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.

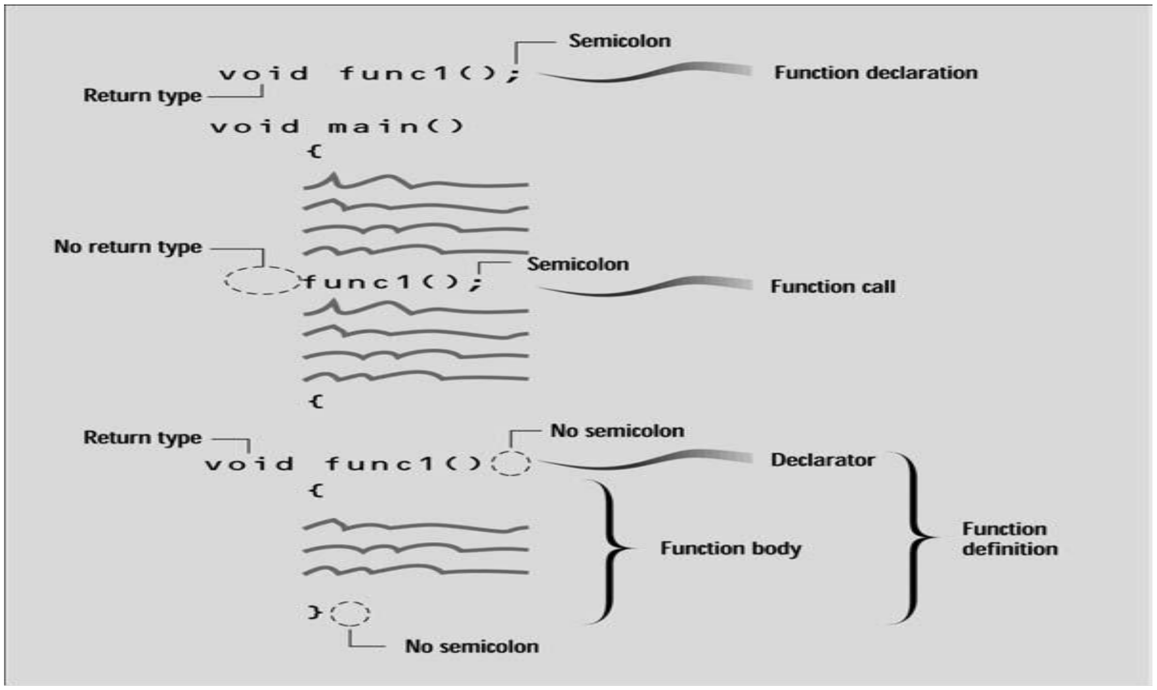


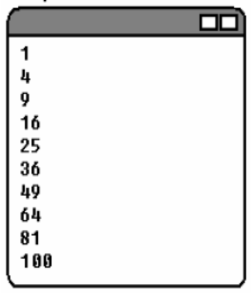
Figure 1: *Function syntax.*:

Example 1:

Design a function to calculate the squared value of an integer passed from the main function. Use this function in a program to calculate the squares of integers from 1..10.

```
#include<iostream.h>

int square(int);

void main( )
{
        int x;
        for (x=1;x<=10;x++)
                cout << square(x)<<endl;
}

int square(int y)
{
        return (y*y);
}
```

Output



```
1
4
9
16
25
36
49
64
81
100
```

Local and Global Variables:

All variables define inside the block of a function are called local variables. These variables belong to the function exclusively. That is no other function has access to it.

Example 2:

Write a function to find the largest integer among three integers entered by the user in the main function.

```
#include <iostream.h>

int max(int,int,int);

void main( )
{
        int largest,x1,x2,x3;
        cout<<"Enter 3 integer numbers:";
        cin>>x1>>x2>>x3;
        largest=max(x1,x2,x3);
        cout<<largest;
}
```

```
int max(int y1, int y2, int y3)
{
        int big;
        big=y1;
        if (y2>big) big=y2;
        if (y3>big) big=y3;
        return (big);
}
```

When variables declared outside any function, it is called global variables. This type of variables can be accessed by all the functions in the program as well as the main function.

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.

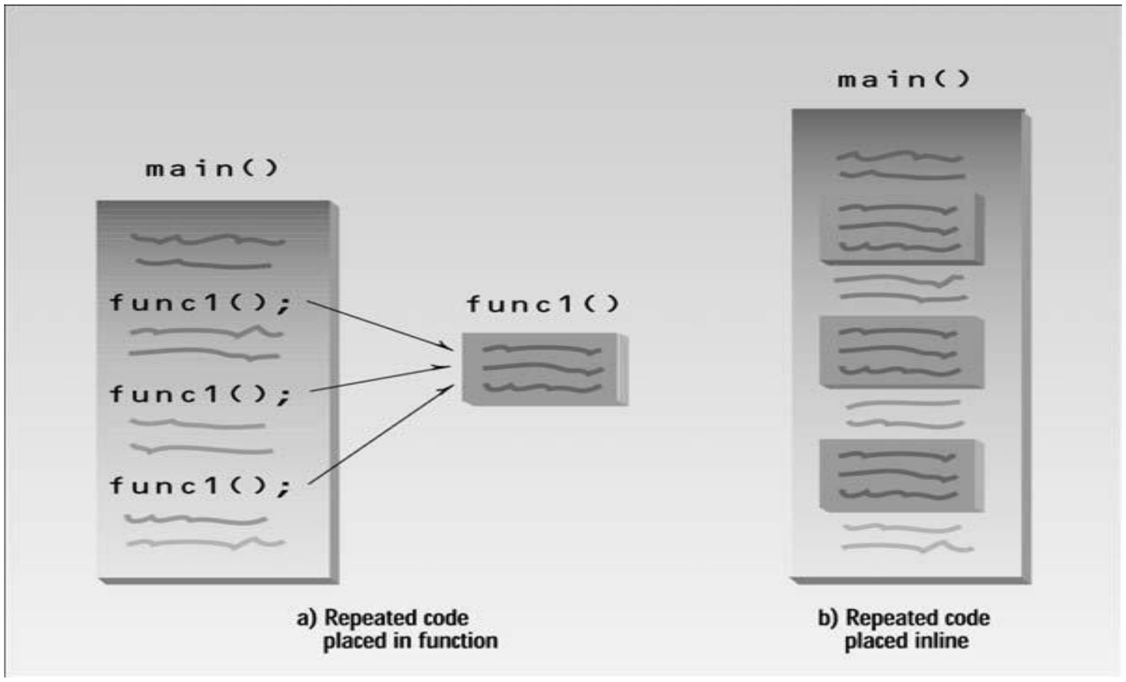

main()
func1();
func1();
func1();
func1()
main()
a) Repeated code placed in function
b) Repeated code placed inline

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

Example 3:

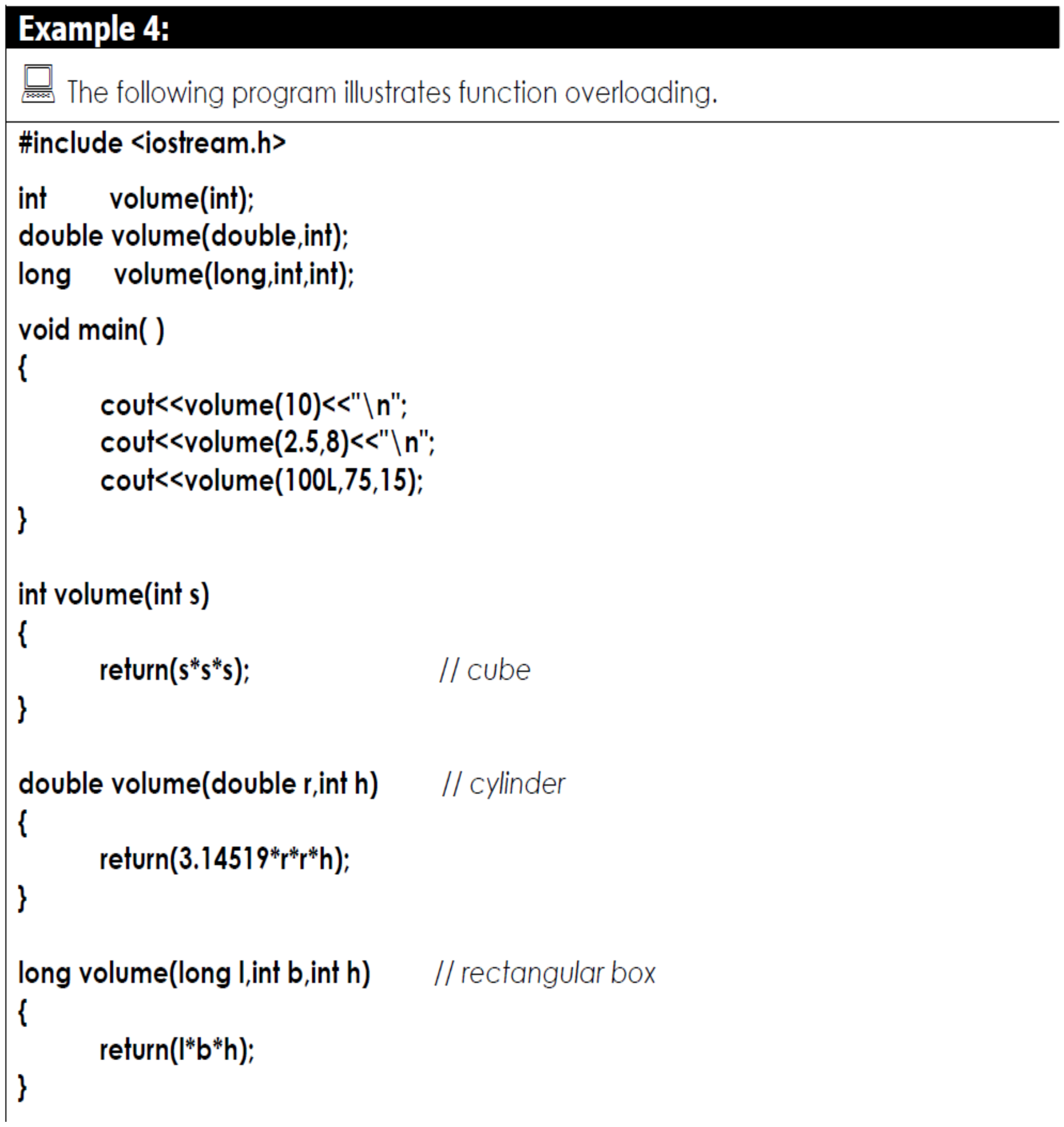
The following MAX is an inline function that returns the maximum of two values.

```
#include <iostream.h>

inline int MAX(int a, int b)
{
        return (a>b)?a:b;
}

void main( )
{
        int x1,x2;
        cout<<"Enter 2 integer numbers:";
        cin>>x1>>x2;
        cout<<MAX(x1,x2);
}
```

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.



Example 4:

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.

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.

This method is more efficient and provides higher execution speed than the call by value method, but call by value is more direct and easy to use.

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:

Write a function that counts uppercase letter in a string entered by the user in the main program. Assume the maximum string length is 100.

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

int const size=100;

int upperCount(char[]);

void main( )
{
        char str[size];
        cout<<"Enter Your String: ";
        cin.getline(str,size,'\n');
        cout<<upperCount(str);
}

int upperCount(char x[])
{
        int count=0;
        for (int i=0;i<strlen(x);i++)
           if (x[i]>='A'&&x[i]<='Z')     count++;
        return (count);
}
```

Example 8:

Write a function that counts the number of words in a string entered by the user in the main program. Assume the maximum string length is 100.

<u>Hint</u>: *the number of words in a sentence can be found by counting the number of spaces.*

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

int const size=100;

int wordCount(char[]);

void main( )
{
        char str[size];
        cout<<"Enter a Sentenc: ";
        cin.getline(str,size,'\n');
        cout<<wordCount(str);
}

int wordCount(char x[])
{
        int count=1;

        for (int i=0;i<strlen(x);i++)
                if (x[i]==' ')    count++;
        return (count);
}
```

Example 9:

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 1: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. It's called three times from main(). 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. Thus the three calls to the function all work, even though each has a different number of arguments.

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.

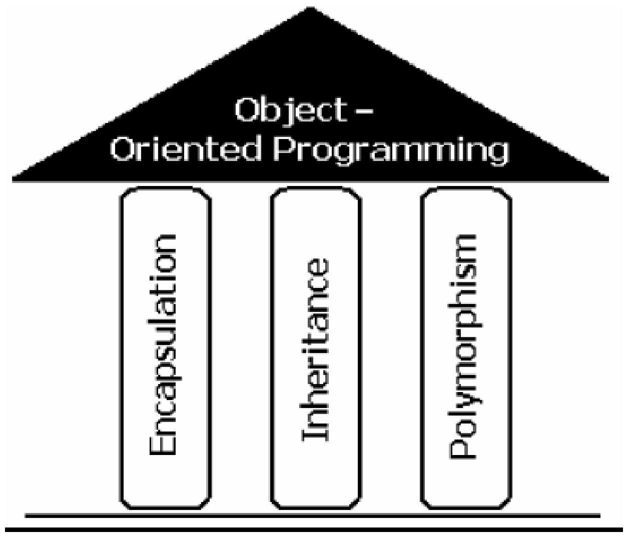


Figure 1: The three pillars of OOP.

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:

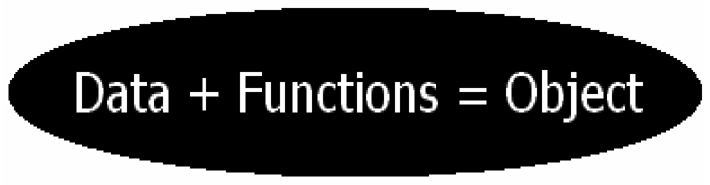
C++ supports the idea that different objects do "the right thing" through what is called function ***polymorphism*** and class polymorphism.

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

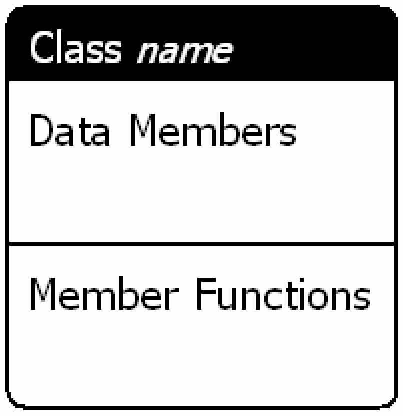
Class Definition:

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.

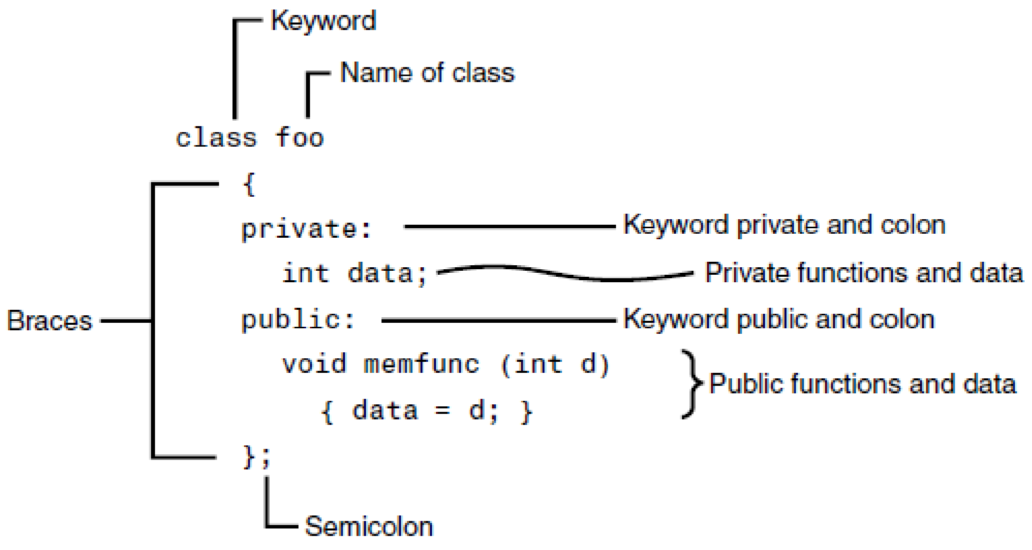


Note: Class is a specification for number of objects.



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**.


Keyword
Name of class
Braces
Keyword private and colon
Private functions and data
Keyword public and colon
Public functions and data
Semicolon

```
class foo
  {
  private:
     int data;
  public:
     void memfunc (int d)
       { data = d; }
  };
```

Figure2: Syntax of a class definition

We will **encapsulate** the data member which found in an object with member functions, and as you extend a class you will **"overload"** and **"override"** the functions of the *base class*.

When the permanently associate all related functions to the data of the class. This process is known as **encapsulation**.

Classes can be used to produce more specialized versions through a concept called **Inheritance**.

We can inherit features of the "**base class**", when we create a new "**derived class**". That is, we will create general object classes and then make more specific classes from them, deriving the particular from the general.

With such concept; new classes can be built on top of existing ones and extending their base classes; therefore allowing software designers to ***reuse code*** easily, which result in reducing the ***development time***.

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.

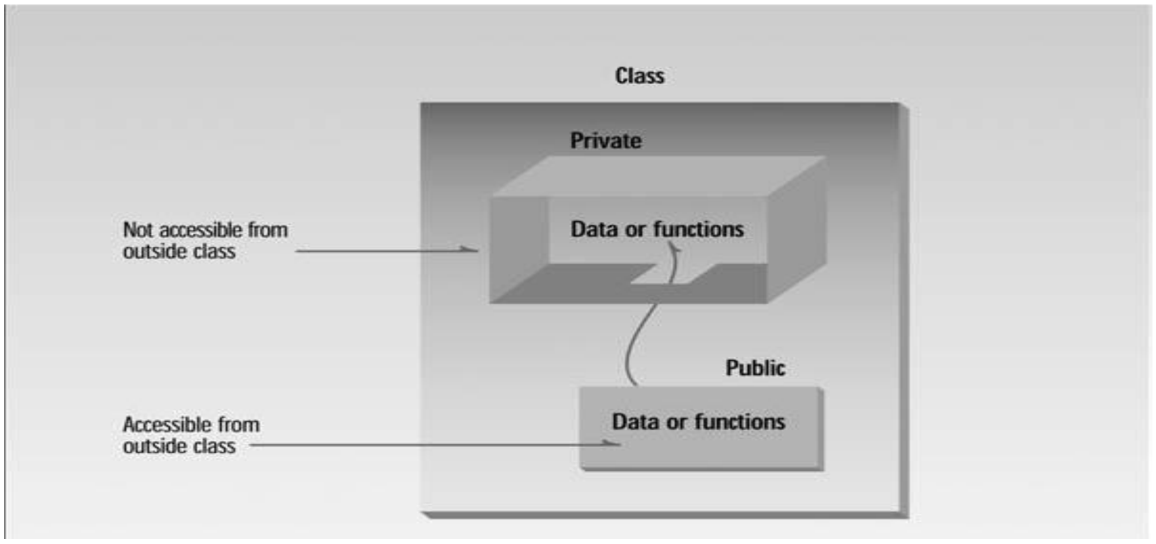

Class
Private
Data or functions
Not accessible from outside class
Public
Data or functions
Accessible from outside class

Figure 3: Public and Private Definition

Example 1:

Write a C++ program to modeling the Rectangle class.

```
# include <iostream.h>

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


int main( )
{
  Rectangle my_rectangle;

  my_rectangle.length = 6;
  my_rectangle.width = 7;
  cout<< my_rectangle.area( );

  return 0;
}
```

Example 2:A write an oo program to read and write somedata

```
#include <iostream>
using namespace std;
class smallobj     //define a class
{
private:
int  somedata; //class data
public:
void  setdata(int d)              //member function to set data
{
somedata = d;
 }
void showdata()                 //member function to display data
{ cout << “Data is “ << somedata << endl; }
};
int main()
{
smallobj s1, s2;              //define two objects of class smallobj
s1.setdata(1066);       //call member function to set data
s2.setdata(1776);
s1.showdata();           //call member function to display data
s2.showdata();
return 0;
}
```

Example 3: write an oo program to define the coordinate of point and change the 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);
}
```

Class Constructors and Destructors:

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

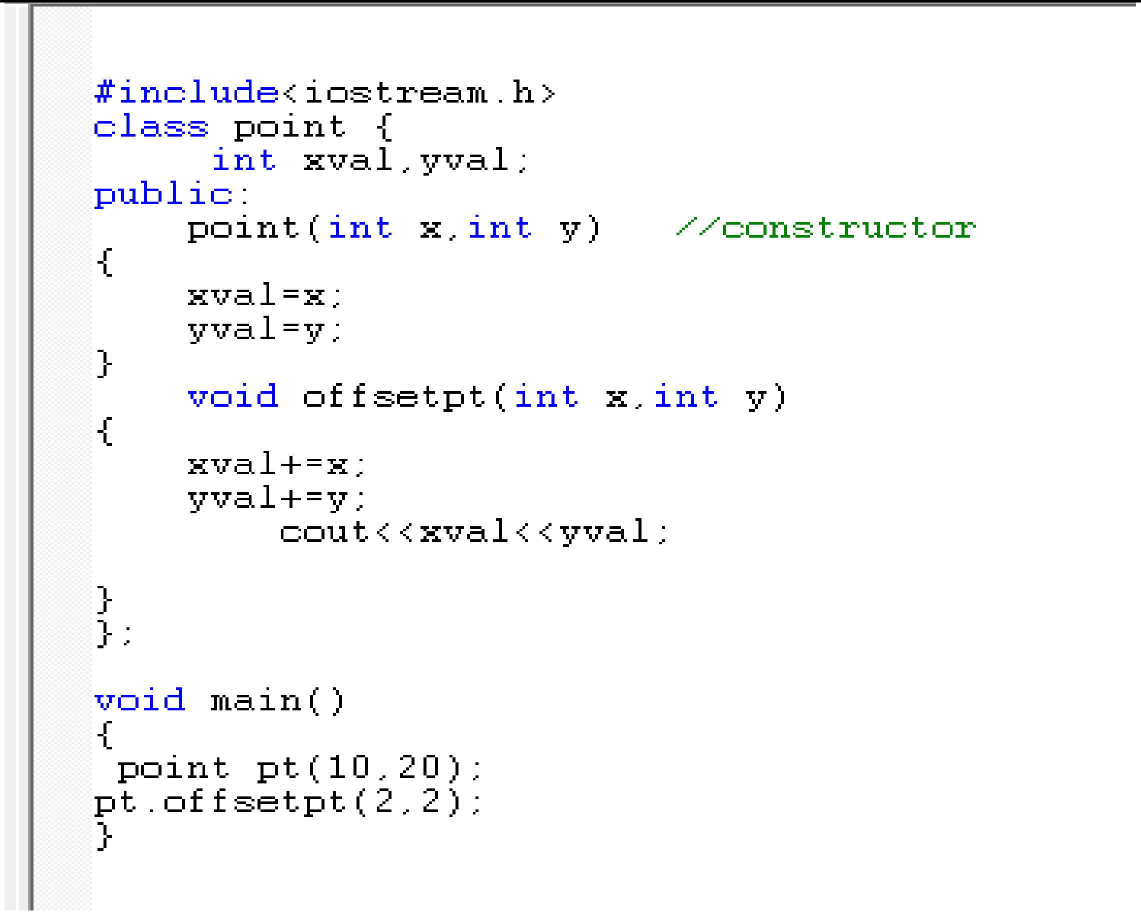
The main purpose of a class constructor is to perform any initializations related to the class instances via passing of some parameter values as initial values and allocate proper memory locations for that object.

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.

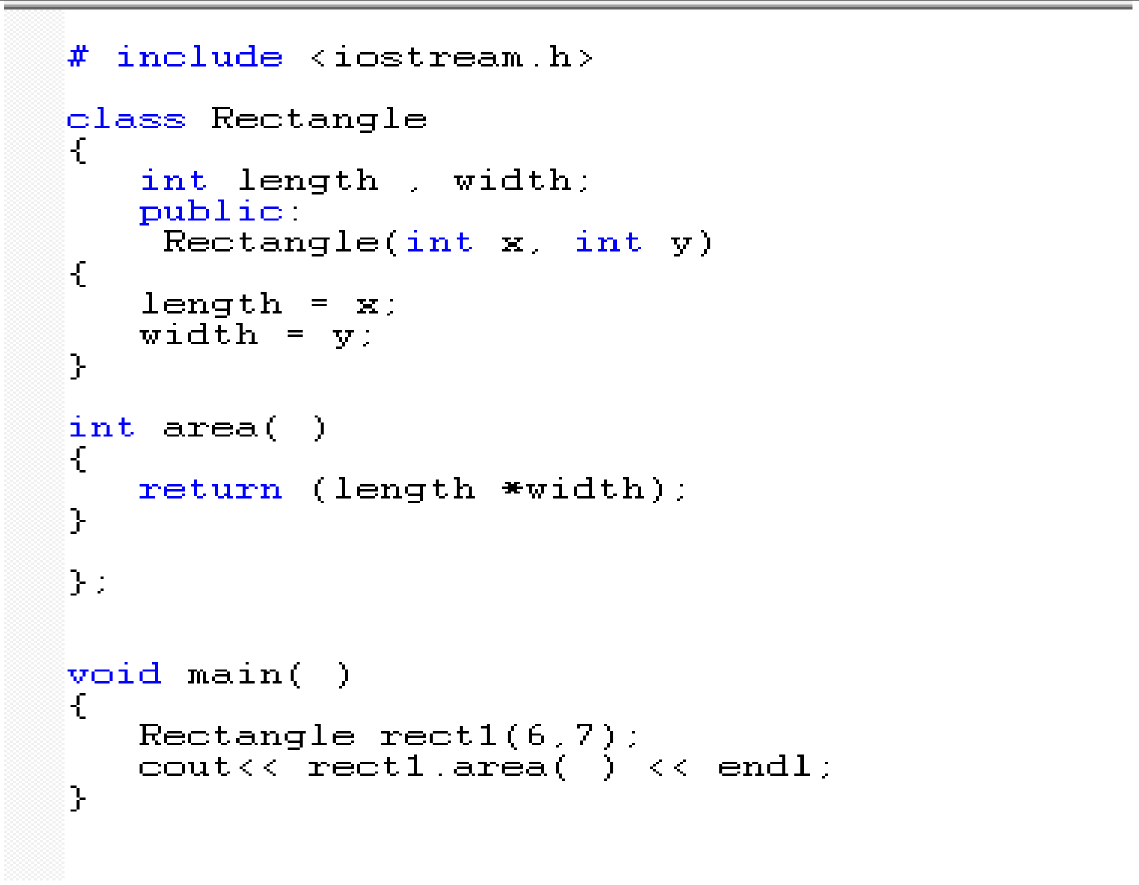
Note3: A class constructor can be overloaded

Example 1: Write an oo program 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);
}
```

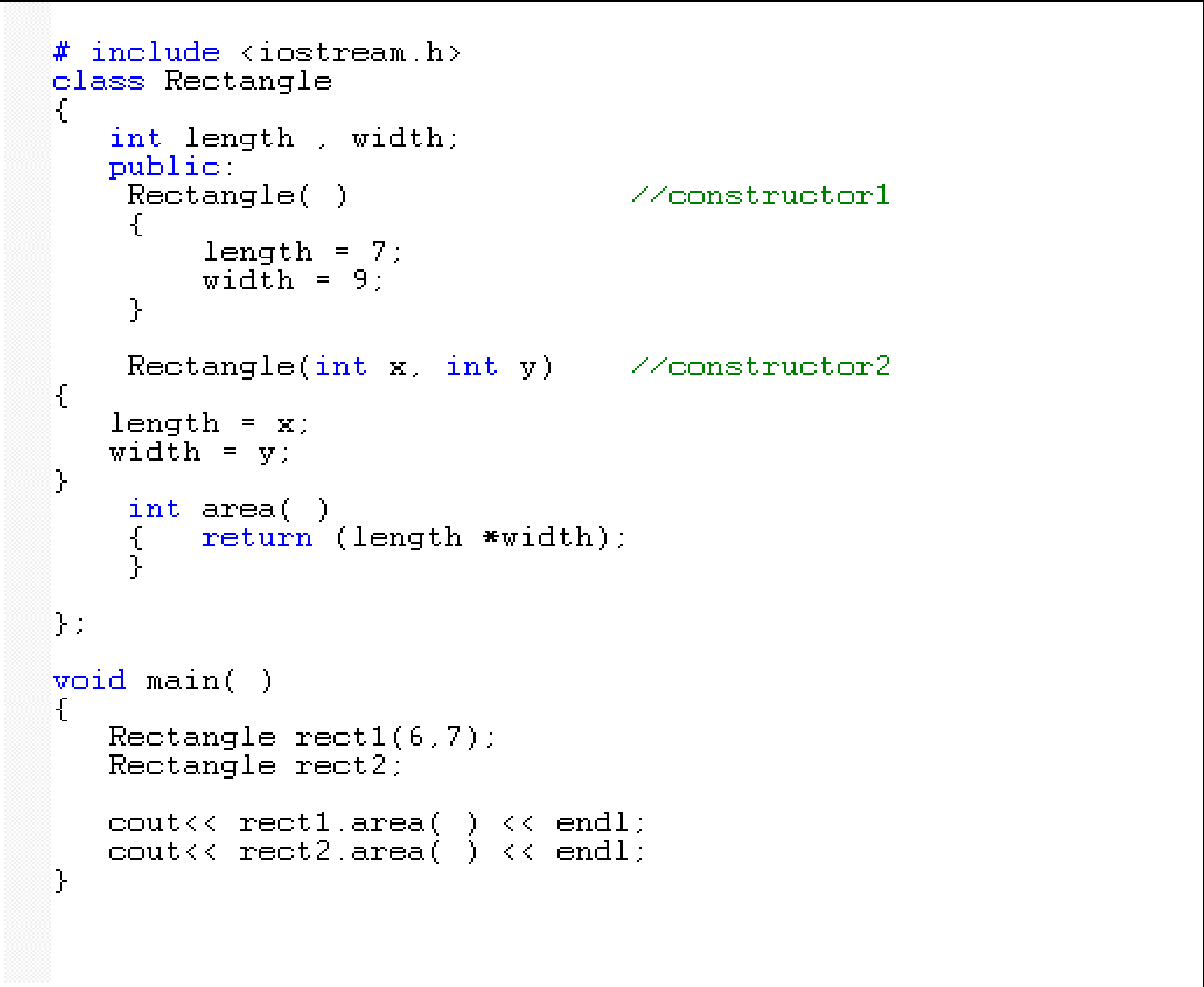
Example 2: Write an oo program to represent a rectangle constructor.



```
# 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 oo program to represent multiple constructors.



```
# 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;
}
```

Example 4 Write an oo program to represent simple constructor with class counter which contains one data members count.

```
#include <iostream>
class Counter
{
private:
int count; //count
public:
Counter() : count(0)       //constructor
{ /*empty body*/ }
void inc_count()          //increment count
{ count++; }
int get_count()           //return count
{ return count;  }
};

int main()
{
Counter c1, c2; //define and initialize
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count();
c1.inc_count(); //increment c1
c2.inc_count(); //increment c2
c2.inc_count(); //increment c2
cout << "\nc1="<< c1.get_count(); //display again
cout << "\nc2=" << c2.get_count();
cout << endl;
return 0;
};
```

Output:

```
C1=0

C2=0

C1=1

C2=2
```

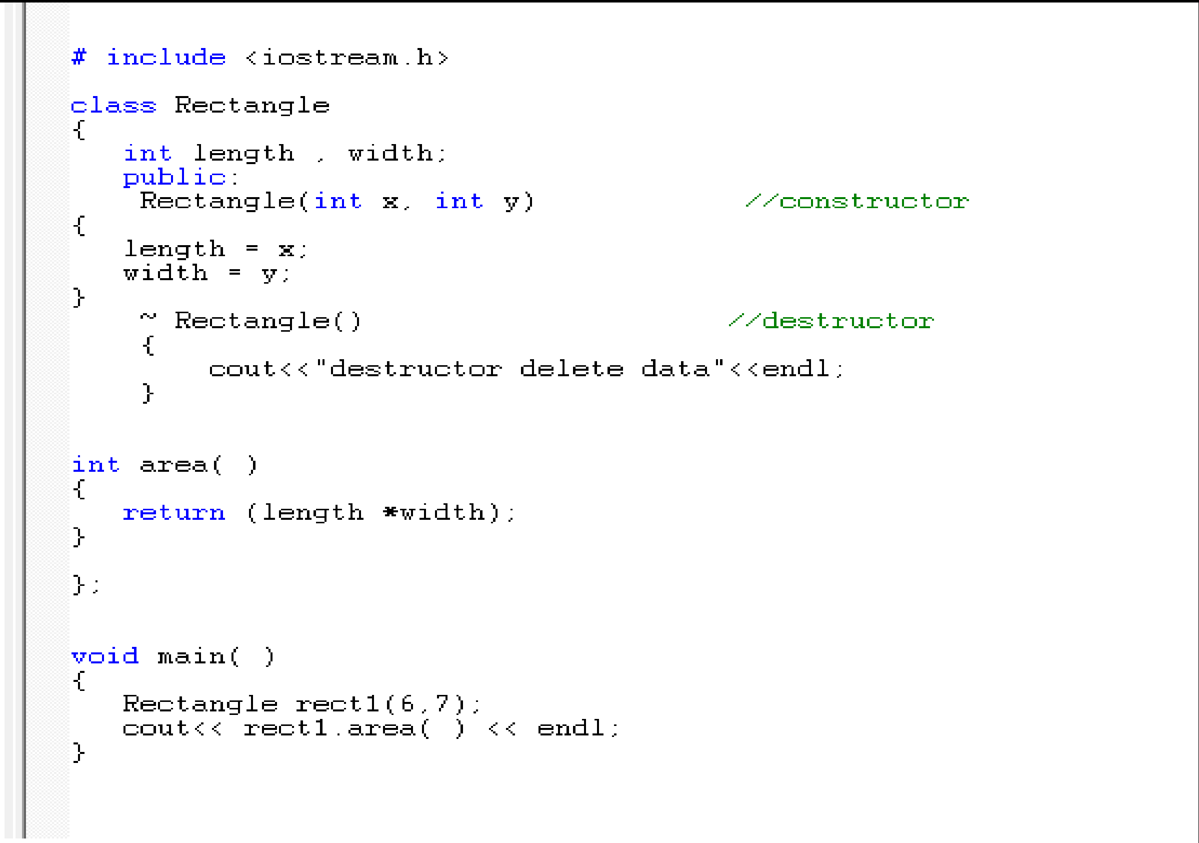
Destructor

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.

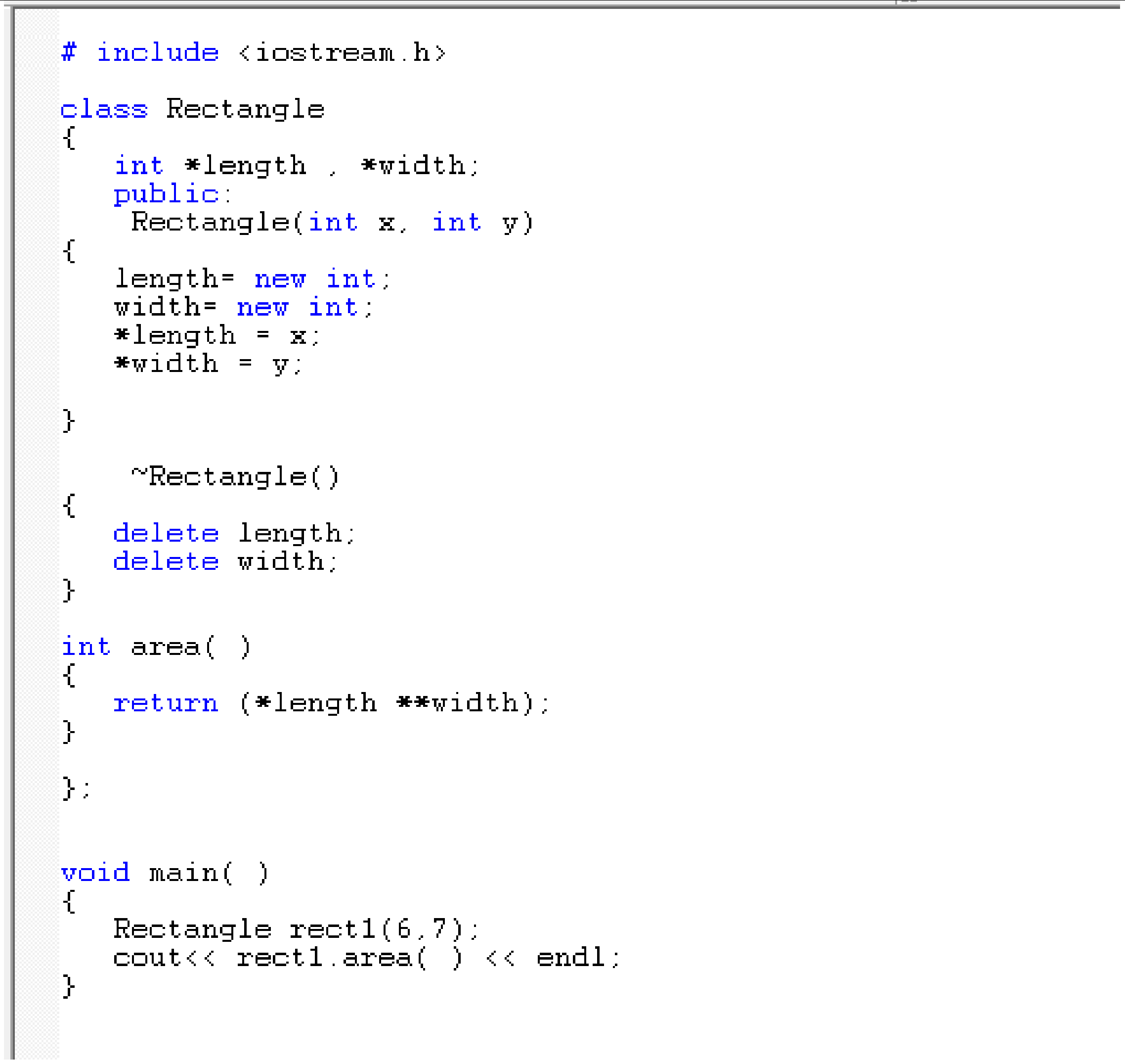
In such cases it is important to release member-allocated memory before the object is destroyed. A destructor can do just that.

Example 1: Write an oo program 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
    {
        cout<<"destructor delete data"<<endl;
    }


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

};


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

Example2: Write an oo program to represent destructor of pointer members.



```
# 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 oo program to find the summation of point using friend function.

```
#include <iostream.h>

class point
{
private:
int xval,yval;
public:
    point()
    {
        xval=0;
        yval=0;
    cout<<"xval= "<<xval<<"yval= "<<yval<<endl;

    }
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 ss = p.xval + p.yval;
return (ss);
}

void main( )
{ point p2;
    point p(2,2);
    int dd;
        dd=sum(p);
    cout<<" the summation of coordinate x & y = "<<dd<<endl;
}
```

Example2: Write an oo program to represent a friend function between two classes (alpha and beta) .

```
#include <iostream.h>

class beta;                        //needed for add declaration
class alpha
{
private:
int data;
public:
alpha()
 {    data=3;    }
friend int add(alpha, beta);      //friend function
};
class beta
{
private:
int data;
public:
beta()
{  data=7; }
friend int add(alpha, beta);       //friend function
};

int add(alpha a, beta b)       //function definition
{
return( a.data + b.data );
}

int main()
{
alpha aa;
beta bb;
cout << add(aa, bb) << endl; //call the function
return 0;
}
```

Example3:Write an oo program to find the square of distance using friend function with overloading constructors

```
#include <iostream.h>
class Distance
{
private:
int feet;
float inches;
public:
Distance()                          //constructor (no arguments)
{
    feet=0;
 inches=0.0 ;
}

Distance(int ft, float in)          //constructor (two arguments)
{
    feet=ft;
 inches=in;
}
void showdist()                               //display distance
{ cout<<feet <<" "<<inches <<" "; }
friend float square(Distance);                  //friend function
};
//----------------------------------------------------------------
float square(Distance d)                      //return square of
{                                           //this Distance
float fltfeet = d.feet + d.inches/12;        //convert to float
float feetsqrd = fltfeet * fltfeet;           //find the square
return feetsqrd;                             //return square feet
}
/////////////////////////////////////////////////////////////////
int main()
{
Distance dist(3, 6.0);                          //two-arg constructor (3′-6″)
float sqft;
sqft = square(dist);                              //return square of dist
                                                  //display distance and square
cout << "\nDistance = "; dist.showdist();
cout << "\nSquare = " << sqft << " square feet\n";
return 0;
}
```

Example 4: Write an oo program to represent a friend function between two classes (point and D3) .

```
#include <iostream.h>
class D3;
class point
{
     int xVal;
     int yVal;
 public:
   point(int x, int y)
       {   xVal=x;
           yVal=y ;
     }
   friend void func1(point a, D3 b);
};
class D3
{
private:
int zVal;
public:
    D3(int zz)
{ zVal=zz;
}
  friend void func1(point a, D3 b);
};
void func1(point a, D3 b) {
cout << "\nxVal=" << a.xVal; cout << "\nyVal=" << a.yVal; cout << "\nzVal="
<< b.zVal; }
int main()
{
point d1(3,2);
D3 d2(6);
func1(d1,d2);
cout << endl;
return 0;
}
```

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.

The extreme case of having all member functions of a class A as friends of another class B can be expressed in an *abbreviated* form:

class A;

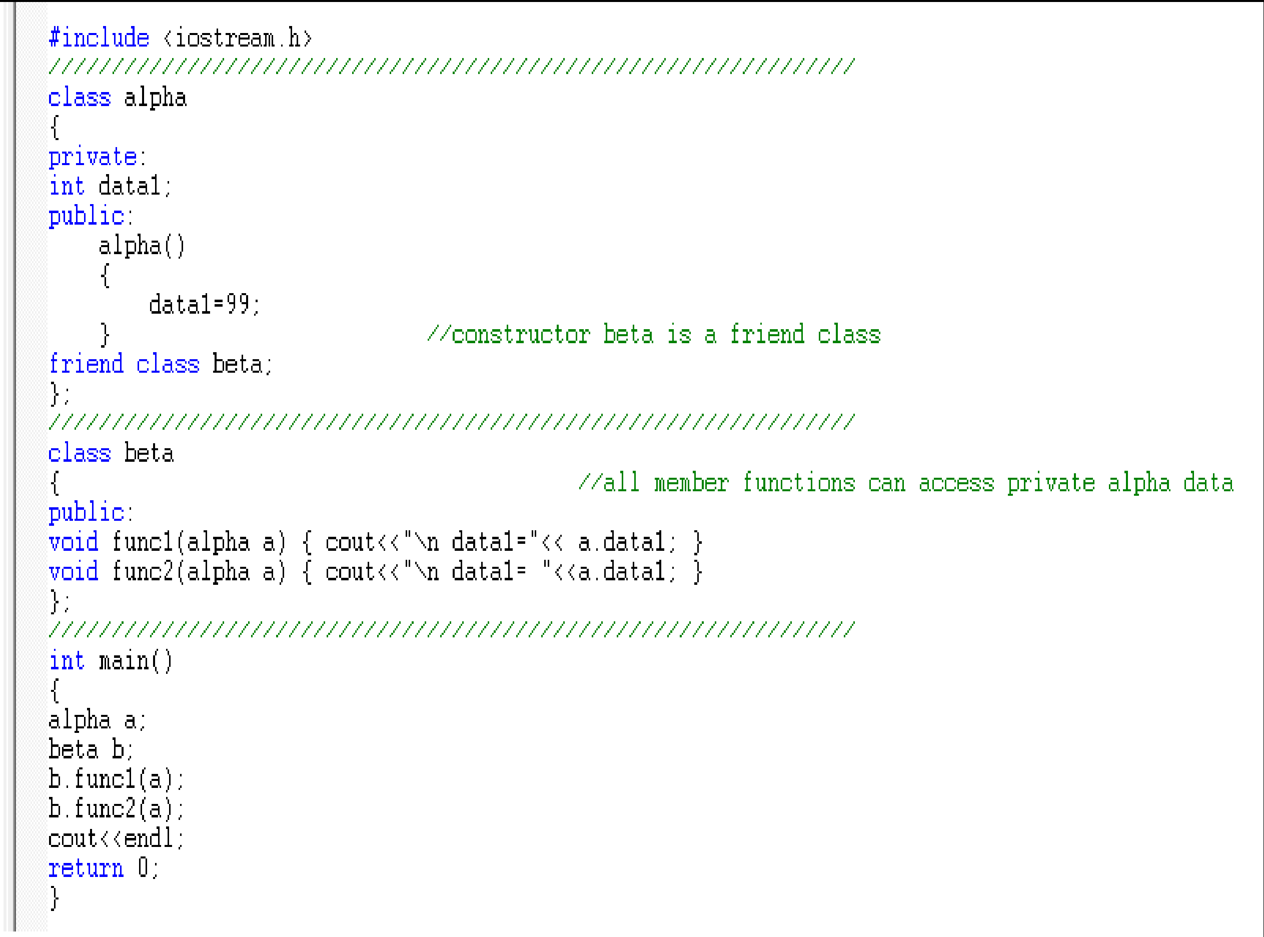
class B {

//...

friend class A; // abbreviated form

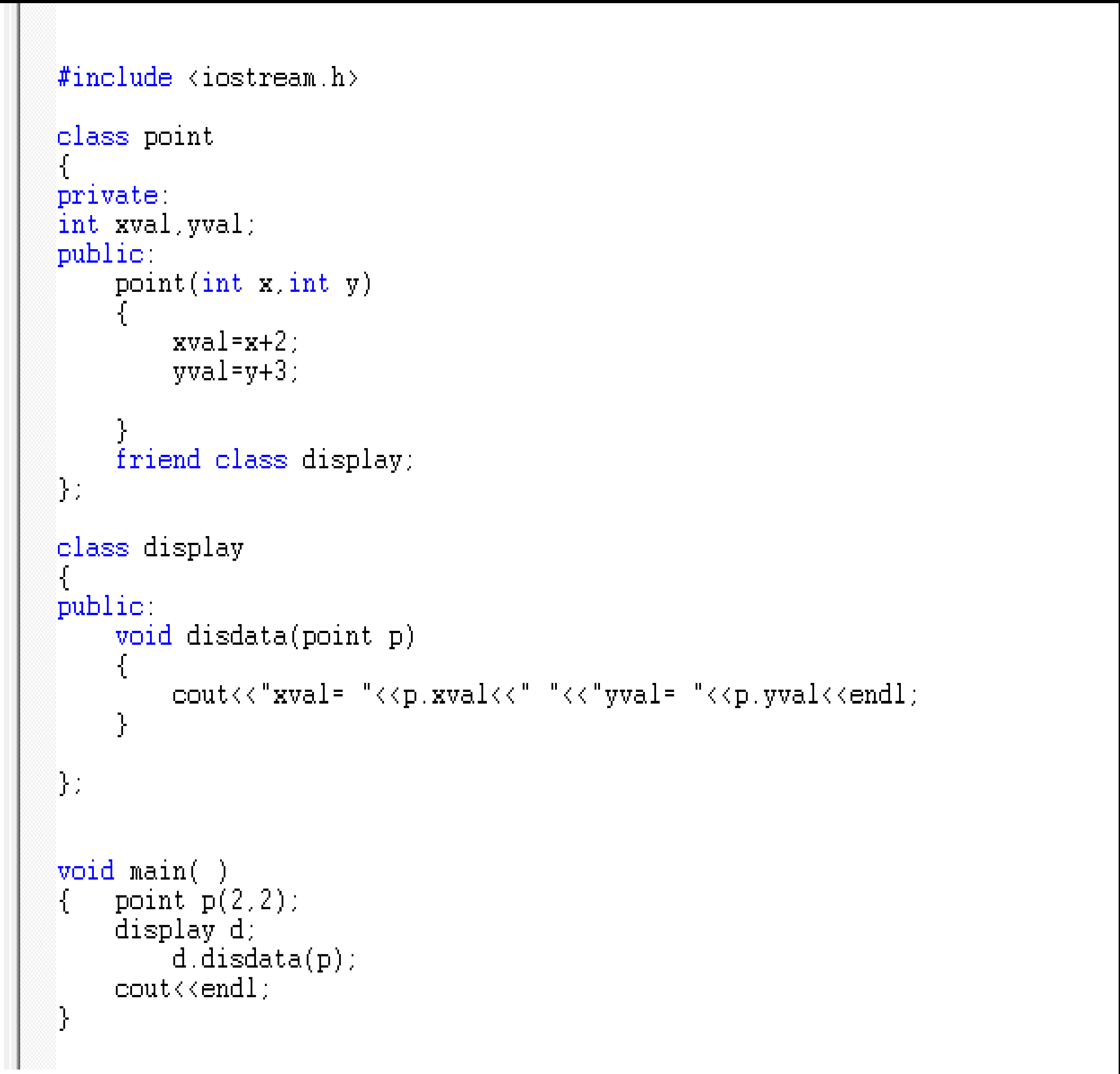
};

Example 1:Write an oo program to represent a friend class



```
#include <iostream.h>
////////////////////////////////////////////////////////////////////////
class alpha
{
private:
int data1;
public:
    alpha()
    {
        data1=99;
    }                              //constructor beta is a friend class
friend class beta;
};
////////////////////////////////////////////////////////////////////////
class beta
{                                          //all member functions can access private alpha data
public:
void func1(alpha a) { cout<<"\n data1="<< a.data1; }
void func2(alpha a) { cout<<"\n data1= "<<a.data1; }
};
////////////////////////////////////////////////////////////////////////
int main()
{
alpha a;
beta b;
b.func1(a);
b.func2(a);
cout<<endl;
return 0;
}
```

Example 2:Write an oo program to represent a friend class between point and display classes



```
#include <iostream.h>

class point
{
private:
int xval,yval;
public:
    point(int x,int y)
    {
        xval=x+2;
        yval=y+3;

    }
    friend class display;
};

class display
{
public:
    void disdata(point p)
    {
        cout<<"xval= "<<p.xval<<" "<<"yval= "<<p.yval<<endl;
    }

};


void main( )
{   point p(2,2);
    display d;
        d.disdata(p);
    cout<<endl;
}
```

Example 3: Write an oo program to represent a friend class between two classes (point and D3) .

```
#include <iostream.h>
class point
{
int xVal;
int yVal;
public:
point(int x, int y)
{ xVal=x;
 yVal=y;
}
friend class D3;
};
class D3
{
private:
int  zVal;
public:
D3(int  zz)
{ zVal=zz;
}
void func1(point a)
{ cout << "\nxVal=" << a.xVal; cout << "\nyVal=" << a.yVal; cout <<
"\nzVal=" << zVal; }
};
int main()
{
D3 d1(6);
point d2(3,2);
d1.func1(d2);
cout << endl;
  return 0;
}
```

Output:

```
3
2
6
```

Scope Operator

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
```

The full form uses the binary **scope operator ::** to indicate that OffsetPt is a member of Point.

In some situations, using the scope operator is essential.

For example, the case where the name of a class member is hidden by a local variable (e.g., member function parameter) can be overcome using the scope operator:

```
class Point {
public:
Point (int x, int y) { Point::x = x; Point::y = y; }
//...
private:
int x, y;
}
```

Here **x** and **y** in the constructor (inner scope) hide **x** and **y** in the class (outer scope). The latter are referred to explicitly as **Point::x** and **Point::y.**

Example 1:

Write a C++ program under the concept of class constructor to modeling the Rectangle class, using scope resolution operator with the member function.

Note: all data member are private.

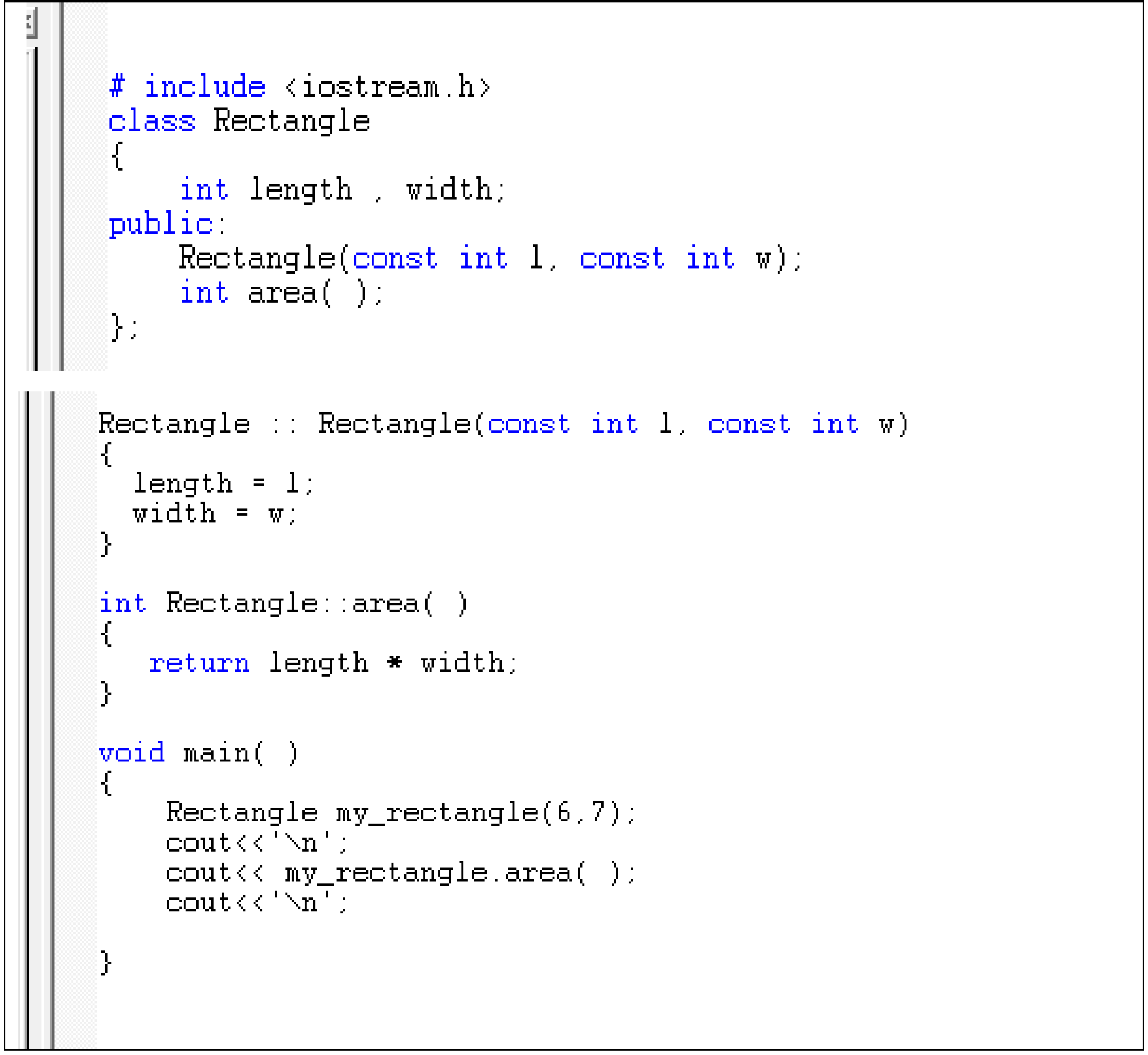
```
# include <iostream.h>

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

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

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

int main( )
{
  Rectangle my_rectangle(6,7);
  cout<< my_rectangle.area( );
  return 0;
}
```

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:

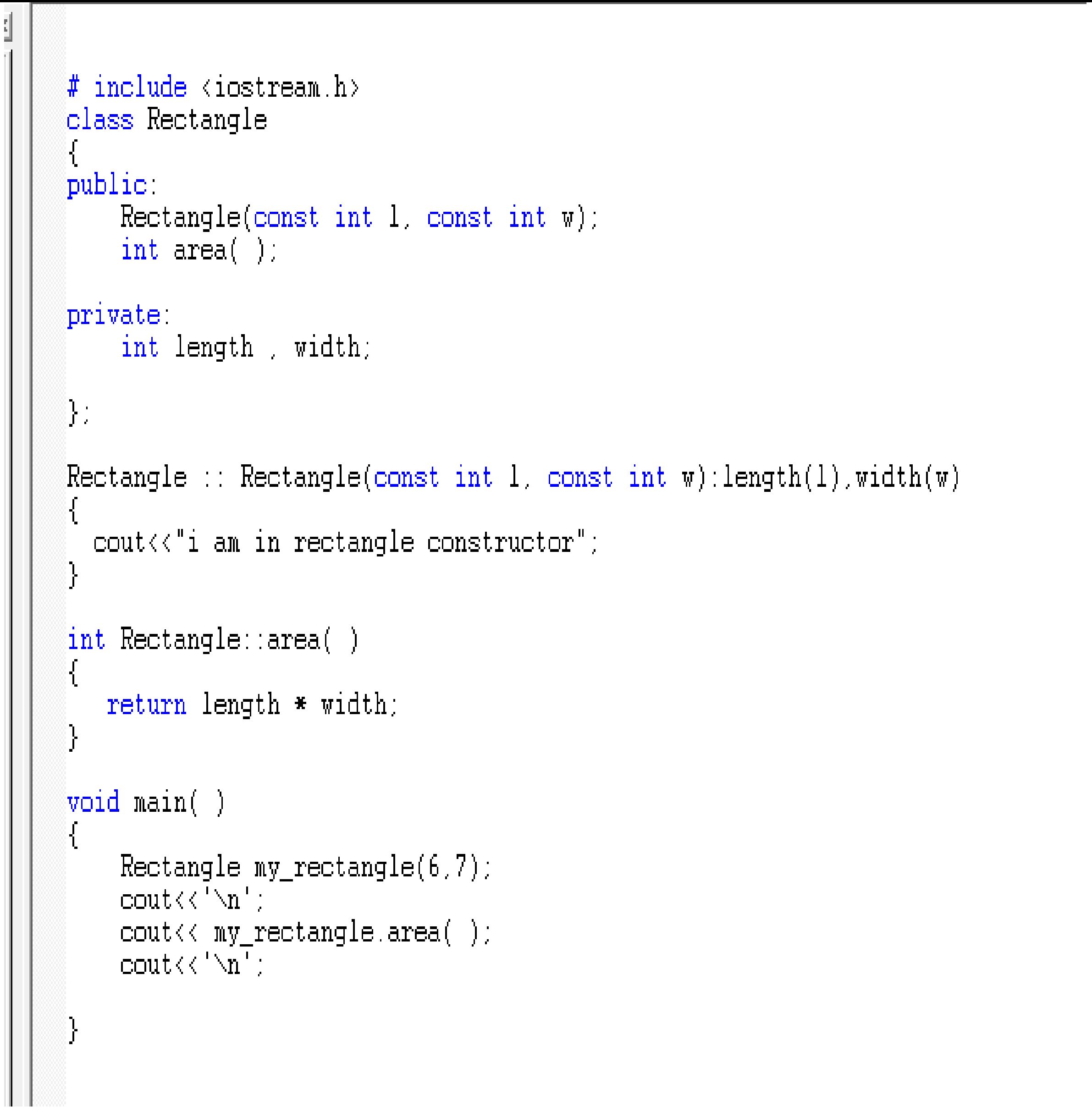
Example 1: Write an oo program to initialize the data member of a class using assignments in the body of constructor.



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

Rectangle :: Rectangle(const int l, const int w)
{
  length = l;
  width = w;
}

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

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

}
```

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

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



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

private:
    int length , width;

};

Rectangle :: Rectangle(const int l, const int w):length(l),width(w)
{
  cout<<"i am in rectangle constructor";
}

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

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

}
```

Constant member

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

```
class Image {
const int width;
const int height;
//...};
```

However, data member constants cannot be initialized using the same syntax as for other constants:

```
class Image {
const int width = 256; // illegal initializer!
const int height = 168; // illegal initializer!
//...
};
```

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

```
class Image
{
public:
Image (const int w, const int h);
private:
const int width;
const int height;
//...     };
Image::Image (const int w, const int h) : width(w), height(h)
{     //...     }
```

As one would expect, no member function is allowed to assign to a constant data member.

Constant Function Argument

If an argument is large, passing by reference is more efficient because, behind the scenes, only an address is really passed, not the entire variable. 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 simple program to represent a constant member

```
//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 oo program to represent a constant member function

```
// constObj.cpp
// constant Distance objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                        //2-arg constructor
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //user input; non-const func
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance; const func
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   const Distance football(300, 0);

// football.getdist();            //ERROR: getdist() not const
   cout << "football = ";
   football.showdist();           //OK
   cout << endl;
   return 0;
   }
```

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 oo program to represent a static members.

```
// statfunc.cpp
// static functions and ID numbers for objects
#include <iostream>

 ////////////////////////////////////////////////////////////////
 class gamma
    {
    private:
       static int total;         //total objects of this class
                                 //   (declaration only)
       int id;                   //ID number of this object
    public:
       gamma()                   //no-argument constructor
          {
          total++;               //add another object
          id = total;            //id equals current total
          }
       ~gamma()                  //destructor
          {
          total--;
          cout << "Destroying ID number " << id  << endl;
          }
       static void showtotal()  //static function
          {
          cout << "Total is " << total << endl;
          }
       void showid()             //non-static function
          {
          cout << "ID number is " << id << endl;
          }
    };
 //--------------------------------------------------------------
 int gamma::total = 0;           //definition of total
 ////////////////////////////////////////////////////////////////
 int main()
    {
    gamma g1;
    gamma::showtotal();

    gamma g2, g3;
    gamma::showtotal();

    g1.showid();
    g2.showid();
    g3.showid();
    cout << "----------end of program----------\n";
    return 0;
    }
```

Now the function can be accessed using only the class name. Here’s the output:

```
Total is 1
Total is 3
ID number is 1
ID number is 2
ID number is 3
----------end of program--------
Destroying ID number 3
Destroying ID number 2
Destroying ID number 1
```

Example 2: Write an oo program to represent a static member.

```
# include <iostream.h>
class test{

static int count;//count is static
int code;
public:
void setcode()
{
    cout<<"i am in set code"<<endl;
    code=++count;
}
void showcode()

{   cout<<"i am in show code"<<endl;

    cout<<"object number"<<code<<"\n";
}
static void showcount()
{   cout<<"i am in showcount"<<endl;

    cout<<"count:"<<count<<"\n";
}
};
int test::count;//count defined
void main ()
{ test t1,t2;
t1.setcode ();
t2.setcode ();
test::showcount ();
test t3;
t3.setcode ();
test::showcount ();
t1.showcode();
t2.showcode();
t3.showcode();
}
```

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 .

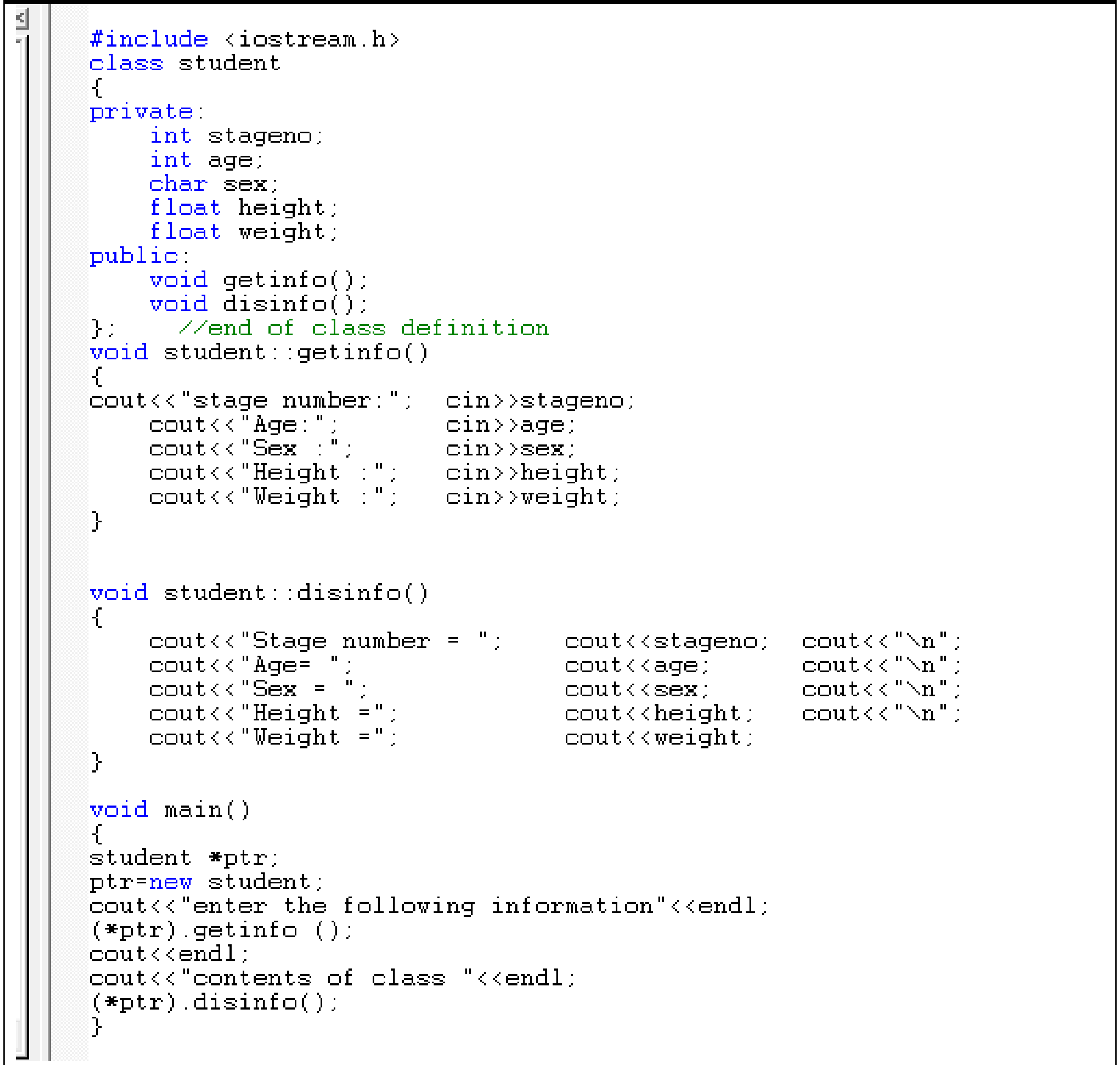
<u>First way :-</u> ***(*object name).member name=variable;***

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;***

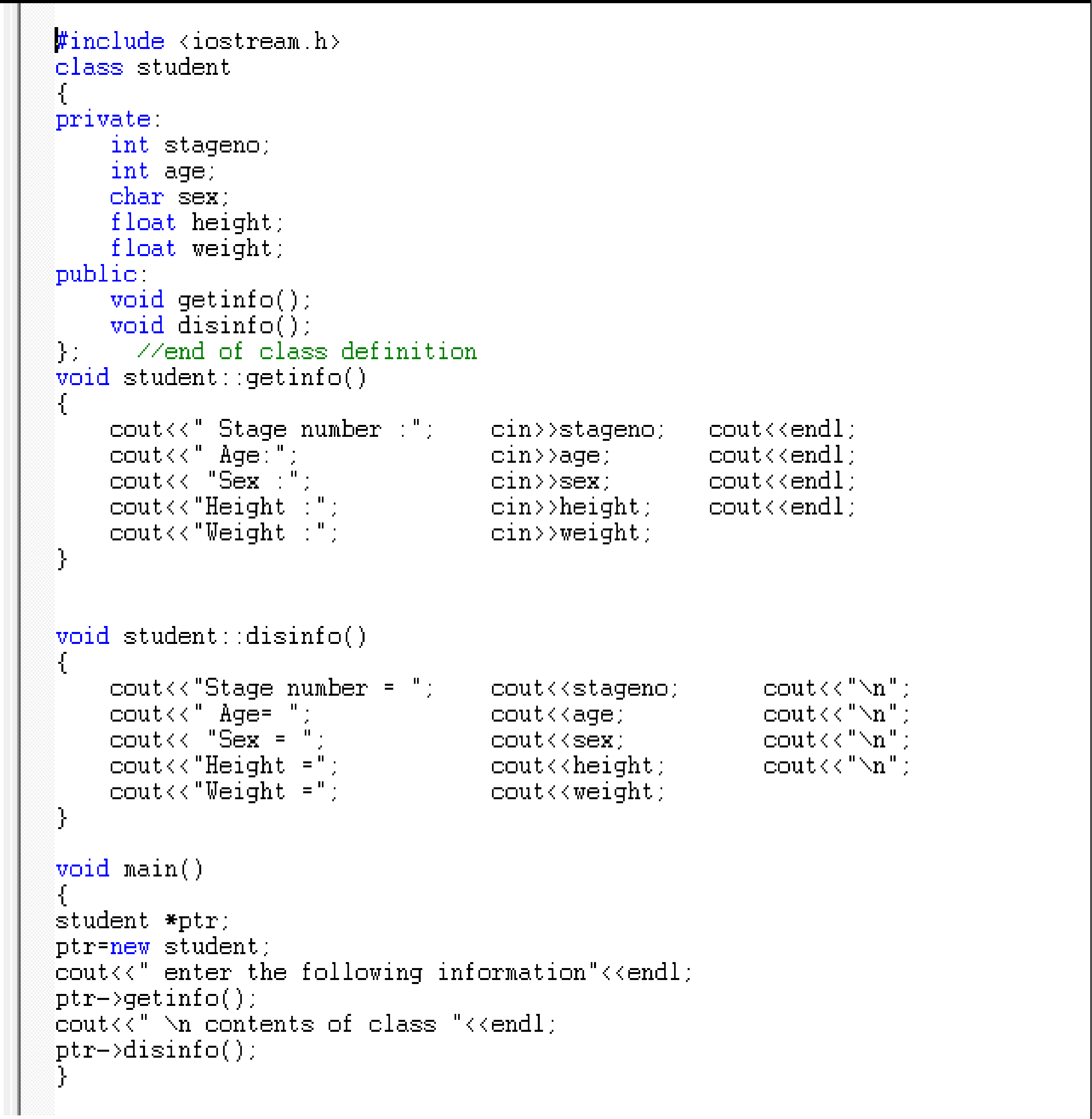
The pointer to the member of a class can be expressed using dash(-) followed by the greater than(>).

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.



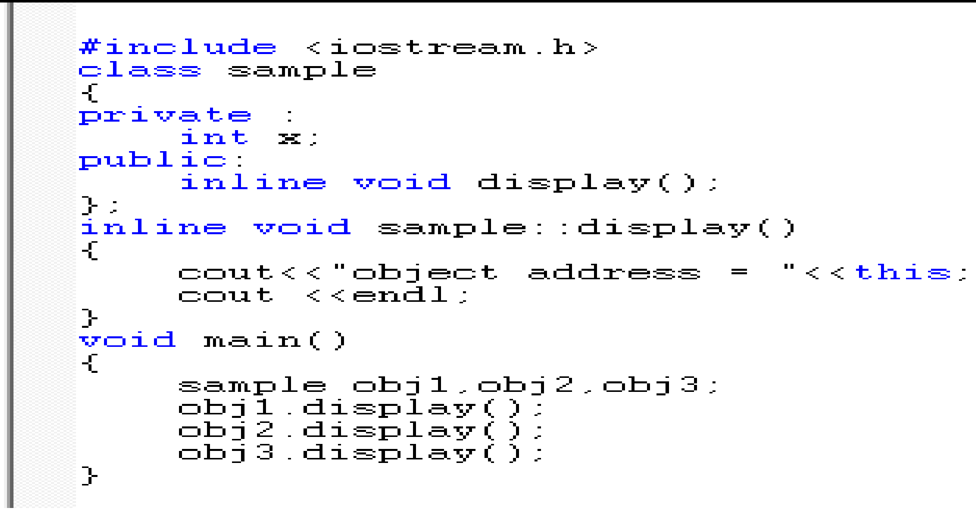
```
#include <iostream.h>
class student
{
private:
    int stageno;
    int age;
    char sex;
    float height;
    float weight;
public:
    void getinfo();
    void disinfo();
};    //end of class definition
void student::getinfo()
{
    cout<<" Stage number :";    cin>>stageno;   cout<<endl;
   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

This pointer

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.

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

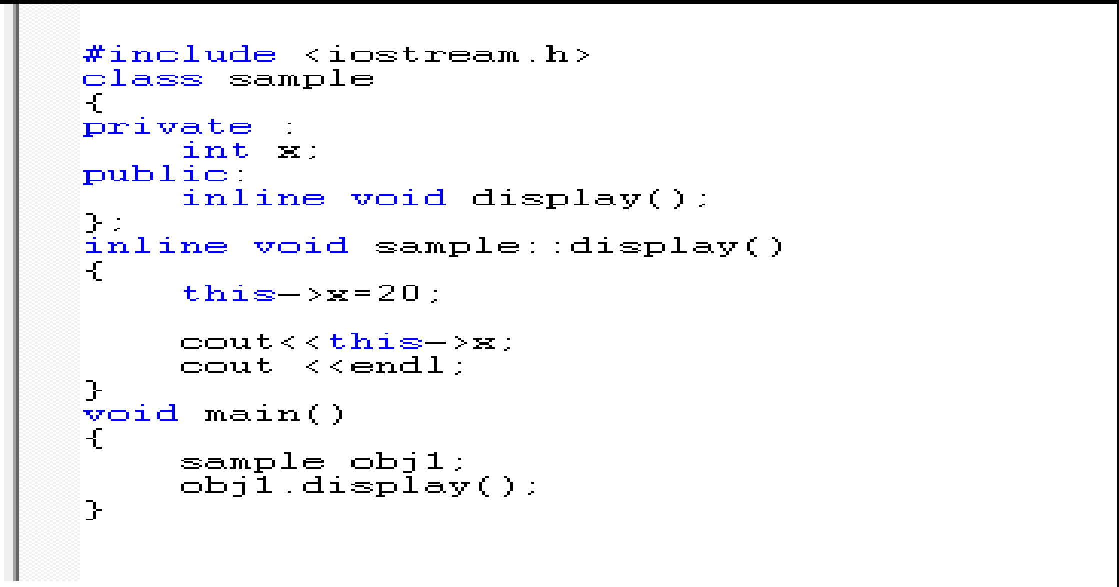


```
#include <iostream.h>
class sample
{
private :
        int x;
public:
        inline void display();
};
inline void sample::display()
{
        cout<<"object address = "<<this;
        cout <<endl;
}
void main()
{
        sample obj1,obj2,obj3;
        obj1.display();
        obj2.display();
   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

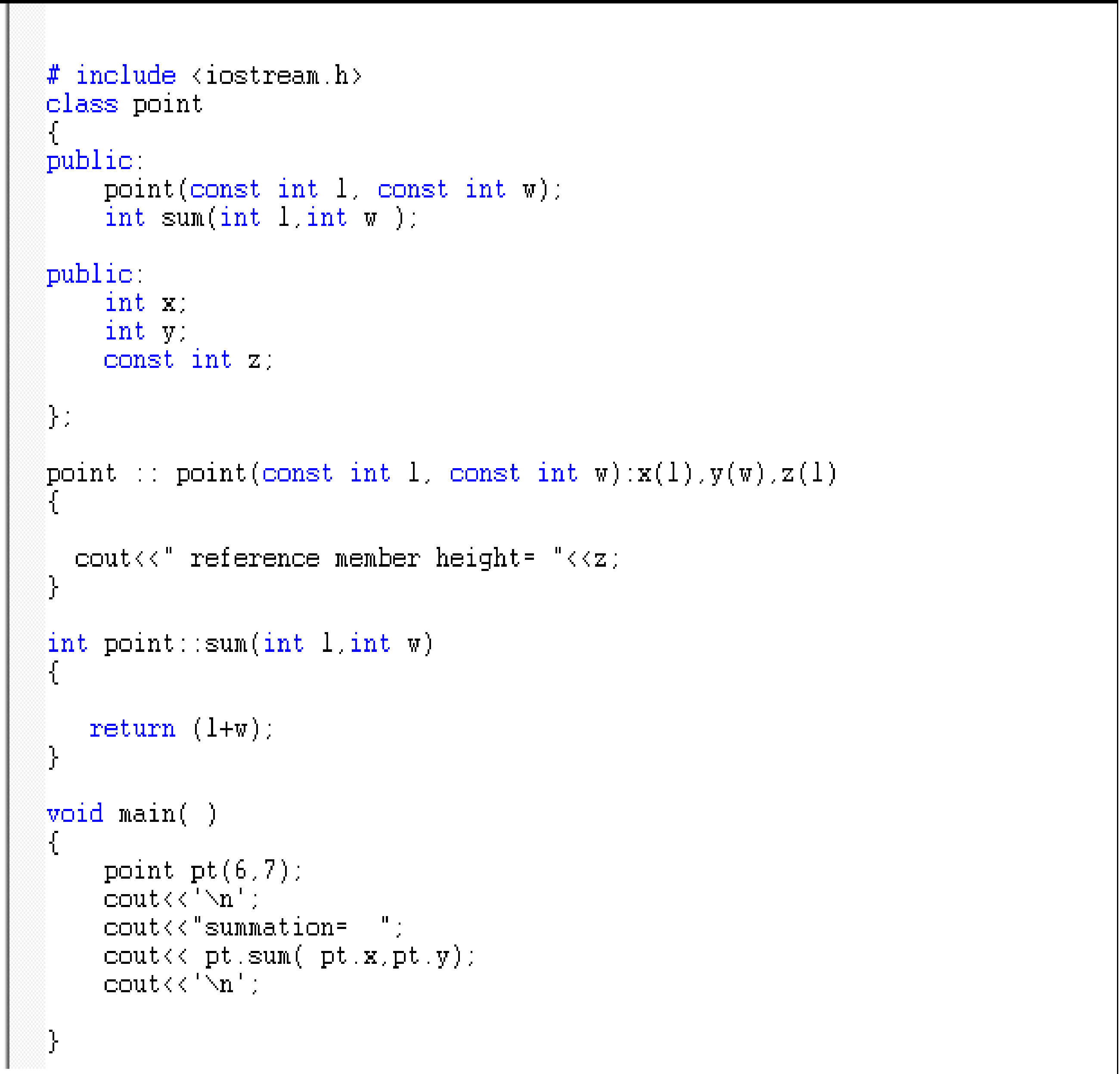
```
class Image {
public:
Image (const int w, const int h);
private:
int  width;
int  height;
int &widthRef;
//...
};
Image::Image (const int w, const int h) : widthRef(width)
{
//...
}
```

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



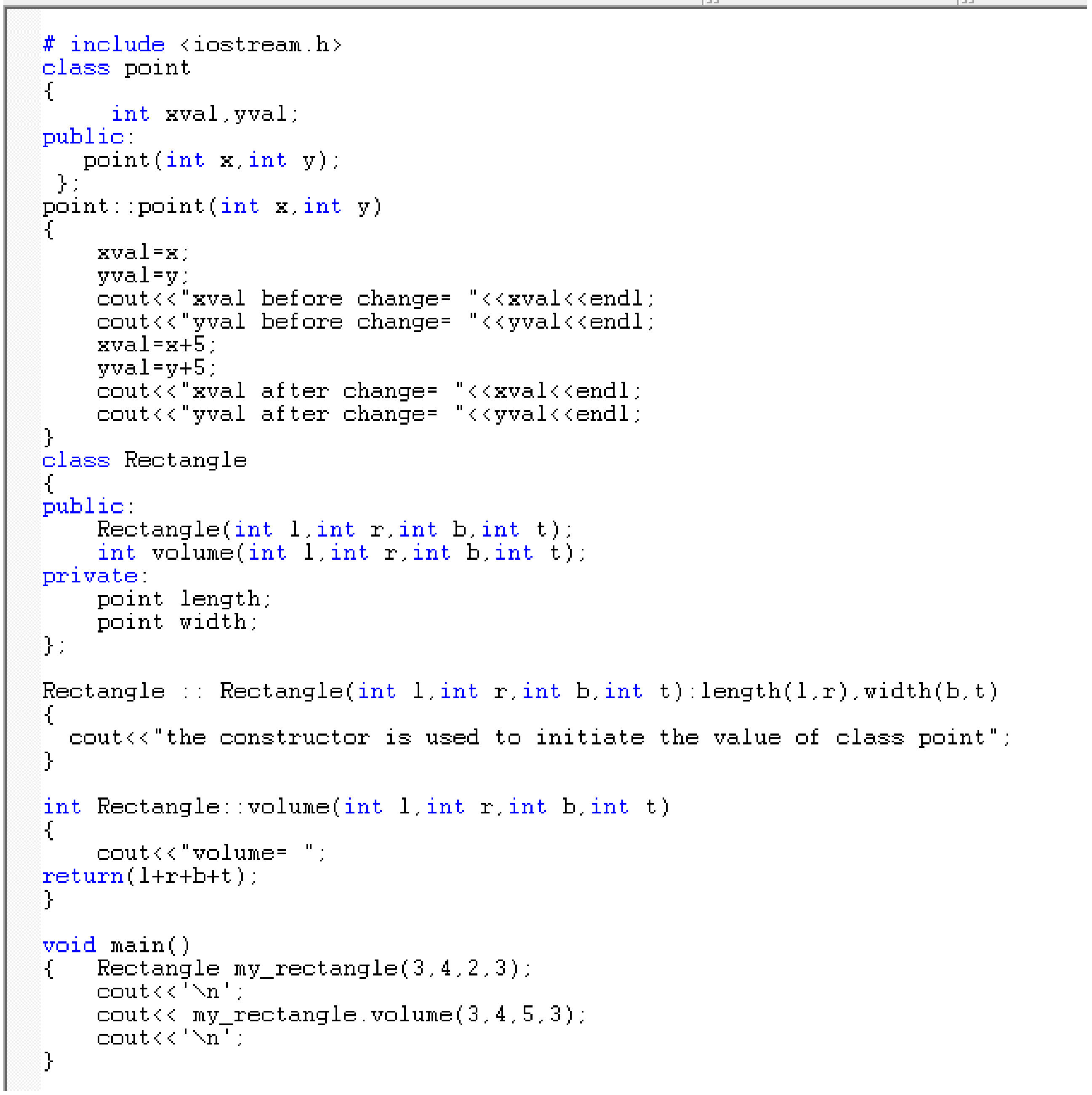
```
# include <iostream.h>
class point
{
public:
     point(const int l, const int w);
     int sum(int l,int w );

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

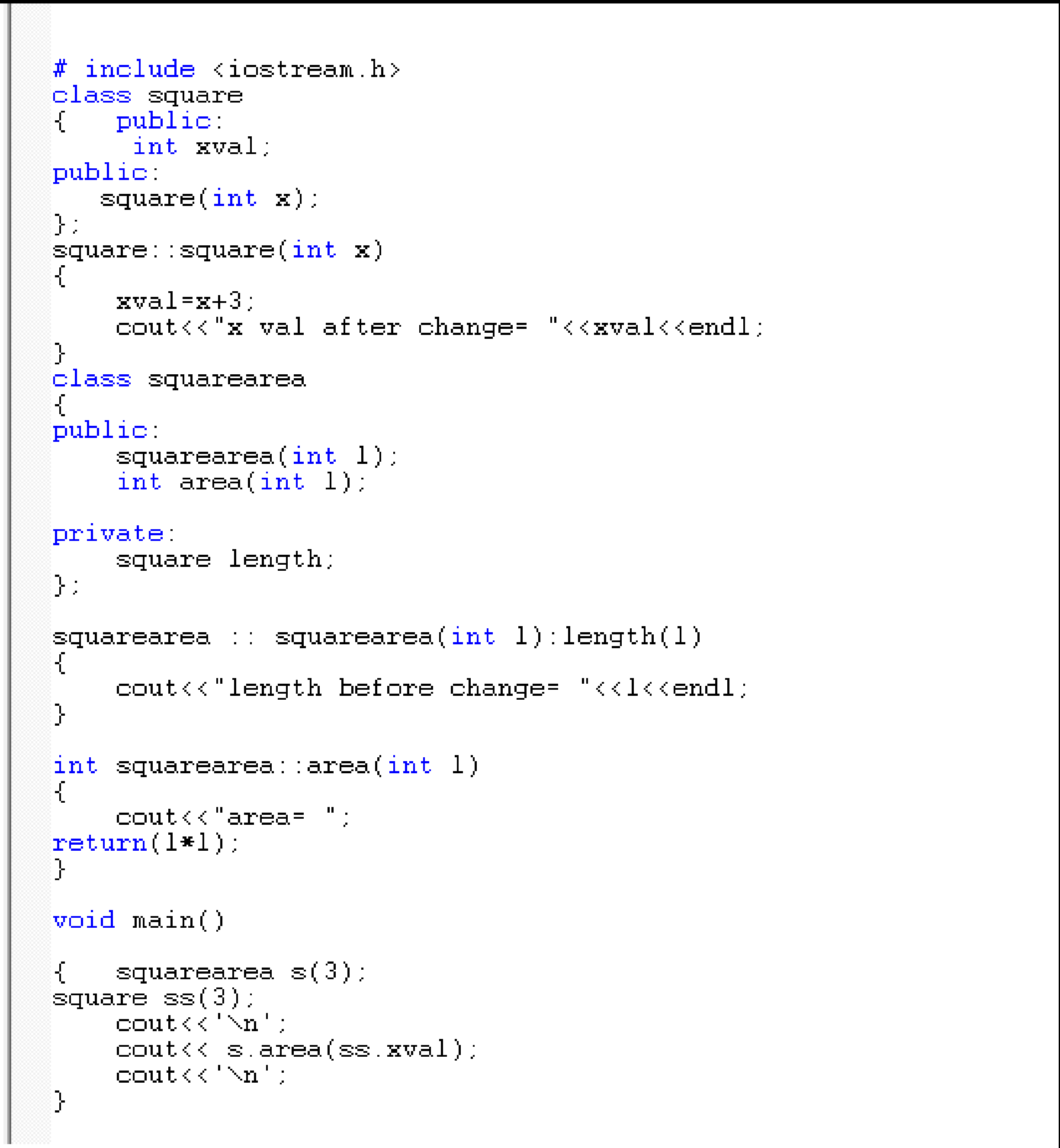
Point data members which represent the top and bottom-right corners of the rectangle:

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.

Figure 1 shows the syntax of an array definition.

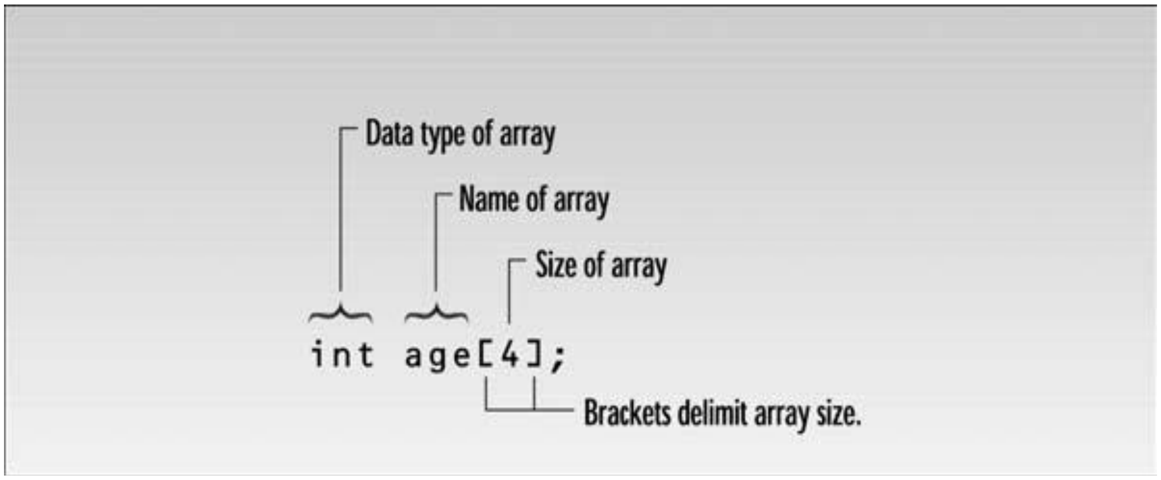


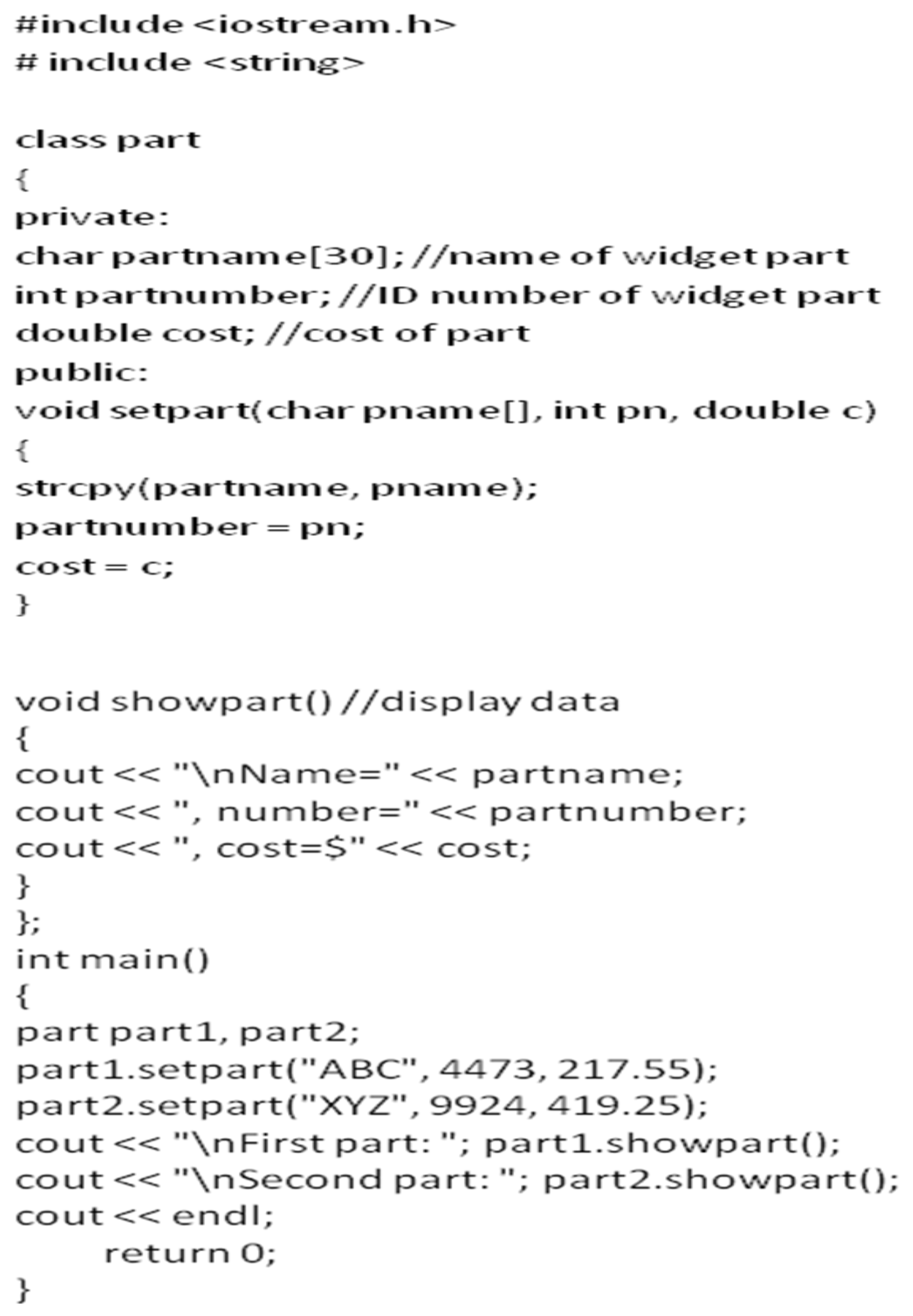
Figure1: syntax of array definition

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;
       }
```

Strings as Class Members



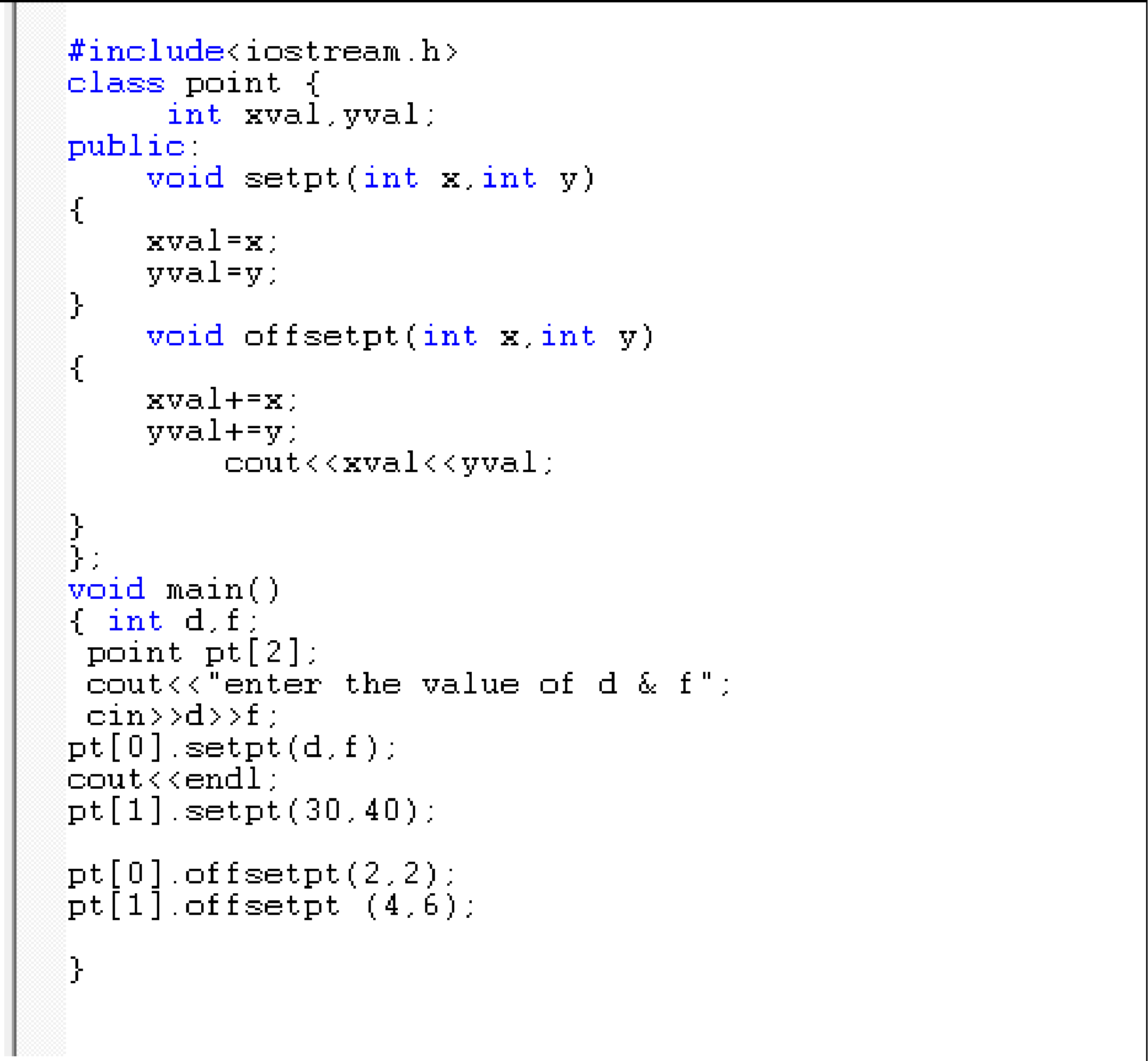
```
#include <iostream.h>
# include <string>

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;
}
```

<u>Object Arrays</u>

- 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 p[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,

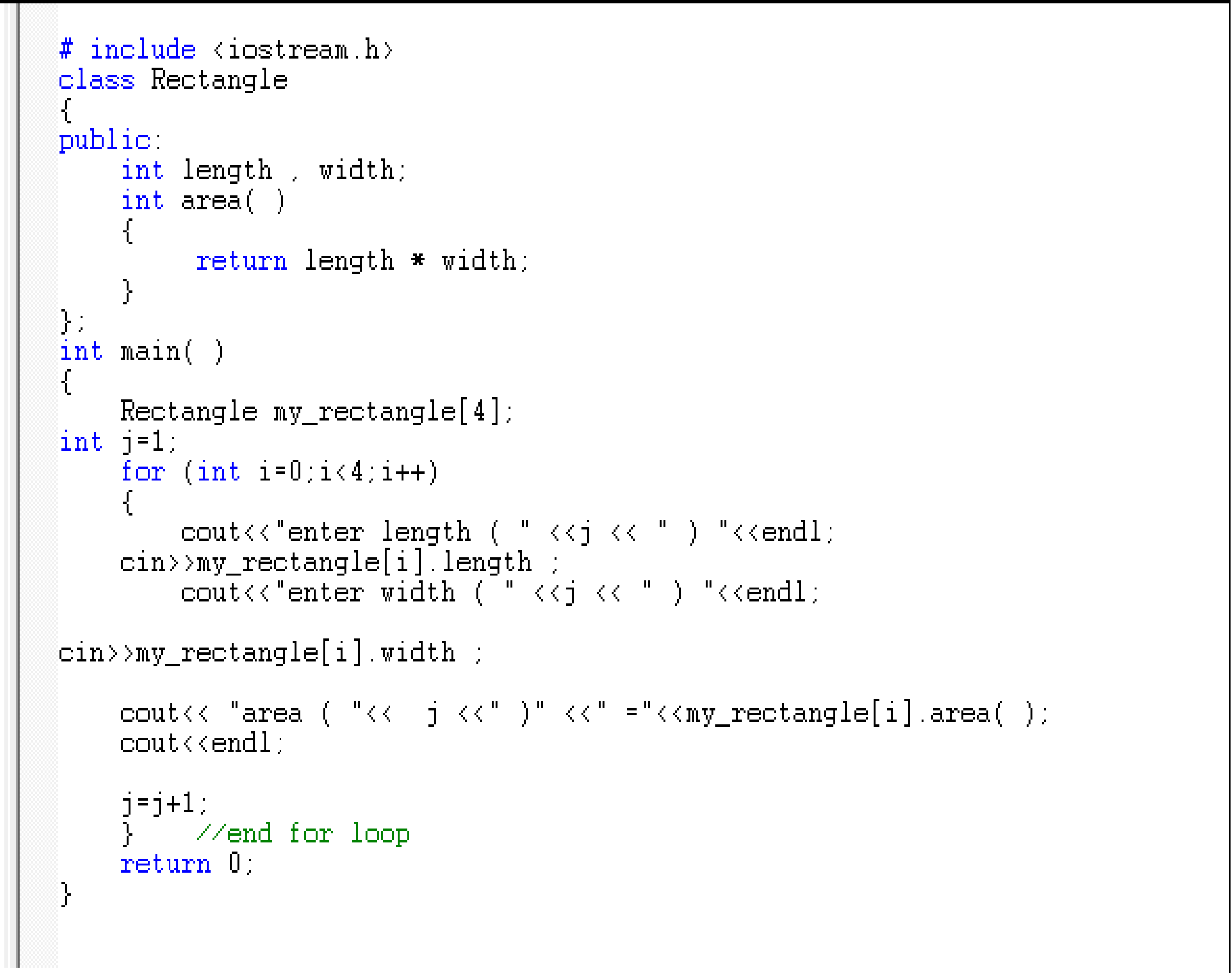
  Point p[5] = { Point(10,20), Point(10,30), Point(20,30), Point(30,20) };
- initializes the first four elements of p to explicit points, and the last element is initialized to (0,0).

Example 1: Write a simple program to represent array of class object point .



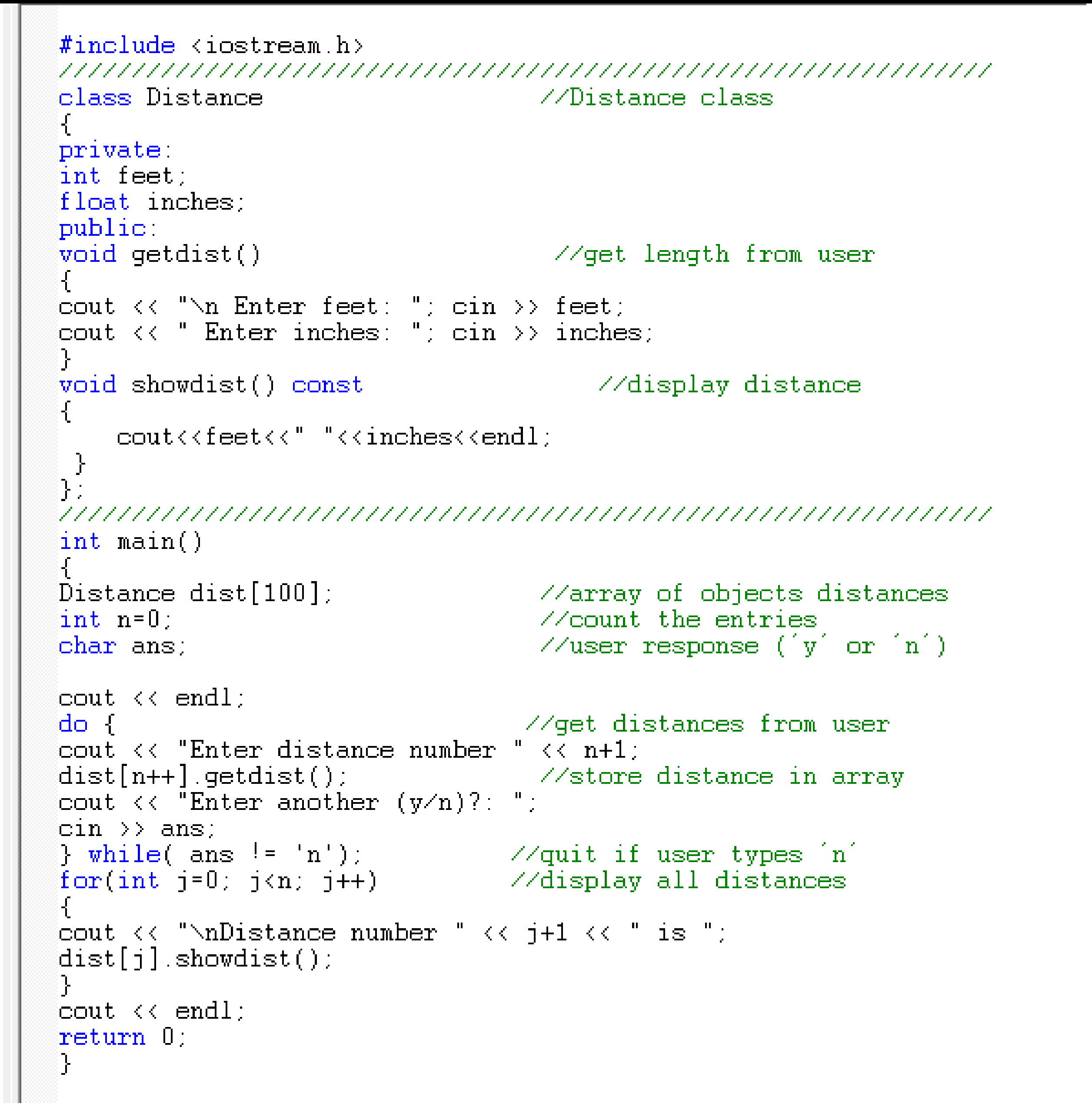
```
#include<iostream.h>
class point {
     int xval,yval;
public:
    void setpt(int x,int y)
{
    xval=x;
    yval=y;
}
    void offsetpt(int x,int y)
{
    xval+=x;
    yval+=y;
        cout<<xval<<yval;

}
};
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()
```