INPUT OUTPUT STATEMENTS

getchar

***getchar** is a function for reading characters one at a time from the keyboard.*

The general format of *getchar*:

variable_name = getchar();

When this statement is encountered, the computer waits until a key is pressed and then assigns the typed character as a value to *getchar* function. Since *getchar* is used on the right-hand side of an assignment statement, the character value of *getchar* is assigned to the variable name on the left.

Example: *char name;*

name = getchar();

putchar

***putchar** is a function for writing characters one at a time to the terminal.*

The general format :

putchar (variable_name);

Where *variable_name* is a *char* type variable containing a character. This statement displays the character contained in the code at the terminal. For example, the statements

answer = 'Y';

putchar (answer);

will display the character Y on the screen.

FORMATTED OUTPUT (printf)

The general form of *printf* statement is

Printf ("control string", arg1, arg2,……argn);

The control string indicates how many arguments follow and what their types are. The arguments arg1, arg2,…..argn are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in **number, order** and **type** with the format specifications.

FORMATTED INPUT (scanf)

Formatted input refers to an input data that has been arranged in a particular format.

The general form of scanf is

scanf ("control string", arg1, arg2,…..argn);

The arguments arg1, arg2,….argn specify the address of locations where the data is stored.

Control string contains field specifications which direct the interpretation of input data. It may include:

- Field (or format) specification, consisting of the conversion character %, a data type character (or type specifier), and an optional number, specifying the field width.
- Blanks, tabs, or newlines.

Inputting Integer Numbers

The field specification for reading an integer number is:

% wd

The percent sign (%) indicates that a conversion specification follows. **'w'** is an integer number that specifies the field width of the number to be read. **'d'** known as data type character, indicates that the number to be read is in integer mode.

Example:

scanf ("%2d %5d", &num1, &num2);

DECISION MAKING AND BRANCHING STATEMENTS

The statements which branch the execution control are called branching statements. They are if, switch, goto, break and continue

if statement

The ***if*** statement is a powerful decision making statement and is used to control the flow of execution of statements. The ***if*** statement may be implemented in different forms depending upon the complexity of conditions to be tested. They are:

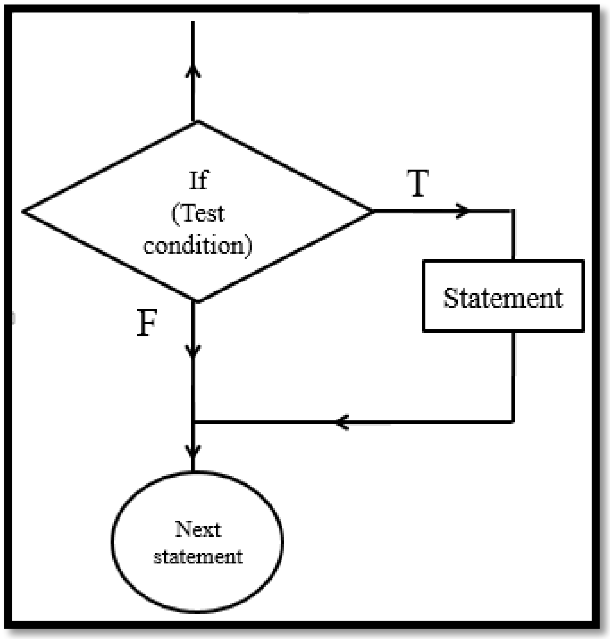
1. simple *if*   2. *if …else*   3. nested *if...else*   4. *else if ladder*

Simple if

The general format is:

```
if (test condition)
{
statement-block;
}
next statement ;
```

If the test condition is true, the statement block will be executed: otherwise the statement block- will be skipped and the execution will jump to the next statement. This is illustrated in the following fig.



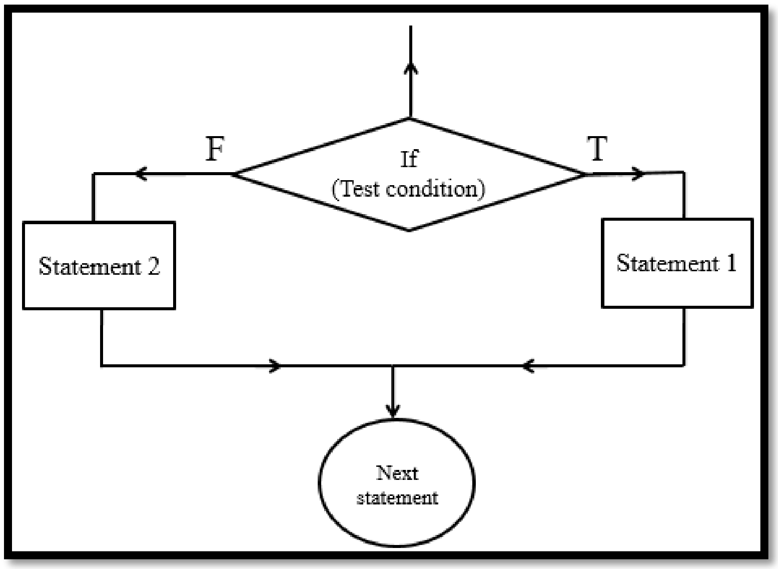
if ... else statement

The **if ...else** statement is an extension of the simple if statement.

The general form is:

```
if (test condition)
{
  true-block statement(s)
}
else
{
  false-block statement (s)
 }
next statement
```

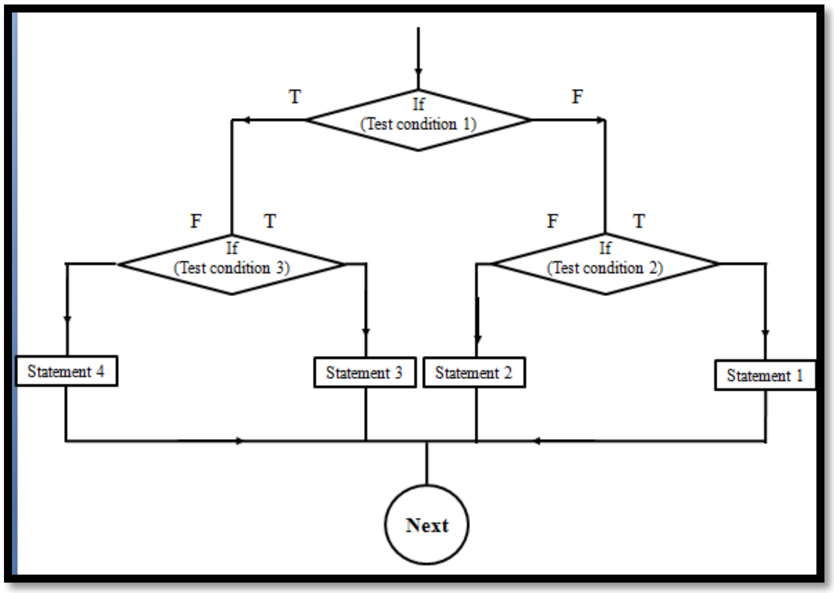
If the test expression is true, then the true-block statements are executed; otherwise, the false-block statements are executed.



Nested if...else

When a series of decisions are involved, we may have to use more than one **if...else** statement in nested form as follows:

```
if (test-condition 1)
  {
      if (test-condition 2)
           {
              statement-1
            }
      else
           {
             statement-2
            }
else
      if (test condition)
           {
             statement-3
           }
      else
          {
             statement - 4
          }
  }
```

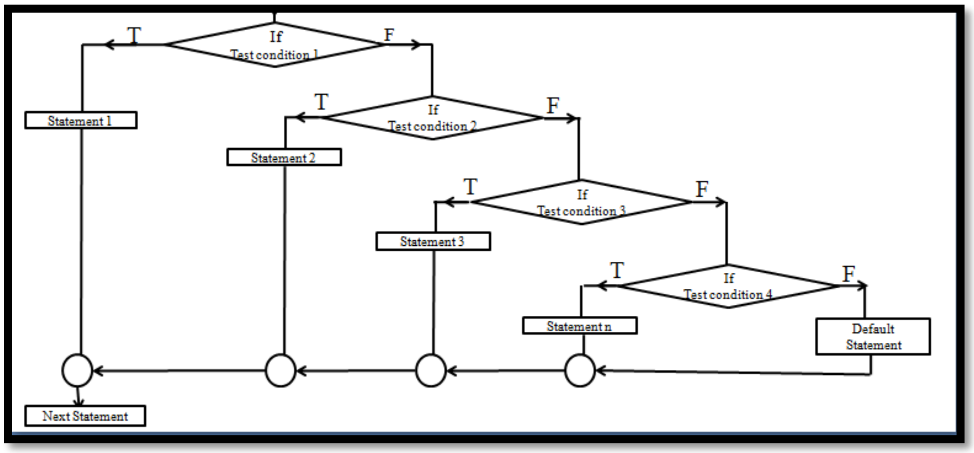


else if ladder

It is a multi alternative selection structure. The 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;
next statement.
```

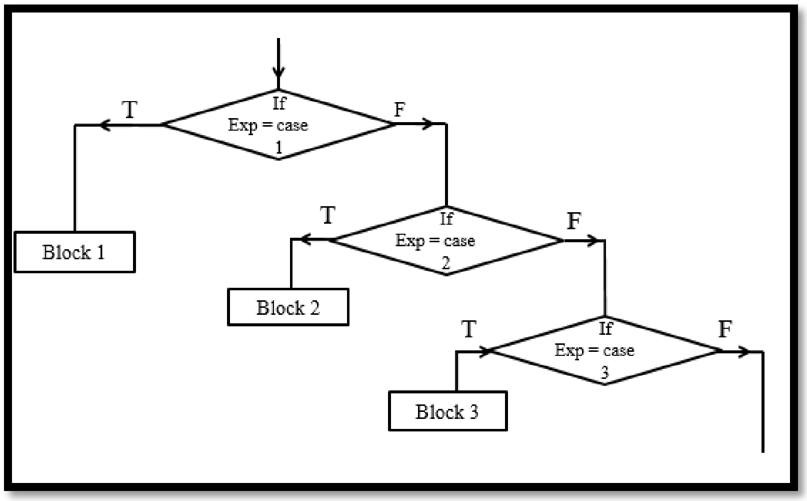
In this statement, each ‘else’ is associated with an ‘ if ’



The switch statement

When one of the many alternatives is to be selected, the switch statement is used. The switch statement tests the value of the given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form is:

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



The *goto* statement

The *goto* statement is used to branch unconditionally from one point to another in 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.

The general format of goto statement is

```
goto label;
  -----------  -----------
  -----------  -----------
label:
statement;


label:
statement;
  -----------  -----------
  -----------  -----------
goto label;
```

Forward jump backward jump

The *label* can be anywhere in the program either before or after the **goto** statement. During running of 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. The goto breaks the normal sequential execution of the program. If the *label:* is placed before the statement **goto label**; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as *backward jump.* If the *label:* is placed after the **goto label;** some statements will be skipped and the jump is known as a forward jump.

The break statement

The break statement causes an exit from a loop (such as while or for loops) or a switch statement. The general form of break statement is **break;**

Example:

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

The break statement at the end of each block signals the end of a particular case and causes an exit from the *switch* statement. It transfers the control to the statement-x following the *switch* statement.

The continue statement

The *continue* statement causes the loop to be continued with the next iteration after skipping any statements following the *continue* statement. The continue statement tells the compiler "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is

continue;

The use of the *continue* statement in loops is illustrated in the following examples:

In ***while*** and do loops, ***continue*** causes the control to go directly to the test condition and then to continue the iteration process. But in the case of ***for*** loop, the increment section of the loop is executed before the test-condition is evaluated.

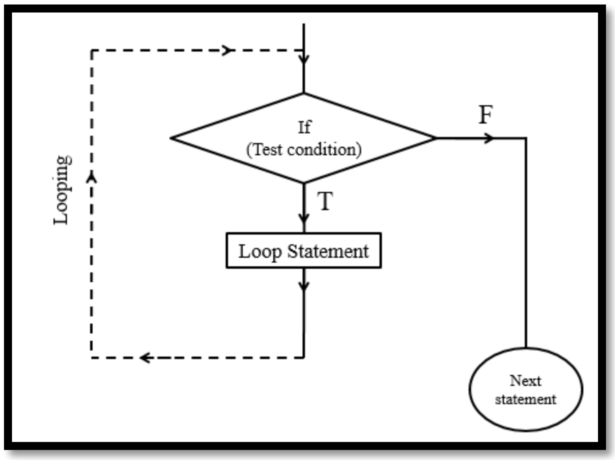
Looping statements

The while statement

The basic format of the while statement is:

```
while (test condition)
   {
     body of the loop
   }
```

The while is an ***entry controlled loop*** statement. The test condition is evaluated first and if the condition is true, then the body of the loop is executed. After the execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false then the control is transferred out of the loop.



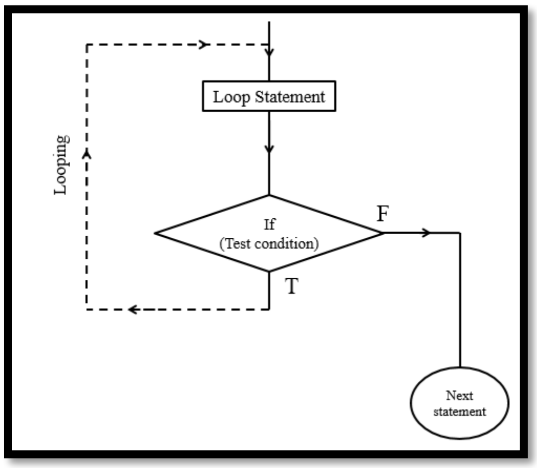
The do while statement

It is an ***exit controlled loop*** statement.

The general format is:

```
do
 {
   body of the loop
  }
while (test-condition);
```

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test-condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.



The *for* statement

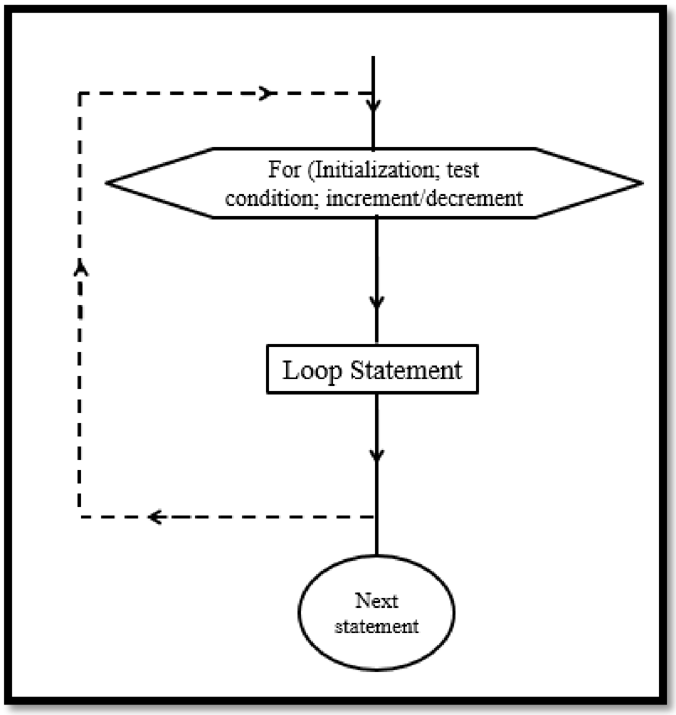
The *for* loop is an entry-controlled loop. The general form of the for loop is:

```
for ( initialization; test-condition; increment)
{
   body of the loop
 }
```

The execution of the ***for*** statement is as follows:

1. Initialization of the control variables is done first.
2. The value of the control variable is tested using the test-condition. If the condition is true, the body of the loop is executed; otherwise the loop is terminated.
3. When the body of the loop is executed, the control is transferred back to the ***for*** statement after evaluating the last statement in the loop. Now, the control variable is incremented and

the new value of the control variable is again tested. This process continues till the value of the control variable fails to satisfy the test-conditions.



Features of *for* loop

The *for* loop has several capabilities that are not found in other loop constructs. They are:

1. More than one variable can be initialized at a time in the *for* statement.
2. Like the initialization section, the increment section may also have more than one part.
3. The test-condition may have any compound relation and the testing need not be limited only to the loop control variable.
4. One or more sections such as initialization can be omitted.
5. Nesting of *for* loops, that is, one *for* statement within another *for* statement is allowed. For example, two loops can be nested as follows:

ARRAYS

An array is a group of related data items that share a common name. Arrays can be of any variable type.

ONE DIMENSIONAL ARRAYS

A list of items that can be given one variable name with only one subscript is called an one-dimensional array or single subscripted variable.

In C, single subscripted variable x can be expressed as x[0], x[1], x[2], x[3]………x[n-1]. If we want to represent a set of five numbers by an array variable x, then we can declare the variable x, as follows : *int x [5];*
The computer reserves five storage locations as shown below:

x[0]	
x[1]	
x[2]	
x[3]	
x[4]	

The subscript of an array can be an integer constant, integer variable or an expressions that yield an integer.

The general form of array declaration is

data type variable name [size] ;

The data type specifies the type of element that will be contained in the arrays such as int , float or char. Size indicates the maximum number of elements that can be stored inside the array.
For example: *int mark [57];* declares the *mark*_to be an array containing 57 integer elements. Any subscript 0 to 56 are valid.
We can initialize the elements of arrays in the same way as the ordinary variables where they are declared.
The general form of initialization of array is:

datatype array name [size] = {list of values} ;
eg: int a [5] = {25,3,4,70,81} ;

TWO DIMENSIONAL ARRAYS

Table type arrays with two subscripts are called two dimensional arrays. When tables of values are to be processed, two dimensional arrays are used. Two –dimensional arrays are declared as follows:

data type array name [row_ size] [column_size] ;

Example: int a [2] [3];

Two dimensional arrays are stored in memory.

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

The first index selects the row and the second index selects the column within that row.

Initializing two –dimensional arrays

Two – dimensional arrays can be initialized during the declaration with a list of initial values enclosed in braces.

For example,

int x [2] [3]= {0,0,0,1,1,1};

Initialize the elements of the first row to zero and the second row to one. The above statement can also be written as

int x [2] [3] = {{0,0,0},{1,1,1}};

by surrounding the elements of each row by braces.

If the values are missing in an initialization, they are automatically set to zero.

DECLARATION OF ARRAYS AND STORING ARRAYS IN MEMORY

The general form of array declaration is

type variable-name [size];

The *type* specifies the type of elements that will be contained in the array, such as **int, float,** or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example, **float height [50];** declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly, **int group[10];** declares the group as an array to contain a maximum of 10 integer constants.

The C language treats character strings simply as arrays of characters. The size in a character string represents the maximum number of characters that the string can hold. For instance,

char name[10]; declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name.**

“WELL DONE”

Each character of the string is treated as an element of the array and is stored in the memory as follows:

‘W’
‘E’
‘L’
‘L’
‘ ’
‘D’
‘O’
‘N’

‘E’
‘\O’

INITIALISATION OF ARRAYS

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialisation of array is:

type array-name [size] = {list of values};

The values in the list are separated by commas. For example, the statement

int number[3] = {0,0,0};

will declare the variable number as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

float total[5] = {0.0,15.75,-10};

will initialize the first three elements to 0.0, 15.75, and -10.0 and remaining two elements to zero.

The *size* may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

int counter[ ] = {1,1,1,1};

will declare the counter array to contain four elements with initial values 1.

Character arrays may be initialized in a similar manner. Thus, the statement

char name[ ] = {‘J’, ‘o’, ‘h’, ‘n’};

declares the name to be an array of four characters, initialized with the string “John”.