# PROGRAMMING IN C AND C++

## MODULE 4.2


**Centre for Electronics Design & Technology of India**
A Scientific Society under Department of Electronics,
Govt. of India, New Delhi

**Published by CFS Documentation Cell**
**Centre for Electronics Design and Technology of India**
**An Autonomous Scientific Society under Department of Electronics,**
**Govt. of India,**
**New Delhi.**

**First Edition: 1999**



CEDTI/CFS/99/6/4.2/R1

## FOREWORD

The information technology and telecom sectors have suddenly opened up avenues, which require a very large specially trained manpower. These sectors are highly dynamic and need training and re-training of manpower at a rapid rate. The growing gap of requirement of the industry and its fulfillment has created a challenging situation before manpower training institutes of the country. To meet this challenge most effectively, Centre for Electronics Design and Technology of India (CEDTI) has launched its nation-wide franchising scheme.

Centre for Electronics Design and Technology of India (CEDTI) is an Autonomous Scientific Society under the Govt. of India, Department of Electronics with its Headquarters at New Delhi. It operates seven centres located at Aurangabad, Calicut, Gorakhpur, Imphal, Mohali, Jammu and Tezpur. The scheme will be implemented and coordinated by these centres.

The scheme endeavours to promote high quality computer and information technology education in the country at an affordable cost while ensuring uniform standards in order to build a national resource of trained manpower. Low course fees will make this education available to people in relatively small, semi urban and rural areas. State-of-the-art training will be provided keeping in view the existing and emerging needs of the industrial and Govt. sectors. The examinations will be conducted by CEDTI and certificates will also be awarded by CEDTI. The scheme will be operated through all the seven centres of CEDTI.

The CEDTI functions under the overall control and guidance of the Governing Council with Secretary, Department of Electronics as its Chairman. The members of the council are drawn from scientific, government and industrial sectors. The Centres have separate executive committees headed by Director General, CEDTI. The members of these committees are from academic/professional institutes, state governments, industry and department of electronics.

CEDTI is a quality conscious organisation and has taken steps to formally get recognition of the quality and standards in various activities. CEDTI, Mohali was granted the prestigious ISO 9002 certificate in 1997. The other centres have taken steps to obtain the certification as early as possible. This quality consciousness will assist CEDTI in globalizing some of its activities. In keeping with its philosophy of 'Quality in every Activity', CEDTI will endeavour to impart state of the art – computer and IT training through its franchising scheme.

The thrust of the Software Courses is to train the students at various levels to carry out the Management Information System functions of a medium sized establishment, manufacture Software for domestic and export use, make multimedia presentations for management and effectively produce various manufacturing and architectural designs.

The thrust of the Hardware Courses at Technician and Telecommunication Equipment Maintenance Course levels is to train the students to diagnose the faults and carry out repairs at card level in computers, instruments, EPABX, Fax etc. and other office equipment. At Engineer and Network Engineer levels the thrust is to train them as System Engineers to install and supervise the Window NT, Netware and Unix Networking Systems and repair Microcontrollers / Microprocessor based electronic applications.

An Advisory Committee comprising eminent and expert personalities from the Information Technology field have been constituted to advise CEDTI on introduction of new courses and revising the syllabus of existing courses to meet the changing IT needs of the trade, industry and service sectors. The ultimate objective is to provide industry-specific quality education in modular form to supplement the formal education.

The study material has been prepared by the CEDTI, document centre. It is based on the vast and rich instructional experience of all the CEDTI centres. Any suggestions on the improvement of the study material will be most welcome.

**(R. S. Khandpur)**
Director General (CEDTI)

# TABLE OF CONTENTS

# CHAPTER - 1

# DATA INPUT & OUTPUT

## SINGLE CHARACTER INPUT - THE GETCHAR FUNCTION

Single characters can be entered in to the computer using the C library function getchar. The getchar function is a part of the standard C Language I/o Library. It returns a single character from a standard input device. The function does not require any arguments, though a pair of empty parentheses must follow the word getchar.

In general terms a reference to the getchar function is written as

character variable = getchar( );

Here character variable refers to some previously declared character variable

## SINGLE CHARACTER OUTPUT-THE PUTCHAR FUNTION

The putchar function, like getchar, is a part of the standard C language I/o library. It transmits a single character to a standard output device. The character being transmitted will normally be represented as a character- type variable. It must be expressed as an argument to the function enclosed in parentheses following the word putchar.

In general a reference to the putchar function is written as .

Putchar( Character variable )

e.g A C Program contains the following statement :

```
Char C;
—  –
—  –
—  –
_  _
Putchar(C);
```

The first statement declares that C is a character type variable. The second statement causes the current value of C to be transmitted to the standard output device.

## ENTERING INPUT DATA THE SCANF FUNTION

Input data can be entered into the computer from a standard input device by means of the C library function scanf.

In general terms, the scanf function is written as

Scanf ( Control string, arg1,arg2,.....,argn)

Here control string refers to a string containing certain required formatting information, and arg1, arg2,...arg n are arguments that represent the individual input data items. The arguments represent pointers that indicate the addresses of the data item within the computers memory.

The control string comprises individual groups of characters with one character group for each input data item. Each character group must begin witha a percent sign( % ). In its simplest form a single character group will consist of the percent sign, followed by a conversion character which indicates the type of the corresponding data item.

## COMMONLY USED CONVERSION CHARACTERS FOR DATA INPUT

| Conversion character | Meaning |
|---|---|
| c | data item is a single character |
| d | data item is a decimal integer |
| f | data item is a floating point value |
| h | data item is a short integer |
| i | data item is a decimal, hexadecimal or octal integer |
| o | data item is an octal integer |
| s | data item is a string followed by white space character |
| u | data item is an unsigned decimal integer |
| x | data item is a hexadecimal integer |

**e.q of scanf function**

```
# include <stdio.h>
main( )
{
        char item [20];
        int partno;
        float cost;
        .  .  .
        Scanf("%S% d % f", item, &partno, & cost);
        .  .  .
  }
```

The first character group % s indicates that the first argument (item) represents a string the second character group, %d indicates that the second argument ( & partno) represents a decimal integer value. The third character group % f, indicates that the third argument (& cost) represents a floating point value.

e.g. consider the skeleton of the following program

```
# include <stdio.h>
main( )
{
        char item [20];
        int partno;
        float cost;
        . . .
        scant("%s%d%f", item &partno, &scost);
        }
```

The following data items could be entered from the standard input device when the program is executed.

```
fastener 12345 0.05
                or
fastener
12345
 0.0 5
```

Now let us again consider the skeleton structure of a C program

```
# include <stdio.h>
main ( )
{
        int a,b,c;
        . . .
        Scanf ("%3d %3d %3d", & a, & b, & c);
        . . .
}
```

Suppose the input data items are entered as

1 2 3

Then the following assignment is will result

a = 1, b = 2, c = 3

It data is entered as

123 456 789

The following assignments would be

a = 123, b = 456, c = 789

Now suppose that the data had been entered as

123456789

The assignments would be

a = 123, b = 456, c = 789

Again consider the following example

```
# include <stdio.h>
main ( )
{ int i ;
float x;
char c ;
.  .  .
Scanf ("%3d %5f %c", & i, & x, & c);
.  .  .
}
```

If the data item are entered as

10256.875 T

Thees when the program will be executed

10 will be assigned to i

256.8 will be assigned to X

and the character 7 will be assigned to C. the remaining two input characters(5 and T) will be ignored.

**WRITING OUTPUT DATA - THE printf FUNCTION:-**

Output data can be written from the computer on to a standard output device using the library function printf the general form of printf function is

Printf(control string, arg 1, arg 2, . . ., argn)

where control string refers to a string that contains formatting information.

arg1, arg 2, . . ., argn are arguments that represent the individual output data items.

**e.g:-** Following is a simple program that makesuse of the printf fuction.

```
# include <stadio.h>
# include <math.h>
main ( )  /* Print several floating-point numbers */
{
        float i = 2.0, j = 3.0 ;
        Printf ("%f %f %f %f", i, j, i+j, sqrt (i + j ));
        }
```

Executing the program produces the following output

2.000000 3.000000 5.000000 2.236068

**The Gets and Puts FUCTIONS**

The gets and puts are the functions which facilitate the transfer of strings between the computer and the standard input/output devices.

Each of these functions accepts a single argument.
The argument must be a data item that represents a string (e.g, a character array). The string may include white space characters. In the case of gets, the string will be entered from the keyboard and will terminate with a newline character ("i'e" the string will end when the user presses the RETURN key).

```
# include <stdio.h>
main ( )
{
        char line [80];
        gets(line);
        puts(line);
        }
```

Now suppose following string is entered from the standard input device

```
I am happy
```

Now the output of the program will be

```
I am happy
```

# CHAPTER - 2

# THE DECISION CONTROL & LOOP STRUCTURE

The decision control structure in C can be implemented in C using

(a) The if statement
(b) The if - else statement
(c) The conditional operators

**The if Statement**

The general form of if statement looks like this:

```
if (this condition is true)
        execute this statement;
```

Here the keyword if tells the compiler that what follows, is a decision control instruction. The condition following the keyword if is always enclosed within a pair of parentheses. If the condition, whatever it is true, then the statement is executed. It the condition is not true then the statement is not executed instead the program skips past it. The condition in C is evaluated using C's relational operators. The relational operators help us to build expression which are either true or false

for e.g

| This expression | is true if |
|---|---|
| X = = Y | X is equal to Y |
| X ! = Y | X is not equal to Y |
| X <Y | X is less than Y |
| X>Y | X is greater than Y |
| X< = Y | X is less than or equal to Y |
| X> = Y | X is greater than or equal to Y |

Demonstration of if statement

```
main( )
{
```

```
int num;
Printf("Enter a number less than 10");
Scanf("%d", & num);
if(num < = 10)
Printf("The number is less than 10");
}
```

**Multiple Statements within if**

If it is desired that more than one statement is to be executed if the condition following if is satisfied, then such statements must be placed within pair of braces.
for e.g. The following program demonstrate that if year of service greater than 3 then a bonus of Rs. 2500 is given to employee. The program illustrates the multiple statements used within if

```
/* calculation of bonus */
main( )
{
int bonus, CY, Yoj, yr-of-ser;
Printf("Enter current year and year of joining");
Scanf("%d %d", Scy, Syoj);
yr-of-ser = CY-Yoj;
if(yr-ofser > 3)
{
bonus = 2500;
Printf("Bonus = Rs. %d", bonus);
}
}
```

**If - Else** The if statement by itself will execute a single statement or a group of statements when the condition following if is true. it does nothing when the condition is false. It the condition is false then a group of statements can be executed using else statement. The following program illustrates this

```
/* Calculation of gross salary */
main( )
{
float bs, gs, da, hra;
Printf("Enter basic salary");
Scanf("/.f", & bs);
if(bs <1500)
{ hra = bs * 10/100;
  da  = bs * 90/100;
}
else
{ hra = 500;
   da  = bs * 98/100;
}
gs = bs+hra+da;
```

```
Printf("gross salary = Rs. /.f", gs);
}
```

**Nested if - else** It we write an entire if - else construct within the body of the if statement or the body of an else statement. This is called 'nesting' of **if**. For e.g.

```
if(condition)
{
  if (condition)
        do this;
   else
        { do this;
          and this;
          }
}
        else
              do this;
```

**The Loop Control Structure**

These are three methods by way of which we can repeat a part of a program. They are:

(a) Using a for statement
(b) Using a while statement
(c) Using a do-while statement

**The While Loop**

The general from of while is as shown below:

```
initialise loop counter;
while (test loop counter using a condition)
{
        do this;
        and this;
        increment loop counter;
}
```

The parentheses after the while contains a condition so long as this condition remains true all statements within the body of the while loop keep getting executed repeatedly
for e.g.

```
/* calculation of simple interest for 3 sets of p, n and r */
main( )
{     int p,n, count;
        float r, si;
        count = 1;
        while(count < = 3 )
        {
              printf("\n Enter values of p, n and r");
```

```
        scanf("%d %d %f", & p, & n, & r);
        si = p*n*r/100;
        printf("simple interest = Rs. %f", si);
        count = count +1;
}
```

**The do-while Loop**

The do-while loop takes the following form

```
do
{
        this;
        and this;
        and this;
        and this;
        } while (this condition is true);
```

There is a minor difference between the working of while and do-while loops. The difference is the place where the condition is tested. The while tests the condition before executing any of the statements within the while loop. As against this the do-while tests the condition after having executed the statements within the loop.

for e.g.

```
main( )
{
while(5<1)
        printf("Hello \n");
}
```

In the above e.q. the printf will not get executed at all since the condition fails at the first time itself. Now let's now write the same program using a do-while loop.

```
main( )
{
        do
        {
        Printf ("itello \n");
        } while (5<1);
}
```

In the above program the printf( ) would be executed once, since first the body of the loop us executed and then the condition is tested.

**The for Loop**

The general form of for statement is as under:

```
for( initialise counter; test counter; increment counter)
{ do this;
and this;
and this;
}
```

Now let us write the simple interest problem using the for loop

```
/* calculation of simple interest for 3 sets of p,n and main( )
{
        int p,n, count;
        float r, si;
        for(count = 1; count <=3; count = count +1)
        {
                printf("Enter values of p,n and r");
                Scanf("%d %d %f", & p, & n, & r);
                si = p * n * r / 100;
                printf("Simple interest = Rs %f \n", si);
        }
}
```

**The Break Statement**

The keyword break allows us to jump out of a loop instantly without waiting to get back to the conditional test. When the keyword break is encountered inside any C loop, control automatically passes to the first statement after the loop.

For e.g. the following program is to determine whether a number is prime or not.

***Logic:*** To test a number is prime or not, is to divide it successively by all numbers from 2 to one less than itself. It the remainder of any of the divisions is zero, the number is not a prime.

Following program implements this logic

```
        main( )
        {
                int num, i;
                printf("Enter a number");
                scanf("%d", &num);
                i = 2
                while (i < = num -1)
                {
                        if (num%i= = 0)
                        {
```

```
                printf("Not a prime number");
                break;
                }
                i + +
        }
        if(i = = num)
                printf("Prime number");
}
```

**The Continue Statement**

The keyword continue allows us to take the control to the beginning of the loop bypassing the statements inside the loop which have not yet been executed. When the keyword continue is encountered inside any C loop control automatically passes to the beginning of the loop. for e.g.

```
main( )
{
        int i,j;
        for(i = 1; i< = 2; i++)
        {
                for(j=1; j<=2; j++)
                {
                if (i= =j)
                        continue;
                printf("\n%d%d\n", i,j);
        }
}
```

The output of the above program would be....

```
12
21
```

when the value of i equal to that of j, the continue statement takes the control to the for loop (inner) bypassing rest of the statements pending execution in the for loop (inner).

**The Case Control Structure**

***Switch Statement:-***

The switch statement causes a particular group of statements to be chosen from several available groups. The selection is based upon the current value of an expression that is included within the switch statement. The form of switch statement is.

```
switch (integer expression)
```

```
{       case constant 1:
                do this;
          case constant 2:
                do this;
          case constant 3:
                do this;
          default;
                do this;
}
```

# CHAPTER - 3

# ARRAYS

## WHAT ARE ARRAYS

Let us suppose we wish to arrange the percentage marks obtained by 100 students in ascending order. In such a case there are two options store these marks in memory:

(a) Construct 100 variables to store percentage marks obtained by 100 different students i.e "each variable containing one students marks.

(b) Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Clearly, the second alternative is better because it would be much easier to handle one array variable than handling 100 different variables

Now we can give a formal definition of array . An array is a collective name given to a group of similar quantities. These similar quantities could be percentage marks of 100 students, or salaries of 300 employee or ages of 50 employees. Thus an array is a collection of similar elements. These similar elements could be all ints, or all floats or all chars etc. usually, the array of characters is called a 'string', where as an array of ints or floats is called simply an array. All elements of any given array must be of the same type i.e we can't have an array of 10 numbers, of which 5 are ints and 5 are floats.

## ARRAY DECLARATION

To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how. large an array we want.

for e.g. int marks [30];

Here int specifies the type of variable, marks specifies the name of the variable. The number 30 tells how many elements of the type int will be in our array. This number is often called the 'dimension' of the array. The bracket [ ] tells the compiler that we are dealing with an array.

## ACCESSING ELEMENTS OF AN ARRAY

To access an individual element in the array we have to subscript it, that is we have to put the number in the brackets following the array name. All the array elements are numbered starting with 0. Thus, marks [2] is not the second element of array but it is actually the third element. Thus marks [i] refers to i + 1 th element of the array

Let us take an example of a program using array

```
main ( )
{ float avg, sum=0;
int i;
int marks [30]; /* array declaration*/
for ( i =0; i < = 29; i ++)
{
printf ("\n Enter marks ");
scanf ("%d", &marks [i]);
}
for ( i = 0; i <= 29; i ++)
sum = sum +  marks [i];
avg = sum /30;
printf ("\n Average marks = % f",  avg);
}
```

## ENTERING DATA IN TO THE ARRAY

The section  of code which places data in to an array is

```
for (i=0; i<= 29; i++)
{
printf ("\n Enter marks")
scanf ("%d", &marks [i]);
}
```

The above section will read about 30 elements numbered from 0 to 29 in to the marks array. This will take input from the user repeatedly 30 times.

## READING  DATA FROM ARRAY

```
for ( i=0; i <= 29; i++);
sum = sum + marks [i];
avg = sum / 30;
printf ("\n Average marks = % f", avg );
```

The rest of the program reads the data back out of the array and uses  it to calculate the average. The for loop is much the same, but now the body of loop causes each student's marks to be added to a running total stored in a variable called sum. When all the marks have been added up, the result is divided  by 30, the numbers of students to get the average.

Let us summarize the facts about array

(a) An array is a collection of similar elements.
(b) The first element in the array is numbered 0, 50 the last element is 1 less than the size of the array.
(c) An array is also known as subscripted variable .
(d) Before using an array its type and dimension must be declared.
(e) However big an array is its elements are always stored in contiguous memory locations.

**ARRAY INITIALISALION**

To initialise an array while declaring it. Following are a few examples which demonstrate this

```
int num [6]      =    {2, 4, 12, 5, 45, 5};
int n [ ]        =    {2, 4, 12, 5, 45, 5};
float press [ ]  =    { 12.3, 34.2, -23.4, - 11.3}
```

The following points should be noted

(a) Till the array elements are not given any specific values, they are suppose to contain garbage values.
(b) If the array is initialized where it is declared mentioning the dimension of the array is optional as in the 2nd example above.

**MULTIDIMENSIONAL ARRAYS**

In C one can have arrays of any dimensions. To understand the concept of multidimensional arrays let us consider the following 4 X 5 matrix

| | | *Column numbers (j)* | | | | |
|---|---|---|---|---|---|---|
| | **0** | **10** | **4** | **3** | **-10** | **12** |
| | **1** | **2** | **3** | **0** | **61** | **8** |
| *Row number (i)* | **2** | **0** | **16** | **12** | **8** | **0** |
| | **3** | **12** | **9** | **18** | **45** | **-5** |

Let us assume the name of matrix is x

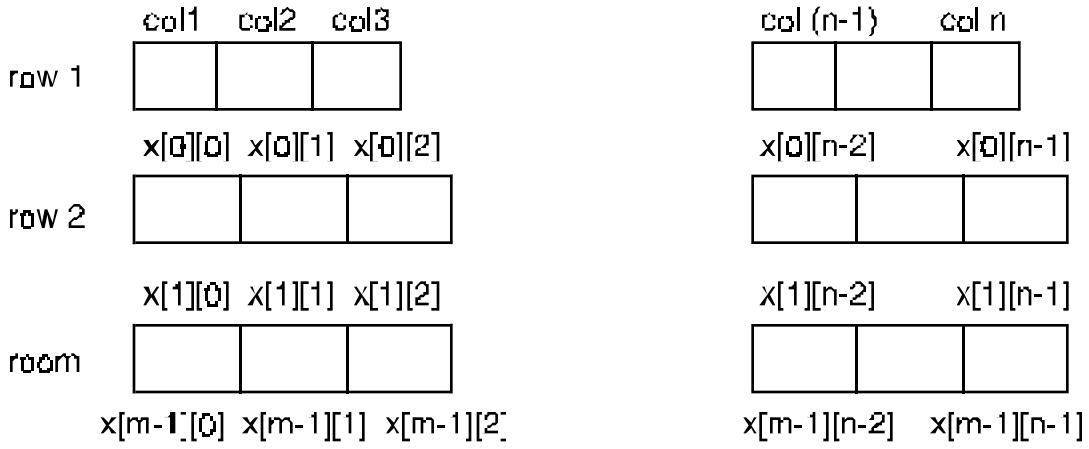
To access a particular element from the array we have to use two subscripts on for row number and other for column number the notation is of the form

X [i] [j] where i stands for row subscripts and j stands for column subscripts.

Thus X [0] [0] refers to 10, X [2] [1] refers to 16 and so on In short multi dimensional arrays are defined more or less in the same manner as single dimensional arrays, except that

for subscripts you require two squire two square brackets. We will restrict our decision to two dimensional arrays.

An array of M rows and n columns can be pictorially shown as




Below given are some typical two-dimensional array definitions

```
float table [50] [50];
char line [24] [40];
```

The first example defines tables as a floating point array having 50 rows and 50 columns. the number of elements will be 2500 (50 X50).

The second declaration example establishes an array line of type character with 24 rows and 40 columns. The number of elements will be (24 X 40) 1920 consider the following two dimensional array definition int values [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10. 11, 12, };

Thus the array values can be shown pictorially as

Column number

| Row number | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 5 | 6 | 7 | 8 |
| 2 | 9 | 10 | 11 | 12 |

Values [0] [0] = 1 Values [0] [1] = 2 Values [0] [2] = 3 Values [0] [3] = 4
Values [1] [0] = 5 Values [1] [1] = 6 Values [1] [2] = 7 Values [1] [3] = 8
Values [2] [0] = 9 Values [2] [1] = 10 Values [2] [2] = 11 Values [2] [3] = 12

Here the first subscript stands for the row number and second one for column number. First subscript ranges from 0 to 2 and there are altogether 3 rows second one ranges from 0 to 3 and there are altogether 4 columns.

Alternatively the above definition can be defined and initialised as

```
int values [3] [4] = {
                        { 1, 2, 3, 4}
                        { 5, 6, 7, 8}
                        {9, 10, 11, 12}
                        };
```

Here the values in first pair of braces are initialised to elements of first row, the values of second pair of inner braces are assigned to second row and so on. Note that outer pair of curly braces is required.

If there are two few values within a pair of braces the remaining elements will be assigned as zeros.

Here is a sample program that stores roll numbers and marks obtained by a student side by side in matrix

```
main ( )
{
        int stud [4] [2];
        int i, j;
        for (i =0; i < =3; i ++)
        {
                printf ("\n Enter roll no. and marks");
                scanf ("%d%d", &stud [i] [0],  &stud [i] [1] );
        }
        for (i = 0; i < = 3; i ++)
        printf ("\n %d %d", stud [i] [0], stud [i] [1]);
}
```

The above example illustrates how a two dimensional array can be read and how the values stored in the array can be displayed on screen.

# CHAPTER - 4

# FUNCTIONS

A computer program cannot handle all the tasks by it self. Instead its requests other program like entities - called 'functions in C - to get its tasks done. A function is a self contained block of statements that perform a coherent task of same kind.

e.g

```
main ( )
{
        message ( );
        printf ("\n I am in main ");
}
        message ( )
{
        printf (\n Hello");
}
```

Output of the program will be

```
I am in main
Hello
```

Here main( ) is the calling function and message is the called function. When the function message( ) is called the activity of main( ) is temporarily suspended 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.

The General form of a function is:

```
function (arg1, arg2, arg3)
type arg1, arg2, arg3
{
        statement 1;
        statement2;
        statement3;
        statement4;
}
```

There are basically two types of functions
(i) Library functions Ex. printf ( ), scanf ( ) etc
(ii) user defined function e.g the function message mentioned above.

The following point must be noted about functions
(i) C program is a collection of one or more functions
(ii) A function gets called when the function name is followed by a semicolon for e.g.

```
main ( )
    {
        message ( );
    }
```

(iii) A function is defined when function name is followed by a pair of braces in which one or more statements may be present for e.g.

```
message ( )
    {
        statement 1;
        statement2;
        statement 3;
    }
```

(iv) Any function can be called from any other function even main ( ) can be called from other functions. for e.g.

```
main ( )
    {
        message ( );
    }
        message ( )
    {
        printf (" \n Hello");
        main ( );
    }
```

(v) A function can be called any number of times for eg.

```
main ()
    {
        message ( );
        message ( );
    }
        message ( )
    {
        printf ("\n Hello");
    }
```

(vi) The order in which the functions are defined in a program and the order in which they get called need not necessarily be same for e.g.

```
main ( );
        {
                message 1 ( );
                message 2 ( );
        }
                message 2 ( )
        {
                printf ("\n I am learning C");
        }
                message 1 ( )
        {
                printf ( "\n Hello ");
        }
```

(vii) A function can call itself such a process as called 'recursion'.

(viii) A function can be called from other function, but a function cannot be defined in another function. These the following program code would be wrong, since argentina is being defined inside another function main ( ).

```
main ( )
        {
                printf ("\n I am in main");
                argentina ( )
                {
                printf {"\n I am in argentina");
                }
        }
```

(ix) Any C program contains at least one function

(x) If a program contains only one function, it must be main( ).

(xi) In a C program if there are more than one functional present then one of these functional must be main( ) because program execution always begins with main( ).

(xii) There is no limit on the number of functions that might be present in a C program.

(xiii) Each function in a program is called in the sequence specified by the function calls in main( )

(xiv) After each function has done its thing, control returns to the main( ), when main( ) runs out of function calls, the program ends.

**WHY USE FUNCTIONS**

Two reasons :

(i) Writing functions avoids rewriting the same code over and over. Suppose that there is a section of code in a program that calculates area of a triangle. If, later in the program we want to calculate the area of a different triangle we wont like to write the same instructions all over again. Instead we 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.

(ii) 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 in to separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code in to modular functions also makes the program easier to design and understand.

(a) Functions declaration and prototypes

Any function by default returns an int value. If we desire that a function should return a value other than an int, then it is necessary to explicitly mention so in the calling functions as well as in the called function.

for e.g

```
main ( )
{
        float a,b,
        printf ("\n Enter any number");
        scanf ("\% f", &a );
        b = square (a)
        printf ("\n square of  % f is % f", a,b);
}
        square (float X)
{
        float y;
        Y = x * x;
        return (y);
}
```

the sample run of this program is

```
Enter any number 2.5
square of 2.5 is 6.000000
```

Here 6 is not a square of 2.5 this happened because any C function, by default, always returns an integer value. The following program segment illustrates how to make square ( ) capable of returning a float value.

```
main ( )
{
        float square ( );
        float a, b;
        printf (“\n Enter any number “);
        scanf (“%f” &a);
        b = square (a);
        printf (“\n square of % f is % f, “ a, b);
}
        float square (float x)
{
        float y;
        y= x *x;
        return ( y);
}
        sample run
        Enter any number 2.5
        square of 2.5 is 6.2500000
```

**CALL BY VALUE**

In the preceding examples we have seen that whenever we called a function we have always passed the values of variables to the called function. Such function calls are called ‘calls by value’ by this what it meant is that on calling a function we are passing values of variables to it.

The example of call by value are shown below ;

```
sum = calsum (a, b, c);
f =     factr (a);
```

In this method the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual argument in the calling function. The following program illustrates this:

```
main ( )
{
        int a = 10, b=20;
        swapy (a,b);
        printf (“\na = % d b = % d”, a,b);
}
        swapy (int x, int y)
{
        int t;
        t = x;
```

```
        x = y;
        y = t;
        printf ( "\n x = % d y = % d" , x, y);
}
```

The output of the above program would be;
x = 20 y = 10
a =10 b =20

**CALL BY REFERENCE**

In the second method the addresses of actual arguments in the calling function are copied in to formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them the following program illustrates this.

```
main ( )
{
        int a = 10, b =20,
        swapv (&a, &b);
        printf ("\n a = %d b= %d", a, b);
}
        swapr (int **, int * y)
{
        int t;
        t = *x
        *x = *y;
        *y = t;
}
```

The output of the above program would be
a = 20 b =10

# CHAPTER - 5

# DYNAMIC DATA STRUCTURES IN C

## POINTERS

### THE and * Operators

A pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are used frequently in C, as they have a number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point. Pointers provide a way to return multiple data items from a function via function arguments to be specified as arguments to a given function.

Pointers are also closely associated with arrays and therefore provide an alternate way to access individual array elements.

Within the computer's memory, every stored data item occupies one or more adjacent memory cells. The number of memory cells required to store a data item depends on the type of data item. For example, a single character will be stored in 1 byte of memory integer usually requires two adjacent bytes, a floating point number may require four adjacent bytes.

Suppose V is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can be accessed if we know the location of the first memory cell. The address of V'S memory location can be determined by the expression & V, where & is a unary operator, called the address operator, that evaluates the address of its operand.

Now let us assign the address of V to another variable, PV. Thus,

PV = & V

This new variable is called a pointer to V, since it "Points" to the location where V is stored in memory. Remember, however, that PV represents V's address, not its value. Thus, Pv is called pointer variable.

| address of V | value of V |
|---|---|
| PV | V |

**Relationship between PV and V (where PV = & V and V = * PV)**

The data item represented by V can be accessed by the expression *PV where * is a unary operator, that operates only on a pointer variable. Therefore, PV and V both represent the same data item. Furthermore, if we write PV = &V and U = PV, then u and v will both represent the same values i.e., the value of V will indirectly be assigned to u.
Example :

```
int quantity = 179 ;
```

The statement instructs the system to find a location for the integer quantity and puts the value 179 in that location. Let us reassume that the system has chosen the address location 5000 for quantity.

| Quantity | Variable |
|---|---|
| 179 | Value |
| 5000 | Address |

**Representation of a Variable**
Remember, since a pointer is a variable, its value is also stored in the memory in another location.

The address of P can be assumed to be 5048.

| Variable | Value | Address |
|---|---|---|
| Quantity | 179 | 5000 |
| P | 5000 | 5048 |

**POINTER AS A VARIABLE**

**Declaring and initializing Pointers**
Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:
data type * Pt _ name

This tells the compiler three things about the variable Pt_name.

1. The * tells that the variable Pt_name is a pointer variable.
2. Pt_name needs a memory location.
3. Pt_name ponts to a variable of type data type.

Example :

```
int * P ;
```

Declares the variable P as a pointer variable that points to an integer data type.

```
float * y ;
```

declares y as a pointer to a floating point variable.

Once pointer variable has been declared, it can be mode to point to a variable using an assignment statement such as

```
P = & quantity ;
```

which causes P to point to quantity. P contains the address of quantity. This is known as pointer initialization.

**Pointer expressions**

Like other variables, pointer variables can be used in expressions. For example, if P1 and P2 are properly declared and initialized pointers, then the following statements are valid.

```
1)   Y     =   * P1  ;
2)   Sum   =   Sum + * P1  ;
3)   Z     =   S * - * P2 / * P1  ;
4)   * P2  =   * P2 + 10  ;
```

Note that there is a blank space between / and * in the item 3 above.

If P1 and P2 are pointers then the expressions such as,

P1 + 4 ,  P2 - 2  ,  P1 - P2  ,  P1 ++ , — P2 are allowed

also,

```
Sum = Sum + *P2 ;
      P1 ++  ;
- -    P2  ;
P1 > P2
      P1 = = P2
      P1 ! = P2
```

are all allowed expressions.

The expressions such as,

P1 / P2  or  P1 * P2  or  P1/3

are not allowed.

**Pointer Assignments**

After declaring a pointer, pointer is assigned a value, so that it can point to a particular variable. eg.

```
int * P  ;
int i    ;
P = & i ;
```

This is called assignment expression in which pointer variable P is holding the address of i.

**Pointer arithmetic**

Two pointer values can be added, multiplied, divided or subtracted together.
eg.

```
if      int 2  ;
        int  j  ;
        int * P , * q ;
1 = 5 , j = 10 ;
```

Now, various pointer arithmetic can be performed
eg.

```
* j =  * j  + * j  ;
```

The value of variable j is changed from 10 to 15.

```
* j =  * j -  * i  ;
```

The value of variable j is changed from 10 to 5.

```
* i =  * i  ** j   ;
```

The value of i is changed from 5 to 50 ;

Consider another example,
if there is array and a pointer is pointing to it

```
int i [10] ;
int * P ;
P = i   ;
```

Now, arithmetic operations like

```
P = P + 4  ;
```

Will move the pointer P from the starting address of the array to the fourth subscript of array.

Similarly, if P1 and P2 are both pointers to the same array, then P2 - P1 gives the number of elements between P1 and P2.

arithmetic operations like

P1/P2  or P1 x P2  or P/3 are not allowed.

**Pointer Comparison**

In addition to arithmetic operations, pointers can also be compared using the relational operators. The expressions such as

P1 > P2 ,  P1 = = P2 ,  P1 ! = P2  are allowed.

However, any comparison of pointers that refer to separate and unrelated variables make no sense. Comparisons can be used meaningfully in handling arrays and strings.

**The dynamic allocation functions - malloc and calloc**

Most often we face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program for example, consider a program for processing the list of customers of a company. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Such situations can be handled more easily and effectively by using (6) what is called dynamic data structures.

**DYNAMIC MEMORY ALLOCATION**

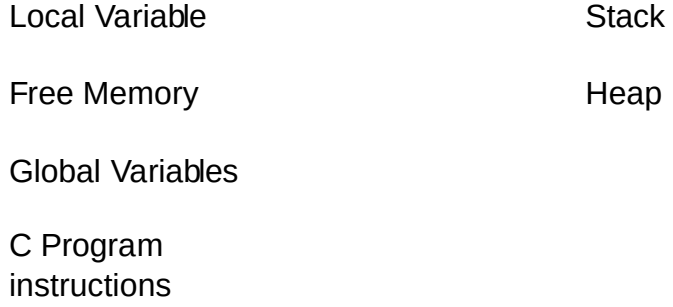
C language requires that the number of elements in an array should be specified at compile time. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages take the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as dynamic memory allocation. The library functions used for allocating memory are :

| Function | Task |
|---|---|
| Malloc ( ) | Allocates requested size of bytes and returns a pointer to the first byte of the allocated space. |
| Calloc ( ) | Allocates space for an array of element, initializes them to zero and then returns a pointer to the memory. |

**Memory Allocation Process**

Let us first look at the memory allocation process associated with a C program. Fig. below shows the conceptual view of storage of a C program in memory.



The program instructions and global and static variables are stored in a region known as permanent storage area and the local variables are stored in another area called stack. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. The free memory region is called the heap. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory "overflow" during dynamic allocation process. In such situations, the memory allocations functions mentioned above returns a NULL pointer.

**ALLOCATING A BLOCK OF MEMORY**

A block of memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form;

```
Ptr = ( Cast type * ) malloc ( byte size ) ;
```

Ptr is a pointer of type cast type. The malloc returns a pointer (of cast type) to an area of memory with size byte - size.

Example :

```
X = ( int * ) malloc ( 100 * size of ( int )) ;
```

On successful execution of this statement, a memory space equivalent to "100 times the size of an int" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer X of type int.

Similarly, the statement

```
Cptr = ( char * ) malloc ( 10 ) ;
```

allocates 10 bytes of space for the pointer cptr of type char.

This is illustrated below :

Cptr
Address of
first byte

**10 bytes of Space**

Remember, the malloc allocates a block of adjacent bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it foils, it returns a NULL. We should therefore check whether the allocation is successful before using the memory pointer.

Example :
Write a program that uses a table of integers whose size will be specified interactively at run time.
Program :

```
        # include    <      Stdio. h ;
        # include    <      Std lib. h ;
        # define NULL O

main ( )
        {
            int * P,  * table ;
            int size ;
        Printf ( “In What is the size of table ? “ ) ;
        Scanf ( “ % d”, size ) ;
        Printf ( “In” ) ;

    if (( table = (int * ) malloc (size * size of (int)) = = NULL )
    {
        Printf (“No space available \ n”) ;
            exit ( i ) ;
    }
    Printf (“\n address of the first byte is % 1d\n”, table );
    Printf(“\n Input table values”);
    for ) P = table; P < table + size; P++ )
            Scanf (“%d”, *P );

for ( P = table + size - 1; P > = table; P-- )
    Printf (“%d is stored at address %1d \n”, *P, P );
}
```

**Allocating Multiple Blocks of Memory**

Calloc is another memory allocation function that is normally used for requesting memory space at runtime for storing derived data types such as arrays and structures. While malloc allocates a single block of storage space, calloc allocates multiple blocks of storage, each of the same size, and then allocates all bytes to O. The general form of calloc is :

Ptr = (Cast type * ) Calloc ( n, elem-size );

The above statement allocates contiguous space for n blocks, each of size elem-size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

The following program allocates space for a structure variable.

```
#    include < Stdio. h;
#    include < Std lib.h;
```

```
Struct Student
        {
        char name (25);
        float age;
        long int num;
        } ;
        typedef struct student record ;
                record * ptr ;
                int class-size = 30 ;

Ptr = ( record * ) calloc ( class-size, size of ( record )) ;
        - - - -
        - - - -
```

record is of type struct student having three number :

name, age and num. The calloc allocates memory to hold data for 30 such records. We should check if the requested memory has been allocated successfully before using the ptr. This may be done as follows:

```
if ( ptr = = NULL )
{
Printf ( "Available memory not sufficient") ;
        exit ( 1 ) ;      }
```

**POINTERS VS. ARRAY**

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element suppose we declare an array X as follows :

```
Static int X [ 6 ] = { 1, 2, 3, 4, 5, 6 } ;
```

Suppose the base address of X is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows :

| ELEMENTS | x [0] | x[1] | x[2] | x[3] | x[4] | x[5] |
|---|---|---|---|---|---|---|
| VALUE | 1 | 2 | 3 | 4 | 5 | 6 |
| Address | 1000 | 1002 | 1004 | 1006 | 1008 | 1010 |

**BASE ADDRESS**

The name X is defined as a constant pointer pointing to the first clement, x [0] and therefore the value of X is 1000, the location whose X[0] is stored. That is ;

X = & x[0] = 1000

If we declare P as an integer pointer, then we can make the pointer P to point to the array X by the following assignment :

```
P = X ;
```

This is equivalent to P = & X[0] ;

Now we can access every value of x using P+ + to more from one element to another. The relationship between P and X is shown below :

```
P     =     & x[0] ( = 1000)
P+1   =     & x[1] ( = 1002)
P+2   =     & x[2] ( = 1004)
P+3   =     & x[3] ( = 1006)
P+4   =     & x[4] ( = 1008)
P+5   =     & x[5] ( = 1010)
```

The address of an element is calculated using its index and the scale factor of the data type. For instance,

```
address of X[3]     =     base address + (3 X Scale factor of int)
                    =     1000  +  (3 x 2)     =     1006
```

When handling array, instead of using array indexing, we can use pointers to access array elements. Note that X(P+3) gives the value of X[3]. The pointer accessing method is more faster than array indexing.

**POINTERS AND FUNCTIONS**

When an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal fashion.

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling function using pointers to pass the address of variable is known as call by reference. The function which is called by reference can change the value of the variable used in the call.

eg.

```
main ( )
  {
        int X ;
        X = 40 ;
        Change ( & X ) ;
        Printf ( " %d", X ) ;
   {
        Change ( int * P )
```

```
{
    * P = * P + 10 ;
}
```

When the function change is called, the address of the variable X, not its value, is passed into the function change ( ). Inside change ( ), the variable P is declared as a pointer and therefore P is the address of the variable X. The statement,

```
* P =  * P + 10 ;
```

means add 10 to the value stored at address P. Since P represents the address of X, the value of X is changed from 50. Therefore, the output of the program will be 50 not 40.

These, call by reference provides a mechanism by which the function can change the stored values in the calling function.

## POINTERS TO FUNCTIONS

A function like a variable, has an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type ( * fp) ( ) ;
```

This tells the compiler that fp is a pointer to a function which returns type value.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

For example,

```
double (*P1)( ), mul ( ) ;
      P1 = mul ;
```

declare P1 as a pointer to a function and mul as a function and then make P1 to point to the function mul. To call the function mul, we may now use the pointer P1 with the list of parameters. That is,

```
(*P1) (x,y)
```

is equivalent to mul ( x,y )

## FUNCTIONS RETURNING POINTERS

The way functions return an int, a float, a double or any other data type, it can even return a pointer. However, to make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function declaration. The following program illustrates this

```
main ( )
    {
        int     * P ;
        int     * fun ( ) ;
```

```
    P = fun ;
    Printf ( "\n % Id", P ) ;
}
    int * fun (  )
{
    int   i = 20;
    return (& i) ;
}
```

In this program, function funllis declared as pointer returning function can return the address of integer type value and in the body of the function fun (  ) we are returning the address of integer type variable i into P which is also integer type pointer.

**POINTERS AND VARIABLE NUMBER OF ARGUMENTS**

We use printf ( ) so often without realizing hove it works correctly irrespective of hove many arguments we pass to it. How do we about writing such routines which can take variable number of arguments? There are three macros available in the file "stdarg.g" called va-start, va-arg and va-list which allow us to handle this situation. These macros provide a method for accessing the arguments of the function when a function takes a fixed number of arguments followed by a variable number of arguments. The fixed number of arguments are accessed in the normal way, whereas the optional arguments are accessed using the macros va-start and va-arg. Out of these macros va-start is used to initialise a pointer to the beginning of the list of optional arguments. On the other hand the macro va-arg is used to initialise a pointer to the beginning of the list of optional arguments. On the other hand the macro va-arg is used to advance the pointer to the next argument.

eg.

```
#       include "stdarg. h"
#       include < stdio. h >
main (   )
        {
                int max ;
                max = find max ( 5, 23, 15, 1, 92, 50 ) ;
                Printf ("\n max = % d", max ) ;
                max = find max (3, 100, 300, 29 ) ;
                Printf ("\n Max = %d", max ) ;
        }
        findmax (int totnum)
                {
                int max, count, num ;
                Va-list ptr ;
        Va-Start ( Ptr, tot-num ) ;
        max = Va-arg (Ptr, int ) ;

for ( count = 1 ; Count < tot-num ; count + + )
        {
```

```
                num = Va-arg (Ptr, int ) ;
                if ( num > max )
                        max = num ;
                }
                return ( max ) ;
        }
```

Here we are making two calls to findmax( ) first time to find maximum out of 5 values and second time to find maximum out of 3 values. Note that for each call the first argument is the count of arguments that are being passed after the first argument. The value of the first argument passed to findmax( ) is collected in the variable tot-num findmax( ) begins with a declaration of pointer ptr of the type Va -list. Observe the mext statement carefully

```
Va-Start ( Ptr, tot_num ) ;
```

This statements sets up ptr such that it points to the first variable argument in the list. If we are considering the first call to findmax ( ) ptr would now point to 23. The next statement max = Va_arg ( ptr, int ) would assign the integer being pointed to by ptr to max. Thees 23 would be assigned to max, and ptr will point to the next argument i.e. 15.

## POINTERS TO POINTERS

The concept of pointers can be further extended. Pointer we know is a variable which contains address of another variable. Now this variable itself could be another pointer. These we now have a pointer which contains another pointer's address. The following example should make this point clear.

```
main ()
        {
            int i = 3 ;
            int * j ;
            int * * k ;
                    j = & i ;
                    k = & j ;
Printf ("\n address of i = % ld", & i );
Printf ("\n address of i = % ld",  j );
Printf ("\n address of i = % ld",  * k );
Printf ("\n address of j = % ld", & j );
Printf ("\n address of j = % ld",  k );
Printf ("\n address of k = % ld", & k );
Printf ("\n address of j = % ld", j );
Printf ("\n address of k = % ld", k );
Printf ("\n address of i = % ld", i );
Printf ("\n address of i = % ld", * (& i));
Printf ("\n address of i = % ld", * j);
Printf ("\n address of i = % ld", ** k );
  }
```

In this program i is an integer type value, j is a pointer to this variable and k is another pointer type variable pointing to j.

| i | j | k |
|---|---|---|
| 3 | 6485 | 3276 |
| 6485 | 3276 | 7234 |

All the addresses are assumed addresses K is pointing to the variable j. These K is a pointer to pointer. In principle, there could be a pointer to a pointer's pointer, of a pointer to a pointer to a pointer's pointer. There is no limit on how far can we go on extending this definition.

**ARRAY OF POINTERS**

The way there can be an array of ints or an array of floats, similarly there can be an array of pointers. Since a pointer variable always contain an address, an array of pointers would be nothing but collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses. All rules that apply to an ordinary array apply to the array of pointers as well.

```
eg.        main ( )
                {
                    int * arra [ 4 ];
                    int     i = 31, j = 5, k = 19, L = 71, m;
                    arra [0] = & i ;
                    arra [1] = & j ;
                    arra [2] = & k ;
                    arra [3] = & l ;
                    for (m=0; m<=3 ; m+ +)
                        Printf ("\n% d", * (arr[m])) ;
                    }
```

The output will be -

```
31
 5
19
71
```

| i | j | k | l | |
|---|---|---|---|---|
| 31 | 5 | 19 | 71 | 71 |
| 4008 | 5116 | 6010 | 7118 | 7118 |

| arr[0] | arr[1] | arr[2] | arr[3] |
|---|---|---|---|
| 4008 | 5116 | 6010 | 7118 |
| 7602 | 7604 | 7606 | 7608 |

Fig.

Figure shows the contents and the arrangement of the array of pointers in memory. An contains addresses of isolated int variables i,j,k and l. The for loop in the program picks up the addresses present in arr and prints the values present at these addresses.

An array of pointers can even contain the addresses of other arrays.
eg.

```
main ( )
    {
static int a [ ]  = { 0, 1, 2, 3, 4, };
static int*p [ ] = { a, a+1, a+2, a+3, a+4 } ;
int * * ptr = p ;
    Printf("\n %1d %1d", a, *a) ;
    Printf("\n %1d %1d %d",  P, *P, **P) ;
    Printf("\n %1d %1d %d", ptr, *ptr, **ptr) ;
    }
```

Output

| | | | a[0] | a[1] | a[2] | a[3] | a[4] |
|---|---|---|---|---|---|---|---|
| 6004 | 0 | | 0 | 1 | 2 | 3 | 4 |
| 9016 | 6004 | 0 | 6004 | 6006 | 6008 | 6010 | 6012 |
| 9016 | 6004 | 0 | P[0] | P[1] | P[2] | P[3] | P[4] |
| | | | 6004 | 6006 | 6008 | 6010 | 6012 |
| | | | 9016 | 9018 | 9020 | 9022 | 9024 |

Ptr
9016
7888

**Explanation**

Look at the initialization of the array P[ ]. During initialization, the addresses of various elements of the array a[ ] are stored in the array P[ ]. Since the array P[ ] Contains addresses of integers, it has been declared as an array of pointers to integers. In the variable ptr. the base address of the array P [ ], i.e. 9016 is stored. Since this address is the address of P[0], which itself is a pointer, ptr has been declared as pointer to an integer pointer. Consider statement.

```
Printf("\n %1d %1d %d", P, *P, **P) ;
```

Here P would give the base address of the array P[ ] i.e. 9016 ; *P would give the value at this address i.e. 6004, **P would give the value at the address given by *P, i.e. value at address 6004, which is 0.

# CHAPTER - 6

# STRUCTURES & UNIONS

## INTRODUCTION

A structure is a convenient tool for handling a group of logically related data items. Structure help to organize complex data is a more meaningful way. It is powerful concept that we may after need to use in our program Design. A structure is combination of different data types using the & operator, the beginning address of structure can be determined. This is variable is of type structure, then & variable represent the starting address of that variable.

## STRUCTURE DEFINITION

A structure definition creates a format that may be used to declare structure variables consider the following example.

```
Struct book-bank
        {
        Char title [20];
        Char author [15];
        int pages;
        float price;
        };
```

Here keyword **Struct** hold the details of four fields these fields are title, author, pages, and price, these fields are called **structure elements**. Each element may belong to different types of data. Here **book-bank** is the name of the structure and is called the structure tag.

It simply describes as shown below.

| | Struct book-bank |
|---|---|
| Title | array of 20 charecters |
| Author | array of 15 charecters |
| Pages | integer |
| Price | float |

The general format of a structure definition is as follows

```
struct teg_name
{
        data_type     member 1;
        data_type     member 2;
          - - -           - - -
          - - -           - - -
          - - -           - - -                }
```

## ARRAY OF STRUCTURES

Each element of the array itself is a structure see the following example shown below. Here we want to store data of 5 persons for this purpose, we would be required to use 5 different structure variables, from sample1 to sample 5. To have 5 separate variable will be inconvenient.

```
# include < stdio.h>
main( )
{
        struct person
        {
                char name [25];
                char age;
                };
        struct person sample[5];
                int index;
                char into[8];
        for( index = 0; index <5; index ++)
                {
                        print("Enter name;");
                        gets(sample [index]. name);
                        printf("%Age;");
                        gets(info);
                        sample [index]. age = atoi (info);
                        }
                for (index = 0; index <5; index++)
                {
                        printf("name = %5\n", sample [index]. name);
                        printf("Age = %d \n", sample [index]. age);
                                getch( );
                                }
                                }
```

The structure type person is having 2 elements:

Name an array of 25 characters and character type variable **age**

## USING THE STATEMENT

Struct person sample[5]; we are declaring a 5 element array of structures. Here, each element of sample is a separate structure of type person.

We, then defined 2 variables into index and an array of 8 characters, **info.**

Here, the first loop executes 5 times, with the value of index varying from 0 to 4. The first printf statement displays. Enter name gets( ) function waits for the input string. For the first time this name you enter will go to sample[0]. name. The second printf display **age** the number you type is will be 5 stored as character type, because the member age is declared as character type. The function **atoi( )** converts this into an integer. atoi stands for **alpha to integer**. This will be store in sample[0] age. The second **for** loop in responsible for printing the information stored in the array of structures.

## STRUCTURES WITHIN STRUCTURES

Structure with in a structure means nesting of structures. Let us consider the following structure defined to store information about the salary of employees.

```
Struct salary
        {
                char name[20];
                char department[10];
                int basic_pay;
                int dearness_allowance;
                int city_allowance;
                }
                employee;
```

This structure defines name, department, basic pay and 3 kinds of allowance. we can group all the items related to allowance together and declare them under a substructure are shown below:

```
struct  salary
{
        char name [20];
        char department[10];
        struct
        {
        int dearness;
        int hous_rent;
        int city;
                }
                allowance;
        }
        employee;
```

The salary structure contains a member named **allowance** which itself is a structure with 3 members. The members contained in the inner, structure namely dearness, hous_rent, and city can be referred to as :

employee allowance. dearness
employee. allowance. hous_rent
employee. allowance. city

An inner-most member in a nested structure can be accessed by chaining all the concerned. Structure variables (from outer-most to inner-most) with the member using dot operator. The following being invalid.

employee. allowance (actual member is missing)
employee. hous_rent (inner structure variable is missing)

**Passing a Structure as a whole to a Function**

Structures are passed to functions by way of their pointers. Thus, the changes made to the structure members inside the function will be reflected even outside the function.

```
# include <stdio.h>
typedef  struct
        {
                char  *name;
                int acc_no;
                char acc_types;
                float balance;
        }       account;
main( )
{
        void change(account *pt);
        static account person = {“chetan”, 4323, ‘R’, 12.45};
        printf(“%s      %d      %c      %.2f \n”, person. name,
                        person. acc_type, person. acc_type, person.
                        balance);
change(&person);
        printf(“%s      %d  %c  %2f \n”, person. name, person. acc_type,person.
                acc-type, person. balance);
        getch( );
}
void change(account *pt)
{

pt - > name = Rohit    R”;
pt - > acc_no = 1111;
pt - > acc_type = ‘c’;
```

```
pt - > balance = 44.12;
return;

}
```

**Output**

```
chetan      4323        R     12.45
Rohit       R     1111  c     44.12
```

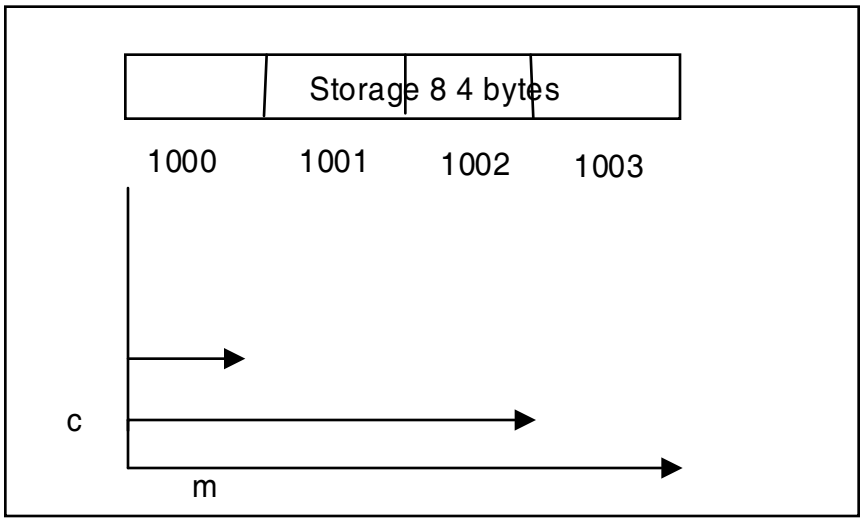
## UNIONS

Unions, like structure contain members, whose individual data types may vary. These is major distinction between them in terms of storage .In structures each member has its own storage location, where as all the members of a union use the same location.

Like structures, a union can be declared using the keyword **union** is follows:

```
union item
    {
    int m;
    float x;
    char c;
    } code;
```

This declares a variable **code** of type **union** them. The union contains them members, each with a different date type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.




The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. The above figure shown how all the three variables share the same

address, this assumes that a **float** variable requires 4 bytes of storage.

To access a union member, we can use the same syntax that we as for structure members, that is,

```
code.m
code.x
code.c          are all valid
```

Member variables, we should make sure that we can accessing the member whose value is currently storage. For example

```
code.m = 565;
code.x = 783.65;
printf("%d", code.m); would produce erroroneous output.
```

```
# include <stdio.h>
main( )
{
union
        {
        int one;
        char two;
        } val;
val.one = 300;
printf("val.one = %d \n", val.one);
printf("val.two = %d \n", val.two);
        }
```

The format of union is similar to structure, with the only difference in the keyword used.

The above example, we have 2 members **int one** and **char two** we have then initialised the member '**one**' to 300. Here we have initilised only one member of the **union**. Using two **printf** statements, then we are displaying the individual members of the union **val** as:

```
val.one = 300
val.two = 44
```

As we have not initialised the char variable **two,** the second **printf** statement will give a random value of 44.

The general formats of a union thus, can be shown as.

```
union tag {
        member 1;
        member 2;
          - - -
```

```
    - - -
member m;
        };
```

The general format for defining individual union variables:

**Storage-class Union tag** variable 1, variable 2,....., variable n;

**Storage-class** and **tag** are optional variable 1, variable 2 etc, are **union** variable of type **tag.** Declaring **union** and defining variables can be done at the same time as shown below:

```
Stroage-calss union teg {
        member 1;
        member 2;
          - - -
          - - -
          - - -
        member m;
        }
        variable 1, variable 2,   - - -  , variable n;
```

# CHAPTER - 7

# CONCEPTS OF OBJECT ORIENTED PROGRAMMING TECHNIQUES

## INTRODUCTION

Since the invention of the computer, many programming approaches have been tried. These included techniques such as modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each case has been the concern to handle the increasing complexity of programs that are reliable and maintainable. These techniques became popular among programmers over the last two decades.

With the advent of languages such as C, Structured programming became very popular and was the main technique of the 1980s.Structured programming was a powerful tool that enabled programmers to make moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object- Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by in incorporating the best of structured programming features with several powerful new concepts. It is a new way of organising and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OPP concepts easily.

Object- oriented programming was developed because limitations were discovered in earlier approach to programming To appreciate what OOP does, first all discuss what these limitations one and how they arose from traditional programming languages.

## PROCEDURE-ORIENTED PROGRAMMING

Conventional programming using high level languages such as COBOL, FORTRON, and C is commonly known as procedure- oriented programming. In the procedure - oriented approach, the problem is viewed as a sequence of things to be done, such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical program structure for procedural programming is shown in fig. 1. 1 . The technique of hierarchical decomposition has been used to specify the tasks to be completed in order to solve a problem.

Procedure-oriented programming basically consists of writing a list of instructions for the computer to follow, and organising these instructions into groups known as functions, We normally use a flowchart to organize these actions and represent the flow of control from one action to another.

While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them ?

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data- Figure 1.2 shows the relationship of data and functions in a procedure- oriented program.

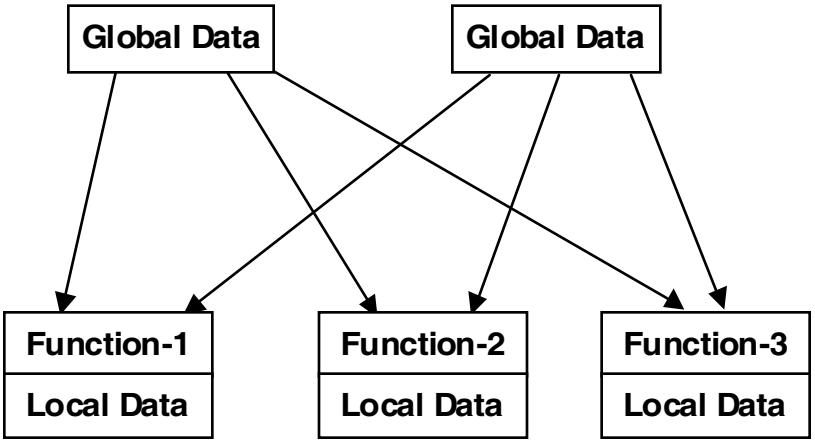



Fig.: Relationship of Data and Functions in Procedural Programming

Global data are more vulnerable to inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revse an external data structure, we should also revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

For example, suppose you are writing a program to create the elements of a graphics user interface, menus, windows, and so on. Quick now what functions will you need? what data structures? The answers are not obvious, to say the least. It would be better if windows and menus corresponded more closely to actual program elements.

Some characteristics exhibited by procedure-oriented programming are:

* Emphasis is on doing things (algorithms).
* Large programs are divided into smaller programs known as functions.
* Most of the functions share global data.

* Data move openly around the system from function to function.
* Functions transforms data from one form to another
* Employs top-down approach in program design.

## BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

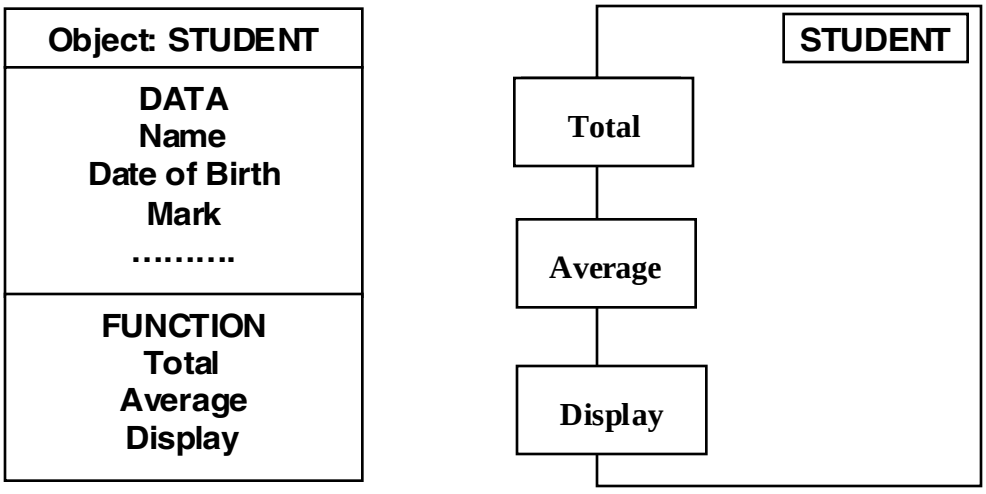
General concepts Used extensively in object-oriented programming are:

1. Object
2. Classes
3. Data abstraction
4. Data encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic binding
8. Message Passing

## OBJECTS

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle. They may also represent user defined data such as vectors, time and lists. Programming problem is analysed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects.

When a program is executed, the objects interact by sending massage to one another. For example, if "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance, Each object contains data and code to manipulate the data. Object can interact without having to know details of each other's data or code. It is sufficient to know the " of massage accepted and the type of response returned by the objects. Fig 1.4 shows two notations that are popularly used in object-oriented analysis and design.




Two ways of representing an object

## CLASSES

A class is an orgnisation of data and functions which operate on them. Data structures are called data members and the functions are called member functions, The combination of data members and member functions constitute a data object or simply an object.

In non-technical language, we can say that a class is a collection of similar object containing a set of shared characteristics.

For example , mango, apple and orange are members of the class fruit In a way, a class and its objects have the relationship of a data type and variables. A class is simply a template for holding objects. A class is abstract but objects are real. Simply by defining a class we don't create an object just like the mere declaration of a data " does not create variables.

One or more classes grouped together constitute a program. Once a class has been defined, we can create any number of objects belonging to that class. For example , the syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

fruit mango;

will create an object mango belonging to the class fruit.

A class is declared in the beginning of a program. A class can contain any number of data members and any number of member functions. A class can also have only data members.

## DATA ABSTRACTION AND ENCAPSULATION

The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those function which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the object that are to be created. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

## INHERITANCE

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying bird which is again a part of the class bird. As illustrated in Fig. 1.5 The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side effects into the rest of the classes.

It is noted that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

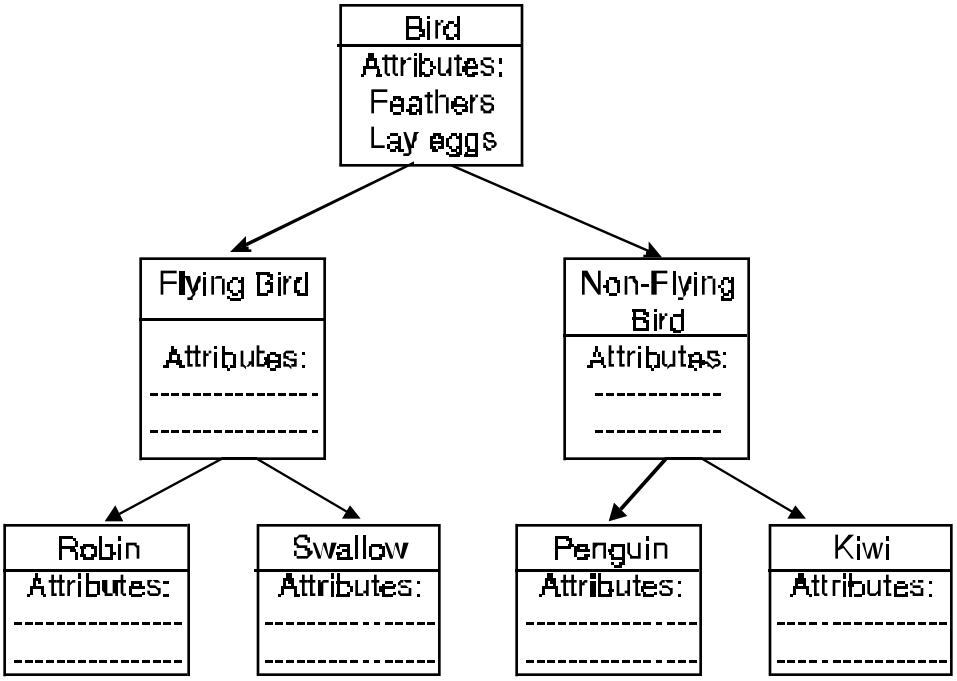



Fig.: Property inheritance

**POLYMORPHISM**

Polymorphism is another important OOP concept. Polymorphism mean the ability to take more than one form. For example, an operation may exhibit different behaviour in different instances The behaviour depends upon the types of data used in the operation. For example , consider the operation of addition. For two numbers, the operation will generate a sum.. If the operands are strings, then the operation would produce a third string by concatenation . Figure 1.6 illustrates that a single function name can be used to handle different number and different“s of arguments. This is something similar to a particular word having several different meanings depending on the context.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

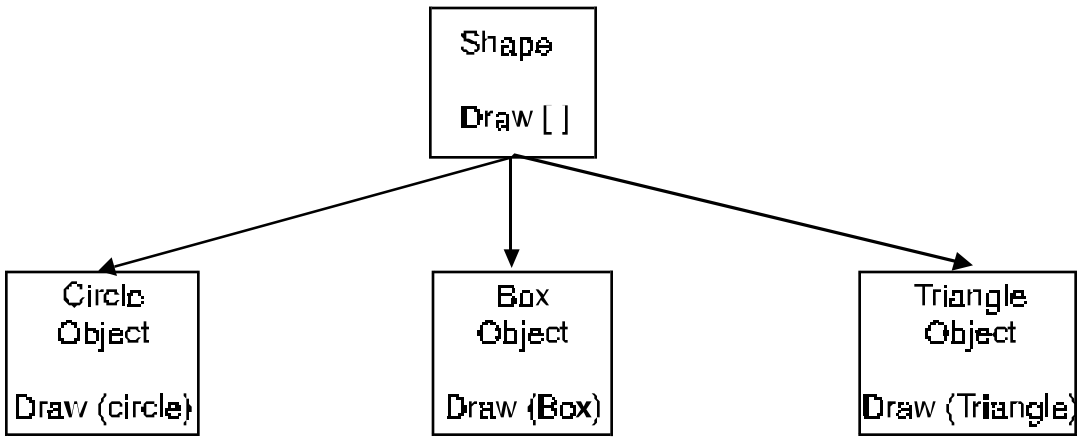



Fig.: Polymorphism

**DYNAMIC BINDING**

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic " of that reference.

Consider the procedure "draw" in fig I.6, by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

**MESSAGE COMMUNICATION**

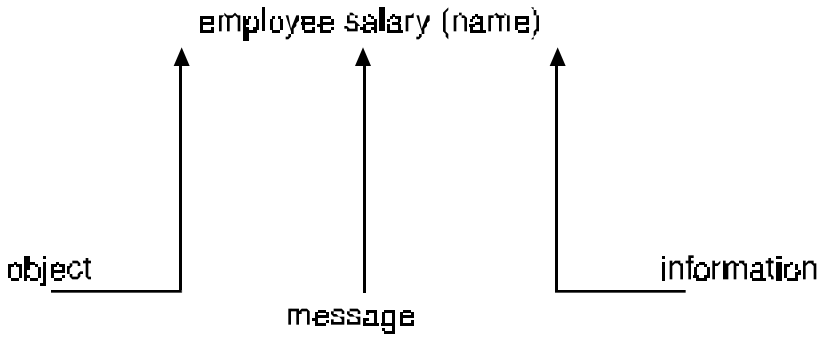
An object - oriented program consists of a set of objects that communicate with each other. The process of programming in an object- oriented language therefore involves the following basic steps:

1. Creating classes that define objects and their behaviour.
2. Creating objects from class definitions.
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure)in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

Example: **employee salary (name)**




Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

**BENEFITS OF OOP**

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

* Through inheritance, we can eliminate redundant code and extend the use of existing classes.

* We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

* The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

* It is possible to have multiple instances of an object to co-exist without any interference.

* It is easy to partition the work in project based on object.

* Object-oriented systems can be easily upgraded from small to large systems.

* Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.

* Software complexity can be easily managed.

# CHAPTER - 8

# C++ PROGRAMMING BASICS

## INTRODUCTION

C ++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

Since C++ allows us to create hierarchy-related object we can build special object-oriented libraries which can be used later by many programmers. C++ programs are easily maintainable and expendable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.

## BASIC PROGRAM CONSTRUCTION

Let us begin with a simple example of a c++ program that prints a string on the screen

```
# include <iostream.h>
void main( )
        {
cout << "every age has a language of its own";
        }
```

Despite its small size, this program demonstrates a great deal about the construction of C++ programs. Let's examine it in detail.

## FUNCTIONS

Functions are one of the fundamental building blocks of C++. The FIRST program consists almost entirely of a single function called main. The only part of this program that is not part of the function is the first line-the one that starts with include.

### Function Name

The parentheses following the word main are the distinguishing feature of a function. Without the parentheses the compiler would think that main referred to a variable or to some other program element. The word void preceding the function name indicates that this particular

function does not have a return value.

## BRACES AND THE FUNCTION BODY

The body of a function is surrounded by braces (sometimes called curly brackets). Every function must use this pair of braces. In this example there is only one statement within the braces: the line starting with cout. However, a function body can consist of many statements.

### Always Start with main ( )

When run a C++ program, the first statement executed will be at the beginning of a function called main ( ). The program may consist of many functions, classes, and other program elements, but on startup control always goes to main If there is no function called main in program, the linker will signal an error.

In most C++ programs, main ( ) calls member function in various objects to carry out the pro gram's real work. The main ( ) function may also contain call to other stand-alone functions.

## PROGRAM STATEMENTS

Ale program statement is the fundamental unit of C++ programming. 7lere's only one statement in the first program: the line

cout << "every age as a language of its own";

This statement tell the computer to display the quoted phrase. A semicolon signals the end of the statement. If you leave out the semicolon, the compiler will signal an error

## WHITE SPACE

Actually, the C++ compiler ignores white space almost completely. White space is defined as spaces, carriage returns linefeeds, tabs, vertical tabs, and form feeds. These characters are invisible to the compiler. There are actually several exceptions to the rule that white space is invisible to the compiler. The first line of the program, starting with # include , is a pre processor directive, which must be written on one line. Also string constants, such as "every age has a language of its own 'cannot be broken into separate lines.

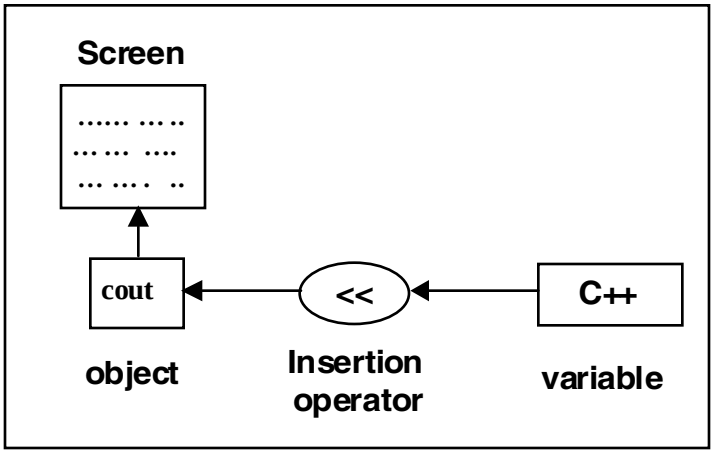
## OUTPUT USING COUT

The statement
cout <<"every age has a language of its own";

causes the phrase in quotation marks to be displayed on the screen. How does this work?

The identifier cout (pronounced "cout" ) is actually an object. it is predefined in
Turbo C++ to correspond to the standard output stream. A stream is an abstraction that refers

to a flow of data. The operator << is called the insertion or put to operator. It directs the contents of the variable on its right to the object on its left.




Output using Insertion Operator

**STRING CONSTANTS**

The phrase in quotation marks, “every age has a language of its own”, is an example of a string constant. As we’ll see later, on the one hand there is no real string variable type; instead you use an array of type char to hold string variables. On the other hand C++ recognizes string constants, surrounded by quotation marks as shown.

**PREPROCESSOR DIRECTIVES**

The first line of the FIRST program,

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

might look like a program statement, but it’s not. It isn’t part of a function body and doesn’t end with a semicolon, as program statements must. Instead, it starts with a number sign (#). It’s called a pre-processor directive. Recall that program statements are instructions to the computer. A pre-processor directive, on the other hand, is an instruction to the compiler itself A part of the compiler called the pre-processor deals with these directives before it begins the real compilation process.

**THE # include DIRECTIVE**

The pre-processor directive # include tells the compiler to insert another file into your source file. In effect, the # include directive is replaced by the contents of the file indicated.

## HEADER FILES

In the FIRST example the pre-processor directive # include tells the compiler to add the source file IOSTREAM.H to the source file before compiling. Why do this ? IOSTREAM.H is an example of a header file (sometimes called an include file ). It contains declarations that are needed by the cout indentifier and the << operator. Without these declarations, the compiler won't recognize cout and will think << is being used incorrectly.

## COMMENT SYNTAX

Comments start with a double slash symbol (//) and terminate at the end of the line. (This is on of the exceptions to the rule that the compiler ignores white space.) A comment can start at the beginning of the line or on the same line following a program statement.

## INTEGER VARIABLES

A variable is a symbolic name that can given a variety of values. Variables are stored in particular places in the computer's memory. When a variable is given a value, that value is actually placed in the memory space occupied by the variable. Most popular languages use the same general variable " such as integers, floating -point number and characters. Integer variables represent integer number like 1;30,000; and -27. Unlike floating-point numbers, integers have no fractional part:

## ASSIGNMENT STATEMENTS

The statements

```
var 1 = 20;
 var2 = varl+10;
```

assign values to the two variables. The equal sign =,causes the value on the right to be assigned to the variable on the left. In the first line shown here, var 1, which previously had no value, is given the value20.

## INTEGER CONSTANTS

Ale number 20 is an integer constant. Constant don't change during the course of the program. An integer constant consists of numerical digits, as shown. There can be no decimal point in an integer constant, and it must lie within the range of integers. In the second program line shown here, the plus sign + adds the value of var 1 and 10, in which 10 is another constant. The result of this addition is the assigned to var 2.

## CHARACTER VARIABLES

An other integer variable is type char. This " stores integer that range in value from - 128 to 127. Variables of this " occupy only one byte of memory. Character variables are sometimes used to store numbers that confine themselves to this limited range, but they are more commonly used to store ASCH characters.

## CHARACTER CONSTANTS

Character constants use single quotation marks around a character, as shown in the previous paragraph. When the C++ compiler encounters such a character constant, it translates it in to the corresponding ASCH code and stores that number in the program. The constant 'a' appearing in a program, for example, will be stored as 97.

## INITIALIZATION

Variables can be initialized at the same time they are defined.

INPUT WITH cin
The statement

```
cin >>number1;
```

is an input statement and causes the program to wait for the user to " in number. The number keyed in is placed in the variable number 1. The identifier cin (pronounced 'cin') is a predefined object in c+ that corresponds to the standard input stream. Here, this stream represents the keyboard. The operator >> is known as extraction or get from operator. It extracts (or takes ) the value from the keyboard an assigns it to the variable on its right (fig). This corresponds to the familiar scanf operation.

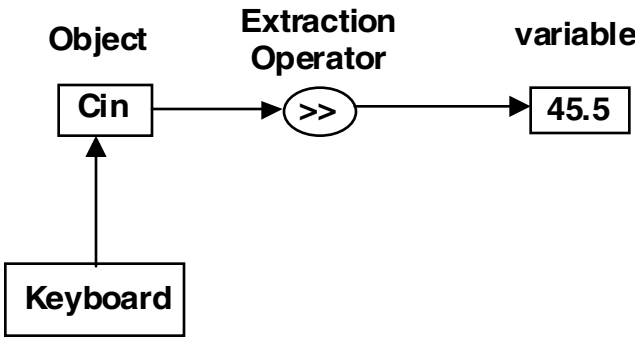



Fig.: Input using Extraction Operation

## CASCADING OR 1/0 OPERATORS

The statement
cout <<sum<< "\n";

first sends the string "sum =" to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items.

## EXPRESSIONS

Any arrangement of variables and operators that specifies a computation is called an expression. Thus alpha + 12 and (alpha -37)*beta/2 are expressions. When the computations specified in the expression are performed, the result is usually a value. Thus if alpha is 7, the first expressions shown has the value 19. Part of expressions may also be expressions.

## TYPE Float

Floating-point variables represent numbers with a decimal place- like 3.1415927,0.0000625, and 10.2.They have both an integer part to the left of the decimal point, and fractional part of the right .Floating - point variables represent real numbers, which are used for measurable quantities like distance, area temperature, and so on, and typically have a fractional part. There are three kinds of floating - point variables in c++' " float , type double, and " long double.

## THE # define DIRECTIVE

The line
define PI 3.14159
appearing at the beginning of your program specifies that the identifier PI will be replaced by the text 3.14159 throughout program.

## MANIPULATORS

Manipulators are operators used with the insertion operator n<< to modify - or manipulate- the way data is displayed. The most common manipulator are here: endl and setw

## THE endl MANIPULATOR

This is mainpulator that causes a linefeed to be inserted into the stream. It has the same effect as sending the single 'n' character.

## THE setw MANIPULATOR

The setw manipulator causes the number (or string) that follows it in the stream to be printed within a field n characters wide, where n is the argument to setw (n).The value is right-justified within the field.

## THE IOMANIP.H HEADER FILE

The declarations for the manipulators are not in the usual IOSTRF-AM.H header file, but in separate header file called IOMANIP.H. When you use these manipulators you must # include this header file in your program

**unsigned DATA TYPES**

By eliminating the sign of the character and integer types, you can change then range to start at 0 and include only positive numbers. This allows them to represent numbers twice as big as the signed type The unsigned type are used when the quantities represented are always positive - such as when representing a count of something - or when the positive range of the signed types is not quite long enough.

**TYPE CONVERSION**

Some time if variable of type int is multiplied by a variable of type float to yield a result of type double. This program compiles without error, the compiler considers it normal that you want to multiply (or perform any other arithmetic operation on) number of different types.

**ARITHMETIC OPERATORS**

C++ uses the four normal arithmetic operators +, and 1 for addition, subtraction , multiplication, and division. These operators work on all the data " both integer and floating-point . They are used in much the same way as in other languages, and are closely analogous to their use in algebra.

**THE REMAINDER OPERATOR**

There is a fifth arithmetic operator that works only with integer variables (" char, int and long). It's called the remainder operator, and is represented by % , the percent symbol. This operator (also called the modulus operator) finds the remainder when one number is divide by another.

**ARITHMETIC ASSIGNMENT OPERATORS**

C++ offers several ways to shorten and clarify your code. One of these is the arithmetic assignment operator. While not a key feature of the language, this operator is commonly used, and it helps to give c++ listings their distinctive appearance. The following kind of statement is common in most languages:
total = total + item; // adds "Item" to "total"

C++ offers a condensed approach: the arithmetic assignment operator, which combines an arithmetic operator and an assignment operator, and eliminates the repeated operand. Here's a statement that has exactly the same effect as the one above:
total =item; // adds "item" to "total"

There are arithmetic assignment operations corresponding to all the arithmetic operations: +=, =, and % ( and some other operators as well )

**INCREMENT OPERATORS**

Here's an even more specialised operator. You often need to add 1 to the value of an existing variable. You can do this the "normal" way. count = count + 1; // adds 1 to "count' or you can use

an arithmetic assignment operator:
count += 1; // adds 1 to “count”

But there’s an even more condensed approach:
++count; // adds 1 to “ count “
The ++ operator increments (adds 1 to ) its argument.

**PREFIX AND POSTFIX**

The increment operator can be used in two ways: as a prefix meaning that the operator precedes the variable; and as a postfix, meaning that the operator follows the variable. What’s the difference ? Often a variable is incremented within a statement that performs some other operation on it. For example,
Total weight = avg weight * ++count;
The question here is, Is the multiplication performed before or after count is incremented ? In this case count is incremented first. How do we know that ? Because prefix notation is used: ++count. If we had used postfix notation, count++ , the multiplication would have been performed first then count would have been incremented.

**THE DECREMENT(-) OPERATOR**

The decrement operator ,-, behaves very much like the increment operator, except that it subtracts 1 from its operand it can also be used in both prefix and postfix forms.

**LIBRARY FUNCTIONS**

Many activities in c++ are carried out library function. These functions perform file access, mathematical computations, graphics, memory management, and data conversion, among other things.

**HEADER FILES**

As with cout and other such objects, you must # include a header file that contains various declarations describing any library function you use. In the documentation for the sqrt () function, you’ll see that the specified header file in MA’TH.H. In SQRT the pre processor directive

include <math.h>

takes care of incorporating this header file into source file.

**LIBRARY FILES**

We mentioned earlier that a file containing library functions and objects will usually be linked to program to create an executable file.

# CHAPTER - 9

# LOOPING & BRANCHING

## RELATIONAL OPERATORS

A relational operator compares two values. The values can be any built-in c++ data type, such as char, int, and float, or they can be user - defined classes. The comparison involves such relationships as equal to, less than, greater than and so on. The result of the comparison is true or false; for example, either two values are equal (true), or they're not (false).

Here's the list of C++ relational operators:

| Operator | Meaning |
|---|---|
| > | Greater than |
| < | Less than |
| | Equal to |
| | Not equal to |
| >= | Greater than or equal to |
| <= | Less than or equal to |

## LOOPS

Loops cause a section of your program to be repeated a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statements following the loop. There are three kinds of loops in c++ : the for loop, the while loop, and the do loop.

## THE for LOOP

The for loop executes a section of code a fixed number of times. It's usually (although not always) used when you know, before entering the loop, how many times you want execute the code.

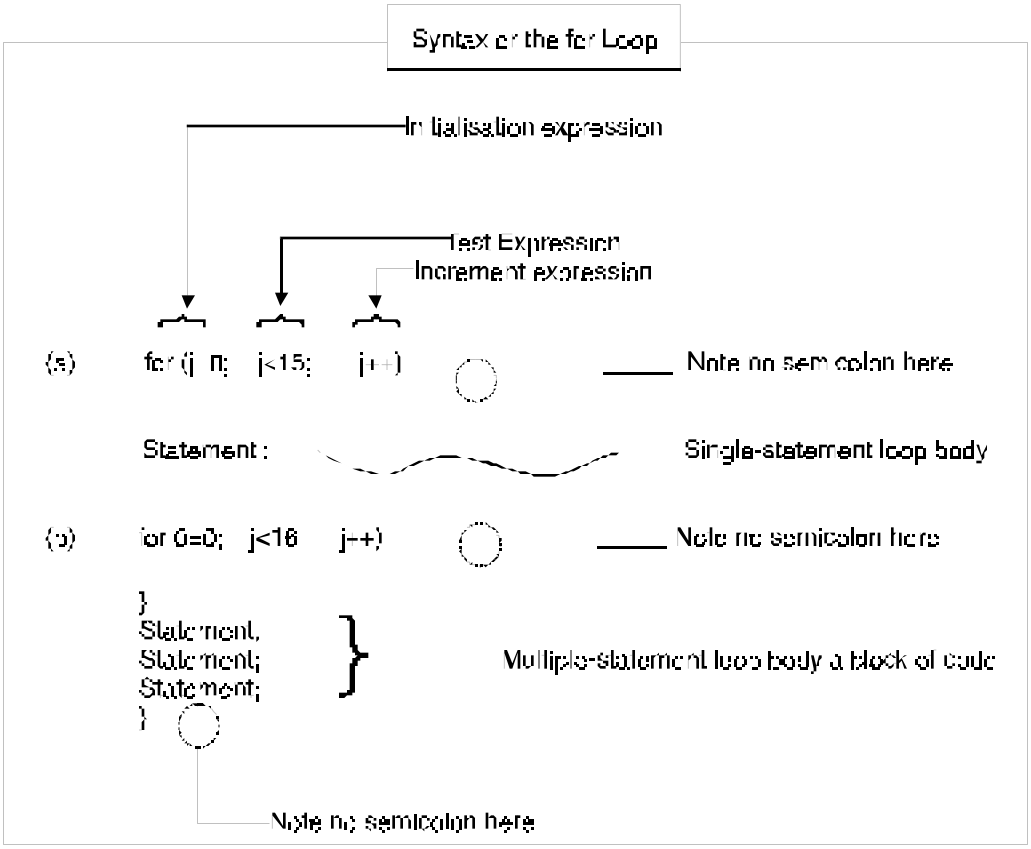

Syntax of the for Loop
Initialisation expression
Test Expression
Increment expression
(a) for (i=0; i<15; i++)
Note no semi colon here
Statement :
Single-statement loop body
(b) for 0=0; j<16 j++)
Note no semicolon here
{
Statement;
Statement;
Statement;
}
Multiple-statement loop body a block of code
Note no semicolon here

Fig.1

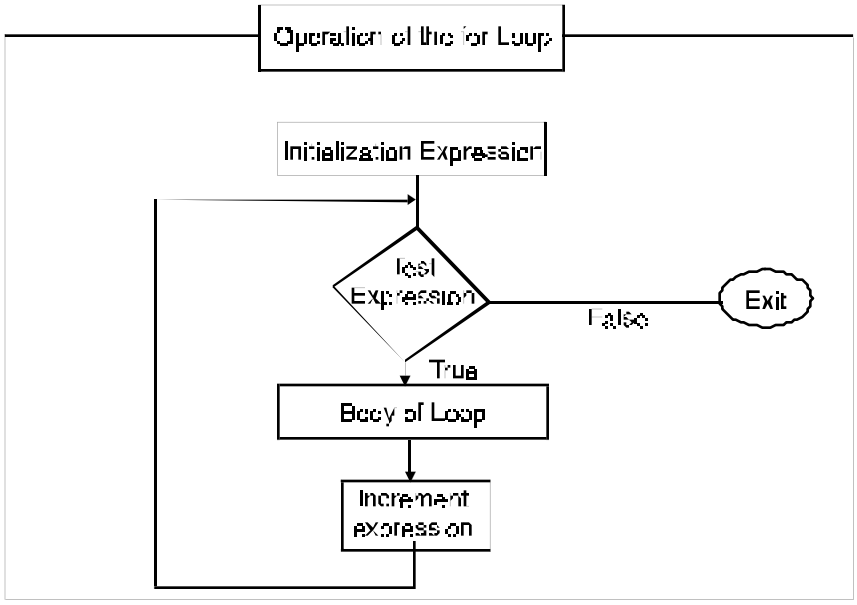

Operation of the for Loop
Initialization Expression
Test Expression
False
Exit
True
Body of Loop
Increment expression

Fig.2

**THE While LOOP**

The for loop does something a fixed number of times. What happens if you don't know how many times you want to do something before you start the loop? In this case a different kind of loop may be used:

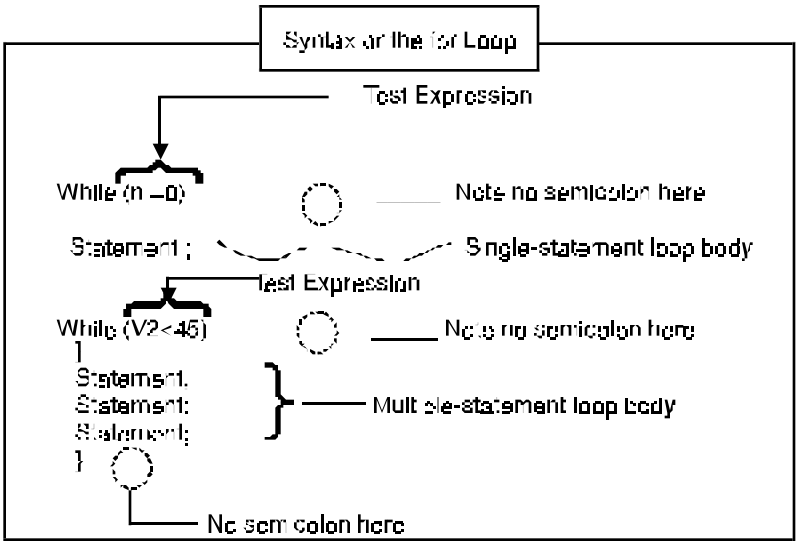



Fig.3

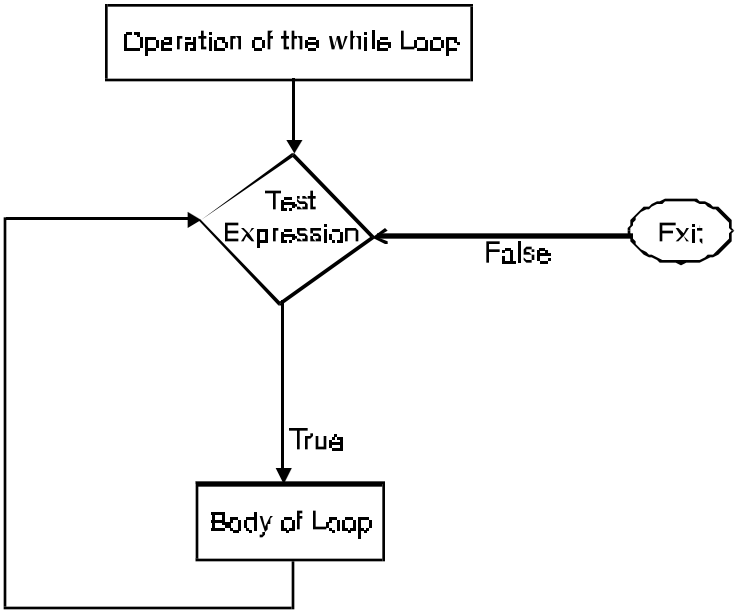



Fig.4

**THE do LOOP**

In a while loop, the test expression is evaluated at the beginning of die loop. If the test expression is false when loop is entered, the loop body won't be executed at all. Sometimes you want to guarantee that the loop body is executed at least once, no matter what the initial state of the test expression. When this is the case you should use the do loop, which places the test expression at the end of the loop.

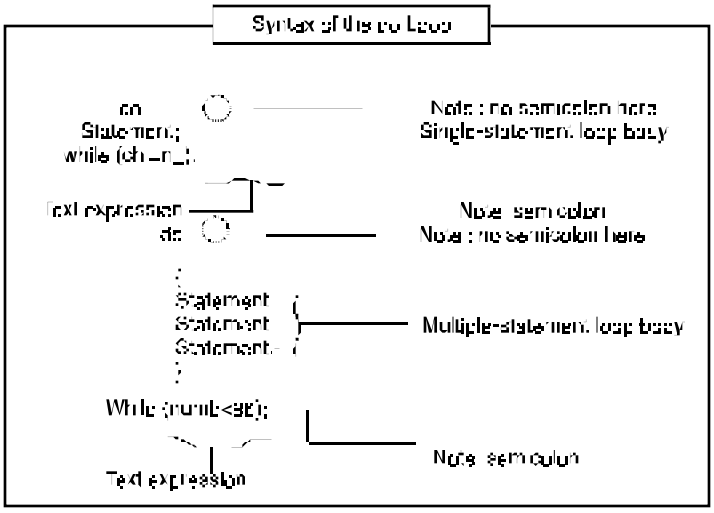



Fig.5

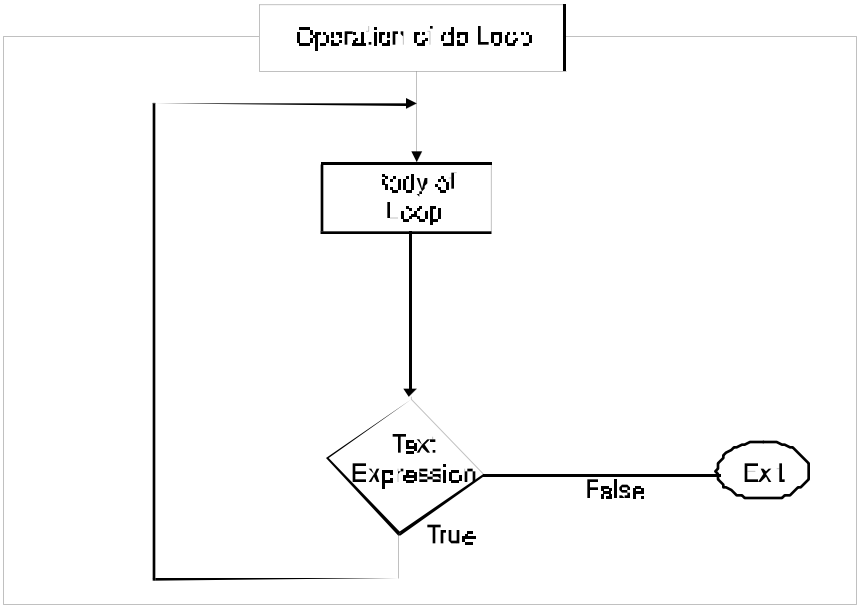



Fig.6

## DECISIONS

The Most important is with the if... else statement which chooses between two alternatives. This statement can be used without tile else, as a simple if statement. Another decision statement, switch, creates branches for multiple alternative sections of code, depending on the value of a single variable. Finally the conditional operator is used in specialised situations.

## THE if STATEMENT

The if statement is the simplest of the decision statements. Our next program, IFDEMO, provides an example.

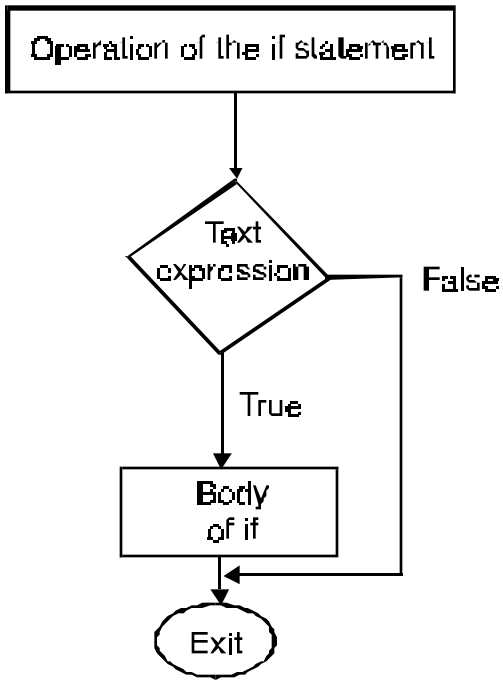



Fig.7

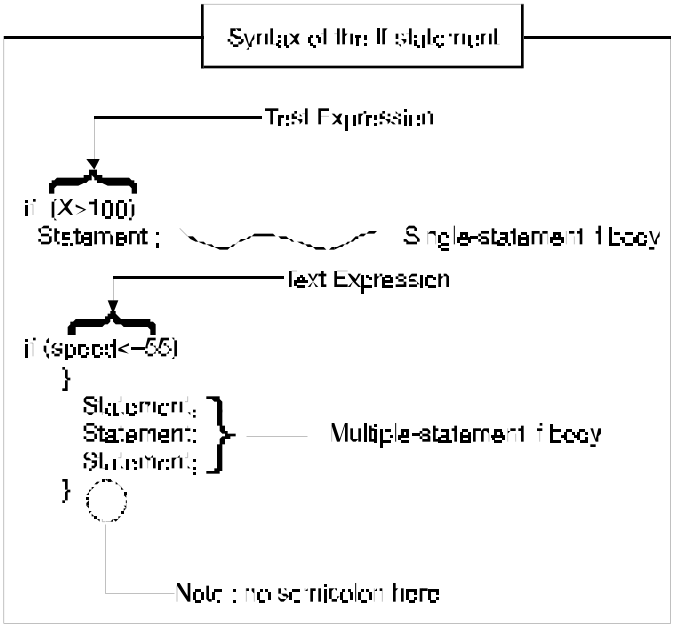



Fig.8

**THE if.....else STATEMENT**

The if statement lets you do something if a condition is true. If it is not true, nothing happen But suppose we want to do one thing if a condition is true, and do something else if it is false. That where the if..... else statement comes in. It consists of an if statement, followed by a statement or block of statements, followed by the keyword else, followed by another statement of block of statements.

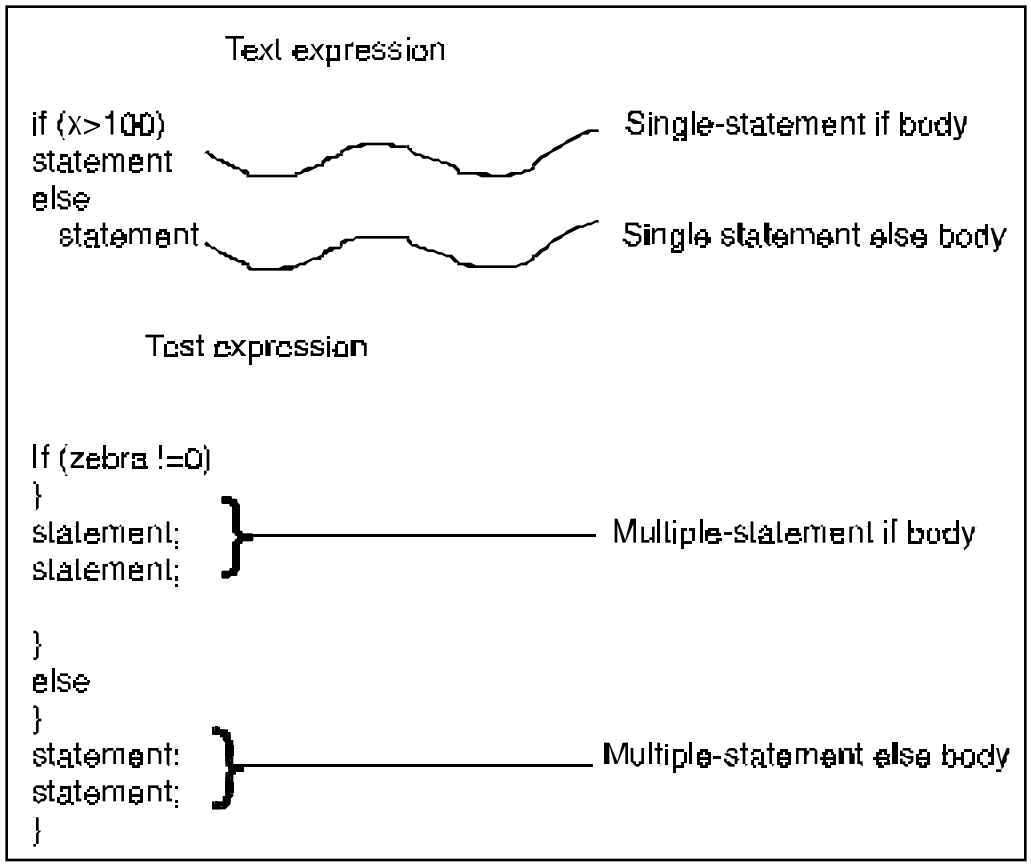

Text expression
if (x>100)
statement
else
statement
Single-statement if body
Single statement else body
Test expression
If (zebra !=0)
{
statement;
statement;
}
Multiple-statement if body
else
{
statement;
statement;
}
Multiple-statement else body

Fig.9

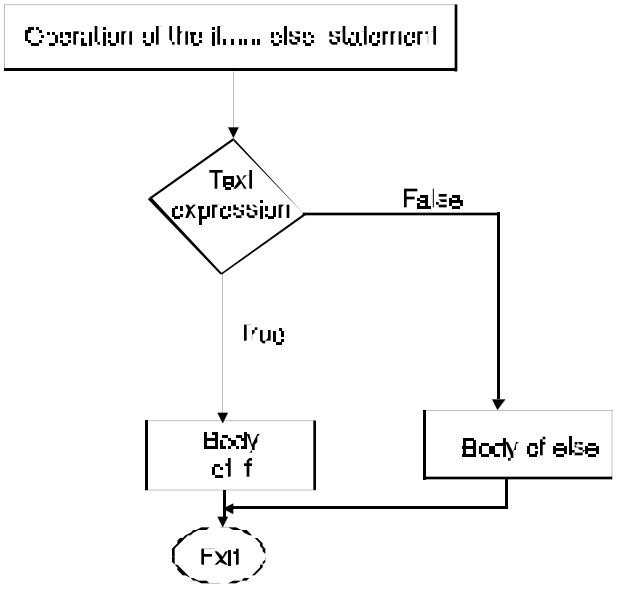

Operation of the if..... else statement
Test expression
False
True
Body of if
Body of else
Exit

Fig.10

## THE switch STATEMENT

If you have a large decision tree, and all the decisions depend on the value of the same variable, you will probably want to consider a switch statement instead of a series of if .... else or else if constructions.

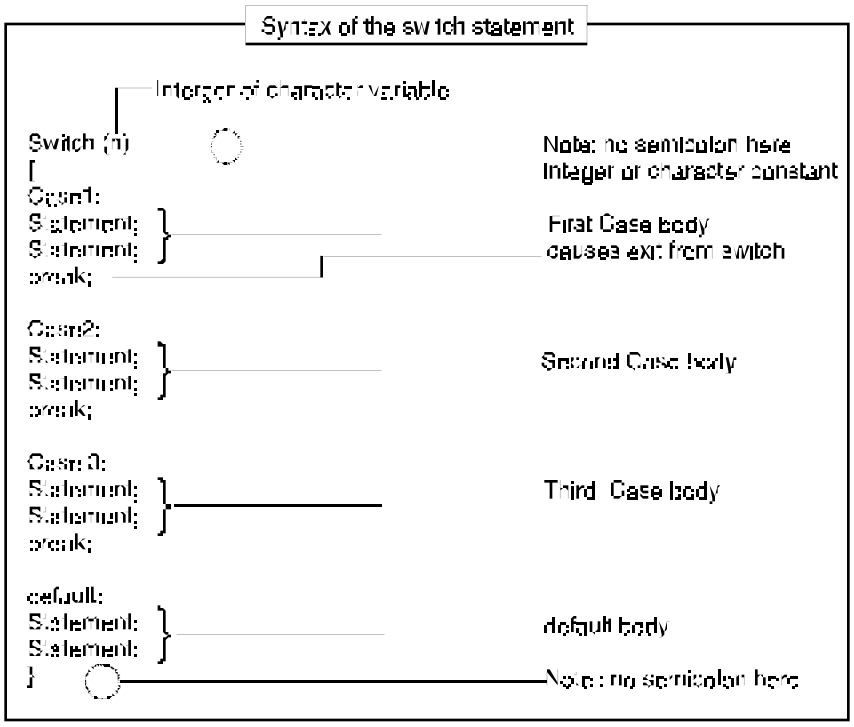



Fig.11

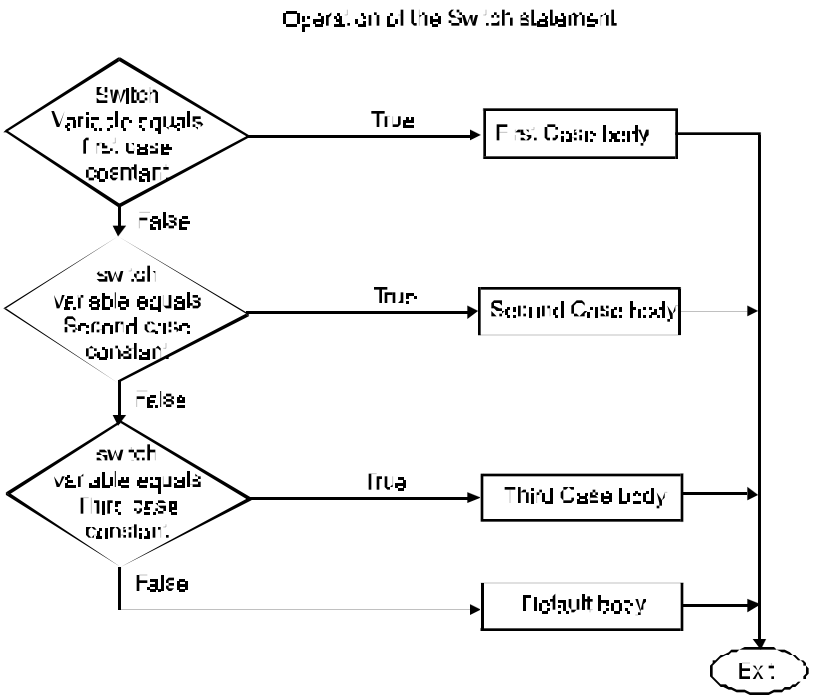



Fig.12

**THE break STATEMENT**

The break- statement cause an exit from a loop, statement after the break is executed is the statement following the loop.

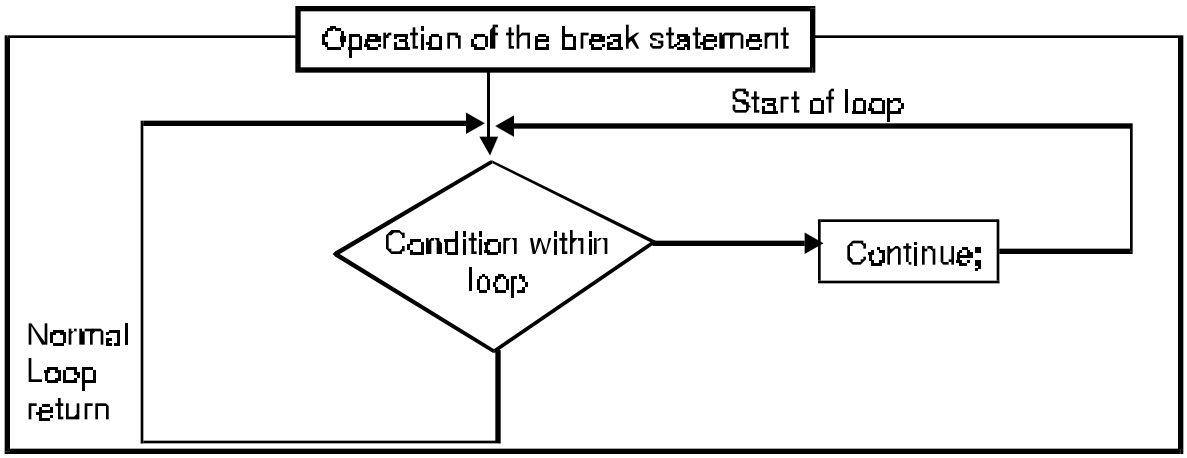



Fig.13

## THE CONDITIONAL OPERATOR

This operator consists of two symbols, which operate on three operands. It is the only sue operator in C ++, other operators operate on one or two operands.

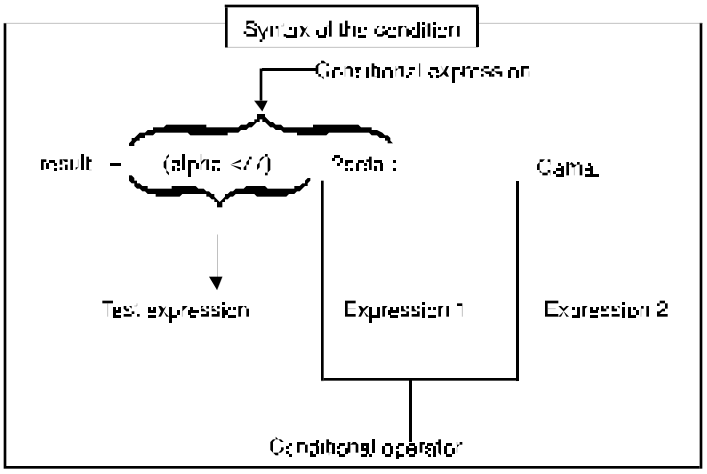



Fig.14

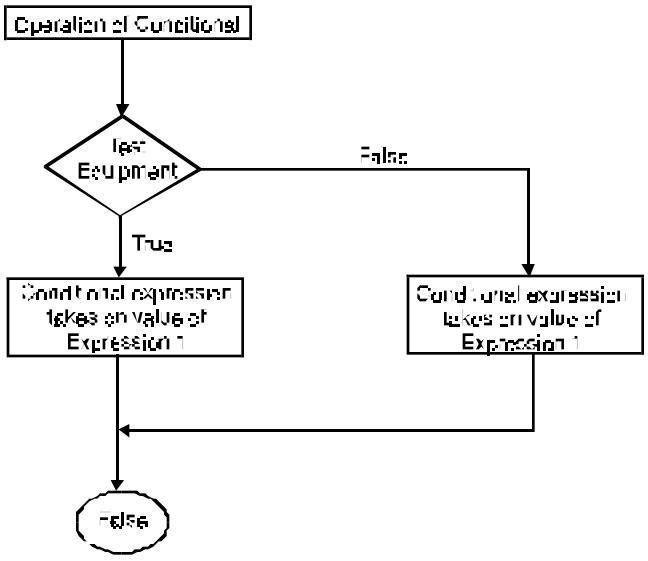



Fig.15

## LOGICAL OPERATORS

These operators allow you to logically combine Boolean (true/false) values. For example, today is a weekday has a Boolean value, since it is either true or false. Another Boolean expression is Maria took the car. We can connect these expressions logically; If today is a weekday,, and Maria took the car, then 1 will have to take the bus. The logical connection here is the word and, which provides a true or false value to the combination of the two phrases. Only if they are both true will I have to take the bus.

## OTHER CONTROL STATEMENTS

There are several other control statement in c++ we've already seen one break, used in switch statement , but it can be used other places as well. Another statement, continue, is used only in loop, and a third, go to should be avoided.

## THE CONTINUE STATEMENT

The break statement takes you out of the bottom of a loop. Sometime however, you want to go back to the top of the loop when something unexpected happens. Executing continue has this effect.

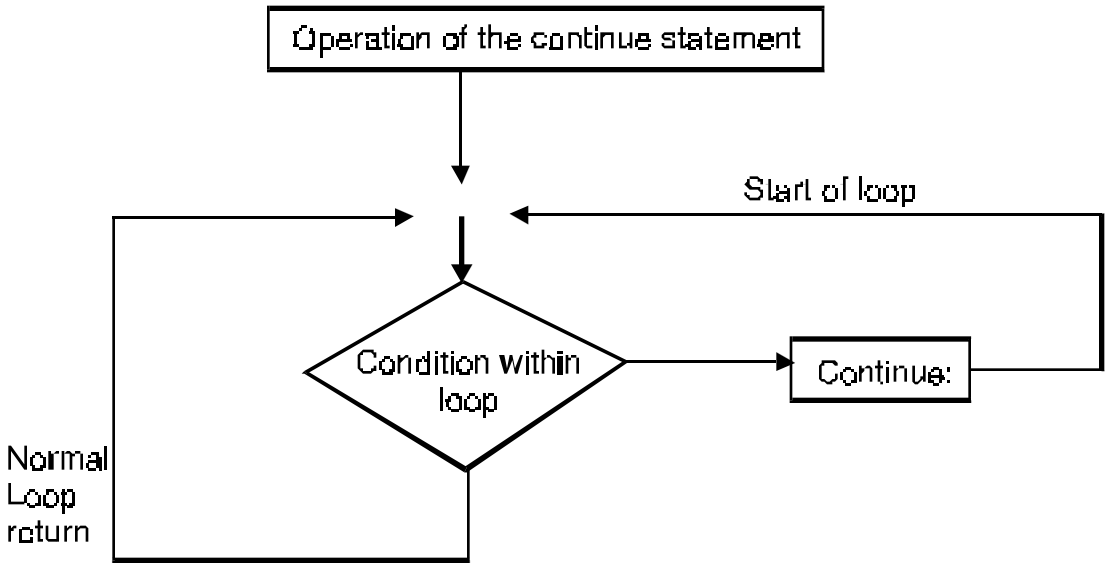



Fig.16

# CHAPTER - 10

# FUNCTIONS

## INTRODUCTION

Dividing a program into functions is one of the major principles of top-down, structured programming. Another 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++ is no exception. Functions continue to be the building blocks of C++ programs. In fact C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

## RETURN VALUES FROM FUNCTION:

In C++; the main ( ) retums a value of type int to the operating system. C++ therefore, explicitly defines main( ) as matching one of the following prototypes:
int main ( )
int main (int argc,char*argv[]);

The function that have a return value should use the return statement for termination. The main ( ) function in C++ is, therefore, defined as follows:

```
int main ( )
                {
                ……
                …...
return (0);
                }
```

Since the return type of functions is int by default the keyword int in the main header is optional. Most C++ compilers will generate an error or warning if there is no return statement. Turbo C++ issues the warning

Function should return a value

and then proceeds to compile the program. It is good programming practice to actually return a value from main ( ).

## FUNCTION PROTOTYPING

Function prototyping is one of the major improvement added to C++ functions. Function prototype is a declaration statement in the calling program and is of the following form: The argument-list contains the types and names of arguments that must be passed to the function. Note that each argument variable must be declared independently inside the parentheses. That is a combined declaration like is illegal.

float volume (int x, float y, z);

In a function declaration, names of the arguments are dummy variables and therefore, they are optional. That is, the form: is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just acts as placeholders and, therefore, if names are use they don't have to match the names used in the function call or function definition. In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume (int a, float b, float c)
                    {
                    float v =a*b*c;
                          ……..
                          ……..
                    }
```

The function volume ( ) can be invoked in a program as follows:

```
cube1 = volume ( b1,w1, h1); // function call
```

The variable b1, w1, and h1 are known as the actual parameters which specify the dimensions of cube1. Their types (which have been declared earlier) should match witch with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an empty argument list as in the following example:

```
void display( );
```

In C++, this means that the function does not pass any parameters. It is identical to the statement `void display (void);`

A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do-something(....)
```

## CALL BY REFERENCE

Provision of the reference variables in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called, function

become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int & a, int & b)
        {
                int t=a;    //Dynamic initialization
                a=b;
                b=t,
        }
```

Now, if m and n are two integer variables, then the function call

```
swap (m, n);
```

will exchange the values of m and n using their aliases (reference variables) a and b.

**RETURN BY REFERENCE**

A function can also return a reference. Consider the following function:

```
max(int&x,int&y)
        {
                if (X>Y)
                return x;
                else
                return y;
          }
```

Since the return " of max ( ) is int &, the function reference to x or y (and not the values). Then a function call such as max (a, b) will yield a reference to either a or b depending on their values. This means that this function call can appear on the left -hand side of an assignment statement.

That is the statement

```
max(a,b)=-1;
```

is legal and assigns- 1 to a if it is larger, otherwise- 1 to b.

**INLINE FUNCTIONS**

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as macros. Pre-processor macros are popular in 'C'. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation. C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function. An inline function is a function that is

expanded in line when it is invoked. That is the compiler replaces the function call with the corresponding function code (something similar to macros expansion). The inline functions are defined as follows:
Example:

```
inline double cube(double a)
{
        return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c=cube(3.0);
d=cube(2.5+1.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5+1.5; the function passes the value of the expression, 4 in this case. This makes the inline feature fir superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword inline to the function definition. All inline functions must be defined before they are called.

We should take care before making a function inline. The speed benefits of inline functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline function may be lost. In such cases, the use of normal functions will be more meaningful. Usually the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube (double a){return(a*a*a);}
```

Remember that the inline keyword merely sends a request, not command to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch, or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If function contain static variables.
4. If line functions are recursive.

**DEFAULT ARGUMENTS**

C++ allows us to call a function without specifying all its arguments. In such cases the function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

Here is an example of prototype (i.e. function declaration) with default values-.

```
float amount (float principal, int period, float rate=0.15)-,
```

The default value is specified in a manner syntactically similar to variable initialization. The above prototype declares a default value of 0. 15 to the argument rate. A subsequent function call like

```
value=amount (5000,7); // one argument missing
```

passes the value of 5000 to principal and 7 to period and then lets the function use default value of 0.15 for rate. the call

```
value= amount (5000,5,0.12); // no missing argument
```

passes an explicit value of 0.12 to rate,

A default argument is checked for type at the time of declaration and calculated at the time of call. One important point to note is that only the trailing arguments can have default value. That is, we must add defaults from right to left. We cannot provide default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i,int j=5,int k=10);          //legal
int mul(int i,=5,int j);      //legal
int mul(int i,=0,int j,int k=10);      //legal
int mul(int i,=2,in t=5,int k=10);     //legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation.

**FUNCTION OVERLOADING**

Over loading refers to the use of the same thing for different purpose. 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. This is known as function polymorphism in OOP.

Using the concept of function overloading, we can design a family of functions with one function name but with deferent argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function ".For example, an overloaded add function handles different types of data as shown below:

```
// Declarations
int add (int a, int b);                    // prototype 1
int add(int a, int b, int c,)              // prototype 2
double add (double x,double y);            // prototype 3
double add(int p, double q);               // prototype 4
double add(double p, int q);               // prototype 5

// Function calls

cout << add (5,10);                        // uses prototype 1
cout << add (15,10.0);                     // uses prototype 2
cout << add (12.5,7.5);                    // uses prototype 3
cout << add (5,10,15);                     // uses prototype 4
cout << add (0.75,5);                      // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique.

# CHAPTER - 11

# OPERATOR OVERLOADING

## INTRODUCTION

Operator overloading is one of the most exciting features of the most exciting features of object-oriented programming. It can transform complex and obscure program listings into intutively obvious ones.

For example, a statement like

d3.addobjects(d1, d2);

Can be changed to the much more readable form as
d3 = d1 + d2;

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

As we all know, + and = operators can be used with a variety of data types like int, float char etc. These two operators are implicitly overloaded already, in that the same operator can be used with simple data “s. When you use + with int values, one specific set of machine language instructions is invoked. When used with float values, a completely different set of machine language instructions are invoked. If you find yourself limited by the way the C++ operators work, you can change them to do whatever you like. By using classes to create new kinds of variables, and operator overloading to create new definitions for operations, you can extend C++ to be a new language of your own design.

## OVERLOADING UNARY OPERATORS

Here is the newest version of the counter class, rewritten to use an overloaded increment operator. The program which defines the overloaded ++ operator **is** shown below

```
//program to illustrate(++) operator overloading
#include <iostream.h>
class counter
```

```
                    {
private:
unsigned int count;
public:
counter(
{count = ( ); //constructor
int getcount( )
{return count;} //return count
void operator++ ( )
{count++;} //increment count

void main( )
                    {
counter c1, c2;     //define and initialise

cout<<"value of c1: "<<c1.getcount( )<<endl;
cout<<"value of c2:"<<c2. Getcount( )<<endl;
c1++; c1++; c2++; c2++; c2++; c2++; c2++;
cout<<"value of c1:"<<c1. getcount( )<<endl;
cout<<"value of c2:"<<c2.getcount( )<<endl;
```

The output appears as

```
value of c1:0
value of c2: 0
value of c1:3
value of c2:5
```

A new function is defined with the name operator. The word operator is a keyword. It is preceded by the return type void. The operator to be overloaded is immediately written after this name followed by the void function symbol as operator ++( ).This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand is of " counter. In main( ) the operator is applied to a specific object as shown. Yet operator ++( ) takes no arguments. What does this operator increment? The operator ++ function has a subtle defect which you will see if you tried to write c1 = c2++;

To make it work, we must provide a way for it to return a value. The program shown below does this.

```
//program to illustrate overloaded++ operator
#Include <iostream.h>
class counter
                        {
                        private:
unsigned int count;
                        public:
counter ( ) {count= 0;}     //constructor
int getcount( ) {return count;}     //function which does the job
{count++;                   //increment count
```

```
counter temp;                 //make a temporary counter
temp.count=count;             //gives it the same value as the object
return temp;                  //return the copy

void main( )
{ counter c1, c2; //c1 = 0 and c2 = 0
cout<<endl<<"c1 = "<<c1.getcount( );     //display the initial values
                              cout<<endl<<"c2="<<c2.getcount( );
                              c1++;     //c1 = 1
                              c2 =c1++;       //c1 = 2 and c2=2
                              cout<<endl<<"c1+"<<c1.getcount( );     //display again
                              cout<<endl<<"c2+"<<c2.getcount( );
                              cout<<endl<<"c2 = "<<c2++.getcount( );
```

The output appears as

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

In this program, the operator++ ( ) function creates a new object of type counter called temp, to be used as a return value. It increments the count data in its own object as before but creates the new temp object and assigns count in the new object the same value as in its own object. Finally, it returns the temp object. This enables us to write statements like

```
c2 = c1++;
c2.getcount( )
```

In the first of these statements, the value returned from c1++ will be assigned to c2. In the second, it will be used to display itself using the getcount( ) member function

**LIMITATION OF INCREMENT OPERATORS**

In our example programs, we showed both postfix and prefix notations. Prefix notation (++c2) causes a basic variable to be incremented before its value is used in an expression. Postfix notation (c2++) causes it to be incremented after its value is used. When ++ and - are overloaded, there is no distinction between prefix and postfix notation. The following expressions have exactly the same effect;

```
c2 = c1++;
c2 = ++c1:
```

**OVERLOADING BINARY OPERATORS**

In our earlier program we have shown how two distance objects could be added using a member function as shown below

dist3. adddist(dist1, dist2),

By overloading the + operator, we can reduce this expression as

dist3.adddistdist1+ dist2;

The program below illustrates overloading the + operator.

```
//program for overloading of + operator
#include <iostream.h>
class distance
{
        private:
        int feet;
        double inches;
        public:
        distance( )
        {feet = 0; inches = 0;}
        distance(int ft, double in)
        {feet=ft; inches = in;}
        void getdist( )
        {cout<<endl<<“enter feet:”; cin>>feet;
        cout<<endl<<“enter inches:”; cin >>Inches;}
        void showdist( )
        {cout<<feet<<“\’-”<<inches<<‘ \”;}
        distance operator +(distance);
}
        distance distance::operator + (distance d2)
        {int f = feet + d2.feet;
        double i = inches + d2.inches;
        if(i>=12.0)
        {i-=12.0;

        return distance(f,i);
        void main( )
        {distance dist1,dist3, dist4,
        dist1.getdist( );
        distance dist2(11,6.25);
        dist3 = dist1 + dist2;
        dist4 = dist1 + dist2 + dist3;
        cout<<endl<<“dist1 is:”, dist1.showdist( )
        cout<<endl<<“dist2 is:”, dist2. showdist( )
        cout<<endl<<“dist3 is:”, dist3.showdist( )
        cout<<endl<<“dist4 is:”, dist4.showdist( )
```

The output appears as

```
enter feet: 10
enter inches: 6.5
dist1 is: 10'-6.5"
dist2 is: 11'-6.25"
dist3 is 22'- 0.75"
```

dist4 is 44'-1.5"

The function operator+ (distance d2) uses only one argument, since overloaded operator always require one less argument than the number of operands.

Similar functions could be created to overload other arithmetic operators in the distance class. You could create inturive operators for each of these:

-,* and /.

**CONVERSION BETWEEN DATA TYPES**

We have discussed the way in which many operators can be overloaded, but not the assignment operator (=). This is a special operator with complex properties. Of course, the = operator will assign a value from one variable to another in a statement like

intvar1 = intvar2;

where intvar1 and intvar2 are integer variables.

The = operator can also assign the value of one user defined object to another, provided they are of the same ". This is possible in

dist3 = dist + dist2;

where the result of the addition (which is of " distance) is assigned to an- other object of distance d3. The assignments between basic types or between user defined types are handled by the compiler with no effort on our part provided the same data type is used on both sides of the assignment operator. But what happens when the variables on either side of the = operator are of different types? We have to explore this further. When we write a statement like

intvar = floatvar;

where intvar is of int type and floatvar is of float type, there is an implicit type conversion from float to int by means of the operator(=) and the compiler will call the built in routine to do this job There are many such possible conversions-from float to double, char to float and so on. Each such conversion has its own routine built into the compiler. The correct conversion routine is automatically called when data "s on either side of the = operator differ. Sometimes we want to force the compiler to convert one " to another. To do this we use the cast operator. For example, to convert float to int we could write intvar=int(floatvar); Casting provides explicit conversion- Such explicit conversions also use the built- in routines as in implicit conversions.

# CHAPTER - 12

# FILE HANDLING

## INTRODUCTION

In this section, we will discuss about files which are very important for large-scale data processing.

Data are stored in data files and programs are stored in program files. A file is simply a machine decipherable storage media where programs and data are stored for machine usage. In C++ we say data flows as streams into and out of programs. There are different kinds of streams of data flow for input and output. Each stream is associated with a class, which contains member functions and definitions for dealing with that particular kind of flow. For example, the if stream class represents the input disc files,. Thus each file in C++ is an object of a particular stream class.

## THE STREAM CLASS HIERARCHY

The stream classes are arranged in a rather complex hierarchy. You do not need to understand this hierarchy in detail to program basic file I/O, but a brief overview may be helpful. We have already made extensive use of some of these classes. The extraction operator >> is a member of istream class and the insertion operator is a member of ostream class. Both of these classes are derived from the ios class. The cout object is a predefined object of the ostream with assign class. It is in turn derived from ostream class. The classes used for input and output to the video display and keyboard are declared in the header file IOSTREAM.H, which we have routinely included in all our programs.

## STREAM CLASSES

The ios class is the base class for the entire I/O hierarchy. It contains many constants and member functions common to input and output operations of all kinds.

The istream and ostream classes are derived from ios and are dedicated to input and output respectively Their member functions perform both formatted and unformatted operations. The iostream class is derived from both istream and ostream by multiple inheritance, so that other classes can inherit both of these classes from it. The classes in which we are most interested for file I/O are ifstream for input files ofsteam for output files and fstream for files that will be

used for both input and output the ifstream and ofsteam classes are declared in the FSTREAM.H. file.

The isteam class contains input functions such as
getline( )
getine( )
read( )
and overloaded extraction operators.
The ostream class contains functions such as

Put( )
write( )
and overloaded insertor.

**WRITING STRINGS**

Let us now consider a program which writes strings in a file.

```
//program for writing a string in a file
#include<fstream.h>
void main( )
{ofstream outfile("fl.fil");//create a file for output
outfile<<"harmlessness, truthfulness, calm"<<endl;
outfile<<"renunciation, absence of wrath and fault-finding"<<endl; outfile<<"compassion for all, non-covetousness, gentleness,
                modesty"<<endl;
outfile<<"stability. vigour, forgiveness, endurance, cleanliness"<<endl; outfile<<"absence of malice and of excessive self-esteem"<<endl; outfile<<"these are the qualities of godmen"<<endl;
```

In the above program, we create an object called outfile, which is a member of the output file stream class. We initialise it to the filename "fl.fil". You can think of outfile as a user-chosen logical name which is associated with the real file on disc called "fl.fil".

When any automatic object (outfile is automatic) is defined in a function, it is created in the function and automatically destroyed when the function terminates.

When our main ( ) function ends, outfile goes out of scope. This automatically calls the destructor, which closes the file. It may be noticed that we do not need to close the file explicitly by any close-file command.

The insertion operator << is overloaded in ofsteam and works with objects defined from ofstream. Thus, we can use it to output txt to the file. The strings are written in the file "fl. fil' in the ASCII mode. One can see it from DOS by giving the type command.

The file "fl. fil" looks as shown below

harmlessness, truthfulness, calm
renunciation, absence of wrath and fault-finding
compassion for all, non-covetousness, gentleness, modesty
stability, vigour, forgiveness, endurance, cleanliness
absence of malice and of excessive self-esteem
these are the qualities of godmen.

**READING OF STRINGS**

The program below illustrates the creation of an object of ifstream class for reading purpose.

```
//program of reading strings
#include <fstream.h>      //for file functions
void main( )
{const int max = 80;    //size of buffer
char buffer[max];    //character buffer
ifstream. infile("fl.fil")-   //create file for input
while (infile)  //until end-of-file
{infile.getline(buffer,max); //read a line of text
cout<<buffer}
```

We define infile as an ifstream object to input records from the file "fl.fil". The insertion operator does not work here. Instead, we read the text from the file, one line at a time, using the getline( ) function. The getline ( ) function reads characters until it encounters the ' \n' character. It places the resulting string in the buffer supplied as an argument. The maximum size of the buffer is given as the second argument. The contents of each line are displayed after each line is input. Our ifstream object called infile has a value that can be tested for various error conditions -one is the end-of-file. The program checks for the EOF in the while loop so that it can stop reading after the last string.

**What is a buffer?**

A buffer is a temporary holding area in memory which acts as an intermediary between a program and a file or other I/0 device. Information can be transferred between a buffer and a file using large chunks of data of the size most efficiently handled by devices like disc drives. Typically, devices like discs transfer information in blocks of 512 bytes or more, while program often processes information one byte at a time. The buffer helps match these two desperate rates of information transfer. On output, a program first fills the buffer and then transfers the entire block of data to a hard disc, thus clearing the buffer for the next batch of output. C++ handles input by connecting a buffered stream to a program and to its source of input. similarly, C++ handles output by connecting a buffered stream to a program and to its output target.

**Using put( ) and get( ) for writing and reading characters**

The put ( ) and get( ) functions are also members of ostream and istream. These are used to output and input a single character at a time. The program shown below is intended to illustrate the use of writing one character at a time in a file.

```
//program for writing characters
#Include <fstream.h>
#include <string.h>
void main( )

charstr[] = “do unto others as you would be done by ;
ofstream outfile(“f2.fil”);
for(int i =0; i<strlen(str); i++)
outfile put(str[i]);
}
```

In this program, the length of the string is found by the strlen( ) function, and the characters are output using put( function in a for loop. This file is also an ASCII file.

**READING CHARACTERS**

The program shown below illustrates the reading of characters from a file.

```
//program for reading characters of a string
#Include <fstream.h>
void main( )
{char ch;
ifstream in file(“f2.fil”)’
while(infile)
infile.get(ch);
cout<<ch;
}
```

The program uses the get( ) and continues to read until eof is reached. Each character read from the file is displayed using cout. The contents of file f2.fil created in the last program will be displayed on the screen.

**WRITING AN OBJECT IN A FILE**

Since C++ is an object-oriented language, it is reasonable to wonder how objects can be written to and read from the disc. The program given below is intended to write an object in a file.

```
//program for writing objects in files
#include<fstream.h>
class employees
{
        protected:
        int empno;
        char name[10];
        char dept[5];
        char desig[5];
```

```
        double basic;
        double deds;
        public:
        void getdata(void)
        {coul<<endl<<"enter empno";cin>>empno;"
        cout<<endl<<"enter empname";cin>>name;

        cout<<endl<<"enter department ";
        cin>>dept; cout<<endl<<"enter designation ";
        cin>>desig; cout<<endl<<"enter basic pay ";cin>>basic:
        cout<<endl<<"enter deds ";cin>>deds;
}

        void main(void)
        {employees emp;
        emp.getdata( );
        ofstream outfile("f3.fil");
        outfile. write((char * )&emp,sizeof(emp));
```

This program uses a class by name employees and an object by name emp. Data can be written only by the function inside the object. This program creates a binary data file by name f3.fil. The write( ) function is used for writing.

The write( ) function requires two arguments, the address of the object to be written, and the size of the object in bytes. We use the size of operator to find the length of the emp object. The address of the object must be cast to type pointer to char.

**BINARY VERSUS CHARACTER FIILES**

You might have noticed that write( ) was used to output binary values, not just characters. To clarify, let us examine this further. Our emp object contained one int data member, three string data members and two double data members. The total number of bytes occupied by the data members comes to 38.

It is as if write( ) took a mirror image of the bytes of information in memory and copied them directly to disc, without bothering any intervening translation or formatting. By contrast, the character based functions take some liberties with the data. for example, they expand the' \n 'character into a carriage return and a line feed before storing it to disk.

**READING AN OBJECT FROM THE FILE**

The program given below is intended to read the file created in the above program.

```
//program for reading data files
#include < istream.h>
class employees
```

```
{
        protected:
        int empno;
        char name[I0];
        char dept[5];
        char desig[5];
        double basic;
        double deds;

        public:
        void showdata(void)
        {cout<<endl<<“employeenumber: “<<empno;
        cout<<endl<<“empname “<<name;
        cout<<endl<<“department “<<dept;
        cout<<endl<<“designation “<<desig;
        cout<<endl<<“basic pay “<<basic:
        cout<<endl<<“deds     “<<deds;}

        void main(void)
        {employees empl;
        ifstream infile(“f3.fil”);
        infile.read((char*)&empl, sizeof(empl));
        empl.showdata( );
}
```

It may be noticed that both read( )and write( functions have similar argument. we must specify the address in which the disc input will be placed. We also use size of to indicate the number of bytes to be read.

The sample output looks as shown below:

```
        employeenumber;         123
        empname                 venkatesa
        department              elec
        designation             prog
        basic pay               567.89
        deds                    45.76
```

In this program, we have created only one record and in the next program, we will see how multiple objects can be input and output.

# CHAPTER - 13

# CLASSES & OBJECT

## WHAT IS A CLASS ?

A class is an organisation of data and function which operate on them. Data structures are called data members and the function are called member functions. The combination of data members and member function constitute a data object or simply an object. In non technical language, we can say that a class is a collection of similar objects containing a set of shared characteristics. In a way, a class and its objects have the relationship of a data " and variables. A class is simply a template for holding objects. A class is abstract but objects are real. Simply by defining a class we don't create an object just like the more declaration of a data type does not create variables. one or more classes grouped together constitute a program.

A class is declared in the beginning of a program. The general form of class declaration is given below.

```
CLASS name-of-the-class
{accessibility type;
dat-type variable-1;
data-type variable-2;
————————————
data-type variable-;
accessibility - type;
data-type function-name( )
{ statements;
return;
        }
        ——
        };
```

A class can contain any number of data members and any number of member functions. A class can also have only data members. The program given below is intended to create a class with three data members and two objects to demonstrate the syntax and general features of classes.

```
#include <iostream.h>
class part
{private;
int modelnum, partnum;
float cost;
public;
void setpart (int mn, int pn, float c) // first member function
{modelnum = mn; partnum = pn; cost =c);
void showpart ( ) //second member function
{cout <<endl <<“ model;”<<modelnum <<endl<< “num;” <<partnum
<<endl <<“cost” << S” <<cost,}
        };

void main ( )
{part p1, p2 \\We define two objects p1 & p2
p1 setpart (644, 73, 217, 55); //values are passed to the two objects
p2 setpart (567, 89, 789,55);
p1 showpart ( )     //Each object displays its values
p2 showpart ( )
        }
```

The output looks as

```
model: 644
num: 73
cost S 217. 550003
model:567
num : 89
cost S 789.549988
```

A key feature of oop is data hiding. This means that the data is hidden inside a class, so that it is not accessed mistakenly by any function outside the class. This is done by declaring the data as private. All data and functions defined in a class are private by default. But for the sake of clarity, we explicitly declare the data items as private.

All data defined in a struct are public by default, but can be declared as private by explicit declaration. In the above example the two member function are declared as public. By default these are private. The explicit declaration public means that these functions can be accessed outside the class. Member function defined inside a class this way are created as inline function by default.

to use a member function from the main, we make use of the dot operator also called the class member access operator The first call p1 setpart (644,73,217.55) sets the variables in the object p1 by parameter passing. The second call p2. setpart (567, 89,789,5) sets the variables in the object p2 by parameter passing. It must be noted that we cannot initialise the data in the object directly from the main but only through member functions.

One cannot initialise modelnum, partum and cost in the data declaration statement itself by writing

```
int modelnum = 644, partnum = 73; float cost = 217.55
```

Since only member function are authorised to manipulate the private data declared in the class. The data members can not 'take law into their own hands'. But one can initialise the data items within the member function. For example, one can write a member function as

```
void init ( )
{modelnum = 644; partnum = 73; cost = 217.55;}
```

Let us now consider a program in which values of data members in an object can supplied interactively through the keyboard.

```
#include <lostrearn.h>
class distance
{private:
int feet;
float inches;
public:
.void setdist (int ft, float in)
{feet=ft, inches =in;}
void getdist ( )
{cout<<"enter feet "; cin>>feet;
cout <<"enter inches;" cin>> inches;}
void showdist ( )
{cout<<feet>>"-" >>inches>>"/-"};

void main ( )
{ distance d1, d2
d1.setdist (11,6, 25);
d2.getdist ( );
cout <<endl>."dist1" d1.showdist ( );
cout << endl>> dist2." d2.showdist ( );
```

The output looks as

```
enter feet 12
enter inches: 6.25
dist1: 11- 6.25"
dist2: 12 - 6.25"
```

When the first member function setdist( ) is invoked for object d1, it sets the values of the data members through the use of arguments 11 and 6.25. When the second member function' getdist ( ) is invoked for object d2, it sets the values of the members by asking the user for keyboard input.

## CONSTRUCTORS

In .the earlier programs we saw two different ways of giving values to the data Items in an object. Sometimes, it is convenient if an object can initialise itself when it is first created, without the need to make a separate call to a member function. Automatic initialisation is carried out using a special member function called a constructor. A constructor is a member function which is executed automatically whenever an object is created.

As an example, we will created a class of objects called the counter which can be used for counting purposes. Each time an event takes place, the counter is incremented. The counter can also be accessed to find the current count. The program below illustrates this idea.

```
//program for illustrating the function of the constructor
#include <iostream.h>
class counter
{
        private;
        unsigned int count,
        public;
        counter()
        {count = 0;}
        ,void increment()
        {count ++;}
        int getcount()
        {return count ;}

        void main()
        {
                counter c1, c2, c3. c4
                count<<endl>>" counter c1="<<cl.getcount ( );
                count<<endl>>" counter c2="<<c2.getcount ( );
                count<<endl>>" counter c3="<<c3.getcount ( );
                count<<endl>>" counter c4="<<c4.getcount();
                c1.increment();
                c2.increment();
                c2.increment();
                c3.increment();
                c3.increment();
                c3.increment();
                c4. Increment();
                44. increment();
                c4. Increment();
                cout<<endl<<"counter cl="<<c1.getcount ( );
                ciytMMebdkMM: clybter c2="<<c2.getcount ( );
                cout<<endl<<"counter c2="<<c3.getcount ( );
                cout <<endl<<"counter c4="<<c4.getcount();
```

The output looks as

```
counter c1 =0
counter c2=0
counter c3—0
counter c4=0
counter c1 =0
counter c2=3
counter c3=3
counter c4=4
```

The compiler will identify the constructor by looking at the name of the member function. the constructor function name is the same as the class name. There is no return type with constructor. When the objects c1, c2, c3, and c4 of counter class are created, they are automatically initialised to zero. Their initial values are displayed. Then they are incremented differently using the increment function. The new values are also output by the program.

**DESTRUCTOR**

We have seen that constructor is a member function called automatically when an object is first created. You might guess that there is also a function called automatically when an object is destroyed. This function is the destructor. A destructor has the same name as the constructor (which is the name of the class), but preceded by a tilde (-).

The program segment given below illustrates this idea.

```
class foo
{
        private;
        int data;
        public;
        foo ( )
          {data=();}
          ~foo()
          { }
};
```

Like constructor, destructors do not have a return value. They do not have any arguments, because it is assumed that there is only one way to destroy an object. There are also no statements in the destructor function body. The most common use of destructors is to de-allocate memory that was allocated for the object by the constructor.

**OVERLOADED CONSTRUCTORS**

The program shown below illustrates constructor overloading.

```
 #include<iostream.h>
class distance
```

```
{
        private:
        int feet;
        float inches;
        public:
        distance ( )    //empty constructor
        distance (Int ft, float in )      // two argument constructor
        {feet =ft; inches =in;}
        void getdist ( ) // get distance from user
        {count <<"enter inches:", cin >> feet,
        cout <<"enter inches :"; cin >>inches;}
        void showidst( )
        {cout <<feet <<"-"<<inches<<"\"}
        void adddist (distance d2, distance d3)
        {inches =d2.inches + d3.inches;
        feet = 0;
        if (Inches>=0) {inches =12.0; feet++;}
        feet += d2.feet+d3.feet
        void main( )
        { distance dist1, dist3);
        distance dist 2 (11, 6, 25);
        dist1.getdist ( );
        dist3. addist (dist1, dist2);
        cout<<endl<<"dist1=:" dist1.showdist( );
        cout <<endl <<"dist2+ "dist2.showdist( );
        cout<<endl<<"dist2+:" dist3.showdist( );
```

The output looks as
enter feet 12
enter inches: 7.25
dist 1=12, 7.25"
dist2= 11, 6.25"
dist3=24.1511

In the above program we have provided an empty constructor. It doesn't do anything. Then why do we need it, since our earlier class program didn't have any constructor at all. Our first program worked because there was an implicit constructor (built into the program automatically by the compiler) that created the object, even though we didn't specify it in the implicit constructor. Since there are two constructor with the same name, distance( ), we say that the constructor is overloaded. Only one constructor is executed when an object is created. Each one depends on how many arguments are used in the object definition.

While distance dist1// calls the first constructor distance (1 1,6.0) calls the second constructor

## MEMBER FUNCTIONS DEFINED OUTSIDE THE CLASS

The program given below shows the definition of a function outside the class. Objects are used as arguments to functions to functions.

```
#include <iostream.h>
class distance
{
        private:
        int feet;
        float inches;
        public:
        void setdist (int ft, float in)
        {feet = ft; inches = in;}
        void getdist( )
        { cout <<“enter inches:”; cin>>inches;}
        void showdist ()
        {cout<<feet<<“=” inches<.’/”;}
        void addist (distance d2, distance d3)     //called declaration

        void distance:: adddist (distance d2, distance d3)
        {inches = d2.inches + d3.inches; feet =0;
        if (inches>=0 {inches = 12.0; feet ++;}
        feet+= d2. feet +d3 feet;
}
        void main ( )
        {distance dist1, dist2, dist3
        dist2.setdist (11,6.25);
        dist1.getdist ( );
        dist3.adddist (dist1, dist2);
        cout<<endl<<“dist1=”, dist1. Showdist ( );
        cout<<endl<<“dist2=”, dist2. showdist ( );
        cout<<endi<<“dist3=”, dist3. showdist ( );
```

The output appears as

```
enter feet 12
inter inches 7.5
dist1 = 12'-7.25"
dist2 = 1 1'-6.25 “
dist3 = 24'-1.5"
```

dist1 and dist3 are created.
dist2 is created with two arguments.
a value is obtained for dist 1.
add dist 1 and dist2 to obtain dist 3.

The function adddist (distance d2. distance d3) is declared within the class. It is fully defined outside the class. To tall the computer that the function belongs to the distance class, we make use of what is called the scope resolution operator (::) and write the function header as

.void distance adddist (distance d2, distance d3)

The two distance dist1 and dist2 to be added are supplied as arguments. The syntax for arguments that are objects is the same as that for arguments that are simple data type like int the object name is supplied in the argument. Since adddist ( ) is a member function of class distance, it can access private data in any distance objects supplied as arguments. Thus it has access to the data members of dist1 and dist2. When a member function is called- with or without arguments- it is given direct access to only one object; the object which called it. In our example, we called:

dist3 adddist (dist1, dist2);

Therefore when variables inches and feet are referenced, they really refer to dist 3. inches and dist3.feet. The return type of adddist ( ) is void. The result is stored automatically in the dist3 object.

**RETURNING OBJECTS FROM FUNCTIONS**

Let us look at an example of a function that returns an object.

```
#include <iostream.h>
class distance
{ private;
int feet;
float inches;
public:
distance ( )
{feet = inches = 0;}
distance (int ft, float in)
{feet = ft; inches = in;}
void showdist ( )
{cout<<feet<<"-"<<inches<<'/";}
distance adddistance (distance d2);
};

distance distance:: adddistance (distance d2)
{distance temp;
temp.inches = inches+ d2 inches;
if (temp.inches>=12.0
{temp.inches=12.0;
temp. feet = 1;}
temp.feet+=feet + d2.feet;
return temp;
```

```
void main ( )
distance dist3. Dist1 (6,4,5), dist2 (7,4,8);
dist3= dist1.adddistance (dist2);
cout<<endl<<“distance(dist2);
cout<<endl<<“distance1=”dist1.showdist ( );
cout<<endl<<“distance2=”dist2.showdist ( );
cout<<endl<<“distance3=”dist3.showdist ( );
```

The output looks as

```
distance1 = 6'-4.5"
distance2 = 7'-4.8"
distance3 = 13'-9.3"
```

Here only one distance object is passed.

In main( ), three distance objects are defined as dist1, dist 2, and dist3. values are established for dist1 and dist 2. These are added together using adddistance and the returned object is then assigned to dist 3 in the following line:

```
dist 3 = dist1.adddistance (dist2);
```

# CHAPTER - 14

# POINTERS AND ARRAYS

## INTRODUCTION

A pointer is a variable that holds an address of a memory location. An address is an integer, but pointers are of various target types such as char, integer, float, double. This distinction is required as each data type requires different number of bytes of memory, so that when a painter is increments it is done in block of bytes of memory of the particular target type. Pointers enable efficient way of accessing array elements, passing arguments, passing arrays and strings to function, obtaining memory from system, creating data structures such as linked lists and binary trees.

## INTRODUCTION TO POINTER VARIABLES

The content is what is stored within a memory location. Its address is a label that identifies its location. Using this simple concept a program is presented.

```
//content.cpp
// contents and address of a memory
#include <iosteam.h>
void main ( )
        {
int loc1 = 37; int loc2 = 73; int loc3 = 97;
cout <<"\nloc1 contains:"<<loc1;
cout<,"\nloc2 contains: "<<loc2;
cout<<"\nloc3 contains: "<<loc3;
cout <<"\nAddr of loc1: "<<&loc1;
cout <<"\nAddr of loc2: "<<&loc2;
cout <<"\nAddr of loc3: "<<& loc3;
```

The output is

```
loc1 contains : 37
loc2 contains: 73
loc3 contains: 96
Addr of loc1 : 0x4do 2fff4
```

Addr of'loc2: 0x4do 2fff2
Addr of loc3: 0x4do 2fff0

**ANALYSIS**

1. Three integer variables loc1, loc2, loc3 are defined and initialized,
2. The cout objects cause the contents and addresses of these variables to be output
3. "&" is called the address operator. When this operator is prefixed to a variable, the address of the variable is implied.

&loc1 <-address of loc 1
&loc2 <- address of loc2
&loc3 <- address of loc3

4. Simply using the variable name implies the content of the memory location.

loc1 <- content of loc1
loc2 <- content of loc2
loc3 <- content of loc3

The following program uses pointer variables.

```
pointer.cpp
creating and using pointer variables
include <iostream.h>
    void main ( )
    {
    int *ptr; int loc1 =37; int loc2 = 73;
    ptr = &loc1;
    cout <<"\n loc1 contains: "<<loc1<<" in addr: "<<ptr ;
    ptr = & loc2;
    cout <<"\nloc2 contains : "<<loc2<<" in addr: "<<ptr ;
```

The output is

loc1 contains: 37 in addr: ox4cfffff4
loc2 contains: 73 in addr: ox4cfffff2

**ANALYSIS**

1. A pointer variable is defined by int *ptr;

Now"ptr" is said to be a pointer to target type "int". It can hold any hexadecimal address. The above definition may also be written with asterisk associated with int, such as:

```
    int * ptr;
```

The asterisk'*' implies "pointer to".

2. Two int " variables are defined and initialized.
3. Then the address of "loc1 " is assigned to "ptr" by the following ptr = &loc1;
   by this the address of "loc1' is assigned to "ptr" by the following "ptr"
4. The object cout then outputs the contents of "loc 1 " and "ptr" and the object cout causes the contents of loc2 to be output.

&loc1 <- address of loc1
&loc2 <- address of loc2
&loc3 <- address of loc3

5. Simply using the variable name implies the content of the memory location.

loc 1 <- content of loc 1
loc2 <- content of loc2
loc3 <- content of loc3

The following program uses pointer variables.

```
pointer.cpp
creating and using pointer variables
include <iostream.h>
    void main ( )

    int *ptr; int loc1 =37;  int loc2 = 73;
    ptr = &loc1;
    cout <<"\nloc1 contains: "<<loc1<<" in addr: << ptr;

    ptr = & loc2;

cout <<"\nloc2 contains : <<loc2<<" in addr: "<<ptr;
```

**The output is**

loc1 contains: 37 in addr: ox4cfffff4
loc2 contains: 73 in addr: ox4cfffff2

**ANALYSIS**

1 . A pointer variable is defined by int *ptr;

Now "ptr" is said to be a pointer to target t)W "int". It can hold any hexadecimal address. The above definition may also be written with asterisk associated with int, such as:

int * ptr;
The asterisk'*' implies "pointer to".

2. Two int " variables are defined and initialized.
3. Then the address of "loc 1 " is assigned to"ptr" by the following ptr = &loc 1;
   by this the address of "locl' is assigned to "ptr" by the following " ptrit
4. The obj ect cout then outputs the contents of "loc 1 " and "ptr" and the object cout causes the contents of loc2 to be output.

## POINTERS TO ARRAYS

The versatility of pointers is established due to its ability to handle arrays. The following program illustrates an elementary idea of pointer to an array

```
//ptrarray.cpp
//pointer to an array
#include <iosteream.h>
void main( )
        {
float array[] = { 12.34, 23.45, 34.56, 45.67, 56.78};
float *ptr;
ptr = array;
for (int k =0; k<5; k++)
cout <,"\narray ["<<k<<"] ="<<* (ptr ++);
```

The output is

```
array [0] = 12.34
array [1] = 23.450001
array [2] = 34.560001
array [3] = 45.669998
array [4] = 56.779999
```

## ANALYSIS

1. A float type array called array is defined and initialized with 5 floating point constants.
2. A pointer called "ptr" of target " float is set up.
3. It is made to point ot the starting address of array by the following assignment.
   ptr = array;
   "ptr" now points to memory location where array [0] is located-
4. Within the for loop, the pointer "ptr" is used to access the location whose address it holds to retrieve the constants stored in it.
5. Then the pointer increments to point ot the next array element which is 4 bytes away, as it is of " float.

## PASSING BY POINTERS

Arguments can be passed to a function by value, by reference or by pointer. Passing by value is quite easy. passing by reference implies passing the address of a variable or an element of an array. This is the same as passing of pointer. The following program illustrates the idea.

```
\\reference . cpp
\\ passing by reference
#include<iostream.h>
void main( )
        {
void convert (float &); \\called a prototype float amt = 123.45;
float dollars = amt;
convert (amt);
cout<<"/n name "<<dollars <<"converted to Rs. is    <<amt;
        }
void convert (float &a)
{ a*=31.65;}
```

**The output is**

amt 123.449997 converted to Rs. is 3907.192383

**ANALYSIS**

1. A prototype is declared for the function called "convert" with an address argument, which is pointing to a location having float type data.
   void convert (float &);
2. The function is invoked with argument "amt", which is actually an address.
3. The function "convert" receives the address passed by 'amt' in "a" which is also an address variable
4. The contents of this address is multiplied by 31.65 and the result stored in "a". Remember that "a" and "amt" point to the same address, and therefore there is no need to return the result to the calling program. Hence convert ( ) function is of type void.

**SORTING USING POINTERS**

Here sorting an array containing floating point values is done by pointers. The technique used is called bubble sort. In this technique the first element value is compared with all other elements, until a value less that this is encountered. Then a swap is made. Next the second element is compared ivith the others and if need be a swap is made. Thus the process is continued until the array values are arranged in an ascending order. The program is given.

```
// ptrsort.cpp
// sorting float type values by pointers
# include<iostream.h>
void main ( )
        {
void bubble (float*, int) // prototype
const int L = 12;
float val[L] = { 9.1, 6,3, 5.5, 8.7, 1.2, 4.3, 6.5, 7.8, 11.2, 3.2, 2.4, 1. 1};
 cout <<"\n The original array is :\n";
 for (int k= 0; k<L; k++)
cout <<val [k] <<" ";
```

```
bubble (val, 1);
cout <<"\nThe sorted array is :\n";
for (int j=O; j<L; j++)
cout <<val [k] <<" ";
        }
void bubble (float*, float*)
int m, n;
for (m=0; m<n - 1; m++)
for (n =m + 1; m<n; n++)
sort (ptr + m, ptr + n);
        }
void sort (float * n1, float n2)
        {
if(*n1> *n2)
        {
float temp = *n1;
*nl = *n2;
*n2 = temp;
        }
}
```

**The output is**

The original array is:

9.1 6.3 5.4 8.7 1.2 4.3 6.5 7.8 11.2 3.2 2.4 1.1 The sorted array is :

1. 1 1.2 2.4 3.2 4.3 5.4 6.3 6.5 7.8 8.7 9.1 11.2

**ANALYSIS**

1. A prototype is declared for bubble with a float type pointer an an integer.
2. The array val [] of type float is defined and initialized with 12 elements.
3. The elements of the original array are displayed.
   The function bubble is invoked with the first address of the array val which corresponds to val [0], and the number of elements in the array.
5. The sorted array is then displayed.
6. The function bubble ( ) is furnished two arguments:
7. Within bubble ( ) another prototype is declared void sort (float*, float*);
   the function sort has two float " pointers, which can receive the address of 2 elements of array val [ ].
8 The function bubble invokes the sort function, furnishing two pointer values, to do bubble sort of the array val
9 The sort ( ) function does the sorting of the float values by the technique called bubble sort, which was described at the beginning.

## POINTERS IN STRING HANDLING

A string is an array of char type. While array notation itself is sufficient to point to the elements of a string, in the next program pointers are used to point to the various elements of the string.

```
//str1. cpp
#include <iostrearn.h>
void main
        {
void show (char*);
char s1[ ] = " Love thy neighbour as thyself.",
char* s2 = " D unto other as thou wouldst done unto yow";// prototype
show (s1); show (s2);
s2++; s2++; s2++; s2++;
show (s2)
        }
void show (char* p)
        {
cout <<"/n;
while (*p)
cout << *P++;
        }
```

**The output is**

Love thy neighbour as thyself
Do unto other as thou wouldst done unto you. unto others as thou wouldst done unto you.

**ANALYSIS**

1. The prototype declares availability of a function show (char*).
2. A char type string array s1[ ] is defined and initialized. Note the square brackets in the string declaration of s I and absence of the same in the pointer declaration for s2.
3. Next a char type pointer is defined and initialized. Note the square brackets in the string declaration of s 1 and absence of the same in the pointer declaration for s2.
4. When char pointer s2 is set up, it points to the first char "D" in the array of characters, or to s2 [0] element.
5. The show ( ) function in invoked to display the string in s I and string pointed to by pointer s2,
6. See that either in the case of the array notation or pointer notation, the first address, namely s1[0], or s2 [0] is passed on to the receiving pointer variable in show Thereafter incrementing the pointer "p" in show the entire string is caused to display.
7. Then s2 is incremented 4 times to point to s2 [31 and then display. that is why the last display starts with "unto....." which is the 4th position of the s2 array.

The following program shows how to copy a string into another string array.

```
//strcpy.cpp
//string copying using strings
# include <iostream.h>
void main( )
{
void dup (char*, char*);
char* s1= Leo Tolstoy said only knaves can succeed in life.";
char s2 [53]   //declaration of array
               // s2 of 53 chars
dup (s2, s1);
cout <,"\n"<< s2;
}
void dup (char* d, char* s)
{
while (*s)
*d++ = *s++;
*d = '\0';
}
```

The output
Leo Tolstoy said only knaves can succeed in life.

**ANALYSIS**

1. The prototype declares the presence of a function dup( ) with 2 char type pointers as arguments.
2. Then a char type pointer s2 is defined and initialized.
3. Next a char type array called s2, of 53 characters is declared.
4. By using dup () function the string pointed to by s I is copied into the array pointed to by s2. The copying is done character by character, beginning from s1[0] upto the end of the string.
5. After the copying is over, the null character' is appended to terminate the string in s2.

**DYNAMIC MEMORY ALLOCATION**

A string array has memory space allocated only at the time of execution. No memory allocation is done at the time of declaration. This is called dynamic allocation memory.

How exactly dynamic allocation of memory works is illustrated by the following program.

```
//newoper. cpp
//use of operator new for memory allocation
#include <iosteream.h>
#include <string.h>
void main( )
        {
char* sent "Individuals are occasionally guided by reason, crowds never."; char* pointer;
pointer = new char [strlen (sent) + 1]
```

```
strcpy (pointer, sent);
cout<< "\n The original string is :\n"<<sent;
cout<<"\nThe copy string is : \n"<<pointer; delete pointer;
```

**The output is**

```
The original string is:
Individuals are occasionally guided by reason, crowds never.
The copy string is:
Individuals are occasionally guided by reason, crowds never.
```

**ANALYSIS**

1. A pointer to a char array called "sent" is defined. It points to the first address of the string.
2. Another char type pointer called "pointer" is declared. It is used to set apart memory required to contain the string in "sent' plus a null character. The operator "new" does the task of requisitioning memory from the operating system.
3. Now we copy character by character using the two pointers:
   "pointer" as destination
   " sent" as the source
4. Both the strings pointed to by pointers "sent" and "pointer" are displayed.
5. The "delete" operator returns memory set apart for "pointer" and returns it to the operating system.

**POINTERS TO OBJECTS**

When memory needs to be allocated dynamically with each creation of an object of a class, the "new" operator comes handy.

The following program illustrates its use.

```
//dynam.cpp
//pointers to objects
#Include<iostream.h>
class currency
{          private:
float amt;
public:
void fetch( )
{cout <<"\nEnter amount:"; cin>>amt;}
void display ()
{cout<<"Amount is: "<<amt;}
}
void main( )
{
currency c;
c. fetch ( )
```

```
c. display( );
currency* ptr;
ptr = new currency;
ptr - > fetch ( );       // access by arrow operator
ptr -> display
```

**The output is :**

```
Enter amount: 1234.56
Amount is; 1234.56.0059
Enter amount : 2345.67
Amount is: 2345.669922
```

**ANALYSIS:**

**1.** The class structure has only routine declaration which does not need elaboration.
2. Examine the main ( ) program. An object "c" of "currency" is defined. It access the member functions of the class using the dot or the period operator
3. Next a pointer to an object of "currency" called "ptr" is set up.
4. The "ptr" is returned as an address pointer when we allocated memory to the class "currency" due to the following definition "ptr = new currency";
5. This pointer "ptr" which points to an object is used to access member functions using the arrow operator.
6. Note that the arrow operator (->) is used with a pointer to an object, just like a dot operator is used with an object of a class.)

**ARRAY OF POINTERS**

Creating an array of pointers to objects enables easy access to objects. This is an alternate to setting up an array of objects. The following program shows the use of this idea.

```
//ptrobj. cpp
array of pointers to objects
#include <iostream. h>
class country
{
private:
char coun [20]
public:
void fetch( ) //20 character string
        {
        cout<<"\n Name country:"; cin >>coun;
        }
        void show ( )
        {cout <<"country is: "<<coun;}
```

```
void main( )
{
        country* counptr [20];      //20 pointers to object
        int pt = 0;
        char ne;
        do
        {
        counptr [pt]       new country;     //new allocation
        counptr [pt].fetch ( )                  // arrow operator
        counptr [pt].show ( )
        pt ++;
        cout <<"\n     Continue? (y/n):";    cin >>ne;
}
while (ne == y);
```

**The output is**

```
Name country: India
Country is: India
Continue ( y/n) : n
```