

Kendriya Vidyalaya Sarni

K.V. SARNI

Developed by:
HITESH KUMAR BHABHIWAL
PGT – Computer Science

INDEX

CLASS – XI	[USER DEFINED FUNCTION]	1
	<i>[2 periods]</i>	
	<i>Definition</i>	
	<i>Start With Simple Examples</i>	
	<i>Find the output</i>	
	<i>Some Important Points about Function</i>	
	<i>Why Use Functions</i>	
	<i>Passing Values between Functions</i>	
	<i>The return statement</i>	
	<i>Just Thinking</i>	
	<i>Practical Time</i>	
CLASS – XII	[STRUCTURE]	13
	<i>[2 periods]</i>	
	<i>Definition</i>	
	<i>Declaration of Structure</i>	
	<i>Different ways of declaring structure:</i>	
	<i>Accessing Structure Members</i>	
	<i>Structure Assignments</i>	
	<i>Arrays of Structures</i>	
	<i>Passing Structures to Functions</i>	
	<i>Assignment:</i>	
	<i>Initializing Structure Elements</i>	
	<i>Question from CBSE Board:</i>	
	<i>Practical Assignment</i>	
CLASS – XII	[CLASS]	28
	<i>[2 periods]</i>	
	<i>Definition</i>	
	<i>What is an object?</i>	
	<i>Creating Class</i>	
	<i>Structures and Classes Are Related</i>	
	<i>Conclusion:</i>	
	<i>Flash Back</i>	
	<i>Question from CBSE Board:</i>	

CLASS - XI

USER DEFINED FUNCTION

Large programs are generally avoided because it is difficult to manage a single list of instructions. Thus, a large program is broken down into smaller units known as function.

DEF: *A Function is a subprogram that acts on data and often returns a value.*

Look at a simple example of function:

```
main ( )
{
    message();
    cout << "Hello, how are you feeling?";
}

message( )
{
    cout << "Smile, and the world smiles with you
... \n";
}
```

And here's the output...

```
Smile, and the world smiles with you ...
Hello, how are you feeling?
```

Here, through main() we are calling the function message(). What does mean when we say that main() 'calls' the function message()? We mean that the control passes to the function message(). The activity of main() is temporarily suspended; it falls asleep while the message() function wakes up and goes to work. When the message() function runs out of statements to execute, the control returns to

main(), which comes to life again and begins executing its code at the exact point where it left off. Thus, main() becomes the 'calling' function, whereas message() becomes the 'called' function,

Let us spend some more time on simpler functions. If you have grasped the concept of 'calling' a function you are prepared for a call to more than one function. Consider the following example:

```
main()
{
    cout << "I am in main"<< endl ;
    italy(<< endl ;
    brazil(<< endl ;
    argentina(<< endl ;
}
italy( )
{
    cout << "I am in italy" << endl ;
}
brazil( )
{
    cout << "I am in brazil" << endl ;
}
argentina( )
{
    cout << "I am in argentina" << endl;
}
```

The output of the above program when executed would be as under:

```
I am in main
I am in italy
I am in brazil
I am in argentina
```

Find the output of following program:

```
(a)  main( )
      {
          cout << "I am in main" << endl ;
          italy(<< endl ;
          cout << "I am back in main" ;
      }
      italy( )
      {
          cout << "I am in italy" << endl ;
      }
```

And the output would look like ...

```
I am in main
I am in italy
I am back in main
```

```
(b)  main( )
      {
          cout << "I am in main" << endl ;
          italy( ) ;
          cout << "I am finally back in main" ;
      }
      italy( )
      {
          cout << "I am in italy" << endl ;
          brazil ( ) ;
          cout << "I am back in italy" << endl ;
      }
      brazil( )
      {
          cout << "\nI am in brazil" << endl ;
          argentina( ) ;
      }
      argentina( )
      {
          cout << "I am in argentina" << endl ;
      }
```

And the output would look like ...

```
I am in main
I am in italy
I am in brazil
I am in argentina
I am back in italy
I am finally back in main
```

Here, main() calls other functions, which in turn call still other functions. Trace carefully the way control passes from one function to another. Since the compiler always begins the program execution with main(), every function in a program must be called directly or indirectly by main(). In other words, the main() function drives other functions.

Some important points about function

- a) A C++ program is a collection of one or more functions.
- b) A function gets called when the function name is followed by a semicolon. For example,

```
main()
{
    argentina ();
}
```

- c) A function is defined when function name is followed by a pair of braces in which one or more statements may be present. For example,

```
argentina ( )
{
    statement 1 ;
    statement 2 ;
    statement 3 ;
}
```

- d) Any function can be called from any other function. For example,
- ```
main ()
```

```

 {
 message () ;
 }
message ()
{
 cout << "Can't imagine life without C++" ;
 message1 () ;
}
message1 ()
{
 cout << "You can imagine only with OOPs"
;
}

```

- e) A function can be called any number of times. For example,

```

main()
{
 message() ;
 message() ;
}
message()
{
 cout << "Turbo C++" ;
}

```

- f) The order in which the functions are defined in a program and the order in which they get called need not necessarily be the same. For example,

```

main()
{
 message1 () ;
 message2 () ;
}
message2()
{
 printf ("\nBut the butter was bitter") ;
}

```

```
message1 ()
{
 printf ("\nMary bought some butter");
}
```

Here, even though message1( ) is getting called before message2(), still, message1( ) has been defined after message2( ). However, it is advisable to define the functions in the same order in which they are called. This makes the program easier to understand.

- g) A function can be called from other function.
- h) There are basically two types of functions:
  - i. Library functions Ex. pow( ), sqrt( ) etc.
  - ii. User defined functions Ex. argentina( ), brazil( ) etc.

As the name suggests, library functions are nothing but commonly required functions grouped together and stored in a file that is called a Library. This library of functions is present on the disk and is written for us by people who write compilers for us. Almost always a compiler comes with a library of standard functions. The procedure of calling both types of functions is exactly same.

### **Why Use Functions**

Why write separate functions at all? Why not squeeze the entire logic into one function, main ( )? Two reasons:

- a) Writing functions avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of a triangle. If, later in the program, you want to calculate the area of a different triangle, you won't like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a 'section of code' that calculates area and then jump back to the place from

where you left off. This section of code is nothing but a function.

- b) Using functions, it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

### Passing Values between Functions

Consider the following program. In this program, in `main( )` we receive the values of `a`, `b` and `c` through the keyboard and then output the average of `a`, `b` and `c`. However, the calculation of average is done in a different function called **`calcAvg( )`**. If average is to be calculated in **`calcAvg( )`** and values of `a`, `b` and `c` are received in **`main( )`**, then we must pass on these values to **`calcAvg( )`**, and once **`calcAvg( )`** calculates the average we must return it from **`calcAvg( )`** back to **`main( )`**.

```

/* Sending and receiving values between functions */
main ()
{
 int a, b, c;
 float p;
 cout << "Enter any three numbers" ;
 cin >> a >> b >> c;
 p = calcAvg (a, b, c) ;
 cout << "Average = " << p ;
}

```

Parameters or Arguments also known as actual parameter

```

Function definition {
float calcAvg (int x, int y, int z)
{
 int d;
 d=x+y+z;
 float t = d/3.0 ;
 return (t) ;
}

```

Function header or Formal parameter

And here is the output...

```

Enter any three numbers
10
20
30
Average = 20 .

```

There are a number of things to note about this program:

- In this program, from the function main( ) the values of a, b and c are passed on to the function calcAvg(), by making a call to the function calcAvg( ) and mentioning a, band c in the parentheses:  
`p = calcAvg ( a, b, c ) ;`
- In the calsum( ) function these values get collected in three variables x, y and z:  
`calcAvg ( int x, int y, int z )`
- The variables a, band c are called '**actual arguments**', whereas the variables x, y and z are called '**formal arguments**'. Any number of arguments can be passed to a function being called. However, the type, order and number of the actual and formal arguments must always be same.
- Instead of using different variable names x, y and z, we could have used the same variable names a, band c. But the compiler would still treat them as different variables since they are in different functions.

In the earlier programs the moment closing brace ( } ) of the called function was encountered the control returned to the calling function. No separate return statement was necessary to send back the control.

This approach is fine if the called function is not going to return any meaningful value to the calling function. In the above program, however, we want to return the average of x, y and z. Therefore, it is necessary to use the return statement.

**The return statement serves two purposes:**

- a) On executing the return statement, it immediately transfers the control back to the calling program.
- b) It returns the value present in the parentheses after return to the calling program. In the above program, the value of average of three numbers is being returned.

# Just Start THINKING...

**Point out the errors, if any, in the following programs:**

Page | 12

```
a) main()
 {
 int i = 135, a = 135, k ;
 k = pass (i, a) ; .
 cout << k ;
 }
 int pass (int i, int b)
 {
 c = i + b;
 return (c) ;
 }
```

```
b) main()
 {
 int i = 135, a = 135, k ;
 k = pass (i, a) ; .
 cout << k ;
 }

 pass (int i, int b)
 {
 int c;
 c = i + b;
 return c ;
 }
```

```
c) main()
 {

 int p = 23, f = 24 ;
 fun (p, f) ;
 cout << p << f;
 }
 int fun(int p, int f)
 {
 return p+f;
 }
```

```
d) main()
 {
 int k = 35, z ;
 z = check (k) ;
 cout << z;
 }
int check (m)
{
 if(m>40)
 return (1) ;
 else
 return (0) ;
}
```

```
e) main()
 {
 int i = 35, z ;
 z = function (i) ;
 cout << z;
 }
int function (int m) ;
{
 return (m + 2) ;
}
```

```
f) main()
 {
 int i = 35 ;
 function (i) ;
 cout << z;
 }
void function (int m)
{
 z = m*10;
}
```

# Don't Go....

## Practical Time

1. A 4 digit positive integer is entered through the keyboard, write a function to calculate sum of digits of the 5 digit number:
2. A positive integer is entered through the keyboard; write a program to obtain the prime factors of the number.
3. Write a function IsLeapYear( ). It receives a 4 digit year and returns 1 if year is leap year otherwise returns 0.
4. Write a function ConvertToUpper() that receives a character and convert received character in uppercase and return it.
5. Write a function Reverse ( ) it accepts a long integer number and returns its reverse number.
6. Write a function IsPalindrome (long) that uses function Reverse ( ) mentioned in Q. 5. It returns 1 if given number is palindrome otherwise return 0.
7. Write a function CompoundInterest ( ) receives three float values and calculate compound interest and return interest in double .
8. Write a function that receives three integer number and return the largest value.
9. Write a function Factorial ( ) that receives an integer values, calculate factorial in the form of long of given integer values and return it.
10. With the help of above function Factorial ( ) write a function SUM() that receives two parameters X and N and returns the sum of following series:

$$X + \frac{X^3}{3!} + \frac{X^5}{5!} + \dots + \frac{X^{2N-1}}{(2N-1)!}$$

K.V. SARMI

## CLASS - XII

### STRUCTURE

A structure is a grouping of several variables under one name and is sometimes called conglomerate data type.

**DEF:** *Structure is a set of variables reference under one name.*

Or

*A structure specifies the grouping of various data items into a single unit.*

#### **Declaration of Structure**

```
struct StructureName
{
 DataType variable(s);
 DataType variable(s);
 . . .
 DataType variable(s);
};
```

These are collection of variables

**Keyword** – it is reserve work. We have to write “**struct**” before declaring structure data type.

Structure declaration terminated by ; (semicolon). If we forget to put semicolon at end of structure declaration computer gives error.

Examples:

```
struct Distance
{
 int feet, inches;
```

```

};

void main()

{ Distance length, breadth;

 ...

}

```

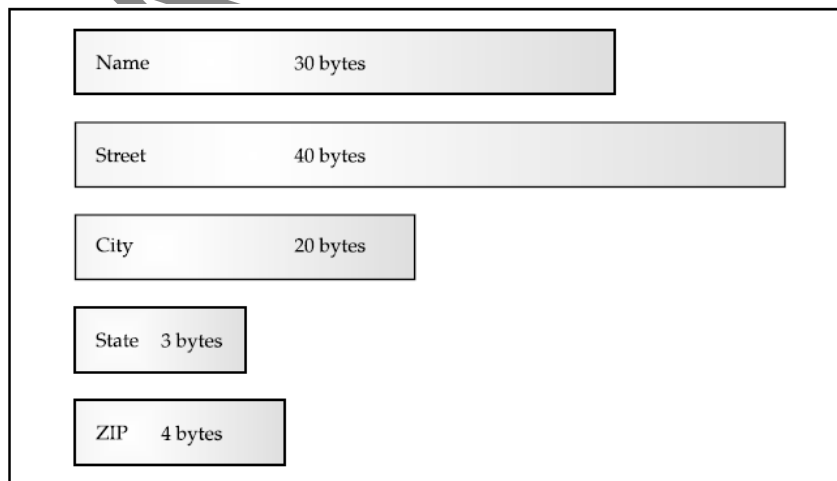
In the above example Distance is user defined data type. It consists of two variable feet and inches. In main function we have two variable named length and breadth of Distance type. Variable length and breadth consists from two variables feet and inches. Now variable length has tow distances feet (2 bytes) and inches (2 bytes), breadth also has two distance same as length. The size of length is 4 bytes ( feet ->2 bytes + inches ->2 bytes)

Let us take another example:

```

struct ADDRESS
{
 char name[30];
 char street[40];
 char city[20];
 char state[3];
 unsigned long int zip;
} addr_info, binfo, cinfo;

```



Now, it is very easy to combine variables into a single unit i.e. structure. In above mention example variable s has complete record of a student.

The declaration of a structure will not serve any purpose without its definition. It only acts as a blueprint for the creation of variables as mention in above examples. The structure definition creates structure variables and allocates storage space for them.

### Different ways of declaring structure:

#### ▣ First way

```
struct Student
{
 int rollno;
 char name[20];
 char subject[20];
 int marks[5],total;
 float percentage;
}; ← Variable is omitted
```

#### ▣ Second way

```
struct Student
{
 int rollno;
 char name[20];
 char subject[20];
 int marks[5],total;
 float percentage;
} s1, s2, s3;
```

#### ▣ Third way (**Anonymous structure**)

```
struct ← Structure Name is omitted
```

```

 { int rollno;
 char name[20];
 char subject[20];
 int marks[5],total;
 float percentage;
 } s1, s2, s3;

```

In this type of declaration, we cannot declare more variable in future.

- ◆ *We can omit either Structure Name or variable(s), but not both.*

### Accessing Structure Members

Individual members of a structure are accessed through the use of the ‘.’ operator (usually called the *dot operator* or *period period*).

For example, the following code assigns the ZIP code 12345 to the **zip** field of the structure variable **addr\_info** declared earlier:

```
addr_info.zip = 12345;
```

The structure variable name followed by a period and the member name references that individual member. The general form for accessing a member of a structure is

structure-name.*member-name*

Therefore, to print the ZIP code on the screen, write

```
cout << addr_info.zip ;
```

This prints the ZIP code contained in the **zip** member of the structure variable **addr\_info**

In the same fashion, the character array **addr\_info.name** can be used to call **gets()** , as shown here:

```
gets(addr_info.name);
```

This passes a character pointer to the start of **name**. Since **name** is a character array, you can access the individual characters of **addr\_info.name** by indexing **name**. For example, you can print the contents of **addr\_info.name** one character at a time by using the following code:

```
int t;
for(t=0; addr_info.name[t] != NULL; ++t)
 cout << addr_info.name[t];
```

### Structure Assignments

The information contained in one structure may be assigned to another structure of the same type using a single assignment statement. That is, you do not need to assign the value of each member separately. The following program illustrates structure assignments:

```
struct DISTANCE
{
 int feet, inches;
};
void main()
{
 DISTANNCE a,b;
 a.feet=10;
 a.inches=7;
 b=a;
 cout << b.feet << '\t' << b.inches;
}
}
```

And here's the output...

```
10 7
```

Lets us take one more example

```
struct ABC
{
 int a, b;
};

struct XYZ
{
 int a, b;
};

void main()
{
 ABC ob1;
 XYZ ob2 ;
 ob1.a = 5 ;
 ob1.b = 8;

 ob2 = ob1 ; //error
}
```

The above code fragment will produce an error because ob1 and ob2, though have similar elements but are of *different type* ABC and XYZ respectively, and hence, *they cannot be assigned to one another*.

Assignment:

*Write a program that swaps two same type of structure.*

### **Arrays of Structures**

Perhaps the most common usage of structures is in arrays of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type **ADDRESS**, write

```
struct ADDRESS
{
 char name[30];
 char street[40];
 char city[20];
 char state[3];
 unsigned long int zip;
};
```

```
ADDRESS add[100];
```

This creates 100 sets of variables that are organized as defined in the structure ADDRESS.

To access a specific structure, index the structure name. For example, to print the ZIP code of structure 3, write

```
cout << add[2].zip ;
```

Like all array variables, arrays of structures begin indexing at 0.

To display all NAME and ZIP of all 100 cities following code will be used.

```
for (int I = 0 ; I < 100 ; ++I)
{
 cout << add [I].name << '\t';
 cout << add [I].zip << '\n';
}
```

Assignment:

*Write a program that adds 100 of distances.*

### **Passing Structures to Functions**

This section discusses passing structures and their members to functions.

### Passing Structure Members to Functions

When you pass a member of a structure to a function, you are actually passing the value of that member to the function. Therefore, you are passing a simple variable (unless, of course, that element is compound, such as an array). For example, consider this structure:

```
struct fred
{
 char x;
 int y;
 float z;
 char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x); /* passes character value of x */
func2(mike.y); /* passes integer value of y */
func3(mike.z); /* passes float value of z */
func4(mike.s); /* passes address of string s */
func(mike.s[2]); /* passes character value of s[2] */
```

One more example:

```
struct DISTANCE
{
 int feet, inches;
};
void showDistance(int F, int I)
{
 cout << F << '\t' << I << endl;
}
void main()
{
 DISTANNCE a;
 a.feet=10;
```

```
a.inches=7;

showDistance(a.feet, a.inches);

showDistance(a.inches, a.feet);
}
```

K.V.SARMI

### Passing Entire Structures to Functions

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that any changes made to the contents of the structure inside the function to which it is passed do not affect the structure used as an argument.

When using a structure as a parameter, remember that the type of the argument must match the type of the parameter. For example, in the following program both the argument `arg` and the parameter `parm` are declared as the same type of structure.

```
struct DISTANCE
{
 int feet, inches;
};
void showDistance(DISTANCE obj)
{
 cout << obj.feet << '\t' << obj.inches ;
}
void main()
{
 DISTANNCE a;
 a.feet=10;
 a.inches=7;

 showDistance(a);
}
```

As this program illustrates, if you will be declaring parameters that are structures, you must make the declaration of the structure type global so that all parts of your program can use it. For example, had `struct_type(DISTANCE)` been declared inside `main()` (for

example), then it would not have been visible to showDistance()).

As just stated, when passing structures, the type of the argument must match the type of the parameter. It is not sufficient for them to simply be physically similar; their type names must match. For example, the following version of the preceding program is incorrect and will not compile because the type name of the argument used to call showDisance() differs from the type name of its parameter.

```
/* This program is incorrect and will not compile. */
struct DISTANCE
{
 int feet, inches;
};

struct DISTANCE_M
{
 int mtr, cm;
};

void showDistance(DISTANCE obj)
{
 cout << obj.feet << '\t' << obj.inches ;
}

void main()
{
 DISTANNCE_M a;
 a.mtr = 110;
 a.cm = 77;

 showDistance(a);
}
```

Assignment:

- Write a program that adds two distances using function.
- Write a program that returns larger distance using function.

### Initializing Structure Elements

```
struct DISTANCE
{
 int feet, inches;
};
void main()
{
 DISTANCE ob = { 7, 5 };
 .
 .
}
```

Structure DISTANCE's variable ob is being declared and initialized simultaneously. The values to be assigned to the structure members are surrounded by brace and separated by commas. The first value is assigned to the first member of DISTANCE i.e. feet and the second value is assigned to the second member i.e. inches.

Question from CBSE Board:

- Give the output of the following program:

```
#include<iostream.h>
struct Pixel
{
 int C, R;
};
void Display (Pixel P)
{
 cout << "Col" << P.C;
 cout << "Row << P.R<< endl;
}
void main()
```

```

 {
 Pixel X = { 40, 50 }, Y, Z;
 Z = X;
 X.C += 10 ;
 Y = Z ;
 Y.C += 10;
 Y.R += 20;
 Z.C -= 15;
 Display (X) ;
 Display (Y) ;
 Display (Z) ;
 }

```

[3 marks, 2003]

- Rewrite the corrected code for the following program. Underline each correction (if any)

```

#include <iostream.h>
structure Supergym
{
 int member number;
 char membername[20];
 char membertype[] = "HIG";
};
void main()
{
 Supergym person1, person.2;
 cin<<"Member Number:";
 cin>>person1.membernumber;
 cout<<"Member Name :";
 cin>>person1.membername;
 person1.member type = "MIG";
 person2 = person1;
 cin<<"Member
Number:"<<person2.membernumber;
 cin<<"Member
Name"<<person2.membername;
 cin<<"Member
Number:"<<person2.membertype;

```

}

[3 marks, 2004]

- Find the output of the following program:

```
#include<iostream.h>
struct MyBox
{
 int Length, Breadth, Height;
};
void Dimension (MyBox M)
{
 cout<<M.Length<<"x"<<M.Breadth<<"x";
 cout<<M.Height<<endl;
}
void main()
{
 MyBox B1={10,15,5}, B2, B3;
 ++B1.Height;
 Dimension(B1);
 B3 = B1;
 ++B3.Length;
 B3.Breadth++;
 Dimension(B3);
 B2 = B3;
 B2.Height+=5;
 B2.Length--;
 Dimension(B2);
}
```

[3 marks, 2005]

- Rewrite the corrected code for the following program.  
Underline each correction (if any)

```
#include <iostream.h>
void main()
{
 struct movie
 {
 char movie_name[20];
```

```

 char movie_type;
 int ticket_cost=0;
 } MOVIE;
 gets (movie_name) ;
 gets (movie_type) ;
}

```

[2 marks, 2006]

### Practical Assignment

1. Assume an array S containing elements of structure Student is required to be arranged in descending order of Marks. Write a C++ function MeritList() to arrange the same with the help of bubble sort. The array and its size is required to be passed as parameters to the function. Definition of structure Student is as follows:

```

struct Student
{
 int RollNo;
 char Name[20];
 float Marks;
};

```

2. Assume an array S containing elements of structure Student. Write a C++ function Topper () that displays the detail of those student(s) who got highest marks.

```

struct Student
{
 int RollNo;
 char Name[20];
 float Marks;
};

```

3. Assume an array S containing elements of structure Student. Write a C++ function Display() that displays the detail of those student(s) her city name is Sarni.

*Note:* City name may in any case. It does not affect the result.

```

struct Student

```

```
 { int RollNo;
 char Name[20];
 float Marks;
 char city[20];
 };
```

4. Assume an array S[40] containing elements of structure Student. Write a C++ function Statistics() that displays the statistics in following format:

```
0 - 40 a
41 - 59 b
60 - 79 c
80 - 100 d
```

Here a, b, c and d are number of student.

```
struct Student
{ int RollNo;
 char Name[20];
 float Marks;
};
```

## CLASS - XII

### CLASS

**DEF:** *A class is a way to bind the **data** describing an entity and its associated **function** together.*

Or

*In object oriented programming, data and its associated function are enclosed in one single entity is called class.*

Or

*The class is used to define the nature of an object, and it is C++'s basic unit of encapsulation.*

Or

*The class is a collection of objects of same type.*

For example: furniture, chair, table, stool and sofa. Here **furniture** is a class and chair, table, stool and sofa are the objects of class furniture.

Classes are created using the **keyword class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an *object* is an instance of a class.

What is an object?

*Object is an instance of a class.*

Or

*It is the variable of class type.*

Or

*Any real world entity is known as object.*

### **Creating Class**

```
class class-name
{
 access-specifier
 data and functions
 access-specifier
 data and functions
 access-specifier
 data and functions
 access-specifier
 data and functions
 access-specifier
 data and functions
} object-list;
```

The *object-list* is optional. If present, it declares objects of the class. Here, *access-specifier* is one of these three C++ keywords:

- **public**
- **private**
- **protected**

*By default*, functions and data declared within a class are **private** to that class and may be accessed only by other members of the class. The **public** access specifier allows functions or data to be accessible to other parts of your program. The **protected** access specifier is needed only when inheritance is involved (see Chapter 6). Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

You may change access specifications as often as you like within a **class** declaration. For example, you may switch to **public** for some declarations and then switch back to **private** again. The class declaration in the following example illustrates this feature:

```
#include <iostream.h>
#include <string.h>
class EMPLOYEE
{ char name[80]; // private by default
public:
 void putname(char n[80]); // these are public
 void getname(char n[80]);

private:
 double wage; // now, private again
public:
 void putwage(double w); // back to public
 double getwage();
};

void EMPLOYEE::putname(char n[80])
{ strcpy(name, n);
}

void EMPLOYEE::getname(char n[80])
{ strcpy(n, name);
}

void EMPLOYEE::putwage(double w)
{ wage = w;
}

double EMPLOYEE::getwage()
{ return wage;
}

int main()
{ EMPLOYEE ted;
 char name[80];
 ted.putname("Ted Jones");
 ted.putwage(75000);
 ted.getname(name);
 cout << name << " makes $";
```

```
 cout << ted.getwage() << " per year.";
 return 0;
 }
```

Here, **EMPLOYEE** is a simple class that is used to store an employee's name and wage. Notice that the **public** access specifier is used twice.

Although you may use the access specifiers as often as you like within a class declaration, the only advantage of doing so is that by visually grouping various parts of a class, you may make it easier for someone else reading the program to understand it. However, to the compiler, using multiple access specifiers makes no difference. Actually, most programmers find it easier to have only one **private**, **protected**, and **public** section within each class. For example, most programmers would code the **EMPLOYEE** class as shown here, with all private elements grouped together and all public elements grouped together:

```
class EMPLOYEE
{
 char name[80];
 double wage;
public:
 void putname(char n [80]);
 void getname(char n [80]);
 void putwage(double w);
 double getwage();
};
```

Functions that are declared within a class are called *member functions*. Member functions may access any element of the class of which they are a part. This includes all **private** elements. Variables that are elements of a class are called *member variables* or *data members*. Collectively, any element of a class can be referred to as a member of that class.

K.V. SARIN

Some important points to remember:

- No member can be an object of the class that is being declared.
- No member can be declared as **auto**, **extern**, or **register**.

In general, you should make all data members of a class private to that class. This is part of the way that encapsulation is achieved. However, there may be situations in which you will need to make one or more variables public. When a variable is public, it may be accessed directly by any other part of your program. The syntax for accessing a public data member is the same as for calling a member function: Specify the object's name, the dot operator, and the variable name.

This simple program illustrates the use of a public variable:

```
#include <iostream.h>
class myclass
{
 public:
 int i, j, k; // accessible to entire program
};
int main()
{
 myclass a, b;

 a.i = 100; // access to i, j, and k is OK
 a.j = 4;
 a.k = a.i * a.j;
 b.k = 12; // remember, a.k and b.k are different
 cout << a.k << " " << b.k;

 return 0;
}
```

## Structures and Classes Are Related

As you have seen, a **class** is syntactically similar to a **struct**. But the relationship between a **class** and a **struct** is closer than you may at first think. In C++, the role of the structure was expanded, making it an alternative way to specify a class. In fact, the only difference between a **class** and a **struct** is that by default all members are public in a **struct** and private in a **class**. In all other respects, structures and classes are equivalent.

That is, in C++, a *structure defines a class type*. For example, consider this short program, which uses a structure to declare a class that controls access to a string:

```
// Using a structure to define a class.
#include <iostream.h>
struct mystr
{
 void buildstr(char s[255]); // public
 void showstr();

private: // now go private
 char str[255];
};

void mystr::buildstr(char s [255])
{
 strcat(str, s);
}

void mystr::showstr()
{
 cout << str << "\n";
}

int main()
```

```

 {
 mystr s;
 s.buildstr(""); // init
 s.buildstr("Hello ");
 s.buildstr("there!");
 s.showstr();

 return 0;
 }

```

This program displays the string **Hello there!**.

The class **mystr** could be rewritten by using **class** as shown here:

```

class mystr
{
 char str[255]; // by default private
public:
 void buildstr(char s [255]); // public
 void showstr();
};

```

You might wonder why C++ contains the two virtually equivalent keywords **struct** and **class**. This seeming redundancy is justified for several reasons. First, there is no fundamental reason not to increase the capabilities of a structure. In C, structures already provide a means of grouping data. Therefore, it is a small step to allow them to include member functions. Second, because structures and classes are related, it may be easier to port existing C programs to C++. Finally, although **struct** and **class** are virtually equivalent today, providing two different keywords allows the definition of a **class** to be free to evolve. In order for C++ to remain compatible with C, the definition of **struct** must always be tied to its C definition.

Although you can use a **struct** where you use a **class**, most programmers don't. Usually it is best to use a **class** when you want a class, and a **struct** when you want a C-like structure.

Conclusion:

**Out side world can access only public part of a class though object of that class.**

Flash Back

- Write a program that adds two distances using function.

Write this program again using class.

Question from CBSE Board:

- Define a class Play in C++ with the following specifications:

*Private members*

- Playcode integer
- PlayTitle 25 characters
- Duration float
- Noofscenes integer

*Public members*

- A constructor function to initialize Duration as 45 and Noofscenes as 5.
- Newplay ( ) function to accept values for Playcode and PlayTitle.
- Moreinfo ( ) function to assign the values of Duration and Noofscenes with the help of corresponding values passed as parameters to this function.
- Showplay () function display all the data members on the screen.

[4 marks, 2003]

- Declare a class myfolder with the following specifications :

*Private members of the class*

- Filenames — an array of strings of size [10] [25] (to represent all the names of files inside myfolder)
- Availspace — long (to represent total number of bytes available in myfolder)
- Usedspace — long (to represent total number of bytes used in myfolder)

*public members of the class*

- Newfileentry( ) - A function to accept values of Filenames, Availspace and Usedspace from user
- Retavailspace( ) - a function that returns the value of total Kilobytes available (1 Kilobyte = 1024 bytes)
- Showfiles( ) - a function that displays the names of all the files in myfolder

[4 marks, 2004]

[4 marks, 2005]

Given in  
the text [4 marks, 2006]

book [4 marks, 2007]

- Define a class clothing in C++ with the following descriptions :

*Private Members:*

- Code of type string
- Type of type string
- Size of type string
- Material of type string
- Price of type float
- A function Calc\_Price ( ) which calculates and assigns the value of Price as follows:

For the value of Material as “COTTON”

| <u>Type</u> | <u>Price (Rs.)</u> |
|-------------|--------------------|
| TROUSER     | 1500               |
| SHIRT       | 1200               |

For Material other than “COTTON” the above mentioned Price gets reduced by 25%.

*Public Members:*

- A constructor to assign initial values of Code, Type and Material with the word “NOT ASSIGNED” and Size and Price with 0.
- A function Enter () to input the values of the data members Code, Type, Size and Material and invoke the CalcPrice () function.
- A function Show ( ) which displays the content of all the data members for clothing.

[4 marks, 2008]

K.V.SARMA