C Character Set

Valid set of characters used in c programming language .

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set.

C language character set contains the following set of characters...

1. Alphabets
2. Digits
3. Special Symbols

Alphabets

C language supports all the alphabets from english language.

Lower and upper case letters together supports 52 alphabets.

lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

Digits

C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Input Statemnt

Scanf():

Gf:

Scanf(“format specifier”,&variable1,&variable2,….);

Format specifier string specifies the format of data beeing received

Ex: scanf(“%d”,&x);

scanf(“%c”,&ch);

```
scanf(“%f”,&p);

scanf(“%s”,x);

scanf(“%d%d”,&p,&q);
```

printf()

This ia an output function. It is used to display a text message and to display the mixed type (int, float, char) of data on screen. The general syntax is as:

```
printf("control strings",&v1,&v2,&v3,................&vn);
```

or

```
printf("Message line or text line");
```

Expression

An expression is a valid arrangements of variables, constants, and operators.

Ex   20+20=40

OR

Expression is a combination of

1)Variable
2)Operands
3)Operators
4)Paranthesis

TYPES OF C OPERATORS:

C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators
7. Increment/decrement operators
8. Special operators

Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

Operator	Meaning	Example
+	Addition Operator	10 + 20 = 30
-	Subtraction Operator	20 – 10 = 10
*	Multiplication Operator	20 * 10 = 200
/	Division Operator	20 / 10 = 2
%	Modulo Operator	20 % 6 = 2

The modulus operator(%) finds the remainder of an integer division

6%2=0

15%4=3

7%8=7

12%15=12

2%3=2

C Relational Operators

A relational operator checks the relationship between two operands.

If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 return 0

What will be the output?

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
printf("%d",'A'<'B');
getch();
}
```

What will be the output?

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
printf("%d",10<5);
getch();
}
```

Logical operators

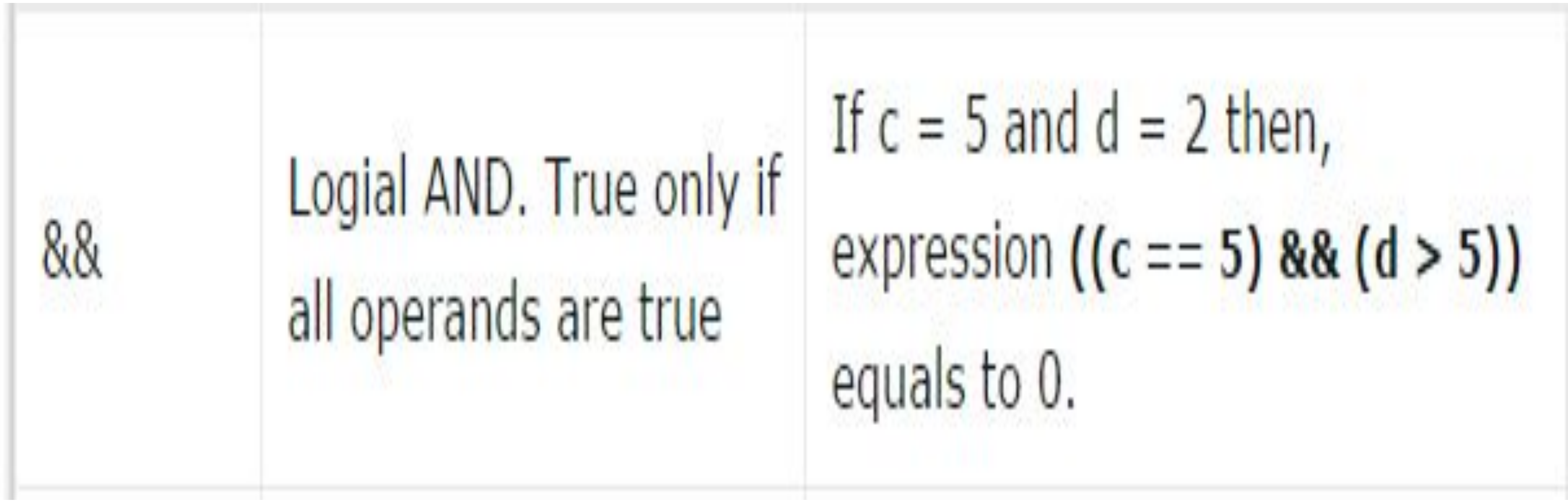
Logical operators are symbols, that are used to combine expressions containing relational operators.

Logical operators are:

&&	Logical AND operator
||	Logical OR Operator
!	Logical NOT Operator

Logical AND

Gf expression1 && expression 2



&&	Logial AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c == 5) && (d > 5)) equals to 0.

```
Logical AND (&&)
false && false: false
false && true : false
true && false : false
true && true  : true
```

Logical OR (||)

A compound expression is true when atleast one of two conditions is true

Gf: Expression1 || Expression2

The expression is true if aither expression1 or expression 2 is true

||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression **((c == 5) || (d > 5))** equals to 1.

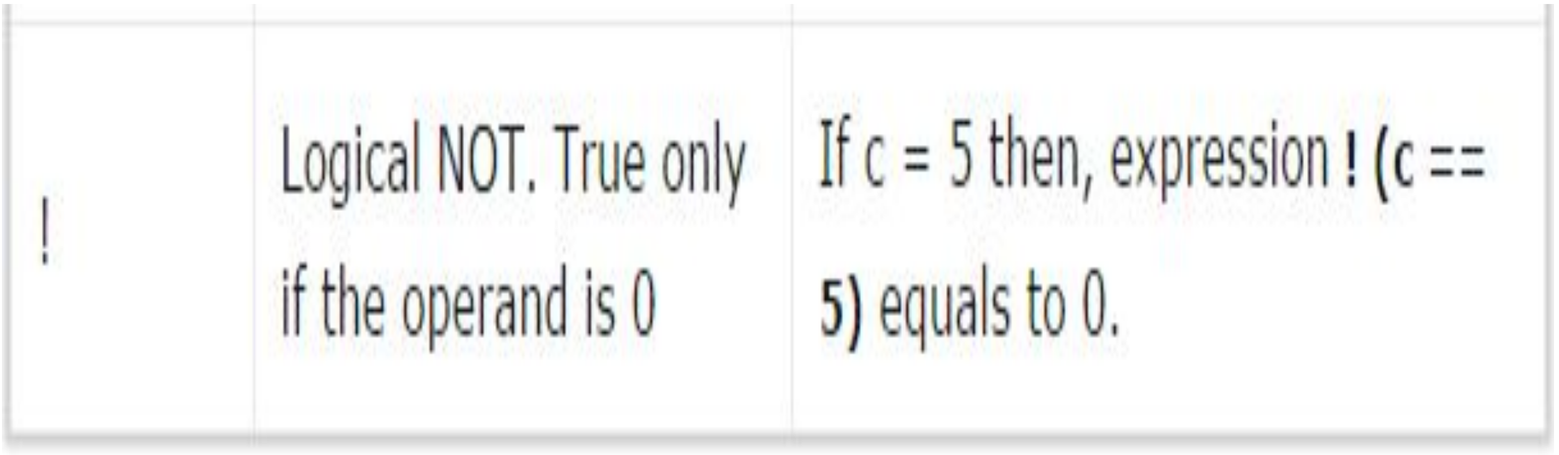
```
false || false:false
false || true : true
true  || false: true
true  || true : true
```

If the first expression is true ,then the entire expression will not be evaluated

Not operator

Changes an expression from false to true or from true to false

Gf !(expression)



!	Logical NOT. True only if the operand is 0	If c = 5 then, expression **! (c == 5)** equals to 0.

Assignment operators

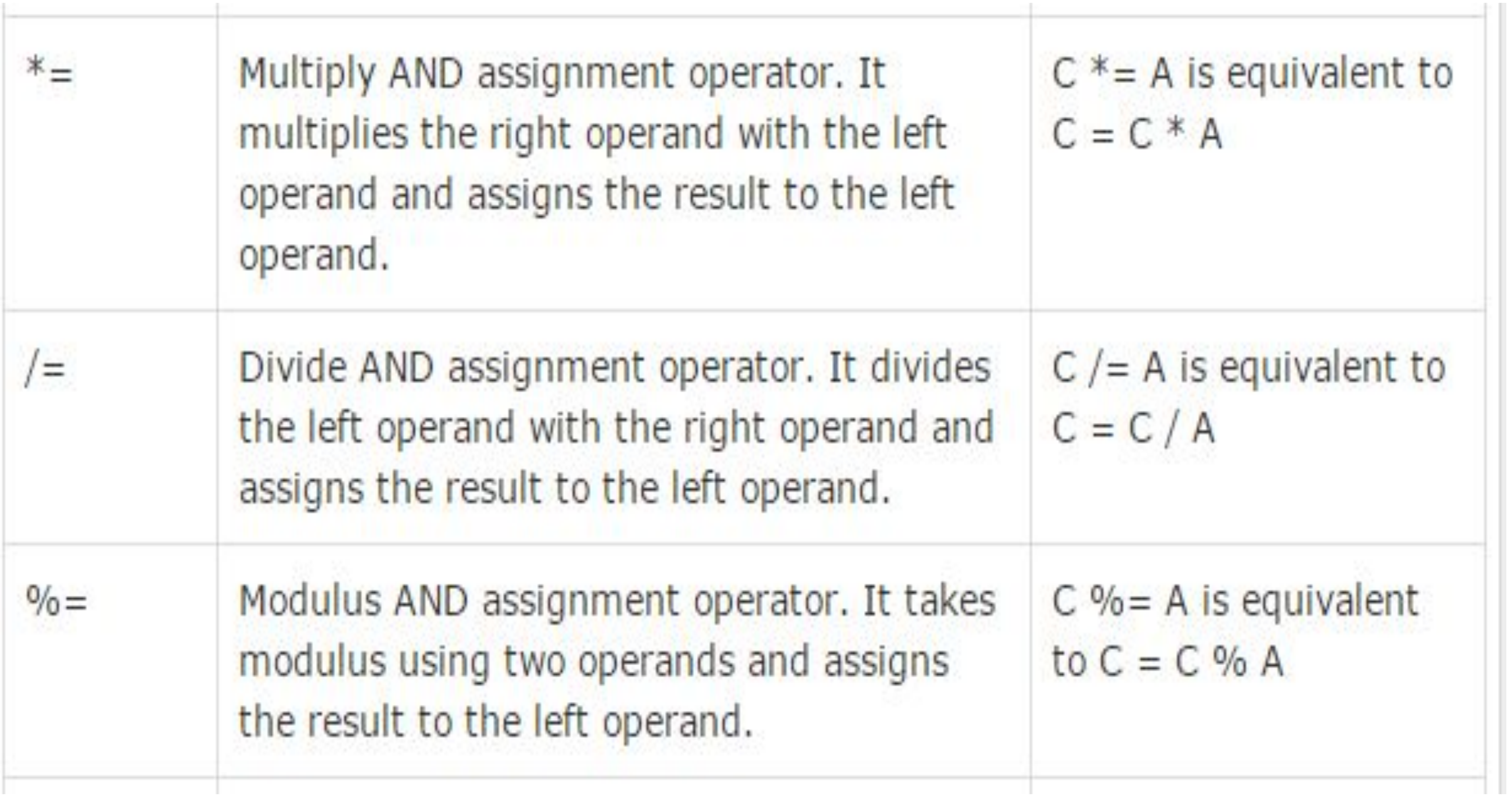
Is used to assign a value to a variable.

Gf variable name=constant or variable name or expression

Assignment statement assign value that is contained in the right of the equal sign to the variable on the left of the equal sign.

C permits multiple assignments in one line.

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
−=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C −= A is equivalent to C = C − A



*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

Increment and decrement operators

The operators ++ and -- called increment and decrement operator respectively. The use of these two operators will results in the value of the variable being incremented or decremented by unity. i=i+1 can be written as i++ and j=j-1 can be written as j—.

The operators can be place either before or after the operand. If the operator is placed before the variable (++I or --I) it is known as pre incrementing or pre decrementing.

If the operator is placed after the variable (I++ or I--) it is known as post incrementing or post decrementing.

Pre incrementing or pre decrementing or post incrementing or post decrementing have different effect when used in expression.

In the pre increment or pre decrement first the value of operand is incremented or decremented then it is assigned.

Control Statement

A program may contain one or more statement, which are usually executed in sequence. This type of executions are known as sequential execution

A statement that controls the sequential executions are known as control statements.
There may be a transfer of control from one part of the pgm to another part.

This transfer of control may be conditional or unconditional. The control statement is classified into 3 catagories:

1)conditional/Branching statement

2) Loop statement

3) Breaking control statement

conditional/Branching statement

These statements are mainly used for decision making.

these statements are also known as selection statement, because this statement select one or a group of statements from the whole statements.

a)If Statement

If statement performs a logical test and then takes one of the 2 possible actions depending on the outcome of the test.

```
Gf If(expression)
    {
     statement1;
     statement2;
     statement3;
       …………
      …………
      …………
     statement n;
    }
```

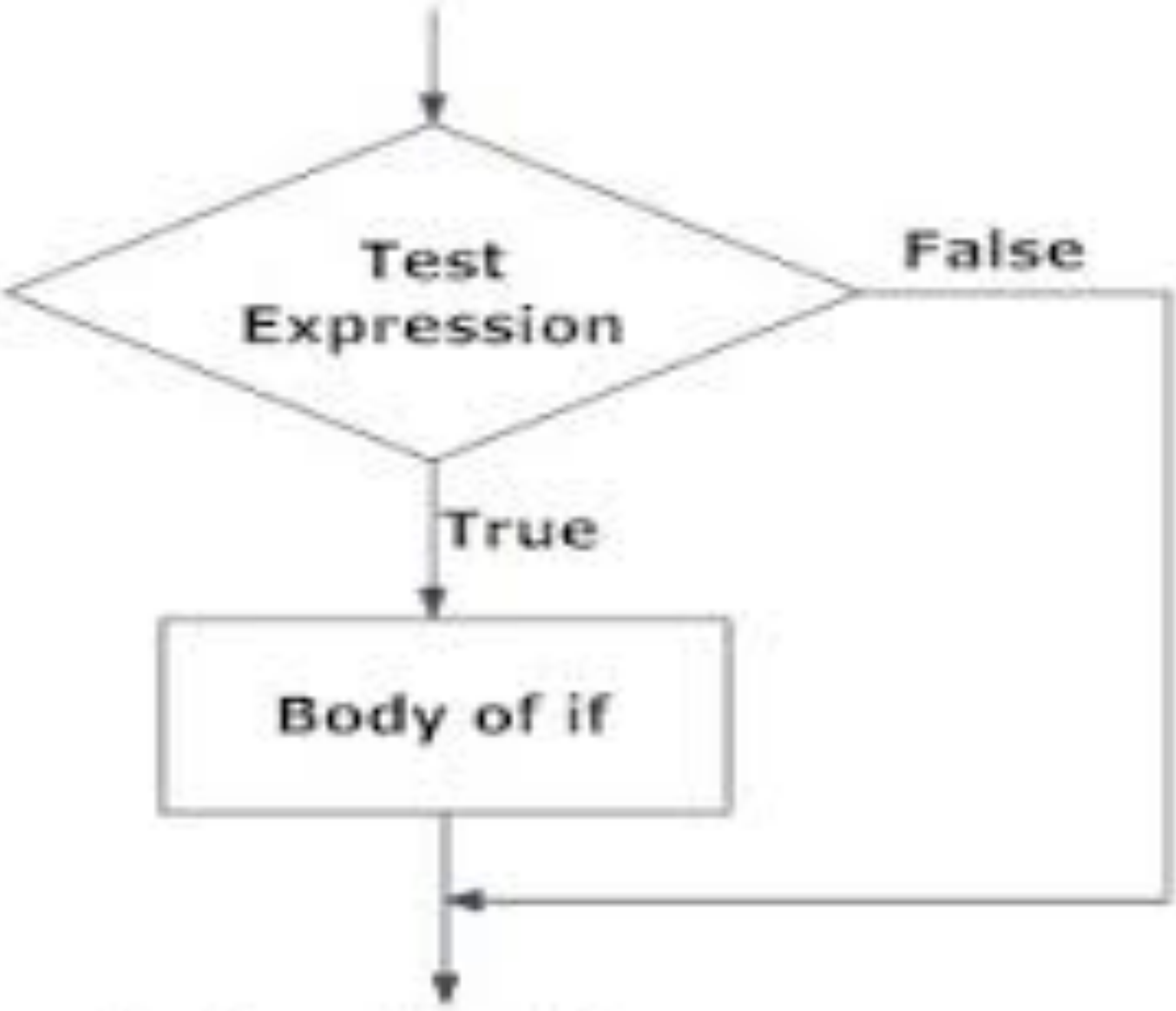


Fig: Operation of if statement

Write a program to find the difference of two numbers

Convert feet into inches.

Write a program to add two distances given in feet and inches

Write a program to print the result of a student

if...else statement

The if...else statement executes some code if the test expression is true (nonzero) and some other code if the test expression is false (0).

Syntax of if...else

```
if (testExpression)
 {
   // codes inside the body of if
}
Else
 {
   // codes inside the body of else
}
```

If test expression is true, codes inside the body of if statement is executed and, codes inside the body of else statement is skipped.

If test expression is false, codes inside the body of else statement is executed and, codes inside the body of if statement is skipped.

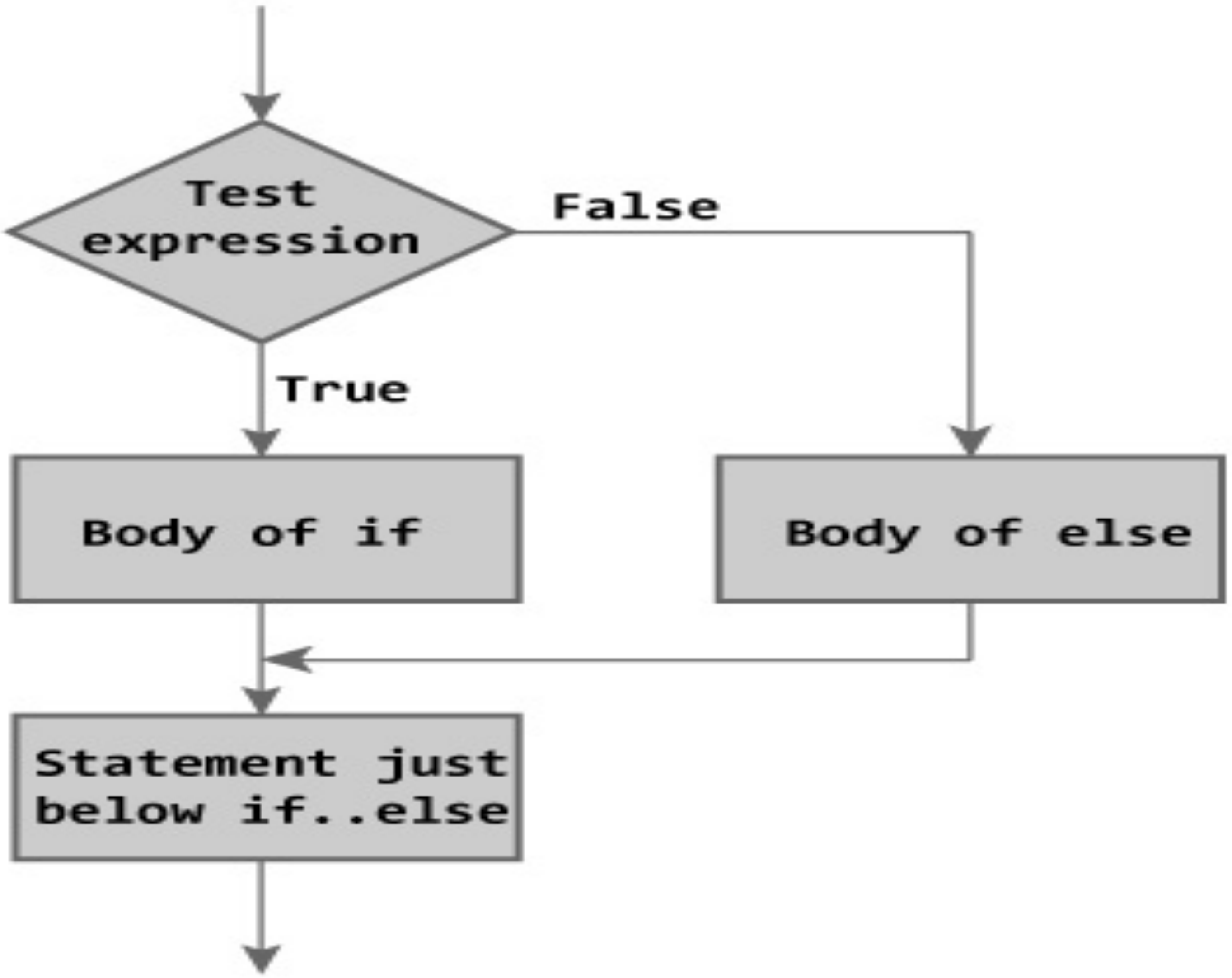


Figure: Flowchart of if...else Statement

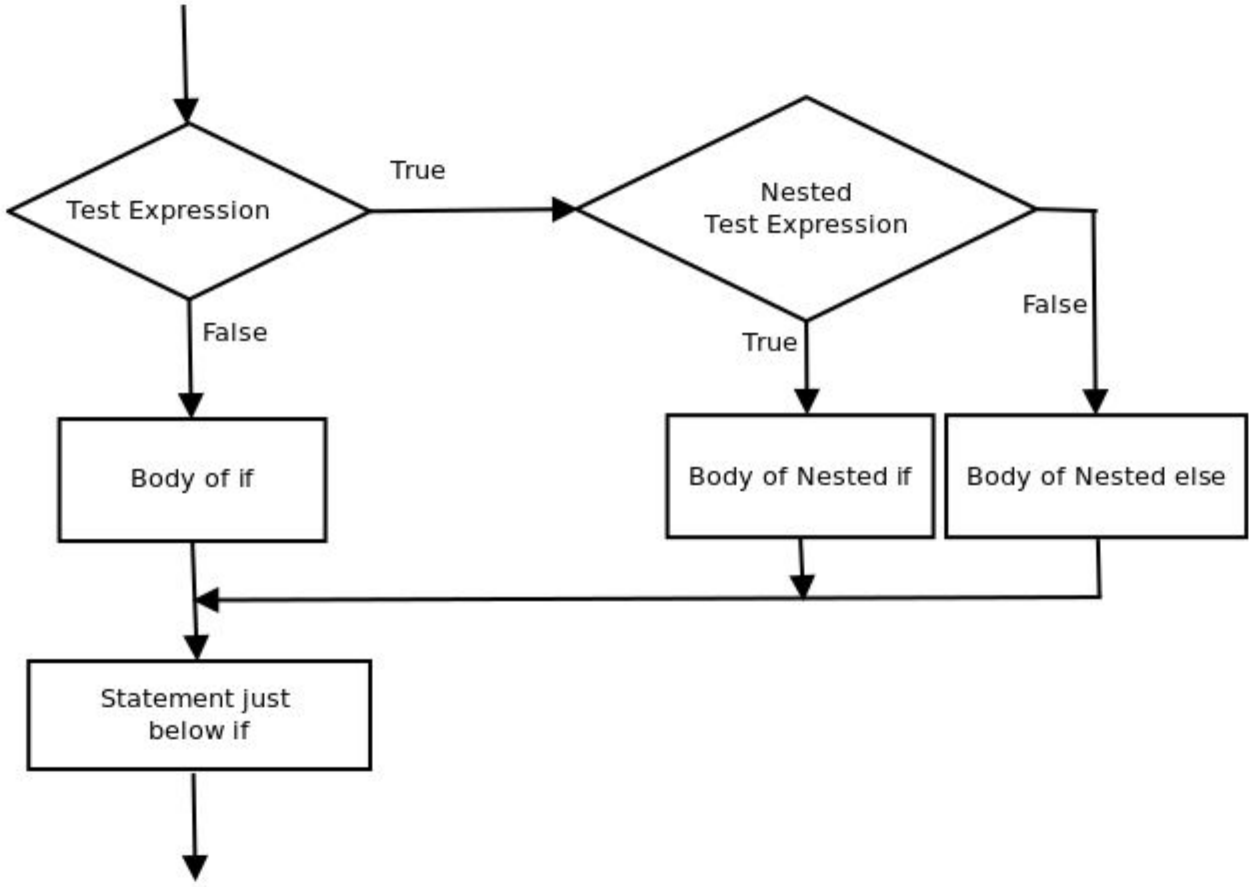
Nested if else

As the name suggests, nesting means writing if-else statements inside another if or else block. Basically there is no limit of nesting. But generally programmers nest up to 3 blocks only.

The nested if else can have one of the following three forms:

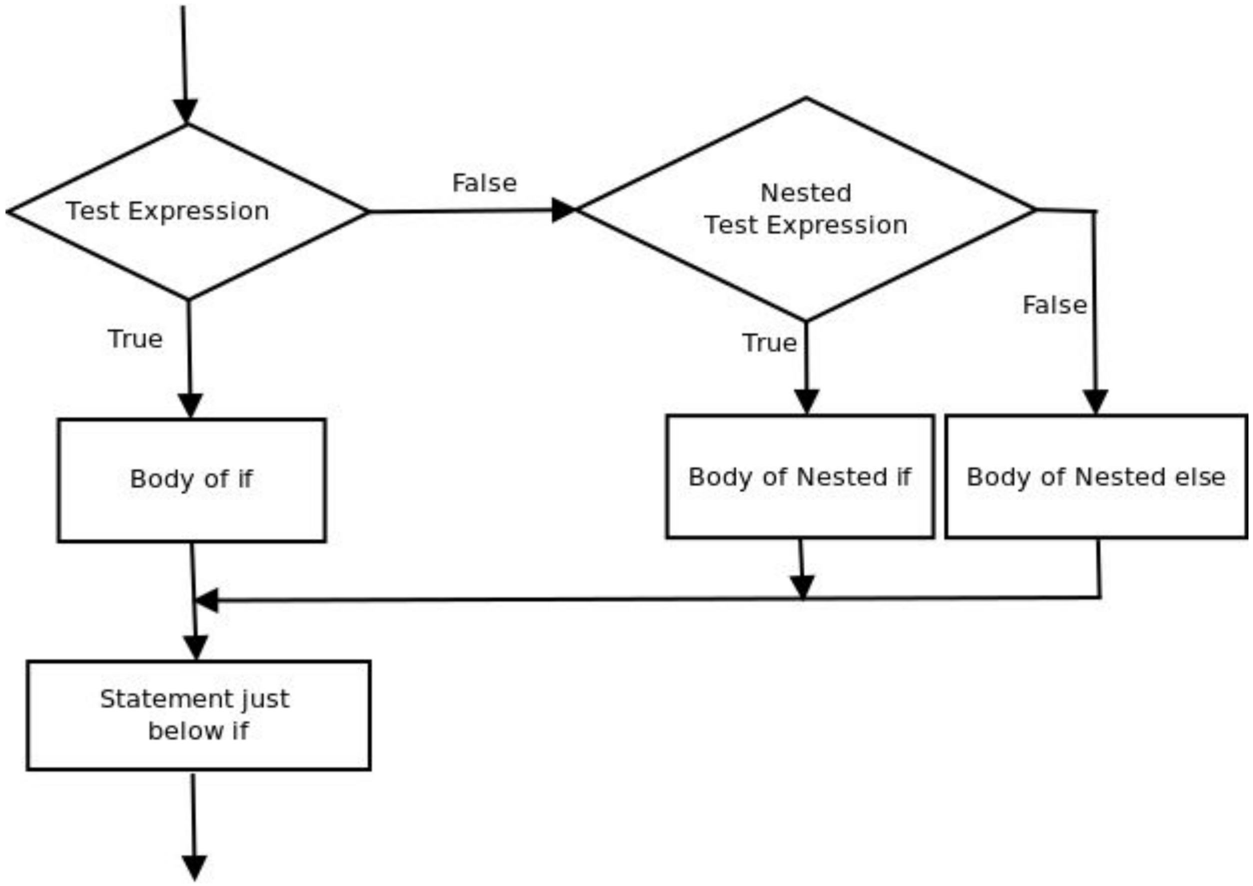
General form:1

```
if (expression 1)
      if (expression 2)
              statement 1
      else
                 statement 2
Else
       Statement block 3
```


Test Expression
True
Nested
Test Expression
False
True
False
Body of if
Body of Nested if
Body of Nested else
Statement just
below if

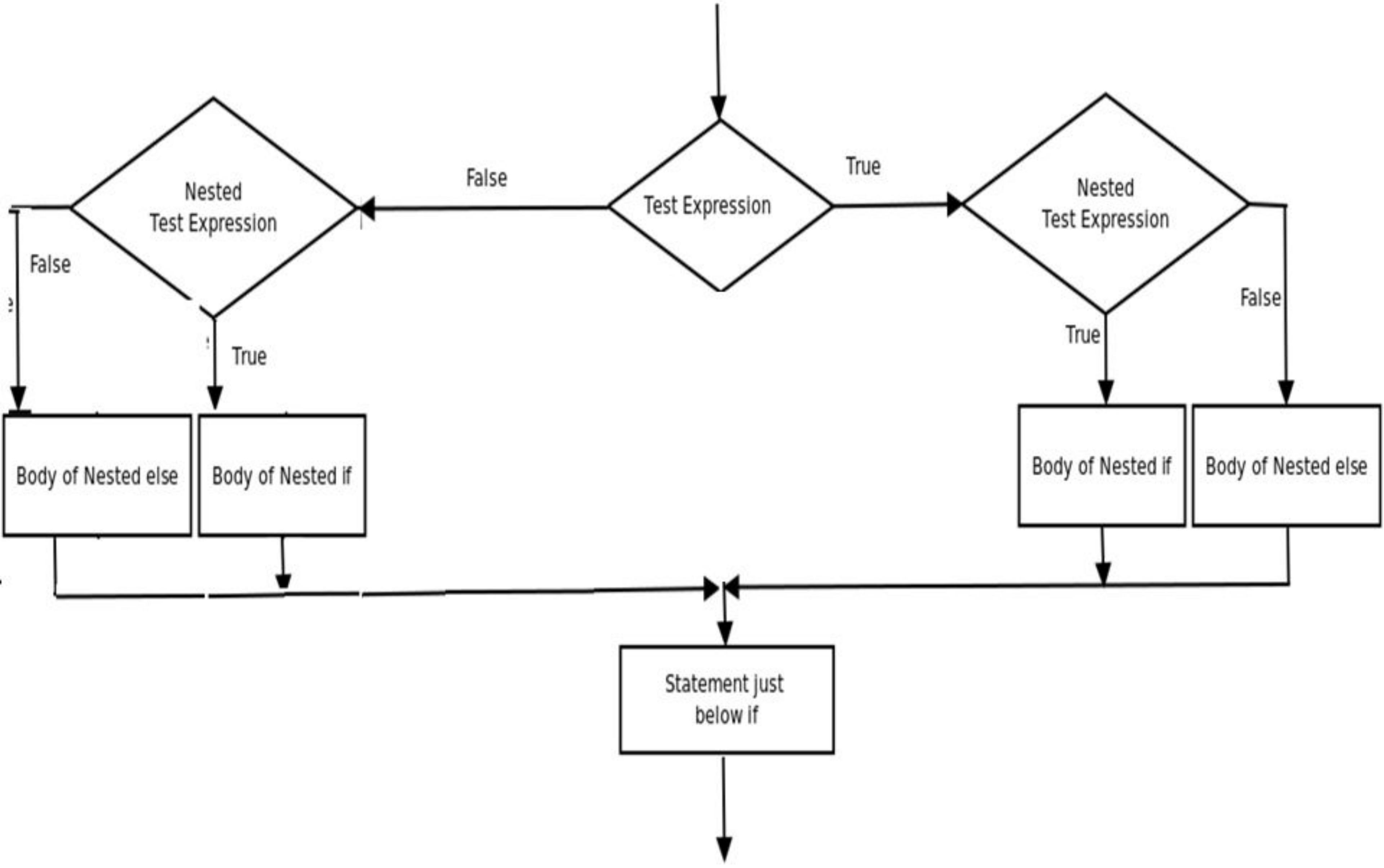
General form: 2

```
if (expression 1)
     statement 1
else
      if (expression 2)
          statement 2
        else
             statement 3
```


Test Expression
False
Nested Test Expression
True
True
False
Body of if
Body of Nested if
Body of Nested else
Statement just below if

General form: 3

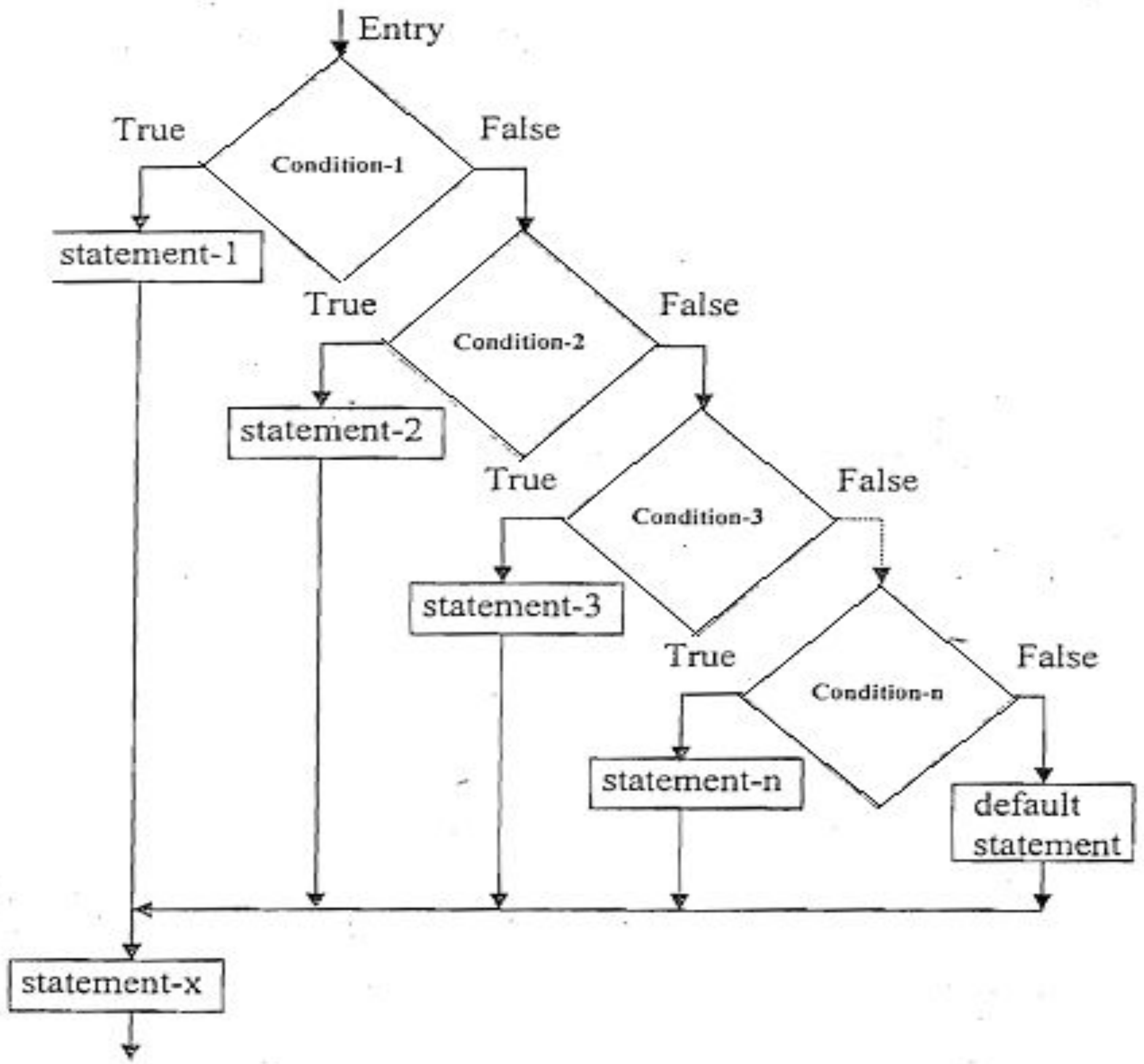
```
if (expression 1)
            if (expression 2)
                statement 1
            else
                statement 2
Else
            if (expression 3)
                statement  3
            else
               statement 4
```


Test Expression
False
True
Nested
Test Expression
Nested
Test Expression
False
True
True
False
Body of Nested else
Body of Nested if
Body of Nested if
Body of Nested else
Statement just
below if

if else ladder statement

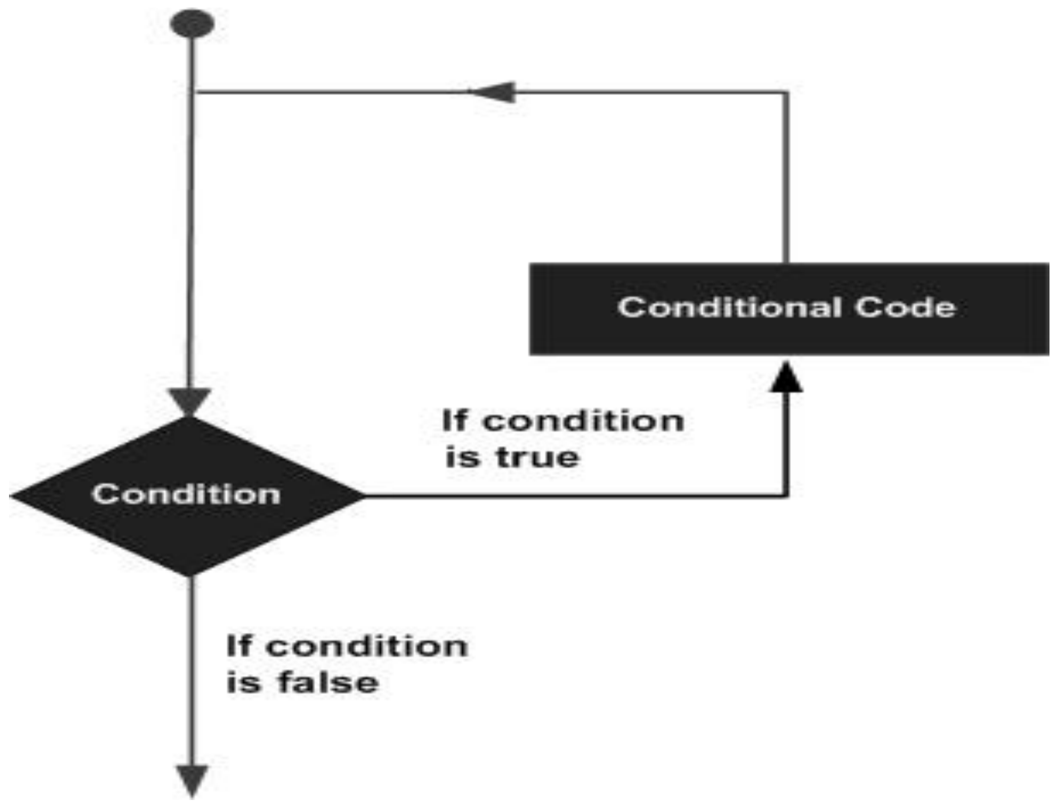
The if else ladder statement is used to test set of conditions in sequence. An if condition is tested only when all previous if conditions in if-else ladder is false. If any of the conditional expression evaluates to true, then it will execute the corresponding code block and exits whole if-else ladder.

```
if(expression 1)
{
   statement-block1;
}
else if(expression 2)
{
   statement-block2;
}
else if(expression 3)
{
   statement-block3;
}
else
   default-statement;
```


Entry
True
Condition-1
False
statement-1
True
Condition-2
False
statement-2
True
Condition-3
False
statement-3
True
Condition-n
False
statement-n
default statement
statement-x

Loop statement

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



There are three loops in C programming:

for loop
while loop
do...while loop

for loop

The syntax of for loop is:

```
for (initialization; testExpression; updateStatement)
{
      // codes
}
```

The initialization statement is executed only once.

Then, the test expression is evaluated. If the test expression is false (0), for loop is terminated. But if the test expression is true (nonzero), codes inside the body of for loop is executed and the update expression is updated.

This process repeats until the test expression is false.

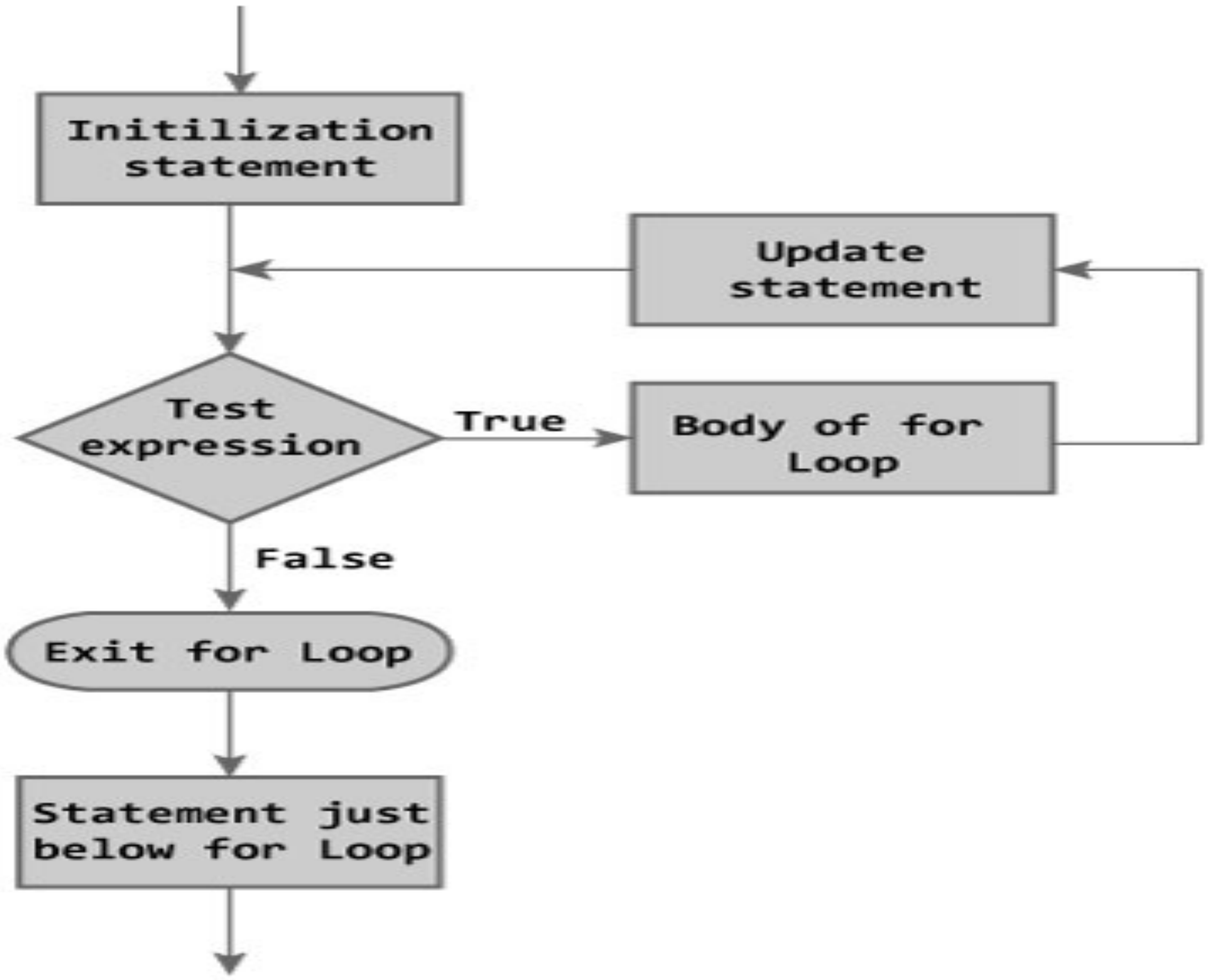
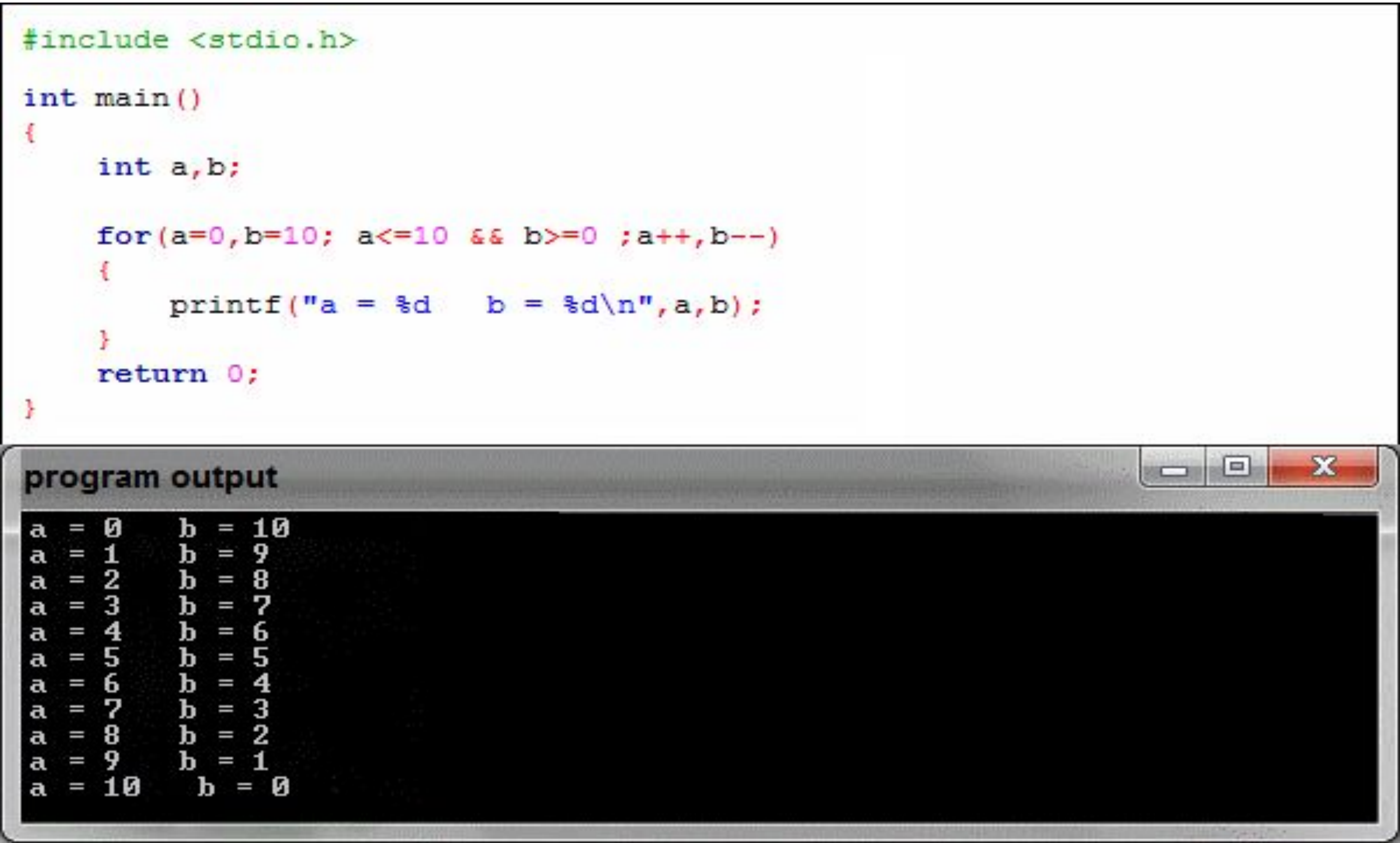


Figure: Flowchart of for Loop

For Loop (Multiple Conditions)

The comma operator is used to define multiple expressions in for loop.



```
#include <stdio.h>

int main()
{
    int a,b;

    for(a=0,b=10; a<=10 && b>=0 ;a++,b--)
    {
        printf("a = %d   b = %d\n",a,b);
    }
    return 0;
}
```

```
a = 0    b = 10
a = 1    b = 9
a = 2    b = 8
a = 3    b = 7
a = 4    b = 6
a = 5    b = 5
a = 6    b = 4
a = 7    b = 3
a = 8    b = 2
a = 9    b = 1
a = 10    b = 0
```

While Loop

The while loop in C language is used to execute a block of code several times until the given condition is true.

The while loop is mainly used to perform repetitive tasks.

Unlike for loop, it doesn't contains any initialization and update statements.

The syntax of a while loop is:

```
while (testExpression)
{
   //codes
}
```

The while loop evaluates the test expression.

If the test expression is true (nonzero), codes inside the body of while loop are exectued. The test expression is evaluated again. The process goes on until the test expression is false.

When the test expression is false, the while loop is terminated.

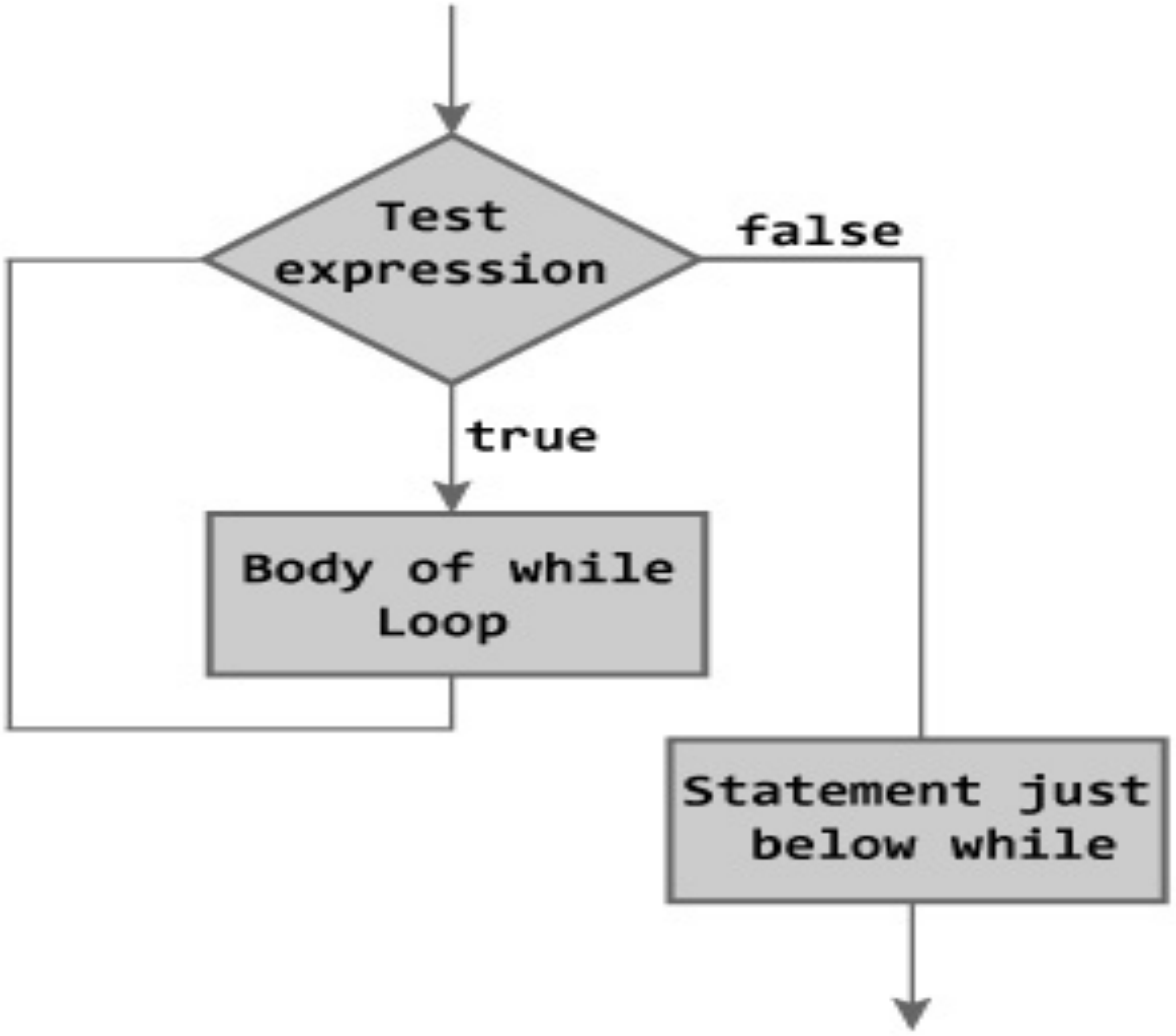


Figure: Flowchart of while Loop

While loop is an entry controlled loop i.e. the condition is checked before entering into the loop.

So if the condition is false for the first time, the statements inside while loop may not be executed at all.

The condition to be checked can be changed inside loop by changing values of variables. When the condition becomes false, the program control exits from the loop.

do while loop in C

The code given between the do and while block will be executed until condition is true.

In do while loop, statement is given before the condition, so *statement or code will be executed at lease one time*. In other words, we can say it is executed 1 or more times.

```
do
{
  // codes
}
while (testExpression);
```

The code block (loop body) inside the braces is executed once.

Then, the test expression is evaluated. If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to 0 (false).

When the test expression is false (nonzero), the do...while loop is terminated.

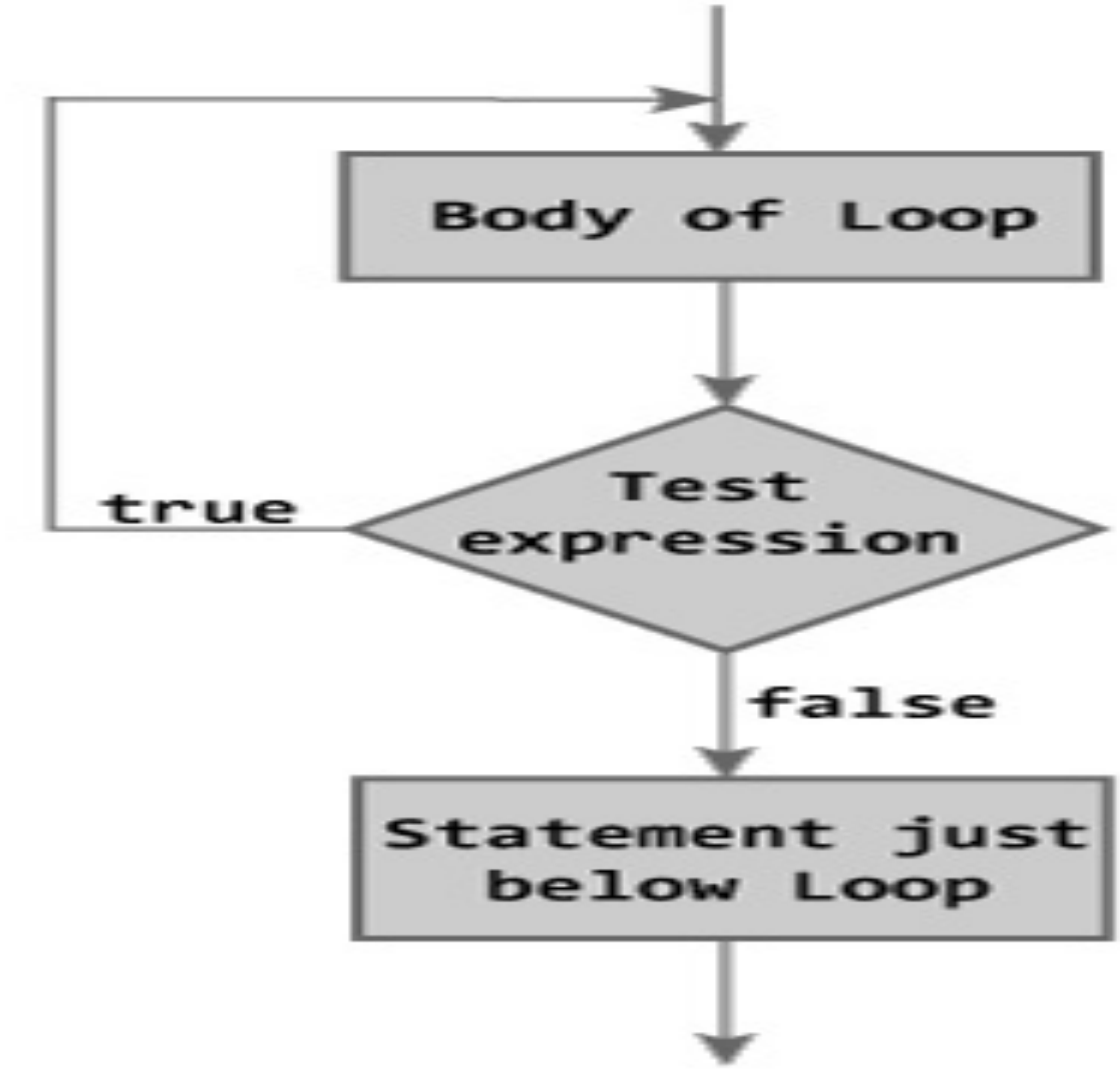


Figure: Flowchart of do...while Loop

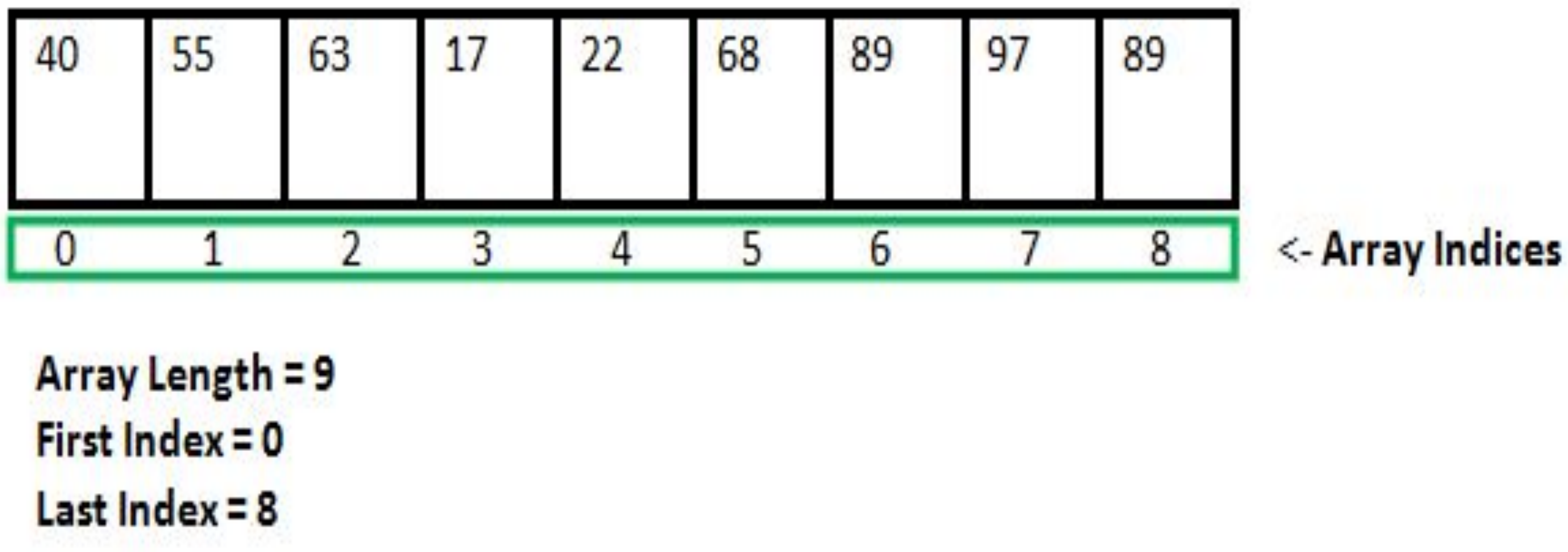
Do-while loop is an exit controlled loop i.e. the condition is checked at the end of loop.

It means the statements inside do-while loop are executed at least once even if the condition is false.

Do-while loop is an variant of while loop. In order to exit a do-while loop either the condition must be false or we should use break statement.

Arrays

An array is a collection of elements of the same type that are referenced by a common name.



Basically, there are two types of arrays:

One dimensional array

Multidimensional array

One Dimensional Array:

Declaration:

```
dataType arrayname[arraySize];

int mark[5];
```

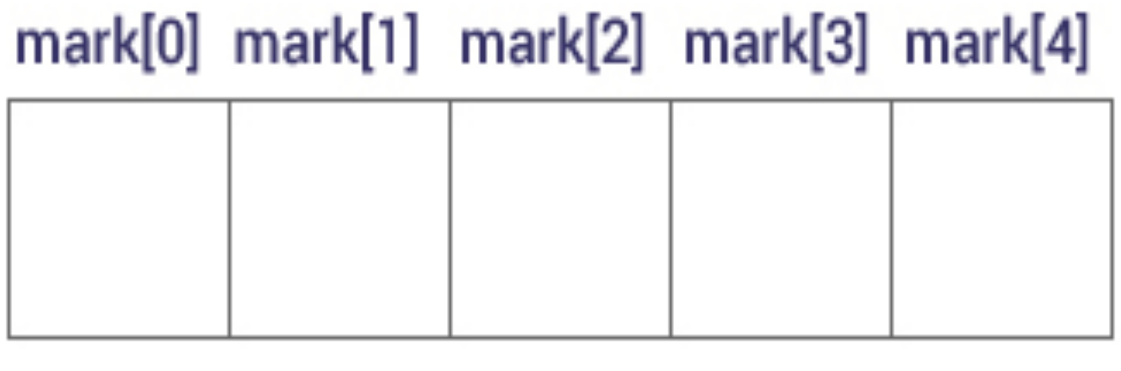
Access Array Elements

You can access elements of an array by indices.

Suppose you declared an array mark .

The first element is mark[0]

second element is mark[1] and so on.



Notes:

Arrays have 0 as the first index, not 1. In this example, mark[0] is the first element.

If the size of an array is n, to access the last element, the n-1 index is used. In this example, mark[4]

Suppose the starting address of mark[0] is 2120.
Then, the address of the mark[1] will be 2122.
the address of mark[2] will be 2124 and so on.

This is because the size of a int is 2 bytes.

initialize an array

Gf:

Datatype array-name[size]={value1,value2,…..,value size-1}

Example 1:

```
int mark[5] = {19, 10, 8, 17, 9};
```

Example 2:

```
int mark[] = {19, 10, 8, 17, 9};
```

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.

```
int x[5]={1,2,3,4,5,6,7,8,9,10};
```

Error: Too many initializers

```
int x[5]={1,2,3,4,5};
clrscr();
for(i=0;i<10;i++)
printf("%d",x[i]);
```

Output:

```
123450344000
```

```
int x[10]={1,2,3,4,5};
clrscr();
for(i=0;i<10;i++)
printf("%d",x[i]);
```

Output:

```
1234500000
```

Change Value of Array elements

```
int mark[5] = {19, 10, 8, 17, 9}

// make the value of the third element to -1
mark[2] = -1;



// make the value of the fifth element to 0
mark[4] = 0;
```

1. Write a program to read n elements and then find the sum of array elements.
2. Write a program to read n elements and then find the average of array elements.
3. Write a program in C to copy the elements of one array into another array.
4. Program to print an array in reverse order
5. C Program to Put Even & Odd Elements of an Array in 2 Separate Arrays
6. C Program to Sort the Array in an Ascending Order
7. Write a program in C to find the maximum and minimum element in an array

Two Dimensional Array

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

The array **int x[10][20]** can store total (10*20) = 200 elements.

Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

declaring a two-dimensional array of size x, y:

Syntax:

```
data_type array_name[x][y];
```

Example

```
int x[10][20];
```

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initializing Two – Dimensional Arrays:

There are two ways in which a Two-Dimensional array can be initialized.

First Method:

int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}

The above array have 3 rows and 4 columns.

The elements will be filled in the array in the order, first 4 elements from the left in first row, next 4 elements in second row and so on

Better Method:

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
```

This type of initialization make use of nested braces.

Each set of inner braces represents one row. In the above example there are total three rows so there are three sets of inner braces.

```
int Employees[ ][ ] = { {10, 20, 30}, {15, 25, 35},
{22, 44, 66}, {33, 55, 77} };
```

Here, We haven’t mentioned the row size and column size.

The compiler calculate the size by checking the number of elements inside the row and column.

We can also write this two dimensional array as:

```
int Employees[ ][3] = { {10, 20, 30}, {15, 25, 35},
{22, 44, 66}, {33, 55, 77} };
```

Function

A **function** is a subprogram that is used to perform a predefined operation and optionally return a value.

Using functions,
we can avoid repetitive coding in programs
speed up program development.

The C language provides two types of functions:

1) **library functions**
2)user-defined functions.

Elements of user defined functions:

Function definition
Function call
Function declaration

Definition of functions:

A function also known as function implementation shall include the following elements.

- Function name
- Return type or function type
- List of parameters
- Local variable declarations
- Function statements and
- A return statement

All the 6 elements are grouped into two parts, namely

1)Function Header (first 3 elements)
2)Function body(second 3 elements)

Gf:

```
return_type function_name (parameter list)
    {
      local variable declaration;
    executable statement1;
    executable statement2;
                    .
                    .
      return statement;
     }
```

Function Call

A function can be called by simply using the function name followed by a list of actual parameters, if any, enclosed in parentheses.

Example.

```
main()
{
int y;
y=mul(10,5);                /*Function Call*\
printf(“%d\n”,y);
}
```

Function Declaration or Function prototype declaration

Like variables, all functions in a C program must be declared, before they are invoked. A function declaration consists of four parts.

- Function type or return type
- Function name
- Parameter list
- Terminating by semicolon

GF:

Function-type function-name (parameter list);

Ex:

```
int   xy  (int  x,int y);    //function prototype
```

Points:

- The parameter list must be separated by commas.
- The parameter names do not need to be the same in the prototype declaration and the function definition.
- The types must match the types of parameters in the function definition, in number and order.
- Use of parameter names in the prototype is optional.

- If the function has no formal parameters, the list is written as void.
- The return type must be void if no value is returned.
- When the declaration types do not match with the types in the function definition, compiler will produce an error.

Arguments (parameters) are the variables or expressions passed from the caller of a function into the function

1)Actual arguments:

2)Formal argument

Actual arguments:

The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function.

Formal arguments:

The formal arguments are the parameters/arguments in a function declaration or definition.

The scope of formal arguments is local to the function definition in which they are used.

Formal arguments belong to the called function.

Formal arguments are a copy of the actual arguments.

A change in formal arguments would not be reflected in the actual arguments.

RETURN VALUES AND THEIR TYPES

A function may or may not return a value. A return statement returns a value to the calling function and assigns to the variable in the left side of the calling function.

If a function does not return a value, the return type in the function definition and declaration is specified as void.
If it does, it is done through the return statement.

The return statement can take one of the following forms

Return;
 Or
Return (expression);
 The first, the ‘plain’ return does not return any value;
When a return is encountered, the control is immediately passed back to the calling function.

category of functions

For better understanding of arguments and return value from the function, user-defined functions can be categorized as:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value.

1.Function with no arguments and no return value

```
#include <stdio.h>
#include <string.h>
void sum();
void main()
{
sum();
getch();
}
void sum()
{
int x=30,y=50,sum;
sum=x+y;
printf("sum=%d",sum);
getch();
}
```

```
#include <stdio.h>
#include <string.h>
void sum(void);
void main()
{
sum();
getch();
}
void sum(void)
{
int x=30,y=50,sum;
sum=x+y;
printf("sum=%d",sum);
}
```

2.Function with no arguments and a return value

```
#include <stdio.h>
#include <string.h>
int sum(void);
void main()
{
int x;
clrscr();
x=sum();
printf("\nsum=%d",x);
getch();
}
int sum(void)
{
int x=30,y=50,sum;
sum=x+y;
printf("sum=%d",sum);
return(sum);
}
```

3.Function with arguments and no return value

```c
#include <stdio.h>
#include <string.h>
void sum(int);
void main()
{
int a=70,x;
clrscr();
sum(a);
printf("\nsum=%d",x);
getch();
}
void sum(int y)
{
int x=30,sum;
sum=x+y;
printf("sum=%d",sum);
}
```

```
#include<stdio.h>
float square ( float x );
int main( )
{
   float m, n ;
   printf ( "\nEnter some number for finding square \n");
   scanf ( "%f", &m ) ;
   // function call
   n = square ( m ) ;
   printf ( "\nSquare of the given number %f is %f",m,n );
   getch();
   return;
}

float square ( float x )   // function definition
{
   float p ;
   p = x * x ;
   return ( p ) ;
}
```

```c
#include<stdio.h>
void swap(int a, int b);
int main()
{
   int m = 22, n = 44;
   printf(" values before swap  m = %d \nand n = %d", m, n);
   swap(m, n);
}
void swap(int a, int b)
{
   int tmp;
   tmp = a;
   a = b;
   b = tmp;
   printf(" \nvalues after swap m = %d\n and n = %d", a, b);
}
```

4.Function with arguments and a return value.

```c
#include <stdio.h>
#include <string.h>
float sum(int,float);
void main()
{
int a=70,b=100.5;
clrscr();
sum(a,b);
printf("\nvalue of sum in main =%d",x);
getch();
}
float sum(int x,float y)
{
float sum;
sum=x+y;
printf("value of sum in function =%d",sum);
return sum;
}
```

Structure

Keyword struct is used for creating a structure.

Syntax of structure definition:

```
struct structure_name
{
data_type member1;
data_type member2;
data_type memeber;
};
```

Example:

```
struct person
{
char name[10];
int age;
float salary;
};
```

```
//Example program

#include<stdio.h>
struct student
{
char name[10];
int age;
float salary;
};
void main()
{
}
```

Example:

Structure definition without tag

```
struct
{
char name[10];
int age;
float salary;
};
```

```
#include<stdio.h>
struct
{
char name[10];
int age;
float salary;
};
void main()
{
}
```

Declaring Structure Variable::

A structure variable can be declared in 2 ways

1. Decalaring structure variable with structure definition:

```
struct
{
char name[10];
int age;
float salary;
}s1,s2,s3;
```

```
#include<stdio.h>
struct
{
char name[10];
int age;
float salary;
}s1,s2,s3;
void main()
{
}
```

2.Decalaring structure variable seperately:

```
struct students
{
char name[10];
int age;
float salary;
};
struct students s1,s2,s3;
```

2.Decalaring structure variable seperately:

```
struct
{
char name[10];
int age;
float salary;
}s1,s2,s3;
```

```
#include<stdio.h>
struct students
{
char name[10];
int age;
float salary;
}P;
void main()
{
struct students s1,s2,s3;
}
```

```
#include<conio.h>
struct students
{
char name[10];
int age;
float salary;
};
void main()
{
clrscr();
 struct students s1,s2,s3;
}
```

compilation error
Declaration is not allowed here

Structure Initialization

```
#include<stdio.h>
#include<conio.h>
struct students
{
char sex;
int age;
float salary;
}x={‘M’,24,33000};
void main()
{
}
```

```
#include<stdio.h>
#include<conio.h>
struct students
{
char sex;
int age;
float salary;
};
void main()
{
struct students x={‘m’,30,40};
}
```

```
#include<stdio.h>
#include<conio.h>
struct students
{
Char sex;
int age;
float salary;
};
void main()
{
struct students x={‘M’,30,4000};
struct students y={‘F’,50,6000};
struct students z={‘M’,100,20000};
}
```

```c
#include<stdio.h>
#include<conio.h>
struct students
{
char sex;
int age;
float salary;
}x={20,30,40},y={40,50,60},z={70,100,200};
void main()
{
}
```

Accessing structure members

1. Array elements are accessed using the Subscript variable , Similarly Structure members are accessed using dot [.] operator.
2. (.) is called as “Structure member Operator”.
3. Use this Operator in between **“Structure variable name”** & **“member name”**

```
#include<stdio.h>
#include<conio.h>
struct students
{
char sex;
int age;
float salary;
}x={‘m’,30,40};
void main()
{
clrscr();
printf("%c\n",x.sex);
printf("%d\n",x.age);
printf("%f\n",x.salary);
getch();
}
```

```
m
30
40
```

```
struct students
{
Int  sex;
int age;
float salary;
}x={20,30,40},y={40,50,6
0},z={70,100,200};
void main()
{
clrscr();
printf("%d\n",x.sex);
printf("%d\n",x.age);
printf("%f\n",x.salary);
printf("%d\n",y.sex);
printf("%d\n",y.age);
printf("%f\n",y.salary);
printf("%d\n",z.sex);
printf("%d\n",z.age);
printf("%f\n",z.salary);
getch();
}
```

Pointers

- Pointer is a memory address.
- Pointer is a variable that stores the address of another variable.

Pointer Operator in C Program :

Operator	Operator Name	Purpose
*	Value at Operator	Gives Value stored at Particular address
&	Address Operator	Gives Address of Variable

In order to create pointer to a variable we use “*” operator.

‘*’ is called as **‘Value at address**’ Operator

also called as **‘Indirection Operator’ or ‘dereference operator’**

To find the address of variable we use "&" operator.

'&' operator is called as address Operator

 'Value at address' Operator gives 'Value stored at Particular address.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n = 20;
clrscr();
printf("\nThe address of n is %u",&n);
printf("\nThe Value of n is %d",n);
printf("\nThe Value of n is %d",*(&n));
getch();
}
```

declaration of pointer

gf:

data-type *pointer-variable

int *ptr;

‘ptr’ is declared as that ‘ptr’ will be used only for storing the address of the integer valued variables

We can also say that ‘ptr’ points to integer

Value at the address contained in ‘ptr’ is integer

.

	Declaration	**Explanation**	**Memory Required**
1	int *p	p is going to store address of integer value	2 bytes
2	float *q	q is going to store address of floating value	4 bytes
3	char *ch	ch is going to store address of character variable	1 bytes

Initialize：

```
pointer = &variable;
```

```
void main()

{

int a=50,b=10,*s,*t;

s=&a;

t=&b;

//t=s;

clrscr();

printf("a=%d",*s);

printf("b=%d",*t);

getch();

}
```

A= 50

B=10

char a[12], *p = &a[0];											
*p	*(p+1)	*(p+2)	*(p+3)	*(p+4)	*(p+5)	*(p+6)	*(p+7)	*(p+8)	*(p+9)	*(p+10)	*(p+11)
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
H	e	l	l	o		W	o	r	l	d	'\0'