UNIT 2:

Conditional Statements and Loops

Control Statements

The control statements enable us to specify the order in which the various instructions in a program are to be executed by the computer. They determine the flow of control in a program.

There are four types of control statements in C. They are:

- Sequence Control Statements
- Decision Control Statements or Conditional Statement.
- Case Control Statement
- Repetition or Loop Control Statements

Sequence Control statements ensure that the instructions in the program are executed in the same order in which they appear in the program.

Decision and Case Control statements allow the computer to take decision as to which statement is to be executed next.

The Loop Control statement helps the computer to execute a group of statements repeatedly.

Conditional Statement

C has two major decision-making statements.

- **If_else statement**
- **Switch statement**

If - else Statement

The if - else statement is a powerful decision-making tool. It allows the computer to evaluate the expression. Depending on whether the value of expression is "True" or "False" certain group of statements are executed.

The syntax of if else statement is:

```
if (condition is true)

     statement 1;

 else

     statement 2; .
```

The **condition** following the keyword is always enclosed in parenthesis. If the condition is true, statement in then part is executed, that is, statement 1 otherwise statement 2 in else part is executed.

Example:

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

void main()
      {
        Int N;
        clrscr();
        printf("Enter the Number :");

         scanf("%d",&N);

      if(N%2 == 0)
           printf("  Given  Number  %d  is  ODD
 ",N);
    else
          printf(" Given Number %d is ODD ",N);

      getch();
    }
```

Output :

```
                    Enter the Number:
      5             Given Number 5 is
      ODD
```

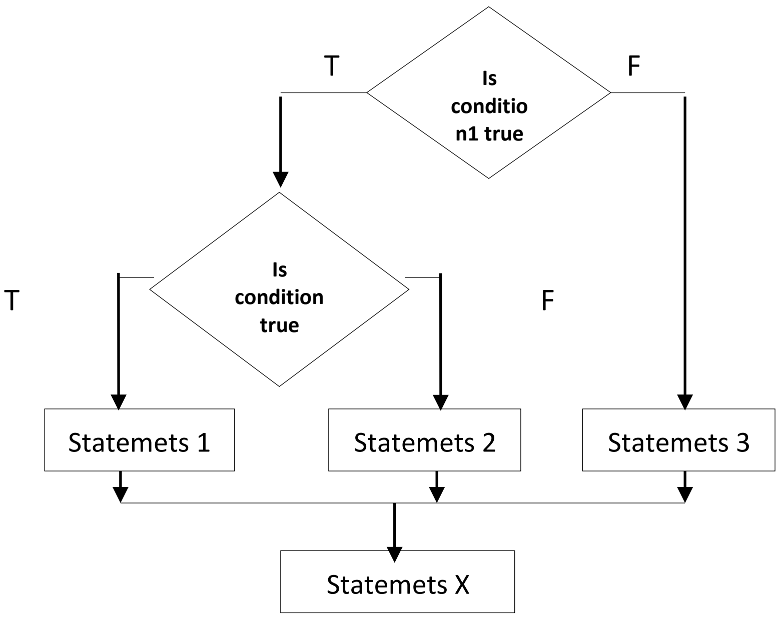
Nested If-Else Statement:

It is a conditional statement which is used when we want to check more than 1 conditions at a time in a same program. The conditions are executed from top to bottom checking each condition whether it meets the conditional criteria or not. If it found the condition is true then it executes the block of associated statements of true part else it goes to next condition to execute.

Syn tax:

```
if(condition 1)

{           if(condition 2)
            {
                        Statements 1;
            }
      else
            Statements 2;
            }
}
else
{
            Statements3;
}
```



In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and again checks the next condition. If it is true then it executes the block of statements associated with it else executes else part.

```
#include
<stdio.h>
#include
<conio.h> void
main()
{       int    no;
        clrscr();
        printf("\n     Enter     Number     :");
        scanf("%d",&no);
        if(no>0)
        {
                printf("\n\n Number is greater than 0 !");
        }
        else
        {
                if(no==0)
                {
                        printf("\n\n It is 0 !");
                }
                else
                {
                        printf("Number is less than 0 !");
                }
        }
        getch();
}
```

Output :

```
Enter Number : 0

It is 0!
```

else if Ladder

In C programming language the else if ladder is a way of putting multiple ifs together when multipath decisions are involved. It is a one of the types of decision making and branching statements. A multipath decision is a chain of if’s in which the statement associated with each else is an if. The general form of else if ladder is as follows -

```
if ( condition 1)
{
statement
- 1;
}        else if
(condtion 2)
    {
        statement - 2;
    }
        else if ( condition n)
```

```
            {
                statement - n;
            }
                else
                {
                     default statment;
                }
statement-x;
```

This construct is known as the else if ladder. The conditions are evaluated from the top of the ladder to downwards. As soon as a 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 ladder). When all the n conditions become false, then the final else containing the default statement will be executed.

Example :

```
#include
<stdio.h>
#include
<conio.h> void
main()
{
        int    no;
        clrscr();
          printf("\n Enter Number :");
          scanf("%d",&no);
          if(no>0)
          {
                    printf("\n Number is greater than 0 !");
          }
          else if(no==0)
           {
                    printf("\n It is 0 !");
          }
          else
          {
                    printf("Number is less than 0 !");
          }
          getch();
}
```

Output :

```
Enter Number : 5

Number is greater than 0 !
```

Switch case Statement :

This is a multiple or multiway branching decision making statement.

When we use nested if-else statement to check more than 1 conditions then the complexity of a program increases in case of a lot of conditions. Thus, the program is difficult to read and maintain. So to overcome this problem, C provides 'switch case'.
Switch case checks the value of an expression against a case values, if condition matches the case values then the control is transferred to that point.

Syntax:

```
switch(expression)
{
        case expr1:

statements;
break;          case
expr2:
                statements;
                break;

-----------------------------
-----------------------------
-------------------------
----    case exprn:
                statements;
                break;
        default:
                statements;
}
```

In above syntax, switch, case, break are keywords.
expr1, expr2 are known as 'case labels.'
Statements inside case expression need not to be closed in braces.
break statement causes an exit from switch statement.
default case is optional case. When neither any match found, it executes.

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

```
#include <conio.h>
void main()
{
            int no;
            clrscr();
        printf("\n   Enter   any   number   from   1   to   3   :");
        scanf("%d",&no);
            switch(no)
            {
                    case 1:
                                printf("\n\n It is 1 !");
                        break;
                case 2:
                                printf("\n\n It is 2 !");
                        break;
                case 3:
                                printf("\n\n It is 3 !");
                        break;
                default:
                                printf("\n\n Invalid number !");
            }
            getch();
}
```

Output :

```
            Enter any number from 1 to 3 : 3

            It is 3 !
```

* *Rules for declaring switch case :*

- The case label should be integer or character constant.
- Each compound statement of a switch case should contain break statement to exit from case.
- Case labels must end with (:) colon.

* *Advantages of switch case :*

- Easy to use.
- Easy to find out errors.
- Debugging is made easy in switch case.
- Complexity of a program is minimized.

Looping Statements / Iterative Statements :

A ' loop' is a part of code of a program which is executed repeatedly.
A loop is used using condition. The repetition is done until condition becomes condition true.
A loop declaration and execution can be done in following ways.

- o Initialize loop with declaring a variable. o Check condition to start a loop o Executing statements inside loop.
- o Increment or decrement of value of a variable.

**** Types of looping statements:***

Basically, the types of looping statements depends on the condition checking mode. Condition checking can be made in two ways as : Before loop and after loop. So, there are 2(two) types of looping statements.

- Entry controlled loop
- Exit controlled loop

1. Entry controlled loop:
In such type of loop, the test condition is checked first before the loop is executed. Some common examples of this looping statements are : o while loop o for loop

2. Exit controlled loop :
In such type of loop, the loop is executed first. Then condition is checked after block of statements are executed. The loop executed atleat one time compulsarily. Some common example of this looping statement is : o do-while loop

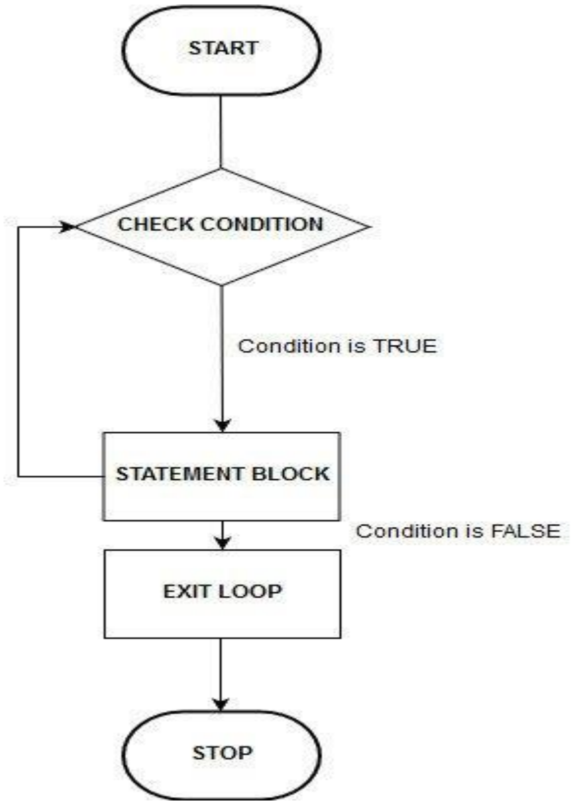
For *loop* :

This is an entry controlled looping statement.
In this loop structure, more than one variable can be initialized. One of the most important feature of this loop is that the three actions can be taken at a time like variable initialization, condition checking and increment/decrement. The for loop can be more concise and flexible than that of while and do-while loops.

Syntax:

```
for(counter initialization ; test-condition ; modify counter)
    {
                statements;
    }
```

In above syntax, the given three expressions are separated by ';' (Semicolon)



Features :

- More concise o Easy to use o Highly flexible o More than one variable can be initialized. o More than one increments can be applied.
- More than two conditions can be used.

Example :

```
      #include <stdio.h>
      #include <conio.h>
      void main()
      {
       int a;
      clrscr();

   for(i=0;i<5;i++)
       {
           printf("MCMT !\t");  // 5 times
       }  getch();
      }
```

Output :

```
MCMT            MCMT            MCMT            MCMT            MCMT
```

More About for Loop

The for loop in C has several capabilities that are not found in other loop constructs. More than one variable can be initialized at a time in the for statement.

```
The Statement                p = 1;
                         for (n = 0; n < 17; + + n)
can be rewritten as      for (p = 1, n = 0; n < 17; ++ n)
```

The increment section may also have more than one part as given in the following

example:

```
for (n = 1, m = 50; n < = m; n = n+1, m = m-j)
      {
        p = m/n;


       printf ("%d %d %d\n",
       n, m, p):
      }
```

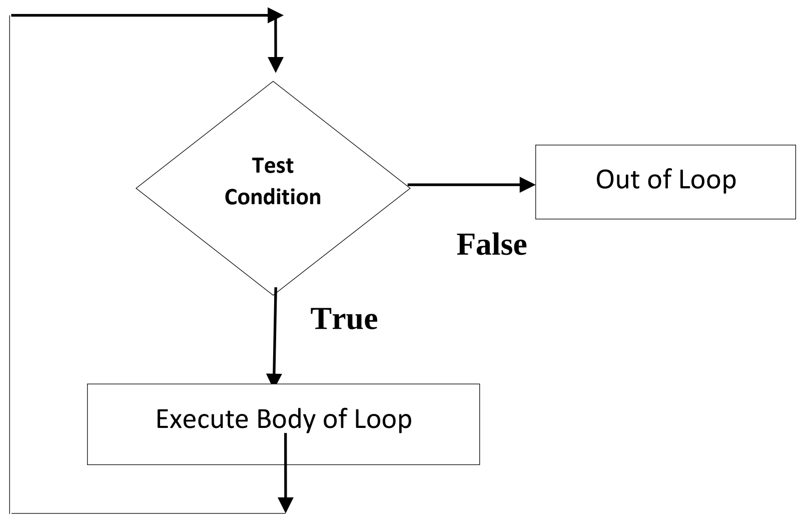
While loop :

This is an entry controlled looping statement. It is used to repeat a block of statements until condition becomes true.

Syntax:

```
while(condition)
{
           statements;
           increment/decrement;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the loop and executes the block of statements associated with it. At the end of loop increment or decrement is done to change in variable value. This process continues until test condition satisfies.



Program :

```
#include <stdio.h>
#include <conio.h>
void main()
{
 int a;
clrscr();
a=1;
while(a
<5)
{
                printf("MCMT \t");
        a+=1     // i.e. a = a + 1
 }
getc
h();
}
```

Output :

```
MCMT        MCMT        MCMT        MCMT        MCMT
```

Do-While loop :

This is an exit controlled looping statement.

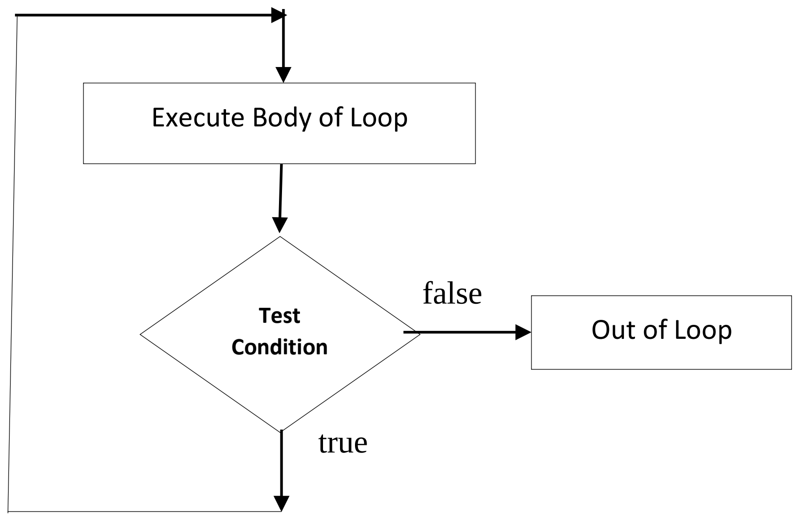
Sometimes, there is need to execute a block of statements first then to check condition. At that time such type of a loop is used. In this, block of statements are executed first and then condition is checked.

Syntax:

```
do
{
        statements;
        (increment/decrement);
}while(condition);
```

In above syntax, the first the block of statements are executed. At the end of loop, while statement is executed. If the resultant condition is true then program control goes to evaluate the body of a loop once again. This process continues till condition becomes true. When it becomes false, then the loop terminates.

Note: The while statement should be terminated with ; (semicolon).



Program :

```
#include <stdio.h> #include
<conio.h> void main()
{
  Int a;
  Clrscr();
   a=1;
   do  {
         printf("MCMT\t");  // 5 times
          a+=1;      // i.e. a = a + 1
        }while(a<=5);
getch();
              }
```

Output : MCMT MCMT MCMT MCMT MCMT

Infinite loop :

A looping process, in general, includes the following four steps:

- Setting of a counter.
- Execution of the statements in the loop.
- Testing of a condition for loop execution.
- Incrementing the counter.

The test condition eventually transfers the control out of the loop. In case, due1 to some reasons, if-it does not do so, the control sets up an infinite loop and the loop body is executed over and over again. Such infinite loops should be avoided. Ctrl+C or Ctrl+Break are used to terminate the program caught in an infinite loop. Two examples of infinite loop are given below:

Example :

```
     #include<stdio.h>
       void main()
              {
                     int i=1;
                     while(i<=10)
                     {
                       Printf(" i= %d\n",i);
                     }
              }
```

This program will never terminate as variable i will always be less than 10. To get the loop terminated, an increment operation (i + +) will be required in the loop.

Nested Loops :

Loops within loops are called nested loops. An overview of nested while, for and do .. while loops is given below:

Nested while :

It is required when multiple conditions are to be tested.

The syntax of nested while is: while (condition 1)

```
            { ----------

                        while (condition 2)
                          { ----------

                            while (condition
                                   n)
                                   { ----------


                                   }
                           }
                  }
```

For example:

```
  #include<stdio.h>
  #include<conio.h>
  void main()
  {
     int  i = 1, N;
     clrscr();
      while( i<= 5 )
     {
         N=1;
           while( N<=5)
          {

               printf("  %d  " , N);

          }

         printf( " \n");

     }

  }
```

Ouput :

```
1  2  3  4  5

1  2  3  4
5 1  2  3  4
```

```
5 1  2  3  4
5

1  2  3  4  5
```

Nested for :

It is used when multiple set of iterations are required. The syntax of nested for is:

```
for (   ;     ;    )
{ ----------
    ----------------
      ---------------
       for  (     ;     ;    )
       { ----------
             ----------
               for  (     ;     ;    )
               { ----------
                    ----------------
                        ---------------
               }
       }
  }
```

For example:

```
  #include<stdio.h >
  #include<conio.h>
  void main()
 {
    int  i , N;
    clrscr();

   for(  i=1 ; i<= 5 ; i++ )
   {
        for(  N=1 ;N<= 5 ; N++ )
         {
                printf(“  %d  ” , N);
         }
      printf( “ \n”);
   }
 }
```

Ouput :

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

The break Statement

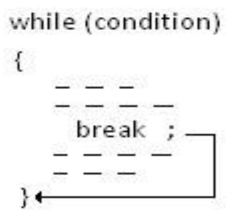
The break statement is used to terminate loops or to exit from a switch. When break is encountered inside any C loop, control automatically passes to the first statement after the loop. It can be used within a while, a do-while, a for loop or a switch statement. The break statement is written simply as break; without any embedded expression of statements.

Sometimes, it is necessary to exit immediately from a loop as soon as the condition is satisfied. The statements after break statement are skipped.

Syntax :

```
break;
```

Figure :



Example :

```
#include   <stdio.h>
#include <conio.h>
void main()
{
 int   i;
clrscr()
;
 for(i=1; i<=20 ; i++)
 {
            if(i>5)
            break;
            printf("%d",i);  // 5 times only
 }
printf(" \nOut of loop");
 getch();
}
```

Output :

```
12345
Out of loop
```

Continue Statement :

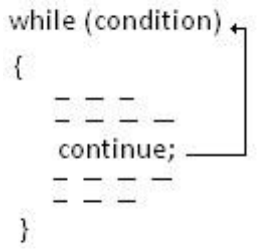
Sometimes, it is required to skip a part of a body of loop under specific conditions. So, C supports 'continue' statement to overcome this anomaly.

The working structure of 'continue' is similar as that of that break statement but difference is that it cannot terminate the loop. It causes the loop to be continued with next iteration after skipping statements in between. Continue statement simply skipps statements and continues next iteration.

Syntax :

```
continue;
```

Figure :



Example:

```
#include    <stdio.h>
#include <conio.h>
void main()
{
        int i;
        clrscr();
        for(i=1; i<=10; i++)
        {
                if( i>=6 && i <=8 )
                continue;
                printf("\t%d",i);  // 6 to 8 is omitted
         }
```

```
            getch();
}
```

Output :

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

Unconditional branching

Unconditional branching is when the programmer forces the execution of a program to jump to another part of the program. Theoretically, this can be done using a good combination of loops and if statements. In fact, as a programmer, you should always try to avoid such unconditional branching and use this technique only when it is very difficult to use a loop.

We can use "goto" statements (unconditional branch statements) to jump to a label in a program.
The **goto statement** is used for unconditional branching or transfer of the program execution to the labeled statement.

The goto Statement

C supports the **goto** statement to branch unconditionally from one point to another in the program. A goto statement breaks the normal sequential execution of the program. 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 a colon. The label is placed immediately before the statement where the control is to be transferred.

It is a well known as "**jumping statement**". It is useful to provide branching within a loop. When the loops are deeply nested at that if an error occurs then it is difficult to get exited from such loops. Simple break statement cannot work here properly. In this situation, goto statement is used.

The general forms of goto and label statements are:

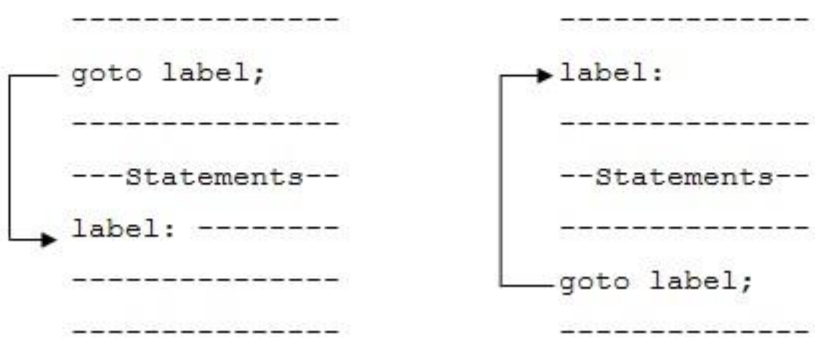
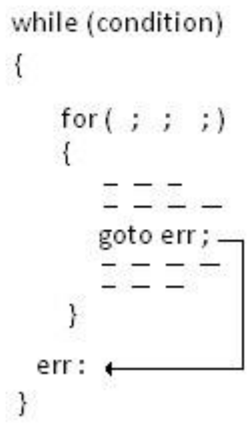


Figure :



Example :

```
 #include <stdio.h>
#include <conio.h>
 void main()
{           int i=1, j;
            clrscr();
            while(i<=3)
            {
                        for(j=1; j<=3; j++)
                                    {
                                                 printf(" I=%d \t J=%d \n", i , j);
                                                if(j==2)
                                                goto stop;
                                    }
                                    i = i + 1;
                        }

                        stop:   printf("\n Exited !");
```

```
    getch();
}
```

Output :

```
I=1      J=1
I=1      J=2

Exited!
```

Structured Programming

Structured programming (sometimes known as *modular programming*) is a programming paradigm that facilitates the creation of programs with readable code and reusable components. All modern programming languages support structured programming, but the mechanisms of support, like the syntax of the programming languages, varies.

Structured programming encourages dividing an application program into a hierarchy of modules or autonomous elements, which may, in turn, contain other such elements. Within each element, code may be further structured using blocks of related logic designed to improve readability and maintainability.

Modular programming, which is today seen as synonymous with structured programming, emerged a decade later as it became clear that reuse of common code could improve developer productivity. In modular programming, a program is divided into semi-independent modules, each of which is called when needed. C is called structured programming language because a program in c language can be divided into small logical functional modules or structures with the help of function procedure.

Advantages of structured programming

The primary advantages of structured programming are:

1. It encourages top-down implementation, which improves both readability and maintainability of code.
2. It promotes code reuse, since even internal modules can be extracted and made independent, residents in libraries, described in directories and referenced by many other applications.
3. It's widely agreed that development time and code quality are improved through structured programming.
4. It is user friendly and easy to understand.
5. Similar to English vocabulary of words and symbols.
6. It is easier to learn.

7. They require less time to write.
8. They are easier to maintain.
9. These are mainly problem oriented rather than machine based.
10. Program written in a higher level language can be translated into many machine languages and therefore can run on any computer for which there exists an appropriate translator.
11. It is independent of machine on which it is used i.e. programs developed in high level languages can be run on any computer.

Disadvantages of structured programming

The following are the disadvantages of structured programming:

1. A high level language has to be translated into the machine language by translator and thus a price in computer time is paid.
2. The object code generated by a translator might be inefficient compared to an equivalent assembly language program.
3. Data types are proceeds in many functions in a structured program. When changes occur in those data types, the corresponding change must be made to every location that acts on those data types within the program. This is really a very time consuming task if the program is very large.
4. Let us consider the case of software development in which several programmers work as a team on an application. In a structured program, each programmer is assigned to build a specific set of functions and data types. Since different programmers handle separate functions that have mutually shared data type. Other programmers in the team must reflect the changes in data types done by the programmer in data type handled. Otherwise, it requires rewriting several functions.