

UNIT III

Character Functions in C

- C functions which are used to perform operations on characters in a program are called character functions
- “ctype.h” header file support all the character functions in C language.
- The following are the most common character functions used in C

Character Functions in C

Functions	Description
<u>isalpha()</u>	checks whether character is alphabet
<u>isdigit()</u>	checks whether character is digit
<u>isalnum()</u>	Checks whether character is alphanumeric
<u>isspace()</u>	Checks whether character is space
<u>islower()</u>	Checks whether character is lower case
<u>isupper()</u>	Checks whether character is upper case
<u>isxdigit()</u>	Checks whether character is hexadecimal
<u>tolower()</u>	Checks whether character is alphabetic & converts to lower case
<u>toupper()</u>	Checks whether character is alphabetic & converts to upper case

LIST OF INBUILT ARITHMETIC FUNCTIONS IN C LANGUAGE:

- C functions which are used to perform mathematical operations in a program are called Arithmetic functions.
- “math.h” and “stdlib.h” header files support all the arithmetic functions in C language.
- All the arithmetic functions used in C language are given below.

Function	Description
<u>abs ()</u>	This function returns the absolute value of an integer. The absolute value of a number is always positive. Only integer values are supported in C.
<u>floor ()</u>	This function returns the nearest integer which is less than or equal to the argument passed to this function.
<u>round ()</u>	This function returns the nearest integer value of the float/double/long double argument passed to this function.
<u>ceil ()</u>	This function returns nearest integer value which is greater than the argument passed to this function.
<u>sin ()</u>	This function is used to calculate sine value.
<u>cos ()</u>	This function is used to calculate cosine.
<u>cosh ()</u>	This function is used to calculate hyperbolic cosine.
<u>exp ()</u>	This function is used to calculate the exponential “e” to the x th power.
<u>tan ()</u>	This function is used to calculate tangent.
<u>tanh ()</u>	This function is used to calculate hyperbolic tangent.
<u>sinh ()</u>	This function is used to calculate hyperbolic sine.
<u>log ()</u>	This function is used to calculates natural logarithm.
<u>log10 ()</u>	This function is used to calculates base 10 logarithm.
<u>sqrt ()</u>	This function is used to find square root of the argument passed to this function.
<u>pow ()</u>	This is used to find the power of the given number.
<u>trunc ()</u>	This function truncates the decimal value from floating point value and returns integer value.

ceil() function

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

```
void main ()
{
    float val1 =1.6, val= 1.2, val3=2.8, val4 =2.3;
    clrscr();
    printf ("value1 = %.1lf\n", ceil(val1));
    printf ("value2 = %.1lf\n", ceil(val2));
    printf ("value3 = %.1lf\n", ceil(val3));
    printf ("value4 = %.1lf\n", ceil(val4));
    getch();
}
```

```
value1 = 2.0
value2 = 2.0
value3 = 3.0
value4 = 3.0
```

floor() function

Input Statements in C

1. Input, integer numbers

%d (int)

%u (unsigned int)

%ld (long int)

%uld (unsigned long int)

%i can also be used instead of %d

(a) int num1, num2;

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

5672 943

num1 = 56 num2 = 72 (Error)

(b) int a, b, c;

scanf ("%d %*d %d", &a, &b);

123 456 789

a = 123 b = 789

scanf ("%d %*d %d", &a, &b, &c);

123 456 789

a = 123 b = 789 c = garbage value
(error)

(c) int a, b;

scanf ("%d - %d", &a, &b);

256 - 489

(The sign '-' should be placed between 256 and 489 while inputting the numbers)

Inputting real numbers

%f %lf %Lf
(float) (double) (long double)

(a) scanf (" %f %lf %Lf", &x, &y, &z);

475.89 43.21e-1 678

x = 475.89 y = 4.321 z = 678.0

(b) void main()

{ double dy;

scanf (" %15le", &dy);

Read dy in exponential notation and
assign a maximum field of width 15
characters

I/p $\Rightarrow -23.12e+02$

(c) void main()

{ ~~double dx, dy;~~

int i;

float x;

char c;

scanf ("%d %f %c", &i, &x, &c);

10 256.875 F

i = 10 x = 256.8 c = F (Error)

Inputting characters ③

`scanf ("%c", &a);` $\begin{matrix} \%c & \%uc \\ \downarrow & \downarrow \\ \text{character} & \text{unsigned character} \end{matrix}$

always leave a space in the control string (or format string) before $\%c$.

This prevents any previously entered extra characters from being assigned to variable `a`.

```
(a) void main ()
{
    char c1, c2, c3;
    scanf ("%c %c %c", &c1, &c2, &c3);
}

Input  $\Rightarrow$  a b c
      c1 = a  c2 = b  c3 = c
```

(b) `void main ()`
`{`
`char c1, c2, c3;`
`scanf ("%c %c %c", &c1, &c2, &c3);`
 Input \Rightarrow a \downarrow b \downarrow c
`c1 = a` `c2 = \downarrow` `c3 = b` (Error)
`}`

```

Using getchar()
void main()
{
    char d;
    d = getchar(); // get a character
                  // and assign it to d.
}

```


Rules for scanf

- * Never end the format string with a whitespace character. It is a fatal error.
- * Leave a blank space when inputting characters.
- * Also, leave a blank space in the format string which begins with inputting characters or strings.

scanf (" %c %d %f", &a, &b, &c);

Output Statements in C
printf statement (Integers)

printf ("%d", 9876)

9	8	7	6
---	---	---	---

printf ("%6d", 9876)

		9	8	7	6
--	--	---	---	---	---

printf ("%2d", 9876)

9	8	7	6
---	---	---	---

printf ("% -6d", 9876)

9	8	7	6		
---	---	---	---	--	--

printf ("%06d", 9876)

0	0	9	8	7	6
---	---	---	---	---	---

printf (real numbers)

y = 98.7654

printf ("%7.4f", y)

9	8	.	7	6	5	4
---	---	---	---	---	---	---

printf ("%7.2f", y)

		9	8	.	7	7
--	--	---	---	---	---	---

printf ("%7.2f", y)

9	8	.	7	7		
---	---	---	---	---	--	--

printf ("%0f", y)

9	8	.	7	6	5	4	0	0
---	---	---	---	---	---	---	---	---

printf ("%10.2e", y)

		9	.	8	8	e	+	0	1
--	--	---	---	---	---	---	---	---	---

printf ("%11.4e", -y)

-	9	.	8	7	6	5	e	+	0	1
---	---	---	---	---	---	---	---	---	---	---

printf ("%10.2e", y)

9	.	8	8	e	+	0	1		
---	---	---	---	---	---	---	---	--	--

printf ("%e", y)

9	.	8	7	6	5	4	e	+	0
---	---	---	---	---	---	---	---	---	---

printf ("%*.*f", 7, 2, y)

printf (character)

c = 'a';

putchar(c);

printf ("%c", c);

o/os. [Input statements for strings]

(1) main()

```
{ int name1[10];
```

```
scanf("%o", name1);
```

Automatically terminates when the first whitespace character is encountered. '\0' is automatically assigned at the end of the string

I/p → New _ Delhi

name1 ← New

```
char a[20];
```

(2) scanf("%s", a)

Terminates when return key is encountered.

I/p New _ Delhi

• a ← New _ Delhi. '\0' is automatically assigned

(3) scanf("%s", a);

Takes in only lower case letters a to z and a blank space.

I/p happy _ Birthday

a ← happy _ . \0 is automatically assigned

(4) char a[40];

```
gets(a);
```

Means → get a string and store it in a.

The string will terminate when a newline character (return key) is encountered.

It will append a null character(`\0`) to the string automatically

```
(5) char a[50];  
    for (i=0; (a[i] = getchar()) != '\n'); ++i  
    {  
    }  
    a[i] = '\0';
```

Since `a` is an array of characters, to enter a string, each character can be entered using loops. '\0' is not automatically appended. You should add it.

```
(6) char a[50];  
    gets(a);
```

`gets(a) ⇒` accepts all character till return key is pressed.

`\0` is automatically appended at the end of the string

Printing strings (O/p statement for strings)

- ① `printf (" %s ", y);` prints all characters in `y` upto `'\0'`.

- ② puts(y) prints all characters in y upto '\0'.

- ```

③ i = 0;
 while (name[i] != '\0')
 {
 putchar (name[i]);
 i++;
 }

```

```
scanf("%d", &n);
printf("%d", n);
```

снач  $y[30]$ ;

```
scanf ("%s", y); y = NEW DELHI / 11000010
```

`printf ("%s", y)`

|   |   |   |   |   |   |   |   |  |  |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|--|--|---|---|---|---|---|---|
| N | E | W | D | E | L | H | I |  |  | 1 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|--|--|---|---|---|---|---|---|

`printf("%20s", y)`

|  |  |  |  |   |   |   |  |   |   |   |   |   |  |  |  |   |   |   |   |   |   |   |
|--|--|--|--|---|---|---|--|---|---|---|---|---|--|--|--|---|---|---|---|---|---|---|
|  |  |  |  | N | E | W |  | D | E | L | H | I |  |  |  | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
|--|--|--|--|---|---|---|--|---|---|---|---|---|--|--|--|---|---|---|---|---|---|---|

```
printf("%020.10s", y) |
```

```
printf("%0.5s", y) | N | E | W | | D |
```

`printf("%0-20.10s", y)`

```
printf("%5s", y) NEW DELHI 110001
```

# Decision Making in C

- There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next.
- Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.
- For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r.



# Decision Making

## If-else

## Switch

### if

```
if(condition)
{
 //true
}
```

### if-else

```
if(condition)
{
 //true
}
else
{
 //false
}
```

### if-else if

```
if(condition 1)
{
 //true
}
else if(condition 2)
{
 //true
}
else
{
}
```

### Nested if

```
if(condition 1)
{
 if(condition)
 {
 }
 else
 {
 }
}
else
{
 if(condition)
 {
 }
 else
 {
 }
}
```

```
switch(expression)
{
 case 1:
 break;
 case 2:
 break;
 case 3:
 break;
 default;
}
```

# Decision making statements in C

- Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in C or C++ are:
  - if statement
  - if..else statements
  - nested if statements
  - if-else-if ladder
  - switch statements
  - Jump Statements:
    - break
    - continue
    - goto

# if statement in C

- if statement is the most simple decision making statement.
- It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

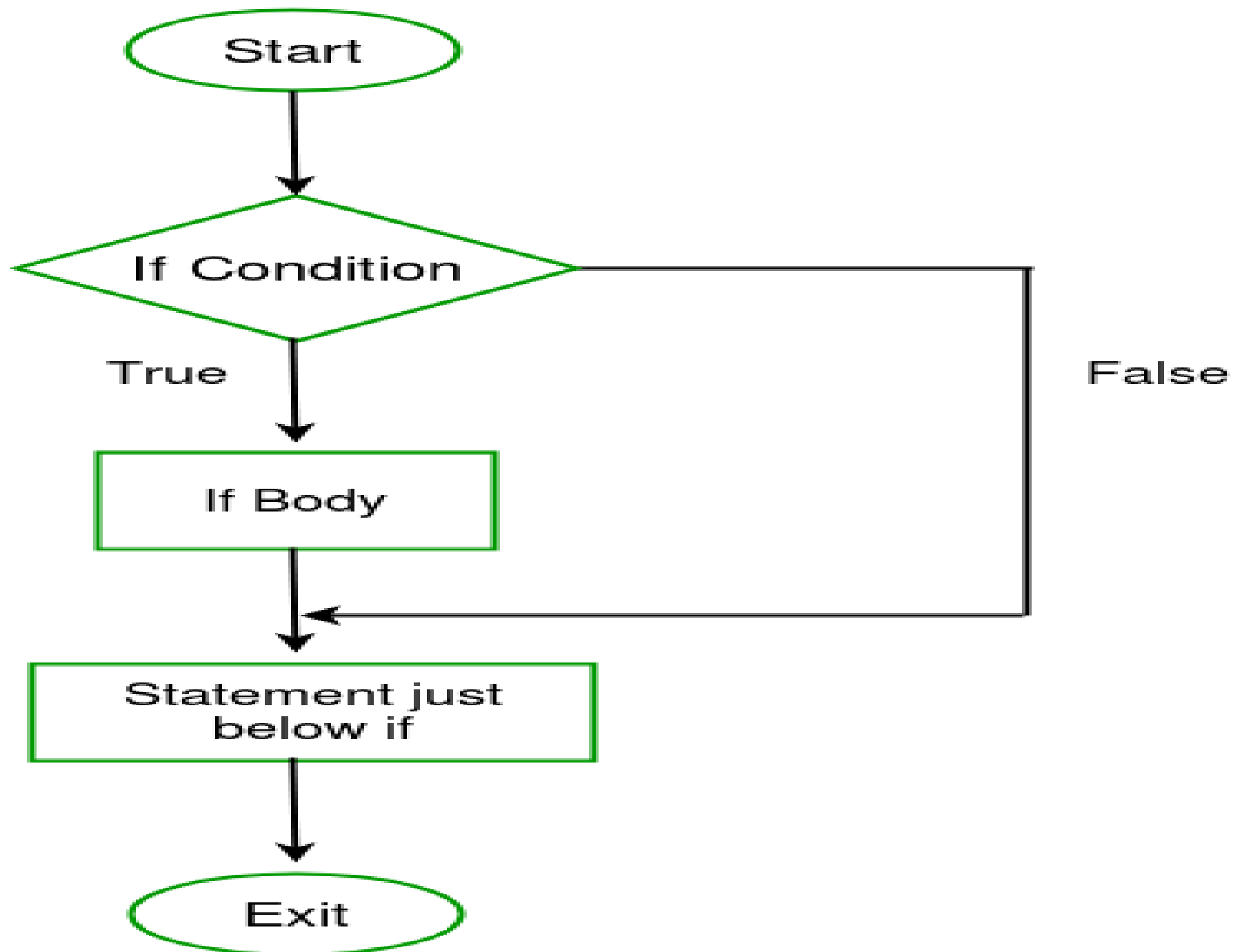
- **Syntax:**

```
if(condition)
{
 // Statements to execute if condition is true
}
```

- Here, condition after evaluation will be either true or false.
- if the value is true then it will execute the block of statements below it otherwise not.

# if statement in C

- If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.
- if(condition)  
    statement1;  
    statement2;
- // Here if the condition is true, if block
- // will consider only statement1 to be inside
- // its block.

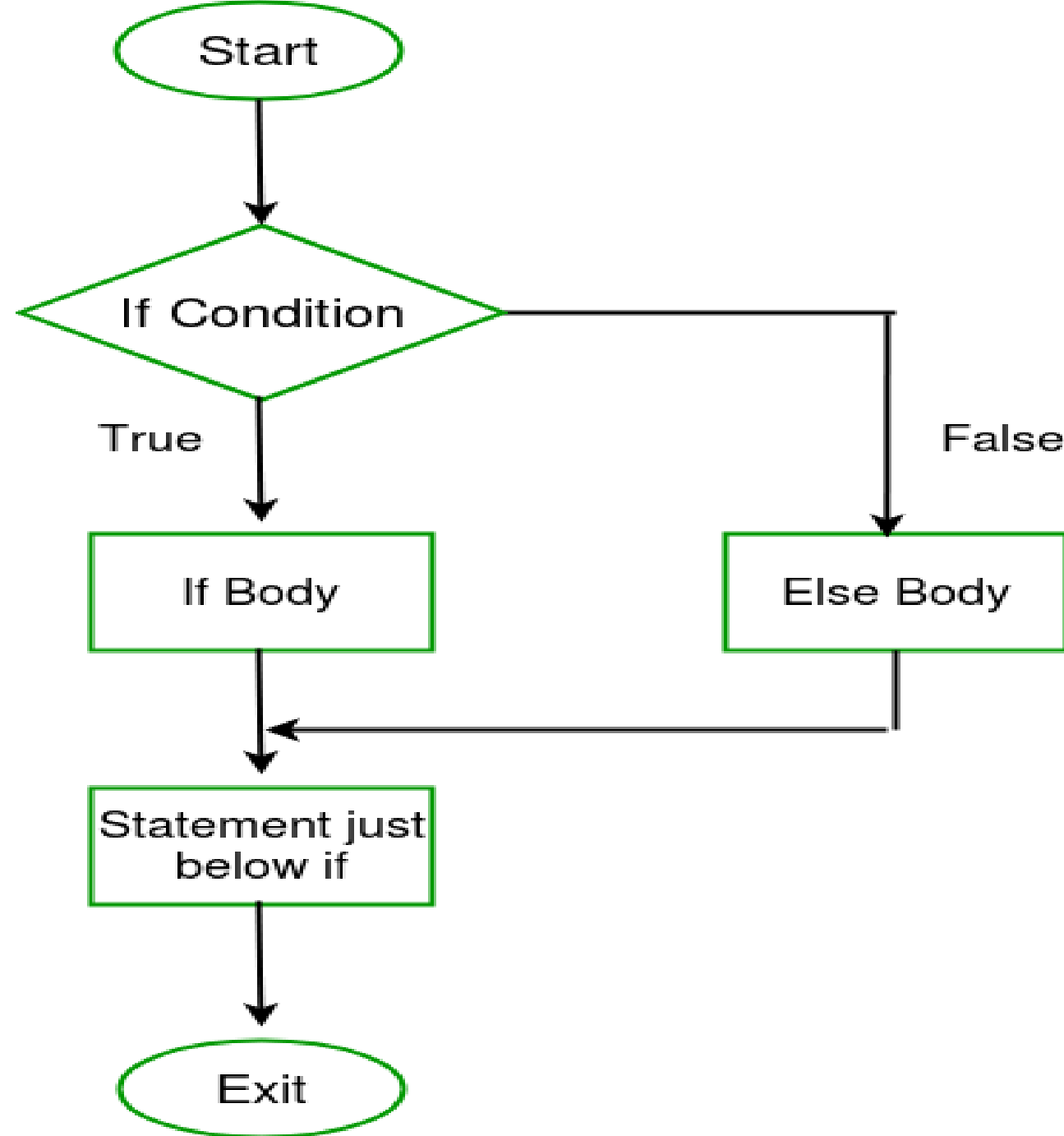


# if-else in C

- The *if-else* statement tells us that if a condition is true it will execute a block of statements else if the condition is false it will execute another block of statements

## SYNTAX

```
if (condition)
{
 // Executes this block if
 // condition is true
}
else
{
 // Executes this block if
 // condition is false
}
```

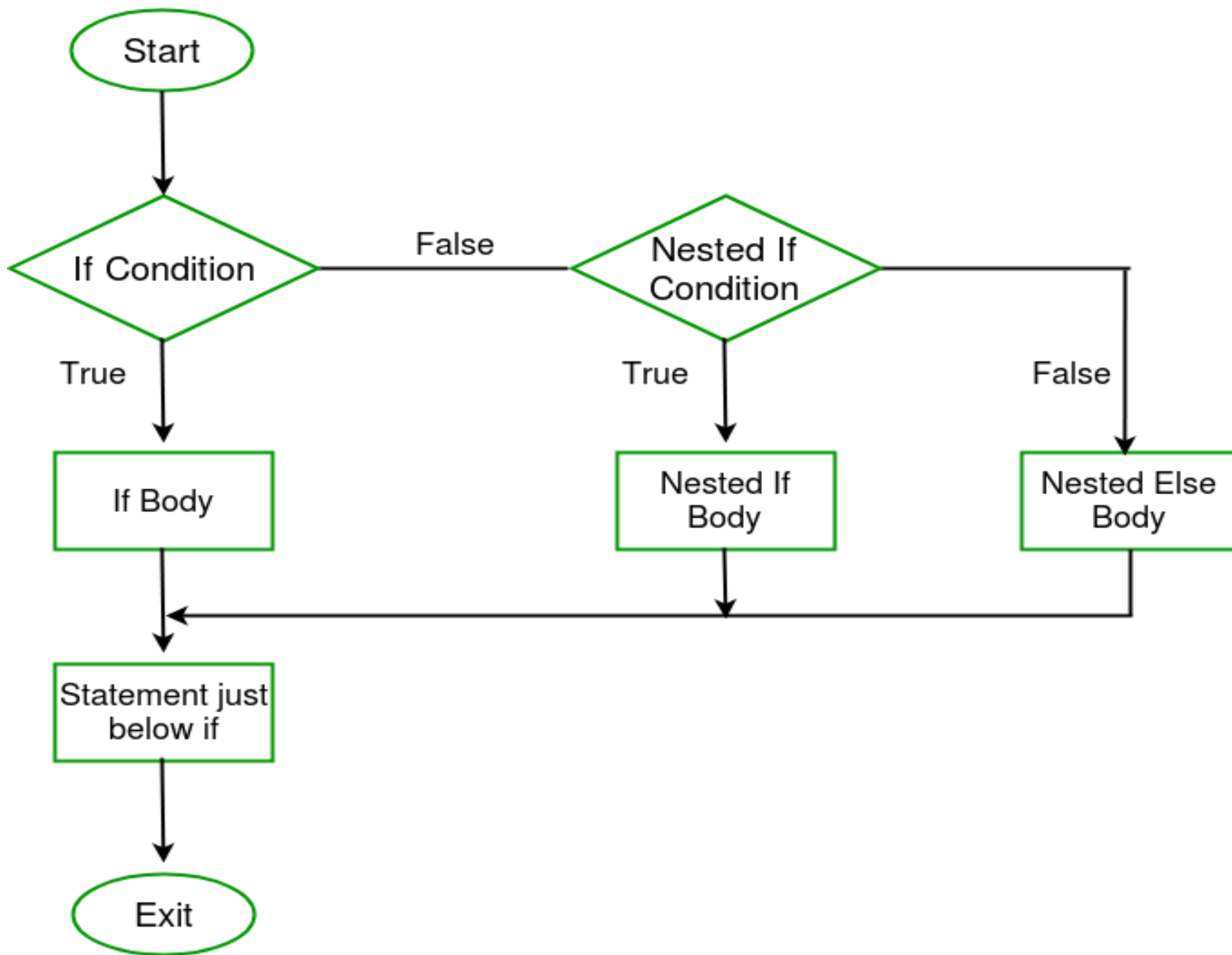


# nested-if in C

- Nested if statements means an if statement inside another if statement.

```
if (condition1)
{
 // Executes when condition1 is true
 if (condition2)
 {
 // Executes when condition2 is true
 }
}
```



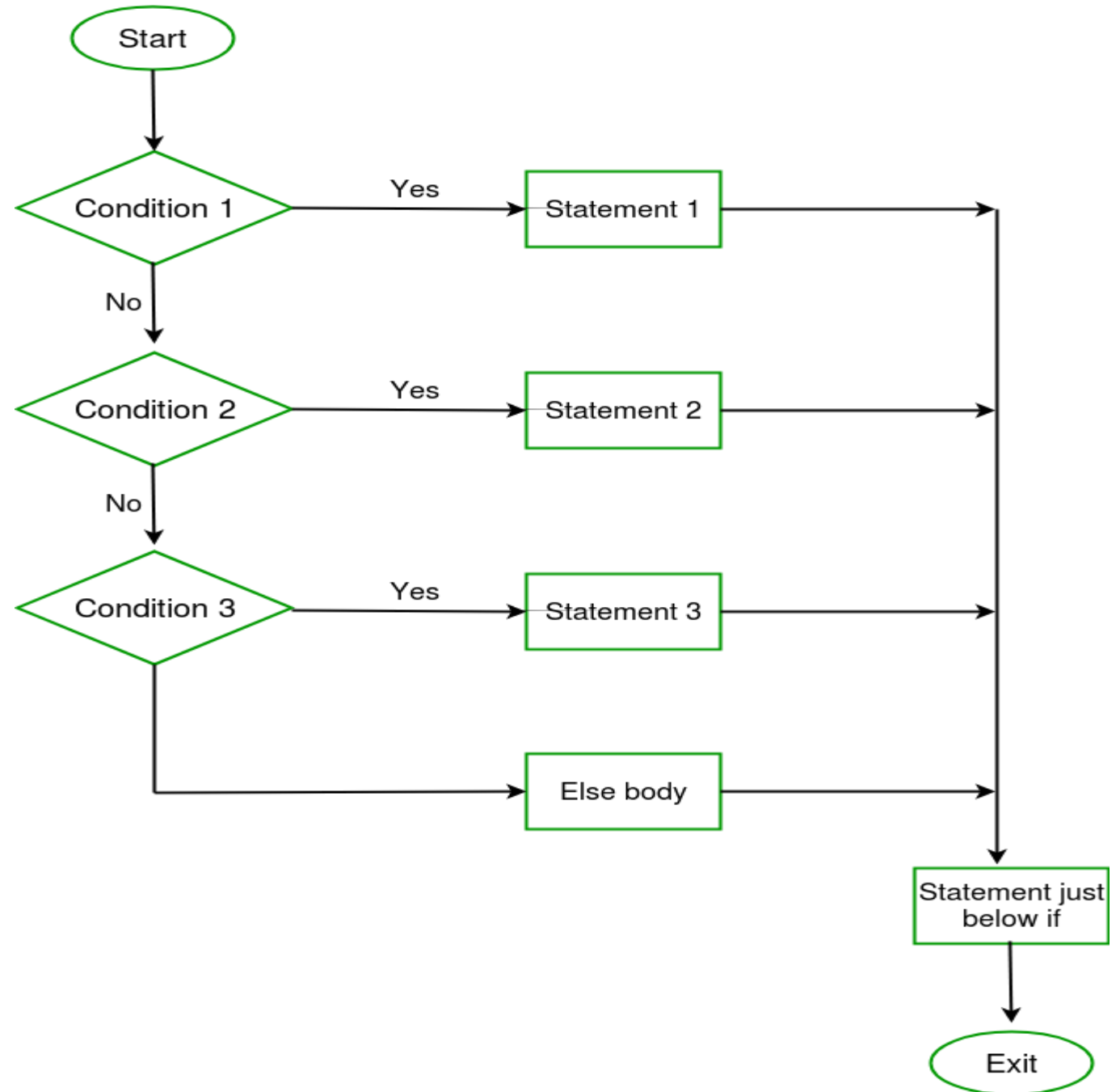


# if-else-if ladder in C

- If-else-if ladder is used when we have to choose one of the multiple options.
- if-else-if ladder is executed from the top down.
- As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed.
- If none of the conditions are true, then the final else statement will be executed.

# SYNTAX

```
if (condition)
 statement;
else if (condition)
 statement;
.
.
else
 statement;
```



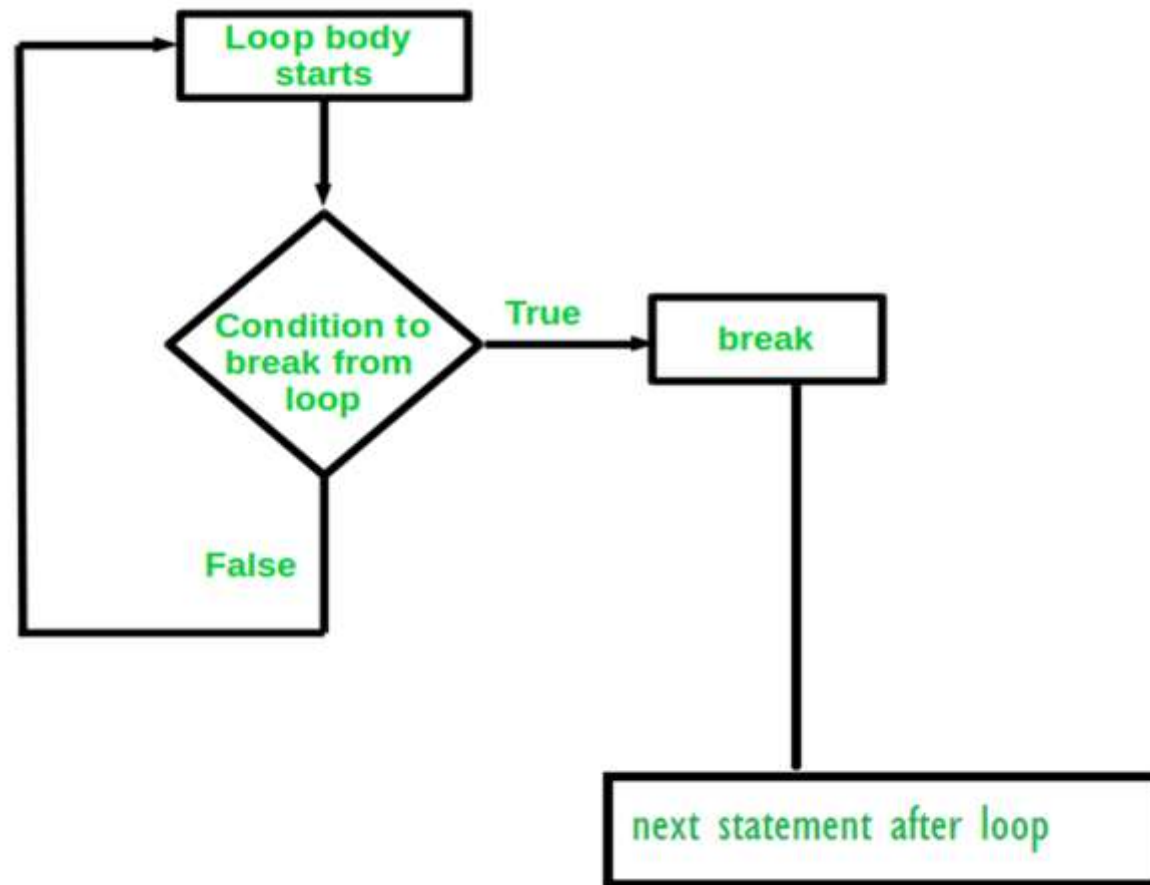
# Jump Statements in C

- These statements are used in C for unconditional flow of control in a program. C support four type of jump statements:
- **Break statement:**
  - This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stops there and control returns from the loop immediately to the first statement after the loop.
  - Basically break statements are used in the situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.

Syntax:

```
break;
```

# Flowchart of Break statement



# Jump Statements in C

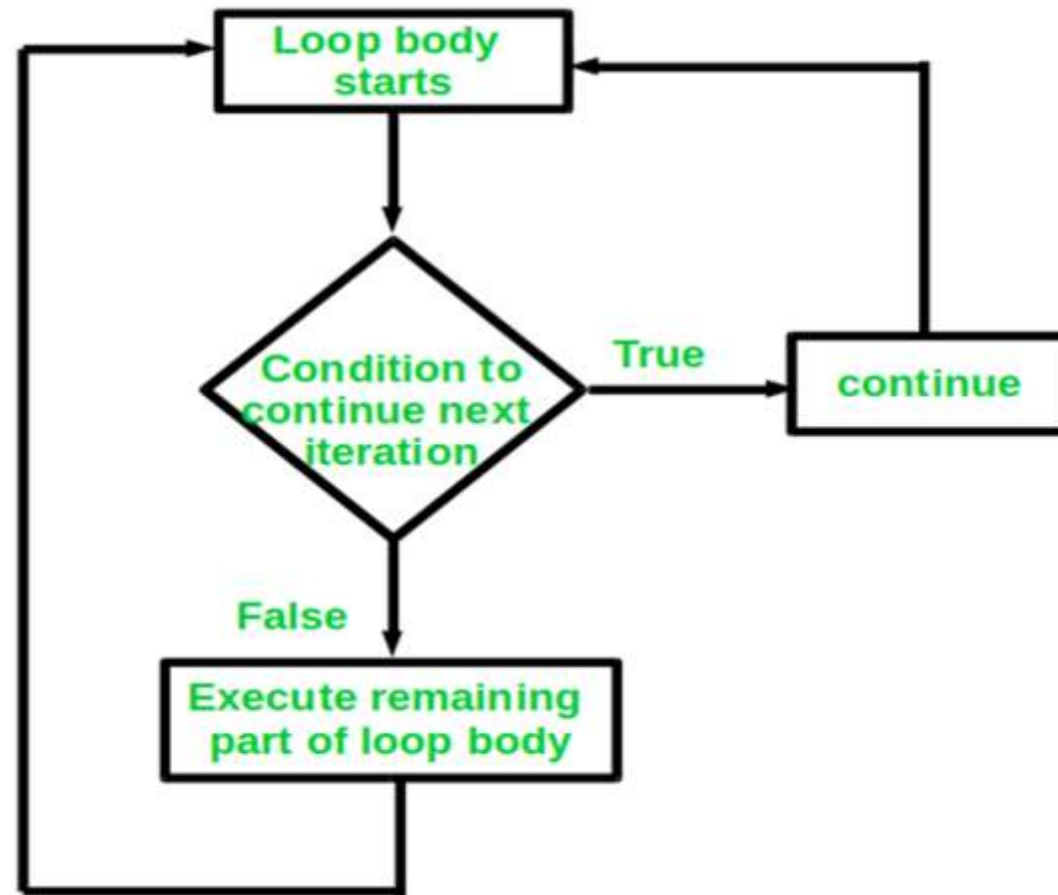
- **Continue Statement in C :**

- Continue is also a loop control statement just like the break statement. continue statement, instead of terminating the loop, forces to execute the next iteration of the loop.
- As the name suggest the continue statement forces the loop to continue or execute the next iteration.
- When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and next iteration of the loop will begin.

Syntax:

```
continue;
```

# Flowchart of Continue statement



# Jump Statements in C

- **goto statement:**

- The goto statement in C also referred to as unconditional jump statement.
- It can be used to jump from one point to another within a function.

- Syntax:

| Syntax1 |  | Syntax2 |
|---------|--|---------|
|---------|--|---------|

-----

|             |  |             |
|-------------|--|-------------|
| goto label; |  | label:      |
| .           |  | .           |
| .           |  | .           |
| .           |  | .           |
| label:      |  | goto label; |



# goto Statement

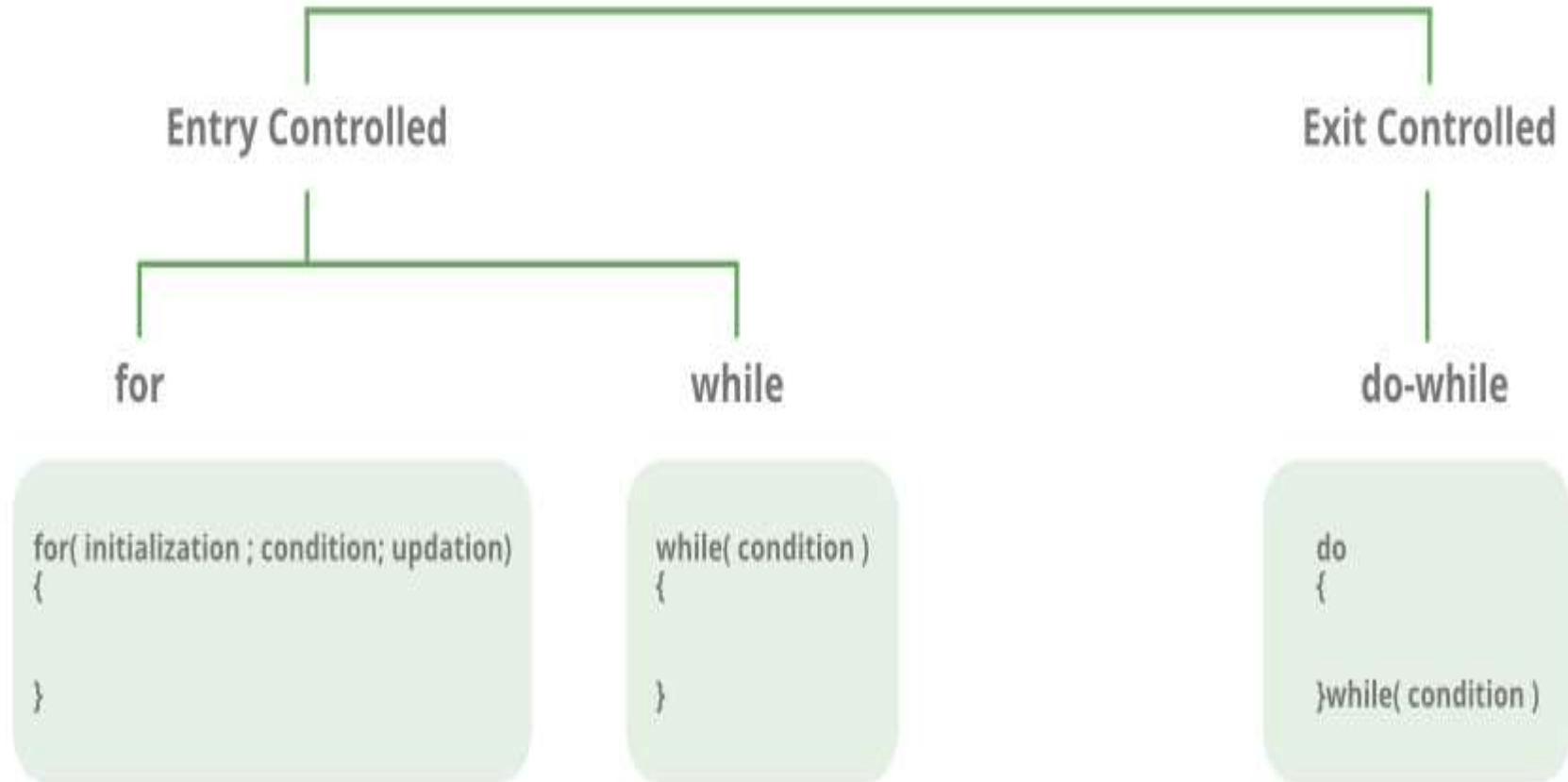
- In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label.
- Here label is a user-defined identifier which indicates the target statement.
- The statement immediately followed after 'label:' is the destination statement.
- The 'label:' can also appear before the 'goto label;' statement in the above syntax.

# Loops in C

- Loops in programming come into use when we need to repeatedly execute a block of statements.
- In computer programming, a loop is a sequence of instructions that is
- In Loop, the statement needs to be written only once and the loop will be executed until a certain condition is reached.
- An operation is done and then some condition is checked such as whether a counter has reached a prescribed number.
  - **Counter not Reached:** If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
  - **Counter reached:** If the condition has been reached, the next instruction “falls through” to the next sequential instruction or branches outside the loop.

- There are mainly two types of loops:
- **Entry Controlled loops:** In this type of loops the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.
- **Exit Controlled Loops:** In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute atleast once, irrespective of whether the test condition is true or false. **do – while loop** is exit controlled loop.

# Loops



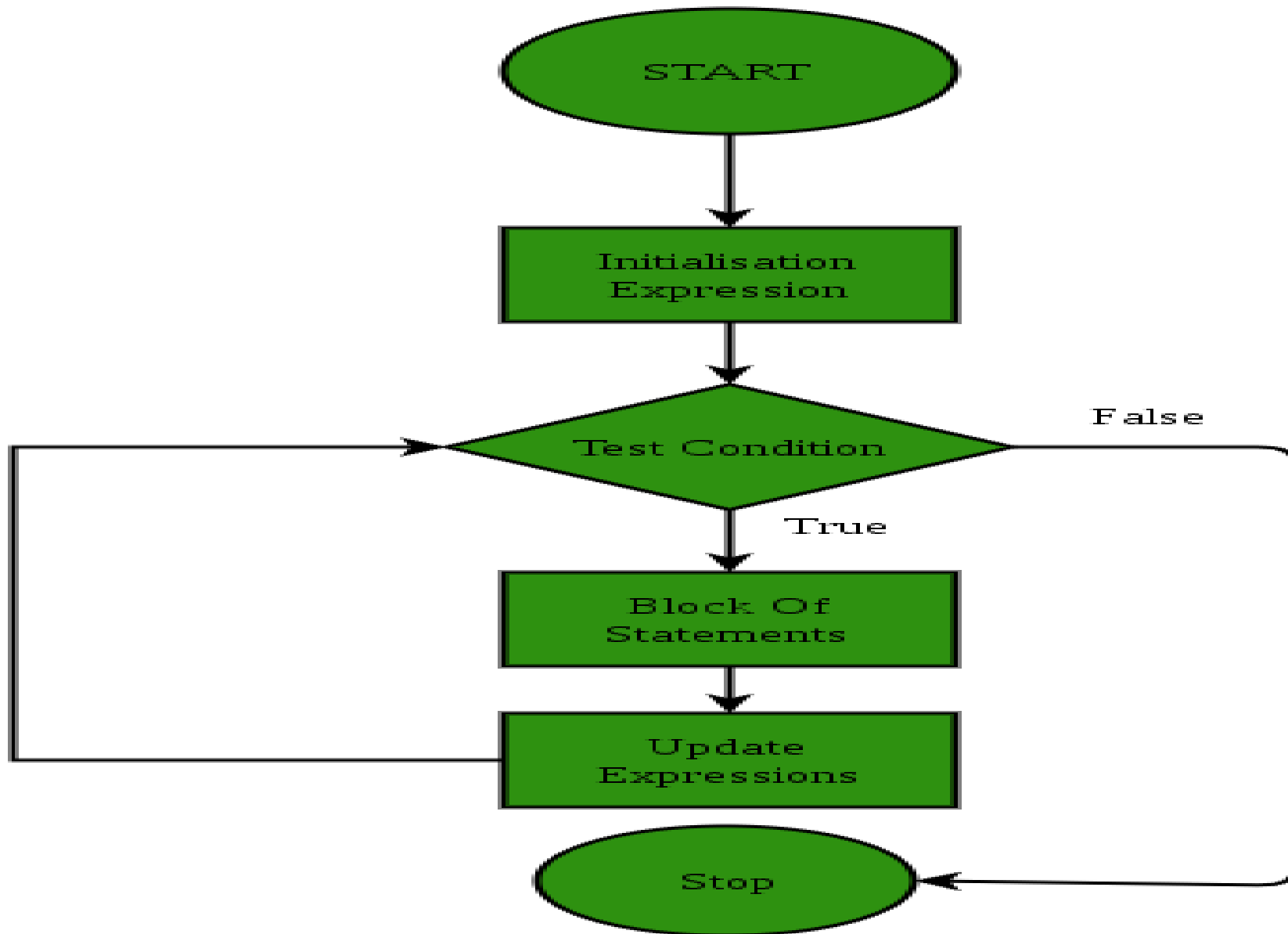
# for Loop

- A for loop is a repetition control structure which allows us to write a loop that is executed a specific number of times.
- Syntax:

```
for (initialization expr; test expr; update expr)
{
 // body of the loop
 // statements we want to execute
}
```

# For Loop

- In for loop, a loop variable is used to control the loop. First initialize this loop variable to some value, then check whether this variable is less than or greater than counter value. If statement is true, then loop body is executed and loop variable gets updated . Steps are repeated till exit condition comes.
- A for loop consists of:
  - Initialization Expression: In this expression we have to initialize the loop counter to some value. for example: `i=1;`
  - Test Expression: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: `i <= 10;`
  - Update Expression: After executing loop body this expression increments/decrements the loop variable by some value. for example: `i++;`



- // C program to illustrate for loop

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

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

```
void main()
```

```
{
```

```
 int i=0;
```

```
 clrscr();
```

```
 for (i = 1; i <= 10; i++)
```

```
 {
```

```
 printf("Hello World\n");
```

```
 }
```

```
 getch();
```

```
}
```



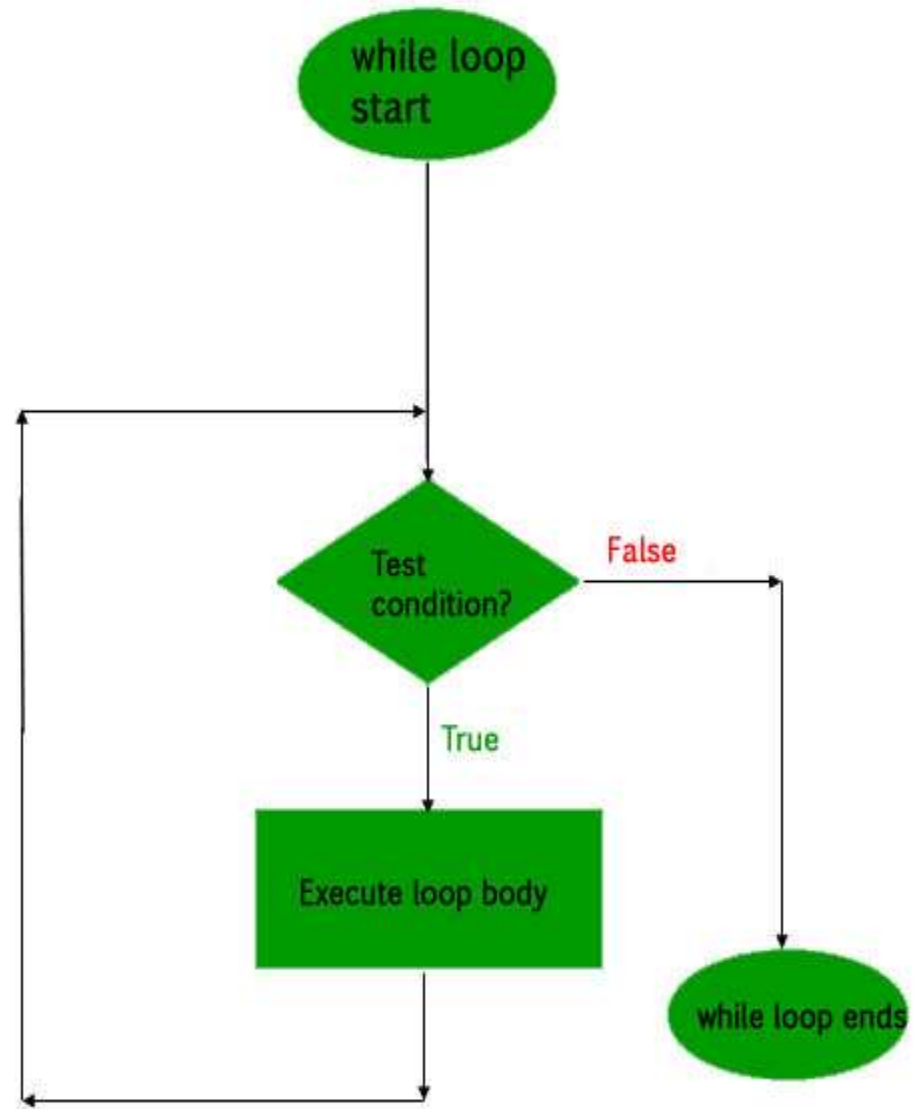
# While Loop

- In **for loop** the number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known to us.
- while loops are used in situations where we do not know the exact number of iterations of loop beforehand. The loop execution is terminated on the basis of test condition.

## Syntax:

```
initialization expression;
while (test_expression)
{
 // statements

 update_expression;
}
```



# While loop-example

C program to illustrate while loop

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

```
void main()
```

```
{
```

```
 int i ;
```

```
 clrscr();
```

```
// initialization expression
```

```
 i=1;
```

```
// test expression
```

```
while (i < 6)
```

```
{
```

```
 printf("Hello World\n");
```

```
 // update expression
```

```
 i++;
```

```
}
```

```
getch();
```

```
}
```

# do while loop

- In do while loops also the loop execution is terminated on the basis of test condition.
- The main difference between do while loop and while loop is in do while loop the condition is tested at the end of loop body, i.e do while loop is exit controlled whereas the other two loops are entry controlled loops.

**Note: In do while loop the loop body will execute at least once irrespective of test condition.**

- Syntax:

initialization expression;

do

{

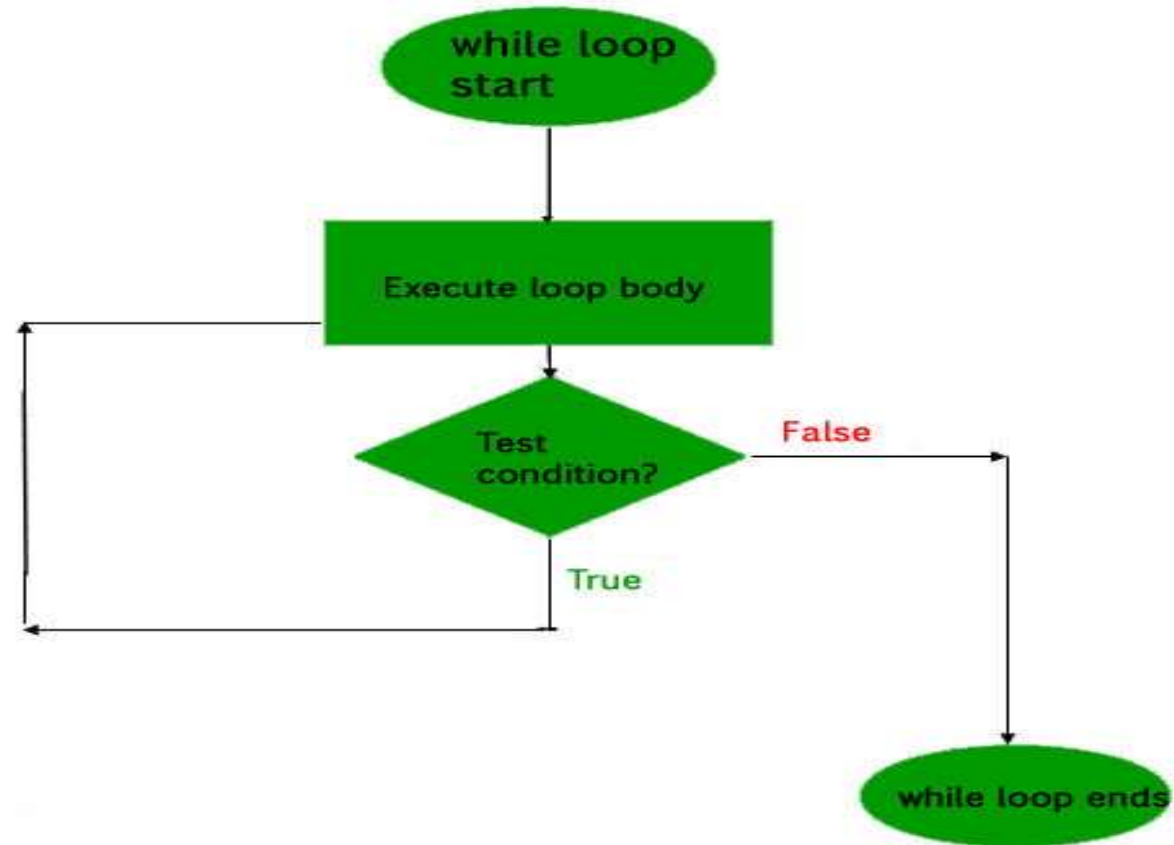
    // statements

    update\_expression;

} while (test\_expression);

- **Note: Notice the semi – colon(“;”) in the end of loop.**

# Flow Diagram:



# Example: do-while loop

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

```
void main()
```

```
{
```

```
 int i = 2; // Initialization expression
```

```
 clrscr();
```

```
 do
```

```
 {
```

```
 // loop body
```

```
 printf("Hello World\n");
```

```
 // update expression
```

```
 i++;
```

```
 } while (i < 1); // test expression
```

```
 getch();
```

```
}
```

# Nested for loops in C

- C programming allows to use one loop inside another loop.

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

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

// first for loop

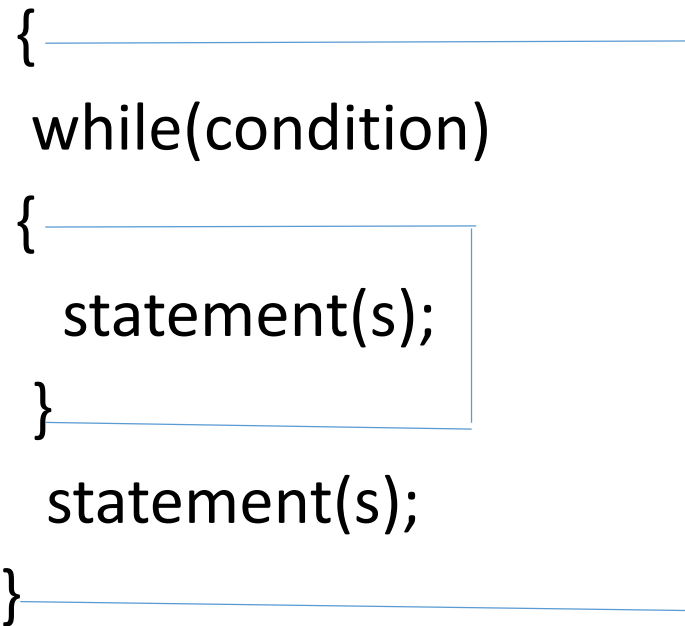


# nested while loop

- The syntax for a **nested while loop** statement in C programming language is as follows –

```
while(condition)
```

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



# nested do-while loop

- 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);
```

# nested do-while loop

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

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

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

```
void main ()
```

```
{ /* local variable definition */
```

```
 int i, j;
```

```
 for(i = 2; i<100; i++)
```

```
{
```

```
 for(j = 2; j <= (i/j); j++)
```

```
 if(!(i%j))
```

```
 break; // if factor found, not prime
```

```
 if(j > (i/j))
```

```
 printf("%d is prime\n", i);
```

```
 }
```

```
 getch();
```

```
}
```





