LECTURE NOTE -10

CONTROL STATEMENTS

In C, programs are executed sequentially in the order of which they appear. This condition does not hold true always. Sometimes a situation may arise where we need to execute a certain part of the program. Also it may happen that we may want to execute the same part more than once. Control statements enable us to specify the order in which the various instructions in the program are to be executed. They define how the control is transferred to other parts of the program. Control statements are classified in the following ways:

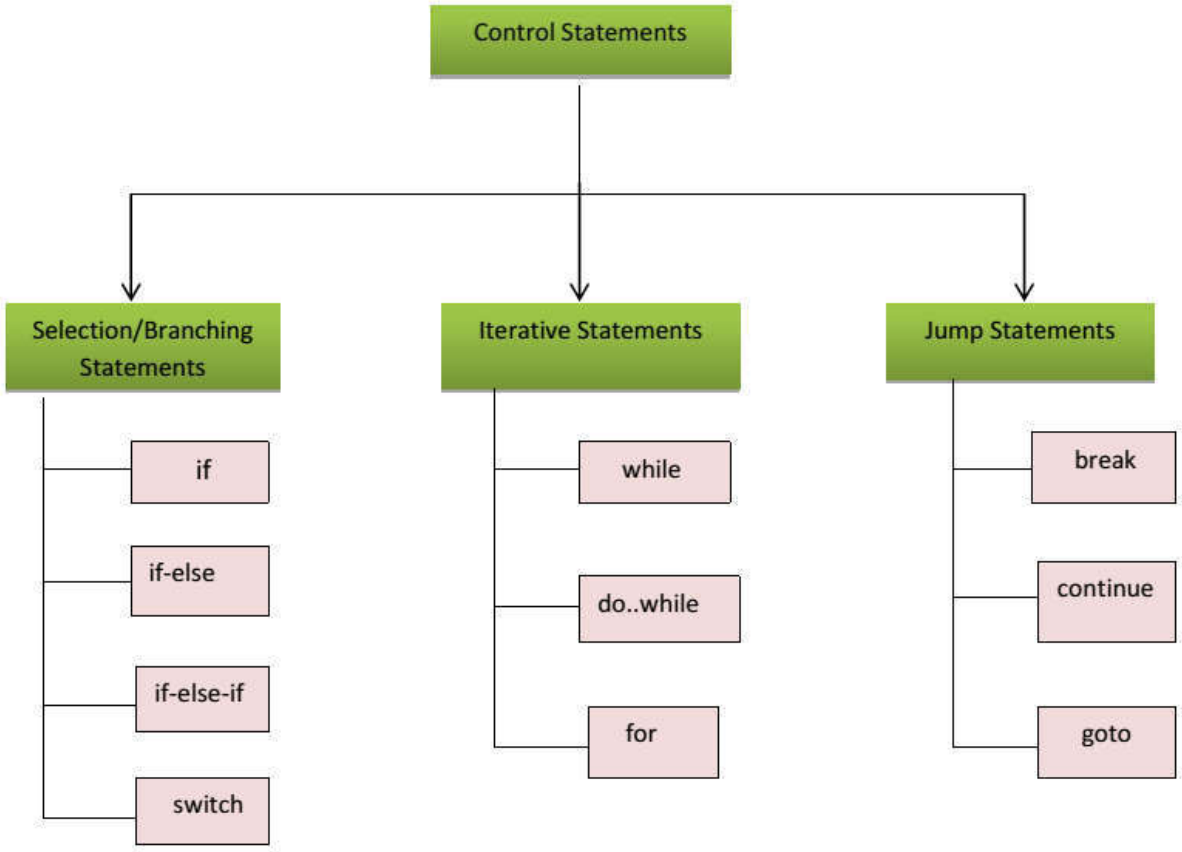


Fig: 1 Classification of control statements

SELECTION STATEMENTS

The selection statements are also known as *Branching* or *Decision Control Statements.*

Introduction to Decision Control Statements

Sometime we come across situations where we have to make a decision. E.g. *If the weather is sunny, I will go out & play, else I will be at home.* Here my course of action is governed by the kind of weather. If it's sunny, I can go out & play, else I have to stay indoors. I choose an option out of 2 alternate options. Likewise, we can find ourselves in situations where we have to select among several alternatives. We have decision control statements to implement this logic in computer programming.

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

if Statement

The keyword *if* tells the compiler that what follows is a decision control instruction. The *if* statement allows us to put some decision -making into our programs. The general form of the *if* statement is shown Fig 2:

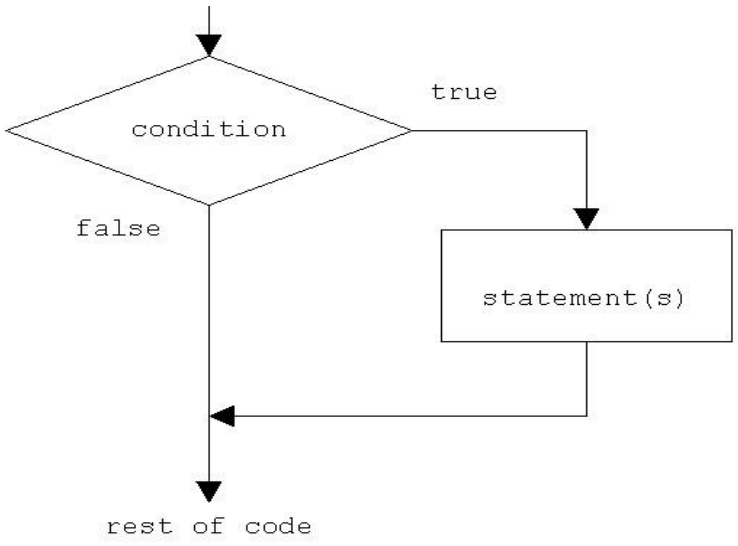


Fig 2: *if* statement construct

Syntax of if statement:

```
if (condition )
{
        Statement 1;
        ……..
        Statement n;
 }
        //Rest of the code
```

If the condition is true(nonzero), the statement will be executed. If the condition is false(0), the statement will not be executed. For example, suppose we are writing a billing program.

```
if (total_purchase >=1000)
    printf("You are gifted a pen drive.\n");
```

Multiple statements may be grouped by putting them inside curly braces {}. For example:

```
if (total_purchase>=1000)
{
        gift_count++;
        printf("You are gifted a pen drive.\n");
}
```

For readability, the statements enclosed in {} are usually indented. This allows the programmer to quickly tell which statements are to be conditionally executed. As we will see later, mistakes in indentation can result in programs that are misleading and hard to read.

Programs:

1. Write a program to print a message if negative no is entered.

```
#include<stdio.h>
int main()
{
        int no;
        printf("Enter a no : ");
```

```
        scanf("%d", &no);
        if(no<0)
         {
                printf("no entered is negative");
                no = -no;
         }
        printf("value of no is %d \n",no);
        return 0;
}
```

Output:

Enter a no: 6

value of no is 6

Output:

Enter a no: -2

value of no is 2

2. Write a program to perform division of 2 nos

```
#include<stdio.h>
int main()
{
        int a,b;
        float c;
     printf("Enter 2 nos : ");
    scanf("%d %d", &a, &b);
    if(b == 0)
         {
                printf("Division is not possible");
         }
        c = a/b;
        printf("quotient is %f \n",c);
        return 0;
}
```

Output:
Enter 2 nos: 6 2
quotient is 3

Output:
Enter 2 nos: 6 0
Division is not possible

if-else Statement

The *if* statement by itself will execute a single statement, or a group of statements, when the expression following *if* evaluates to true. By using *else* we execute another group of statements if the expression evaluates to false.

```
if (a > b)
        { z = a;
         printf("value of z is :%d",z);
        }
else
        { z = b;
         printf("value of z is :%d",z);
        }
```

The group of statements after the *if* is called an 'if block'. Similarly, the statements after the else form the 'else block'.

Programs:

3. Write a program to check whether the given no is even or odd

```
#include<stdio.h>
int main()
 {
         int n;
         printf("Enter an integer\n");
         scanf("%d",&n);
        if ( n%2 == 0 )
                printf("Even\n");
        else
```

```
                printf("Odd\n");
        return 0;
}
```

Output:
Enter an integer 3
Odd

Output:
Enter an integer 4
Even

4. Write a program to check whether a given year is leap year or not

```
#include <stdio.h>
int main()
{
  int year;
  printf("Enter a year to check if it is a leap year\n");
 scanf("%d", &year);
if ( (year%4 == 0) && (( year%100 != 0) || ( year%400 == 0 ))
        printf("%d is a leap year.\n", year);
 else
        printf("%d is not a leap year.\n", year);

 return 0;
}
```

Output:
Enter a year to check if it is a leap year 1996
1996 is a leap year

Output:
Enter a year to check if it is a leap year 2015
2015 is not a leap year

Nested *if-else*

An entire *if-else* construct can be written within either the body of the *if* statement or the body of an *else* statement. This is called 'nesting' of *if*s. This is shown in the following structure.

```
if (n > 0)
{
        if (a > b)
                z = a;
}
else
        z = b;
```

The second *if* construct is nested in the first *if* statement. If the condition in the first *if* statement is true, then the condition in the second *if* statement is checked. If it is false, then the *else* statement is executed.

Program:

5. Write a program to check for the relation between 2 nos

```
#include <stdio.h>
int main()
{
  int m=40,n=20;
  if ((m >0 ) && (n>0))
  {
          printf("nos are positive");
          if (m>n)
          {
                  printf("m is greater than n");
          }
          else
          {
                  printf("m is less than n");
          }
}
else
```

```
{
        printf("nos are negative");
 }
return 0;
}
```

Output

40 is greater than 20

else-if Statement:

This sequence of if statements is the most general way of writing a multi−way decision. The expressions are evaluated in order; if an expression is true, the statement associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group of them in braces.

```
If (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else
        statement
```

The last *else* part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing can be omitted, or it may be used for error checking to catch an “impossible” condition.

Program:

6. The above program can be used as an eg here.

```
#include <stdio.h>
int main()
{
```

```
 int m=40,n=20;
if (m>n)
    {
        printf("m is greater than n");
    }
 else if(m<n)
    {
        printf("m is less than n");
    }
 else
    {
        printf("m is equal to n");
    }
}
```

Output:

m is greater than n

switch case:

This structure helps to make a decision from the number of choices. The *switch* statement is a multi−way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly [3].

```
switch( integer expression)
{
    case constant 1 :
        do this;
    case constant 2 :
        do this ;
    case constant 3 :
        do this ;
    default :
        do this ;
}
```

The integer expression following the keyword switch is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an integer. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labelled *default* is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

Consider the following program:

```
main( )
{ int i = 2; switch
        ( i )
        {
        case 1:
          printf ( "I am in case 1 \n" ) ;
        case 2:
          printf ( "I am in case 2 \n" ) ;
        case 3:
          printf ( "I am in case 3 \n" ) ;
        default :
          printf ( "I am in default \n" ) ;  }
}
```

The output of this program would be:

```
I am in case 2
I am in case 3
I am in default
```

Here the program prints case 2 and 3 and the default case. If you want that only case 2 should get executed, it is up to you to get out of the switch then and there by using a *break* statement.

```
main( )
{
int i = 2 ;
switch ( i )
{
 case 1:
    printf ( "I am in case 1 \n" ) ;
```

```
    break ;
  case 2:
   printf ( "I am in case 2 \n" ) ;
    break ;
  case 3:
   printf ( "I am in case 3 \n" ) ;
    break ;
  default:
   printf ( "I am in default \n" ) ;
  }
}
```

The output of this program would be:
I am in case 2

Program

7. WAP to enter a grade & check its corresponding remarks.

```
#include <stdio.h>
int main ()
{
    char grade;
    printf("Enter the grade");
   scanf("%c", &grade);
  switch(grade)
  {
        case 'A' :printf("Outstanding!\n" );
                  break;
        case 'B' : printf("Excellent!\n" );
                   break;
        case 'C' :printf("Well done\n" );
                  break;
        case 'D' : printf("You passed\n" );
                  break;
        case 'F' : printf("Better try again\n" );
                  break;
        default :  printf("Invalid grade\n" );
  }
```

```
   printf("Your grade is  %c \n", grade );
return 0;
}
```

Output

```
Enter the grade
B
Excellent
Your grade is B
```

LECTURE NOTE -11

ITERATIVE STATEMENTS

while statement

The *while* statement is used when the program needs to perform repetitive tasks. The general form of a *while* statement is:

```
while ( condition)
        statement ;
```

The program will repeatedly execute the statement inside the *while* until the condition becomes false(0). (If the condition is initially false, the statement will not be executed.) Consider the following program:

```
main( )
{  int p, n, count;
float r, si;
count = 1;
while ( count <= 3 )
        {
                printf ( "\nEnter 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;
        }
}
```

Some outputs of this program:

```
Enter values of p, n and r 1000 5 13.5
Simple Interest = Rs. 675.000000
Enter values of p, n and r 2000 5 13.5
Simple Interest = Rs. 1350.000000
Enter values of p, n and r 3500 5 13.5
Simple Interest = Rs. 612.000000
```

The program executes all statements after the *while* 3 times. These statements form what is called the 'body' of the *while* loop. The parentheses after the *while* contain a condition. As long as this condition remains true all statements within the body of the *while* loop keep getting executed repeatedly.

Consider the following program;

```
/* This program checks whether a given number is a palindrome or not */

#include <stdio.h>
int main()
{
  int n, reverse = 0, temp;
  printf("Enter a number to check if it is a palindrome or not\n");
  scanf("%d",&n);
   temp = n;
  while( temp != 0 )
  {
       reverse = reverse * 10;
      reverse = reverse +temp%10;
     temp = temp/10;
  }
  if ( n == reverse )
    printf("%d is a palindrome number.\n", n);
else
    printf("%d is not a palindrome number.\n", n);

  return 0;
}
```

Output:

```
Enter a number to check if it is a palindrome or not
12321
12321 is a palindrome

Enter a number to check if it is a palindrome or not
12000
12000 is not a palindrome
```

do-while Loop

The body of the *do-while* executes at least once. The *do-while* structure is similar to the *while* loop except the relational test occurs at the bottom (rather than top) of the loop. This ensures that the body of the loop executes at least once. The *do-while* tests for a positive relational test; that is, as long as the test is True, the body of the loop continues to execute.
The format of the do-while is

do
{ block of one or more C statements; }
while (test expression)

The test expression must be enclosed within parentheses, just as it does with a while statement.

Consider the following program

```
// C program to add all the numbers entered by a user until user enters 0.

#include <stdio.h>
int main()
{   int sum=0,num;
  do          /* Codes inside the body of do...while loops are at least executed once. */
  {
      printf("Enter a number\n");
     scanf("%d",&num);
      sum+=num;
  }
  while(num!=0);
  printf("sum=%d",sum);
return 0;
}
```

Output:

```
Enter a number
3
Enter a number
-2
Enter a number
```

0
sum=1

Consider the following program:

```
#include <stdio.h>
main()
{
    int i = 10;
   do
  {
      printf("Hello %d\n", i );
      i = i -1;
  }while ( i > 0 );
}
```

Output

```
Hello 10
Hello 9
Hello 8
Hello 7
Hello 6
Hello 5
Hello 4
Hello 3
Hello 2
Hello 1
```

Program

8. Program to count the no of digits in a number

```
#include <stdio.h>
int main()
{
  int n,count=0;
  printf("Enter an integer: ");
  scanf("%d", &n);
```

```
do
  {
        n/=10;          /* n=n/10 */
        count++;
 } while(n!=0);

 printf("Number of digits: %d",count);
}
```

Output

Enter an integer: 34523
Number of digits: 5

for Loop

The *for* is the most popular looping instruction. The general form of *for* statement is as under:

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

The *for* allows us to specify three things about a loop in a single line:

(a) Setting a loop counter to an initial value.
(b) Testing the loop counter to determine whether its value has reached the number of repetitions desired.
(c) Updating the value of loop counter either increment or decrement.

Consider the following program

```
int main(void)
{
int num;
 printf("    n   n cubed\n");
for (num = 1; num <= 6; num++)
```

```
        printf("%5d %5d\n", num, num*num*num);
return 0;
}
```

The program prints the integers 1 through 6 and their cubes.

```
n   n cubed
1    1
2    8
3    27
4    64
5    125
6    216
```

The first line of the *for* loop tells us immediately all the information about the loop parameters: the starting value of num, the final value of num, and the amount that num increases on each looping [5].

Grammatically, the three components of a *for* loop are expressions. Any of the three parts can be omitted, although the semicolons must remain.

Consider the following program:

```
main( )
{
        int i ;
        for ( i = 1 ; i <= 10 ; )
        {
         printf ( "%d\n", i ) ;
         i = i + 1 ;
        }
}
```

Here, the increment is done within the body of the *for* loop and not in the *for* statement. Note that in spite of this the semicolon after the condition is necessary.

Programs:

9. Program to print the sum of 1st N natural numbers.

```
#include <stdio.h>
int main()
```

```
{
 int n,i,sum=0;
 printf("Enter the limit: ");
scanf("%d", &n);
for(i=1;i<=n;i++)
 {
            sum = sum +i;
 }
printf("Sum of N natural numbers is: %d",sum);
}
```

Output

Enter the limit: 5
Sum of N natural numbers is 15.

10. Program to find the reverse of a number

```
#include<stdio.h>
int main()
{
int num,r,reverse=0;
printf("Enter any number: ");
scanf("%d",&num);
for(;num!=0;num=num/10)
{
        r=num%10;
        reverse=reverse*10+r;
}
 printf("Reversed of number: %d",reverse);
return 0;
 }
```

Output:

Enter any number: 123
Reversed of number: 321

NESTING OF LOOPS

C programming language allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

The syntax for a nested for loop statement in C is as follows:

```
for ( init; condition; increment )
{

 for ( init; condition; increment)

   {
            statement(s);
   }
   statement(s);
}
```

The syntax for a nested while loop statement in C programming language is as follows:

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

The syntax for a nested do...while loop statement in C programming language is as follows:

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

}while(condition);

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Programs:

11. program using a nested for loop to find the prime numbers from 2 to 20:

```
#include <stdio.h>
int main ()
{
  /* local variable definition */
int i, j;
  for(i=2; i<20; i++)
 {
    for(j=2; j <= (i/j); j++)
      if(!(i%j))
          break; // if factor found, not prime
     if(j > (i/j)) printf("%d is prime\n", i);
  }

  return 0;
}
```

Output

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

12.
```
    *
   ***
  *****
 *******
*********
```

```
#include <stdio.h>
int main()
{
  int row, c, n,I, temp;
 printf("Enter the number of rows in pyramid of stars you wish to see ");
 scanf("%d",&n);
 temp = n;
 for ( row = 1 ; row <= n ; row++ )
 {
    for ( i= 1 ; i < temp ; i++ )
    {
          printf(" ");
          temp--;
          for ( c = 1 ; c <= 2*row - 1 ; c++ )
          {
                printf("*");
                printf("\n");
          }
    }
 }
  return 0;
}
```

13. Program to print series from 10 to 1 using nested loops.

```
#include<stdio.h>
void main ()
{
          int a;
          a=10;
        for (k=1;k=10;k++)
       {
          while (a>=1)
```

```
        {
                printf ("%d",a);
                a--;
        }
        printf("\n");
        a= 10;
        }
}
```

Output:

```
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
```

LECTURE NOTE-12

JUMP STATEMENTS

The break Statement

The *break* statement provides an early exit from *for*, *while*, and *do*, just as from *switch*. A *break* causes the innermost enclosing loop or switch to be exited immediately. When *break* is encountered inside any loop, control automatically passes to the first statement after the loop.

Consider the following example;

```
main( )
{
        int i = 1 , j = 1 ;
        while ( i++ <= 100 )
        {
                while ( j++ <= 200 )
                {
                if ( j == 150 )
                  break ;
                 else
                  printf ( "%d %d\n", i, j );
                }
        }
}
```

In this program when j equals 150, break takes the control outside the inner while only, since it is placed inside the inner *while*.

The *continue* Statement

The *continue* statement is related to *break*, but less often used; it causes the next iteration of the enclosing *for*, *while*, or *do* loop to begin. In the *while* and *do*, this means that the test part is executed immediately; in the *for*, control passes to the increment step. The *continue* statement applies only to loops, not to switch.

Consider the following program:

```
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...

1 2
2 1

Note that when the value of I equals that of j, the *continue* statement takes the control to the *for* loop (inner) by passing rest of the statements pending execution in the *for* loop (inner).

The *goto* statement

Kernighan and Ritchie refer to the *goto* statement as "infinitely abusable" and suggest that it "be used sparingly, if at all.
The *goto* statement causes your program to jump to a different location, rather than execute the next statement in sequence. The format of the *goto* statement is;

goto statement label;

Consider the following program fragment

```
if (size > 12)
        goto a;
goto b;

a: cost = cost * 1.05;
flag = 2;
```

*b: bill = cost * flag;*

Here, if the *if* conditions satisfies the program jumps to block labelled as *a:* if not then it jumps to block labelled as *b:*.

Exercise questions:

1. WAP to input the 3 sides of a triangle & print its corresponding type.
2. WAP to input the name of salesman & total sales made by him. Calculate & print the commission earned.

TOTAL SALES	RATE OF COMMISSION
1-1000	3 %
1001-4000	8 %
6001-6000	12 %
6001 and above	15 %

3. WAP to calculate the wages of a labor.

TIME	WAGE
First 10 hrs.	Rs 60
Next 6 hrs.	Rs 15
Next 4 hrs.	Rs 18
Above 10 hrs.	Rs 25

4. WAP to calculate the area of a triangle, circle, square or rectangle based on the user's choice.
5. WAP that will print various formulae & do calculations:
   i. Vol of a cube
   ii. Vol of a cuboid
   iii. Vol of a cyclinder
   iv. Vol of sphere
6. WAP to print the following series
   i. S = 1 + 1/2 + 1/3 ……..1/10
   ii. P= (1*2) + (2 *3) + (3* 4)+…….(8 *9) +(9 *10)
   iii. Q= ½ + ¾ +5/6 +….13/14
   iv. S = 2/5 + 5/9 + 8/13….n
   v. $S = x + x^2 + x^3 + x^4 ..... + x^9 + x^{10}$
   vi. $P= x + x^3/3 + x^5/5 + x^7/7$……n terms
   vii. S= (13 *1) + (12 * 2)……(1 *13)
   viii. $S = 1 + 1/(2^2) + 1/(3^3) + 1/(4^4) + 1/(5^5)$

ix. $S = 1/1! + 1/2! + 1/3! \ldots\ldots\ldots\ldots\ldots +1/n!$

x. $S = 1 + 1/3! + 1/5! + \ldots\ldots\ldots n$ terms

xi. $S = 1 + (1+2) + (1+2+3) + (1+2+3+4) \ldots\ldots\ldots\ldots\ldots + (1+2+3 \ldots\ldots 20)$

xii. $S = x + x^2/2! + x^3/3! + x^4/4! \ldots\ldots + x^{10}/10!$

xiii. $P = x/2! + x^2/3! + \ldots\ldots x^9/10!$

xiv. $S = 1 - 2 + 3 - 4 \ldots\ldots\ldots + 9 - 10$

xv. $S = 1 - 2^2 + 3^2 - 4^2 \ldots\ldots\ldots + 9^2 - 10^2$

xvi. $S = 1/(1 + 2) + 3/(3 + 5) \ldots\ldots 15/(15 + 16)$

xvii. $S = 1 + x^2/2! - x^4/4! + x^6/6! \ldots\ldots n$

xviii. $S = 1 + (1 + 2) + (1+2+3) \ldots\ldots\ldots (1+2+3+4 \ldots\ldots 20)$

xix. $S = 1 + x + x^2/2 + x^3/3 \ldots\ldots\ldots + x^n/n$

xx. $S = 1 * 3/ 2 * 4 * 5 + 2 * 4 / 3 * 5 * 6 + 3 * 5/ 4 * 6 * 7 \ldots\ldots\ldots n * (n+2)/ (n+1) * (n+3) * (n+4)$

7. WAP to input a no & print its corresponding table.
8. WAP to print the table from 1 to 10 till 10 terms.
9. WAP to input a no & print its factorial.
10. WAP to input a no & check whether it is prime or not.
11. WAP to input a no & print all the prime nos upto it.
12. WAP to input a no & print if the no is perfect or not.
13. WAP to find the HCF of 2 nos.
14. WAP to print the Pythagoras triplets within 100. (A Pythagorean triplet consists of three positive integers a, b, and c, such that $a^2 + b^2 = c^2$).
15. WAP to input a no & check whether its automorphic or not. (An automorphic number is a number whose square "ends" in the same digits as the number itself. For example, $5^2 = 2\mathbf{5}$, $6^2 = 3\mathbf{6}$, $76^2 = 5776$, and $890625^2 = 793212\mathbf{890625}$, so 5, 6, 76 and 890625 are all automorphic numbers).
16. WAP to convert a given no of days into years, weeks & days.
17. WAP to input a no & check whether it's an Armstrong no or not. (An Armstrong no is an integer such that the sum of the cubes of its digits is equal to the number itself. For example, 371 is an Armstrong number since $3^3 + 7^3 + 1^3 = 371$).
18. A cricket kit supplier in Jalandhar sells bats, wickets & balls. WAP to generate sales bill. Input form the console the date of purchase, name of the buyer, price of each item & quantity of each item. Calculate the total sale amount & add 17.5 % sales tax if the total sales amount >300000 & add 12.5 % if the total sales amount is >150000 & 7 % otherwise. Display the total sales amount, the sales tax & the grand total.
19. WAP to check whether a given number is magic number or not.

    (What is a magic number? Example: 1729

    - Find the sum of digits of the given number.(1 + 7 + 2 + 9 => 19)
    - Reverse of digit sum output. Reverse of 19 is 91

- Find the product of digit sum and the reverse of digit sum.(19 X 91 = 1729)
- If the product value and the given input are same, then the given number is a magic number.(19 X 91 <=> 1729)

20. Write a C program to calculate generic root of the given number. (To find the generic root of a no we first find the sum of digits of the no until we get a single digit output. That resultant no is called the generic no. Eg: 456791: 4+5+6+7+9+1=32. 3 +2 =5. So, 5 becomes the generic root of the given no)

MODULE 2

LECTURE NOTE-13

FUNCTION

MONOLITHIC VS MODULAR PROGRAMMING:

1. Monolithic Programming indicates the program which contains a single function for the large program.
2. Modular programming help the programmer to divide the whole program into different modules and each module is separately developed and tested. Then the linker will link all these modules to form the complete program.
3. On the other hand monolithic programming will not divide the program and it is a single thread of execution. When the program size increases it leads inconvenience and difficult to maintain.

Disadvantages of monolithic programming: 1. Difficult to check error on large programs. 2. Difficult to maintain. 3. Code can be specific to a particular problem. i.e. it can not be reused.

Advantage of modular programming: 1. Modular program are easier to code and debug. 2. Reduces the programming size. 3. Code can be reused in other programs. 4. Problem can be isolated to specific module so easier to find the error and correct it.

FUNCTION:

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

Function Declaration OR Function Prototype:

1. It is also known as function prototype .
2. It inform the computer about the three things
    a) Name of the function
    b) Number and type of arguments received by the function.
    c) Type of value return by the function

Syntax :

return_type function_name (type1 arg1 , type2 arg2);

OR

return_type function_name (type1 type2);

3. Calling function need information about called function .If called function is place before calling function then the declaration is not needed.

Function Definition:

1. It consists of code description and code of a function .

   It consists of two parts

   a) Function header

   b) Function coding

   Function definition tells what are the I/O function and what is going to do.

   Syntax:

   return_type function_name (type1 arg1 , type2 arg2)

   {
   local variable;

   statements ;

   return (expression);

   }

2. Function definition can be placed any where in the program but generally placed after the main function .

3. Local variable declared inside the function is local to that function. It cannot be used anywhere in the program and its existence is only within the function.

4. Function definition cannot be nested.

5. Return type denote the type of value that function will return and return type is optional if omitted it is assumed to be integer by default.

USER DEFINE FUNCTIONS VS STANDARD FUNCTION:

User Define Fuction:

A function that is declare, calling and define by the user is called user define function. Every user define function has three parts as:

1. Prototype or Declaration
2. Calling
3. Definition

Standard Function:

The **C standard library** is a standardized collection of header files and library routines used to implement common operations, such as input/output and character string handling. Unlike other languages (such as COBOL, FORTRAN, and PL/I) C does not include built in keywords for these tasks, so nearly all C programs rely on the standard library to function.

LECTURE NOTE 14

FUNCTION CATAGORIES

There are four main categories of the functions these are as follows:

1. Function with no arguments and no return values.
2. Function with no arguments and a return value.
3. Function with arguments and no return values.
4. Function with arguments and return values.

Function with no arguments and no return values:

syntax:

```
void funct (void);
main ( )
{
funct ( );
}
void funct ( void );
{
}
```

NOTE: There is no communication between calling and called function. Functions are executed independently, they read data & print result in same block.

Example:

```
void  link (void) ;
int  main ()
{
link ();
}
void link ( void );
{
printf (" link the file ")
}
```

Function with no arguments and a return value: This type of functions has no arguments but a return value

example:

```
 int msg (void) ;
int main ( )
{
int s = msg (  );
printf( "summation = %d" , s);
}
 int msg ( void )
{
int a, b, sum ;
sum = a+b ;
return (sum) ;
}
```

NOTE: Here called function is independent, it read the value from the keyboard, initialize and return a value .Both calling and called function are partly communicated with each other.

Function with arguments and no return values:

Here functions have arguments so, calling function send data to called function but called function does no return value. such functions are partly dependent on calling function and result obtained is utilized by called function .

Example:

```
 void  msg ( int , int );

int main ( )

{

int a,b;
```

```
a= 2; b=3;

msg( a, b);

}

void msg ( int a , int b)

{

int s ;

sum = a+b;

printf ("sum = %d" , s ) ;

}
```

Function with arguments and return value:

Here calling function of arguments that passed to the called function and called function return value to calling function.

example:

```
 int  msg ( int , int ) ;

int main ( )

{

int a, b;

a= 2; b=3;
```

```
int s = msg (a, b);

printf (“sum = %d” , s ) ;

}

int msg( int a , int b)

{

int sum ;

sum =a+b ;
return (sum);

}
```

LECTURE NOTE 15

ACTUAL ARGUMENTS AND FORMAL ARGUMENTS

Actual Arguments:

1. Arguments which are mentioned in the function in the function call are known as calling function.
2. These are the values which are actual arguments called to the function.

It can be written as constant , function expression on any function call which return a value .

ex: funct (6,9) , funct (a,b)

Formal Arguments:

1. Arguments which are mentioned in function definition are called dummy or formal argument.
2. These arguments are used to just hold the value that is sent by calling function.
3. Formal arguments are like other local variables of the function which are created when function call starts and destroyed when end function.

Basic difference between formal and local argument are:

a) Formal arguments are declared within the ( ) where as local variables are declared at beginning.

b) Formal arguments are automatically initialized when a value of actual argument is passed.

c) Where other local variables are assigned variable through the statement inside the function body.

Note: Order, number and type of actual argument in the function call should be matched with the order , number and type of formal arguments in the function definition .

PARAMETER PASSING TECHNIQUES:

1. call by value
2. call by reference

Call by value:

Here value of actual arguments is passed to the formal arguments and operation is done in the formal argument.

Since formal arguments are photo copy of actual argument, any change of the formal arguments does not affect the actual arguments

Changes made to the formal argument t are local to block of called function, so when control back to calling function changes made vanish.

Example:

```
void swap (int a , int b)       /* called function */
    {
    int t;
    t = a;
    a=b;
    b = t;
  }
main()
 {
    int k = 50,m= 25;
    swap( k, m) ;    / * calling function */ print
    (k, m);      / * calling function */
    }
```

Output:

50, 25

Explanation:

int k= 50, m=25 ;

Means first two memory space are created k and m , store the values 50 and 25 respectively.

swap (k,m);

When this function is calling the control goes to the called function.

void swap (int a , int b),
k and m values are assigned to the ‘a’ and ‘b’.
then a= 50 and b= 25 ,

After that control enters into the function a temporary memory space ‘t’ is created when int t is executed.

t=a; Means the value of a is assigned to the t , then t= 50.

a=b; Here value of b is assigned to the a , then a= 25;

b=t; Again t value is assigned to the b , then b= 50;

after this control again enters into the main function and execute the print function print (k,m). it returns the value 50 , 25.

NOTE:

Whatever change made in called function not affects the values in calling function.

Call by reference:

Here instead of passing value address or reference are passed. Function operators or address rather than values .Here formal arguments are the pointers to the actual arguments.

Example:

```
#include<stdio.h>
void add(int *n);
int main()
    {
    int num=2;
    printf("\n The value of num before calling the function=%d", num);
    add(&num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
    }
    void add(int *n)
    {
```

```
   *n=*n+10;
printf("\n The value of num in the called function = %d", n);
}
```

Output:

The value of num before calling the function=2

The value of num in the called function=20 The

value of num after calling the function=20

NOTE:

In call by address mechanism whatever change made in called function affect the values in calling function.

EXAMPLES:

1: Write a function to return larger number between two numbers:

```
int fun(int p, int q)
{
int large;
if(p>q)
 {
large = p;
 }
 else
  {
large = q;
}
return large;
}
```

2: Write a program using function to find factorial of a number.

```
#include <stdio.h> int
factorial (int n)
{
   int i, p;
   p = 1;
 for (i=n; i>1; i=i-1)
```

```
    {
            p = p * i;
    }

    return (p);
}

void main()
{
    int a, result;
    printf ("Enter an integer number: ");
    scanf ("%d", &a);
    result = factorial (a);
    printf ("The factorial of %d is %d.\n", a, result);
}
```

EXERCISE:

1. What do you mean by function?
2. Why function is used in a program?
3. What do you mean by call by value and call by address?
4. What is the difference between actual arguments and formal arguments?
5. How many types of functions are available in C?
6. How many arguments can be used in a function?
7. Add two numbers using
   a) with argument with return type
   b) with argument without return type
   c) without argument without return type
   d) without argument with return type
8. Write a program using function to calculate the factorial of a number entered through the keyboard.
9. Write a function power(n,m), to calculate the value of n raised to m.
10. A year is entered by the user through keyboard. Write a function to determine whether the year is a leap year or not.
11. Write a function that inputs a number and prints the multiplication table of that number.
12. Write a program to obtain prime factors of a number. For Example: prime factors of 24 are 2,2,2 and 3 whereas prime factors of 35 are 5 and 7.
13. Write a function which receives a float and a int from main(), finds the product of these two and returns the product which is printed through main().

14. Write a function that receives % integers and returns the sum, average and standard deviation of these numbers. Call this function from main() and print the result in main().
15. Write a function that receives marks obtained by a student in 3 subjects and returns the average and percentage of these marks. Call this function from main() and print the result in main().
16. Write function to calculate the sum of digits of a number entered by the user.
17. Write a program using function to calculate binary equivalent of a number.
18. Write a C function to evaluate the series
    $\sin(x) = x - (x^3/3!) + (x^5/5!) - (x^7/7!) +\ldots\ldots\ldots$ to five significant digit.
19. If three sides of a triangle are p, q and r respectively, then area of triangle is given by
    area $= (S(S-p)(S-q)(S-r))^{1/2}$, where $S = (p+q+r)/2$.
    Write a program using function to find the area of a triangle using the above formula.
20. Write a function to find GCD and LCM of two numbers.
21. Write a function that takes a number as input and returns product of digits of that number.
22. Write a single function to print both amicable pairs and perfect numbers.(Two different numbers are said to be amicable if the sum of proper divisors of each is equal to the other. 284 and 220 are amicable numbers.)
23. Write a function to find whether a character is alphanumeric.

LECTURE NOTE 16

RECURSION

Recursion is a process in which a problem is defined in terms of itself. In 'C' it is possible to call a function from itself. Functions that call themselves are known as **recursive** functions, i.e. a statement within the body of a function calls the same function. Recursion is often termed as 'Circular Definition'. Thus recursion is the process of defining something in terms of itself. To implement recursion technique in programming, a function should be capable of calling itself.

Example:

```
void main()
{
……..……..……..        /* Some statements*/
fun1();

……..……..……..        /* Some statements */
} void
fun1()

{
……..……..……..         /* Some statements */ fun1();
/*RECURSIVE CALL*/

……..……..……..        /* Some statements */
}
```

Here the function **fun1()** is calling itself inside its own function body, so fun1() is a recursive function. When main() calls fun1(), the code of fun1() will be executed and since there is a call to fun1() insidefun1(), again fun1() will be executed. It looks like the above program will run up to infinite times but generally a terminating condition is written inside the recursive functions which end this recursion. The following program (which is used to print all the numbers starting from the given number to 1 with successive decrement by 1) illustrates this:

```
void main()
{
 int a;
 printf("Enter a number");
 scanf("%d",&a);
 fun2(a);
}
int fun2(int b)
{
 printf("%d",b);
 b--;
 if(b>=1)    /* Termination condition i.e. b is less than 1*/
{
 fun2(b);
}
}
```

How to write a Recursive Function?

Before writing a recursive function for a problem its necessary to define the solution of the problem in terms of a similar type of a smaller problem.

Two main steps in writing recursive function are as follows:

(i). Identify the Non-Recursive part(base case) of the problem and its solution(Part of the problem whose solution can be achieved without recursion).
(ii). Identify the Recursive part(general case) of the problem(Part of the problem where recursive call will be made).

Identification of Non-Recursive part of the problem is mandatory because without it the function will keep on calling itself resulting in infinite recursion.

How control flows in successive recursive calls?

Flow of control in successive recursive calls can be demonstrated in following example:

Consider the following program which uses recursive function to compute the factorial of a number.

```
void main()
{
int n,f;
printf("Enter a number");
scanf("%d",&n);
f=fact(a);
printf("Factorial of %d is %d",n,f);
}
int fact(int m)
{
int a;
if (m==1)
  return (1);
else
  a=m*fact(m-1);
return (a);
}
```

In the above program if the value entered by the user is 1 i.e.**n**=1, then the value of **n** is copied into **m**. Since the value of **m** is 1 the condition 'if(m==1)' is satisfied and hence 1 is returned through return statement i.e. factorial of 1 is 1.
When the number entered is 2 i.e. n=2, the value of n is copied into m. Then in function fact(), the condition 'if(m==1)' fails so we encounter the statement a=m*fact(m-1); and here we meet

recursion. Since the value of **m** is 2 the expression (m*fact(m-1)) is evaluated to (2*fact(1)) and the result is stored in **a**(factorial of a). Since value returned by fact(1) is 1 so the above expression reduced to (2*1) or simply 2. Thus the expression m*fact(m-1) is evaluated to 2 and stored in **a** and returned to main(). Where it will print 'Factorial of 2 is 2'.

In the above program if **n**=4 then main() will call fact(4) and fact(4) will send back the computed value i.e. **f** to main(). But before sending back to main() fact(4) will call fact(4-1) i.e. fact(3) and wait for a value to be returned by fact(3). Similarly fact(3) will call fact(2) and so on. This series of calls continues until **m** becomes 1 and fact(1) is called, which returns a value which is so called as termination condition. So we can now say what happened for **n**=4 is as follows

fact(4) returns (4*fact(3))

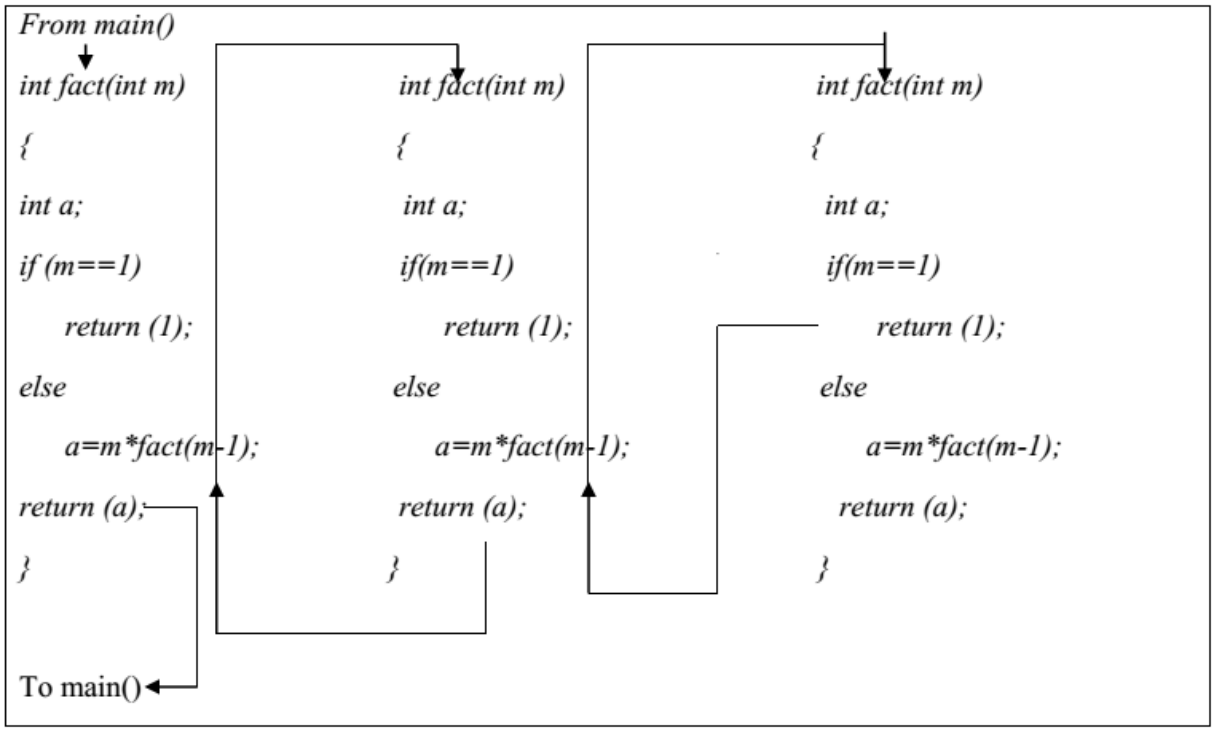
fact(3) returns (3*fact(2))

fact(2) returns (2*fact(1))

fact(1) returns (1)

So for **n**=4, the factorial of 4 is evaluated to 4*3*2*1=24.

For **n**=3, the control flow of the program is as follows:



Winding and Unwinding phase

All recursive functions work in two phases- winding phase and unwinding phase.

Winding phase starts when the recursive function is called for the first time, and ends when the termination condition becomes true in a call i.e. no more recursive call is required. In this phase a function calls itself and no return statements are executed.

After winding phase unwinding phase starts and all the recursive function calls start returning in reverse order till the first instance of function returns. In this phase the control returns through each instance of the function.

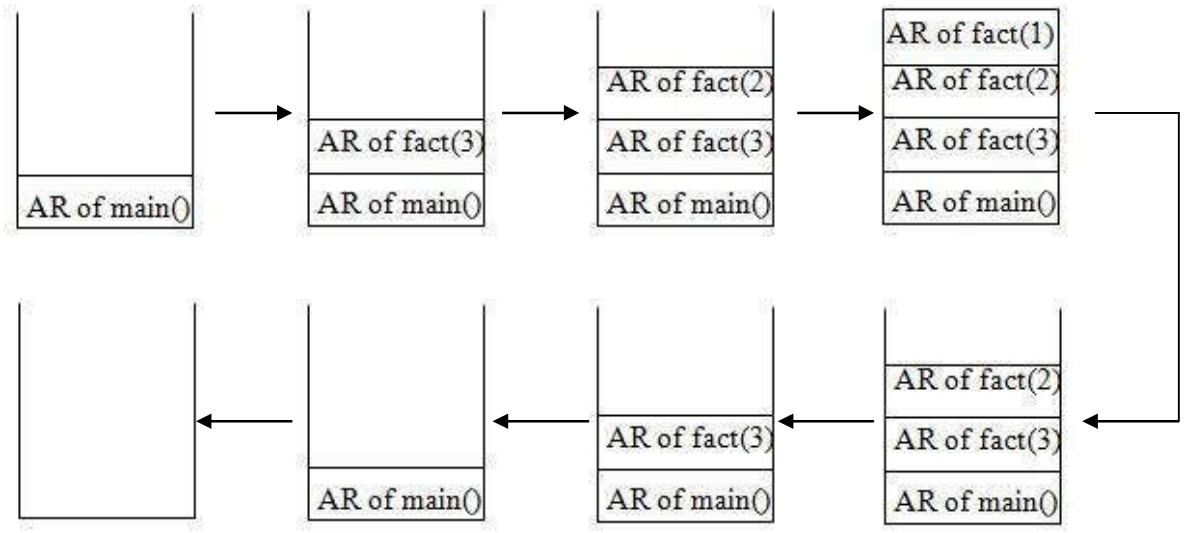
Implementation of Recursion

We came to know that recursive calls execute like normal function calls, so there is no extra technique to implement recursion. All function calls(Whether Recursive or Non-Recursive) are implemented through run time stack. Stack is a Last In First Out(LIFO) data structure. This means that the last item to be stored on the stack(PUSH Operation) is the first one which will be deleted(POP Operation) from the stack. Stack stores Activation Record(AR) of function during run time. Here we will take the example of function fact() in the previous recursive program to find factorial of a number.

Suppose fact() is called from main() with argument 3 i.e.

*fact(3); /*From main()*/*

Now will see how the run time stack changes during the evaluation of factorial of 3.



The following steps will reveal how the above stack contents were expressed:

First main() is called, so PUSH AR of main() into the stack. Then main() calls fact(3) so PUSH AR of fact(3). Now fact(3) calls fact(2) so PUSH AR of fact(2) into the stack. Likewise PUSH AR of fact(1). After the above when fact(1) is completed, POP AR of fact(1), Similarly after completion of a specific function POP its corresponding AR. So when main() is completed POP AR of main(). Now stack is empty.

In the winding phase the stack content increases as new Activation Records(AR) are created and pushed into the stack for each invocation of the function. In the unwinding phase the Activation Records are popped from the stack in LIFO order till the original call returns.

Examples of Recursion

Q1. Write a program using recursion to find the summation of numbers from 1 to n.

Ans: We can say 'sum of numbers from 1 to n can be represented as sum of numbers from 1 to n-1 plus n' i.e.

Sum of numbers from 1 to n = n + Sum of numbers from 1 to n-1

= n + n-1 + Sum of numbers from 1 to n-2

= n+ n-1 + n-2 + ... +1

The program which implements the above logic is as follows:

```
#include<stdio.h>

void main()

{

int n,s;

 printf("Enter a number");

 scanf("%d",&n);

 s=sum(n);

 printf("Sum of numbers from 1 to %d is %d",n,s);

}
```

```
 int sum(int m) int r;

if(m==1)

        return (1);

else

        r=m+sum(m-1);/*Recursive Call*/

        return r;

}
```

Output:

Enter a number 5
15

Q2. Write a program using recursion to find power of a number i.e. n^m.

Ans: We can write,

$n_m = n*n_{m-1}$

$=n*n*n^{m-2}$

$=n*n*n*$……m times $*n^{m-m}$

The program which implements the above logic is as follows:

```
#include<stdio.h>

int power(int,int);

 void main()

{

        int n,m,k;

        printf("Enter the value of n and m");

        scanf("%d%d",&n,&m);
```

```
        k=power(n,m);
        printf("The value of n^m for n=%d and m=%d is %d",n,m,k);
}
int power(int x, int y)
{
        if(y==0)
        {
                return 1;
        }
        else
        {
                return(x*power(x,y-1));
        }
}
```

Output:

Enter the value of n and m

3

5

The value of n^m for n=3 and m=5 is 243

Q3.Write a program to find GCD (Greatest Common Divisor) of two numbers.

Ans: The GCD or HCF (Highest Common Factor) of two integers is the greatest integer that divides both the integers with remainder equals to zero. This can be illustrated by Euclid's remainder Algorithm which states that GCD of two numbers say x and y i.e.

GCD(x, y) = **x** if y is 0

= **GCD(y, x%y)** otherwise

The program which implements the previous logic is as follows:

```
#include<stdio.h>
int GCD(int,int);
void main()
{
 int a,b,gcd;
printf("Enter two numbers");
scanf("%d%d",&a,&b);
gcd=GCD(a,b);
printf("GCD of %d and %d is %d",a,b,gcd);
}
int GCD(int x, int y)
{
if(y==0)
return x;
 else
 return GCD(y,x%y);
}
```

Output:

Enter two numbers 21

35

GCD of 21 and 35 is 7

Q4:Write a program to print Fibonacci Series upto a given number of terms.

Ans: Fibonacci series is a sequence of integers in which the first two integers are 1 and from third integer onwards each integer is the sum of previous two integers of the sequence i.e.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …….....

The program which implements the above logic is as follows:

```
#include<stdio.h>
int Fibonacci(int);
void main()
{
 int term,i;
printf("Enter the number of terms of Fibonacci Series which is going to be printed");
scanf("%d",&term);
 for(i=0;i<term;i++)
{
printf("%d",Fibonacci(i));
}
}
int Fibonacci(int x)
{
if(x==0 || x==1)
return 1;
else
return (Fibonacci(x-1) + Fibonacci(x-2));
}
```

Output:

Enter the number of terms of Fibonacci Series which is going to be printed 6

1 1 2 3 5 8 13

LECTURE NOTE 17

RECURSION VERSES ITERATION

Every repetitive problem can be implemented recursively or iteratively

Recursion should be used when the problem is recursive in nature. Iteration should be used when the problem is not inherently recursive

Recursive solutions incur more execution overhead than their iterative counterparts, but its advantage is that recursive code is very simple.

Recursive version of a problem is slower than iterative version since it requires PUSH and POP operations.

In both recursion and iteration, the same block of code is executed repeatedly, but in recursion repetition occurs when a function calls itself and in iteration repetition occurs when the block of code is finished or a continue statement is encountered.

For complex problems iterative algorithms are difficult to implement but it is easier to solve recursively. Recursion can be removed by using iterative version.

Tail Recursion

A recursive call is said to be tail recursive if the corresponding recursive call is the last statement to be executed inside the function.

Example: Look at the following recursive function

```
void show(int a)
{
if(a==1)
 return;
printf("%d",a);
show(a-1);
 }
```

In the above example since the recursive call is the last statement in the function so the above recursive call is Tail recursive call.

In non void functions(return type of the function is other than void) , if the recursive call appears in return statement and the call is not a part of an expression then the call is said to be Tail recursive, otherwise Non Tail recursive. Now look at the following example

```
int hcf(int p, int q)
{
if(q==0)
return p;
else

return(hcf(q,p%q));    /*Tail recursive call*/
}
int factorial(int a)
{
if(a==0)
return 1;
else

 return(a*factorial(a-1));       /*Not a Tail recursive call*/
}
```

In the above example in hcf() the recursive call is not a part of expression (i.e. the call is the expression in the return statement) in the call so the recursive call is Tail recursive. But in factorial() the recursive call is part of expression in the return statement(a*recursive call) , so the recursive call in factorial() is not a Tail excursive call.

A function is said to be Tail recursive if all the recursive calls in the function are tail recursive.

Tail recursive functions are easy to write using loops,

In tail recursive functions, the last work that a function does is a recursive call, so no operation is left pending after the recursive call returns. Therefore in tail recursive functions , there is nothing to be done in unwinding phase, so we can jump directly from the last recursive call to the place where recursive function was first called.

Tail recursion can be efficiently implemented by compilers so we always will try to make our recursive functions tail recursive whenever possible.

Functions which are not tail recursive are called augmentive recursive functions and these types of functions have to finish the pending work after the recursive call finishes.

Indirect and Direct Recursion

If a function fun1() calls another function fun2() and the function fun2() in turn calls function fun1(), then this type of recursion is said to be **indirect** recursion, because the function fun1() is calling itself indirectly.

```
fun1( )
{
……..…….    /* Some statements*/
fun2();
……..…….    /* Some statements*/
}
fun2( )
{
……..…….    /* Some statements*/
fun1();
……..…….     /* Some statements*/
}
```

The chain of functions in indirect recursion may involve any number of functions.For example suppose n number of functions are present starting from f1() to fn() and they are involved as following: f1() calls f2(), f2() calls f3(), f3() calls f4() and so on with fn() calls f1().

If a function calls itself directly i.e. function fun1() is called inside its own function body, then that recursion is called as direct recursion. For example look at the following

```
fun1()
{
...      /* Some statements*/
fun2();
...      /* Some statements*/
}
```

Indirect recursion is complex, so it is rarely used.

Exercise:

Find the output of programs from 1 to 5.

1.
```
void main()
{
 printf("%d\n",count(17243));
 }
  int count(int x)
{
 if(x==0)
       return 0;
 else
        return 1+count(x/10)
 }
```

2.
```
void main()
 {
 printf("%d\n",fun(4,8));
 printf("%d\n",fun(3,8));
}
int fun(int x. int y)
{
 if(x==y)
    return x;
 else
    return (x+y+fun(x+1,y-1));
}
```

3. void main()

```
{
 printf("%d\n",fun(4,9));
printf("%d\n",fun(4,0));
 printf("%d\n",fun(0,4));
}
 int fun(int x, int y)
 {
 if(y==0)
        return 0;
 if(y==1)
       return x;
  return x+fun(x,y-1);
}
```

4. void main()

```
{
 printf("%d\n",fun1(14837));
}
int fun1(int m)
{
return ((m)? m%10+fun1(m/10):0);
```

```
}
```

5.
```
void main()
{
printf("%d\n",fun(3,8));
}
int fun(int x, int y)
{
if(x>y)
   return 1000;
return x+fun(x+1,y);
}
```

6. What is the use of recursion in a program, Explain?
7. Explain the use of stack in recursion.
8. What do you mean by winding and unwinding phase?
9. How to write a recursive function, Explain with example?
10. What is the difference between tail and non-tail recursion, explain with example.
11. What is indirect recursion?
12. What is the difference between iteration and recursion?
13. Write a recursive function to enter a line of text and display it in reverse order, without storing the text in an array.
14. Write a recursive function to count all the prime numbers between number p and q(both inclusive).
15. Write a recursive function to find quotient when a positive integer m is divided by a positive integer n.
16. Write a program using recursive function to calculate binary equivalent of a number.
17. Write a program using recursive function to reverse number.
18. Write a program using recursive function to find remainder when a positive integer m is divided by a positive integer n.
19. Write a recursive function that displays a positive integer in words, for example if the integer is 465 then it is displayed as -four six five.
20. Write a recursive function to print the pyramids

1	abcd
1 2	abc
1 2 3	ab
1 2 3 4	a

21. Write a recursive function to find the Binomial coefficient C(n,k), which is defined as:
    C(n,0)=1
    C(n,n)=1
    C(n,k)=C(n-1,k-1)+C(n-1,k)

LECTURE NOTE 18

STORAGE CLASSES

To completely define a variable one needs to mention its type along with its **storage class**. In other words we can say not only variables have a data type but also they have 'storage classes.

Till now the storage class of any variable is not mentioned because storage classes have defaults. If someone doesn't specify the storage class of a variable during its declaration, the compiler assumes a storage class depending upon the situation in which the variable is going to be used. Therefore we can now say every variable have certain default storage class.

Compiler identifies the physical location within the computer where the string of bits which represents the variable's values are stored from the variable name itself.

Generally there are two kinds of locations in a computer where such a value can be present, these are Memory and CPU registers. The storage class of a particular variable determines in which of the above two locations the variable's value is stored.

There are four properties by which storage class of a variable can be recognized.These are scope, default initial value, scope and life.

A variable's storage class reveals the following things about a variable

(i) Where the variable is stored.
(ii) What is the initial value of the variable if the value of the variable is not specified?
(iii)What is the scope of the variable (To which function or block the variable is available).
(iv) What is the life of particular variable (Up to what extent the variable exists in a program).

There are four storage classes in C, these are Automatic, Static, Register and External. The keywords auto, static, registers and exter are used for the above storage classes respectively.

We can specify a storage class while declaring a variable. The

general syntax is

storage_class datatype variable_name;

The following table shows different properties of a variable which according to its storage class.

Properties / Storage Class	**Storage**	**Default Initial Value**	**Scope**	**Life**
Automatic	Memory	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
Register	CPU Registers	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
Static	Memory	Zero	Local to the block in which the variable is defined	Value of the variable continues to exist between different function calls
External	Memory	Zero	Global	Till the program's execution doesn't come to an end

Automatic Storage Class

Syntax to declare automatic variable is:

auto datatype variablename;

Example:

auto int i;

Features of Automatic Storage Class are as follows

Storage: Memory

Default Initial Value: Garbage Value
Scope: Local to the block in which the variable is defined

Life: Till the control remains within the block in which the variable is defined

All the variables which are declared inside a block or function without any storage class specifier are automatic variables. So by default every variable is automatic variable. We can use auto keyword to declare automatic variables but generally it is not done. Since automatic variables are known inside a function or block only, we may have variables of same name in different functions or blocks without any doubt. Look at the following two functions:

```
void fun1()
{
  int x,y;
/* Some statements*/
}
```

```
void fun1()
 {
   auto int x,y;
   /*Some statements*/
   }
```

In the above two functions declaration statements are equivalent as both declare variables x and y as automatic variables.

The following program illustrates the work of automatic variables.

```
void test();
void main()
{
test();
test();
test();
}
 void test()
{
auto int k=10;
printf("%d\n",k);
k++;
}
```

Output:

10

10

10

In the above program when the function test() is called for the first time ,variable k is created and initialized to 10. When the control returns to main(), k is destroyed. When function test() is called for the second time again k is created , initialized and destroyed after execution of the function. Hence automatic variables came into existence each time the function is executed and destroyed when execution of the function completes.

Register Storage Class

Syntax to declare register variable is:

register datatype variablename;

Features of Register Storage Class are as follows:

Storage: CPU Registers

Default Initial Value: Garbage Value

Scope: Local to the block in which the variable is defined

Life: Till the control remains within the block in which the variable is defined

Register storage class can be applied only to automatic variables. Its scope, life and default initial value are same as that of automatic variables. The only difference between register and automatic variables is the place where they are stored. Automatic variables are stored in memory but register variables are stored in CPU registers. Registers are small storage units present in the processor. The variables stored in registers can be accessed much faster than the variables stored in registers. Hence the variables which are frequently used in a program can be assigned register storage class for faster execution. For example loop counters are declared as register variables, which are defined as follows:

```
int main()
{
 register int a;
for(a=0;i<50000;i++)
printf("%d\t",a);
return 0;
 }
```

In the above program, variable **a** was used frequently as a loop counter so the variable a is defined as a register variable. Register is a not a command but just a request to the compiler to allocate the memory for the variable into the register. If free registers are available than memory will be allocated in to the registers. And if there are no free registers then memory will be allocated in the RAM only (Storage is in memory i.e. it will act as automatic variable).

Every type of variables can be stored in CPU registers. Suppose the microprocessor has 16 bit registers then they can't hold a float or a double value which requires 4 and 8 bytes respectively. But if you use register storage class for a float or double variable then you will not get any error message rather the compiler will treat the float and double variable as be of automatic storage class(i.e. Will treat them like automatic variables).

Static Storage Class

Syntax to declare static variable is:

static datatype variablename;

Example:

static int i;

Features of Static Storage Class are as follows

Storage: Memory

Default Initial Value: Zero

Scope: Local to the block in which the variable is defined

Life: Value of the variable continues to exist between different function calls

Now look at the previous program with k is declared as static instead of automatic.

```
void test();
void main()
{
 test();
 test();
 test();
 }
void test()
{
```

```
static int k=10;
 printf("%d\n",k);
 k++;
}
```

Output:

10

11

12

Here in the above program the output is 10, 11, 12, because if a variable is declared as static then it is initialized only once and that variable will never be initialized again. Here variable k is declared as static, so when test() is called for the first time k value is initialized to 10 and its value is incremented by 1 and becomes 11. Because k is static its value persists. When test() is called for the second time k is not reinitialized to 10 instead its old value 11 is available. So now 11 is get printed and it value is incremented by 1 to become 12. Similarly for the third time when test() is called 12(Old value) is printed and its value becomes 13 when executes the statement 'k++;'

So the main difference between automatic and static variables is that static variables are initialized to zero if not initialized but automatic variables contain an unpredictable value(Garbage Value) if not initialized and static variables does not disappear when the function is no longer active , their value persists, i.e. if the control comes back to the same function again the static variables have the same values they had last time.

General advice is avoid using static variables in a program unless you need them, because their values are kept in memory when the variables are not activc which means they occupies space in memory that could otherwise be used by other variables.

External Storage Class

Syntax to declare static variable is:

extern datatype variablename;

Example:

```
extern int i;
```

Features of External Storage Class are as follows

Storage: Memory

Default Initial Value: Zero

Scope: Global

Life: Till the program's execution doesn't come to an end

External variables differ from automatic, register and static variables in the context of scope, external variables are global on the contrary automatic, register and static variables are local. External variables are declared outside all functions, therefore are available to all functions that want to use them.

If the program size is very big then code may be distributed into several files and these files are compiled and object codes are generated. These object codes linked together with the help of linker and generate ".exe" file. In the compilation process if one file is using global variable but it is declared in some other file then it generate error called undefined symbol error. To overcome this we need to specify global variables and global functions with the keyword extern before using them into any file. If the global variable is declared in the middle of the program then we will get undefined symbol error, so in that case we have to specify its prototype using the keyword extern.

So if a variable is to be used by many functions and in different files can be declared as external variables. The value of an uninitialized external variable is zero. The declaration of an external variable declares the type and name of the variable, while the definition reserves storage for the variable as well as behaves as a declaration. The keyword **extern** is specified in declaration but not in definition. Now look at the following four statements

1. auto int a;
2. register int b;
3. static int c;
4. extern int d;

Out of the above statements first three are definitions where as the last one is a declaration.
Suppose there are two files and their names are File1.c and File2.c respectively. Their contents are as follows:

File1.c
int n=10; *void hello()* *{* *printf("Hello");* *}*

File2.c
extern int n;
extern void hello();
void main()
{
printf("%d", n);
hello();
}

In the above program File1.obj+ File2.obj=File2.exe and in File2.c value of n will be 10.

Where to use which Storage Class for a particular variable:

We use different storage class in appropriate situations in a view to

1. Economise the memory space consumed by the variables
2. Improve the speed of execution of the program.

The rules that define which storage class is to be executed when are as follows:

(a) Use static storage class if you want the value of a variable to persist between different function calls.

(b) Use register storage class for those variables that are being used very often in a program, for faster execution. A typical application of register storage class is loop counters.

(c) Use extern storage class for only those variables that are being used by almost all the functions in a program, this avoids unnecessary passing of these variables as arguments when making a function call.

(d) If someone do not have any need mentioned above, then use auto storage class.

Exercise:

[A] Answer the following Questions.

(a) What do you mean by storage class of a variable?

(b) Why register storage class is required?

(c) How many storage classes are there in c and why they are used?

(d) Why someone needs static storage variable in a program?

(e) Which storage class is termed as default storage class, if the storage class of a variable is not mentioned in a program? Explain its working with the help of a suitable example.

(f) Can you store a floating point variable in CPU Registers, Explain?

(g) What do you mean by scope and life of a variable?

(h) Justify where to use which storage class.

(i) What do you mean by external variable and where it is defined?

(j) What is the difference between auto and static variables in a program?

(k) What do you mean by definition and declaration of a variable?

[B] What is the output of the following programs:

(a)
```
void main()
{
static int a=6;
printf("\na=%d",a--);
if(a!=0)
main();
}
```

(b)
```
void main()
{
int i,j;
for(i=1;i<5;i++)
{
j=fun1(i);
printf("\n%d",j);
}
}
int fun1(int a)
```

```
{
 static int b=2;
  int c=3;
  b=a+b;
  return(a+b+c);
 }
```

(c)
```
void main()
{
fun1();
fun1();
}
void fun1()
{
 auto int x=0;
  register int y=0;
 static int z=0;
 x++;
 y++;
 z++;
 printf("\n%d%d%d",x,y,z);
}
```

(d) int a=10;

```
void main()
    {
    int a=20;
    {
    int a=30;
printf("%d\n",a);
    }
    printf("%d",a);
    }
```

[C] State whether the following statements are True or False:

(a) The register storage class variables cannot hold float values.

(b) The value of an automatic storage class variable persists between various function invocations.

(c) If the CPU registers are not available, the register storage class variables are treated as static storage class variables.

(d) The default value for automatic variable is zero.

(e) If a global variable is to be defined, then the extern keyword is necessary in its declaration.

(f) If we try to use register storage class for a float variable the compiler will flash an error message.

(g) Storage for a register storage class variable is allocated each time the control reaches the block in which the variable is present.

(h) An extern storage class variable is not available to the functions that precede its definition, unless the variable is explicitly declared in the above functions.

(i) The life of static variable is till the control remains within the block in which it is defined.

(j) The address of a register variable is not accessible.

[D] Following program calculates the sum of digits of the number 25634. Go through it and find out why is it necessary to declare the storage class of the variable sum as static.

```
#include<stdio.h>

 void main()

{ int m;

m=sumdigit(25634);

 printf("\n%d",m);

}

int sumdigit(int n)

{

static int sum;

 int p,q;

p=n%10;

q=(n-p)/10;

sum=sum+p;

if(q!=0) sumdigit(q);

else

return(sum);
 }
```

LECTURE NOTE 19

ARRAYS

Introduction

A *data structure* is the way data is stored in the machine and the functions used to access that data. An easy way to think of a data structure is a collection of related data items. An *array* is a data structure that is a collection of variables of one type that are accessed through a common name. Each element of an array is given a number by which we can access that element which is called an index. It solves the problem of storing a large number of values and manipulating them.

Arrays

Previously we use variables to store the values. To use the variables we have to declare the variable and initialize the variable i.e, assign the value to the variable. Suppose there are 1000 variables are present, so it is a tedious process to declare and initialize each and every variable and also to handle 1000 variables. To overcome this situation we use the concept of array .In an Array values of same type are stored. An array is a group of memory locations related by the fact that they all have the same name and same type. To refer to a particular location or element in the array we specify the name to the array and position number of particular element in the array.

One Dimensional Array

Declaration:

Before using the array in the program it must be declared

Syntax:

data_type array_name[size];

data_type represents the type of elements present in the array.

array_name represents the name of the array.

Size represents the number of elements that can be stored in the array.

Example:

int age[100];

float sal[15];

char grade[20];

Here age is an integer type array, which can store 100 elements of integer type. The array sal is floating type array of size 15, can hold float values. Grade is a character type array which holds 20 characters.

Initialization:

We can explicitly initialize arrays at the time of declaration.

Syntax:

data_type array_name[size]={value1, value2,……..valueN};

Value1, value2, valueN are the constant values known as initializers, which are assigned to the array elements one after another.

Example:

int marks[5]={10,2,0,23,4};

The values of the array elements after this initialization are:

marks[0]=10, marks[1]=2, marks[2]=0, marks[3]=23, marks[4]=4

NOTE:

1. In 1-D arrays it is optional to specify the size of the array. If size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

Example:

int marks[]={10,2,0,23,4};

Here the size of array marks is initialized to 5.

2. We can't copy the elements of one array to another array by simply assigning it.

Example:

```
int a[5]={9,8,7,6,5};
int b[5];
b=a;     //not valid
```

we have to copy all the elements by using for loop.

```
for(a=i; i<5; i++)
    b[i]=a[i];
```

Processing:

For processing arrays we mostly use for loop. The total no. of passes is equal to the no. of elements present in the array and in each pass one element is processed.

Example:

```
#include<stdio.h>

main()

{

 int a[3],i;

 for(i=0;i<=2;i++)                    //Reading the array values

{

 printf("enter the elements");

scanf("%d",&a[i]);

}

 for(i=0;i<=2;i++)                    //display the array values

{

 printf("%d",a[i]);

printf("\n");

}

}
```

This program reads and displays 3 elements of integer type.

Example:1

C Program to Increment every Element of the Array by one & Print Incremented Array.

```
#include <stdio.h>

void main()

{

    int i;

   int array[4] = {10, 20, 30, 40};

for (i = 0; i < 4; i++)

    arr[i]++;

 for (i = 0; i < 4; i++)

    printf("%d\t", array[i]);

}
```

Example: 2

C Program to Print the Alternate Elements in an Array

```
 #include <stdio.h>
  void main()
 {
    int array[10];
 int i, j, temp;
    printf("enter the element of an array \n");
 for (i = 0; i < 10; i++)
    scanf("%d", &array[i]);
    printf("Alternate elements of a given array \n");
 for (i = 0; i < 10; i += 2)
     printf( "%d\n", array[i]) ;
 }
```

Example-3

C program to accept N numbers and arrange them in an ascending order

```
#include <stdio.h>
void main()
{
 int i, j, a, n, number[30];
printf("Enter the value of N \n");
scanf("%d", &n);
printf("Enter the numbers \n");
for (i = 0; i < n; ++i)
          scanf("%d", &number[i]);
for (i = 0; i < n; ++i)
{
    for (j = i + 1; j < n; ++j)
    {
        if (number[i] > number[j])
        {
            a =number[i];
            number[i] = number[j];
            number[j] = a;
        }
    }
}
  printf("The numbers arranged in ascending order are given below \n");
for (i = 0; i < n; ++i)
       printf("%d\n", number[i]);
}
```

LECTURE NOTE 20

TWO DIMENSIONAL ARRAYS

Arrays that we have considered up to now are one dimensional array, a single line of elements. Often data come naturally in the form of a table, e.g. spreadsheet, which need a two-dimensional array.

Declaration:

The syntax is same as for 1-D array but here 2 subscripts are used.

Syntax:

data_type array_name[rowsize][columnsize];

Rowsize specifies the no.of rows Columnsize

specifies the no.of columns.

Example:

int a[4][5];

This is a 2-D array of 4 rows and 5 columns. Here the first element of the array is a[0][0] and last element of the array is a[3][4] and total no.of elements is 4*5=20.

	col 0	col 1	col 2	col 3	col 4
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
row 3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

Initialization:

2-D arrays can be initialized in a way similar to 1-D arrays.

Example:

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

The values are assigned as follows:

m[0][0]:1 m[0][1]:2 m[0][2]:3
m[1][0]:4 m[1][1]:5 m[3][2]:6

m[2][0]:7 m[2][1]:8 m[3][2]:9

m[3][0]:10 m[3][1]:11 m[3][2]:12

The initialization of group of elements as follows:

int m[4][3]={{11},{12,13},{14,15,16},{17}};

The values are assigned as:

m[0][0]:1 1 m[0][1]:0 m[0][2]:0

m[1][0]:12 m[1][1]:13 m[3][2]:0

m[2][0]:14 m[2][1]:15 m[3][2]:16

m[3][0]:17 m[3][1]:0 m[3][2]:0

Note:

In 2-D arrays it is optional to specify the first dimension but the second dimension should always be present.

Example: *int*

m[][3]={

{1,10},

{2,20,200},

{3},

{4,40,400} };

Here the first dimension is taken 4 since there are 4 roes in the initialization list. A 2-D array is known as matrix.

Processing:

For processing of 2-D arrays we need two nested for loops. The outer loop indicates the rows and the inner loop indicates the columns.

Example:

```
int a[4][5];
```

a) Reading values in a

```
for(i=0;i<4;i++)
    for(j=0;j<5;j++)
        scanf("%d",&a[i][j]);
```

b) Displaying values of a

```
for(i=0;i<4;i++)
    for(j=0;j<5;j++)
      printf("%d",a[i][j]);
```

Example 1:

Write a C program to find sum of two matrices

```
#include <stdio.h>
#include<conio.h>
void  main()
{
  float a[2][2], b[2][2], c[2][2];
   int i,j;
   clrscr();
  printf("Enter the elements of 1st matrix\n");
/* Reading two dimensional Array with the help of two for loop. If there is an array of 'n' dimension, 'n' numbers of loops are needed for inserting data to array.*/
 for(i=0;i<2;I++)
  for(j=0;j<2;j++)
  {
     scanf("%f",&a[i][j]);
  }
  printf("Enter the elements of 2nd matrix\n");
   for(i=0;i<2;i++)
   for(j=0;j<2;j++)
   {
```

```
      scanf("%f",&b[i][j]);
      }
/* accessing corresponding elements of two arrays. */
for(i=0;i<2;i++)
       for(j=0;j<2;j++)
         {
              c[i][j]=a[i][j]+b[i][j];  /* Sum of corresponding elements of two arrays. */
         }
 /* To display matrix sum in order. */
printf("\nSum Of Matrix:");
for(i=0;i<2;++i)
     {
       for(j=0;j<2;++j)
         printf("%f", c[i][j]);
       printf("\n");
     }
getch();
}
```

Example 2: Program for multiplication of two matrices

```
#include<stdio.h>
#include<conio.h>
int main()
  { int i,j,k;
    int row1,col1,row2,col2,row3,col3;
    int mat1[5][5], mat2[5][5], mat3[5][5];
    clrscr();
   printf("\n enter the number of rows in the first matrix:");
  scanf("%d", &row1);
   printf("\n enter the number of columns in the first matrix:");
  scanf("%d", &col1);
   printf("\n enter the number of rows in the second matrix:");
  scanf("%d", &row2);
  printf("\n enter the number of columns in the second matrix:");
  scanf("%d", &col2);
 if(col1 != row2)
   {
      printf("\n The number of columns in the first matrix must be equal to the number of rows
      in the second matrix ");
```

```
      getch();
      exit();
   }
row3= row1;
col3= col3;
printf("\n Enter the elements of the first matrix");
for(i=0;i<row1;i++)
  {
     for(j=0;j<col1;j++)
      scanf("%d",&mat1[i][j]);
  }

 printf("\n Enter the elements of the second matrix");
 for(i=0;i<row2;i++)
 {
     for(j=0;j<col2;j++)
     scanf("%d",&mat2[i][j]);
  }
for(i=0;i<row3;i++)
 {
   for(j=0;j<col3;j++)
      {
         mat3[i][j]=0;
         for(k=0;k<col3;k++)
             mat3[i][j]  +=mat1[i][k]*mat2[k][j];
        }
     }
printf("\n The elements of the product matrix are"):
for(i=0;i<row3;i++)
 {
     printf("\n");
    for(j=0;j<col3;j++)
     printf("\t %d", mat3[i][j]);
 }
return 0;
}
```

Output:

Enter the number of rows in the first matrix: 2

```
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
 1 2 3 4
Enter the elements of the second matrix
 5 6 7 8
The elements of the product matrix are
 19 22
 43 50
```

Example 3:

Program to find transpose of a matrix.

```
#include <stdio.h>
 int main()
 {
    int a[10][10], trans[10][10], r, c, i, j;
    printf("Enter rows and column of matrix: ");
    scanf("%d %d", &r, &c);
    printf("\nEnter elements of matrix:\n");
    for(i=0; i<r; i++)
     for(j=0; j<c; j++)
     {
       printf("Enter elements a%d%d: ",i+1,j+1);
       scanf("%d", &a[i][j]);
   }
/* Displaying the matrix a[][] */
printf("\n Entered Matrix: \n");
for(i=0; i<r; i++)
   for(j=0; j<c; j++)
   {
      printf("%d  ",a[i][j]);
      if(j==c-1)
          printf("\n\n");
   }

/* Finding transpose of matrix a[][] and storing it in array trans[][]. */
for(i=0; i<r;i++)
   for(j=0; j<c; j++)
   {
```

```
      trans[j][i]=a[i][j];
   }

/* Displaying the array trans[][]. */
printf("\nTranspose of Matrix:\n");
for(i=0; i<c;i++)
   for(j=0; j<r;j++)
   {
      printf("%d  ",trans[i][j]);
       if(j==r-1)
          printf("\n\n");
   }
return 0;
}
```

Output

```
Enter the rows and columns of matrix: 2   3
Enter the elements of matrix:
Enter elements a11: 1
Enter elements a12: 2
Enter elements a13: 9
Enter elements a21: 0
Enter elements a22: 4
Enter elements a23: 7
Entered matrix:
1 2 9
0 4 7
Transpose of matrix:
1 0
2 4
9 7
```

Multidimensional Array

More than 2-dimensional arrays are treated as multidimensional arrays.

Example:

```
int a[2][3][4];
```

Here a represents two 2-dimensional arrays and each of these 2-d arrays contains 3 rows and 4 columns.

The individual elements are:

a[0][0][0], a[0][0][1],a[0][0][2],a[0][1][0]……a[0][3][2]

a[1][0][0],a[1][0][1],a[1][0][2],a[1][1][0]……..a[1][3][2]

the total no. of elements in the above array is 2*3*4=24.

Initialization:

```
int a[2][4][3]={
{
 {1,2,3},
{4,5},
{6,7,8},
{9}
},
{
{10,11},
{12,13,14},
{15,16},
{17,18,19}
}
}
```

The values of elements after this initialization are as:

a[0][0][0]:1 a[0][0][1]:2 a[0][0][2]:3

a[0][1][0]:4 a[0][1][1]:5 a[0][1][2]:0

a[0][2][0]:6 a[0][2][1]:7 a[0][2][2]:8

a[0][3][0]:9 a[0][3][1]:0 a[0][3][2]:0

a[1][0][0]:10 a[1][0][1]:11 a[1][0][2]:0

a[1][1][0]:12 a[1][1][1]:13 a[1][1][2]:14

a[1][2][0]:15 a[1][2][1]:16 a[1][2][2]:0

a[1][3][0]:17 a[1][3][1]:18 a[1][3][2]:19

Note:

The rule of initialization of multidimensional arrays is that last subscript varies most frequently and the first subscript varies least rapidly.

Example:

```
#include<stdio.h>
 main()
{
 int d[5];
int i;
for(i=0;i<5;i++)
{
 d[i]=i;
}
for(i=0;i<5;i++)
{
printf("value in array %d\n",a[i]);
}
}
```

pictorial representation of d will look like

d[0]	d[1]	d[2]	d[3]	d[4]
0	1	2	3	4

LECTURE NOTE 21

ARRAYS USING FUNCTIONS

1-d arrays using functions

Passing individual array elements to a function

We can pass individual array elements as arguments to a function like other simple variables.

Example:

```
#include<stdio.h>
void check(int);
void main()
{
 int a[10],i;
 clrscr();
 printf("\n enter the array elements:");
for(i=0;i<10;i++)
{
 scanf("%d",&a[i]);
 check(a[i]);
}
 void check(int num)
{
if(num%2==0)
  printf("%d is even\n",num);
else
   printf("%d is odd\n",num);
}
```

Output:
enter the array elements:

1 2 3 4 5 6 7 8 9 10

1 is odd

2 is even

3 is odd

4 is even

5 is odd

6 is even

7 is odd

8 is even

9 is odd

10 is even

Example:

C program to pass a single element of an array to function

```
#include <stdio.h>
 void display(int a)
   {
    printf("%d",a);
   }
int main()
{
    int c[]={2,3,4};
    display(c[2]);  //Passing array element c[2] only.
   return 0;
}
```

Output

2 3 4

Passing whole 1-D array to a function

We can pass whole array as an actual argument to a function the corresponding formal arguments should be declared as an array variable of the same type.

Example:

```
#include<stdio.h>
main()
{
int i, a[6]={1,2,3,4,5,6};
 func(a);
 printf("contents of array:");
for(i=0;i<6;i++)
        printf("%d",a[i]);
 printf("\n");
}
func(int val[])
{
int sum=0,i;
 for(i=0;i<6;i++)
{
        val[i]=val[i]*val[i];
        sum+=val[i];
}
printf("the sum of squares:%d", sum);
}
```

Output

contents of array: 1 2 3 4 5 6

the sum of squares: 91

Example.2:

Write a C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.

```
#include <stdio.h>
 float average(float a[]);
int main()
{
    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
    avg=average(c);   /* Only name of array is passed as argument. */
    printf("Average age=%.2f",avg);
   return 0;
 }
float average(float a[])
{
   int i;
   float avg, sum=0.0;
  for(i=0;i<6;++i)
          {
                sum+=a[i];
       }
    avg =(sum/6);
   return avg;
}
```

Output

Average age= 27.08

Solved Example:

1. Write a program to find the largest of n numbers and its location in an array.

```
#include <stdio.h>
```

```
#include<conio.h>
void main()
{
 int array[100], maximum, size, c, location = 1;
clrscr();
printf("Enter the number of elements in array\n");
scanf("%d", &size);
printf("Enter %d integers\n", size);

 for (c = 0; c < size; c++)
   scanf("%d", &array[c]);
maximum = array[0];

 for (c = 1; c < size; c++)
 {
  if (array[c] > maximum)
  {
    maximum  = array[c];
    location = c+1;
  }
 }
 printf("Maximum element is present at location %d and it's value is %d.\n", location,
maximum);
 getch();
}
```

Output:

Enter the number of elements in array
5
Enter 5 integers
2
4
7
9
1

Maximum element is present at location 4 and it's value is 9

2. Write a program to enter n number of digits. Form a number using these digits.

```
# include<stdio.h>
#include<conio.h>
#include<math.h>
void  main()
{
 int    number=0,digit[10],    numofdigits,i;
clrscr();
printf("\n Enter the number of digits:");
scanf("%d", &numofdigits);
for(i=0;i<numofdigits;i++)
     {
        printf("\n Enter the %d th digit:", i);
       scanf("%d",&digit[i]);
      }
i=0;
while(i<numofdigits)
{
  number= number + digit[i]* pow(10,i)
  i++;
}
printf("\n The number is : %d",number);
getch();
}
```

Output:

Enter the number of digits: 3
Enter the 0^{th} digit: 5
Enter the 1th digit: 4
Enter the 2th digit: 3
The number is: 543

3. Matrix addition:

```
#include <stdio.h>
#include<conio.h>
 void main()
{
```

```
  int m, n, c, d, first[10][10], second[10][10], sum[10][10];
clrscr();
  printf("Enter the number of rows and columns of matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter the elements of first matrix\n");
  for ( c = 0 ; c < m ; c++ )
for ( d = 0 ; d < n ; d++ )
      scanf("%d", &first[c][d]);

  printf("Enter the elements of second matrix\n");

for ( c = 0 ; c < m ; c++ )
for ( d = 0 ; d < n ; d++ )
        scanf("%d", &second[c][d]);

  for ( c = 0 ; c < m ; c++ )
    for ( d = 0 ; d < n ; d++ )
      sum[c][d] = first[c][d] + second[c][d];

  printf("Sum of entered matrices:-\n");

  for ( c = 0 ; c < m ; c++ )
  {
    for ( d = 0 ; d < n ; d++ )
      printf("%d\t", sum[c][d]);

    printf("\n");
  }

  getch();
}
```

Output:

Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
1 2
3 4
Enter the elements of second matrix
5 6

2 1
Sum of entered matrices:- 6
8
5 5

Exercise

1. Compute sum of elements of an array in a program?
2. Write a program for histogram printing using an array?
3. Write a program for dice-rolling using an array instead of switch?
4. Sorting an array with bubble sort?
5. Write a program for binary search using an array?
6. Write a program to interchange the largest and the smallest number in the array.
7. Write a program to fill a square matrix with value 0 on the diagonals, 1 on the upper right triangle, and -1 on the lower left triangle.
8. Write a program to read and display a 2x2x2 array.
9. Write a program to calculate the number of duplicate entries in the array.
10. Given an array of integers, calculate the sum, mean, variance and standard deviation of the numbers in the array.
11. Write a program that reads a matrix and displays the sum of the elements above the main diagonal.
12. Write a program to calculate XA + YB where A and B are matrices and X=2, and Y=3

LECTURE NOTE 22

FUNDAMENTALS OF STRINGS

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, / and $. String literals or string constants in C are written in double quotation marks as follows:

"1000 Main Street" (a street address)

"(080)329-7082" (a telephone number)

"Kalamazoo, New York" (a city)

In C language strings are stored in array of char type along with null terminating character '\0' at the end.

When sizing the string array we need to add plus one to the actual size of the string to make space for the null terminating character, '\0'.

Syntax:

char fname[4];

The above statement declares a string called fname that can take up to 3 characters. It can be indexed just as a regular array as well.

fname[]={'t','w','o'};

character	t	w	o	\0
ASCII code	116	119	41	0

Generalized syntax is:-

char str[size];

when we declare the string in this way, we can store size-1 characters in the array because the last character would be the null character. For example,

char mesg[10]; can store maximum of 9 characters.

If we want to print a string from a variable, such as four name string above we can do this.

e.g., *printf("First name:%s",fname);*
We can insert more than one variable. Conversion specification %s is used to insert a string and then go to each %s in our string, we are printing.

A string is an array of characters. Hence it can be indexed like an array.

char ourstr[6] = "EED";

– ourstr[0] is 'E'

– ourstr[1] is 'E'

– ourstr[2] is 'D'

– ourstr[3] is '\0'

– ourstr[4] is '\0' – ourstr[5] is '\0'

'E'	'E'	'D'	\0	'\0'	'\0'
ourstr[0]	ourstr[1]	ourstr[2]	ourstr[3]	ourstr[4]	ourstr[5]

Reading strings:

If we declare a string by writing

char str[100];

then str can be read from the user by using three ways;

1. Using scanf() function
2. Using gets() function
3. Using getchar(), getch(), or getche() function repeatedly

The string can be read using scanf() by writing
scanf("%s",str);
Although the syntax of scanf() function is well known and easy to use, the main pitfall with this function is that it terminates as soon as it finds a blank space. For example, if the user enters Hello World, then str will contain only Hello. This is because the moment a blank space is encountered, the string is terminated by the scanf() function.

Example:

```
char str[10];
printf("Enter a string\n");
scanf("%s",str);
```

The next method of reading a string a string is by using gets() function. The string can be read by writing

```
gets(str);
```

gets() is a function that overcomes the drawbacks of scanf(). The gets() function takes the starting address of the string which will hold the input. The string inputted using gets() is automatically terminated with a null character.

Example:

```
char str[10];
 printf("Enter a string\n");
 gets(str);
```

The string can also be read by calling the getchar() repeatedly to read a sequence of single characters (unless a terminating character is encountered) and simultaneously storing it in a character array as follows:

```
 int i=0;

char str[10],ch;

 getchar(ch);

 while(ch!='\0')

{
    str[i]=ch;      // store the read character in str

     i++;

    getch(ch);   // get another character
  }
 str[i]='\0';       // terminate str with null character
```

Writing string

The string can be displayed on screen using three ways:

1. Using printf() function
2. Using puts() function
3. Using putchar() function repeatedly

The string can be displayed using pintf() by writing

printf("%s",str);

We can use width and precision specification along with %s. The width specifies the minimum output field width and the precision specifies the maximum number of characters to be displayed. Example:

printf("%5.3s",str);

this statement would print only the first three characters in a total field of five charaters; also these three characters are right justified in the allocated width.

The next method of writing a string is by using the puts() function. The string can be displayed by writing:

puts(str);

It terminates the line with a newline character ('\n'). It returns an EOF(-1) if an error occurs and returns a positive number on success.

Finally the string can be written by calling the putchar( ) function repeatedly to print a sequence of single characters.

```
int i=0;
char str[10];
while(str[i]!='\0')
{
    putchar(str[i]);    // print the character on the screen
     i++;
 }
```

Example: Read and display a string

```
#include<stdio.h>
#include<conio.h>
void main()
{
 char str[20];
 clrscr();
 printf("\n Enter a string:\n");
gets(str);
 scanf("The string is:\n");
puts(str);
        getch();  }
```

Output:

```
 Enter a string:
vssut burla
The string is:
vssut burla
```

LECTURE NOTE 23

COMMON FUNCTIONS IN STRING

Type	*Method*	*Description*
char	*strcpy(s1, s2)*	Copy string
char	*strcat(s1, s2)*	Append string
int	*strcmp(s1, s2)*	Compare 2 strings
int	*strlen(s)*	Return string length
char	*strchr(s, int c)*	Find a character in string
char	*strstr(s1, s2)*	Find string s2 in string s1

strcpy():

It is used to copy one string to another string. The content of the second string is copied to the content of the first string.

Syntax:

strcpy (string 1, string 2);

Example:

char mystr[10];

mystr = "Hello"; // Error! Illegal !!! Because we are assigning the value to mystr which is not possible in case of an string. We can only use "=" at declarations of C-String.

strcpy(mystr, "Hello");

It sets value of mystr equal to "Hello".

strcmp():

It is used to compare the contents of the two strings. If any mismatch occurs then it results the difference of ASCII values between the first occurrence of 2 different characters.

Syntax:

int strcmp(string 1, string 2);

Example:

```
char mystr_a[10] = “Hello”;
char mystr_b[10] = “Goodbye”;

– mystr_a == mystr_b;    // NOT allowed!
```

The correct way is

```
if (strcmp(mystr_a, mystr_b ))
printf ("Strings are NOT the same.");
else
 printf( "Strings are the same.");
```

Here it will check the ASCII value of H and G i.e, 72 and 71 and return the diference 1.

strcat():

It is used to concatenate i.e, combine the content of two strings.

Syntax:

```
strcat(string 1, string 2);
```

Example:

```
char fname[30]={“bob”};

 char lname[]={“by”};

 printf(“%s”, strcat(fname,lname));
```

Output:

bobby.

strlen():

It is used to return the length of a string.

Syntax:

```
int strlen(string);
```

Example:
char fname[30]={“bob”};

int length=strlen(fname);

It will return 3

strchr():

It is used to find a character in the string and returns the index of occurrence of the character for the first time in the string.

Syntax:

strchr(cstr);

Example:

char mystr[] = "This is a simple string";

char pch = strchr(mystr, ‘s’);

The output of pch is mystr[3]

strstr():

It is used to return the existence of one string inside another string and it results the starting index of the string.

Syntax:

strstr(cstr1, cstr2);

Example:

Char mystr[]="This is a simple string";

char pch = strstr(mystr, “simple”);

here pch will point to mystr[10]

- **String input/output library functions**

Function prototype	Function description
int getchar(void);	Inputs the next character from the standard input and returns it as integer
int putchar(int c);	Prints the character stored in c and returns it as an integer
int puts(char s);	Prints the string s followed by new line character. Returns a non-zero integer if possible or EOF if an error occurs
int sprint(char s, char format,....)	Equivalent to printf,except the output is stored in the array s instead of printed in the screen. Returns the no.of characters written to s, or EOF if an error occurs
int sprint(char s, char format,....)	Equivalent to scanf, except the input is read from the array s rather than from the keyboard. Returns the no.of items successfully read by the function , or EOF if an error occurs

NOTE:

Character arrays are known as strings.

Self-review exercises:

1. Find the error in each of the following program segments and explain how to correct it:
   - *char s[10];*
   - *strcpy(s,"hello",5);*
   - *prinf("%s\n",s);*
   - *printf("%s",'a');*
   - *char s[12]; strcpy(s,"welcome home");*
   - *If (strcmp(string 1, sring 2))*

     { printf("the strings are equal\n");

     }
2. Show 2 different methods of initializing character array vowel with the string of vowels "AEIOU"?
3. Writ a program to convert string to an integer?

4. Write a program to accept a line of text and a word. Display the no. of occurrences of that word in the text?
5. Write a program to read a word and re-write its characters in alphabetical order.
6. Write a program to insert a word before a given word in the text.
7. Write a program to count the number of characters, words and lines in the given text.