Member 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 such as *int, float* or sometimes an *array*. So far, it has been shown that how a pointer variable can be declared as a member of a class.

For example, the following declaration of creating an object

```
Class sample
{
Private :
  int   x;
  float   y;
  char  s;
public:
   void   getdata();
   void   display();
};
Sample    *ptr;
```

Which *ptr* is apointer variable that holds the address of the class object sample and consists of three data members such as *int x, float y, and char s* ,and also holds member functions such as getdata() and display().

The pointer to an object of class vaiable will be accessed and processed in one of the following ways .

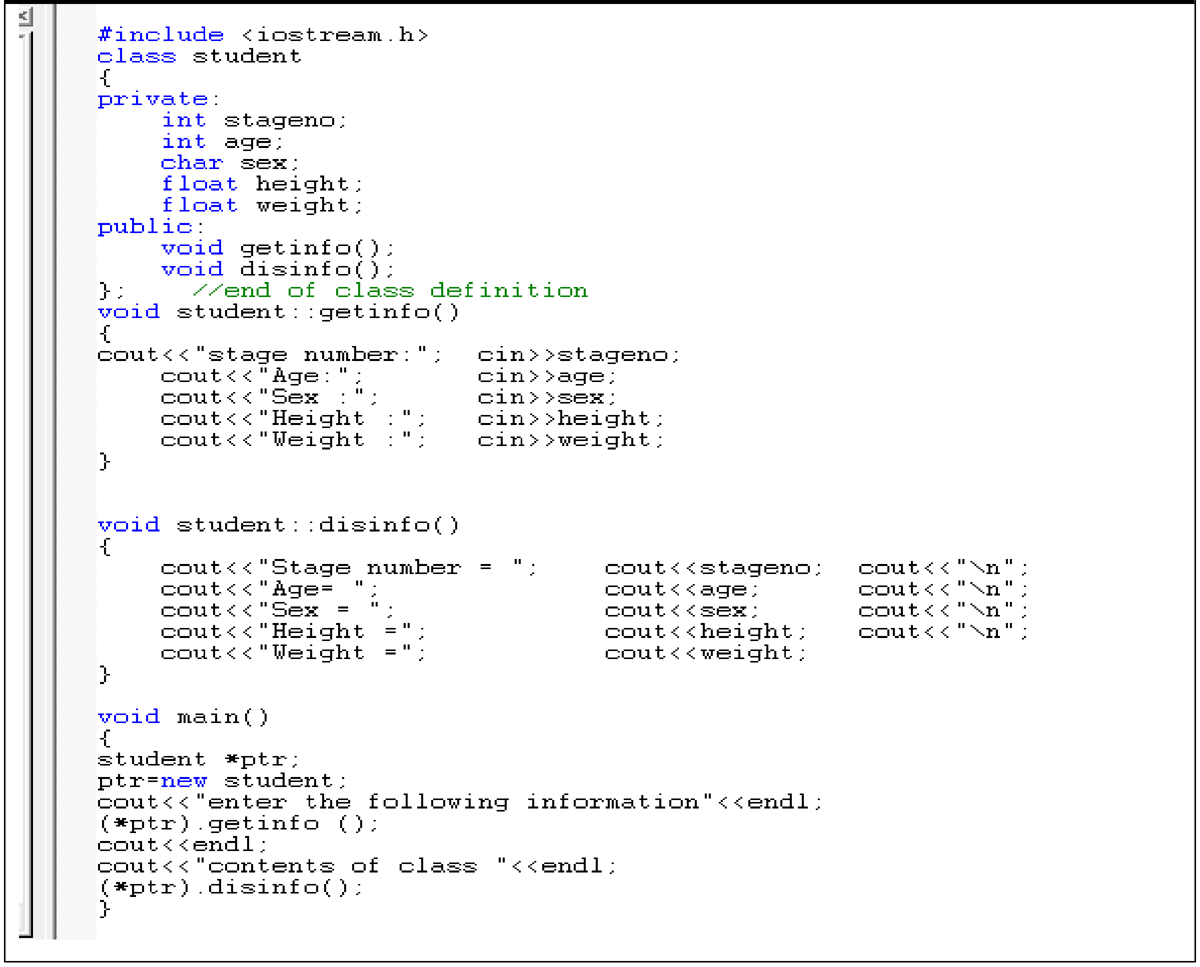
<u>First way :-</u> ***(*object name).member name=variable;***

The parentheses are essential since the member of class period(.) has a higher precedence over the indirection operator (*).

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

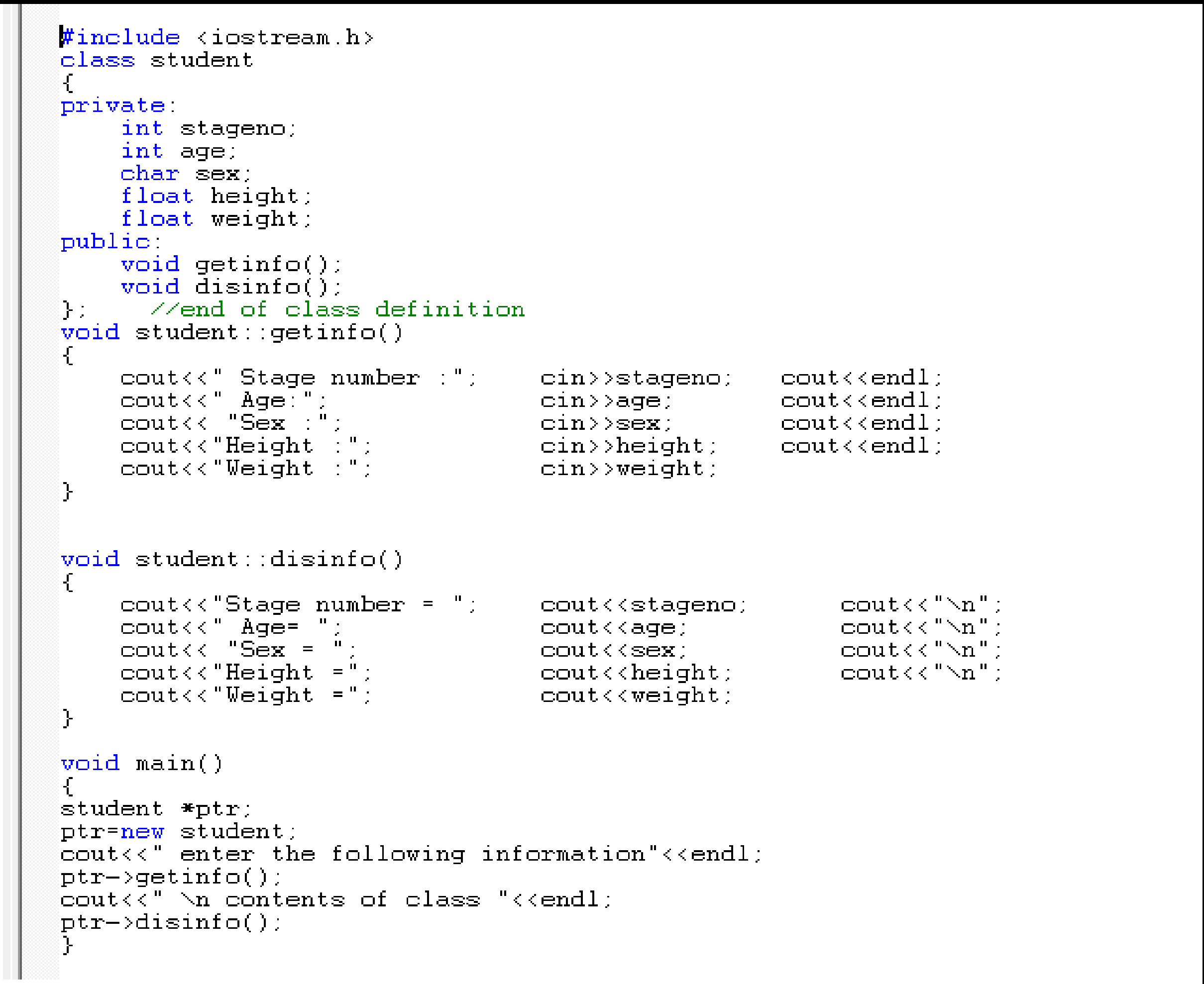
The pointer to the member of a class can be expressed using dash(-) followed by the greater than(>).

Example 1: Write a simple program using (*) to represent class 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();
}
```

Example 2: Write a simple program using (->) to represent class pointer.

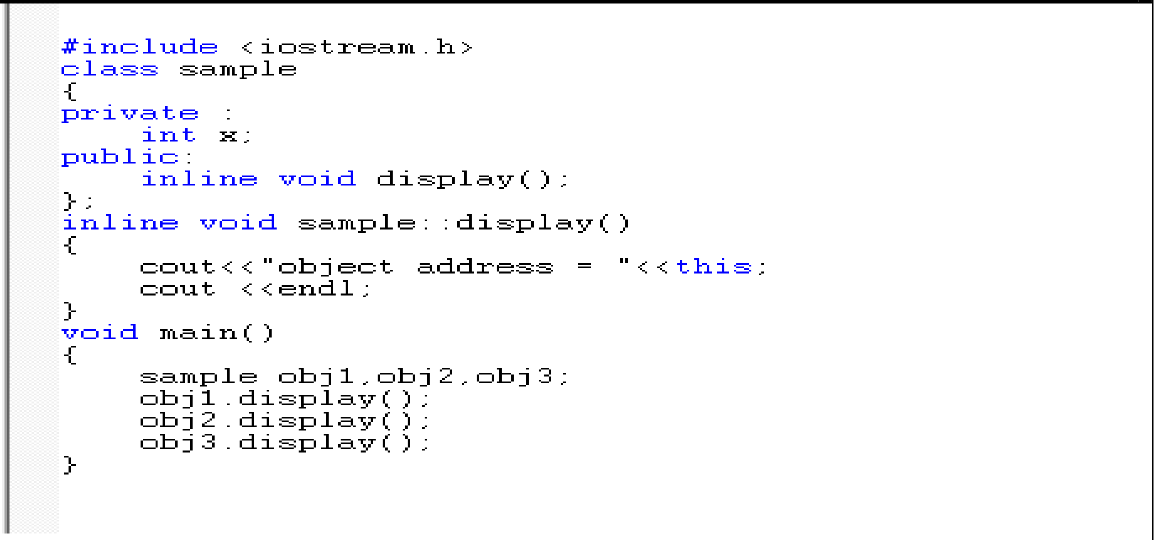


```
#include <iostream.h>
class student
{
private:
    int stageno;
    int age;
    char sex;
    float height;
    float weight;
public:
    void getinfo();
    void disinfo();
};    //end of class definition
void student::getinfo()
{
    cout<<" Stage number :";    cin>>stageno;    cout<<endl;
  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

<u>This pointer</u>

It is well known that a pointer is a variable which hold the memory address of another variable. Using the pointer technique, one can access the data of another variable indirectly. The *This* pointer is a variable which is used to access the address of the class itself.

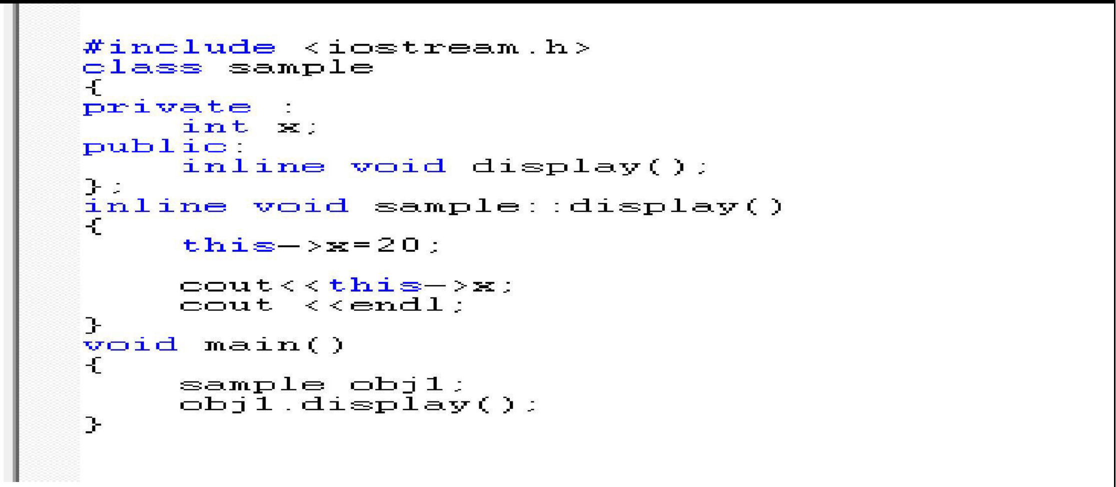
Example 1:Write an oo program to display the address of class using *this* pointer



```
#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. The display() member function is used to give the address of the object

Example 2:Write an oo program 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:

```
class  Image {
int   width;
int   height;
int   &widthRef;
//...
};
```

As with data member constants, a data member reference cannot be initialized using the same syntax as for other references:

```
class Image {
int  width;
int  height;
int  &widthRef = width; // illegal!
//...
};
```

The correct way to initialize a data member reference is through a member initialization list:

```
class Image {
public:
Image (const int w, const int h);
private:
int  width;
int  height;
int &widthRef;
```

//...

};

Image::Image (const int w, const int h) : widthRef(width)

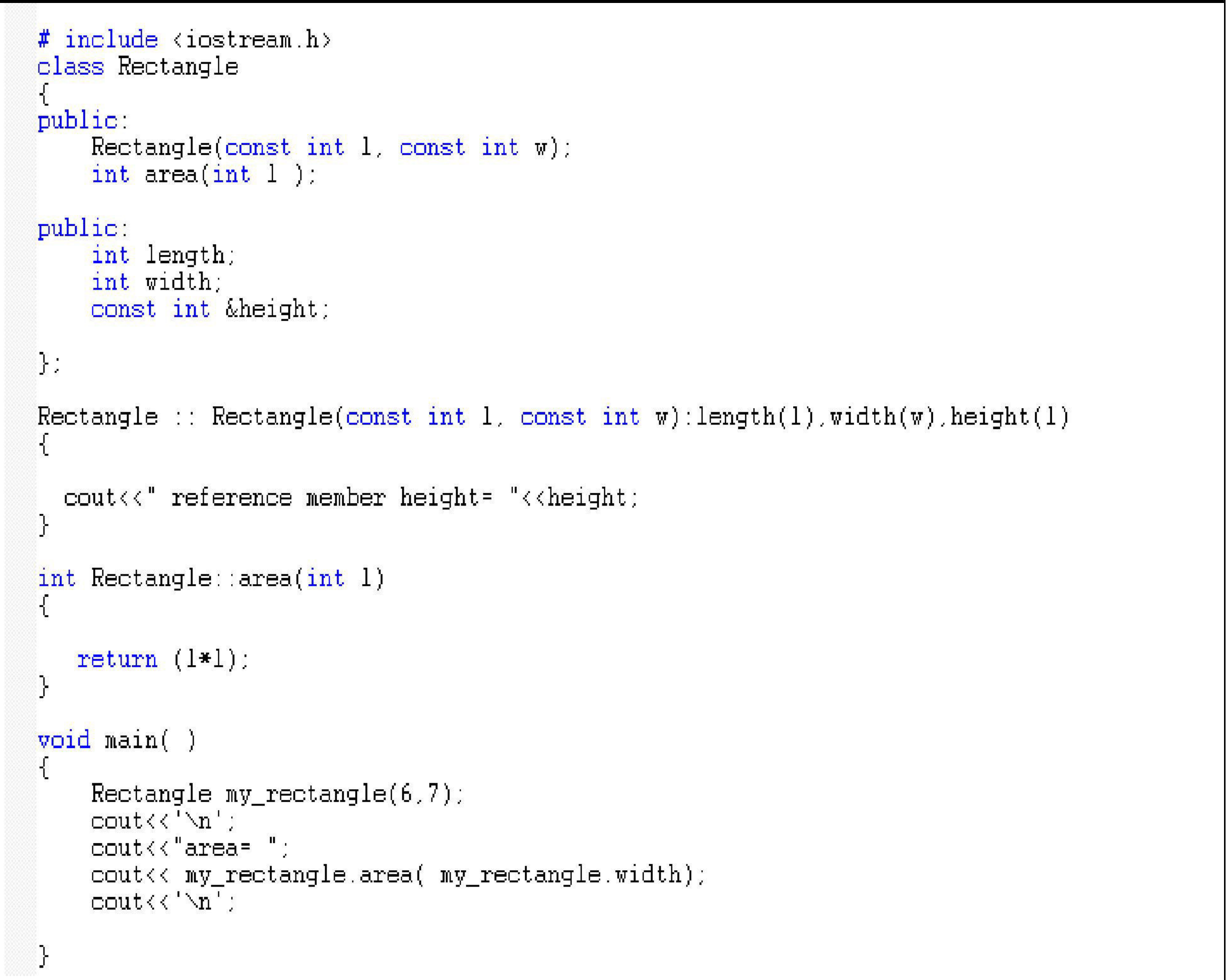
{

//...

}

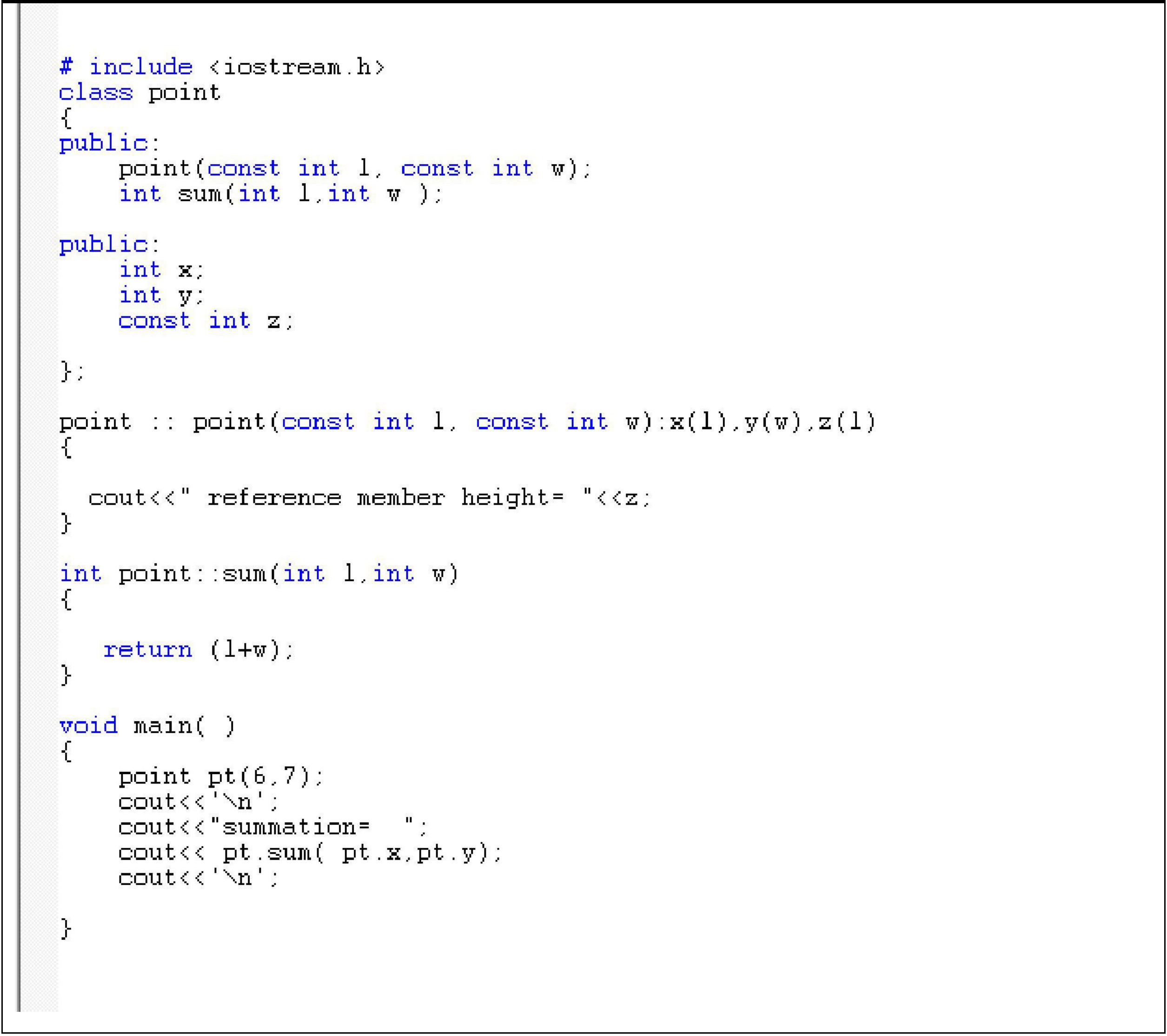
This causes widthRef to be a reference for width.

Example 1:Write an oo program to represent reference member of rectangle class



```
# include <iostream.h>
class Rectangle
{
public:
    Rectangle(const int l, const int w);
    int area(int l );

public:
    int length;
    int width;
    const int &height;

};

Rectangle :: Rectangle(const int l, const int w):length(l),width(w),height(l)
{

  cout<<" reference member height= "<<height;
}

int Rectangle::area(int l)
{

   return (l*l);
}

void main( )
{
    Rectangle my_rectangle(6,7);
    cout<<'\n';
    cout<<"area= ";
    cout<< my_rectangle.area( my_rectangle.width);
    cout<<'\n';

}
```

Example 2:Write an oo program to represent reference member of point class



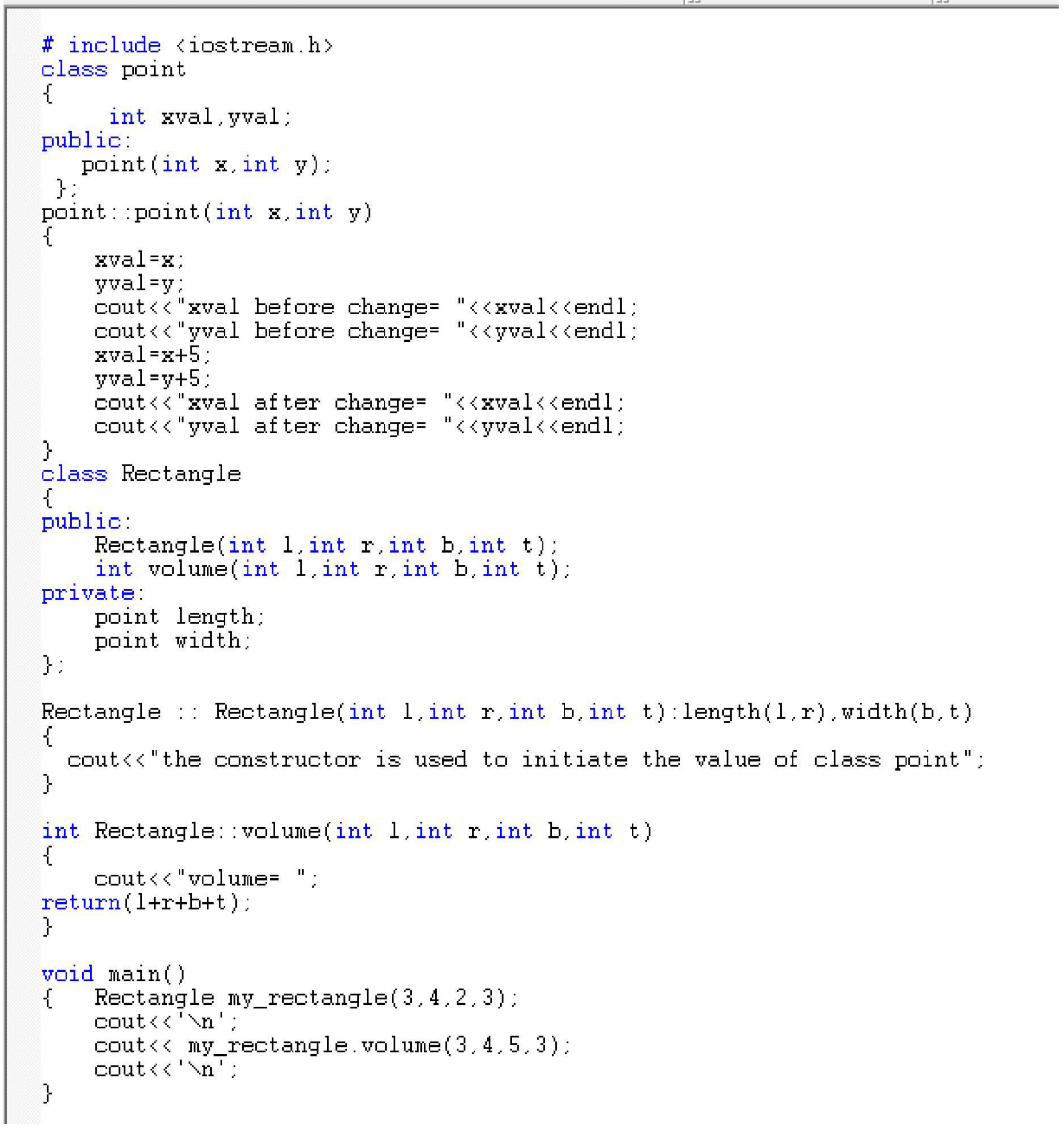
```
# include <iostream.h>
class point
{
public:
    point(const int l, const int w);
    int sum(int l,int w );

public:
    int x;
    int y;
    const int z;

};

point :: point(const int l, const int w):x(l),y(w),z(l)
{

  cout<<" reference member height= "<<z;
}

int point::sum(int l,int w)
{

   return (l+w);
}

void main( )
{
    point pt(6,7);
    cout<<'\n';
    cout<<"summation=  ";
    cout<< pt.sum( pt.x,pt.y);
    cout<<'\n';

}
```

<u>Class Object Member</u>

A data member of a class may be of a user-defined type, that is, an object of another class. For example, a Rectangle class may be defined using two

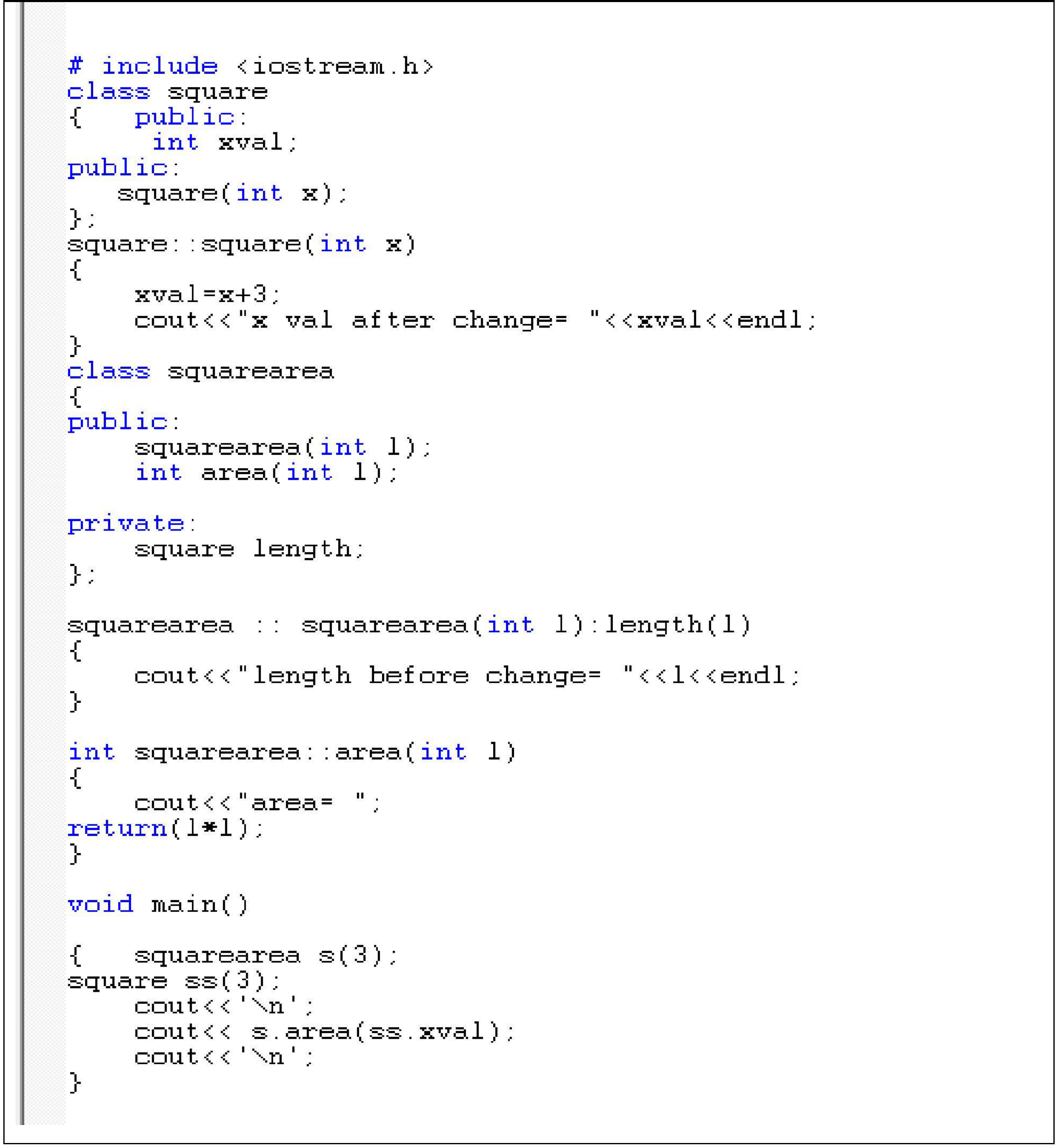
Point data members which represent the top and bottom-right corners of the rectangle:

Example 1: Write a simple program to represent class object member.



```
# include <iostream.h>
class point
{
     int xval,yval;
public:
   point(int x,int y);
 };
point::point(int x,int y)
{
    xval=x;
    yval=y;
    cout<<"xval before change= "<<xval<<endl;
    cout<<"yval before change= "<<yval<<endl;
    xval=x+5;
    yval=y+5;
    cout<<"xval after change= "<<xval<<endl;
    cout<<"yval after change= "<<yval<<endl;
}
class Rectangle
{
public:
    Rectangle(int l,int r,int b,int t);
    int volume(int l,int r,int b,int t);
private:
    point length;
    point width;
};

Rectangle :: Rectangle(int l,int r,int b,int t):length(l,r),width(b,t)
{
  cout<<"the constructor is used to initiate the value of class point";
}

int Rectangle::volume(int l,int r,int b,int t)
{
    cout<<"volume= ";
return(l+r+b+t);
}

void main()
{   Rectangle my_rectangle(3,4,2,3);
    cout<<'\n';
    cout<< my_rectangle.volume(3,4,5,3);
    cout<<'\n';
}
```

Example 2: Write a simple program to represent class object member.



```
# include <iostream.h>
class square
{   public:
     int xval;
public:
   square(int x);
};
square::square(int x)
{
    xval=x+3;
    cout<<"x val after change= "<<xval<<endl;
}
class squarearea
{
public:
    squarearea(int l);
    int area(int l);

private:
    square length;
};

squarearea :: squarearea(int l):length(l)
{
    cout<<"length before change= "<<l<<endl;
}

int squarearea::area(int l)
{
    cout<<"area= ";
return(l*l);
}

void main()

{   squarearea s(3);
square ss(3);
    cout<<'\n';
    cout<< s.area(ss.xval);
    cout<<'\n';
}
```

Arrays as Class Member Data

Defining Arrays

Like other variables in C++, an array must be defined before it can be used to store information.

And, like other definitions, an array definition 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.

It immediately follows the name, and is surrounded by square brackets.

Figure 1 shows the syntax of an array definition.

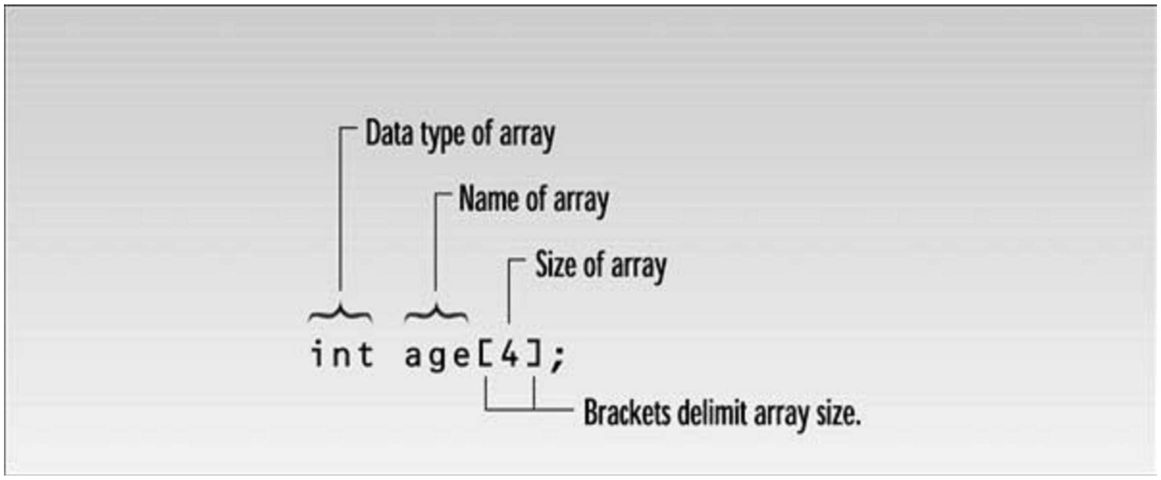


Figure1: syntax of array definition

The items in an array are called *elements* (in contrast to the items in a structure, which are called *members*). As we noted, all the elements in an array are of the same type; only the values vary. As specified in the definition, the array has exactly four elements.

Note:- the first array element is numbered 0. Thus, since there are four elements, the last one is number 3. This is a potentially confusing situation; you might think the last element in a four-element array would be number 4, but it’s not.

```
#include <iostream.h>
class arr
{
private:
enum { MAX = 3 }; //constant definition
int ar[MAX]; // array of integers
public:
void get() //put number on stack
{
    int x, i;
    for (i=0;i<=MAX;i++)
    { cin >> x;
    ar[i] = x;  };
}

      void show()
      {
           int  i;
           for (i=0;i<=MAX; i++)
           cout << ar[i];
      }
      }; // end class
      int main()
      {
      arr a1;
      a1.get();
      a1.show();
      return 0;
      }
```

Defining Multidimensional Arrays

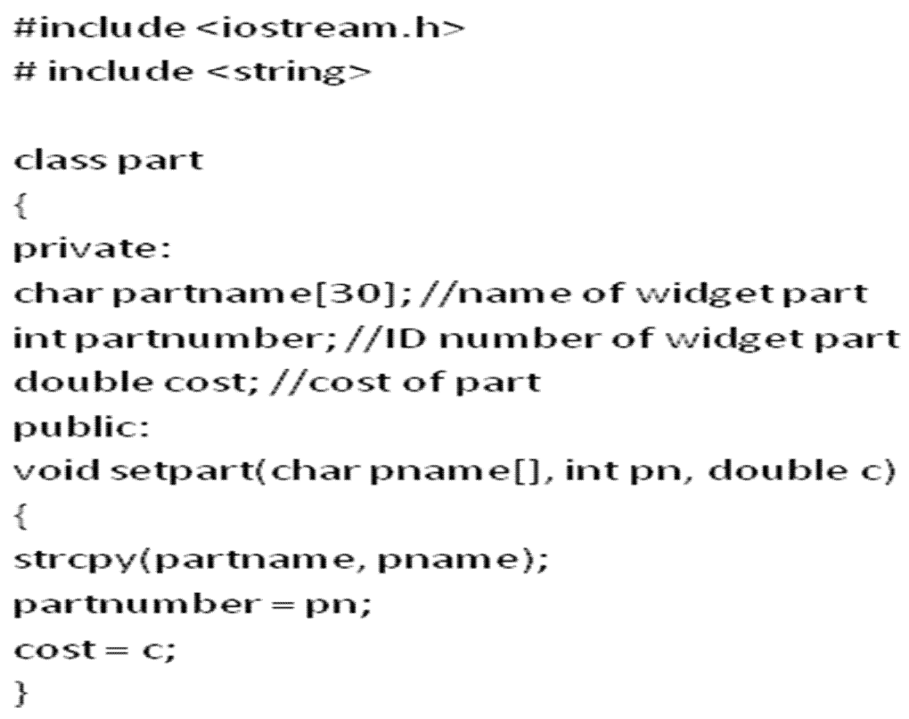
The array is defined with two size specifies, each enclosed in brackets:

```
int dimen2[5][3];
 elem = dimen2[x][y];
```

Of course there can be arrays of more than two dimensions. A three-dimensional array is an array of arrays of arrays. It is accessed with three indexes:

```
int dimen3[5][3][4];
elem = dimen3[x][y][z];
```

Strings as Class Members



```
#include <iostream.h>
# include <string>

class part
{
private:
char partname[30]; //name of widget part
int partnumber; //ID number of widget part
double cost; //cost of part
public:
void setpart(char pname[], int pn, double c)
{
strcpy(partname, pname);
partnumber = pn;
cost = c;
}
void showpart() //display data
{
cout << "\nName=" << partname;
cout << ", number=" << partnumber;
cout << ", cost=$" << cost;
}
};
int main()
{
part part1, part2;
part1.setpart("ABC", 4473, 217.55);
part2.setpart("XYZ", 9924, 419.25);
cout << "\nFirst part: "; part1.showpart();
cout << "\nSecond part: "; part2.showpart();
cout << endl;
     return 0;
}
```

Strings as Class Members – cont.

- This program defines two objects of class part and gives them values with the setpart() member function. Then it displays them with the showpart() member function. Here's the output:
- First part:
- Name= ABC, number=4473, cost=$217.55
- Second part:
- Name= XYZ, number=9924, cost=$419.25
- In the setpart() member function, we use the strcpy() string library function to copy the string from the argument pname to the class data member partname. Thus this function serves the same purpose with string variables that an assignment statement does with simple

variables. (A similar function, strncpy(), takes a third argument, which is the maximum number of characters it will copy. This can help prevent overrunning the array.)

- ☐ Besides those we've seen, there are library functions to add a string to another, compare strings, search for specific characters in strings, and perform many other actions.

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. For example, a pentagon can be defined as an array of 5 points:

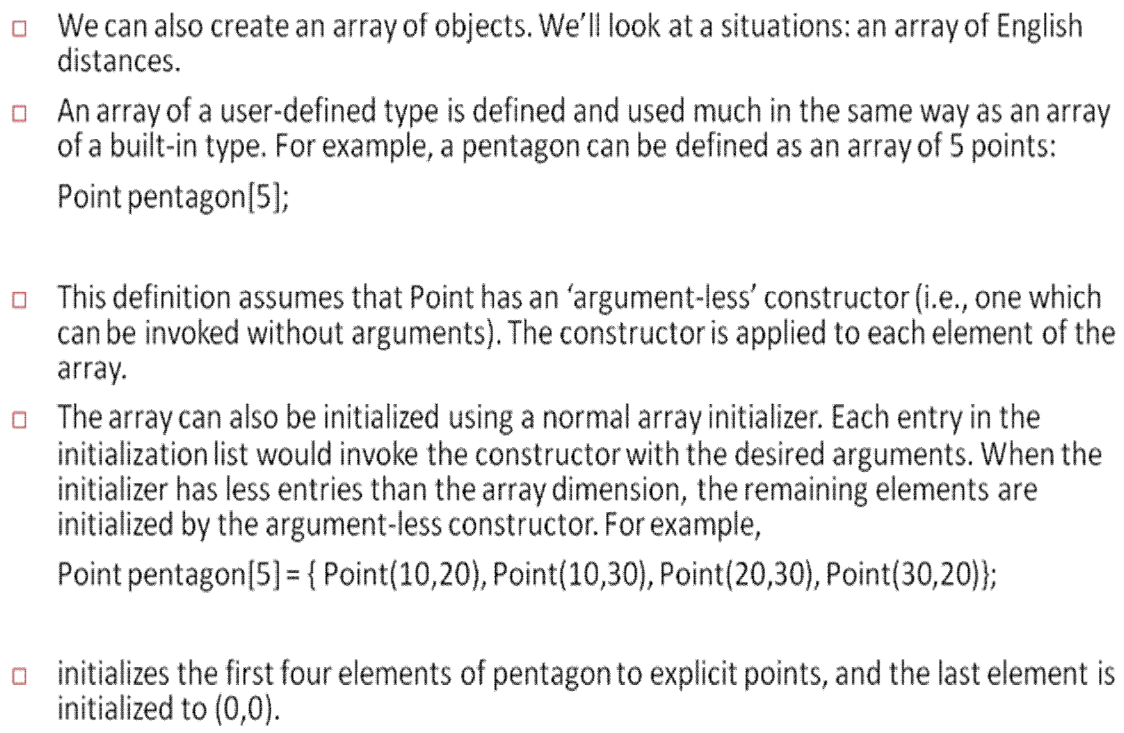
  Point pentagon[5];
- This definition assumes that Point has an 'argument-less' constructor (i.e., one which can be invoked without arguments). The constructor is applied to each element of the array.
- The array can also be initialized using a normal array initializer. Each entry in the initialization list would invoke the constructor with the desired arguments.

When the initializer has less entries than the array dimension, the remaining elements are initialized by the argument-less constructor. For example,

Point pentagon[5] = { Point(10,20), Point(10,30), Point(20,30), Point(30,20) };

- initializes the first four elements of pentagon to explicit points, and the last element is initialized to (0,0).

<u>Arrays of Objects</u>



- We can also create an array of objects. We'll look at a situations: an array of English distances.
- An array of a user-defined type is defined and used much in the same way as an array of a built-in type. For example, a pentagon can be defined as an array of 5 points:

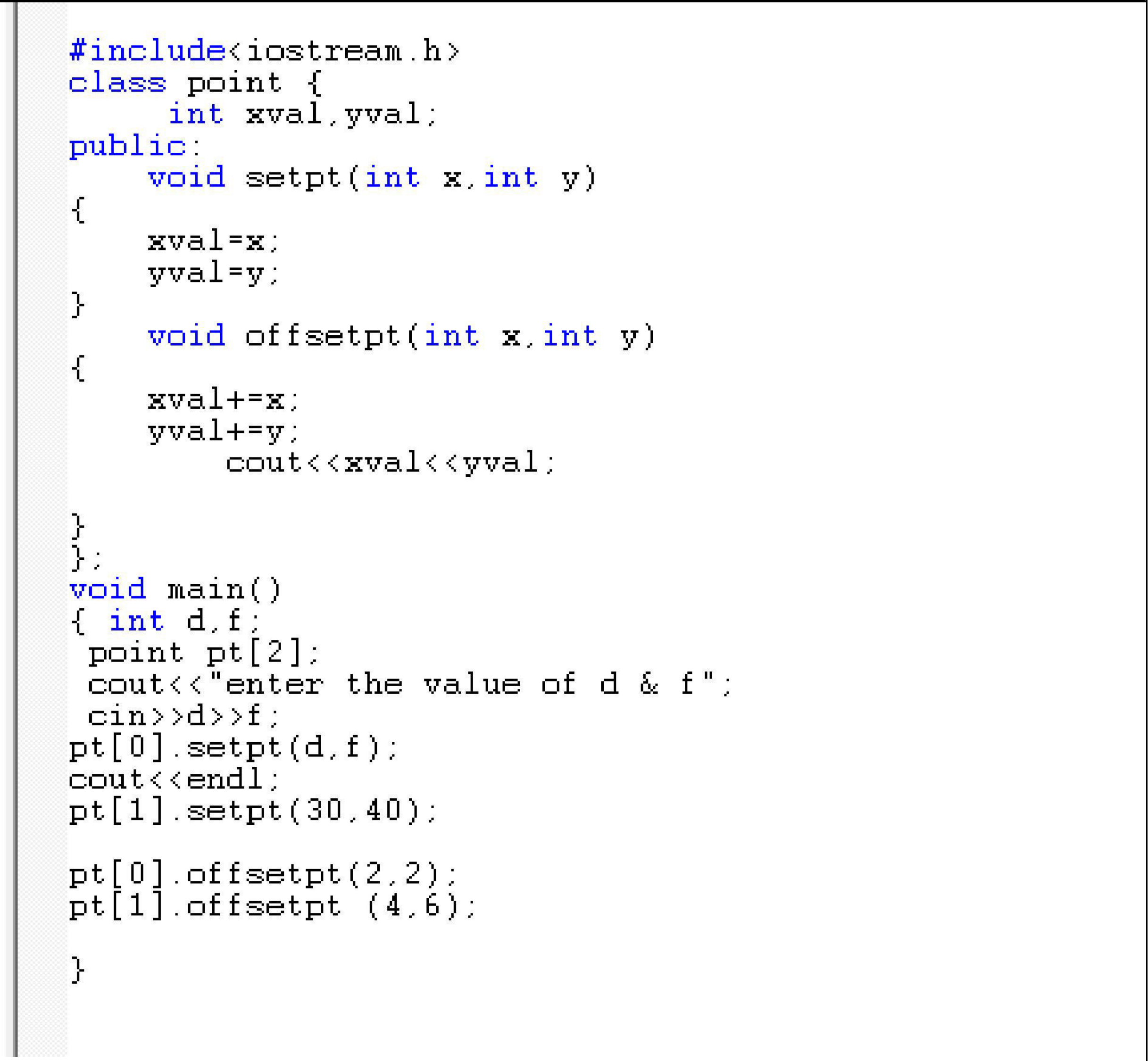
  Point pentagon[5];

- This definition assumes that Point has an 'argument-less' constructor (i.e., one which can be invoked without arguments). The constructor is applied to each element of the array.
- The array can also be initialized using a normal array initializer. Each entry in the initialization list would invoke the constructor with the desired arguments. When the initializer has less entries than the array dimension, the remaining elements are initialized by the argument-less constructor. For example,

  Point pentagon[5] = { Point(10,20), Point(10,30), Point(20,30), Point(30,20)};

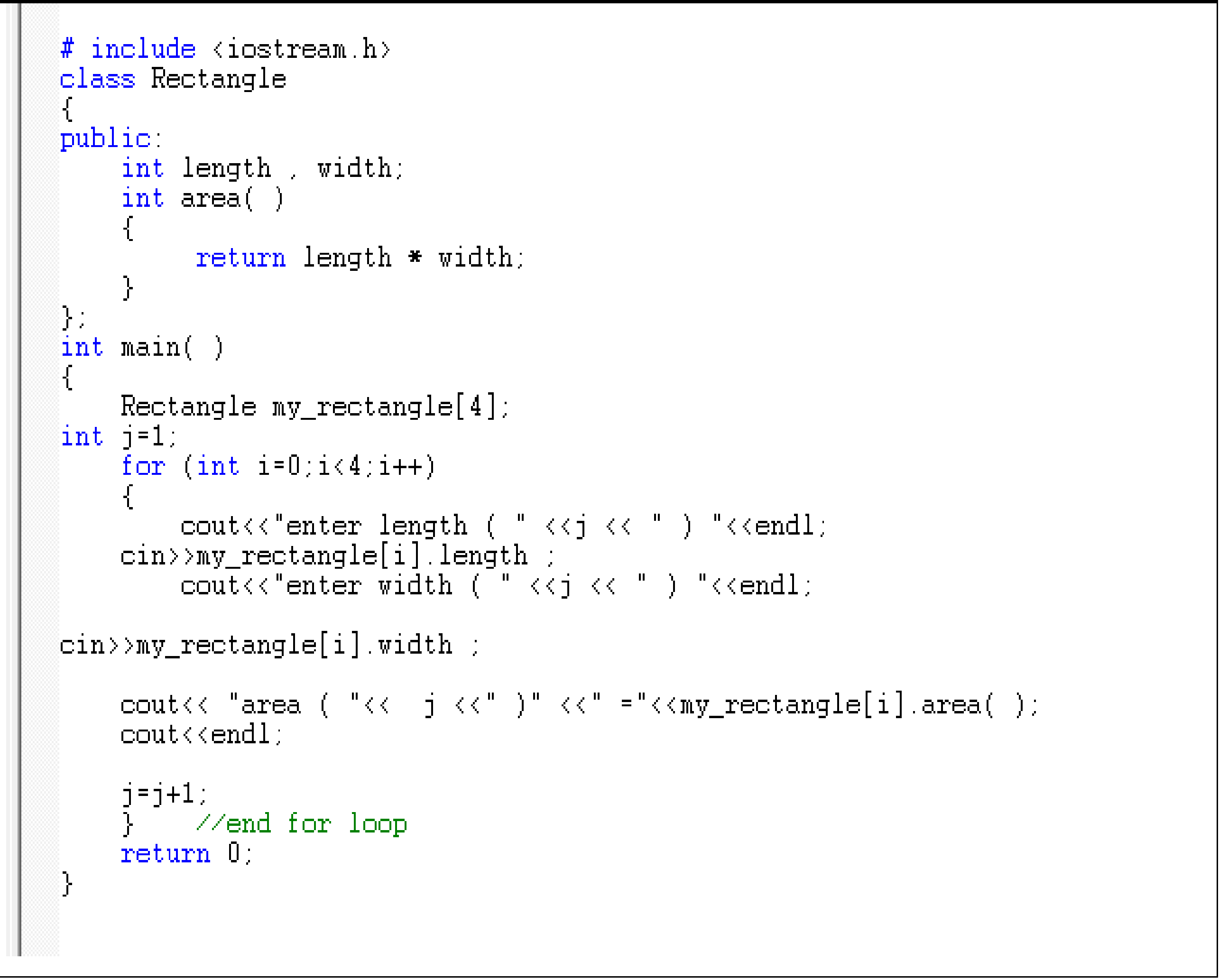
- initializes the first four elements of pentagon to explicit points, and the last element is initialized to (0,0).

Example 1: Write a simple program to represent array of class object 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()
{ int d,f;
 point pt[2];
 cout<<"enter the value of d & f";
 cin>>d>>f;
pt[0].setpt(d,f);
cout<<endl;
pt[1].setpt(30,40);

pt[0].offsetpt(2,2);
pt[1].offsetpt (4,6);

}
```

Example 2: Write a simple program to represent array of class object rectangle.



```
# include <iostream.h>
class Rectangle
{
public:
    int length , width;
    int area( )
    {
         return length * width;
    }
};
int main( )
{
    Rectangle my_rectangle[4];
int j=1;
    for (int i=0;i<4;i++)
    {
        cout<<"enter length ( " <<j << " ) "<<endl;
    cin>>my_rectangle[i].length ;
        cout<<"enter width ( " <<j << " ) "<<endl;

cin>>my_rectangle[i].width ;

    cout<< "area ( "<<  j <<" )" <<" ="<<my_rectangle[i].area( );
    cout<<endl;

    j=j+1;
    }    //end for loop
    return 0;
}
```

Example 3: Write a simple program to represent array of class object distance

```
#include <iostream.h>
////////////////////////////////////////////////////////////////
class Distance                       //Distance class
{
private:
int feet;
float inches;
public:
void getdist()                        //get length from user
{
cout << "\n Enter feet: "; cin >> feet;
cout << " Enter inches: "; cin >> inches;
}
void showdist() const                    //display distance
{
    cout<<feet<<" "<<inches<<endl;
 }
};
////////////////////////////////////////////////////////////////
int main()
{
Distance dist[100];                  //array of objects distances
int n=0;                             //count the entries
char ans;                            //user response ('y' or 'n')

cout << endl;
do {                                //get distances from user
cout << "Enter distance number " << n+1;
dist[n++].getdist();                 //store distance in array
cout << "Enter another (y/n)?: ";
cin >> ans;
} while( ans != 'n');               //quit if user types 'n'
for(int j=0; j<n; j++)              //display all distances
{
cout << "\nDistance number " << j+1 << " is ";
dist[j].showdist();
}
cout << endl;
return 0;
}
```

In this program the user types in as many distances as desired. After each distance is entered, the program asks if the user desires to enter another. If not, it terminates, and displays all the distances entered so far. Here's a sample interaction when the user enters three distances:

Output:-

```
Enter distance number 1
Enter feet: 5
Enter inches: 4
Enter another (y/n)? y
Enter distance number 2
Enter feet: 6
Enter inches: 2.5
Enter another (y/n)? y
Enter distance number 3
Enter feet: 5
Enter inches: 10.75
Enter another (y/n)? n
Distance number 1 is 5'-4"
Distance number 2 is 6'-2.5"
Distance number 3 is 5'-10.75"
```

Pointers to Objects

Pointers can point to objects as well as to simple data types and arrays. We've seen many examples of objects defined and given a name, in statements like

Distance dist;

where an object called dist is defined to be of the Distance class. Sometimes, however, we don’t know, at the time that we write the program, how many objects we want to create. When this is the case we can use new to create objects while the program is running. As we’ve seen, new returns a pointer to an unnamed object.

Example 4

```
#include <iostream.h>
class Distance          //English Distance class
{
private:
int feet;
float inches;
public:
void getdist()          //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist()         //display distance
{ cout << feet << "\'-" << inches << "\""; }
};
int main()
{
Distance dist;          //define a named Distance object
dist.getdist();           //access object members
dist.showdist();          // with dot operator
Distance* distptr;     //pointer to Distance
distptr = new Distance;   //points to new Distance object
distptr->getdist();         //access object members
distptr->showdist();      // with -> operator
cout << endl;
return 0;
}
```

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 5

```
#include <iostream.h>
class person            //class of persons
{
protected:
char name[40];       //person’s name
public:
void setName()       //set the name
{
cout << "Enter name: ";
cin >> name;
}
void printName() //get the name
{      cout << "\n Name is: " << name;   }
};
int main()
{
person* persPtr[100];    //array of pointers to persons
int n = 0;                         //number of persons in array
char choice;
do //put persons in array
{
persPtr[n] = new person;         //make new object
persPtr[n]->setName();           //set person's name
n++; //count new person
cout << "Enter another (y/n)? "; //enter another
cin >> choice; //person?
}
while( choice=='y' );             //quit on 'n'
for(int j=0; j<n; j++)           //print names of
{                  //all persons
cout << "\nPerson number " << j+1;
persPtr[j]->printName();
}
cout << endl;
return 0;
  } //end main()
```

Operator overloading

Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure program listings into intuitively obvious ones. For example, statements like

d3.addobjects(d1, d2);

or the similar but equally obscure

d3 = d1.addobjects(d2);

can be changed to the much more readable

d3 = d1 + d2;

The rather forbidding term *operator overloading* refers to giving the normal C++ operators, such as +, *, <=, and +=, additional meanings when they are applied to user-defined data types.

Normally

a = b + c;

works only with basic types such as int and float, and attempting to apply it when a, b, and c are objects of a user-defined class will cause complaints from the compiler. However, using overloading, you can make this statement legal even when a, b, and c are user-defined types.

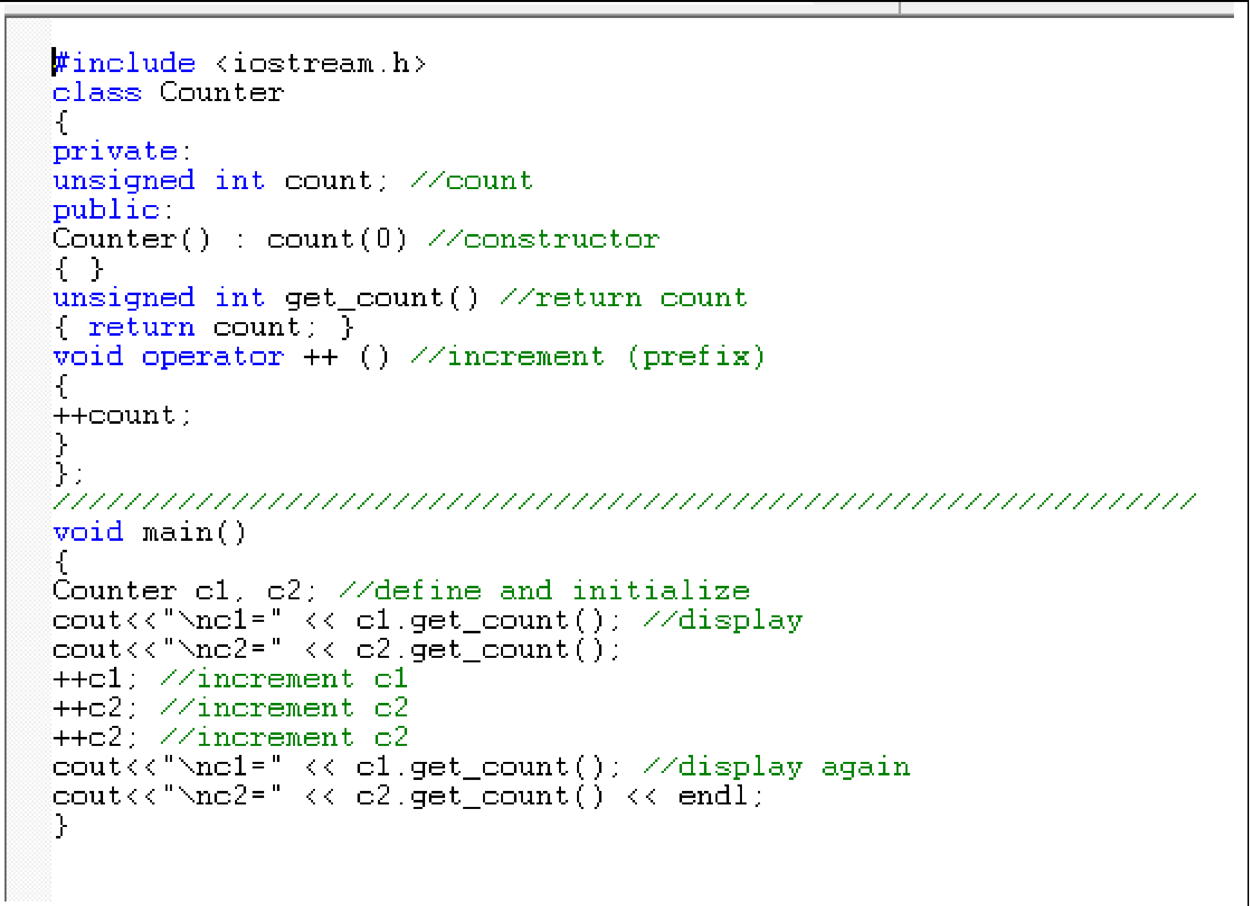
In effect, operator overloading gives you the opportunity to redefine the C++ language. If you find yourself limited by the way the C++ operators work, you can change them to do whatever you want.

By using classes to create new kinds of variables, and operator overloading to create new definitions for operators, you can extend C++ to be, in many ways, a new language of your own design.

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 oop program to increment the counter variable with ++ operator.



```
#include <iostream.h>
class Counter
{
private:
unsigned int count; //count
public:
Counter() : count(0) //constructor
{ }
unsigned int get_count() //return count
{ return count; }
void operator ++ () //increment (prefix)
{
++count;
}
};
/////////////////////////////////////////////////////////////////
void main()
{
Counter c1, c2; //define and initialize
cout<<"\nc1=" << c1.get_count(); //display
cout<<"\nc2=" << c2.get_count();
++c1; //increment c1
++c2; //increment c2
++c2; //increment c2
cout<<"\nc1=" << c1.get_count(); //display again
cout<<"\nc2=" << c2.get_count() << endl;
}
```

In this program we create two objects of class Counter: c1 and c2. The counts in the objects are displayed; they are initially 0. Then, using the overloaded ++ operator, we increment c1 once and c2 twice, and display the resulting values.

Here’s the program’s output:

```
c1=0 ←———    counts are initially 0
c2=0 ←———
c1=1 ←———     incremented once
c2=2 ←———     incremented twice
```

The statements responsible for these operations are

```
++c1;
++c2;
++c2;
```

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

The operator Keyword

How do we teach a normal C++ operator to act on a user-defined operand? The keyword operator is used to overload the ++ operator in this declarator:

void operator ++ ()

The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here). This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand (the variable operated on by the ++) is of type Counter. The compiler can distinguish between overloaded functions is by looking at the data types and the number of their arguments. In the same way, the only way it can distinguish between overloaded operators is by looking at the data type of their operands. If the operand is a basic type such as an int, as in **++intvar;** then the compiler will use its built-in routine to increment an int. But if the operand is a counter variable, the compiler will know to use our user-written operator++() instead.

Operator Arguments

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.

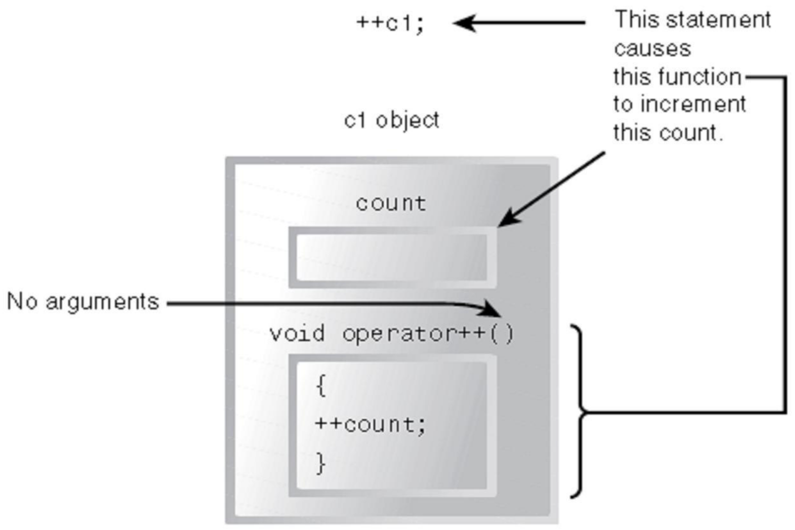

++c1;
This statement causes this function to increment this count.
c1 object
count
No arguments
void operator++()
{
++count;
}

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

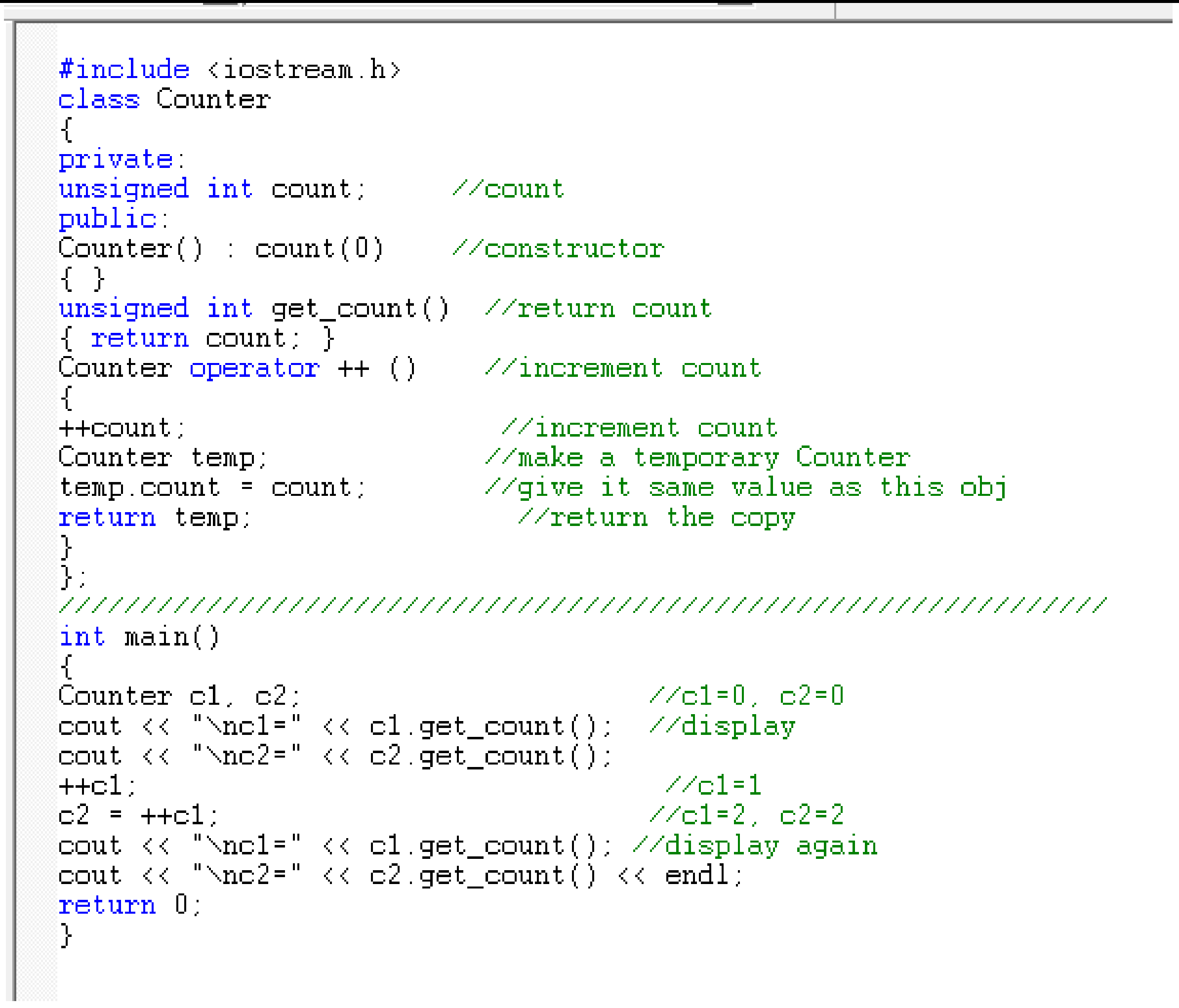
Operator Return Values

The operator++ () function in the Example1 has a subtle defect. You will discover it if you use a statement like this in main ():

c1 = ++c2;

The compiler will complain. Why? Because we have defined the ++ operator to have a return type of void in the operator++ () function, while in the assignment statement it is being asked to return a variable of type Counter. That is, the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and assign this value to c1.

Example 2:-Write an oop program increment the counter variable with ++ operator and return value.



```
#include <iostream.h>
class Counter
{
private:
unsigned int count;      //count
public:
Counter() : count(0)     //constructor
{ }
unsigned int get_count()  //return count
{ return count; }
Counter operator ++ ()    //increment count
{
++count;                   //increment count
Counter temp;             //make a temporary Counter
temp.count = count;       //give it same value as this obj
return temp;                //return the copy
}
};
/////////////////////////////////////////////////////////////////
int main()
{
Counter c1, c2;                       //c1=0, c2=0
cout << "\nc1=" << c1.get_count();  //display
cout << "\nc2=" << c2.get_count();
++c1;                                  //c1=1
c2 = ++c1;                            //c1=2, c2=2
cout << "\nc1=" << c1.get_count(); //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;
}
```

Here the operator++() function creates a new object of type Counter, called temp, to use as a return value. It increments the count data in its own object as before, then creates the new temp object and assigns count in the new object the same value as in its own object. Finally, it returns the temp object. This has the desired effect. Expressions like ,**++c1** now return a value, so they can be used in other expressions, such as,**c2 = ++c1;** as shown in main(), where the value returned from c1++ is assigned to c2.

The output from this program is

```
c1=0
c2=0
c1=2
c2=2
```

Nameless Temporary Objects

In Example 2 we created a temporary object of type Counter, named temp, whose sole purpose was to provide a return value for the ++ operator. This required three statements.

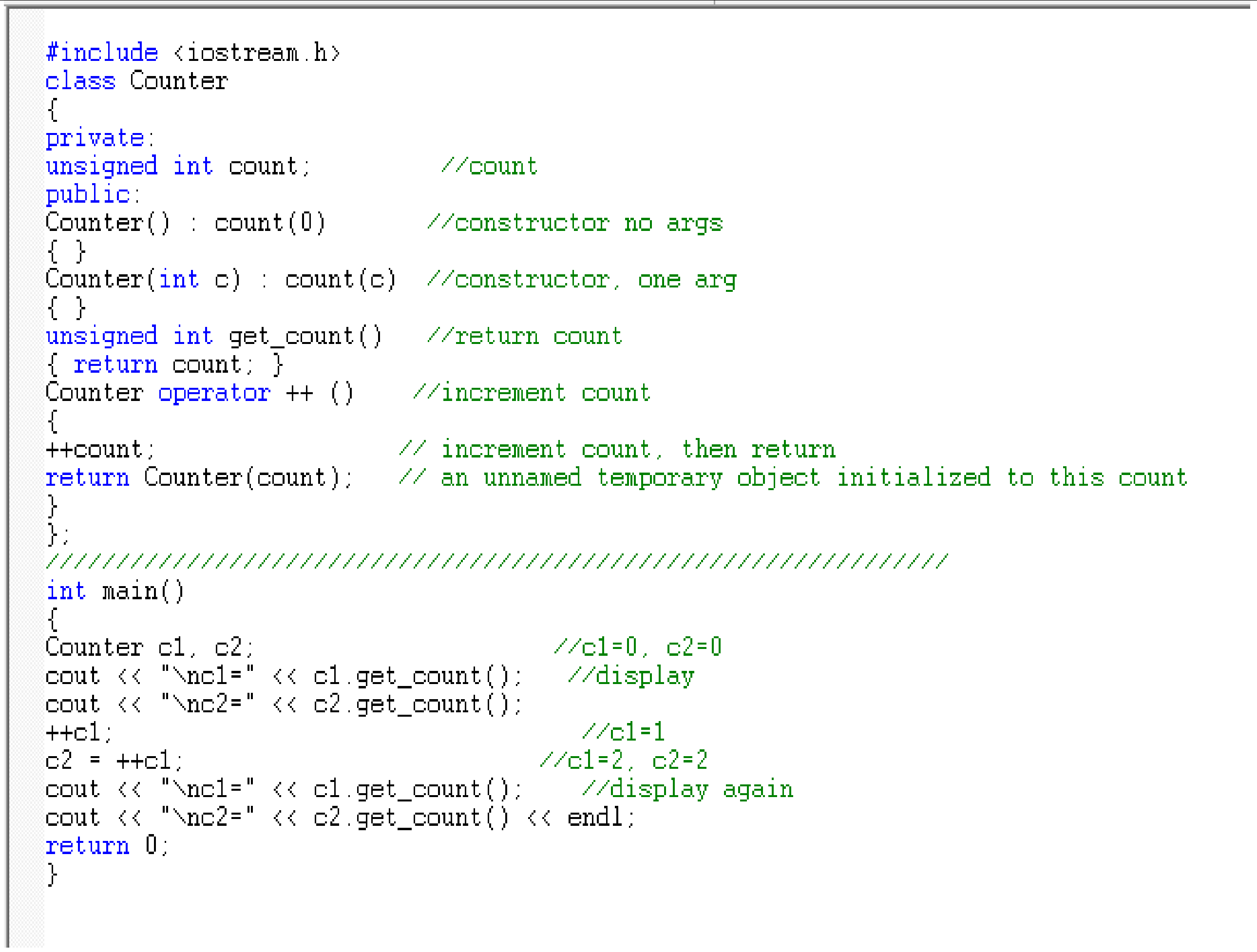
Counter temp; // make a temporary Counter object

temp.count = count; // give it same value as this object

return temp; // return it

There are more convenient ways to return temporary objects from functions and overloaded operators. Let's examine another approach, as shown in the example3

Example 3:-Write an oop program increment the counter variable with ++ operator and unnamed temporary object.



```
#include <iostream.h>
class Counter
{
private:
unsigned int count;          //count
public:
Counter() : count(0)        //constructor no args
{ }
Counter(int c) : count(c)   //constructor, one arg
{ }
unsigned int get_count()    //return count
{ return count; }
Counter operator ++ ()     //increment count
{
++count;                   // increment count, then return
return Counter(count);     // an unnamed temporary object initialized to this count
}
};
////////////////////////////////////////////////////////////////
int main()
{
Counter c1, c2;                        //c1=0, c2=0
cout << "\nc1=" << c1.get_count();     //display
cout << "\nc2=" << c2.get_count();
++c1;                                    //c1=1
c2 = ++c1;                             //c1=2, c2=2
cout << "\nc1=" << c1.get_count();      //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;
}
```

In this program a single statement

return Counter(count);

This statement creates an object of type Counter.

Counter(int c) : count(c) //constructor, one arg

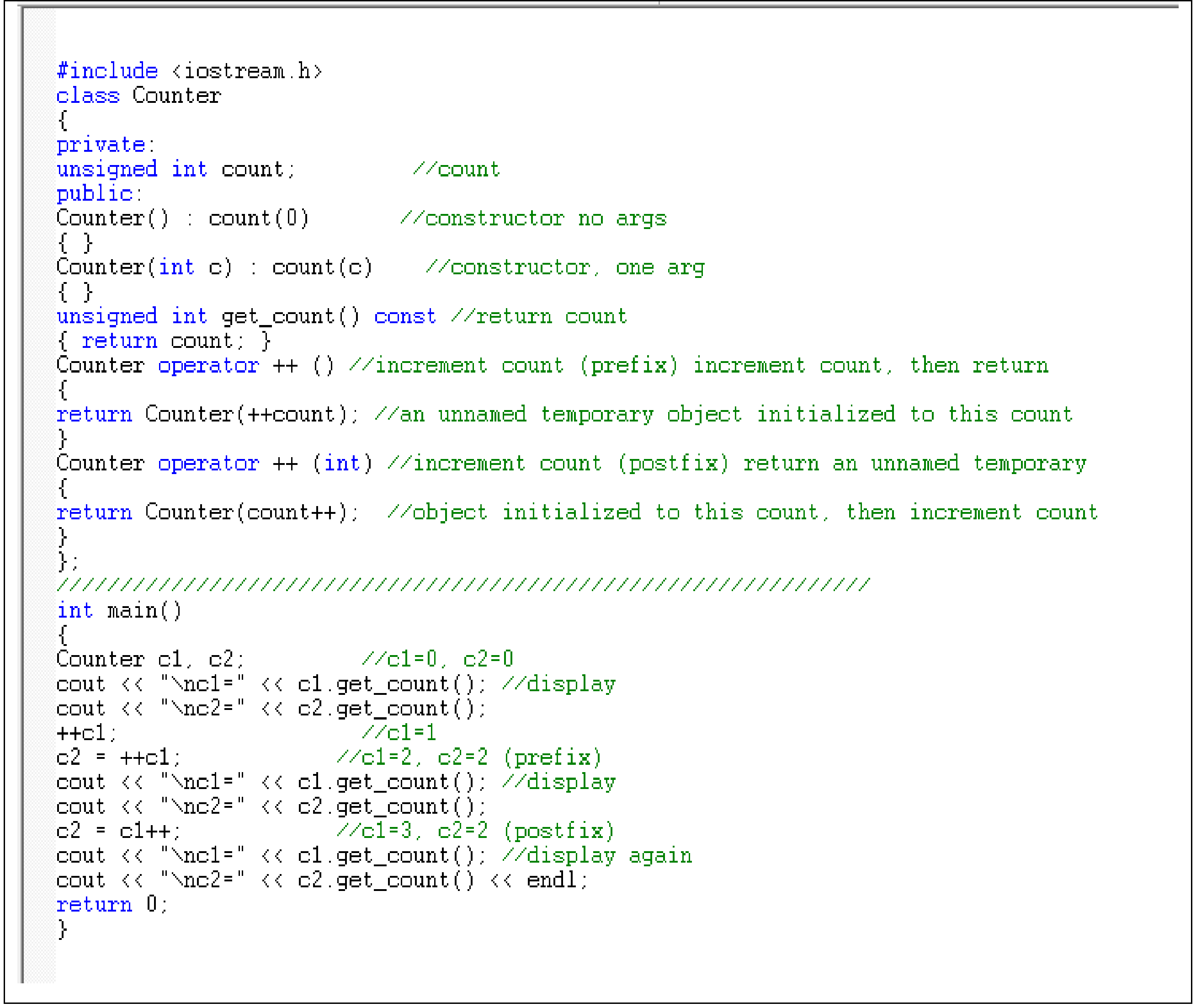
{ }

Once the unnamed object is initialized to the value of count, it can then be returned. The output of this program is the same as that of Example 2.

Postfix Notation

We’ve shown the increment operator used only in its prefix form **++c1**. What about postfix, where the variable is incremented after its value is used in the expression? **c1++** to make both versions of the increment operator work, we define two overloaded ++ operators, as shown in the Example4:

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



```
#include <iostream.h>
class Counter
{
private:
unsigned int count;          //count
public:
Counter() : count(0)        //constructor no args
{ }
Counter(int c) : count(c)    //constructor, one arg
{ }
unsigned int get_count() const //return count
{ return count; }
Counter operator ++ () //increment count (prefix) increment count, then return
{
return Counter(++count); //an unnamed temporary object initialized to this count
}
Counter operator ++ (int) //increment count (postfix) return an unnamed temporary
{
return Counter(count++);  //object initialized to this count, then increment count
}
};
////////////////////////////////////////////////////////////////
int main()
{
Counter c1, c2;          //c1=0, c2=0
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count();
++c1;                    //c1=1
c2 = ++c1;             //c1=2, c2=2 (prefix)
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count();
c2 = c1++;             //c1=3, c2=2 (postfix)
cout << "\nc1=" << c1.get_count(); //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;
}
```

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.

Here's the output from the program:

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

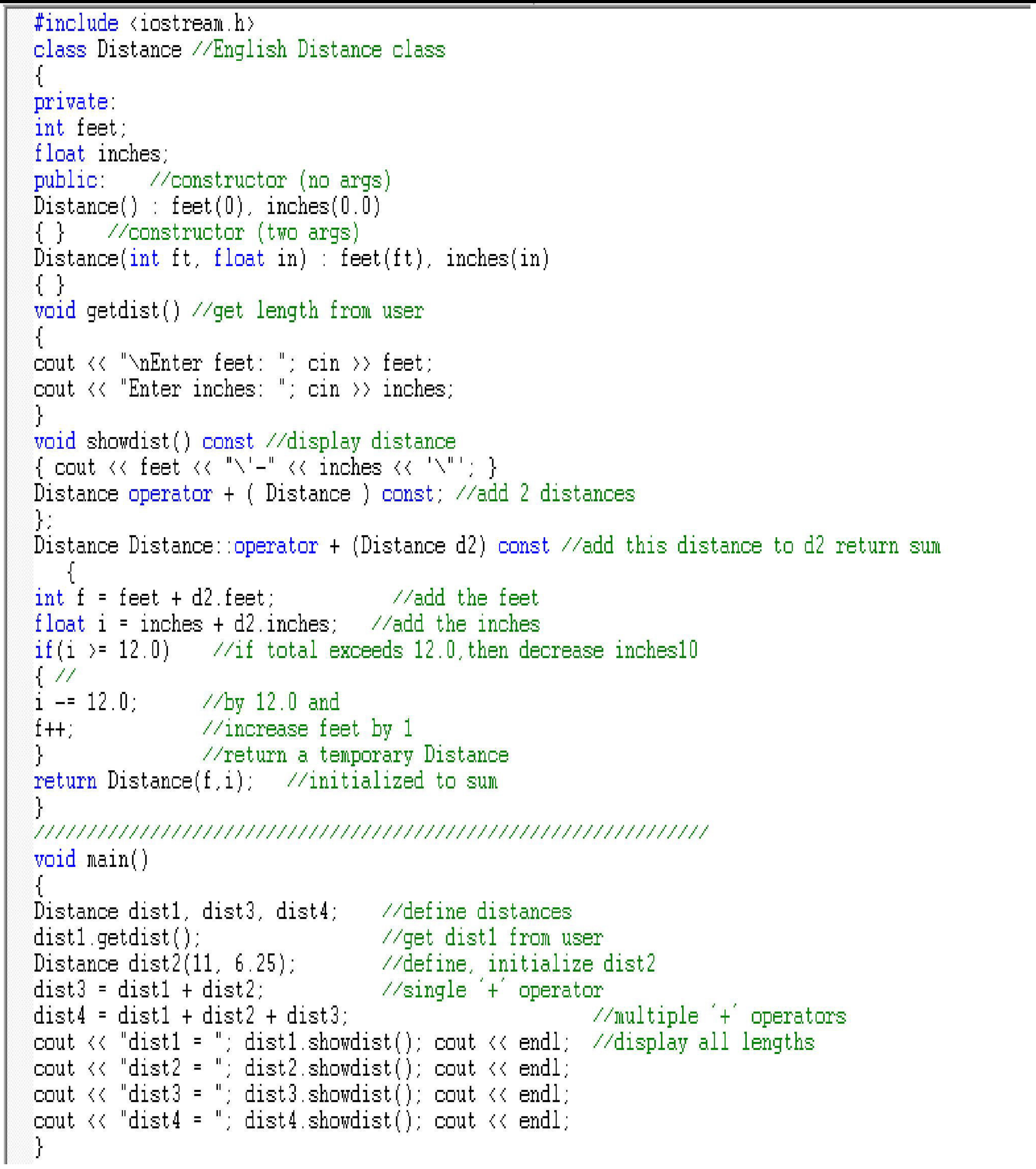
Overloading Binary Operators

Binary operators can be overloaded just as easily as unary operators. We'll look at examples that overload arithmetic operators, comparison operators, and arithmetic assignment operators.

Arithmetic Operators

Example 5 shows how add two distances **dist3.add_dist(dist1, dist2);** By overloading the + operator we can reduce this dense-looking expression to **dist3 = dist1 + dist2;**

Example 5:-Write an oop program to add two distances objects.

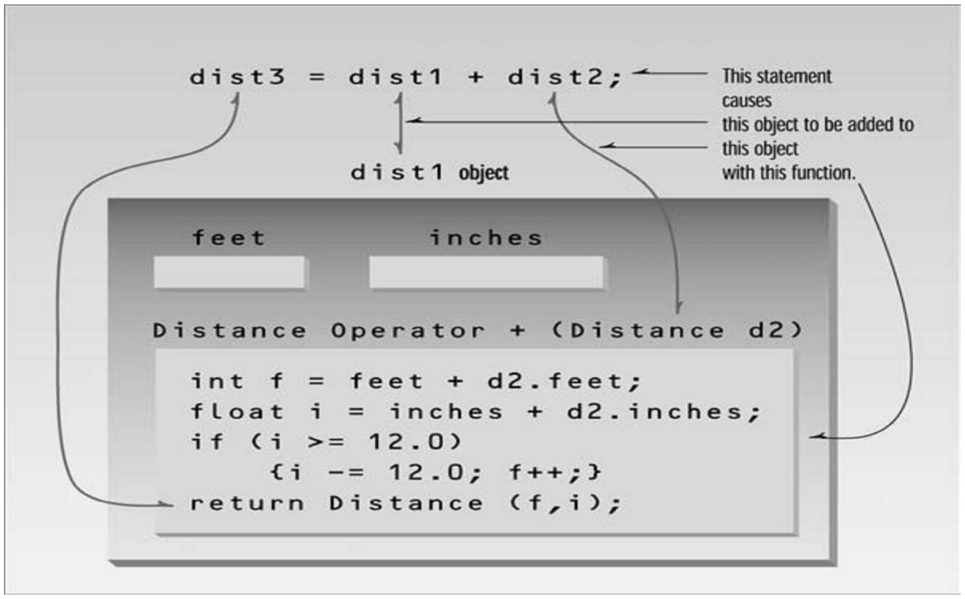


```
#include <iostream.h>
class Distance //English Distance class
{
private:
int feet;
float inches;
public:    //constructor (no args)
Distance() : feet(0), inches(0.0)
{ }    //constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "\'-" << inches << '\"'; }
Distance operator + ( Distance ) const; //add 2 distances
};
Distance Distance::operator + (Distance d2) const //add this distance to d2 return sum
   {
int f = feet + d2.feet;           //add the feet
float i = inches + d2.inches;   //add the inches
if(i >= 12.0)    //if total exceeds 12.0,then decrease inches10
{ //
i -= 12.0;      //by 12.0 and
f++;            //increase feet by 1
}               //return a temporary Distance
return Distance(f,i);   //initialized to sum
}
////////////////////////////////////////////////////////////////
void main()
{
Distance dist1, dist3, dist4;    //define distances
dist1.getdist();                 //get dist1 from user
Distance dist2(11, 6.25);        //define, initialize dist2
dist3 = dist1 + dist2;           //single '+' operator
dist4 = dist1 + dist2 + dist3;                       //multiple '+' operators
cout << "dist1 = "; dist1.showdist(); cout << endl;  //display all lengths
cout << "dist2 = "; dist2.showdist(); cout << endl;
cout << "dist3 = "; dist3.showdist(); cout << endl;
cout << "dist4 = "; dist4.showdist(); cout << endl;
}
```

Here’s the output from the program:

```
Enter feet: 10
Enter inches: 6.5
dist1 = 10’-6.5” ←   from user
dist2 = 11’-6.25” ← initialized in program
dist3 = 22’-0.75” ← dist1+dist2
dist4 = 44’-1.5” ←   dist1+dist2+dist3
```

When the compiler sees this expression it looks at the argument types, and finding only type Distance, it realizes it must use the Distance member function operator+(). The argument on the *left side* of the operator (dist1 in this case) is the object of which the operator is a member. The object on the *right side* of the operator (dist2) must be furnished as an argument to the operator. The operator returns a value, which can be assigned or used in other ways; in this case it is assigned to dist3.



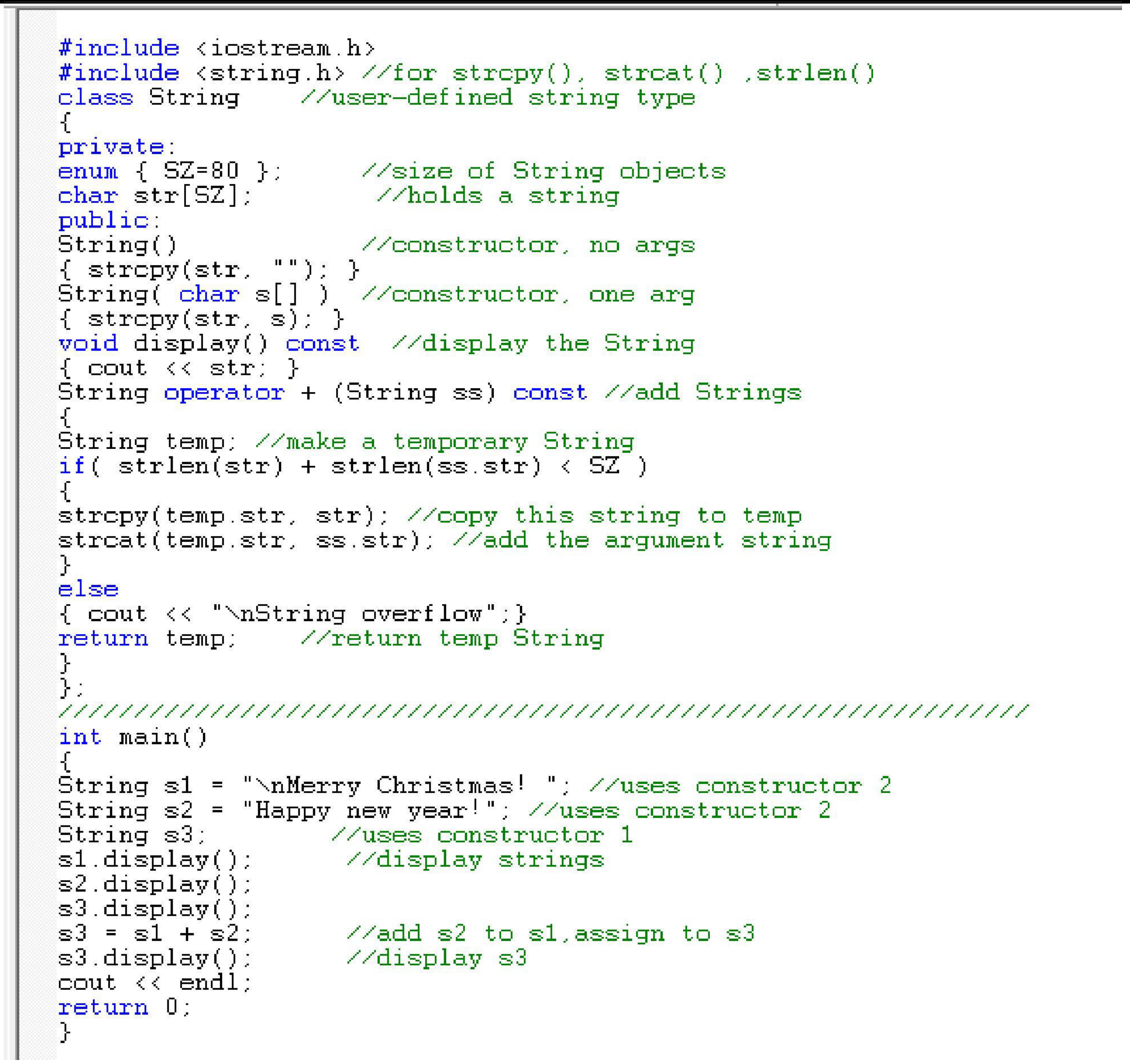
```
dist3 = dist1 + dist2;

Distance Operator + (Distance d2)
  int f = feet + d2.feet;
  float i = inches + d2.inches;
  if (i >= 12.0)
     {i -= 12.0; f++;}
  return Distance (f,i);
```

Figure 2: ***Overloaded binary operator: one argument.***

Concatenating Strings

We can overload the + operator to perform such concatenation. Here’s the listing for Example 6:

Example 6:-Write an oop program to concatenate two strings.



```
#include <iostream.h>
#include <string.h> //for strcpy(), strcat() ,strlen()
class String    //user-defined string type
{
private:
enum { SZ=80 };      //size of String objects
char str[SZ];         //holds a string
public:
String()             //constructor, no args
{ strcpy(str, ""); }
String( char s[] )   //constructor, one arg
{ strcpy(str, s); }
void display() const  //display the String
{ cout << str; }
String operator + (String ss) const //add Strings
{
String temp; //make a temporary String
if( strlen(str) + strlen(ss.str) < SZ )
{
strcpy(temp.str, str); //copy this string to temp
strcat(temp.str, ss.str); //add the argument string
}
else
{ cout << "\nString overflow";}
return temp;    //return temp String
}
};
/////////////////////////////////////////////////////////////////
int main()
{
String s1 = "\nMerry Christmas! "; //uses constructor 2
String s2 = "Happy new year!"; //uses constructor 2
String s3;          //uses constructor 1
s1.display();        //display strings
s2.display();
s3.display();
s3 = s1 + s2;        //add s2 to s1,assign to s3
s3.display();        //display s3
cout << endl;
return 0;
}
```

The program first displays three strings separately. (The third is empty at this point, so nothing is printed when it displays itself.) Then the first two strings are concatenated and placed in the third, and the third string is displayed again.

Here's the output:

```
Merry Christmas!  Happy new year! ◄──── s1, s2, and s3 (empty)
Merry Christmas! Happy new year! ◄────  s3 after concatenation
```

Multiple Overloading

We've seen different uses of the + operator: to add English distances and to concatenate strings.

You could put both these classes together in the same program, and C++ would still know how to interpret the + operator: It selects the correct function to carry out the "addition" based on the type of operand.

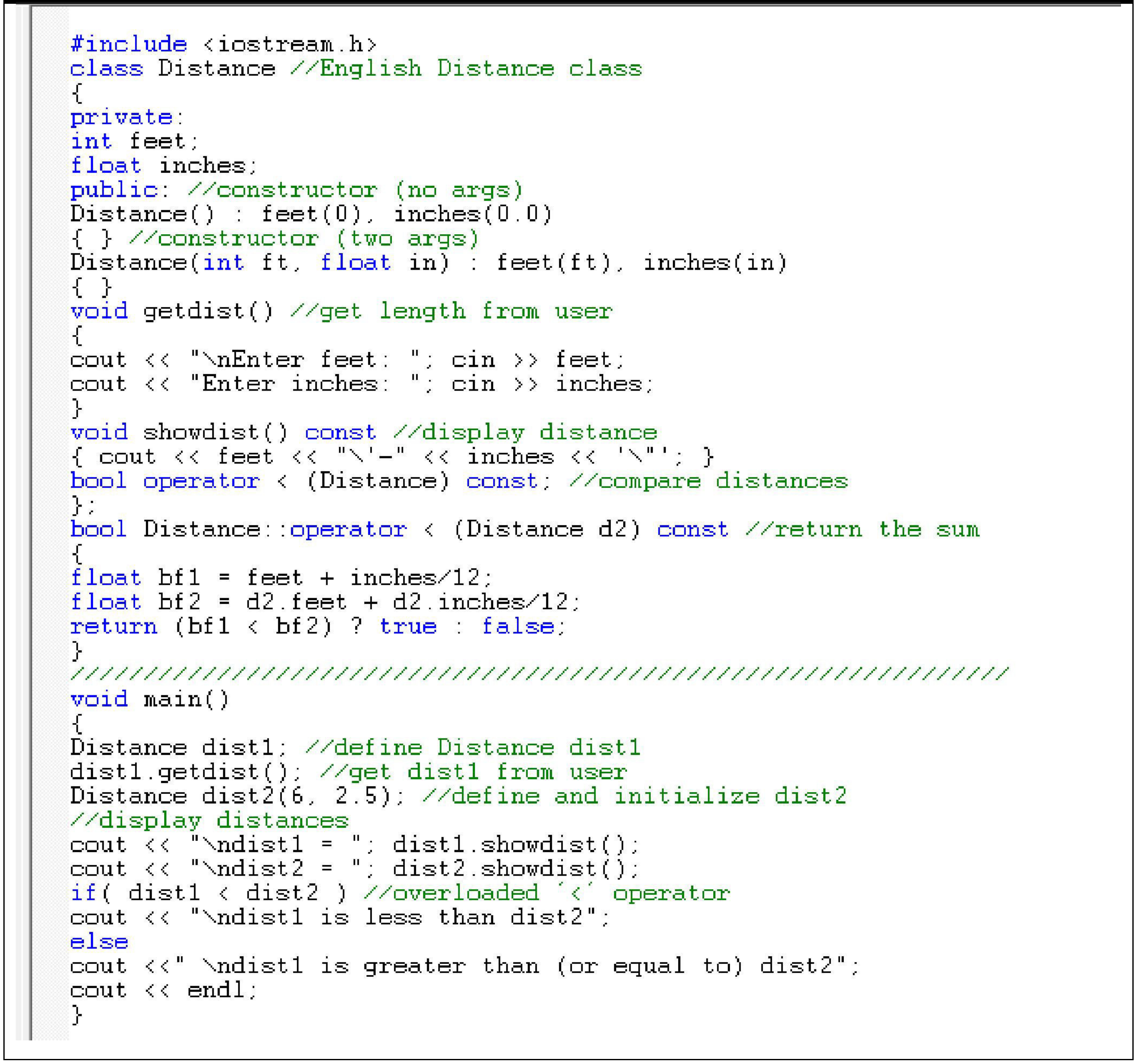
Comparison Operators

Let's see how to overload a different kind of C++ operator: comparison operators.

Comparing Distances

In our first example we'll overload the *less than* operator (<) in the Distance class so that we can compare two distances.

Example 7:-Write an oop program to compare two distances.



```
#include <iostream.h>
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ } //constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "\'-" << inches << '\"'; }
bool operator < (Distance) const; //compare distances
};
bool Distance::operator < (Distance d2) const //return the sum
{
float bf1 = feet + inches/12;
float bf2 = d2.feet + d2.inches/12;
return (bf1 < bf2) ? true : false;
}
////////////////////////////////////////////////////////////////
void main()
{
Distance dist1; //define Distance dist1
dist1.getdist(); //get dist1 from user
Distance dist2(6, 2.5); //define and initialize dist2
//display distances
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
if( dist1 < dist2 ) //overloaded '<' operator
cout << "\ndist1 is less than dist2";
else
cout <<" \ndist1 is greater than (or equal to) dist2";
cout << endl;
}
```

This program compares a distance entered by the user with a distance, 6'–2.5", initialized by the program. Depending on the result, it then prints one of two possible sentences.

Here’s some typical output:

```
Enter feet: 5
Enter inches: 11.5
dist1 = 5’-11.5”
dist2 = 6’-2.5”
dist1 is less than dist2
```

The approach used in the operator<() function is similar to overloading the + operator in the Example 7 except that here the operator<() function has a return type of Boolean. The return value is false or true, depending on the comparison of the two distances.

The comparison is made by converting both distances to floating-point feet, and comparing them using the normal < operator. Remember that the use of the conditional operator

```
return (bf1 < bf2) ? true : false;
```

is the same as

```
if (bf1 < bf2)
return true;
else
return false;
```

Comparing Strings

Here’s another example of overloading an operator, this time the *equal to* (==) operator. We’ll use it to compare two of our homemade String objects, returning true if they’re the same and false if they’re different.

Example 8:-Write an oop program to compare two strings.

```
#include <iostream.h>
#include <string.h>    //for strcmp()
////////////////////////////////////////////////////////////////
class String       //user-defined string type
{
private:
enum { SZ = 80 };   //size of String objects
char str[SZ];      //holds a string
public:
String()           //constructor, no args
{ strcpy(str, ""); }
String( char s[] )   //constructor, one arg
{ strcpy(str, s); }
void display() const  //display a String
{ cout << str; }
void getstr()      //read a string
{ cin.get(str, SZ); }
bool operator == (String ss) const   //check for equality
{
return ( strcmp(str, ss.str)==0 ) ? true : false;
}
};
////////////////////////////////////////////////////////////////
int main()
{
String s1 = "yes";
String s2 = "no";
String s3;
cout << "\nEnter 'yes' or 'no': ";
s3.getstr();          //get String from user
if(s3==s1)            //compare with "yes"
cout << "You typed yes\n";
else if(s3==s2)       //compare with "no"
cout << "You typed no\n";
else
cout << "You didn't follow instructions\n";
return 0;
}
```

The main() part of this program uses the == operator twice, once to see if a string input by the user is “yes” and once to see if it’s “no.”

Here's the output when the user types "yes":

```
Enter 'yes' or 'no': yes
You typed yes
```

The operator==() function uses the library function strcmp() to compare the two C-strings.

This functions return 0 if the strings are equal, a negative number if the first is less than the second, and a positive number if the first is greater than the second. Here *less than* and *greater than* are used in their lexicographical sense to indicate whether the first string appears before or after the second in an alphabetized listing.

Other comparison operators, such as < and >, could also be used to compare the lexicographical value of strings. Or, alternatively, these comparison operators could be redefined to compare string lengths. Since you're the one defining how the operators are used, you can use any definition that seems appropriate to your situation.

Arithmetic Assignment Operators

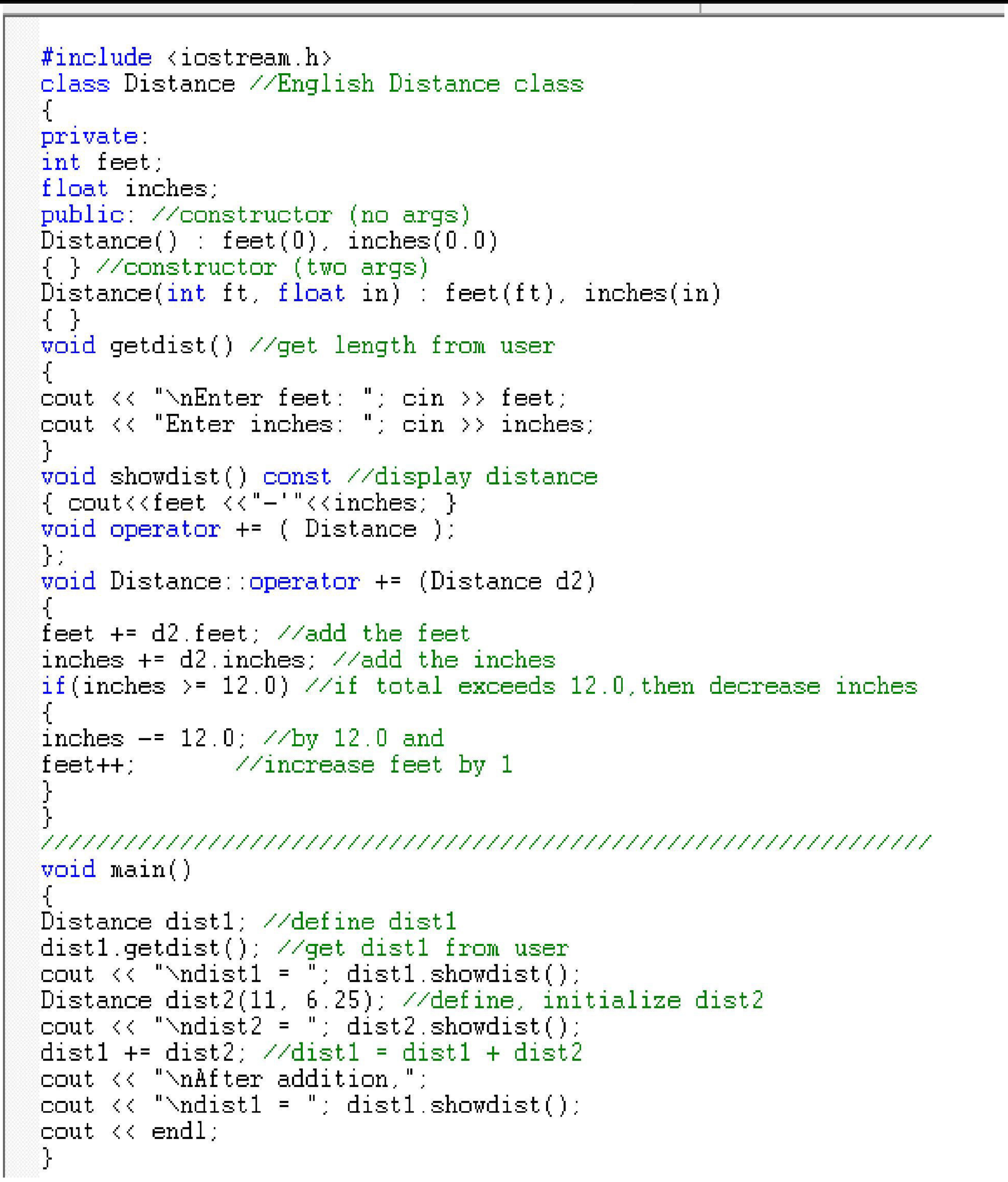
The += operator combines assignment and addition into one step. We'll use this operator to add one distance to a second, leaving the result in the first.

In this Example 9 we obtain a distance from the user and add to it a second distance, initialized to 11'–6.25" by the program.

In this program the addition is carried out in main() with the statement

dist1 += dist2; This causes the sum of dist1 and dist2 to be placed in dist1.

Example 9:-Write an oop program to add two distances using += operator.



```
#include <iostream.h>
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ } //constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout<<feet <<"-'"<<inches; }
void operator += ( Distance );
};
void Distance::operator += (Distance d2)
{
feet += d2.feet; //add the feet
inches += d2.inches; //add the inches
if(inches >= 12.0) //if total exceeds 12.0,then decrease inches
{
inches -= 12.0; //by 12.0 and
feet++;       //increase feet by 1
}
}
/////////////////////////////////////////////////////////////////
void main()
{
Distance dist1; //define dist1
dist1.getdist(); //get dist1 from user
cout << "\ndist1 = "; dist1.showdist();
Distance dist2(11, 6.25); //define, initialize dist2
cout << "\ndist2 = "; dist2.showdist();
dist1 += dist2; //dist1 = dist1 + dist2
cout << "\nAfter addition,";
cout << "\ndist1 = "; dist1.showdist();
cout << endl;
}
```

Here’s a sample of interaction with the program:

Enter feet: 3

Enter inches: 5.75

dist1 = 3’-5.75”

dist2 = 11’-6.25”

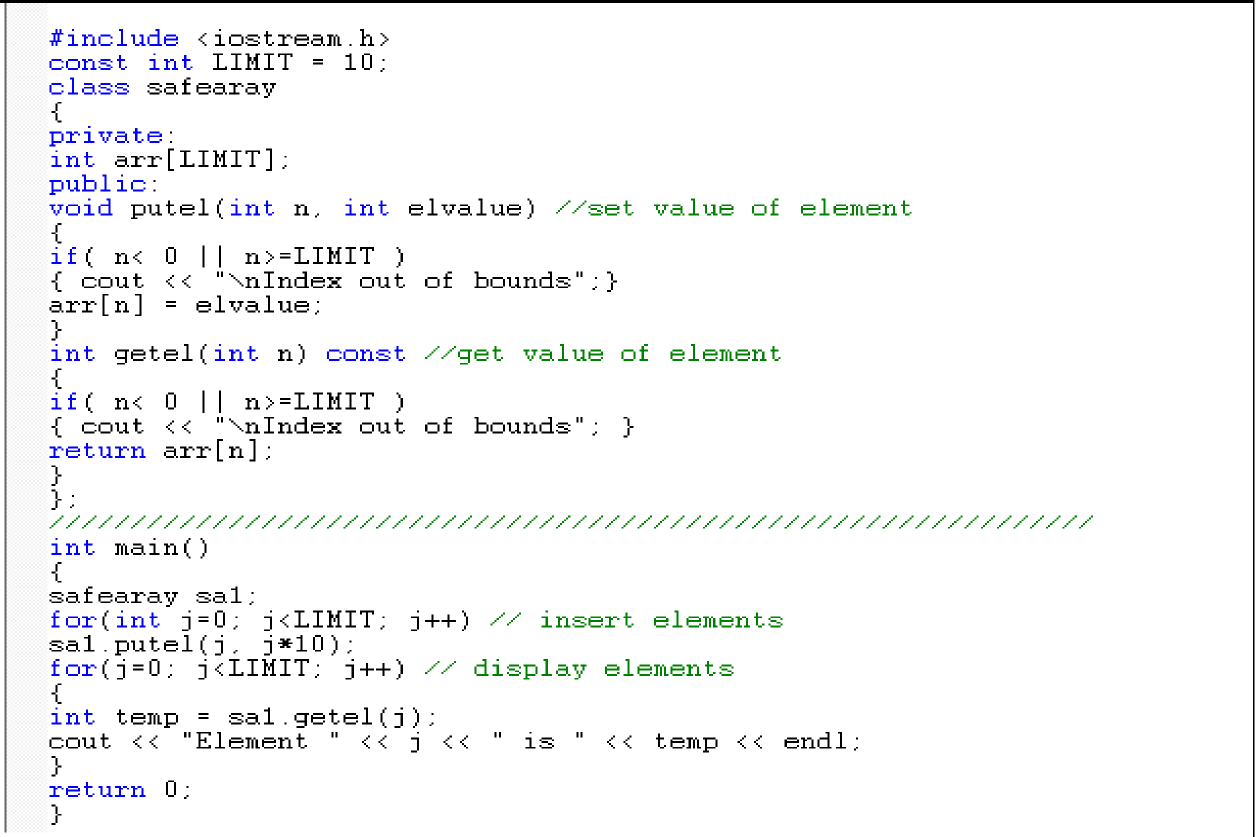
After addition,

dist1 = 15’-0”

The Subscript Operator ([])

The subscript operator, [], which is normally used to access array elements, can be overloaded.

Example 10:-Write an oop program to create an array.

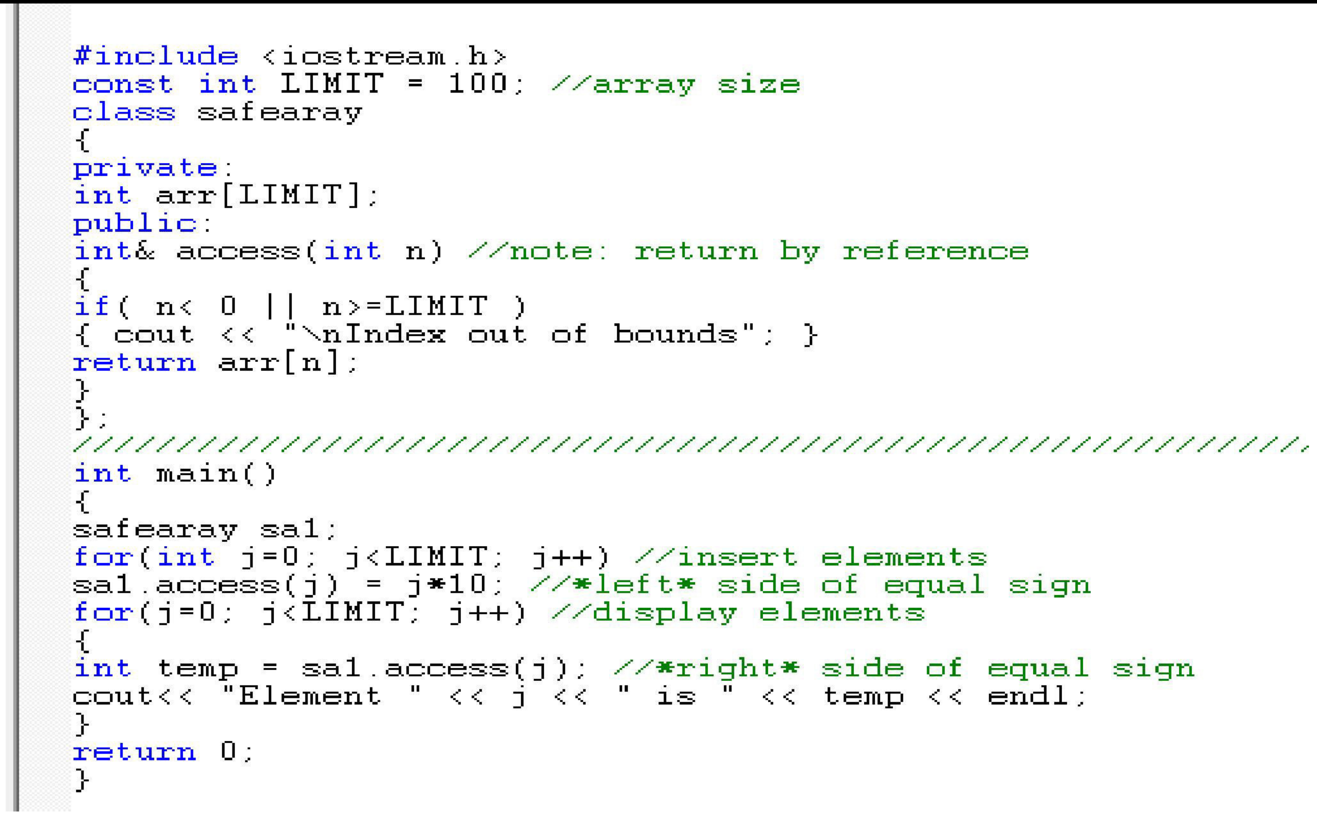


```
#include <iostream.h>
const int LIMIT = 10;
class safearay
{
private:
int arr[LIMIT];
public:
void putel(int n, int elvalue) //set value of element
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds";}
arr[n] = elvalue;
}
int getel(int n) const //get value of element
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds"; }
return arr[n];
}
};
/////////////////////////////////////////////////////////////////
int main()
{
safearay sa1;
for(int j=0; j<LIMIT; j++) // insert elements
sa1.putel(j, j*10);
for(j=0; j<LIMIT; j++) // display elements
{
int temp = sa1.getel(j);
cout << "Element " << j << " is " << temp << endl;
}
return 0;
}
```

Single access() Function Returning by Reference

As it turns out, we can use the same member function both to insert data into the safe array and to read it out. The secret is to return the value from the function by reference. This means we can place the function on the left side of the equal sign, and the value on the right side will be assigned to the variable returned by the function.

Example 11:-Write an oop program to create an array and return by reference.



```
#include <iostream.h>
const int LIMIT = 100; //array size
class safearay
{
private:
int arr[LIMIT];
public:
int& access(int n) //note: return by reference
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds"; }
return arr[n];
}
};
////////////////////////////////////////////////////////////////
int main()
{
safearay sa1;
for(int j=0; j<LIMIT; j++) //insert elements
sa1.access(j) = j*10; //*left* side of equal sign
for(j=0; j<LIMIT; j++) //display elements
{
int temp = sa1.access(j); //*right* side of equal sign
cout<< "Element " << j << " is " << temp << endl;
}
return 0;
}
```

The statement

sa1.access(j) = j*10; // *left* side of equal sign causes the value j*10 to be placed in arr[j], the return value of the function. It's perhaps slightly more convenient to use the same function for input and output of the safe array than it is to use separate functions; there's one less name to remember. But there's an even better way, with no names to remember at all.

Overloaded [] Operator Returning by Reference

To access the safe array using the same subscript ([]) operator that's used for normal C++ arrays, we overload the subscript operator in the safearay class. However, since this operator is commonly used on the left side of the equal sign, this overloaded function must return by reference.

Example 12:-Write an oop program to create an array using operator [] with overload by reference.

```
#include <iostream.h>
const int LIMIT = 100; //array size
class safearay
{
private:
int arr[LIMIT];
public:
int& operator [](int n) //note: return by reference
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds"; }
return arr[n];
}
};
///////////////////////////////////////////////////////////
int main()
{
safearay sa1;
for(int j=0; j<LIMIT; j++) //insert elements
sa1[j] = j*10; //*left* side of equal sign
for(j=0; j<LIMIT; j++) //display elements
{
int temp = sa1[j]; //*right* side of equal sign
cout << "Element " << j << " is " << temp << endl;
}
return 0;
}
```

In this program we can use the natural subscript expressions

sa1[j] = j*10; and **temp = sa1[j];** for input and output to the safe array.

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.

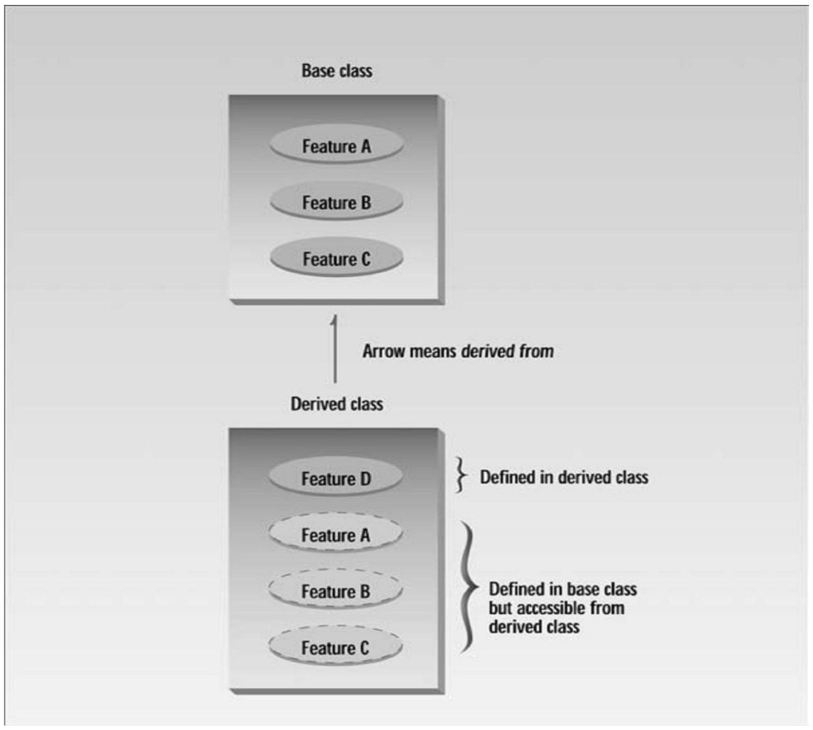


Figure 1: ***Inheritance.***

The arrow in Figure 1 goes in the opposite direction from the derived class to the base class, and to think of it as a "derived from" arrow.

Inheritance is an essential part of OOP. Its big payoff is that it permits code *reusability*. Once a base class is written and debugged, it need not be touched again, but, using inheritance can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability. Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.

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.

<u>Derived Class and Base Class</u>

Let's suppose that we have worked long and hard to make the Counter class operate just the way we want, and we're pleased with the results, except for one thing. We really need a way to decrement the count.

We could insert a decrement routine directly into the source code of the Counter class.

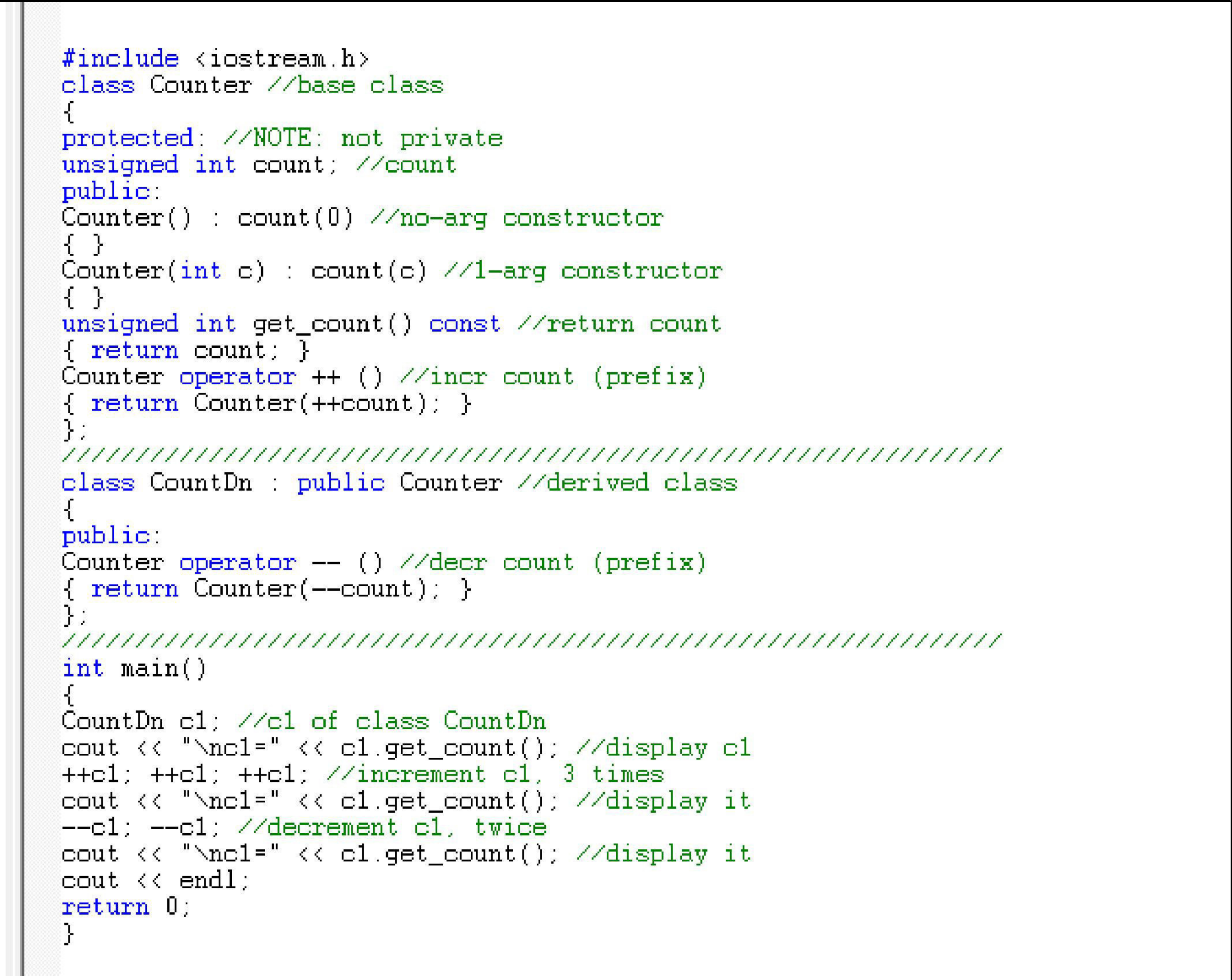
However, there are several reasons that we might not want to do this.

- First, the Counter class works very well and has undergone many hours of testing and debugging. If we start fooling around with the source code for Counter, the testing process will need to be carried out again, and of course we may foul something up and spend hours debugging code that worked fine before we modified it.

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

To avoid these problems we can use inheritance to create a new class based on Counter, without modifying counter itself. A new class, CountDn, that adds a decrement operator to the Counter class:

Example 1:-Write a program to decrement the counter variable using inheritance.



```
#include <iostream.h>
class Counter //base class
{
protected: //NOTE: not private
unsigned int count; //count
public:
Counter() : count(0) //no-arg constructor
{ }
Counter(int c) : count(c) //1-arg constructor
{ }
unsigned int get_count() const //return count
{ return count; }
Counter operator ++ () //incr count (prefix)
{ return Counter(++count); }
};
/////////////////////////////////////////////////////////////////
class CountDn : public Counter //derived class
{
public:
Counter operator -- () //decr count (prefix)
{ return Counter(--count); }
};
/////////////////////////////////////////////////////////////////
int main()
{
CountDn c1; //c1 of class CountDn
cout << "\nc1=" << c1.get_count(); //display c1
++c1; ++c1; ++c1; //increment c1, 3 times
cout << "\nc1=" << c1.get_count(); //display it
--c1; --c1; //decrement c1, twice
cout << "\nc1=" << c1.get_count(); //display it
cout << endl;
return 0;
}
```

Output of Example1
In main() we increment c1 three times, print out the resulting value, decrement c1 twice, and finally print out its value again. Here’s the output:

c1=0 ← after initialization

c1=3 ← after ++c1, ++c1, ++c1

c1=1 ← after --c1, --c1

The ++ operator, the constructors, the get_count() function in the Counter class, and the -- operator in the CountDn class all work with objects of type CountDn.

<u>Specifying the Derived Class</u>

Following the Counter class in the listing is the specification for a new class, CountDn. This class incorporates a new function, operator--(), which decrements the count. However and here’s the key point the new CountDn class inherits all the features of the Counter class.

CountDn doesn’t need a constructor or the get_count() or operator++() functions, because these already exist in Counter.

The first line of CountDn specifies that it is derived from Counter: class **CountDn : public Counter**

Here we use a single colon (not the double colon used for the scope resolution operator), followed by the keyword public and the name of the base class Counter. This sets up the relationship between the classes. This line says that CountDn *is derived from the base class* Counter.

Accessing Base Class Members

An important topic in inheritance is knowing when a member function in the base class can be used by objects of the derived class. This is called *accessibility*. Let's see how the compiler handles the accessibility issue in the example 1.

Substituting Base Class Constructors

In the main () part of Example1 we create an object of class CountDn:

CountDn c1;

This causes c1 to be created as an object of class CountDn and initialized to 0. But wait—how is this possible? There is no constructor in the CountDn class specifier, so what entity carries out the initialization? It turns out that—at least under certain circumstances—if you don't specify a constructor, the derived class will use an appropriate constructor from the base class. In example1 there's no constructor in CountDn, so the compiler uses the no-argument constructor from Count.

This flexibility on the part of the compiler using one function because another isn't available appears regularly in inheritance situations. Generally, the substitution is what you want, but sometimes it can be unnerving.

Substituting Base Class Member Functions

The object c1 of the CountDn class also uses the operator++() and get_count() functions from the Counter class. The first is used to increment c1:

++c1;

The second is used to display the count in c1:

cout << “\nc1=” << c1.get_count();

Again the compiler, not finding these functions in the class of which c1 is a member, uses member functions from the base class.

<u>The protected Access Specifier</u>

We have increased the functionality of a class without modifying it. Well, almost without modifying it. Let’s look at the single change we made to the Counter class. In the Counter class in example1, count is given a new specifier: protected. What does this do?

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 (using the dot operator, for example) public members of the class. It’s not allowed to use private members. For instance, suppose an object objA is an instance of class A, and function funcA() is a member function of A. Then in main() (or any other function that is not a member of A) the statement

objA.funcA();

will not be legal unless funcA() is public. The object objA cannot invoke private members of class A. Private members are, well, *private*. This is shown in Figure 2.

This is all we need to know if we don’t use inheritance. With inheritance, however, there is a whole raft of additional possibilities. The question that concerns us at the moment is, can member functions of the derived class access members of the base class? In other words, can operator--() in

CountDn access count in Counter? The answer is that member functions can access members of the base class if the members are public, or if they are protected. They can’t access private members.

We don’t want to make count public, since that would allow it to be accessed by any function anywhere in the program and eliminate the advantages of data hiding. A protected member, on the other hand, can be accessed by member functions in its own class or—and here’s the key—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 2.

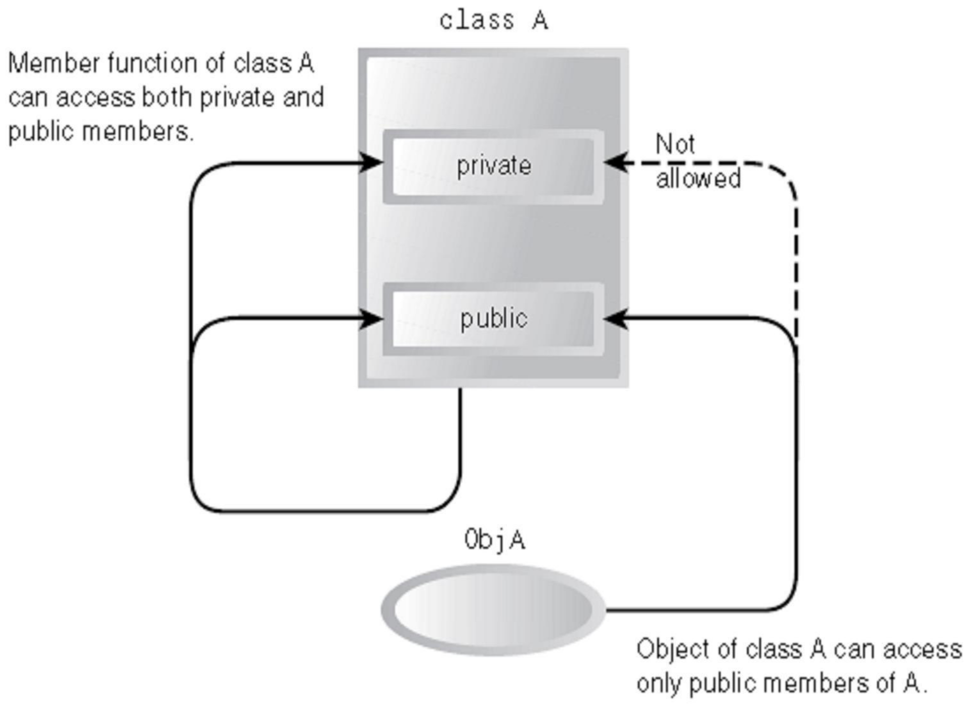

class A
Member function of class A can access both private and public members.
private
Not allowed
public
ObjA
Object of class A can access only public members of A.

Figure 2: Access *specifiers without inheritance*

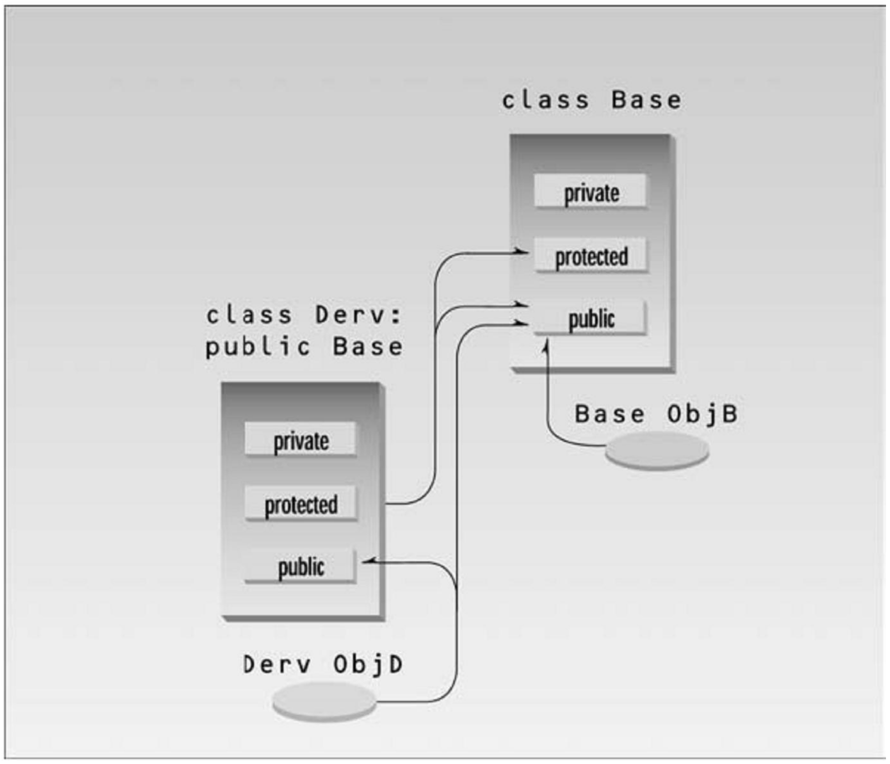


Figure 3: ***Access specifiers with inheritance***.

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

The moral is that if you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any member data that the derived classes might need to access should be made protected rather than private. This ensures that the class is “inheritance ready.”

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 in the base class, just as ordinary main() programs must do. Using the protected specifier leads to simpler programming,

Base Class Unchanged

Remember that, even if other classes have been derived from it, the base class remains unchanged. In the main() part of Example1 , we could define objects of type Counter:

Counter c2; ← object of base class

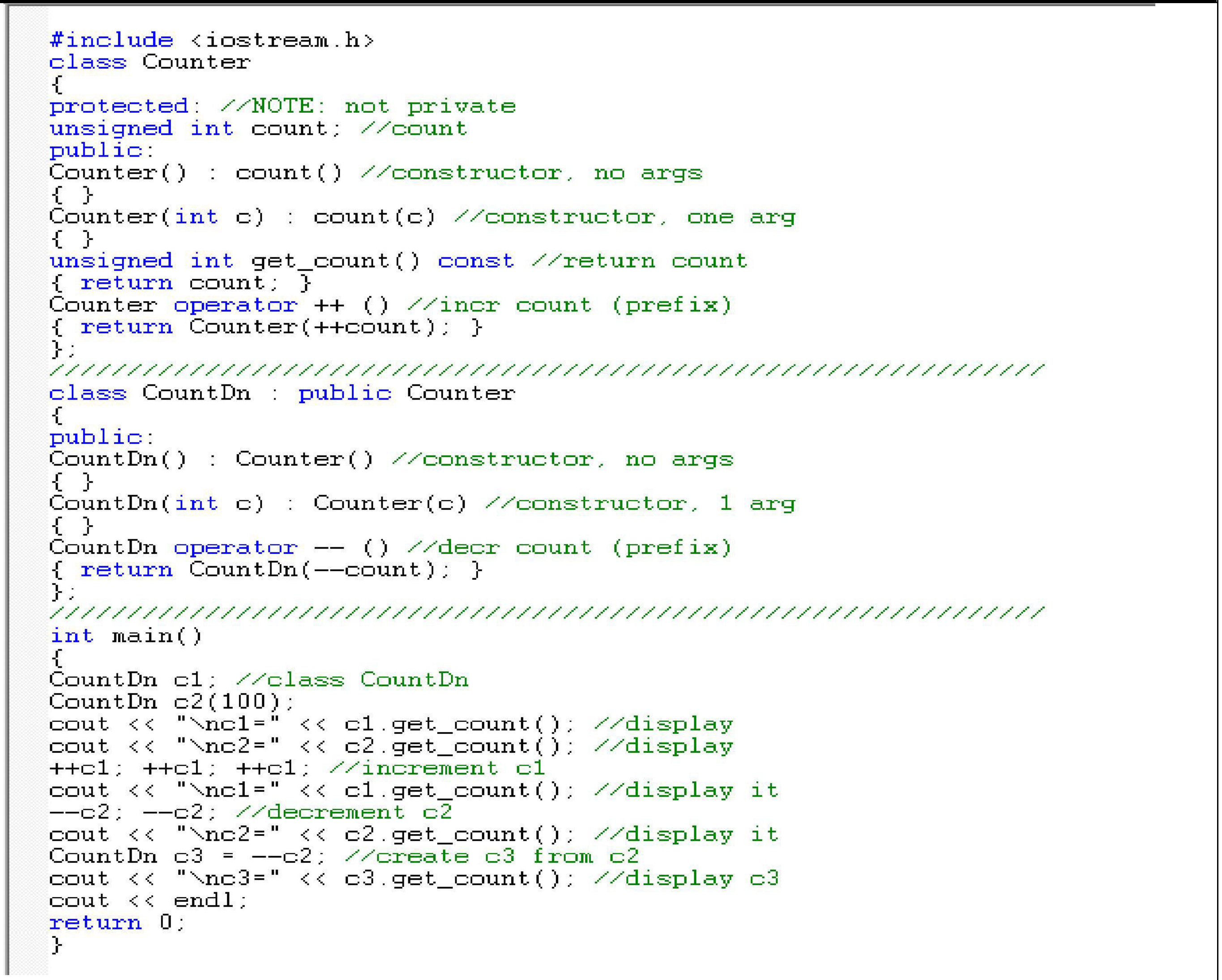
Such objects would behave just as they would if CountDn didn’t exist.

Note also that inheritance doesn’t work in reverse. The base class and its objects don’t know anything about any classes derived from the base class. In this example that means that objects of class Counter, such as c2, can’t use the operator--() function in CountDn. If you want a counter that you can decrement, it must be of class CountDn, not Counter.

Derived Class Constructors

There’s a potential glitch in the example1 program. What happens if we want to initialize a CountDn object to a value? Can the one-argument constructor in Counter be used? The answer is no. As we saw in example1, the compiler will substitute a no-argument constructor from the base class, but it draws the line at more complex constructors. To make such a definition work we must write a new set of constructors for the derived class. This is shown in the example2.

Example 2:-Write an oop program to decrement the counter variable using constructor in the derived class.



```
#include <iostream.h>
class Counter
{
protected: //NOTE: not private
unsigned int count; //count
public:
Counter() : count() //constructor, no args
{ }
Counter(int c) : count(c) //constructor, one arg
{ }
unsigned int get_count() const //return count
{ return count; }
Counter operator ++ () //incr count (prefix)
{ return Counter(++count); }
};
////////////////////////////////////////////////////////////////
class CountDn : public Counter
{
public:
CountDn() : Counter() //constructor, no args
{ }
CountDn(int c) : Counter(c) //constructor, 1 arg
{ }
CountDn operator -- () //decr count (prefix)
{ return CountDn(--count); }
};
////////////////////////////////////////////////////////////////
int main()
{
CountDn c1; //class CountDn
CountDn c2(100);
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count(); //display
++c1; ++c1; ++c1; //increment c1
cout << "\nc1=" << c1.get_count(); //display it
--c2; --c2; //decrement c2
cout << "\nc2=" << c2.get_count(); //display it
CountDn c3 = --c2; //create c3 from c2
cout << "\nc3=" << c3.get_count(); //display c3
cout << endl;
return 0;
}
```

This program uses two new constructors in the CountDn class. Here is the no-argument constructor:

```
CountDn() : Counter()
{ }
```

This constructor has an unfamiliar feature: the function name following the colon. This construction causes the CountDn() constructor to call the Counter() constructor in the base class. In main(), when we say

CountDn c1;

the compiler will create an object of type CountDn and then call the CountDn constructor to initialize it. This constructor will in turn call the Counter constructor, which carries out the work. The CountDn() constructor could add additional statements of its own, but in this case it doesn't need to, so the function body between the braces is empty.

The statement

CountDn c2(100);

in main() uses the one-argument constructor in CountDn. This constructor also calls the corresponding one-argument constructor in the base class:

CountDn(int c) : Counter(c) ← argument c is passed to Counter

{ }

This construction causes the argument c to be passed from CountDn() to Counter(), where it is used to initialize the object. In main(), after initializing the c1 and c2 objects, we increment one and decrement the other and then print the results.

The one-argument constructor is also used in an assignment statement.

CountDn c3 = --c2;

<u>Overriding Member Functions</u>

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 “Arrays and Strings.” That program modeled a stack, a simple data storage device. It allowed you to push integers onto the stack and pop them off. If you tried to push too many items onto the stack, the program might bomb, since data would be placed in memory beyond the end of the st[] array. Or if you tried to pop too many items, the results would be meaningless, since you would be reading data from memory locations outside the array.

To cure these defects we’ve created a new class, Stack2, derived from Stack. Objects of Stack2 behave in exactly the same way as those of Stack, except that you will be warned if you attempt to push too many items on the stack or if you try to pop an item from an empty stack.

Example 3:-Write an oo program to overload functions in base and derived stack classes.

```cpp
#include <iostream.h>
#include <process.h> //for exit()
class stack
{
protected:          //NOTE: can't be private
enum { MAX = 3 };   //size of stack array
int st[MAX];        //stack: array of integers
int top;        //index to top of stack
public:
stack()         //constructor
{ top = -1; }
void push(int var)   //put number on stack
{ st[++top] = var; }
int pop()            //take number off stack
{ return st[top--]; }
};
/////////////////////////////////////////////////////////////////
class stack2 : public stack
{
public:
void push(int var) //put number on stack
{
if(top >= MAX-1) //error if stack full
{ cout << "\nError: stack is full"; exit(1); }
stack::push(var);  //call push() in Stack class
}
int pop()        //take number off stack
{
if(top < 0)      //error if stack empty
{ cout << "\nError: stack is empty\n"; exit(1); }
return stack::pop(); //call pop() in Stack class
}
};
/////////////////////////////////////////////////////////////////
```

```cpp
void main()
{
stack2 s1;
s1.push(11);      //push some values onto stack
s1.push(22);
s1.push(33);
cout << endl << s1.pop();   //pop some values from stack
cout << endl << s1.pop();
cout << endl << s1.pop();
cout << endl << s1.pop();   //oops, popped one too many...
cout << endl;
}
```

<u>Which Function Is Used?</u>

The Stack2 class contains two functions, push() and pop(). These functions have the same names and the same argument and return types, as the functions in Stack. When we call these functions from main(), in statements like

s1.push(11);

how does the compiler know which of the two push() functions to use? Here's the rule: When the same function exists in both the base class and the derived class, the function in the derived class will be executed. (This is true of objects of the derived class. Objects of the base class don't know anything about the derived class and will always use the base class functions.) We say that the derived class function ***overrides*** the base class function. So in the preceding statement, since s1 is an object of class Stack2, the push() function in Stack2 will be executed, not the one in Stack.

The push() function in Stack2 checks to see whether the stack is full. If it is, it displays an error message and causes the program to exit. If it isn't, it calls the push() function in Stack. Similarly, the pop() function in Stack2 checks to see whether the stack is empty. If it is, it prints an error message and exits; otherwise, it calls the pop() function in Stack. In main() we push three items onto the stack, but we pop four. The last pop elicits an error message

The output

```
33
22
11
```

Error: stack is empty and terminates the program.

<u>Scope Resolution with Overridden Functions</u>

How do push() and pop() in Stack2 access push() and pop() in Stack? They use the scope resolution operator, ::, in the statements

Stack::push(var);

and

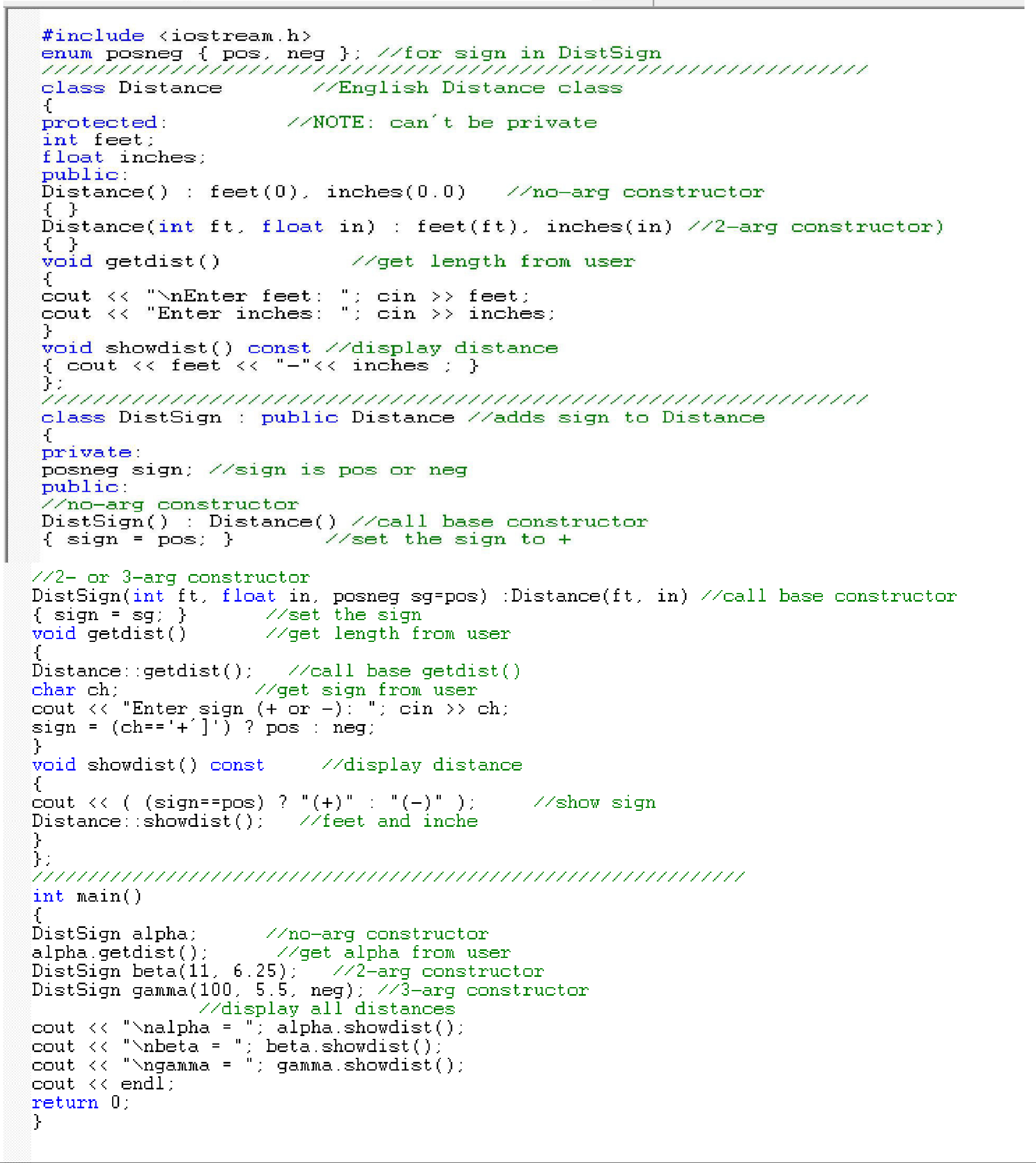
return Stack::pop();

These statements specify that the push() and pop() functions in Stack are to be called. Without the scope resolution operator, the compiler would think the push() and pop() functions in Stack2 were calling themselves, which—in this case—would lead to program failure. Using the scope resolution operator allows you to specify exactly what class the function is a member of.

<u>Inheritance in the English Distance Class</u>

Let's derive a new class from Distance. This class will add a single data item to our feet-and inches measurements: a sign, which can be positive or negative. When we add the sign, we'll also need to modify the member functions so they can work with signed distances.

Example 4:-Write an oo program to overload functions in base and derived distence classes.



```
#include <iostream.h>
enum posneg { pos, neg }; //for sign in DistSign
////////////////////////////////////////////////////////////////
class Distance        //English Distance class
{
protected:            //NOTE: can´t be private
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)   //no-arg constructor
{ }
Distance(int ft, float in) : feet(ft), inches(in) //2-arg constructor)
{ }
void getdist()             //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "-"<< inches ; }
};
////////////////////////////////////////////////////////////////
class DistSign : public Distance //adds sign to Distance
{
private:
posneg sign; //sign is pos or neg
public:
//no-arg constructor
DistSign() : Distance() //call base constructor
{ sign = pos; }         //set the sign to +

//2- or 3-arg constructor
DistSign(int ft, float in, posneg sg=pos) :Distance(ft, in) //call base constructor
{ sign = sg; }        //set the sign
void getdist()        //get length from user
{
Distance::getdist();    //call base getdist()
char ch;              //get sign from user
cout << "Enter sign (+ or -): "; cin >> ch;
sign = (ch=='+´]') ? pos : neg;
}
void showdist() const      //display distance
{
cout << ( (sign==pos) ? "(+)" : "(-)" );      //show sign
Distance::showdist();    //feet and inche
}
};
////////////////////////////////////////////////////////////////
int main()
{
DistSign alpha;      //no-arg constructor
alpha.getdist();      //get alpha from user
DistSign beta(11, 6.25);    //2-arg constructor
DistSign gamma(100, 5.5, neg); //3-arg constructor
               //display all distances
cout << "\nalpha = "; alpha.showdist();
cout << "\nbeta = "; beta.showdist();
cout << "\ngamma = "; gamma.showdist();
cout << endl;
return 0;
}
```

Here the DistSign class adds the functionality to deal with signed numbers. The Distance class in this program is just the same as in previous programs, except that the data is protected.
Actually in this case it could be private, because none of the derived-class functions accesses it. However, it's safer to make it protected so that a derived-class function could access it if necessary.

Here's some sample output:

```
Enter feet: 6
Enter inches: 2.5
Enter sign (+ or -): -
alpha = (-)6'-2.5"
beta = (+)11'-6.25"
gamma = (-)100'-5.5"
```

The DistSign class is derived from Distance. It adds a single variable, sign, which is of type posneg. The sign variable will hold the sign of the distance. The posneg type is defined in an enum statement to have two possible values: pos and neg.

Constructors in DistSign

DistSign has two constructors, mirroring those in Distance. The first takes no arguments; the second takes either two or three arguments. The third, optional, argument in the second constructor is a sign, either pos or neg. Its default value is pos. These constructors allow us to define variables (objects) of type DistSign in several ways.

Both constructors in DistSign call the corresponding constructors in Distance to set the feet and- inches values. They then set the sign variable. The no-argument constructor always sets it to pos. The second constructor sets it to pos if no third-argument value has been provided, or to a value (pos or neg) if the argument is specified.

The arguments ft and in, passed from main() to the second constructor in DistSign, are simply forwarded to the constructor in Distance.

Member Functions in DistSign

Adding a sign to Distance has consequences for both of its member functions. The getdist() function in the DistSign class must ask the user for the sign as well as for feet-and-inches values, and the showdist() function must display the sign along with the feet and inches. These functions call the corresponding functions in Distance, in the lines

Distance::getdist();

and

Distance::showdist();