Chapter 4

CONTROL STRUCTURES

Definition

Control structures represent the forms by which statements in a program are executed. **Flow of control** refers to the order in which the individual statements, instructions or function calls of a program are executed or evaluated.

IMPORTANCE OF CONTROL STRUCTURES

Generally, a program should execute the statements one by one until the defined end. This type of a program structure is called **sequential structure**. The functionality of this type of program is limited since it flows in a single direction. However, all high-level programming languages enable the programmer to change the flow of program execution. This is done by the use of control structures whose main benefits are to enable **decision making** and **repetition** as well as giving the power to do far more **complex processing** and **provide flexibility with logic.** The sophisticated logic is necessary for a program to solve complex problems.

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- continuation at a different statement i.e. unconditional jump e.g. GoTo statements
- executing a set of statements only if some condition is met i.e. choice
- executing a set of statements zero or more times, until some condition is met i.e. loop
- executing a set of distant statements, after which the flow of control usually returns e.g. subroutines/functions

TYPES OF CONTROL STRUCTURES

There are three types in C:

1. Sequence structures

Program statements are executed in the sequence in which they appear in the program.

2. Selection structures/Decision Structures

Statement block is executed only if some condition is met. These structures include **if**, **if**/**else**, and **switch.** Selection structures are extensively used in programming because they allow the program to decide an action based upon user's input or other processes for instance in password checking.

3. Repetition/Iterative structures

This is where a group of statements in a program may have to be executed repeatedly until some condition is satisfied. These include **while**, **do/while** and **for**

SELECTION STRUCTURES

(a) THE IF SELECTION STRUCTURE

– Used to choose among alternative courses of action i.e. the if statement provides a junction at which the program has to select which path to follow. The **if selection** performs an action only if the condition is **true,**

General form

```
If (expression)
        statement
```

Pseudocode:

```
If student's marks is greater than or equal to 600
Print "Passed"
```

As in

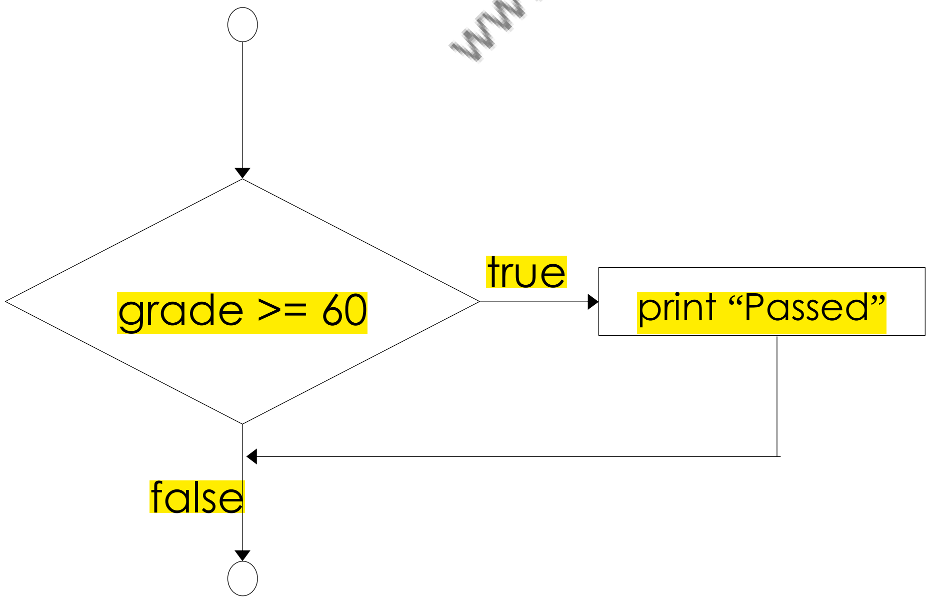
```
if (marks>=600)
        printf("Passed");
```

If condition is **true**

– Print statement executed and program goes on to next statement

– If **false**, print statement is ignored and the program goes onto the next statement

NB/ Indenting makes programs easier to read



Flow chart for the **if** selection structure

NB/ The statement in the if structure can be a single statement or a block (Compound statement). If it's a block of statements, it must be marked off by braces.

```
if (expression)
        {
                Block of statements
}
```

As in

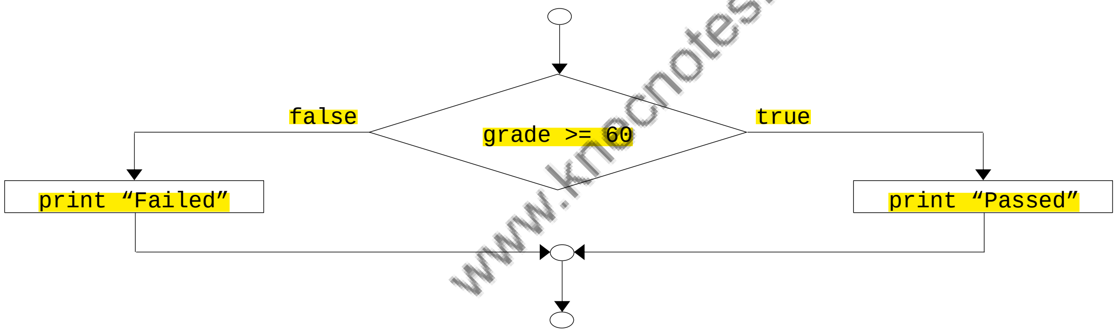
```
If (salary>5000)
        {
                tax_amount = salary * 1.5;
                printf("Tax charged is %f", tax_amount);
        }
```

(b) THE IF/ELSE

While **if** only performs an action if the condition is **true, if**/**else** specifies an action to be performed both when the condition is **true** and when it is **false. E.g.**

Pseudocode:

```
If student's grade is greater than or equal to 60
Print "Passed"
else
Print "Failed"
```



Flow chart for the **if/else** selection structure

Example

```
if (x >=100)
        {
                printf("Let us increment x:\n");
                x++;
        }

else

                printf("x < 0 \n);
```

(c) THE IF...ELSE IF...ELSE STATEMENT

– Test for multiple cases/conditions.
– Once a condition is met, the other statements are skipped
– Deep indentation usually not used in practice

Pseudocode for an if..else if..else structure

```
If student's grade is greater than or equal to 90
Print "A"
Else If student's grade is greater than or equal to 80
  Print "B"
else  If student's grade is greater than or equal to 70
    Print "C"
  else  If student's grade is greater than or equal to 60
      Print "D"
    else
      Print "F"
```

Example

```
#include <stdio.h>

main()
{
int marks;
printf("Please enter your MARKS:");
scanf("%d", &marks);

if (marks>=90 && marks <=100)
        printf("Your grade is A\n");

else if (marks>=80 && marks <=89)
        printf("Your grade is B\n");

else if (marks>=70 && marks <=79)
        printf("Your grade is C\n");

else if (marks>=60 && marks <=69)
        printf("Your grade is D\n");
else if (marks >100)
        printf("Marks out of range\n");
else
        printf("Your grade is F\n");
}
```

(d) NESTED IF STATEMENTS

One **if** or **else if** statement can be used inside another **if** or **else if** statement(s).

Syntax

The syntax for a **nested if** statement is as follows:

```
if (boolean_expression 1)
{
/* Executes when the boolean expression 1 is true */
        if(boolean_expression 2)
```

```
	{
	/* Executes when the boolean expression 2 is true */
	}
}
```

You can nest else **if...else** in the similar way as you have **nested if** statement.

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
/* check the boolean condition */
	if( a = = 100 )
	{
		/* if condition is true then check the following */
		if( b = = 200 )
		{
		/* if condition is true then print the following */
		printf("Value of a is 100 and b is 200\n" );
		}
	}
	return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

(e) SWITCH STATEMENT

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax

The syntax for a switch statement in C programming language is as follows:

```
switch(expression)
{
case constant-expression
statement(s);
break;
case constant-expression :
statement(s);
break;
/* you can have any number of case statements */
default :
statement(s);
}
```

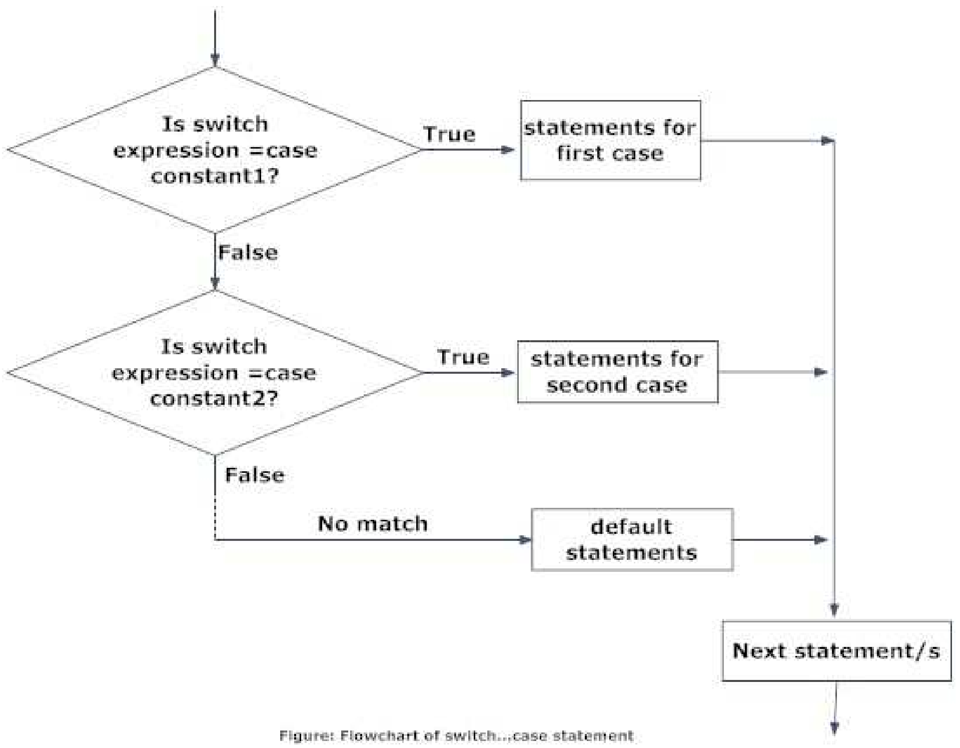

Is switch expression =case constant1?
True
statements for first case
False
Is switch expression =case constant2?
True
statements for second case
False
No match
default statements
Next statement/s

Figure: Flowchart of switch...case statement

The following rules apply to a switch statement:

1) You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
2) The constant-expression for a case must be the same data type as the variable in the switch
3) When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
4) When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
5) Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
6) A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Sample Switch Statement

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

char grade;

printf("Enter your grade:");
scanf("%c", &grade);

switch (grade)
{
case 'A':
        printf("Excellent!\n");
        break;
case 'B':
        printf("Very Good!\n");
        break;
case 'C':
        printf("Good!\n");
        break;
case 'D':
        printf("Work harder!\n");
        break;
default:
        printf("Fail!\n");
}
}
```

(f) NESTED SWITCH STATEMENTS

It is possible to have a **switch** as part of the statement sequence of an **outer switch**. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax

The syntax for a **nested switch** statement is as follows:

```
switch(ch1) {
case 'A':
printf("This A is part of outer switch" );
        switch(ch2) {
        case 'A':
        printf("This A is part of inner switch" );
        break;
        case 'B':
        }
break;
case 'B':
}
```

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
switch(a) {
case 100:
printf("This is part of outer switch\n", a );
        switch(b) {
        case 200:
        printf("This is part of inner switch\n", a );
        printf("A is equals to %d and B is equals to %d", a, b);
        }
}
printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );
return 0;
```

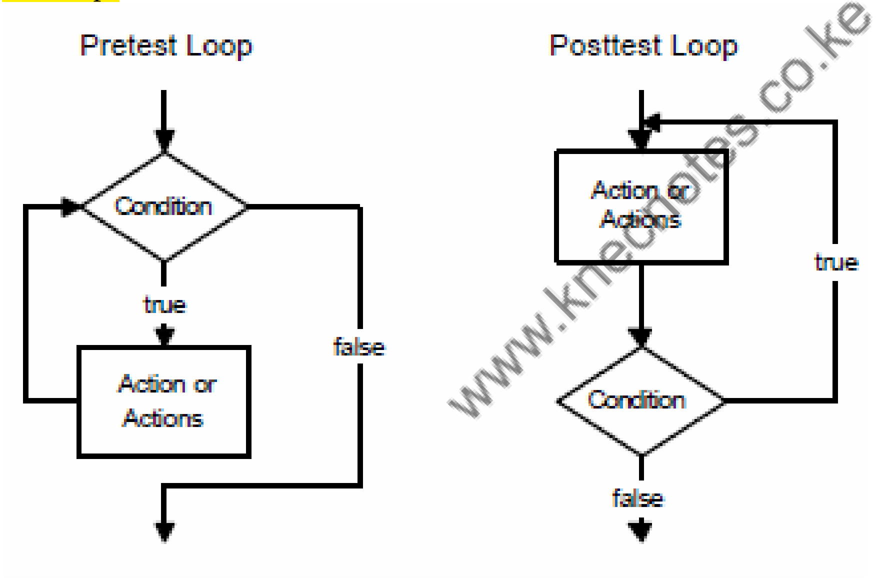
}

When the above code is compiled and executed, it produces the following result:

```
This is part of outer switch
This is part of inner switch
A is equals to 100 and B is equals to 200
Exact value of a is : 100
Exact value of b is : 200
```

REPETITION/ITERATIVE/LOOP STRUCTURES

A loop statement allows the execution of a statement or a group of statements multiple times until a condition either tests true or false. There are two types of loops: **Pre-test** and **post-test** loops.
In a pretest loop, a logical condition is checked before each repetition to determine if the loop should terminate. These loops include:
– *while loop*
– *for loop*
Post-test loops check a logical condition after each repetition for termination. The do-while loop is a *post-test loop*.



(a) WHILE LOOP IN C

A **while** loop statement repeatedly executes a target statement **as long as** a given condition is **true**.

The syntax of a while loop in C programming language is:

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

The statement(s) may be a single statement or a block of statements. The loop iterates while the condition is true.
When the condition becomes false, program control passes to the line immediately following the loop.

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 10; //loop index

/* while loop execution */
while( a < 20 )
{
printf("value of a: %d\n", a);
a++;
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

(b) FOR LOOP IN C

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is:

```
for ( initial expression; test expression/logical codition; update
expression )
{
statement(s);
}
```

Here is the flow of control in a for loop:

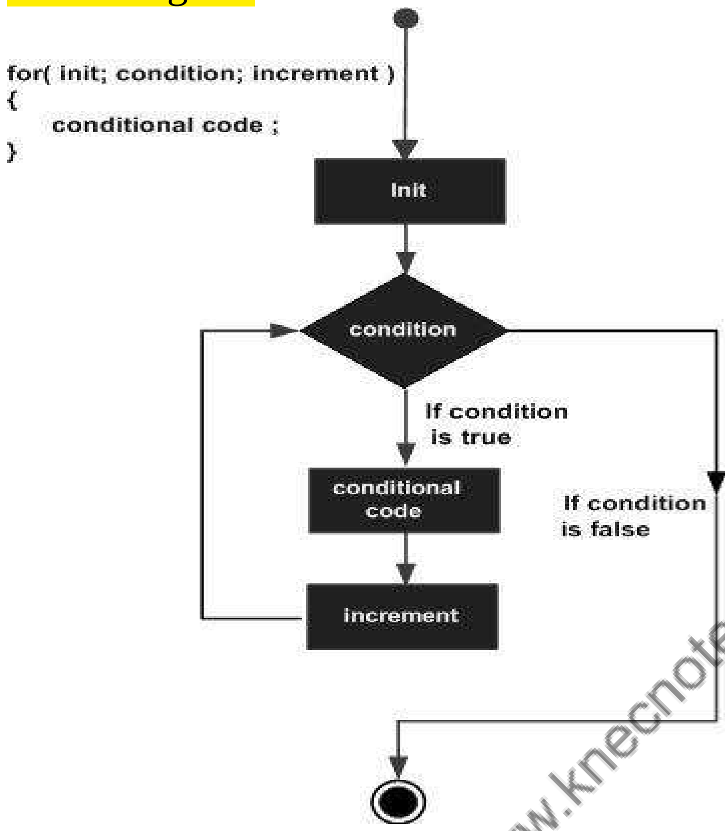
1. This step initializes any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

3. After the body of the for loop executes, the flow of control jumps back up to the update expression. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself. After the condition becomes false, the for loop terminates.

Flow Diagram



Example

```
#include <stdio.h>
int main ()
{
int a;//loop index
/* for loop execution */
for(a = 10; a < 20; a++)
{
printf("value of a: %d\n", a);
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

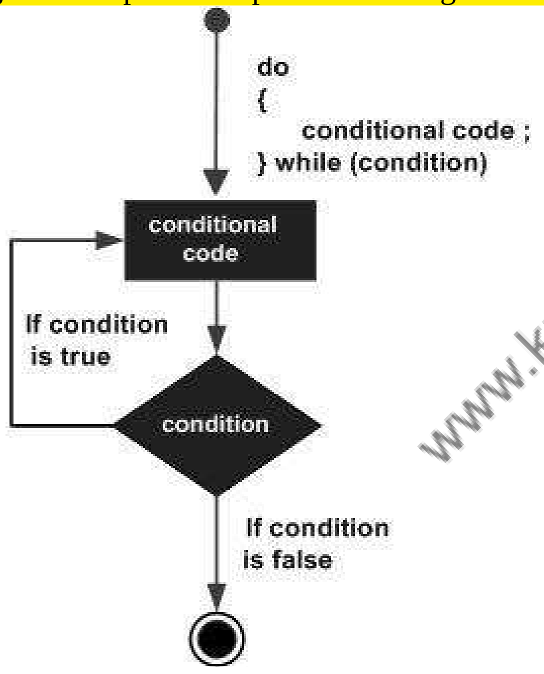
(c) DO...WHILE LOOP IN C

Unlike for and while loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.
A **do...while** loop is similar to a while loop, except that a **do...while** loop is guaranteed to execute at least one time. The structure, therefore loops until a condition tests false i.e. loop until.

Syntax

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

If the condition is **true**, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes **false**.



Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* do loop execution */
do
{
printf("value of a: %d\n", a);
a = a + 1;
}while( a < 20 );
return
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

(d) NESTED LOOPS IN C

C programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

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

```
for ( init; condition; increment )
{
        for ( init; condition; increment )
        {
        statement(s);
        }
statement(s);
}
```

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

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

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

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

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

Example

```
#include <stdio.h>

int main()
{
   int n, c, k;

   printf("Enter number of rows:");
   scanf("%d",&n);

   for ( c = 1 ; c <= n ; c++ )
   {
      for( k = 1 ; k <= c ; k++ )
   {
         printf("%d",k);
  }
         printf("\n");
   }

   return 0;
}
```

Result:
If the user interred 5 as the number of rows, the output would be:

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

TERMINATING LOOPS

• Counter-controlled loops - a loop controlled by a counter variable, generally where the number of times the loop will execute is known ahead of time especially in *for loops*.
• Event-controlled loops - loops where termination depends on an event rather than executing a fixed number of times for example when a zero value is keyed in or search through data until an item is found. Used mostly in while loops and do-while loops.

Using a Sentinel

• The value -999 is sometimes referred to as a sentinel value. The value serves as the “guardian” for the termination of the loop. It is a good idea to make the sentinel a constant:

```
#define STOPNUMBER -999
while (number != STOPNUMBER) ...
```

BRANCHING STATEMENTS

(a) BREAK STATEMENT IN C

The **break** statement has the following two uses:

1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **switch** statement.
3. If you are using **nested loops** (i.e., one loop inside another loop), the **break** statement will stop the execution of the innermost loop and start executing the next line of code after the block.

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* do loop execution */
do
{
if( a = = 15)
{
/* skip the iteration */
break;
}
printf("value of a: %d\n", a);
a++;
}while( a < 20 );
return 0;
}
```

(b) CONTINUE STATEMENT IN C

The **continue** statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between. For the **for loop**, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do**...**while** loops, **continue** statement causes the program control to pass to the conditional tests.

Example

```
//program to demonstrate the working of continue statement in C programming
# include <stdio.h>
int main(){
    int i,num,product;
    for(i=1,product=1;i<=4;++i){
        printf("Enter num%d:",i);
        scanf("%d",&num);
        if(num==0)
            continue;  /*In this program, when num equals to zero, it skips
the statement product*=num and continue the loop. */
        product*=num;
}
    printf("product=%d",product);
return 0;
}value of a: 19
```

(c) GOTO STATEMENT IN C

A **goto** statement provides an unconditional jump from the **goto** to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a **goto** can be rewritten so that it doesn't need the **goto**.

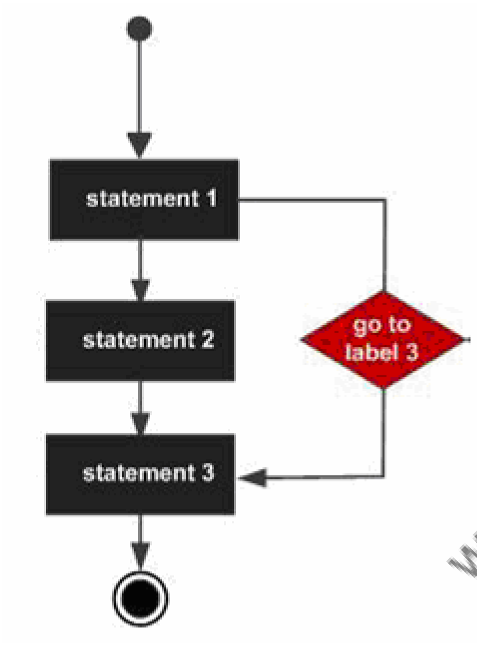
Syntax

The syntax for a **goto** statement in C is as follows:

goto label;

..

.



Example

```
#include <stdio.h>
int main ()
{
/* for loop execution */
     int a,userinput,sum=0;

     for(a = 0; a < 5;a++)
     {
      printf("Enter a number: ");
      scanf("%d",&userinput);
      if (userinput <1)
      goto jump;

      sum+=userinput;
     }

jump:
```

```
printf("The sum of the values is %d\n", sum);
return 0;
}
```

(d) THE RETURN STATEMENT

The last of the branching statements is the `return` statement. The `return` statement exits from the current function, and control flow returns to where the function was invoked. The `return` statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the `return` keyword.

```
return count;
```

The data type of the returned value must match the type of the method's declared return value. When a function is declared `void`, use the form of `return` that doesn't return a value.

```
return;
```

THE INFINITE LOOP

A loop becomes **infinite** loop if a condition never becomes **false**. The **for loop** is traditionally used for this purpose. Since **none** of the three expressions that form the **for loop** are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>
int main ()
{
for( ; ; )
{
printf("This loop will run forever.\n");
}
return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the **for(;;)** construct to signify an infinite loop.
NOTE: You can terminate an infinite loop by pressing **Ctrl + C** keys.