Selection / Branching

Looping

Selection / Branching Statement :-

(i) Simple if Statement

(ii) if else Statement

(iii) else if ladder

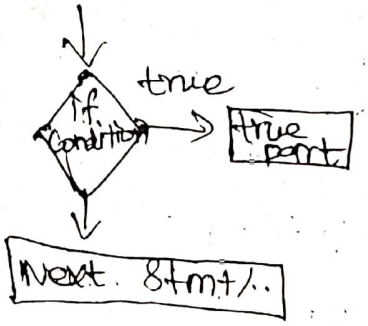
(iv) Nested if

(v) Switch statement

(i) Simple if Statement :-

Syntax :-

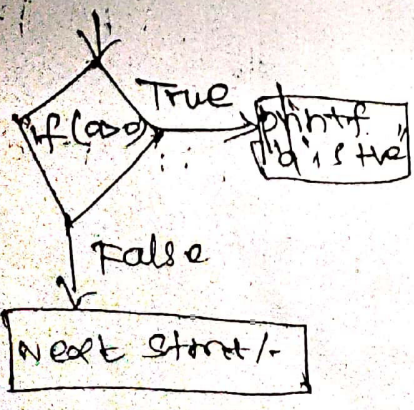
```
if (condition)
{
    true part ;
}
```



if the condition is true, true part will be executed.

eg :

```
if (a>0)
{
  printf ("a is positive");
```



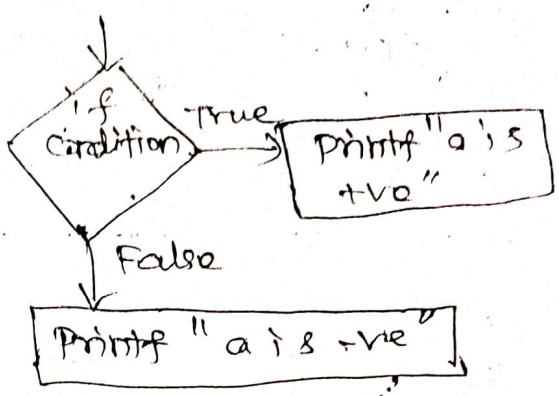
if the value of a is greater than zero

to display the string a positive, otherwise control will transfer to next statement.

(ii) if else statement :-

Syntax :-

```
if (condition)
{
  true part;
}
else
{
  false part;
}
```



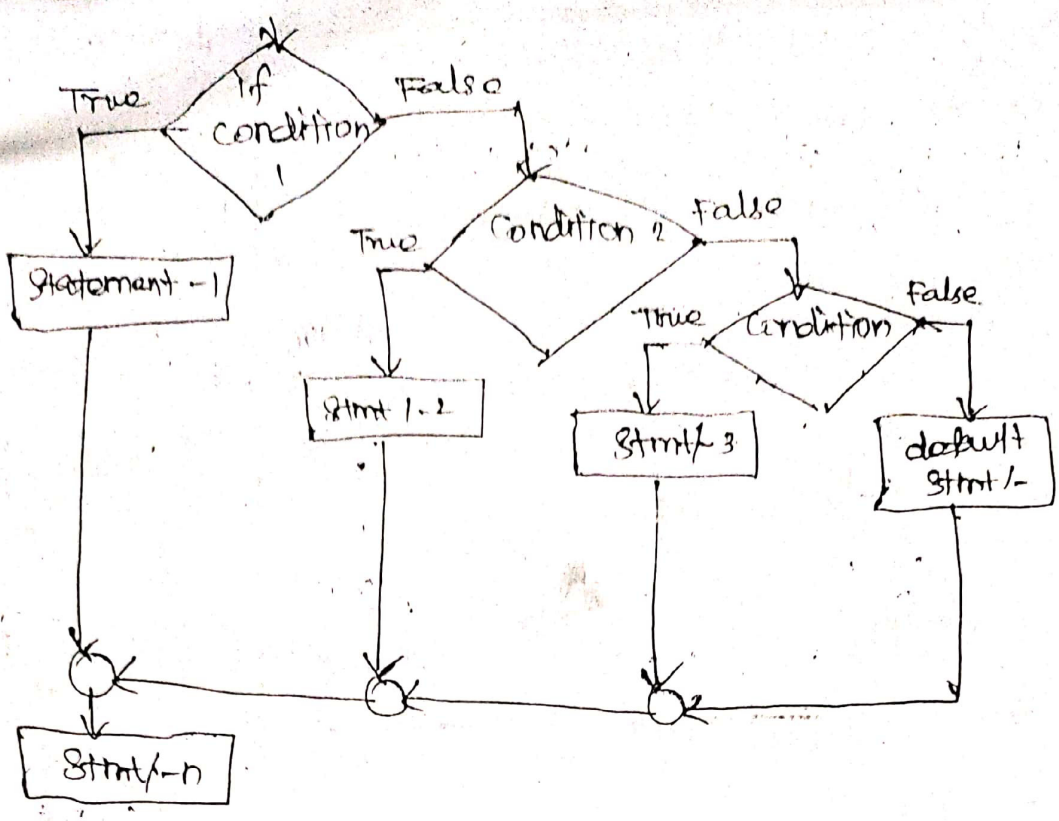
if the condition is true, true part will be executed, if the condition is false, else (false) part will be executed.

eg:

```
if (a>0)
{
  printf ("a is +ve");
}
else
{ printf ("a is -ve");
}
```

if the value of a is greater than zero, to display the string "a is positive" or if the condition is false, to display the string "a is negative"

(iii) else if ladder :-



Syntax :-

```
if (condition)
{
    true part ;
}
else if (condition 2)
{
   true part ;
}
else if (condition n)
{
    true part ;
}
else
{
    false part ;
}
```

if the condition 1 is true, the true part

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

statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

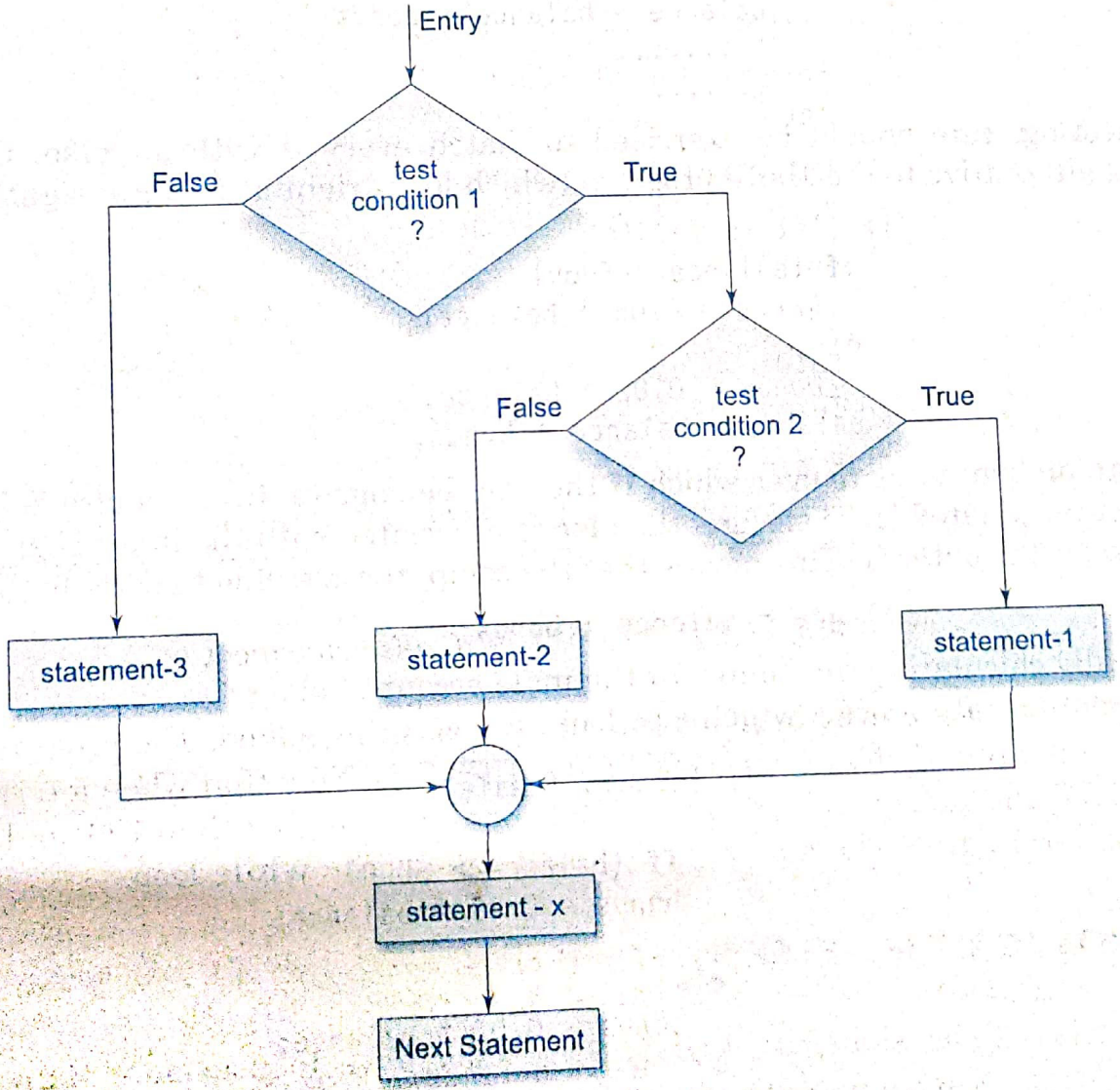


Fig. 5.7 *Flow chart of nested* **if...else** *statements*

the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statements associated with that **case** is executed. The general form of the **switch** statement is as shown below:

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

The *expression* is an integer expression or characters. *Value-1, value-2* ..... are constants or constant expressions (evaluable to an integral constant) and are known as *case labels*. Each of these values should be unique within a **switch** statement. **block-1, block-2** .... are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that **case** labels end with a colon (:).

When the **switch** is executed, the value of the expression is successfully compared against the values *value-1, value-2,....* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

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

The **default** is an optional case. When present, it will be executed if the value of the *expression* does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the **statement-x**. (ANSI C permits the use of as many as 257 case labels).

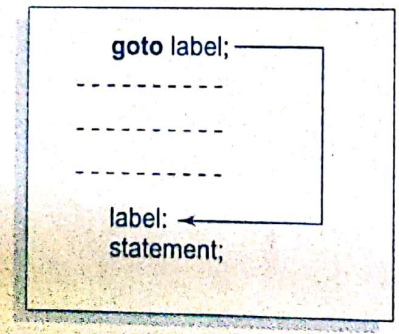
The selection process of **switch** statement is illustrated in the flow chart shown in Fig. 5.11.

- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.
- Use the most probable condition first. This will eliminate unnecessary tests, thus improving the efficiency of the program.
- The choice between the nested if and switch statements is a matter of individual's preference. A good rule of thumb is to use the switch when alternative paths are three to ten.
- Use proper indentations (See Rules for Indentation).
- Have the habit of using default clause in switch statements.
- Group the case labels that have similar actions.

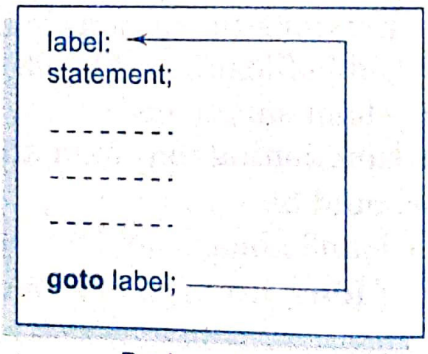
5.9 THE GOTO STATEMENT

So far we have discussed ways of controlling the flow of execution based on certain specifie conditions. Like many other languages, C supports the **goto** statement to branc unconditionally from one point to another in the program. Although it may not be essentia to use the **goto** statement in a highly structured language like C, there may be occasion 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. *label* is any valid variable name, and must be followed by a colon. The *label* is place immediately before the statement where the control is to be transferred. The general form of **goto** and *label* statements are shown below:



Forward jump



Backward jump

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

Note that a **goto** breaks the normal sequential execution of the program. If the *label*: before the statement **goto** *label*; a *loop* will be formed and some statements will be executed repeatedly. Such a jump is known as a *backward jump*. On the other hand, if the *label*:

placed after the **goto** *label*; some statements will be skipped and the jump is known as a *forward 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.

Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite loop*. The computer goes round and round until we take some special steps to terminate the loop. Such infinite loops should be avoided. Example 5.7 illustrates how such infinite loops can be eliminated.

Example 5.7 Program presented in Fig. 5.13 illustrates the use of the **goto** statement. The program evaluates the square root for five numbers. The variable count keeps the count of numbers read. When count is less than or equal to 5, **goto read**; directs the control to the label **read**; otherwise, the program prints a message and stops.

Program

```
#include <math.h>
main()
{
    double x, y;
    int count;
    count = 1;
    printf("Enter FIVE real values in a LINE \n");
read:
    scanf("%lf", &x);
    printf("\n");
    if (x < 0)
      printf("Value - %d is negative\n",count);
```

The true part of if condition contains another if condition, false part also having another if block.

Looping

1. for loop
2. while loop
3. Do while loop.

1) For loop:-

1) Syntax: for (initialization; condition; incre/decr)
{
body of the loop;
}

eg: for (i=1; i<4; i++)
printf ("%d", i);

2) while loop:-

Syntax: initialization;
while (condition)
{
body of the loop;
increment / decrement;
}

eg: int i=1;
while (i<=5)
{
printf ("%d", i);
i++;
{

Syntax:

```
initialization ;
do
{
body of the loop ;
increment / decrement ;
}
while (condition) ;
```

eg:

```
int i=1;
do
{
printf ("%d ", i);
i++ ;
}
while (i<=5);
```

Differentiate the while: and do while loop:-

while	do... while
1. Entry condition	Exit condition.

Arrays:-

Collection of data items with same data type and continous memory allocation for a single variable.

Syntax..

Data type V name [size];

: int a[3];

a[0]

a[0]	5?
a[1]	1251
a[2]	1018

Types of Arrays:-

1. one - dimensional array
2. two - dimensional array
3. Multi - dimensional array.

one - dimensional array:-

Syntax:-

Data type v. name [size];

eg: int a[3];

```
eg: void main()
{
  int a[5], i, j, t;
  Printf ("Enter array values");
for (i=0; i<5; i++)
 Scanf ("%d", &a[i]);
for (i=0; i<5; i++)
{
 for (j=i+1; j<5; j++)
{
  if (a[i]>a[j])
  {
  t=a[i];
  a[i]=a[j];
  a[j]=a[j];
  a[j]=t
```

0<5 i++
1<5
2<5 a[0] =
3<5 a[1] =
4<5 a[2] =
5 a[3] =

0<5
1<5
2<5
3<5
4<5

D > 1

Swap

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

Two-dimensional Array:-

Syntax:-

Datatype V.name [row size] [col size];

eg: int a[3][3];

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

a[0][0]=1 a[1][0]=4 a[2][0]=7
a[0][1]=2 a[1][1]=5 a[2][1]=8
a[0][2]=3 a[1][2]=6 a[2][2]=9

eg:

```
void main()
{
int a[3][3], i, j, b[3][3], c[3][3];
Printf("Enter a array values");
for(i=0; i<3; i++)
 for(j=0; j<3; j++)
 Scanf("%d", a[i][j]);
Printf("Enter b array values");
 for(i=0; i<3; i++)
 for(j=0; j<3; j++)
 Scanf("%d", &b[i][j]);
 Printf("Ascending order");
 for(i=0; i<3; i++)
 {
 for(j=0; j<3; j++)
```

row
column

Strings

A one dimensional array of characters terminated by a null ["\0"]. Ex.

Char name[5] = {'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0'}

strlen(), strcpy(), Strcat() and strcmp().

(1) Strlen

This function counts the number of characters present in a string.

```
# include <Stdio.h>
# include <String.h>
Void main( )
{
  Char arr[ ] = " Bamboozled ";
  int len1, len2;
  len1 = Strlen (arr);
  len2 = Strlen ("Humpty Dumpty");
  printf ("\nstring = %s length = %d", arr, len1);
  printf ("\nstring = %s length = %d", Humpty Dumpty, len2);
}
```

O/p would be

String = Bamboozled length = 10

String = Humpty Dumpty length = 12

Strcpy

This function copies the contents of one string into another. E.g.

```
#include <stdio.h>
#include <string.h>
void main( )
{
char source ( )= "Sayonara";
char target (20)
Strcpy (target, source);
printf ("\ntarget string= %s", target)
}
```

O/p will be

Source string= Sayonara.

target string= Sayonara.

Strcpy( ) goes on copying the characters in source string into target string till it encounters the end of the source string ('\0')

Strcmp( )

This function compares two strings and find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of the one strings is reached, whichever occurs first. If two strings are identical, strcmp( ) returns a value zero. If they are not, it returns the numeric difference between the ASCII Values of the first non-matching pair of

characters.

```
#include <stdio.h>
#include <string.h>
Void main ( )
{
Char string1[ ]= "Jerry"
Char string2[ ]= "Ferry"
int i, j, k.
i = Strcmp ( String1, "Jerry");
j = Strcmp (string1, string);
k = Strcmp (string1, "Jerry boy");
printf ("\n%d %d %d", i, j, k);
}
```

O/p is 04 - 32.
= ⇓
ASCII value

Strcat This function concatenates the source string at the end of the target string. For example "Bombay" and Nagpur on concatenation would result into a string "BombayNagpur".

Here is an example of strcat( ).

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

```
{
  Char soure [] = "Folks!" ;
  Char target [30] = "Hello";
  Strcat ( target, source );
  printf ("\nsource string = %s", source);
  printf ("\ntarget string = %s", target);
}
```

O/p is

Source string = Folks!

target string = HelloFolks!