ch;

      if(ch=='s')
       persPtr[i] = new student;
     else
       persPtr[i] = new professor;

  persPtr[i]->getData();

  persPtr[i]->putName();

  if(persPtr[i]->is_success()==true )
    cout << " This person is successful\n";
  else
   cout << " This person is fail\n";
}
}
```

Virtual Base Classes

Consider the situation shown in Figure 4, with a base class, **A**; two derived classes, **B** and **C**; and a fourth class, **D**, derived from both **B** and **C**.In this arrangement a problem can arise if a member function in the **D** class wants to access data or functions in the A class.

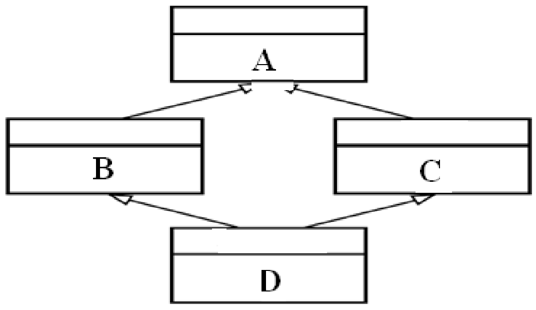


Figure 4 ***Virtual base classes.***

```
// ambiguous reference to base class
class  A
{    protected:
           int  d;
};
class B : public A          {       };
class C : public A           {        };
class  D : public B, public C
{      public:
     void  show()
           { cout<< d; }    // ERROR: ambiguous
};
```

```
                    // virtual base classes
class  A
{    protected:
                int  d;
};
class   B : virtual public  A
{ };
class   C : virtual public  A
{ };
class  D : public B, public  C
{    public:
void  show()       {  cout<< d;   }
};
```

Function Template

Suppose you want to write a function that prints the absolute value of two numbers. The absolute value of a number is its value without regard to its sign: The absolute value of 3 is 3, and the absolute value of –3 is also 3. Ordinarily this function would be written for a particular data type:

```
void  abs(int n)
{
if (n<0)
cout<<-n
else
cout<<n;
}
```

Here the function is defined to take an argument of type **int** and to print a value of this same type. But now suppose you want to find the absolute value of a type **long**. You will need to write a completely new function:

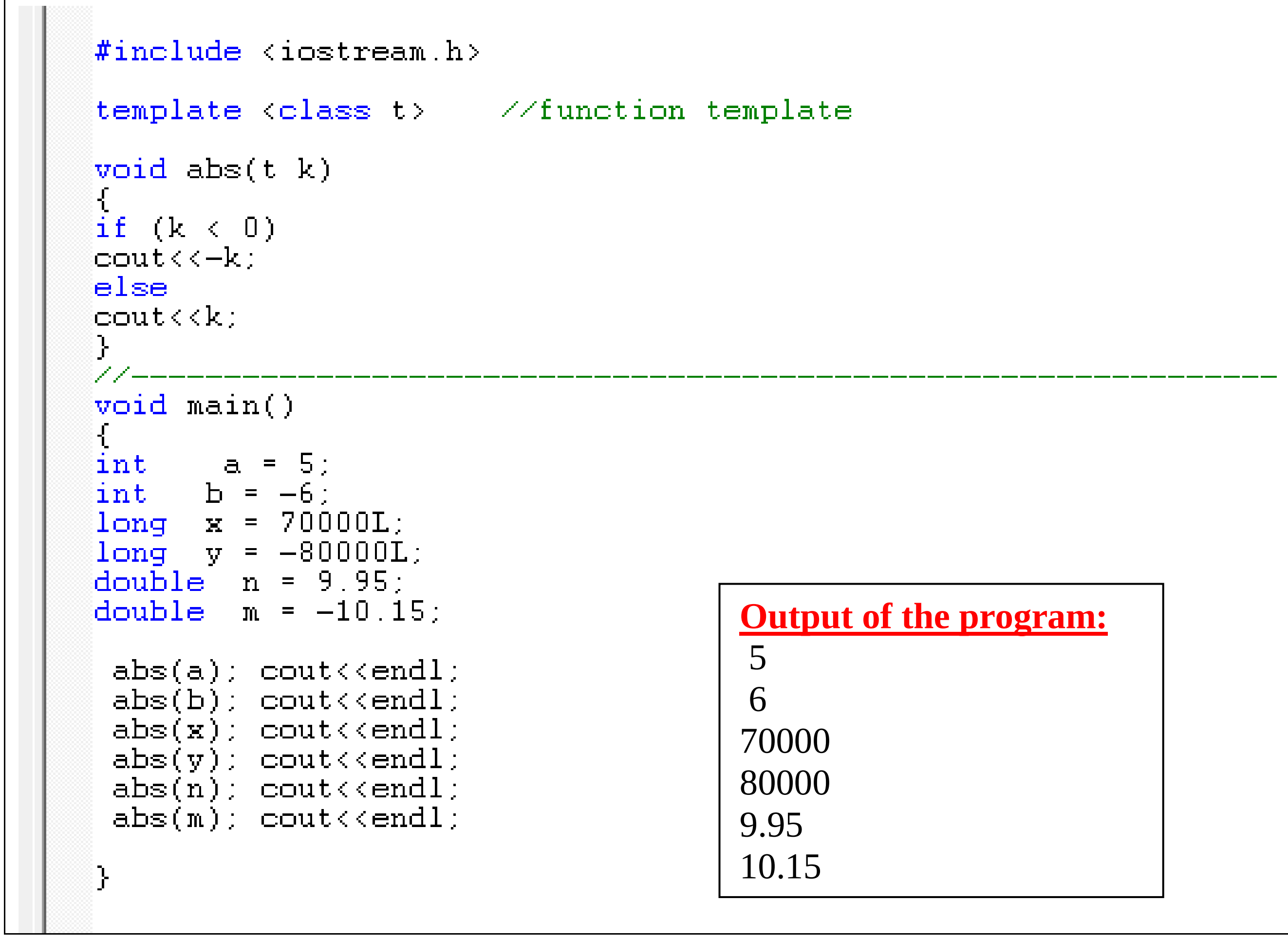
```
void  abs(long  n)
{
if (n<0)
cout<<-n
else
cout<<n;
}
```

The body of the function is written the same way in each case, but they are completely different functions because they handle arguments and the values of different types.

A) Simple Function Template

This program defines a template version of **abs**().

Example 1:Write an OO Program to find the absolute value using template function.



```
#include <iostream.h>

template <class t>    //function template

void abs(t k)
{
if (k < 0)
cout<<-k;
else
cout<<k;
}
//----------------------------------------------------------------
void main()
{
int     a = 5;
int    b = -6;
long   x = 70000L;
long   y = -80000L;
double  n = 9.95;
double  m = -10.15;

 abs(a); cout<<endl;
 abs(b); cout<<endl;
 abs(x); cout<<endl;
 abs(y); cout<<endl;
 abs(n); cout<<endl;
 abs(m); cout<<endl;

}
```

Output of the program:

```
 5
 6
70000
80000
9.95
10.15
```

The **abs ()** function now works with all three of the data types (int, long, and double) that we use as arguments.

This entire syntax, with a first line starting with the keyword ***template*** and the function definition following, is called a ***function template***.

Function Template Syntax

The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, but by a name that can stand for *any* type. In the preceding function template, this name is **t**. The **template** keyword signals the compiler that we're about to define a function template. The keyword **class**, within the angle brackets, might just as well be called type. You can define your own data types using classes, so there's really no distinction between types and classes. The variable following the keyword class (**t** in this example) is called the ***template argument***.

What the Compiler Does

What does the compiler do when it sees the template keyword and the function definition that follows it? The function template itself doesn't cause the compiler to generate any code. It can't generate code because it doesn't know yet what data type the function will be working with. It simply remembers the template for possible future use.

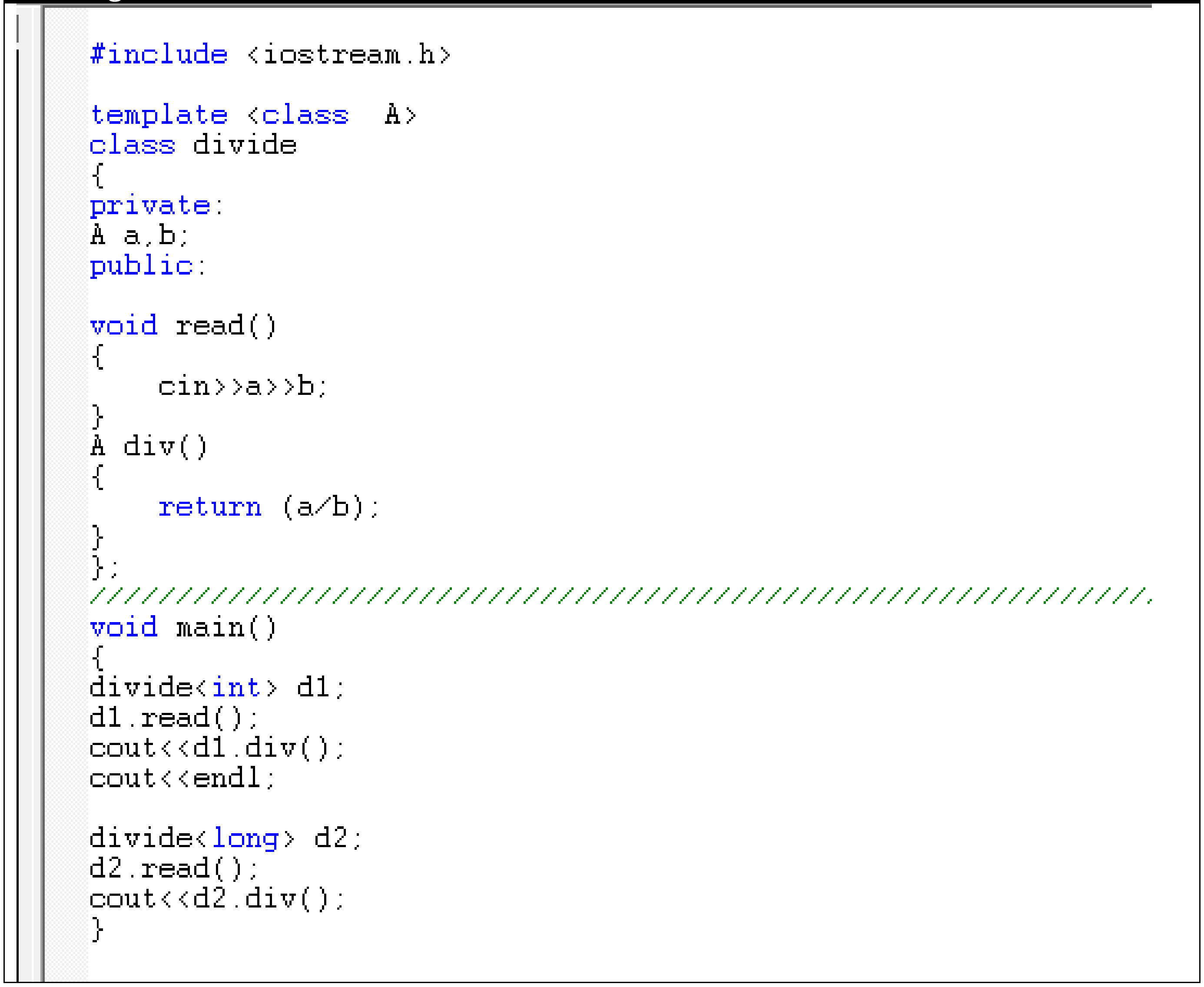
Code generation doesn't take place until the function is actually called (invoked) by a statement within the main program. In example 1 this happens in expressions like abs(a) in the statement **abs(a);**

When the compiler sees such a function call, it knows that the type to use is **int**, because that's the type of the argument a. So it generates a specific version of the **abs()** function for type int, substituting int wherever it sees the name **t** in the function template. This is called ***instantiating*** the function template, and each instantiated version of the function is called a ***template function***.

B) ***Class Template***

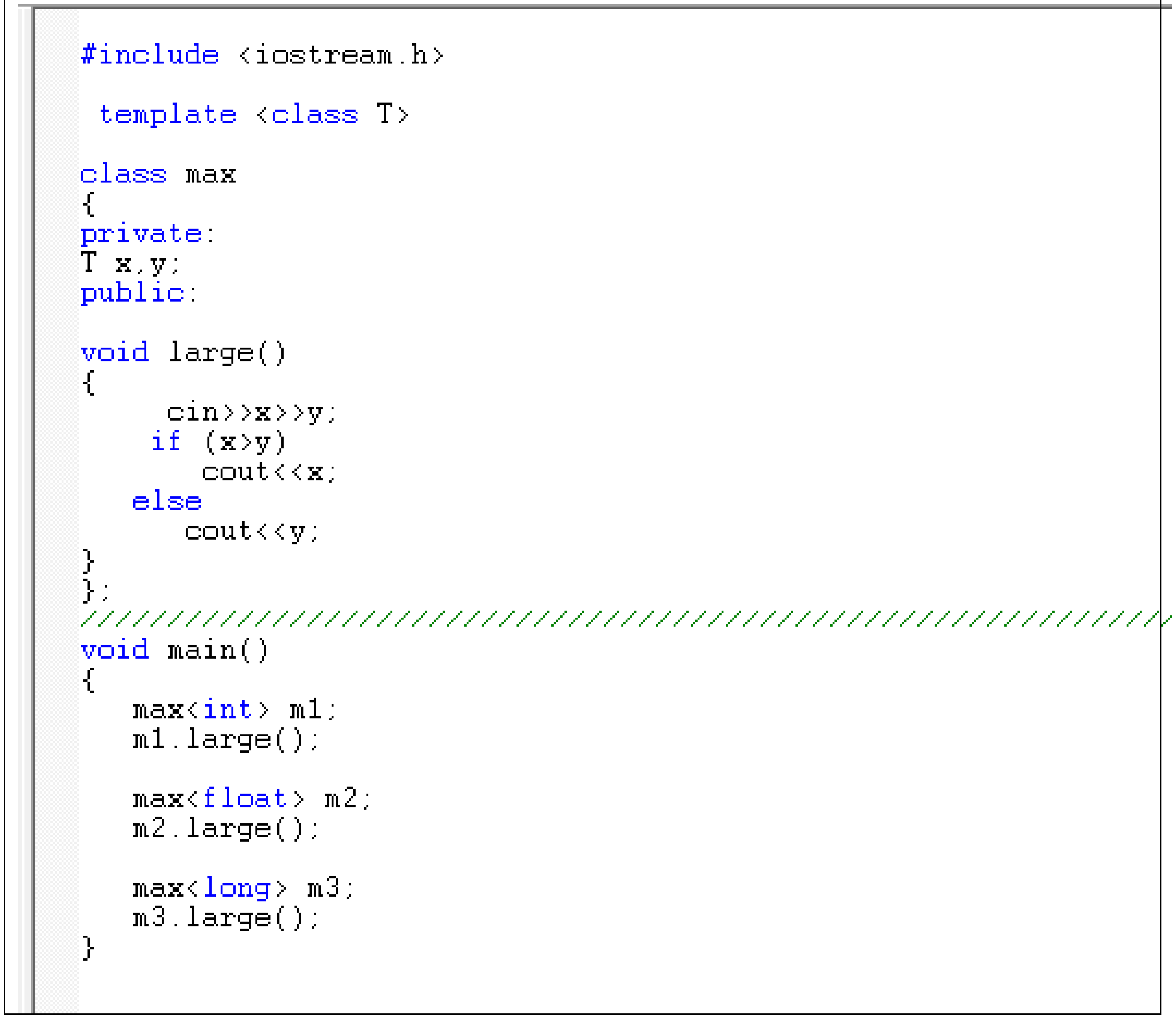
The template concept can be extended to classes. Class templates are generally used for data storage (container) classes.

Example 2: Write an OOP that include a class called divide . Use the concept of template class to write the program. The divide class includes a and b as private, and two functions read() and a function div() to divide a on b. The main program includes the call of the objects of type **integer** and **long**.



```
#include <iostream.h>

template <class  A>
class divide
{
private:
A a,b;
public:

void read()
{
    cin>>a>>b;
}
A div()
{
    return (a/b);
}
};
//////////////////////////////////////////////////////////////////////////////.
void main()
{
divide<int> d1;
d1.read();
cout<<d1.div();
cout<<endl;

divide<long> d2;
d2.read();
cout<<d2.div();
}
```

Example 3: Write an OOP that include a class called **max** . Use the concept of template class to write the program. The max class includes **x** and **y** as private attributes, and a functions **large()** to find the largest number between x and y. The main program includes the call of the functions to objects of type **integer**, **float** and **long**.



```
#include <iostream.h>

 template <class T>

class max
{
private:
T x,y;
public:

void large()
{
     cin>>x>>y;
    if (x>y)
       cout<<x;
   else
      cout<<y;
}
};
//////////////////////////////////////////////////////////////
void main()
{
   max<int> m1;
   m1.large();

   max<float> m2;
   m2.large();

   max<long> m3;
   m3.large();
}
```