Unit 3

Conditional Program Execution / Decision Making

3 Introduction

We have seen that a C program is a set of statements which are normally executed sequentially in the order in which they appear. However, in practice, we have a number of situations where we may have to change the order of execution of the statements based on certain conditions or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.
C language possesses such decision-making capabilities by supporting the following statements:

- **1.1 IF statement**
- **1.2 IF-ELSE statement**
- **1.3 Nesting of IF-ELSE statements**
- **1.4 ELSE-IF ladder**
- **1.5 SWITCH statement**
- **1.6 GOTO statement**
- **1.7 Conditional operator/ Ternary operator (? :)**

These statements are popularly known as decision-making statements.

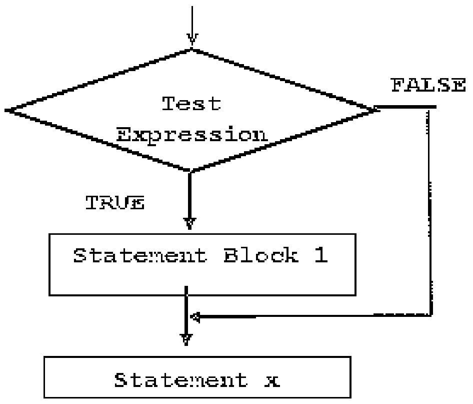
3.1 IF Statement:

Theif statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a two way decision statement and is used in conjunction with an expression. It takes the following form:

```
if (test expression)
{
statement block 1;
}
statement x;
```

The statement block 1 may be a single statement or a group of statements. If the test expression is true, the statement block 1 will be executed; otherwise the statement block 1 will be skipped and the control will jump to the statement-x. When the condition is true both the statement block 1 and the statement x are executed in sequence.

Flowchart



Program

```
void main()
{
int age;
printf("\nEnter Age");
scanf("%d",&age);
if(age>=18)
printf("\nYou are eligible to vote");
}
```

Output

```
Enter Age
20
You are eligible to vote
Enter Age
10
No output
```

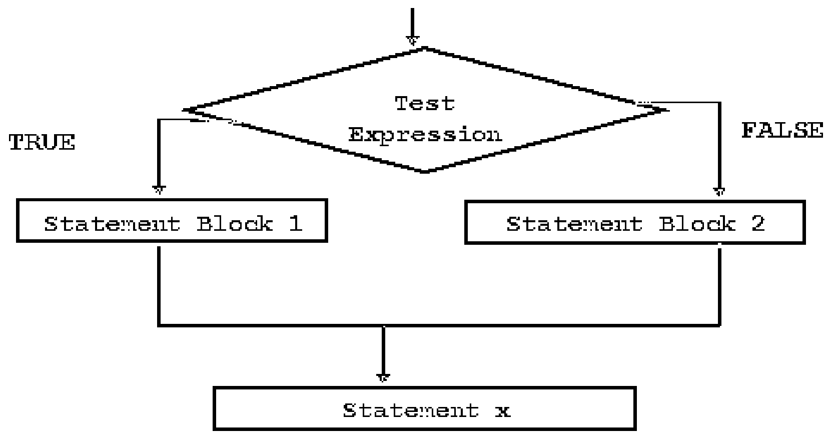
The second run of the program does not produced any message because the expression (age>=18) results to false.

3.2 IF-ELSE Statement:

In the if-else construct, first the test expression is evaluated. If the expression is true, statement block 1 is executed and statement block 2 is skipped. Otherwise, if the expression is false, statement block 2 is executed and statement block 1 is ignored. In any case after the statement block 1 or 2 gets executed and finally the control will pass to statement x. Therefore, statement x is executed in every case. It takes the following form:

```
if (test expression)
{
statement block1;
}
else
{
statement block 2;
}
statement x;
```

Flowchart



Program

```
// Program to find whether a number is even or odd.
#include<stdio.h>
void main()
{
        int a;
        printf("\n Enter the value of a : ");
        scanf("%d", &a);
        if(a%2==0)
                printf("\n %d is even", a);
        else
                printf("\n %d is odd", a);
        return 0;
}
```

Output

```
Enter the value of a
20
20 is even
Enter the value of a
13
13 is odd
```

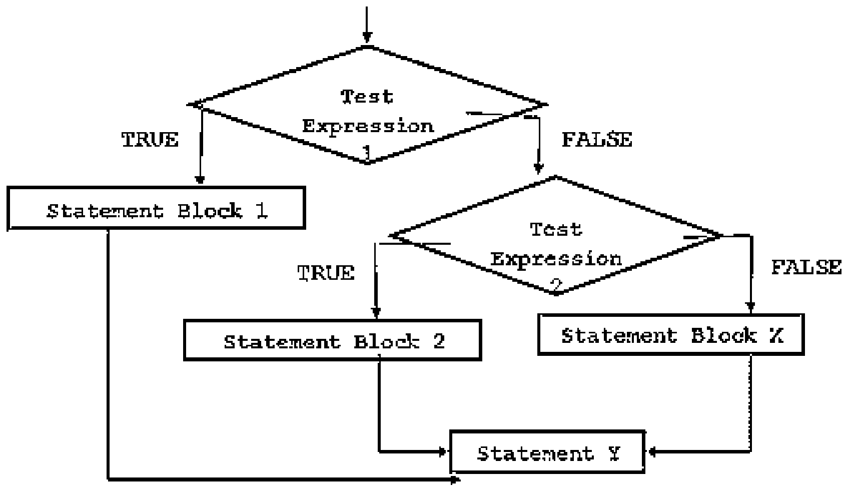
3.3 Nesting of IF-ELSE Statements:

When a series of decisions are involved we may have to use more than one if-else statement in nested form as shown below:

```
if(test condition1)
{
        if(test condition2)
        {
                statement block 1;
        }
        else
        {
                statement block 2;
        }
}
else
{
        statement block x;
}
statement y;
```

If the condition 1 is false, the statement block x will be executed; otherwise it continues to perform second test. If the condition 2 is true the statement block 1 will be evaluated otherwise statement block 2 will be executed and then the control is transferred to statement y.

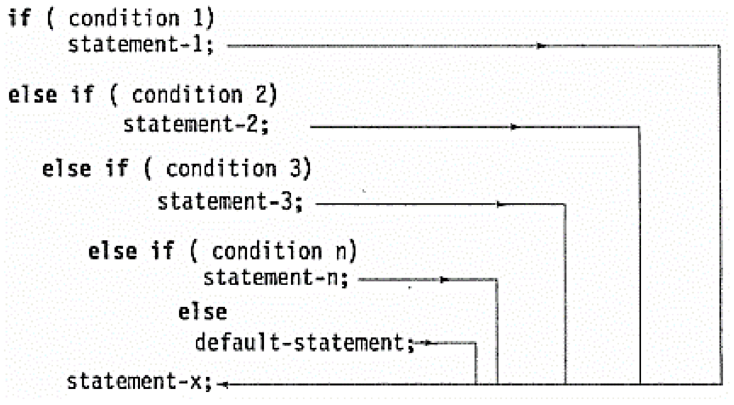
Flowchart



Program

```
void main()
{
int A,B,C;
printf("\nEnter the numbers");
scanf("%d%d%d", &A,&B,&C);
if(A>B)
{
        if(A>C)
        printf("A");
        else
        printf("C");
}
else
{
        if(B>C)
        printf("B");
        else
        printf("C");
}
```

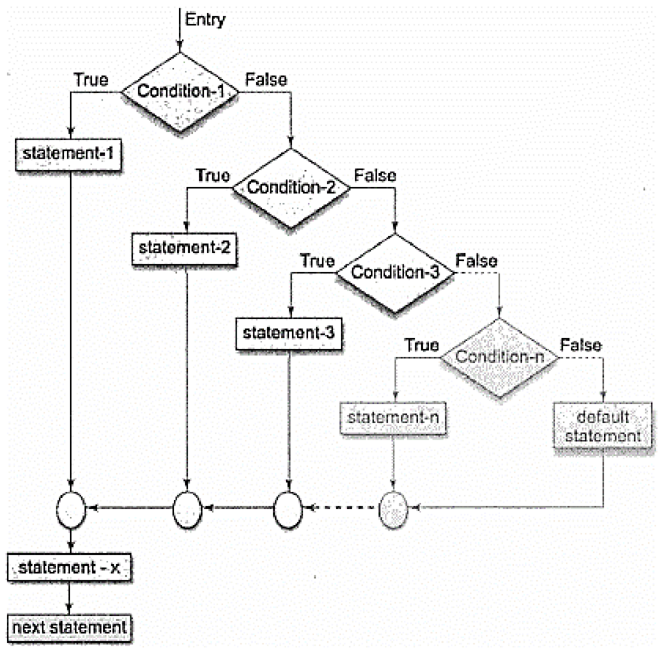
3.4 ELSE-IF Ladder:
There is another way of putting ifs together when multipath decisions are involved. A multipath decisions is a chain of ifs in which the statements associated with each else is an if. It takes the following general form:



```
if ( condition 1)
    statement-1;

else if ( condition 2)
        statement-2;

   else if ( condition 3)
           statement-3;

       else if ( condition n)
              statement-n;
             else
              default-statement;
    statement-x;
```

This construct is known as the else if ladder. The conditions are evaluated from the top (of the ladder) downwards. As soon as true condition is found the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladders). When all the conditions are false then the final else containing default statement will be executed.

Flowchart



Program

```
// Program to classify a number as positive, negative or zero.
#include<stdio.h>
main()
{
        int num;
        printf("\n Enter any number : ");
        scanf("%d", &num);
        if(num==0)
                printf("\n The value is equal to zero");
        else if(num>0)
                printf("\n The number is positive");
        else
                printf("\n The number is negative");
        return 0;
}
```

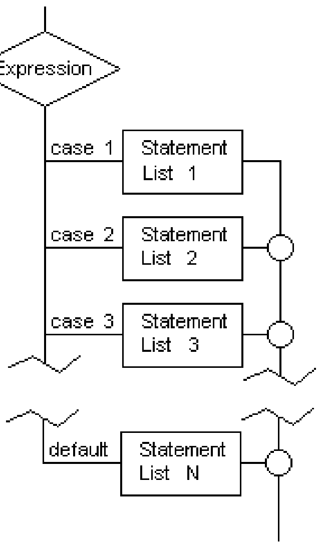
3.5 SWITCH Statement:

We have seen that when one of the many alternatives is to be selected we can use an if statement to control the selection. However the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times it may confuse even the person who designed it. Fortunately, C has a built in multi-way decision statement known as switch. The switch statement tests the value of a given variable (or expression) against the list of case values and when a match is found a block of statement associated with that case is executed.

The general form if given below:

```
switch ( expression )
{
        case value-1:
                        block-1
                        break;
        case value-2:
                        block-2
                        break;
        ………………
        ………………
        default:
                default-block
}
statement-x;
```

Flowchart



Program

```
void main()
{
char grade;
printf("Enter the grade of student");
scanf("%c",&grade);
switch(grade)
{
                case 'A':
                        printf("\n Excellent");
                        break;
                case 'B':
                        printf("\n Good");
                        break;
                case 'C':
                        printf("\n Fair");
                        break;
                default:
                        printf("\n Invalid Grade");
                        }  }
```

3.6 GOTO Statement:

So far we discussed ways of controlling the flow of execution based on certain specified conditions. C supports the goto statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the goto statement in a highly structured language like C, there may be occasions when the use of goto might be desirable.

The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by colon. The label is placed immediately before the statement where the control is to be transferred. The general forms of goto and label statements are shown below:

goto label;
…………
…………
lablel:
statement;

a) Forward jump

label:
statement;
…………
…………
goto label;

b) Backward jump

The label can be anywhere in the program either before or after the goto label; statement.

During running a program when a statement like
goto begin;
is met, the flow of control will jump to the statement immediately following the label **begin:.** This happens unconditionally.
If the label is placed after the goto statement then it is called a forward jump and in case it is located before the goto statement, it is said to be a backward jump.

A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

```
main()
{
    double x, y;
    read:
    scanf("%f", &x);
    if (x < 0) goto read;
    y = sqrt(x);
    printf("%f %f\n", x, y);
    goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statement and the other to skip any further computation when the number is negative.

Such infinite loops should be avoided in programming.
Another use of the **goto** statement is to transfer the control out of a loop (or nested loops) when certain peculiar conditions are encountered. Example:

```
----
----
while (----)
{
  for (----)
  {
  ----
  ----
  if (----)goto end_of_program;
  ----
  }                                  Jumping
----                                 out of
----                                 loops
}
end_of_program:
```

We should try to avoid using **goto** as far as possible. But there is nothing wrong, if we use it to enhance the readability of the program or to improve the execution speed.

Program

```
int num, sum=0;
read:   // label for go to statement
        printf("\n Enter the number. Enter 999 to end : ");
        scanf("%d", &num);
        if (num != 999)
        {
                if(num < 0)
                        goto read;      // jump to label- read
                sum += num;
                goto read;              // jump to label- read
        }
        printf("\n Sum of the numbers entered by the user is = %d", sum);
```

3.7 Conditional Operator / Ternary Operator (? :):

The conditional operator is a combination of? and:, and it takes three operands. This operator is popularly known as the conditional or ternary operator. The general form of use of the conditional operator is as follows:

Conditional expression? expression1: expression2;

The conditional expression is evaluated first. If the result is nonzero, expression1 is evaluated and is returned as the value of the conditional expression. Otherwise expression2 is evaluated and its value is returned. For example the segment

```
if (x<0)
flag=0;
else
flag=1;
```

Can be written as

```
flag=(x<0)?0:1;
```

Program

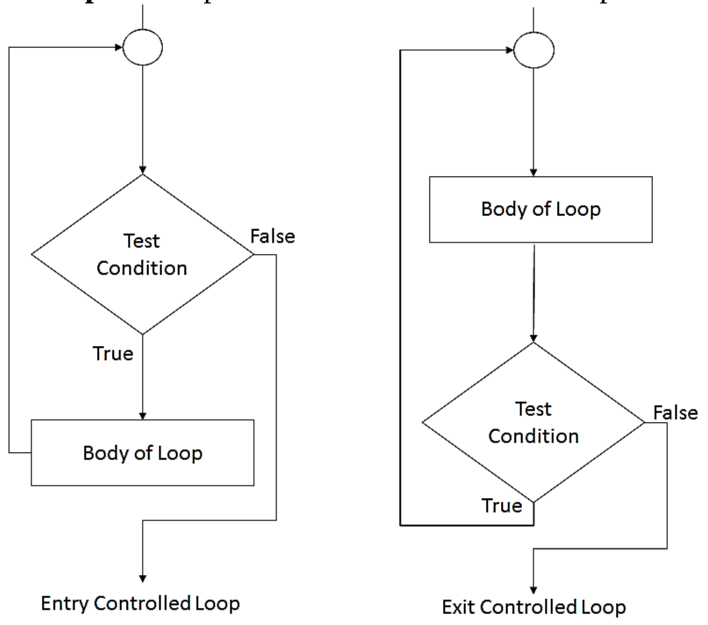
```
#include <stdio.h>
main()
{
  int a , b;
  a = 10;
  printf( "Value of b is %d\n", (a == 1) ? 20: 30 );
  printf( "Value of b is %d\n", (a == 10) ? 20: 30 );
}
```

3.8. Looping / Iterative Statements

Iterative statements are used to repeat the execution of a list of statements, depending on the value of an integer expression.

A program loop (or iteration) therefore consists of two segments, one is known as body of the loop and other is known as the control statements. Depending on the position of the control statement in a loop a control structure may be classified as:

a) **Entry controlled loop:** condition is tested before the start of the loop.
b) **Exit controlled loop:** test is performed at the end of the loop.



The entry controlled loop is also known as pre-test loop and exit controlled loop is known as post-test loop.
Iterative statements are used to repeat the execution of a list of statements, depending on the value of an integer expression. In this section, we will discuss all these statements.

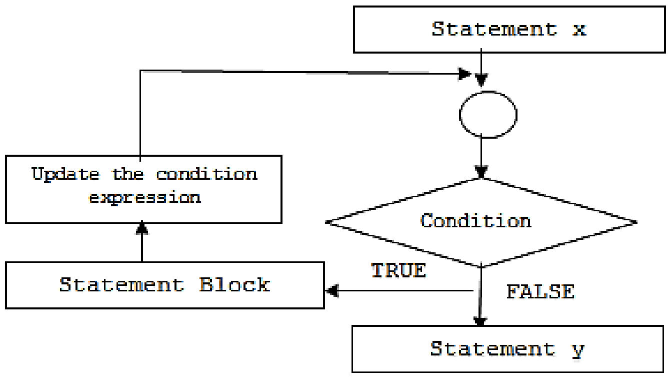
3.9 While loop
3.10 Do-while loop
3.11 For loop

A looping process in general would include the following 4 steps:
1. Initialization of counter
2. Test condition (check the value of counter)
3. Body of loop
4. Increment / Decrement (update counter)

3.9 While Loop

It is entry controlled loop. The while loop is used to repeat one or more statements while a particular condition is true. In the while loop, the condition is tested before any of the statements in the statement block is executed. If the condition is true, only then the statements will be executed otherwise the control will jump to the immediate statement outside the while loop block.We must constantly update the condition of the while loop.

```
statement x;
while (condition)
{
        statement_block;
}
statement y;
```



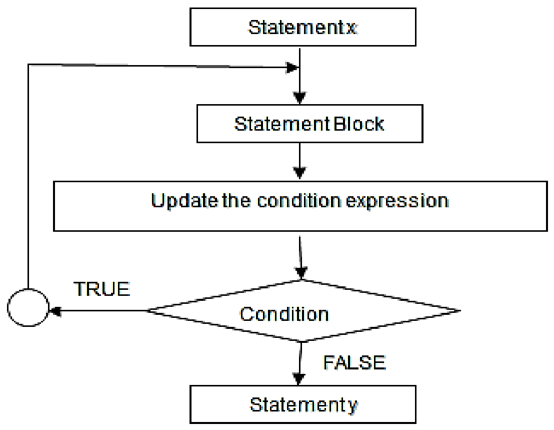
Program to print numbers from 0 to 10 using while loop

```
#include<stdio.h>
int main()
{
	int i = 0;
	while(i<=10)
	{
		printf("\n %d", i);
		i = i + 1;		// condition updated
	}
	return 0;
}
```

3.10 Do While Loop

The do-while loop is similar to the while loop. The only difference is that in a do-while loop, the test condition is tested at the end of the loop. So it is known as exit controlled loop. The body of the loop gets executed at least one time (even if the condition is false). The major disadvantage of using a do while loop is that it always executes at least once, so even if the user enters some invalid data, the loop will execute. Do-while loops are widely used to print a list of options for a menu driven program.

```
statement x;
do
{
	statement_block;
} while (condition);
statement y;
```



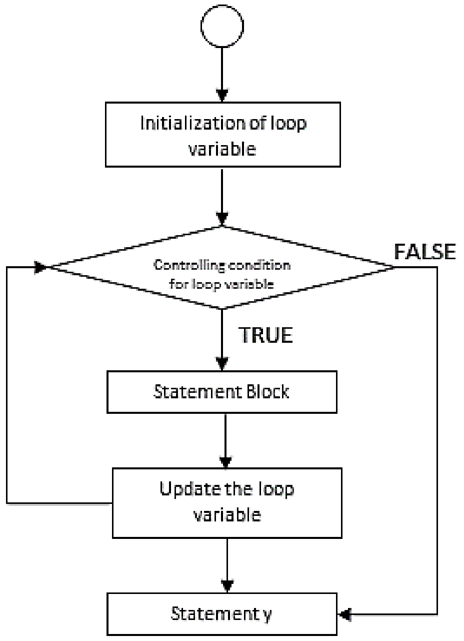
Program to print numbers from 0-10 using do-while loop

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

3.11 For Loop

It is a entry controlled loop. The for loop is used to repeat a task until a particular condition is true. The syntax of a for loop is as follows

```
        for (initialization; condition; increment/decrement/update)
        {
           statement block;
        }
statement y;
```


Initialization of loop variable
Controlling condition for loop variable
FALSE
TRUE
Statement Block
Update the loop variable
Statement y

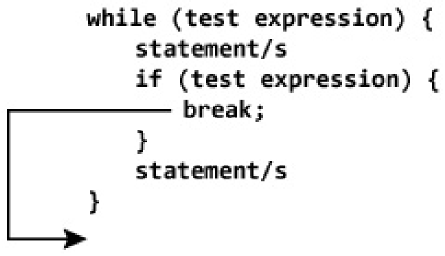
- When a for loop is used, the loop variable is initialized only once.
- With every iteration of the loop, the value of the loop variable is updated and the condition is checked. If the condition is true, the statement block of the loop is executed else, the statements comprising the statement block of the for loop are skipped and the control jumps to the immediate statement following the for loop body.
- Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting it to some other value like, i +=2, where i is the loop variable.

Look at the code given below which print first n numbers using a for loop.

Program
```
#include<stdio.h>
int main()
{
int i, n;
printf("\n Enter the value of n :");
scanf("%d", &n);
for(i=0; i<= n; i++)
{
printf("\n %d", i);
}
return 0;
}
```

3.12 Break statement
The break statement is used to terminate the execution of the nearest enclosing loop in which it appears. When compiler encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears. Its syntax is quite simple, just type keyword break followed with a semi-colon.

break;
In switch statement if the break statement is missing then every case from the matched case label to the end of the switch, including the default, is executed.



```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

Program

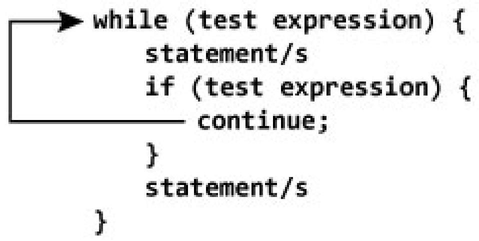
```
int i;
    for(i=1; i<= 5; i++)
        {       if (i==3)
                        break;
                printf("\t %d", i);
        }
```

Output

```
1
2
```

3.13 Continue Statement
The continue statement can only appear in the body of a loop. When the compiler encounters a continue statement then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest enclosing loop. Its syntax is quite simple, just type keyword continue followed with a semi-colon.
continue;

If placed within a for loop, the continue statement causes a branch to the code that updates the loop variable.



```
while (test expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
```

Program

```
int i;
    for(i=1; i<= 5; i++)
        {       if (i==3)
                        continue;
                printf("\t %d", i);
        }
```

Output

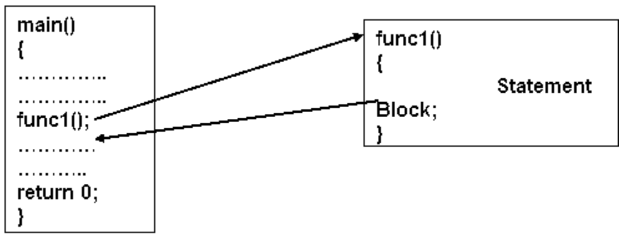
1
2
4
5

Functions

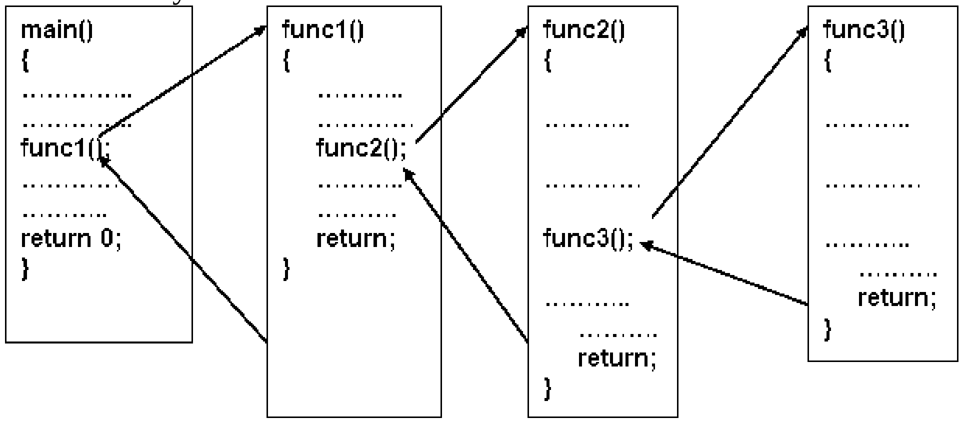
3.14 Introduction

C enables its programmers to break up a program into segments commonly known as functions, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task. Therefore, the program code of one function is completely insulated from that of other functions. Every function has a name which acts as an interface to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it.

In the fig, main() calls another function, func1() to perform a well-defined task. main() is known as the calling function and func1() is known as the called function. When the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling program.



Any function can call any other function.



3.15 Why do we need functions?

- Dividing the program into separate well defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work.
- Understanding, coding and testing multiple separate functions are far easier than doing the same for one huge function.

- If a big program has to be developed without the use of any function (except main()), then there will be countless lines in the main() .
- All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been prewritten and pre-tested, so the programmers use them without worrying about their code details. This speeds up program development.

3.16 Terminology of functions

- A function, f that uses another function g, is known as the calling function and g is known as the called function.
- The inputs that the function takes are known as arguments
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- Main () is the function that is called by the operating system and therefore, it is supposed to return the result of its processing to the operating system.

3.17 Function declaration

- Function declaration is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.
- The general format for declaring a function that accepts some arguments and returns some value as result can be given as:
  return_data_type function_name(data_type variable1, data_type variable2,..);
- No function can be declared within the body of another function.
  Example, float avg (int a, int b);

3.18 Function definition

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function
- When a function defined, space is allocated for that function in the memory.
- The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
    ………….
   statements
   ………….
   return( variable);
         }
```

- The no. and the order of arguments in the function header must be same as that given in function declaration statement.

3.19 Function Call

- The function call statement invokes the function.
- When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function.
- Once the called function is executed, the program control passes back to the calling function.
- Function call statement has the following syntax.

function_name(variable1, variable2, …);

Program

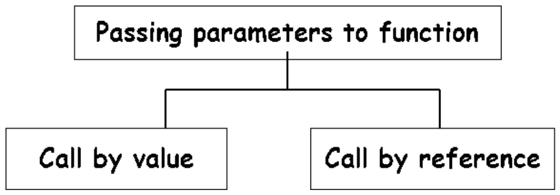
```
#include<stdio.h>
int sum(int a, int b);    // FUNCTION DECLARATION
int main()
{
        int num1, num2, total = 0;
        printf("\n Enter the first number : ");
        scanf("%d", &num1);
        printf("\n Enter the second number : ");
        scanf("%d", &num2);
        total = sum(num1, num2);              // FUNCTION CALL
        printf("\n Total = %d", total);
        return 0;
}
// FUNCTION DEFNITION
int sum ( int a, int b)            // FUNCTION HEADER
{                                  //  FUNCTION BODY
        return (a + b);
}
```

3.20 Return statement

- The return statement is used to terminate the execution of a function and return control to the calling function. When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- By default, the return type of a function is int.
- For functions that has no return statement, the control automatically returns to the calling function after the last statement of the called function is executed.

3.21 Passing parameters to the function

- There are two ways in which arguments or parameters can be passed to the called function.
- Call by value in which values of the variables are passed by the calling function to the called function.
- Call by reference in which address of the variables are passed by the calling function to the called function.



3.21.1 Call by value

In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.
If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

```
#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);
}
```

The output of this program is:

```
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 = 2
```

3.21.2 Call by reference

In call by reference, we declare the function parameters as references rather than normal variables.
To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to that parameter in the called function body will then be reflected in its value in the calling program.

```
#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);
}
```

The output of this program is:

```
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
```

3.22 Recursive functions

A recursive function is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.

Every recursive solution has two major cases, they are **base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function **recursive case**, in which first the problem at hand is divided into simpler sub parts.

Second the function calls itself but with sub parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Finding Factorial of a Number using Recursion

PROBLEM	SOLUTION
5!	5 X 4 X 3 X 2 X 1!
= 5 X 4!	= 5 X 4 X 3 X 2 X 1
= 5 X 4 X 3!	= 5 X 4 X 3 X 2
= 5 X 4 X 3 X 2!	= 5 X 4 X 6
= 5 X 4 X 3 X 2 X 1!	= 5 X 24
	= 120

Base case is when n=1, because if n = 1, the result is known to be 1

Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as

factorial(n) = n X factorial (n-1)

```
#include<stdio.h>
int Fact(int)
{       if(n==1)
                retrun 1;
        return (n * Fact(n-1));
}
main()
{       int num;
```

```
        scanf("%d", &num);
        printf("\n Factorial of %d = %d", num, Fact(num));
        return 0;
}
```

The Fibonacci series can be given as:

```
        0  1   1   2     3    5      8            13      21     34          55……
```

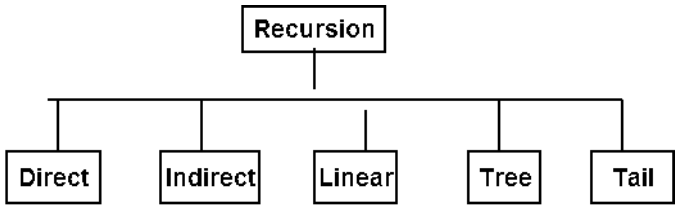
That is, the third term of the series is the sum of the first and second terms. On similar grounds, fourth term is the sum of second and third terms, so on and so forth. Now we will design a recursive solution to find the nth term of the Fibonacci series. The general formula to do so can be given as

```
main()
{       int n;
        printf("\n Enter the number of terms in the series : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
                printf("\n Fibonacci (%d) = %d", i, Fibonacci(i));
}
int Fibonacci(int num)
{       if(num == 0 || num==1)
                return num;
        return ( Fibonacci (num - 1) + Fibonacci(num – 2));
}
```

3.23 Types of recursion

Any recursive function can be characterized based on:

- whether the function calls itself directly or indirectly (direct or indirect recursion).
- whether any operation is pending at each recursive call (tail-recursive or not).
- the structure of the calling pattern (linear or tree-recursive).



3.23.1 Direct Recursion

A function is said to be *directly* recursive if it explicitly calls itself. For example, consider the function given below.

```
int Func( int n)
{
        if(n==0)
                retrun n;
        return (Func(n-1));
}
```

3.23.2 Indirect Recursion

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
                return n;
        return Func2(n);
}
```

```
int Func2(int x)
{
        return Func1(x-1);
}
```

3.23.3 Tail Recursion

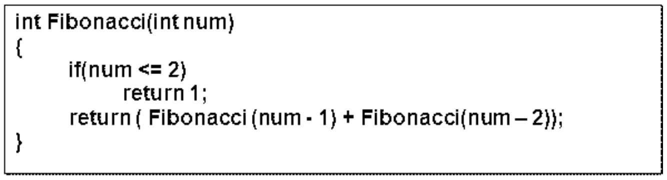
A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller. That is, when the called function returns, the returned value is immediately returned from the calling function.Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
int Fact(n)
{
 return Fact1(n, 1);
}
```

```
int Fact1(int n, int res)
{
        if (n==1)
           return res;
        return Fact1(n-1, n*res);
}
```

3.23.4 Linear and Tree Recursion

- Recursive functions can also be characterized depending on the way in which the recursion grows- in a linear fashion or forming a tree structure.
- In simple words, a recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function. For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to Fact.
- On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function. For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.



```
int Fibonacci(int num)
{
        if(num <= 2)
                return 1;
        return ( Fibonacci (num - 1) + Fibonacci(num – 2));
}
```

3.24 Pros and cons of recursion

- Pros: Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use

- Recursion represents like the original formula to solve a problem.
- Follows a divide and conquer technique to solve problems
- In some (limited) instances, recursion may be more efficient
- Cons: For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow and sometimes nasty.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counter part.
- It is difficult to find bugs, particularly when using global variables

Some Important Programs

Program: To calculate factorial of a number.

```
#include <stdio.h>
 int main()
{
  int c, n, fact = 1;
   printf("Enter a number to calculate it's factorial\n");
  scanf("%d", &n);
   for (c = 1; c <= n; c++)
    fact = fact * c;
   printf("Factorial of %d = %d\n", n, fact);
   return 0;
}
```

Program: To calculate sum of digits of a number.

```
#include <stdio.h>
 int main()
{
   int n, t, sum = 0, remainder;
    printf("Enter an integer\n");
   scanf("%d", &n);
    t = n;
    while (t != 0)
   {
      remainder = t % 10;
      sum       = sum + remainder;
      t         = t / 10;
   }
    printf("Sum of digits of %d = %d\n", n, sum);
    return 0;
}
```

Program: To reverse a number.

```
#include <stdio.h>
 int main()
{
   int n, reverse = 0;
    printf("Enter a number to reverse\n");
   scanf("%d", &n);
    while (n != 0)
   {
      reverse = reverse * 10;
      reverse = reverse + n%10;
      n       = n/10;
   }
    printf("Reverse of entered number is = %d\n", reverse);
    return 0;
}
```

Program: To check whether a given number is armstrong or not.

```
#include <stdio.h>
int main()
{
   int number, sum = 0, temp, remainder;
    printf("Enter an integer\n");
   scanf("%d",&number);
    temp = number;
    while( temp != 0 )
   {
      remainder = temp%10;
      sum = sum + remainder*remainder*remainder;
      temp = temp/10;
   }
    if ( number == sum )
      printf("Entered number is an armstrong number.\n");
   else
      printf("Entered number is not an armstrong number.\n");
    return 0;
}
```

Program: To generate Fibonacci Series.

```
#include<stdio.h>
 int main()
{
   int n, first = 0, second = 1, next, c;
    printf("Enter the number of terms\n");
   scanf("%d",&n);
   printf("First %d terms of Fibonacci series are :-\n",n);
    for ( c = 0 ; c < n ; c++ )
   {
      if ( c <= 1 )
         next = c;
      else
      {
         next = first + second;
         first = second;
         second = next;
      }
      printf("%d\n",next);
   }
    return 0;
}
```

Program: To calculate LCM and GCD.

```
#include <stdio.h>
 int main()
 {
  int a, b, x, y, t, gcd, lcm;
   printf("Enter two integers\n");
  scanf("%d%d", &x, &y);
   a = x;
  b = y;
   while (b != 0)
{
    t = b;
    b = a % b;
    a = t;
  }
  gcd = a;
  lcm = (x*y)/gcd;
   printf("Greatest common divisor of %d and %d = %d\n", x, y, gcd);
  printf("Least common multiple of %d and %d = %d\n", x, y, lcm);
   return 0;
}
```

Question

2016-17

1. What are functions? What is the advantage of using multiple functions in a program?
2. Distinguish between int main() and void main()?
3. What is recursion? Write a program in C to generate Fibonnaci series.
4. Differentiate between:
    a. Actual and formal arguments
    b. Global and extern variables (Unit 2 and Unit 3 both)
5. Write a program to print all prime numbers between 1 to 300
6. What do you mean by parameter passing? Discuss various types of parameters passing mechanism in C with examples.
7. Write a program to print following pattern:
```
A
AB
ABC
ABCD
ABCDE
```
8. Write a program to check whether a given number is Armstrong or not Like $153=1^3+5^3+3^3$
9. A five digit positive integer is entered through the keyboard. Write a C function to calculate sum of digits of a 5 digit number:
    a. Without using recursion
    b. Using recursion

2015-16

1. Write a program to check whether the given character is in uppercase, lower case or non-alphabetic character.
2. What are the disadvantages of if-else-if ladder?
3. What are the principles of recursion? Explain in detail.
4. Write a program in 'C' that will read a positive number from the keyboard and print it in reverse order.
   E.g., 24578   output: 87542
5. What do you mean by parameter passing mechanism?
6. Write a program in C to print following pattern:
```
A  B  C  D  E  F  G  F  E  D  C  B  A
A  B  C  D  E  F     F  E  D  C  B  A
A  B  C  D  E           E  D  C  B  A
A  B  C  D                 D  C  B  A
A  B  C                       C  B  A
A  B                             A  B
A                                   A
```

7. What are different types of functions? Write a program in C to short list of names of students in an ascending order.
8. Write difference between call by value and call by reference with suitable example.

2014-15

1. Give the loop statement to print the following sequence of integer
   -6 -4 -2 0 2 4 6
2. What are the main principles of recursion
3. What is the role of SWITCH statement in C programming language? Explain with example.
4. Distinguish between actual and formal arguments.
5. Describe call by value and call by reference with example.
6. Write a program in C language to generate the Fibonacci series.
7. Describe about the types of looping statements in 'C' with necessary syntax.
8. Write a C program to find the multiplication of two matrices.
9. What are the types of function? Write a C program to find the factorial of a given number using recursion.
10. What is the difference between break and continue? Describe the structure of switch-case with neat example.

2013-14

1. What are the different types of functions? Write a program in C to short list of names of students in an ascending order.
2. Write a program to print following pattern
   1
   2 3
   4 5 6
   7 8 9 10
3. Define recursive function. Write a program in C to generate Fibnocii series (0 1 1 2 3 5 8 13…) using recursive function.
4. Write a C program to find the sum of individual digits in a five digit number.
5. Write the difference between call by value and call by reference with suitable example.
6. Write a program to find greatest among three numbers using conditional operator.
7. Differentiate between nested-if and switch statements in 'C' with example.
8. Write a program in 'C' to sort list of 10 integers in an ascending order.
9. Write a program to multiply the two matrices of MxN.

2012-13

1. Write the purpose and syntax of at least two iterative statements in C.
2. WAP to generate fabonacci series up to the last term less than 100. Also calculate sum and total count of the fabonacci numbers.
3. What is sorting? Give flowchart and algorithm to sort the integer numbers.
4. Given two matrices of 4x4. Write the functions sum_matrix() and multiply_matix() to add and multiply two matices.
5. Differentiate between call by value and call by reference.
6. Write a program to calculate GCD.
7. WAP to calculate the multiplication of all the digits of a 5 digit number.
8. Write a program which stores the marks of N students in integer array. Calculate average marks obtained and deviation from the average.
9. Explain ternary operator.
10. Define user defined and library functions.

11. What are iterative control statements? Differentiate between while loop and do-while loop.
12. Define recursion. Give its advantage. Which data structure is used to implement recursion? Write a program to calculate factorial of a number using recursion.
13. Write a program to check whether a number is perfect number or not. If the sum of factor is equal to number itself then it is a perfect number. E.g Factor of 6 are 1, 2, 3 whose sum 1+2+3=6.
14. Write a program to find the prime numbers between the given range.

2011-12

1. Write a program to generate following pattern.
   ```
   A
   B A
   A B A
   B A B A
   A B A B A
   ```
2. Write a program o read five digit number if it is even then add the digits otherwise multiply them.
3. Write a program to generate the given series upto less than 200.
   1 – 4 + 9 – 16 + 25 ……….
4. Write a program to read age of 100 persons and count the number of persons in the age group 50 to 60. Use for and continue statements.
5. Write a program to check whether a number is even or odd without else option.