PROGRAMMING IN 'C'

INTRODUCTION:

'C' was an offspring of the 'Basic Combined Programming Language' (BCPL) called B, developed in the 1960's at Cambridge University. B Language was modified by Dennis Ritchie and was implemented at Bell Laboratories in 1972. The new language was named C.

MODULARITY:

In programming, the term structure has two interrelated meanings. Program whose structure consists of interrelated segments arranged in a logical and easily understandable order to form an integrated and complete unit, are called *modular programs*. The smaller segments used to construct a modular program are called *modules*. Since each module is designed to perform a specific function, the modules themselves are called *functions* in C.

PORTABILITY:

C matches the capabilities of many computers, it is independent of any particular machine architecture**.**

DATA TYPES AND FORMAT SPECIFIERS:

Data type	Meaning	Size	Format Specifier
char	holds only one character	1 Byte	%c
int	holds an integer	2 Byte	%d
float	holds a single precision floating point number	4 Byte	%f
double	holds a double precision float point number	8 Byte	%lf

Note: Precision refers to the no. of significant digits after the decimal point.

CONSTANTS AND VARIABLES:

A constant does not change its value during the entire execution of the program.

Ex: 3.14, 510, 3

A variable is an entity that has a value and is known to the program by a name. It's value may change during the execution of the program.

Ex: int x; => x is a variable of integer data type.

OPERATORS:

An operator, in general, is a symbol that operates on a certain data type. For example, the operator '+' performs addition operation.

Types of operator:

i) Arithmetic operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division (It gives remainder after division)

ii) Relational operators:

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	No equal to

iii) Logical operators:

&&	Logical AND
||	Logical OR
!	Logical NOT

iv) Assignment operators:

This operator evaluates the expression on the right, and assigns the resulting value to the variable on the left.

Ex: =, -=, +=, /=, *=.

v) Increment and decrement operators:

This operators are extensively used in *for* and *while* loops.

syn: a) ++<variable_name> or <variable_name>++
b) --<variable_name> or <variable_name>--

vi) Conditional operators(Ternary operators):

It consists of two symbols, the question mark(?) and the colon(:).

syn: var = var1>var2 ? var1 : var2 ;

Ex: x = I > j ? i : j ;

vii) Comma operator:

A set of expressions, seperated by commas is a valid construct in the C language. It is used to combine two related expressions into one, making programs more compact.

Ex:
```
int i, j;
for(sum=0, i=1;  i<10;  i++, sum+=i)
```

viii) Other operators:

The operator *sizeof* gives the size of the data type or the variables in terms of bytes occupied in memory. Another operator is the member selection operator (. and ->) which is used with structures and unions. Pointer operators * and & are explained in detailed in later chapter.

ESCAPE SEQUENCE OR ESCAPE CHARACTORS:

\a	Beep (Alert)
\b	Back space
\n	New line
\t	Horizontal tab
\v	vertical tab
\\	Back slash
\'	Single quote
\"	Double quote

SHORT KEY to compile, execute and display the output:

alt + F9 => Compilation
cntrl + F9 => Execution
alt + F5 => Display the Output

BASIC INPUT - OUTPUT

main():

Consider a very simple program, given below.

```
/*      EX-1: SAMPLE PROGRAM TO PRINT A TEXT  */
#include<stdio.h>
main()
{
  printf("NATIONAL INSTITUTE");
}
```

OUPTPUT : NATIONAL INSTITUTE

Note:

1. In C everything is written in lowercase letters. However uppercase letters used for symbolic names representing constants. ***Ex***: # define MAX 80
2. The line beginning with /* and ending with ***/*** are known as *comment line*. It is used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between **/*** and */ is ignored by the compiler.
3. #include<stdio.h>is a special file, which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler.
4. The main() is a special function used by the C system to tell the computer where the program starts. Every program must have *exactly one* ***main*** function. The empty pair of parentheses immediately following main() indicates that the function main has no *arguments*(or parameters). The opening brace '{' in the second line marks the beginning of the function main() and the close brace '}' in the last line indicates the end of the function. Sometimes it may be an end of the program. All the statements between these two braces form the body of the function.

printf():

The statement printf() is used to print the text that contains within two double quotes.

syn: 1. printf("text");
2. printf("access_specifier",variable);

```
/*      EX-2: AN EXAMPLE ILLUSTRATING INTEGER AND FLOATING POINT OUTPUT     */
#include<stdio.h>
main()
{
  int ivar1,ivar2,ivar3;                /* Declaration statement  */
  float fvar;
  clrscr();                             /* Clear the screen     */
  ivar1=4;                              /* Assignment statement */
  ivar2=6;
  fvar=18.235235;
  ivar3=ivar1+ivar2;
  printf("\n Sum of %d and %d is %d", ivar1,ivar2,ivar3);
  printf("\n\t Assigned float value = %f", fvar);
  getch();                      /* Wait statement    */
}
```

Output : Sum of 4 and 6 is 10
Assigned float value = 18.235235

***Note*:** In above program the first printf statement having three format specifier and three parameters. The order of format specifier should be same to the order of parameter in a printf statement.

scanf():

The scanf function reads the values of variables from the standard input device(keyboard) and stores them in variables.
syn: scanf("access_specifier", &variable);

```
/*      EX-3: ILLUSTRATING THE SCANF() WITH NUMBERS   */
#include<stdio.h>
main()
{
  float years,secs;
  clrscr();
  printf("\n\t Enter your age: ");
  scanf("%f",&years);
  secs=years*365*24*60*60;
  printf("\n\n\t CONGRATULATION: \n\t\t You have lived for %f seconds",secs);
  getch();
}
```

Input : Enter your age: 21
Output : CONGRATULATION:
You have lived for 662256000.0 seconds

```
/*       EX-4: ILLUSTRATING SCANF() WITH CHARACTER  */
main()
{
 char x;
 clrscr();
 printf("\n Enter a character: ");
 scanf("%c",&x);
 printf("\n Given character is %c",x);
 getch();
}
```

Input : Enter a character: B
Output : Given character is B

```
/*      EX-5: ILLUSTRATING SCANF() WITH STRINGS          */
#include<stdio.h>
main()
{
  char str1[20],str2[20];
  clrscr();
```

```
  printf("\n\tEnter any two strings:\n");
  scanf("%s",str1);
  scanf("%s",&str2);
  printf("\tGiven two Strings are:  %s and %s", str1,str2);
  getch();
}
```

INPUT : Enter any two strings:
National
Institute
OUTPUT : Given two Strings are: National and Institute

***NOTE*:** 1) In the declaration statement char str1[20], str2[20], str1 and str2 can hold the width of 20 characters, the string wouldn't accept the blank space.
2) %s is a format specifier for string [Array of character].
3) The two statements

```
scanf("%s", str1);
scanf("%s", str2);
```

can be replaced by a single statement

```
scanf("%s %s", str1, str2) with exactly same result.
```

getchar() and putchar():

As the indicates getchar reads a character from the standard input device, while putchar writes a character to the standard output device.

```
/*      EX-6: EXAMPLE INVOLVING BOTH GETCHAR( ) AND PUTCHAR( )      */
#include<stdio.h>
main()
{
  char c;
  clrscr();
  printf("\nInput one Character: ");
  c=getchar();
  printf("\nThe character you have typed : ");
  putchar(c);
  getch();
}
```

INPUT : Input one Character: s
OUTPUT : The character you have typed : s

gets() and puts():

The function gets receives the string from the standard input device, while puts outputs the string to the standard output devices.

```
/*      EX-7: EXAMPLE INVOLVING BOTH GETS( ) AND PUTS( )      */
#include<stdio.h>
main()
{
  char s[80];
  clrscr();
  printf("\n Input a String (maximum of 80 character): ");
  gets(s);
  printf("\n\t The String you have typed: ");
  puts(s);
 getch();
}
```

INPUT : Input a String (maximum of 80 character): Sri Baskaran
OUTPUT : The String you have typed: Sri Baskaran

***Note*:** The function gets accepts more than one word including blank space.

Assignment Suppression Character:

An assignment suppression character tells scanf() that the input should only be scanned and not assigned to any argument. The assignment suppression character is asterisk(*), and it must be specified immediately after the % sign in the format specifier. So no need to declare a variable for assignment suppression character.

```
/*        EX-8: ILLUSTRATING SCANF() USING ASSIGNMENT SUPPRESSION       */
#include<stdio.h>
main()
{
  int date,month,year;
  clrscr();
  printf("\n Input the date (dd.mm.yyyy) with any seperator(. / - *):\n\t\t\t");
  scanf("%d %*[/-.*] %d %*[/-.*] %d", &date, &month, &year);
  printf("Date : %d\n",date);
  printf("Month: %d\n",month);
  printf("Year : %d\n",year);
  getch();
}
```

***Note*:** Here, there are 5 format specifiers, but only 3 arguments following the format string in a scanf() statement.

Input : Input the date (dd.mm.yyyy) with any seperator(. / - *):
01.01.2002

output : Date : 1
Month : 1
Year : 2002

CONTROL STRUCTURES

INTRODUCTION:

'C' program is a set of statement, which normally executed sequentially in the order in which they appears. In practice we have a number of situations where we may have to change the order of execution of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

A) DECISION MAKING STATEMENTS

(a) The if statement:

The if statement is used to specify conditional execution of program statement, or a group of statements enclosed in braces.

***syn*:**
```
if(condition)
{
  True-block statements;
}
```

In a following sample program, if the user enters a number divisible by 3, a message has to be printed. If it is not divisible by 3, then the message need not be printed.

```
/*      EX-9: ILLUSTRATION OF IF STATEMENT          */
#include<stdio.h>
main()
{
  int i;
  clrscr();
  printf("\n Enter a number: ");
  scanf("%d",&i);
  if(i%3==0)
  printf("\n The number you have entered is divisible by 3");
  getch();
}
```

(b) The if-else statement:

When two groups of statements are given and it is desired that one of them be executed if the condition is true and the other group be executes if that condition is not true. In such a situation, the if-else statement can be used.

***syn*:**
```
if(condition)
{
  True-block statements;
}
else
{
```

```
            False-block statements;
          }

/*        EX-10: ILLUSTRATING IF-ELSE STATEMENT  */
#include<stdio.h>
main()
{
  int a,b;
  printf("\n Enter the value of a: ");
  scanf("%d",&a);
  printf("\n Enter the value of b: ");
  scanf("%d",&b);
  if(a>b)
    printf("\n\t %d is greater than %d", a,b);
  else
    printf("\n\t %d is greater than %d", b,a);
  getch();
}
```

C) Ternary operator:

This can be used to replace the statements of the if-else. Ternary operator connects three operands. The first operands always evaluated first. It is usually a conditional expression that uses the logical operators. The next two operands are any other valid expressions, which can be single constants, variables or general expressions. The complete conditional expression consists of all three operands connected by the conditional operator symbols ? and :.

```
/*        EX-11: ILLUSTRATING TERNARY OPERATOR            */
#include<stdio.h>
main()
{
  int i,j,larger;
  clrscr();
  printf("\nInput two integers: \n");
  scanf("%d %d",&i,&j);
  larger = i > j ? i : j ;
  printf("\n The larger of two numbers : %d", larger);
  getch();
}
```

***Note*:** If the value of i is greater than j then the value of i will store into the variable larger, otherwise the value of j will store into the variable large.

d) Nested if-else statement:

***SYN*:**

```
if(condition-1)
{
   if(condition-2)
    {
      statement(s)-1;
    }
  else
  {
    statement(s)-2;
  }
else
{
 statement(s)-3;
}
```

```
/*        EX-12: ILLUSTRATING NESTED IF-ELSE STATEMENT  */
#include<stdio.h>
main()
{
  char g,name[30];
  int a;
  clrscr();
  printf("\nEnter your name: ");
  gets(name);
  printf("\nEnter gender code: ");
  fflush(stdin);
  scanf("%c",&g);
```

```
  if(g=='f' || g=='F' || g=='m' || g=='M')
  {
    printf("\nEnter age: ");
    scanf("%d",&a);
    if(g=='m' || g=='M')
      if(a>=21)
          printf("\n\tHellow Mr. %s",name);
        else
          printf("\n\t Hellow Master. %s",name);
    if(g=='f' || g=='F')
       if(a>=18)
           printf("\nHello Miss. %s",name);
         else
           printf("\nHellow Kumari. %s",name);
  }
  else
  printf("\n\n\n\t\tInvalid gender code, Enter m/f only");
  getch();
}
```

e) Multiple if-else statement (or) else-if ladder:

When multipath decisions are involved, a chain of ***if*** statement associated with each ***else*** is an if. It's construct is as follows.

***syn*:**
```
if(condition-1)
     statement(S)-1;
   else if(condition-2)
        statement(s)-2;
     else if(condition-3)
          else
              statement(s)-3);
```

```
/*        EX-13: ILLUSTRATING ELSE-IF LADDER       */
#include<stdio.h>
main()
{
  int mark;
  clrscr();
  printf("\n Enter the student's mark: ");
  scanf("%d",&mark);
  if(mark>=80)
     printf("\n Grade = \"Distinction\"");
   else if(mark>=60)
       printf("\n Grade = \"First Class\"");
     else if(mark>=50)
         printf("\n Grade = \"Second Class\"");
       else if(mark>=40)
           printf("\n Grade = \"Third Class\"");
         else
           printf("\n Grade = \"Fail\"");
 getch();
}
```

(f) The switch statement:

C has a built-in multiway decision statement known as a ***switch***. The switch statement tests the value of a given variable (or expression) against a list of ***case*** values and when a match is found, a block of statements associated with that case is executed.

The ***break*** statement at the end of each block signals the end of a particular case and causes an exit from the switch statement.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. The expression is an integer or character.

***syn*:**
```
switch(expression)
{
  case value-1 : statement(s);
                 break;
  case value-2 : statement(s);
                 break;
```

```
        . . . . . .
        . . . . . .
        case value-n : statement(s);
                       break;
        default      : statement(s);
                       break;
      }

/*  EX-14: DEMONSTRATING THE USE OF SWITCH  STATEMENT USING INTEGER EXPRESSION  */
#include<stdio.h>
main()
{
  int choice;
  clrscr();
  printf("\n\n\n\t    CHOICE OF DESTINATION \n");
  printf("\n\t\t 1 - MERCURY");
  printf("\n\t\t 2 - VENUS");
  printf("\n\t\t 3 - MARS");
  printf("\n\n\t Enter the number corresponding to your choice: ");
  scanf("%d",&choice);
  switch(choice)
  {
    case 1:
         clrscr();
         puts("\n\n\t Mercury is closest to the sun");
         puts("\n\t So the weather may be quite not hot there");
         break;
    case 2:
         clrscr();
         puts("\n\n\t Venus is the second planet from the sun");
         puts("\n\t The weather may be hot and poisonous");
         break;
    case 3:
         clrscr();
         puts("\n\n\t Mars is the closest planet to the earth");
         puts("\n\t The weather has been excellent till now");
         break;
    default:
         puts("\n\n\t\t \"UNKNOWN DESTINATION\"");
         break;
  }              /* End of switch */
  getch();
}              /* End of main   */


/*        EX-15: PGM TO DEMONSTRATE THE USE OF SWITCH STATEMENT USING
                  CHARACTER EXPRESSION  */
#include<stdio.h>
main()
{
  char choice;
  clrscr();
  puts("\n\n\t CHOICE OF YOUR DESTINATION");
  puts("\n\t\t G - Generator");
  puts("\n\t\t O - Operator");
  puts("\n\t\t D - Destroyer");
  printf("\n\n\t Press the letter corresponding to your choice: ");
  scanf("%c",&choice);
  switch(choice)
  {
   case 'g':
   case 'G':
        puts("\n\n\t\tSelected choice is GENERATOR");
        break;
   case 'o':
   case 'O':
        puts("\n\n\t\tSelected choice is OPERATOR");
        break;
```

```
  case 'd':
  case 'D':
       puts("\n\n\t\tSelected choice is DESTROYER");
       break;
  default:
       puts("\n\n\n\t\tINVALID CHOICE");
       break;
 }
getch();
}
```

break statement:

The break statement transfers the control to the end of the construct. In looping constructs, it transfers control to the next statement after the looping construct. In nested loop construct, the break statement terminates the inner most loop only.

continue statement:

The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. Instead, the remaining loop statements are skipped and the computer proceeds directly to the next pass through the loop.

```
/*        EX-16: PGM TO DEMONSTRATE THE USE OF CONTINUE STATEMENT    */
#include<stdio.h>
#define MAX 5         /* MAX is a predefined constant, having the value of 5 */
main()
{
 int n,i=1,sum=0;
 clrscr();
 printf("\n\t  INPUT ANY 5 INTEGER NOS.");
 for(i=0;i<MAX;i++)     /* for loop will explain later  */
 {
  printf("\n Enter an integer: ");
  scanf("%i",&n);           /* %i - access specifier for unsigned integer */
  if(n<0)
  {
   puts("\n You have entered a negative number");
   continue;
  }
  sum+=n;
 }
 printf("\n The sum of the positive integers entered = %i",sum);
 getch();
}
```

Note :

1. #defined MAX 5 is a predefined constant, where MAX having the value 5.
2. %i is an access specifier for unsigned integer.

Sample Run :

```
Enter an integer: 2
Enter an integer: -4
You have entered a negative number
Enter an integer: 4
Enter an integer: 11
Enter an integer: 7
The sum of the positive integers entered = 24
```

goto statement:

The goto statement is used to alter the normal sequence of program execution by unconditionally transferring control to some other part of the program. This statement is written as

```
goto label;
```

Here, label is an identifier used to identify the target statement to which the control would be transferred. The target statement must be labeled and a colon must follow it. The target statement will appear as:

```
label: statement;
```

B) LOOP CONSTRUCTS

Loops in 'C' cause a section of the program to be executed repeatedly while an expression is true. When the expression becomes false, the loop terminates and the control passes on to the statement following the loop. A loop consists of two segments, one is known as the control statement and the other is the body of the loop.

The 'C' language provides three loop constructs for performing loop operations.
They are:

1. The *do..while* loop constructs
2. The *while* loop constructs
3. The *for* loop constructs

(a) *do..while* loop:

The do..while loop, evaluates the condition after execution of the statements in its construct. The statements within the do..while loop are executed at least once. So, the do..while loop is called a bottom-tested loop.

Syn :
```
do
{
  statement(s);
} while(condition);
```

```
/*      EX-17: ILLUSTRAION OF DO-WHILE LOOP    */
#include<stdio.h>
main()
{
  int x=10;
  clrscr();
  do
  {
    printf("\n%d",x);
    x++;          /* Short form of x=x+1;  */
  }while(x<20);
  getch();
}
```

Note :

After run this program once, change the condition as while (x<=8) and check out the output.

(b)*while* loop:

The while loop is often used, when the numbers of times the loop is to be executed is not known in advance. The while loop is top-tested i.e., it evaluates the condition before executing any of the statements in its body.

Syn :
```
while(condition)
{
 statement(s);
}
```

```
/*      EX-18: ILLUSTRATING WHILE LOOP          */
#include<stdio.h>
main()
{
  int i=5;
  clrscr();
  while(i<10)
  {
    printf("\n National Institute");
    i++;
  }
  getch();
}
```

```
/*  EX-19: ILLUSTRATION OF WHILE LOOP UNTIL USER WISHES TO TERMINATE   */
#include<stdio.h>
main()
{
 char name[30],ch='y';
 clrscr();
 while(ch=='y' || ch=='Y')
 {
   fflush(stdin);
   printf("\n Enter the student's name in a Computer Department: ");
   gets(name);
   printf("\n\n\t Do you want to continue (y/n): ");
   scanf("%c",&ch);
```

```
  }
}
```

(c) *for* loop:

The *for* loop is useful while executing a statement to a number of times. The keyword for is followed by the initialization, condition and iteration are comes into a single statement.

syn : for(initialization; condition; iteration)
```
        {
          statement(s);
        }
```

```
/* EX-20: PGM TO DISPLAY FIRST 5 NUMBERS MULTIPLIES OF 3 ON A SINGLE LINE */
#include<stdio.h>
main()
{
  int i;
  clrscr();
  printf("\n");
  for(i=1;i<=5;i++)
  printf("\t %i", 3*i);
  getch();
}
```

Output : 3 6 9 12 15

Comma operator:

Comma operator (,) is used to combined two related expressions into one, making programs more compact.

```
/*        EX-21: ILLUSTRATING COMMA OPERATOR IN FOR LOOP          */
main()
{
  int i,sum;
  clrscr();
  for(sum=0,i=0; i<8; i++,sum+=i)
  printf("\n Sum = %d", sum);
  getch();
}
```

Output :

```
Sum = 0
Sum = 1
Sum = 3
Sum = 6
Sum = 10
Sum = 15
Sum = 21
Sum = 28
```

ARRAY

INTRODUCTION:

An array is a sequence of data in memory, wherein all data are of the same type, and are placed in physically adjacent locations. *For example*, a sequence of 10 integers stored one after another in memory, represents an ***array.***

Note that a string can be considered as a sequence of characters. The 'C' language treats Strings as an array of characters, the last of which is the null character (Character with an ASCII value zero).

USE OF AN ARRAY:

Consider the problem of accepting the *salary* of 5 employees, printing them out individually and computing the average, without using arrays. Here, it reads the salary of 5 employees, adds them up in to another variable called *sum* and divide it by 5 to get the *average salary*. Needless to say, this is a very clumsy way to write a program. If then are a large number of employees; the number of statements also increases proportionately. To avoid this, use a loop. But the problem is that different variables are needed to store the salary of different employees and hence, the loop will have to store and print different variables each time.

A more elegant way is to use an array of integers to store the salary. So an array is defined as a set of homogeneous data items. Note that we can use any data type to declare an array.

Syn :

```
Data_type array_name[array_size];
```

Example :

int sal[9];

Array_Element	sal[0]	sal[1]	sal[2]	sal[3]	sal[4]	sal[5]	sal[6]	sal[7]	sal[8]
Value	1200	2500	2750	3750	5200	2350	3000	3500	4200

TYPES OF AN ARRAY:

Arrays whose element are specified by one subscript are called single dimensional array. Corresponding arrays whose elements are specified by two and three subscripts are called two-dimensional arrays respectively.

SINGLE-DIMENSIONAL ARRAY:

A list of items can be given a variable name using only one subscript and such variable is called a single dimensional array.

```
/*        EX-22: PRG TO ILLUSTRATING THE SINGLE DIMENSIONAL ARRAY  */
#include<stdio.h>
main()
{
  int i;
  float avg,sal[8],sum=0;
  clrscr();
  printf("\n Enter Basic salary for 8 employees: \n");
  for(i=0;i<8;i++)
   {
     scanf("%f",&sal[i]);
   }
  clrscr();
  printf("\n\n\n\n\n\t\t\t\" LIST OF SALARY \"\n");
  for(i=0;i<8;i++)
   {
     printf("\n\t\t\t\t%-6.2f",sal[i]);
     sum+=sal[i];
   }
   avg=sum/8;
   printf("\n\n\t\t\tSum of Salary : %-10.2f", sum);
   printf("\n\t\t\tAverage Salary: %-6.3f",avg);
   getch();
}

/* EX-23: READ A LINE OF LOWER-CASE TEXT AND WRITE IT OUT IN UPPERCASE
                    USING ONE-DIMENSIONAL CHARACTER ARRAY  */
#include<stdio.h>
#define SIZE 80
main()
{
  char letter[SIZE];
  int i;
  clrscr();
  printf("\n\t Enter any string for only one line (80 Character): \n");
  for(i=0;i<SIZE;++i)          /*   ++i is a pre-increment statement   */
    letter[i]=getchar();
  for(i=0;i<SIZE;++i)
    putchar(toupper(letter[i]));
  getch();
}
```

Note:

The symbolic constant SIZE is assigned a value of 80. This symbolic constant, rather than its value, appears in the array definition and in the two *for* statements. Remember that the value of the symbolic constant will be substituted for the constant itself during the compilation process. Therefore, in order to alter the program to accommodate a different size array, only the *#define* statement must be changed.

```
 /* EX-24: PGM TO FIND THE LARGEST & SMALLEST ELEMENT IN THE VECTOR */
#include<stdio.h>
main()
{
  int i,n;
```

```
 float a[50],large,small;
 clrscr();
 printf("size of vector: ");
 scanf("%d",&n);
 printf("\n Vector elements: \n");
 for(i=0;i<n;i++)
 scanf("%f",&a[i]);

 /* Initialization */
  large=a[0];
  small=a[0];

 /* largest and smallest elements */
 for(i=1;i<n;i++)
 {
   if(a[i]>large)
     large=a[i];
   else if(a[i]<small)
     small=a[i];
 }
 printf("\n Largest element is vector is %5.2f\n",large);
 printf("\n Smallest element in vector is %5.2f",small);
 getch();
}
```

***Note**:*
Output displays with the field width of 5 integer and 2 decimal places(Right Justified).

MULTI-DIMENSIONAL ARRAY:

If an array having two or more subscripts, then an array is called multidimensional array.

Syn: data_type var[sub1][sub2].....[subn];

Often, there is a need to store and manipulate two-dimensional data structures such as matrices and tables. Here, the array has two subscripts. One subscript denotes the row, and the other is column. As before, all subscript (row and column) starts with zero.

subject code - - - - - ->

r o l l n o

	0	1	2
0	**60**	**52**	**81**
1	**39**	**45**	**43**
2	**70**	**68**	**77**
3	**80**	**82**	**73**

Here, the table has 4 roll nos. and each roll no. has 3 subject code.

Syn: data_type var_name[subscript1] [subscript2];

Ex: int mark[4][3];

```
/* EX-25: PGM TO ACCEPT 9 NOS AND PRINT IT IN 3 BY 3 MATRIX FORMAT */
 main()
{
 int n[3][3];
 int r,c;
 clrscr();
 printf("\n Enter any 9 nos: \n");
 for(r=0;r<3;r++)
 {
   for(c=0;c<3;c++)
    scanf("%d",&n[r][c]);
 }
 clrscr();
 gotoxy(23,10);
 printf(" \"OUTPUT IN 3 BY 3 MATRIX\"\n");
 for(r=0;r<3;r++)
 {
   printf("\n\n\t\t\t");
```

```
    for(c=0;c<3;c++)
   printf("%6d",n[r][c]);
  }
  getch();
}

/*          EX-26: ADDITION OF TWO MATRICES          */
#include<stdio.h>
main()
{
  int a[10][10],b[10][10],c[10][10];
  int i,j,m,n,p,q;
  clrscr();
  printf("Enter row & column of Matrix-A\n");
  scanf("%d %d",&m,&n);
  printf("Enter row & column of Matrix-B\n");
  scanf("%d %d",&p,&q);
  if((m==p) && (n==q))
  {
   printf("Matrix can be added\n");
   printf("Input the elements for matrix-A\n");
   for(i=0;i<m;i++)
     for(j=0;j<n;j++)
      scanf("%d",&a[i][j]);
   printf("Input the elements for matrix-B\n");
   for(i=0;i<p;i++)
     for(j=0;j<q;j++)
      scanf("%d",&b[i][j]);

   /* MATRIX ADDITION */
    for(i=0;i<m;i++)
     for(j=0;j<n;j++)
      c[i][j] = a[i][j] + b[i][j];

   printf("\n\t\t SUM OF TWO MATRICES\n");
   for(i=0;i<m;i++)
   {
     printf("\n\n\t\t");
     for(j=0;j<n;j++)
      printf("%5d",c[i][j]);
   }        /* End of if statement */
  }
  else
  printf("\n\t MATRIX CAN NOT BE ADDED");
  getch();
}          /* End of main     */
```

STRINGS

Strings are arrays of characters, i.e., they are characters arranged one after another in memory. To mark the end of the string, C uses the null character. Strings in C are enclosed within double quotes. The string is stored in memory as ASCII codes of the characters, that make up the string appended with 0 (ASCII value of null).

Initializing strings: **Ex:** char day[]={ 'F', 'R', 'I', 'D', 'A', 'Y' 0 };

'C' offers the modified way to initialize the string as follows:

```
char day[] = "FRIDAY";
```

LENGTH OF AN ARRAY:

```
/* EX-27: PGM TO ACCEPT A STRING AND FIND ITS LENGTH USING ARRAY */
#include<stdio.h>
main()
{
  char str[30];
  int i,len=0;
```

```
  clrscr();
  printf("\n Enter a string: ");
  gets(str);
  for(i=0; str[i]!='\0'; i++)
   len++;
  printf("\n Length of %s = %d",str,len);
  getch();
}
```

REVERSE A STRING USING AN ARRAY:

```
/*        EX-28: PGM TO ACCEPT A STRING AND REVERSE IT USING ARRAY  */
#include<stdio.h>
main()
{
  char str[30];
  int i,len=0;
  clrscr();
  printf("\n Enter a string: ");
  gets(str);
  for(i=0;str[i]!='\0';i++)
   len++;
  while(len>=0)
  {
    printf("%c",str[len]);
    len--;
  }
  getch();
}
```

STANDARD LIBRARY FUNCTIONS:

strcat()

This function concatenates two strings, i.e., it appends one string at the end of another. The function accepts two strings as parameters and stores the contents of the second string at the end of the first.

```
 /* EX-29: PGM TO CONCATENATE THE TWO STRINGS USING STD LIBRARY FUNCTION */
#include<string.h>
main()
{
  char str1[30]="SUMITHRA ";
  char str2[]="THIYAGARAJAN";
  clrscr();
  printf("\n\n\t\tCONCATENATED STRING : ");
  strcat(str1,str2);
  puts(str1);
  getch();
}
```

Output: SUMITHRA THIYAGARAJAN

Note:

1. Whenever you use the strcat function, check to see whether the first string is large enough to hold the resultant concatenated string.
2. In the declaration of array, if the subscript is omitted, it is assumed to be of the size of the data with which the array is initialized.

strcmp()

The function strcmp() compares two strings. This function is useful while writing programs for constructing and searching strings as arranged in a dictionary. The function accepts two strings as parameters and returns an integer, whose value is:

Less than 0 if the first sting is less than the second
Equal to 0 if both are identical
Greater than 0 if the first string is greater than the second

The function strcmp() compares the two strings, character by character, to decide the greater one. Whenever two characters in the string differ, the string, which has the character with a higher ASCII value, is greater.

For example, consider the strings hello and Hello. The first character itself differs. The ASCII code for h is 104, while that for H is 72. Since the ASCII code of H is greater, the string hello is greater than the string Hello. Once a differing character is found, there is no need to compare the other characters of the strings.

```
/*  EX-30: PGM TO COMPARE TWO STRINGS USING STD LIBRARY FUNCTION  */
#include<string.h>
main()
{
  char str1[30],str2[30];
  int result;
  clrscr();
  printf("Enter any two strings: \n");
  scanf("%s %s", str1,str2);
  result=strcmp(str1,str2);
  if(result<0)
    printf("str1 < str2");
   else if(result==0)
     printf("str1 == str2");
    else
     printf("str1 > str2");
  getch();
}
```

Note: The operators <, > and == cannot be used directly with strings.

stricmp() or **strcmpi()**

This function also compares two strings, but it ignores case (i.e., upper case or lower case). Try to do the same example, used above.

strcpy()

The function strcpy, copies one string to another, character by character including null character.

syn: strcpy(target, source);

```
/*  EX-31: PGM TO COPYING STRING FROM ONE VAR TO ANOTHER VAR USING STD
                LIBRARY FUNCTION */
#include<string.h>
main()
{
  char s1[60],s2[30];
  clrscr();
  printf("\n Enter a string: ");
  gets(s1);
  strcpy(s2,s1);
  printf("\n String after copied = %s",s2);
  getch();
}
```

strlen()

The function strlen returns an integer, which denotes the length of the string passed. The length of a string is the number of characters presents in it, excluding the terminating null character.

```
/* EX-32: PGM TO FIND THE LENGTH OF STRING USING STD LIBRARY FUNCTION */
#include<stdio.h>
main()
{
  int len;
  char str[40];
  clrscr();
  printf("\n Enter a string: ");
  gets(str);
  len=strlen(str);
  printf("\n Length of the string %s = %d", str,len);
  getch();
}
```

strrev()

The function strrev(), reverse a given string and it stores the reversed string into the same variable.

```
/*        EX-33: PGM TO REVERSE A STRING USING STD LIBRARY FUNCTION */
```

```
#include<string.h>
main()
{
 char str[30];
 clrscr();
 printf("\n Enter a string: ");
 gets(str);
 strrev(str);
 printf("\n After reverse, the string = %s", str);
 getch();
}
```

FUNCTIONS

DEFINITION:

A function is a self-contained program segment that carries out some specific, well-defined task.

USER-DEFINED FUNCTION:

The user has the freedom to choose the function name, return data type and the arguments (no. and type), is called user defined function.

Advantages of using functions:

1. Repeated statement can be grouped into functions.
2. Splitting the programs into separate function makes the program more readable and maintainable (easy to rectify the errors if occurs).

The function main() in the program is executed first. The other functions are executed when the function main() calls them directly or indirectly.

The function main() is also a user defined function except that the name of the function, the no. of arguments, and the argument types are defined by the language. The individual statements are written by the programmer in the body of the function main(). The following, in a sense, is complete C program.

```
main() { }
```

The above statement is sufficient for the execution of the program, though the program does not do any useful operation. The function main() is executed first, when the program starts execution. Other user-defined functions can be called from the function main().

The following example has a function PrintMsg in addition to main. If a call to the function PrintMsg is included in the function main, the statement within the body of the function main. A prefix void is used before the function name, i.e., PrintMsg inticating that the function does not return any value.

```
/*        EX-34: TO PRINT A MESSAGE USING FUNCTION  */
#include<stdio.h>
void PrintMesg()
 {
  printf("\n\n\t\tStudying at National Institute, Pazhavanthangal");
 }

void main()
{
 clrscr();
 printf("\n\n\n\n\n\n\n\n\n\t\t\t I am in Main City - Chennai");
 PrintMesg();
 getch();
}
```

Note:

The function name in the above example (PrintMesg) has mixed case, in contrast to the functions, that we have used so far(such as printf and scanf) which consists entirely of lowercase characters. Using mixed case is a popular alternative to using underscores to make identifier names more readable. Function names are also identifiers. Hence, functions can have any name, as long as they satisfy the rules for identifiers.

```
/*  EX-35: FUNCTION MAIN CALLS MORE THAN ONE USER DEFINED FUNCTION  */
#include<stdio.h>
comp()
{
  printf("\n\n\t\t * Free Computer Courses");
}
```

```
tut()
{
  printf("\n\t\t * Coaching class for all students");
}

main()
{
  clrscr();
  printf("\n\n\n\n\n\t\t   \"NATIONAL INSTITUTE OFFERS\"");
  comp();
  tut();
  printf("\n\n\n\n\n\t\t -->   \"VISIT AND JOIN\"  <--");
  getch();
}
```

Note:

The functions comp() and tut() are placed before the function main. This is appropriate, because the function main used these functions and hence needs to know about it. However, in many cases, it is desirable to place the function main at the beginning in order to give the reader on over view of the logic.

The problem now is that while compiling main(), the functions called in it are not known to the compiler. To assist the compiler, we write the information about each function interface: its name, return type and the parameter, which it appears, prior to main itself. In main, these functions can be called. The actual code (definition) of these functions can be placed after main.

FUNCTION WITH ARGUMENTS BUT NO RETURN VALUES:

In the following program, the function square is supplied with an argument ***no***. This function calculates the square of the given value based on the values sent to it, but it does not return any value (square of no) to the calling function.

When the function is called from the calling function, control passes to the function, then the result is evaluated, and then it displays the output.

```
/*        EX-36: FUNCTION WITH ARGUMENTS BUT NO RETURN VALUE*/
#include<stdio.h>
main()
{
  void square(float);
  float no,result;
  clrscr();
  printf("\n\tEnter the no. whose square has to be found: ");
  scanf("%f",&no);
  square(no);
  getch();
}

void square(float num)
{
  float res;
  res=num*num;
  printf("\n\tThe square of the number is %f",res);
}
```

The function definition starts with the return data type, the function name, and the arguments it accepts (within round braces). However, after the closing braces, there is no semicolon. The function body follows, enclosed within a pair of curly braces.

In large programs having several functions, there is an added advantage in declaring all the functions at the beginning; we do not need to keep track of the order in which the functions are called.

FUNCTION WITH ARGUMENTS AND RETURN VALUES:

In the following program, the function *value* accepts three arguments namely principle, inrate, period and manipulate the values of them. When this function is called from main(), the control passes to the called function, calculated the simple interest and the amount is returned to the calling function from called function.

```
/*        EX-37: FUNCTION WITH ARGUMENTS AND IT RETURNS VALUE          */
#include<stdio.h>
main()
{
  float value(float, float,int);
```

```
 float principal, inrate, amount;
 int period;
 clrscr();
 printf("Enter Principal amount: ");
 scanf("%f",&principal);
 printf("Enter Rate of interst: ");
 scanf("%f",&inrate);
 printf("Enter the period(No. of years): ");
 scanf("%d",&period);
 amount=value(principal,inrate,period);
printf("\n\n   PRINCIPLE AMOUNT   RATE OF INTEREST   PERIOD    SIMPLE INTEREST");
printf("\n\n    %f\t   %f   \t %d   \t   %f",principal,inrate,period,amount);
getch();
}
float value(float p,float r,int n)
{
 float netamt;
 netamt=(p*n*r/100);
 return(netamt);
}
```

Note: A function can return only one value of integer data type to the calling function.

Conditions must satisfy the function call:

1. The no. of arguments in the function call and in the function declaration should be same.
2. The data type of each of the arguments in the function call should be the same as the corresponding parameter in the function declaration statement i.e., the order in which we specify the arguments while calling the function must be same as the order in which the parameters are specified.

However the names of the arguments in the function call and the names of parameters in function definition are unrelated. They can be same or different.

MECHANISMS OF PASSING ARGUMENTS:

C provides two mechanisms to pass arguments to a function

- Pass arguments by value, and
- Pass arguments by address or by pointers.

An argument may take the form of a constant, variable, expression, pointer, structure etc.

i) PASSING BY VALUE:

Function in C passes all arguments by value. Passing arguments by value means that the contents of the arguments in the calling function are not changed, even if they are changed in the called function. This is because the content of the variable is copied to the formal parameter of the function definition, thus preserving the contents of the arguments in the calling function.

Passing arguments by value is useful when the function does not need to modify the values of the original variables in the calling program.

```
/*        EX-38: SWAPPING THE NUMBER USING CALL BY VALUE  */
#include<stdio.h>
void swap(int a,int b)
{
 int temp;
 temp=a;
 a=b;
 b=temp;
printf("\nIn function: Swapped values: %d %d\n",a,b);
}

main()
{
 int x,y;
 clrscr();
 printf("\nEnter any two integers:\n");
 scanf("%d %d",&x,&y);
 printf("\nIn main(): Before swapping : %d %d\n",x,y);
```

```
 swap(x,y);
 printf("\nIn main(): After swapping  : %d %d",x,y);
getch();
}
```

Sample run:

```
Enter any two integers:
          8    3
In main(): Before swapping  : 8  3
In function: Swapped values : 3  8
In main(): After swapping     : 8  3
```

In above program, when the main function calls swap(x,y); the values of the variable x and y are copied into the parameters of the function swap, a and b respectively. The values of a and b are interchanged in the swap function. The function returns and the values of a and b are discarded. i.e., the original value in main() will not affect using call by value.

ii) PASSING BY REFERENCE:

Instead of passing two integers to the swap function, we can pass the addresses of the integers that we want to swap. For this, two changes are required.

First, the function definition must be changed to accept the address of the two integers. Specifying does this

```
void swap(int *a, int *b)
```

instead of

```
void swap(int a, int b)
```

Inside the function also we have to access the value of address of *a*. This is done by using *a instead of a and *b instead of b.

Second, the call in main() must be changed to pass the addresses of x and y instead of their values. This is done by calling swap(&x, &y);

```
/*        EX-39: FUNCTION - CALL BY REFERENCE     */
#include<stdio.h>
void swap(int *a,int *b)
{
 int temp;
 temp=*a;
 *a=*b;
 *b=temp;
 printf("\n\nIn function: Swapped values: %d %d\n",*a,*b);
}

void main()
{
 int x,y;
 clrscr();
 printf("\n\n Enter any two integers: \n");
 scanf("%d %d",&x,&y);
 printf("\n\nIn main(): Before swapping : %d %d\n",x,y);
 swap(&x,&y);
 printf("\n\nIn main(): After swapping   : %d %d",x,y);
 getch();
}
```

ITERATION AND RECURSION:

Expressing an entity in terms of itself is called ***recursion***. In C, a function can call any function that has been defined, including itself.

The definition of factorial can be stated in two ways:

- The factorial of 0 is 1, and the factorial of any positive integer 'n' is the product of all integers from 1 to n.
- The factorial of 0 is 1, and factorial of any positive integer 'n' is the product of 'n' and the factorial of the number n-1.

The first definition is iterative whereas the second is recursive. In the second definition, "what is the factorial of n-1?"It is the product of n-1 and the factorial of n-2[i.e., (n-1)*n-2)!]. It continues till 'n' becomes 0, in which case the factorial is 1.

```
/*        EX-40: PGM TO FIND FACTORIAL OF A NUMBER USING ITERATIVE
                                                    (WITHOUT RECURSION)  */
#include<stdio.h>
```

```
long int fact(int num)
{
  int i;
  long int f=1;
  for(i=num; i>1; i--)
  f*=i;
  return f;
}

void main()
{
  int n;
  clrscr();
  printf("\n\tEnter an integer: ");
  scanf("%d",&n);
  printf("\n\tThe factorial of %d is %ld",n,fact(n));
  getch();
}
```

Note: %ld is a access specifier for the data type of long integer.

Sample Run:

Enter an integer: 5
The factorial of 5 is 120 (i.e., 5! = 5*4*3*2*1)

```
/*        EX-41: PGM TO FIND FACTORIAL OF A GIVEN NUMBER USING RECURSION */
#include<stdio.h>
factorial(n)
int n;
{
  int fact;
  if(n==0)
    return(1);
   else
    fact=n*factorial(n-1);
  return(fact);
}

main()
{
  int j;
  clrscr();
  for(j=0;j<=5;j++)
   printf("\n\t%d\n", factorial(j));
  getch();
}
```

Recursion is implemented through the use of functions. A function that contains a function call to itself, or a function cal to a second function, which eventually calls the first function, is known as a recursive function.

Two important conditions must be satisfied by any recursive function:

- Each time a function calls itself it must be closer, in some sense to a solution.
- There must be a decision criterion for stopping the process of computation.

For the function factorial, each time the function calls itself, its argument is decrement by one (condition 1). The stopping criterion is the *if* statement that checks for the zero argument.

POINTERS

INTRODUCTION:

Pointer in 'C' offers the flexibility in programming. It reduces the length and complexity of a program, increase the execution speed and use of a pointer array to character strings results in saving of memory space in memory. Prior to understanding the importance of pointers briefly in C, we should know about the organization of memory in computers.

Memory is organized as a sequence of byte-sized locations. These bytes are numbered beginning with a zero. The number associated with a byte is known as its address or memory location. A pointer is an entity, which

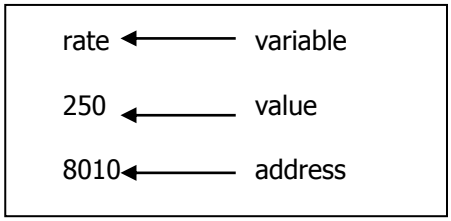
contains a memory address. Thus, a pointer is a number, which specifies a location in memory. The key concepts and terminology associated with memory organization are listed below.

- Each byte in memory is associated with a unique address. An address is an integer having fixed number of bits, labeling a byte in the memory.
- Addresses are positive integer values that range from zero to the maximum size of memory.
- Every object is loaded into memory is associated with a valid range of addresses, that is, each variable and each function in a program starts at a particular location, and spans across consecutive addresses from the point onwards depending upon the size of the data item.
- The size of the data type referenced by the pointer is the number of bytes that may be accessed directly by using that address.

Consider the following declaration statement,

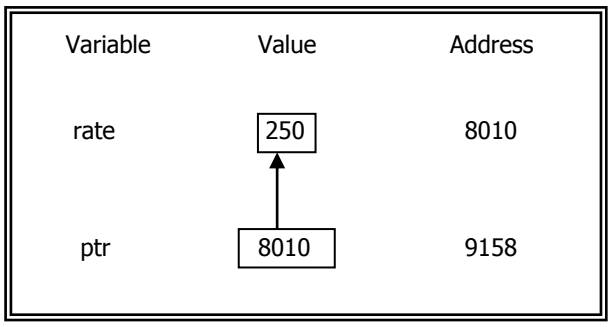
int rate = 250;

This statement instructs the system to find a location for the integer variable *rate* and stores the value 250 in to that location. Assume that the system has selected the address location 8010 for *rate*. We can represents this as follows:



When executing the program, the system always associates the name *rate* with the address 8010. We can access the value 250 by using either the name *rate* or the address 8010. Since memory addresses are simply numbers, they can be assigned to some variable, which can be stored in memory, like other variables. Such variables that hold memory addresses are called pointers. A pointer is, therefore, nothing but a variable that contains an address which is a location of another variable in memory.

Since, a pointer is a variable; its value is also stored in the memory in another location. Suppose, we assign the address of *rate* to the variable *ptr*. The link between the variable *ptr* and *rate* can be visualized as shown in fig. The address of *ptr* may be 9158. So the value of the variable *ptr* is the address of the variable *rate*, we may access the value of *rate* by using the value of *ptr* and therefore, we say that the variable '*ptr*' points to the variable *rate*. Thus, *ptr* get the name "***pointer***".



ADDRESS OPERATOR (&):

The location of variable in memory is system dependent, so user does not know the address of a variable immediately. To determine the address of variable use the address operator '&'. We have already seen the use of this address operator in the scanf function. The operator & immediately preceding a variable, returns the address of the variable associated with it.

For example, If we give a print statement as

printf("%d", &rate);

then, it prints the corresponding address of variable *rate*.

```
/*        EX-42: ILLUSTRATING THE USE OF ADDRESS OPERATOR '&'   */
main()
{
 int p=75;
 float q=14.10;
```

```
 char ch='B';
 clrscr();
 printf("\n\t %d is stored at address %u",p,&p);
 printf("\n\t %f is stored at address %u",q,&q);
 printf("\n\t %c is stored at address %u",ch,&ch);
 getch();
}
```

DE-REFERENCING POINTERS:

De-referencing is an operation performed to access and manipulate data contained in the memory location pointed to by a pointer. The operator * is used to de-reference pointers. A pointer variable is de-referenced when the unary operator *, in this case called the indirection operator, is used as a prefix to the pointer variable or pointer expression. Any operation performed on the de-referenced pointer directly affects the value of the variable it points to.

```
/*        EX-43: ILLUSTRATES DEREFERENCING OF POINTER   */
main()
{
 int *p,x,y;   /* Declaration of pointer variable and ordinary variable */
 p=&x;        /* Initializing pointer p  */
 *p=5;        /* Dereferencing p,x = 5  */
 *p+=10;
 clrscr();
 printf("\n\tVariable x = %i\n",x);
 y=*p;
 printf("\n\tVariable y = %i\n",y);
 p=&y;
 *p+=15;
 printf("\n\tNow the Variable y = %i",y);
 getch();
}
```

OUPTUT:

Variable x=15
Variable y=15
Variable z=30

Note:

1. The asterisk (*) used as an indirection operator has a different meaning from the asterisk used to declare pointer variables.

```
int *p;
```

2. Indirection allows the contents of a variable be accessed and manipulated without using the name of the variable. All variables that can be accessed directly by their names, indirectly by means pointers. The power of pointers becomes evident in situations where indirect access is the only way to access variables in memory.

```
/*  EX-44: POINTER VARIABLE THAT CONTAINS AN ADDRESS OF ANOTHER VARIABLE  */
main()
{
 int i=20;
 int *j;
 int **k;
 j=&i;
 k=&j;
 clrscr();
 printf("\n\t%d \t%d \t%d \t%d",i,*(&i),*j,**k);
 printf("\n\t%d \t%d \t%d", j,*(&j),*k);
 printf("\n\t%d \t%d",k,*(&k));
 getch();
}
```

POINTER ARITHMETIC:

The 'C' language allows arithmetic operations to be performed variables. It is, however, the responsibility of the programmer to see that the result obtained by performing pointer arithmetic is the address of relevant and meaningful data.

* Unary Operator : ++ (increment) and -- (decrement)
* Binary Operator: + (addition) and – (subtraction)

```
/*        EX-45: ILLUSTRATING ARITHMETIC OPERATION USING POINTERS       */
```

```
main()
{
  int a,b,c;
  int *pa,*pb;
  clrscr();
  printf("\n Enter any two numbers: \n");
  scanf("%d %d",&a,&b);
  pa=&a;
  pb=&b;
  c=*pa + *pb;
  printf("\n Sum of %d and %d = %d", *pa,*pb,c);
  getch();
}
```

POINTERS AND FUNCTIONS:

Note:

Once again refer the call by value and call by reference, illustrated in previous chapter (functions).

POINTERS AND ARRAY:

Using a pointer can efficiently access the elements of an array. The following program illustrates the relation between pointers and arrays.

```
/*        EX-46: POINTER AND ARRAY          */
main()
{
  int iarray[5] = {1,2,3,4,5};
  int i,*ptr;
  ptr=&iarray[0];
  clrscr();
  for(i=0;i<5;i++)
  {
   printf("%d",ptr);
   printf("\n%d",*ptr);
   ptr++;              /* Increment the pointer to point to point to the next element
                          and not to the next memory location */
  }
  getch();
}
```

In the above program, an integer array of size 5 is declared and initialized. Then, the address of the 0^{th} element of the array is assigned to the integer pointer ptr. Within the loop, the address of the array elements is stored in ptr, and the contents of the memory location pointed to by ptr are printed. When ptr is incremented, the value stored in ptr is incremented by 2 to point to the element of the array (each integer occupies 2 bytes of memory).

```
/*        EX-47: TO FIND SUM OF 7 NUMBERS USING POINTER AND ARRAY  */
main()
{
  int a[7],i,*ptr,s=0;
  clrscr();
  printf("\n Enter any seven numbers: \n");
  for(i=0;i<7;i++)
   scanf("%d",&a[i]);
  ptr=&a[0];
  clrscr();
  for(i=0;i<7;i++,ptr++)
  {
    printf("\n\t%d",*ptr);
    s+=*ptr;
  }
  printf("\n Sum of above numbers = %d",s);
  getch();
}
```

POINTERS, ARRAYS AND FUNCTIONS:

We can use pointers in an array and functions. The following program illustrates that.

```
/*        EX-48: ILLUSTRATING POINTER, ARRAY AND FUNCTION  */
#include<stdio.h>
main()
{
  void display(int *);
  int no[5],*ptr,i;
  clrscr();
  printf("\n Enter any 10 integers:\n");
  for(i=0;i<5;i++)
   scanf("%d",&no[i]);
  ptr=&no[0];
  display(ptr);
  getch();
}

void display(int *p)
{
  int i;
  printf("\n\t The number you have entered:");
  for(i=0;i<5;i++,p++)
   printf("\n\t\t%d",*p)
}
```

POINTERS AND STRINGS:

Pointers are widely used in handling some of the standard string library functions include strlen(), strcat(), strcpy() and strcmp(). A pointer to the string is passed to these functions instead of the entire string. A pointer thus defined can be assigned the address of an existing string, or allocated memory space, and then assigned the pointer returned by the allocation.

```
/*    EX-49: ILLUSTRATING POINTERS AND STRINGS - TO PRINT CHAR. BY CHAR. */
main()
{
  char str[30],*ptr;
  clrscr();
  printf("\n Enter a string : ");
  gets(str);
  ptr=str;
  while(*ptr!='\0')
  {
    printf("\n\t%c",*ptr);
    ptr++;
  }
  getch();
}
```

INPUT : MEENA
OUTPUT: M
E
E
N
A

```
/*        EX-50: PGM TO ACCEPT A STRING AND CHECK WHETHER
                          IT IS A PALINDROME OR NOT USING POINTERS  */
main()
{
  char str[30];
  char *p1,*p2;
  int flag=0;
  clrscr();
  printf("\n Enter a string: ");
  gets(str);
  p2=str;
  while(*p2)
  p2++;
  p2--;
  p1=str;
  while(*p1)
  {
```

```
   if(*p1 == *p2)
    {
      p1++;
      p2--;
    }
   else
    {
      flag=1;
      break;
    }
 }
 if(flag==0)
   printf("\n\t Yes Palindrom");
 else
   printf("\n\t Not a Palindrom");
 getch();
}
```

strcpy():

This function copies the contents of one string to another. It takes two arguments, the first being the pointer to the beginning of the destination string and the second is a pointer to the first element of the source string. The source string is copied into the destination string.

```
/* EX-51: PGM ILLUSTRATES STRCPY USING STD. LIB. FN AND USER DEFINED FUNCTION  */
#include<string.h>
void udf(char *);
main()
{
/* int len;*/
  char s1[]="SHANKAR";
  char s2[50],*p;
  strcpy(s2,s1);
  clrscr();
  printf("\n\n\t Copied string using standard library function:  %s",s2);
  udf(s1);
}

void udf(char* s1)
{
  int i=0;
  char name2[20],*ptr;
  ptr=&s1[20];
  i=0;
  while(s1[i]!='\0')
  {
    name2[i]=s1[i];
    ptr++;
    i++;
  }
  name2[i]='\0';
  printf("\n\t Copied string using user defined function:  %s",name2);
  getch();
}
```

The function udf can be reduced by using while (*p++ = *q++); the expression *p++ will evaluate p++. The ++ operator is evaluated first, since unary operator associates right to left. The pointer is now de-referenced. The pointer p, is then incremented. Similar operations are done on q. The character pointed to by q is assigned to that pointed to that pointed to by p. The while loop continues till the NULL pointer is encountered.

strchr():

The function strchr() searches for a character in a string. It accepts two parameters. The first one is a string, while the second one is a character. The function searches for a specified character in the string passed and returns the memory address of the first matching symbol character found in the string. For example consider the following program.

```
/*  EX-52: PGM TO FIND POSITION OF A CHARACTER IN GIVEN STRING   */
#include<stdio.h>
#include<string.h>
main()
{
  char string[30];
```

```
 char findchar;
 char *found;
 printf("Input a string: ");
 /* scanf("%s",string); */
 gets(string);
 fflush(stdin);
 printf("Input a character: ");
 scanf("%c",&findchar);
 found=strchr(string,findchar);
 printf("%c was found in position %d",findchar,found-string);
 getch();
}
```

SAMPLE RUN 1:

```
 Input a String: Baskaran
 Input a character: k
 k was found in position 3
```

SAMPLE RUN 2:

```
 Input a String: Rathinavel
 Input a character: R
 R was found in position 0
```

Note: Array subscripts in 'C' always begin with a zero. But what happen if the character is not found in the given string? The function returns a NULL pointer, which is a memory address having the value zero.

STRUCTURE AND UNION

STRUCTURE

INTRODUCTION:

We have seen that arrays can be used to represent a group of data items that belong to the same type, such as *int* or *float*. However, if we want to represent a collection of data items of different types using a single name, then we cannot use an array. Fortunately, C supports a constructed data type known as ***structure***; which is a method for packing data of different types.

USE OF STRUCTURES:

Suppose, we want to store data about a book, we might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books is to be stored, then we can follow two approaches.

i) Construct individual arrays, one for storing names, another for storing prices and still another for storing no. of pages.

ii) Use of structure variable.

In first approach, the program becomes more difficult to handle as the no. of items relating to the book go on increasing. For example, we would be re required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book etc. To solve this problem, C provides a special data type, called structure.

A structure declaration starts with the keyword struct, which introduces the declaration. The declaration then follows with a list of variables enclosed within curly braces. These variables are called members. Members can be accessed using the member access operator . (dot).

Syntax for structure declaration statement:

```
struct <structure name>
{
  data_type  element1;
  data_type  element2;'
     . . . . . .
     . . . . . .
  data_type  elementn;
};
```

Once the new structure data type has been defined one or more variables can be declared to be of that type. This is as follows

struct <struct_name> <str_var1>, <str_var2>, ...<str_var*n*>;

We can combine the declaration of the structure type and the structure variables in one statement.

For example:

```
struct book
{
 char name;
 float price;
 int pages;
};
struct book b1,b2,b3;
```

is same as . . .

```
struct book
{
 char name;
 float price;
 int pages;
}b1,b2,b3;
```

Like primary variables and arrays, structure variables can also be initialised where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book
{
 char name[20];
 float price;
 int pages;
};
struct book b1 = { "MATHS", 130.00, 550 };
struct book b2 = { "COMPUTER", 150.00, 800 };
```

The following program illustrates the use of the structure.

```
/*      EX-53: ILLUSTRATES THE USE OF THE STRUCTURE       */
#include<stdio.h>
main()
{
  struct book
   {
    char name;
    float price;
    int pages;
   };
  struct book b1,b2,b3;
  clrscr();
  printf("\n Enter name, prices and no. of pages of 3 books: \n");
   scanf("%c %f %d", &b1.name, &b1.price, &b1.pages);
   fflush(stdin);
   scanf("%c %f %d", &b2.name, &b2.price, &b2.pages);
   fflush(stdin);
   scanf("%c %f %d", &b3.name, &b3.price, &b3.pages);

  printf("\n\t The 3 records you have entered:\n");
   printf("\n\t %c %f %d", b1.name, b1.price, b1.pages);
   printf("\n\t %c %f %d", b2.name, b2.price, b2.pages);
   printf("\n\t %c %f %d", b3.name, b3.price, b3.pages);
  getch();
}
```

MEMORY MAP OF STRUCTURE ELEMENTS:

Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program can illustrate this.

```
/*          EX-54: MEMORY MAP OF STRUCTURE ELEMENT        */
main()
{
  struct book
  {
    char *name;
    float price;
    int pages;
```

```
 };
 struct book b={ "SARASWATHY",81.25,408 };
 clrscr();
 printf("\n\n\n\n\n");
 printf("\n\t STRUCT_ELEMENT   VALUE \t MEMORY_LOCATION\n\n\n");
 printf("\t b.name  \t   %s  \t  %u (1 byte)\n\n",b.name,&b.name);
 printf("\t b.price \t   %f  \t  %u (4 bytes)\n\n",b.price,&b.price);
 printf("\t b.pages \t   %d  \t  %u (2 bytes)\n\n",b.pages,&b.pages);
 getch();
}
```

Here, the program output (in memory) may be as follows according to the size of the data type.

STRUCT_ELEMENT	VALUE	MEMORY_LOCATION
b.name	SARASWATHY	65484 (1 byte)
b.price	81.250000	65486 (4 bytes)
b.pages	408	65490 (2 bytes)

ARRAY OF STRUCTURES:

In above program, to store data of 100 books, we would be required to use 50 different structure variables from b1 to b50, which is definitely not very convenient. A better approach would be to use an array of structures.

Following program shows how to use an array of structures.

```
/* EX-55: USAGE OF AN ARRAY OF STRUCTURES  */
#include<string.h>
main()
{
 struct employee
 {
  int empno;
  char name[50];
  int salary;
 };
 struct employee e[3];
 int i;
 clrscr();
 for(i=0;i<=2;i++)
 {
  printf("\nEnter Emp.no, Name and Salary:\n");
  scanf("%d %s %d", &e[i].empno,&e[i].name,&e[i].salary);
/*   scanf("\n%d",&e[i].empno);
  scanf("\n%s",&e[i].name);
  scanf("\n%d",&e[i].salary);*/
 }
 for(i=0;i<=2;i++)
  printf("\n %d \t %s \t %d", e[i].empno, e[i].name, e[i].salary);
 getch();
}
```

COPYING BETWEEN STRUCTURE:

The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator. It is not necessary to copy the structure elements in field-wise (piece-meal copying). Obviously, programmers prefer assignment to piece-meal copying. The following program shown, copying between field-wise as well as record-wise.

```
/*        EX-56: COPYING BETWEEN STRUCTURE      */
main()
{
 struct employee
 {
   char name[20];
   int age;
   float salary;
 };
```

```
 struct employee e1 = { "MANICKAM", 10, 7584.75 };
 struct employee e2,e3;
 clrscr();
   /* Copying Fieldwise */
 strcpy(e2.name,e1.name);
 e2.age=e1.age;
 e2.salary=e1.salary;
   /* Copying Recordwise */
 e3=e2;
 printf("\n\tORIGINAL VALUE        : %s %d %f", e1.name,e1.age,e1.salary);
 printf("\n\tAFTER COPIED FIELDWISE : %s %d %f", e2.name,e2.age,e2.salary);
 printf("\n\tAFTER COPIED RECORDWISE: %s %d %f", e3.name,e3.age,e3.salary);
 getch();
}
```

NESTED STRUCTURES:

One structure can be nested within another structure. Using this facility complete data types can be created. The following program shows nested structures.

```
/*        EX-57: PGM ILLUSTRATES NESTED STRUCTURES      */
main()
{
 struct emp
 {
   char *name;
   struct emp_add
          {
            int no;
            char *street;
            char *area;
            long int pincode;
          }add;
          char *desig;
          float salary;
 };
 struct emp e={"KISHORE",98,"KOIL STREET","ERASAI",625515,"PROGRAMMER",12560.75};
 clrscr();
 printf("\n\t Name        = %s",e.name);
 printf("\n\t No          = %d",e.add.no);
 printf("\n\t Street      = %s",e.add.street);
 printf("\n\t Area        = %s",e.add.area);
 printf("\n\t PINCode     = %ld",e.add.pincode);
 printf("\n\t Designation = %s",e.desig);
 printf("\n\t Salary      = %0.2f",e.salary);
 getch();
}
```

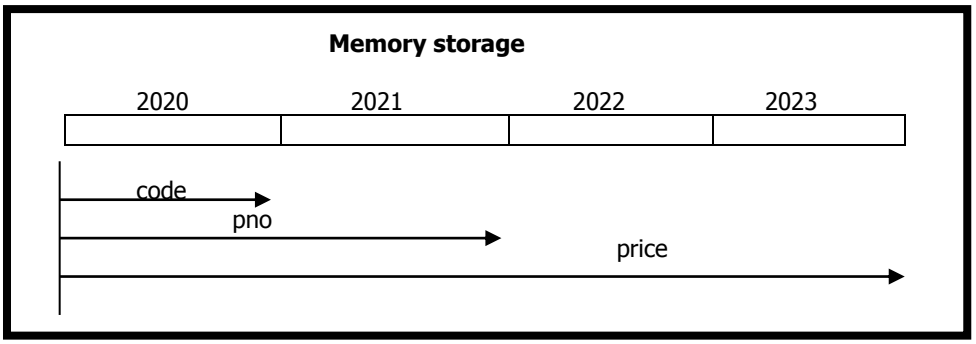
NOTE: %ld is an access specifier for long integer.

UNIONS

Union is a concept derived from structures and therefore follows the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword ***union*** as follows:

```
union product
{
 int pno;
 float price;
 char code;
}prod;
```

This declares a variable *prod* of type *union product*. The union contains three members, each with a different data 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.



Sharing of a storage location by union members

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 *price* requires 4 bytes which is the largest among the members. Figure shows how all the three variables share the same adress. This assumes that a *float* variable requires 4 bytes of storage.

```
/*          EX-58: ILLUSTRATING UNION          */
#include<stdio.h>
union course
{
  int major;
  char minor[10];
};
struct student
{
  char name[20];
  int rollno;
  union course course_no;
}student1;

main()
{
  char c_name;
  clrscr();
  printf("\n Enter the name of the student: ");
  scanf("%s",student1.name);
  fflush(stdin);
  printf("\n Enter the roll no. of the student: ");
  scanf("%d",&student1.rollno);
  fflush(stdin);
  printf("\n Enter the course ('M' for major or 'm' for minor): ");
  scanf("%c",&c_name);
  fflush(stdin);
  if(c_name=='M')
  {
   printf("\n Enter the course('1' or '2'): ");
   scanf("%d",&student1.course_no.major);
   fflush(stdin);
  }
  else
   strcpy(student1.course_no.minor,"others");
 printf("%s",student1.course_no.minor);
 getch();
}
```

FILE HANDLING

INTRODUCTION:

Until, we have been using the functions such as *scanf* and *printf* to read and write date. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.

2. The entire data is lost when either the program is terminated or the computer is turned off.

It becomes necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of ***files*** to store data.

A file is a place on the disk where a group of related data is stored.

Many application require that information be written to or read from an auxiliary storage. Such information is stored on the device in the form of a "data file". Thus, data files allow us to store information permanently, and to access and alter that information whenever necessary.

In C, an extensive set of library functions is available for creating and processing data files. Unlike other programming language, 'C' does not distinguish between sequential and direct access (random access) data files. There are, however, two different types of data files, called stream-oriented (or standard) data files, and system-oriented (or low-level) data files. Stream-oriented data files are generally easier to work with than system-oriented data files and are therefore more commonly used.

"Stream-oriented data files" can be divided into two categories

1. In the first category are data files comprising consecutive characters. These characters can be interpreted as individual data items or as components of strings or numbers.
2. The second category of stream-oriented data files, often referred to as unformatted data files, organizes data into blocks containing contiguous bytes of information.

"System-oriented data files" are more closely related to the computer's operating system than are stream-oriented data files. They are somewhat more complicated to work with, though their use may be more efficient for certain kinds of applications. A separate set of procedures, with accompanying library functions, is required to process system-oriented data files.

OPENING AND CLOSING A DATA FILE:

When working with a stream-oriented data file, the first step is to establish a buffer area, where information, is temporarily stored while being transferred between the computer's memory and the data file. This buffer area allows information to be read from or written to the data file more rapidly than would otherwise be possible. The buffer area is established by writing

```
FILE *ptvar;
```

Where FILE is a special structure type that establishes the buffer area, and ptvar is a pointer variable that indicates the beginning of the buffer area. The structure type FILE is defined within a system "include" file, typically "stdio.h".

The library function 'fopen' is used to open a file. This function is typically written as

```
ptvar=fopen(file_name, file_type);
```

file_type is otherwise called as "mode".

The "fopen" function returns a pointer to the beginning of the buffer area associated with the file. A NULL value is returned if the file cannot be opened as, for example, when an existing data file cannot be found.

Finally, a data file must be "closed" at the end of the program. This can be accomplished with the library function "fclose". The syntax is simply

```
fclose(ptvar);
```

FILE-TYPE SPECIFICATION:

File Type	Meaning
"r"	Open an existing file for reading only.
"w"	Open a new file for writing only. If a file with the specified 'file-name' currently exists, it will be destroyed and a new file created in it's place.
"a"	Open an existing file for appending. A new file will be created if the file with the specified 'file-name' does not exist.
	"r+" Open an existing file for both reading and writing.
"w+"	Open a new file for both reading and writing. If a file with the specified 'file-name' currently exists, it will be destroyed and a new file created in its place.
"a+"	Open an existing file for both reading and appending. A new file will be created if the file with the specified 'file-name' does not exist.

```
/*        EX-59: READ A LINE OF LOWER-CASE TEXT AND STORE ITS UPPER-CASE EQUIVALENT WITHIN A DATA FILE  */
#include<stdio.h>
main()
{
  FILE *fpt;              /* Define a pointer to pre-defined type FILE */
  char c;
  clrscr();
  fpt=fopen("f_ex1.dat","w");  /* Open a new data file for writing only  */
  do
   putc(toupper(c=getchar()),fpt);
  while(c!='\n');
  fclose(fpt);                          /* Close the data file  */
}
```

The program begins by defining the stream pointer 'fpt', indicating the beginning of the data file buffer area.

The loop continues as long as a new line character (\n) is not entered from the keyboard. Once a new line character is detected, the loop ceases and the data file is closed.

```
/*  EX-60: FILE CREATION(WRITE) PROGRAM */
#include<stdio.h>
main()
{
  FILE *fp;
  char ch;
  clrscr();
  fp=fopen("f_ex2.dat","w");
  if(fp==NULL)
  {
   printf("\n\tERROR: Cannot open the designated file");
   exit(1);                     /* abnormal termination  */
  }
  while(1)                      /* Infinite loop  */
  {
    ch=getchar();
    if(ch==EOF)                 /* EOF = ^Z  */
      break;
    else
      fputc(ch,fp);
  }
  fclose(fp);
}
```

Note: Input will terminate when you press ^Z in data file.

We can see the content of file f_wr2.dat from DOS Prompt using

type <file_name.dat>

The content of existing file "f_ex2.dat" can read by accessing a separate program, i.e., as follows.

```
/*        EX-61: FILE READ PROGRAM   */
#include<stdio.h>
main()
{
  FILE *fp;
  char ch;
  clrscr();
  fp=fopen("example.dat","r");
  if(fp==NULL)
  {
   printf("\nFile opening error");
   exit(2);
  }

  while(1)
  {
    ch = fgetc(fp);
    if(ch == EOF)
```

```
      break;
    else
      putchar(ch);
 }
 fclose(fp);
getch();
}
```

FILE COPY:

Until we have seen the program for write the data into file and read the data from file. By using following program, we can copy the content of source file into the target file in a single program.

```
/*        EX-62: FILE COPY   */
#include<stdio.h>
main()
{
 FILE *fs, *ft;
 char ch;
 clrscr();
 fs=fopen("example.dat","r");
 if(fs==NULL)
 {
   printf("\n File opening error");
   exit(0);
 }
 ft=fopen("temp.dat","w");
 if(ft==NULL)
 {
    printf("\n File creation error");
    exit(2);
 }

 while(1)
 {
   ch=fgetc(fs);
   if(ch==EOF)
      break;
    else
     fputc(ch,ft);
 }
 printf("\n Files copied");
 fclose(fs);
 fclose(ft);
 getch();

}
```