Q:- Why 'C' named so? What is the history of 'C' language? How it invented? Discuss the importance of 'C' over other languages?

Ans

History of C

C is a programming language developed at AT & T's Bell laboratories of U.S.A in 1972 by Dennis Ritchie.

In 1960 a no' of comp languages had come into existance almost each for a specific purpose. For eg:- COBOL was being used for commercial purposes, FORTRAN for engineering & scientific appln. and so on. At this stage an international committee was set up to develop such a language which can program all possible applications. This committe came out with a language called ALGOL60. ALGOL60 never really became popular because it seemed too abstract, too general. To reduce this abstraction and generality, a new language called CPL (Combined Programming language) was developed. However, CPL was so big, having so many features that it was hard to learn and difficult to implimet. then BCPL (Basic combined Programming language) developed to solve this problem but unfortunately it was also less powerful and too specific.

( ) → Paranthesis
{ } → Braces.

1,2

Around the same time a language called 'B' was written by Ken Thompson at AT & T's Bell labs. But like BCPL, 'B' too turned out to be very specific. After this Dennis Ritchie inherited the features of 'B' and BCPL added some of his own features and developed 'C'.

As C has all the features of the language B and developed after it as a neck version that's why it is named as C.

Any C Program is a collection of one or more functions. A function name is always followed by a pair of parenthesis, as in case of main, every program must have a special function named main. The program execution starts from this function. Each instruction in a function is written as a separate statement. These statements must appear in the same order in which we wish them to be executed. Usually all C statements are entered in small case letters. Any C statement always ends with a semi-column. The statements within a function are always enclosed within a pair of braces {}. The closing brace of the main function signals the end of the program.

Structure of a C Program:

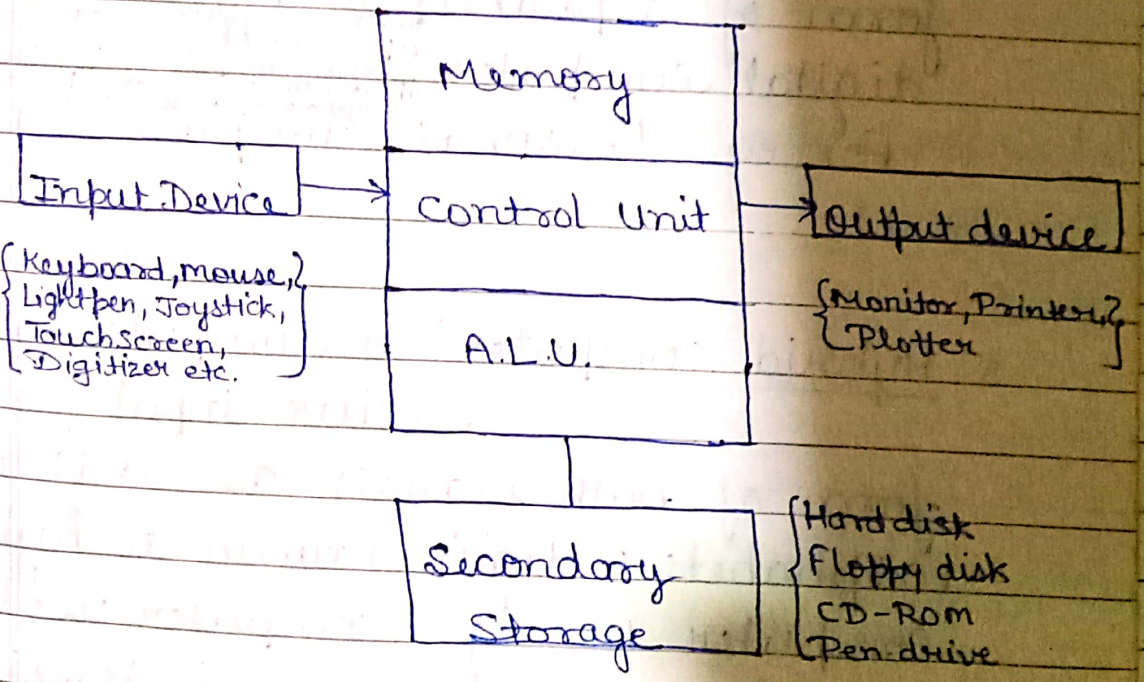
```
main ()
{ // Starting brace of main() function
Statement 1;
Statement 2;
} // end braces of main() function
function 1 ()
{
variable declarations;
Statement 1;
Statement 2;
}
```

Data → Anything like bio-data of applicant when computer is used for short listing candidate for recruiting, marks obtained by students in various subjects when used for preparing results, details (name, age, sex etc.) of passengers when used for making airline or railway reservations, or no. of different parameters when used for solving scientific and research problems etc.

8

* **Computer**:- Computer is an electronic device that accept data from the user through input device, do processing and finally gives the result through output device after processing.

Block Diagram of Digital computer:-



* **Classification of computers**:-

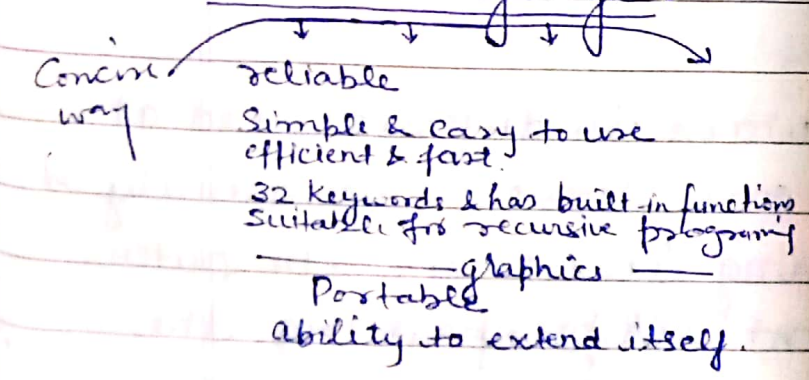
Computers can be classified into maidy three categories:-

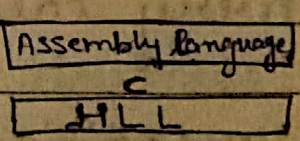
1 Analog computer.
2 Digital computer
3 Hybrid computer.

(Importance)

Characteristics of C :-

1. It is reliable, simple and easy to use.
2. Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators.
3. There are only 32 keywords and its strength lies in its built-in-functions.
4. C helps in developing structured programs.
5. C is highly suitable for recursive programming
6. It is also suitable for graphics programming.
7. C is highly portable.
8. We can add our own functions to the C library. Therefore, important features of C is its ability to extend itself.
9. C is format free language. No line numbers are needed. Need not place statements on a specified location on a line.
10. C stands in b/w the assembly language and high level language. C has all the advantages of assembly language and it ~~hatt~~ has all the ~~significant~~ significant features of modern high level language. That's why it is often called a middle level language.





* * Structured Programming :-

It is an orderly, disciplined approach to produce readable, maintainable and error free program in a quick manner. Thus, the main objective of structured programming technique is to develop programs that can be understood easy and are easy to maintain and debug.

E.W. Dikstra was the first person who introduce the idea of structured programming.

Objectives of structured (or developed) programming:

1. Programs can be devoted quickly.
2. Programs are easily readable.
3. A portion of the program can be modified easily without making changes in the whole program.
4. Programs can be debugged easily.

* * Character Set :- A character denotes any alphabet, digit or special symbol. used to represent infor". Following table shows the valid alphabets, numbers and special symbols allowed in C forms a character set

Alphabets :

uppercase	A, B, ---, Y, Z
lower case	a, b, ---, y, z
Digits	0, 1, ---, 9
Special symbol	+, -, _, ~, ', !, @, #, %, ^, &, *, (,), =, |, \, {, }, [,], ., ;, ", ?

* Tokens :- The smallest individual unit in a program are called as tokens. A token is a simple entity that makes-up a program. For eg :- If, for, sum, 0, &, A, <, ?, the first two are keywords; sum is a identifier; "0, &&A" are constants and "<, ?" are operators.

1. Keywords
2. Identifier
3. Constants
4. Operators

1. Keywords :- The words that have a standard and predefined meaning are known as keywords. It means they are already been defined in the C compiler. These can't be re-defined by the programmer, if we do so we are trying to assign new meaning to the keyword, which is not allowed by the computer. There are only 32 keywords available in C. For example: int, float, long, char, if, do, while, switch, goto, else, auto, double, case, break, for etc.

2. Identifiers :- Identifier refers to the names of variables, functions, arrays, class etc. created by the user.

Rules for using identifiers:

(i) An identifier may consist of letter, digits and underscore (_).
(ii) The first character must be letter.
(iii) Special symbol and blanks are not allowed.
(iv) Keywords or reserved words are not allowed.
(v) Both for uppercase & lowercase letters are allowed.

eg :- The variable SUM is not same as sum or Sum.

3. Constants :- A constant is an entity that doesn't change during the execution.

Types of constants:

(a) Numeric constants or Primary constants.
(b) Character constants or Secondary constants

Types of Primary constants:

(i) Integer constants
(ii) Real constants.

Types of Secondary constants:

(i) Array
(ii) Pointer
(iii) Structure
(iv) Union
(v) Enum etc.

An "integer constant" refers to sequence of digits without decimal point

Types of integers :-

(i) Decimal integer → consist of set of digits 0 to 9.
(ii) Octal integer → consist of combination of digits 0 to 7.
(iii) Hexadecimal integer :- consist of an combination of digits 0 to 9 and alphabets A to F.

"Real constants" A real constant refers to sequence of digits with decimal point. eg:- 40.2

4 Operators :- Operator is a symbol which represents a particular operation that can be peryormed on some data. The data itself is called operand. For example :- 4+2-3, here, the symbol (+) Plus and (-) minus are operators and the constants 4, 2 and 3 are operands. There are different types of operators available in C are:-

(i) Arithmetic operators :- The operators that are used for arithmetic operations such as addition, subtraction, multiplication & division are known as arithmetic operators.
eg:- +, -, *, % all these are arithmetic operators.
These operators are the binary operators because they operate on two operands at a time.

(ii) Unary operators / Increment and decrement operators :- These are unary operators, since, they operate only one operand.

The operand has to be a variable not a constant.

For eg:- 'i++' is valid. whereas 6++ is invalid.

(iii) Modulo-division operator:- It is also a binary arithmatic operator % called modulo division operator.

For eg:- 5 % 3 gives the result 2. while 3 % 5 gives the result 3.

☆ (iv) Relational operators:- These operators are used to compare two operands to see whether they are equal to eachother, unequal, whether one is greater than the other.

For eg:-

operator	Meaning
<	less than
>	greater than
<=	less than equal to
>=	greater than equal to
==	equal to
!=	not equal to

Each relational operator needs two operands for comparison of their values.

eg:- A < B

A == B + 3

(v) Logical operator :- Logical operators are used to determine or evaluate True or false.

For eg : &&, ||, ! standing for logical and, logical or and logical not respectively.

operands		Results	
Exp 1	Exp 2	Exp1 && Exp2	Exp1 || Exp2
0	0	0	0
0	Non-Zero	0	1
Non-Zero	0	0	1
Non-Zero	Non-Zero	1	1

(vi) Bitwise operators :- Bitwise operators are used to manipulate data at bit level. The various bitwise operators available in C are shown below:

Operators	Meaning
~	One's complement
>>	Right shift
<<	Left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

The output produced by these operators are given below:

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

(a) One's complement :- On taking 1's complement of a no. all 1's present in the no. are changed to zeros and zeros are changed to ones.

For eg :- 1's complement of 1010 is 0101

(b) Right shift operator :- It is represented by >> and it shift each bit in the operand to the right.

For eg :- If the variable 'a' contains the bit pattern 1 1 0 1 0 1 1 1 than 'a' >> 1 gives 0 1 1 0 1 0 1 1

Q :- x = 01010101 x >>3

00101010 Ist stage

00010101 IInd stage

00001010 IIIrd stage

(c) Left shift operator :- This is similar to the right shift operator. The only difference is that the bits are shifted to the left and zero is added to the right of number.

It is represented by <<

eg:- x = 01010101 x<<3

x<<1 = 10101010

x<<2 = 01010100

x<<3 = 10101000

(vii) Conditional operators:- The conditional operator (?) and colon (:) are sometimes called ternary operator.

Since they take three operands. Their general form is:

Exp1 ? Exp2 : Exp3

If Exp1 is true then the value returned will be Exp2 otherwise the value returned will be Exp3.

For eg: int x, y;

scanf ("%d", &x);

y = (x > 5 ? 3 : 4);

This statement will store 3 in y, if x > 5, otherwise it will store 4 in y.

(viii) Assignment operators:- This operator is used to assign the value of a constant or variable.

eg:- a = 10;

In this, we assign the value of a by 10

eg :- b = 10 ;
a = b ;

Here, we assign the value of a by b. where value of b is 10.

There are some compound assignment statements are used in programs as:

+= , -= , *= , /= and %=

+= is a compound assignment statement

Program using compound assignment stat.:

```
#include<stdio.h>
void main()
{
  int i=1;
  while(i<=10)
  {
    Printf("%d\n", i);
    i+=1;
  }
}
```

Here, += increment the value of 'i' by 1.

Q:- What is Precedence or hierarchy?

Ans Hiriarchy or Precedence is the order or Priority in which the operations are performed in an operation.

The hierarchy and Associativity of all operators available in C can be shown below:

Precedence or Hierarchy	operators	Associativity
I st	( )	Left to right
II nd	++, --, !, −	Right to left
III rd	* / % (Arithmetic operators)	Left to right
IV th	+ − (Arithmetic operators)	"
V th	<< >> (Bitwise operators)	"
VI th	< <= > >= (Relational operators)	"
VII th	== != (Relational operators)	"
VIII th	& (bitwise AND)	"
IX th	^ (bitwise XOR)	"
X th	| (bitwise OR)	"
XI th	&& (logical AND)	"
XII th	|| (logical OR)	"
XIII th	? : (Conditional operators)	Right to left
XIV th	= (All assign operators)	"
XV th	, (comma)	"

: Table a

'Associativity' means expression is solved either from left to right or from right to left. The associativity of all the operators is given below:

Same Table a

** Constants and Variables :-

A 'constant' is a quantity that does not change. This quantity can be stored at a location in the memory of the computer. A 'variable' can be considered as a name given to the location in memory where this constant is stored. The containts of the variable can change. For eg: $3x+y=20$.

Here, 3 and 20 are constants and x & y are the variables.

Constants can be integer, real, logical and string constants.

For eg:- integer constants are 426, 782, -8000 (Range -32768 to 32767)

Examples of Real constants are: 325.34, -32.76 etc.

Examples of character constants: 'A' is valid character constant whereas 'A' is not

Logical Constants: 'T'/'F'

String constants: A collection of characters enclosed within a pair of double inverted commas is treated as string constant.
"Hello".

* Scope of variable:

Scope of variable means where the variable is available within the program. Variable can have two types of scope: Local and Global.

A variable with a global scope is accessible to all the statements in the program, whereas the variable with local scope is available only to certain selected statements in the program.

* Local and Global variables :-

Variables which are declared inside the body of function or main program are known as local variables. Local variables can only access by a function which it is declared after the end of function. The local variable is automatically destroyed and it cannot be used. Hence, scope of local variable are within function in which it is defined.

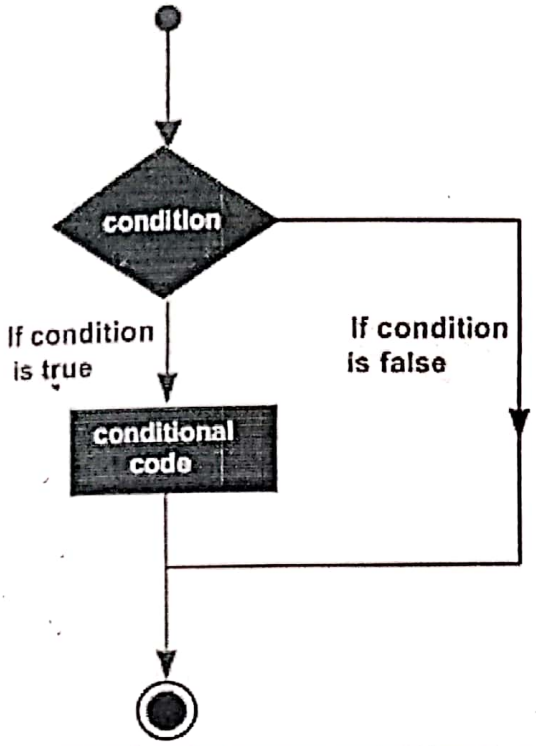
* Diff. b/w Local & Global variables?

Local Variables	Global Variables
1. Variables are local to the block in which it is defined	Global variables can be used anywhere in the program.
2. Life of local variable is upto the life of block in which it is defined	Life of global variable ~~is upto the~~ ends with the termination of program.

C - Decision Making

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision making structure found in most of the programming languages –



C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides the following types of decision making statements.

Sr.No.	Statement & Description
1	**if statement** An **if statement** consists of a boolean expression followed by one or more statements.
2	**if...else statement** An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
3	**nested if statements** You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	**switch statement** A **switch** statement allows a variable to be tested for equality against a list of values.
5	**nested switch statements** You can use one **switch** statement inside another **switch** statement(s).

The ? : Operator

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form –

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this –

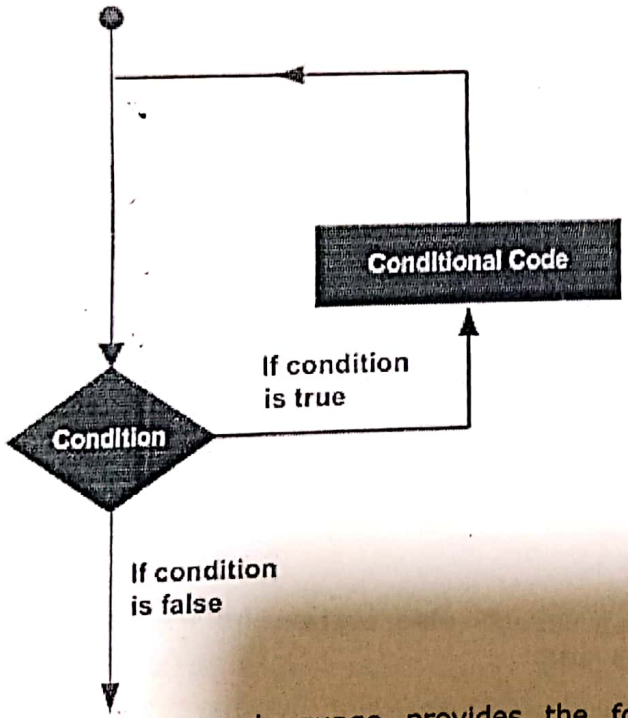
- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

C – Loops

ou may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

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 –



C programming language provides the following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	**while loop** - Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop** - Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** - It is more like a while statement, except that it tests the condition at the end of the loop body.

4	**nested loops** - You can use one or more loops inside any other while, for, or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Sr.No.	Control Statement & Description
1	**break statement** - Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	**continue statement** - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**goto statement** - Transfers control to the labeled statement.

NOTE – You can terminate an infinite loop by pressing Ctrl + C keys.

C - Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example –

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

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

```
value of g = 10
```

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. The loop enables us to perform n number of steps together in one line.

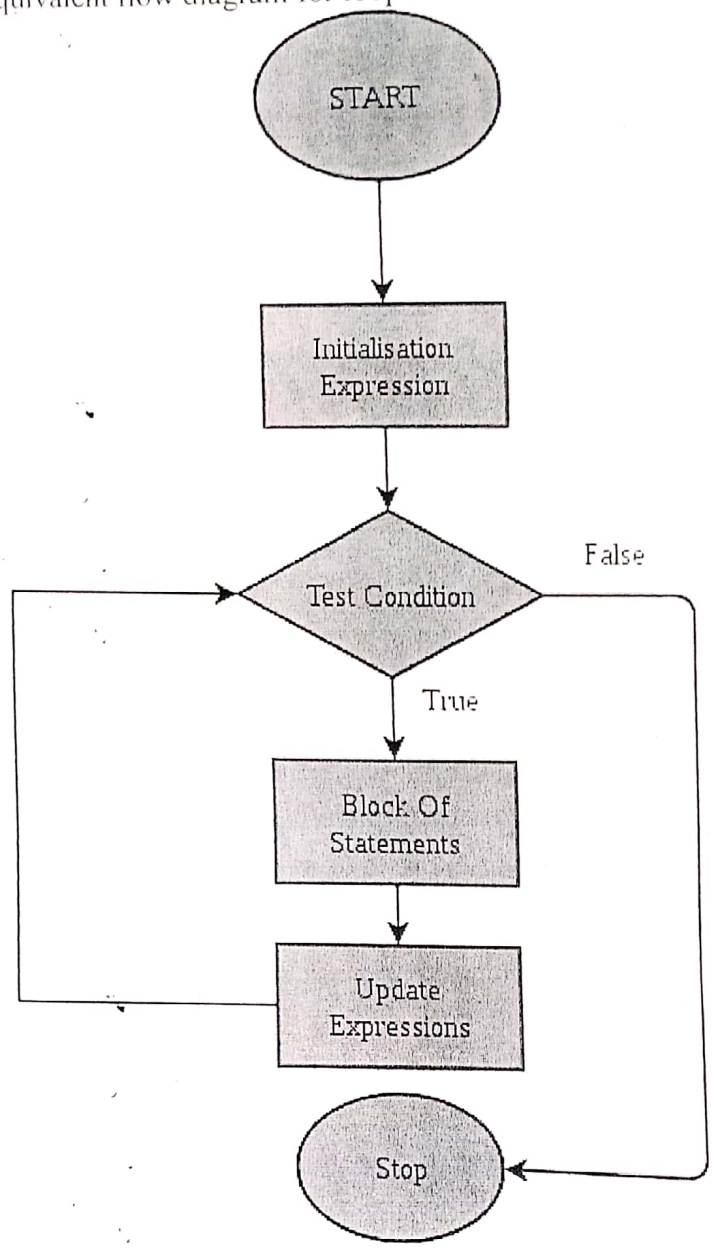
Syntax:

```
for (initialization expr; test expr; update expr)

{

    // body of the loop

    // statements we want to execute

}
```

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.

- **Initialization Expression**: In this expression we have to initialize the loop counter to some value. for example: int 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++;

Equivalent flow diagram for loop :



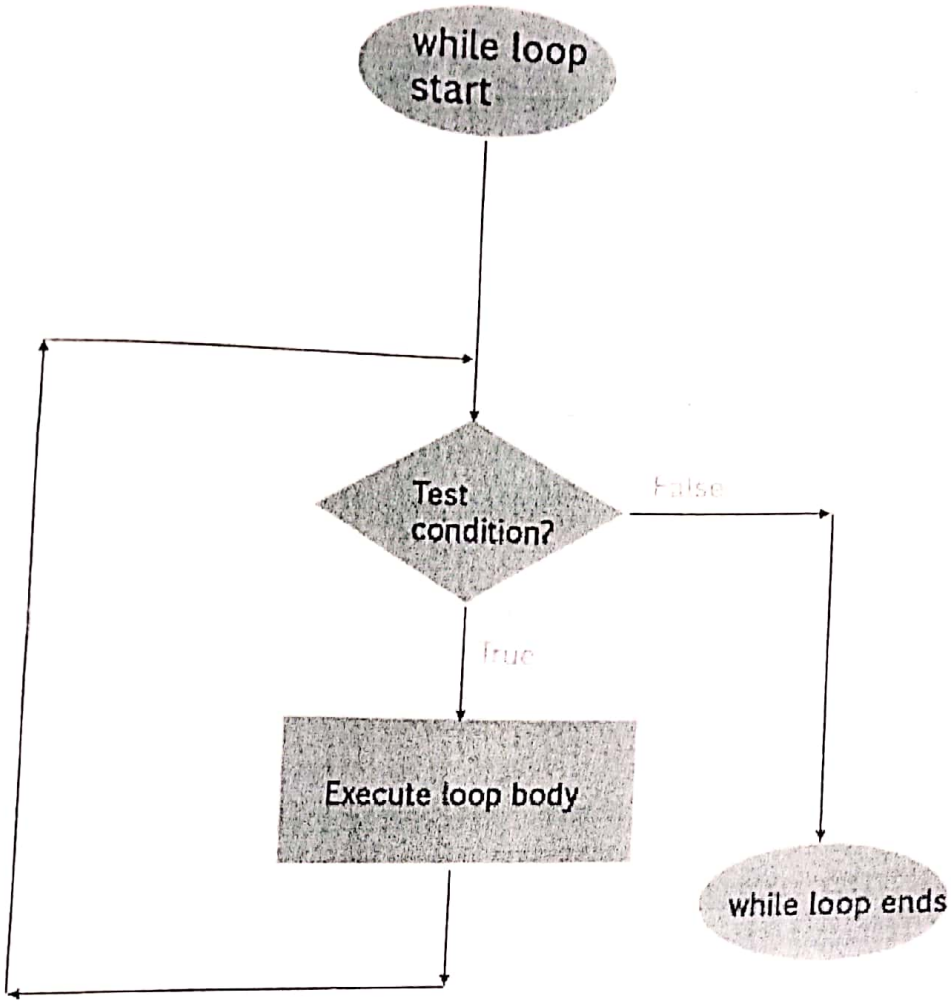
/ C program to illustrate for loop

```
#include <stdio.h>

int main()
{
   int i=0;

   for (i = 1; i <= 10; i++)
   {
      printf( "Hello World\n");
   }

   return 0;
}
```

Output:

```
Hello World

Hello World
```

Flow Diagram:




```
// C program to illustrate for loop
#include <stdio.h>
int main()
{
   // initialization expression
   int i = 1;
   // test expression
   while (i < 6)
   {
      printf( "Hello World\n");
      // update expression
      i++;
  }
   return 0; }
```

```
Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World
```

While Loop

While studying **for loop** we have seen that 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:

We have already stated that a loop is mainly consisted of three statements – initialization expression, test expression, update expression. The syntax of the three loops – For, while and do while mainly differs on the placement of these three statements.

```
initialization expression;

while (test_expression)

{

   // statements


  update_expression;

}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

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.

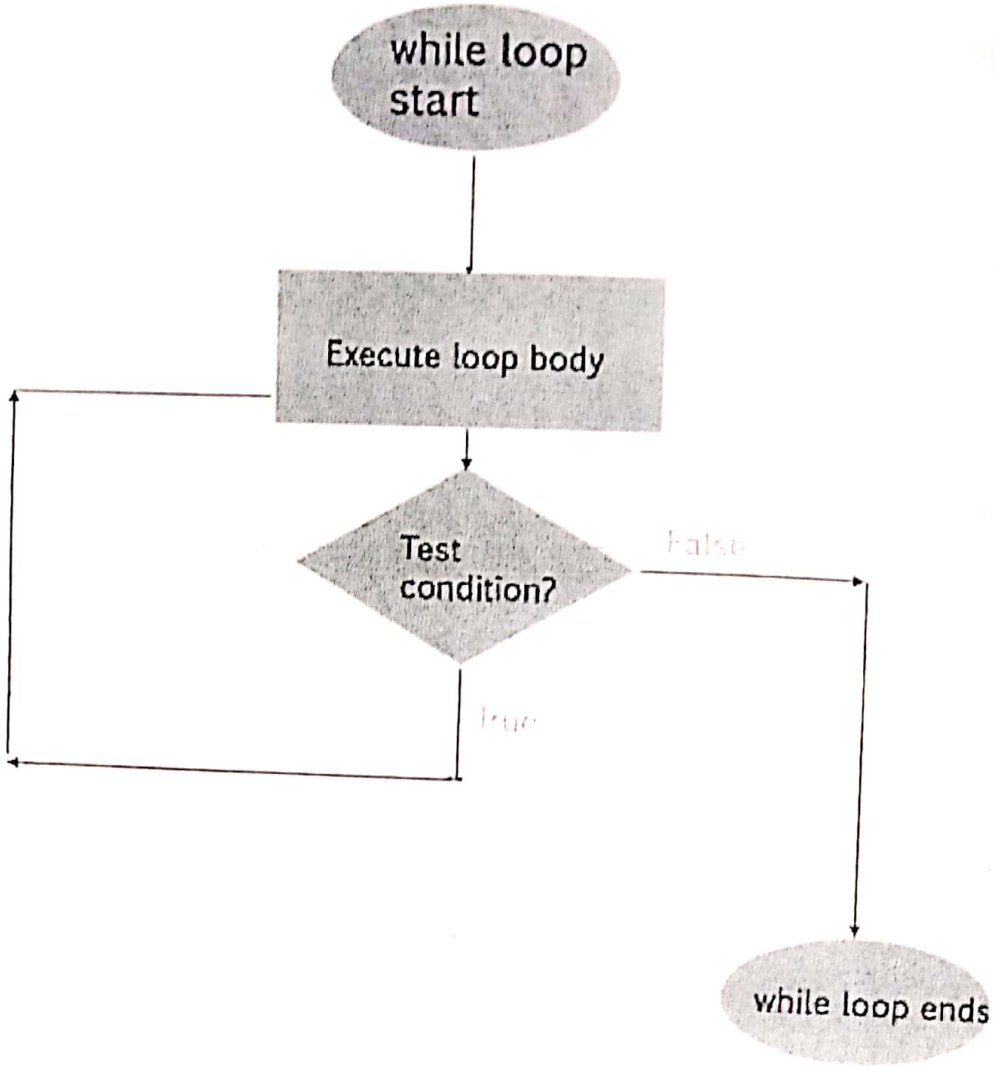
Note: In do while loop the loop body will execute at least once irrespective of test condition.

Syntax:

```
initialization expression;
do
{
   // statements

   update_expression;
} while (test_expression);
```

Note: Notice the semi – colon(";") in the end of loop.

Flow Diagram:




```
// C program to illustrate do-while loop
#include <stdio.h>
int main()
{
    int i = 2; // Initialization expression
do
 {
        printf( "Hello World\n"); // loop body
        i++; // update expression
 }  while (i < 1);   // test expression
 return 0;
}
```

Output:

```
Hello World
```

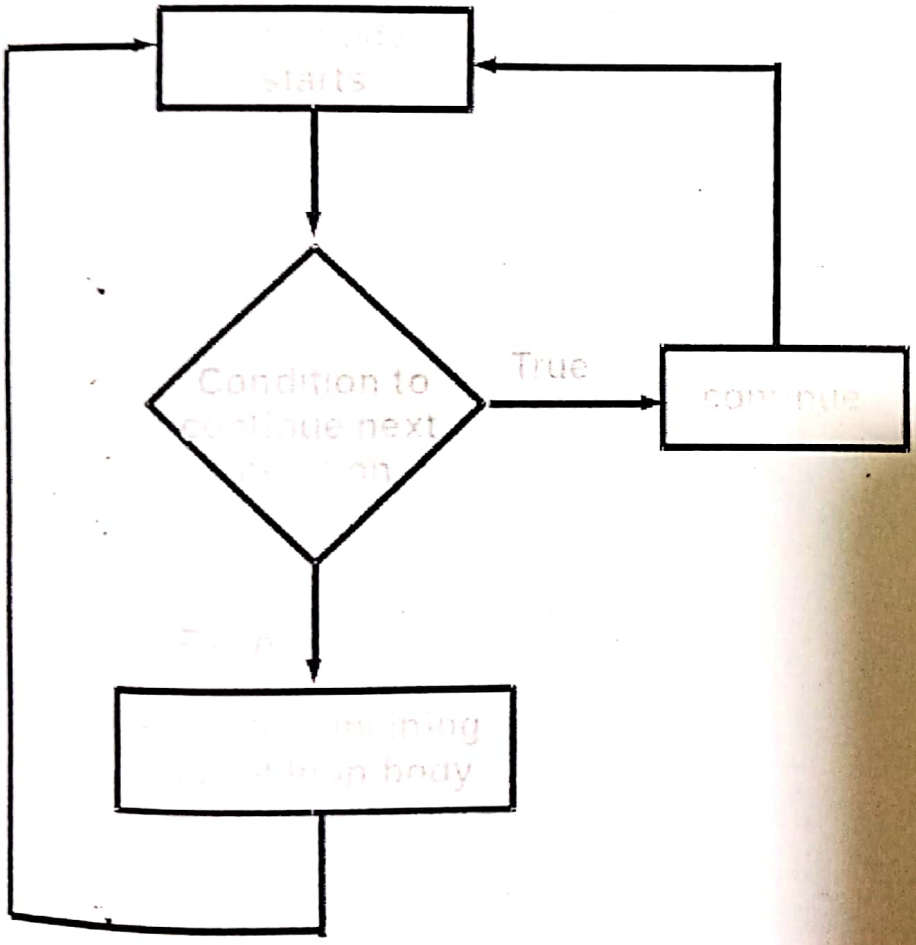
Continue Statement in C/C++

Continue is also a loop control statement just like the break statement. *continue* statement is opposite to that of break *statement*, instead of terminating the loop, it 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;
```



Example:

Consider the situation when you need to write a program which prints number from 1 to 10 and but not 6. It is specified that you have to do this using loop and only one loop is allowed to use. Here comes the usage of continue statement. What we can do here is we can run a loop from 1 to 10 and every time we have to compare the value of iterator with 6. If it is equals to 6 we will use the *continue* statement to continue to next iteration without printing anything otherwise we will print the value.

Below is the implementation of above idea:

```
// C++ program to explain the use
// of continue statement

#include <iostream>
using namespace std;

int main()
{
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            cout << i << " ";
    }

    return 0;
}
```

Output:

```
1 2 3 4 5 7 8 9 10
```

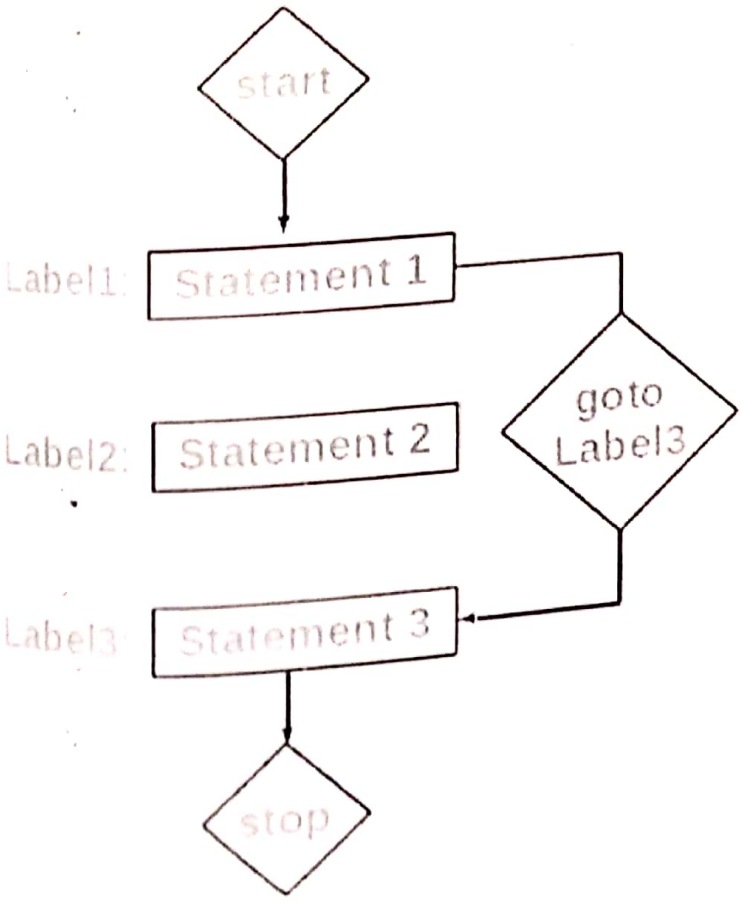
goto statement in C/C++

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.

Syntax:

```
Syntax1          |    Syntax2
-------------------------------
goto label;      |      label:
.                |      .
.                |      .
.                |      .
label:           |      goto label;
```

In the above syntax the first line tells the compiler to go to or jump to the statement marked as 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 above syntax.



```
// C program to print numbers
// from 1 to 10 using goto statement
#include <iostream>
using namespace std;

// function to print numbers from 1 to 10
int main()
{
    int n = 1;
label:
    cout << n << " ";
    n++;
    if (n <= 10)
        goto label;
}
```

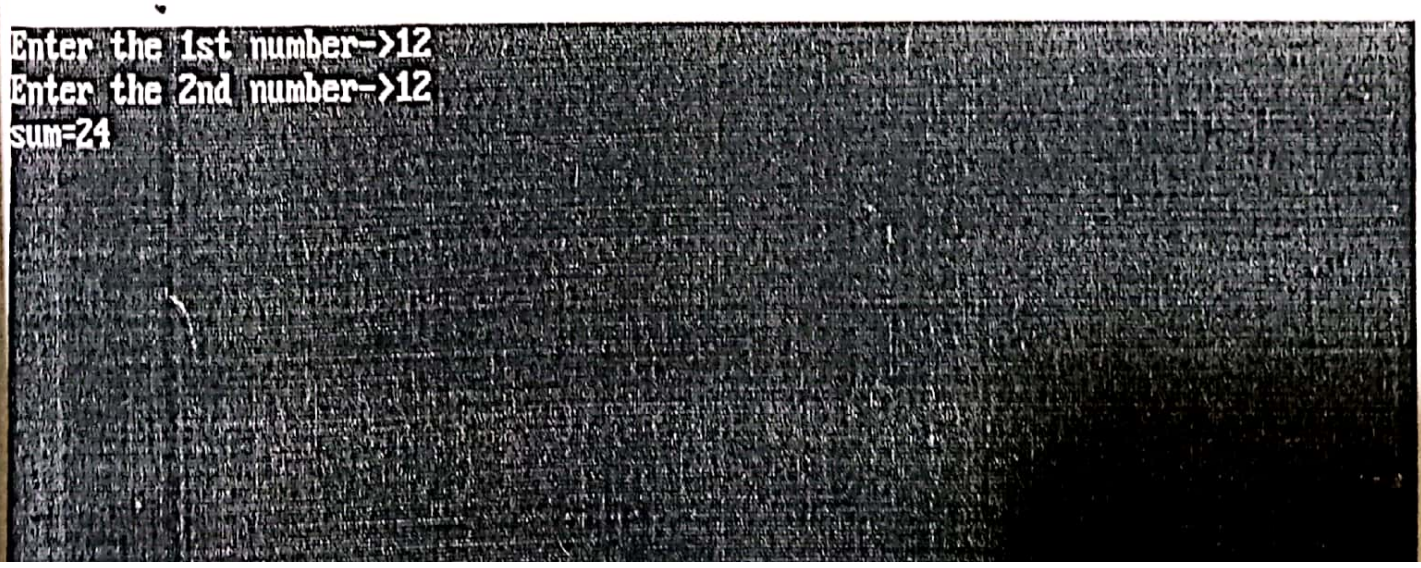
Output:

- 1 2 3 4 5 6 7 8 9 10

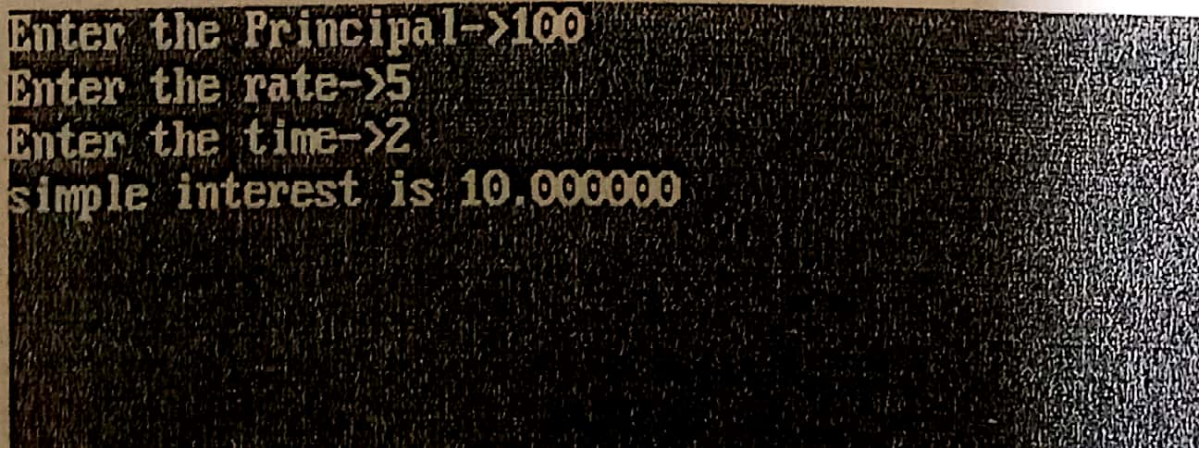
Disadvantages of using goto statement:

- The use of goto statement is highly discouraged as it makes the program logic very complex.
- use of goto makes really hard to modify the program.
- Use of goto can be simply avoided using break and continue statements.

```
/*Write a program to calculate sum of two numbers*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,sum;
clrscr();
printf("Enter the 1st number->");
scanf("%d",&a);
printf("Enter the 2nd number->");
scanf("%d",&b);
sum=a+b;
printf("%d",sum);
getch();
}
```



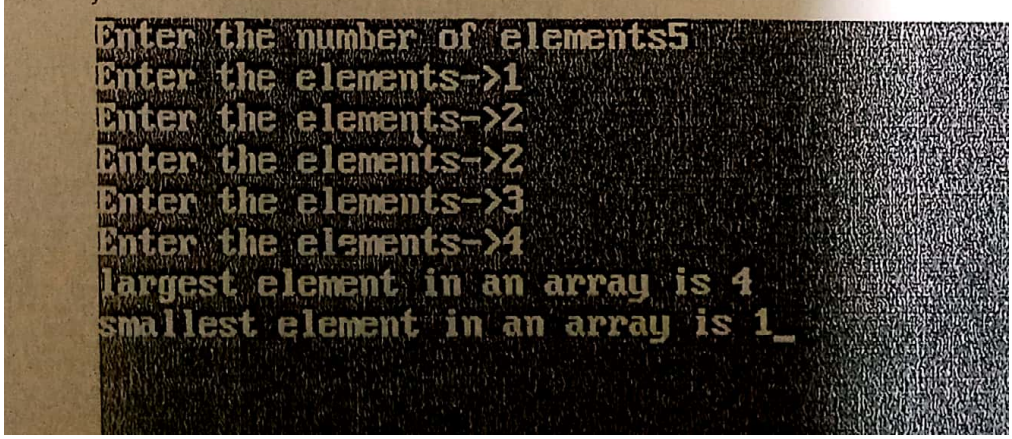
```
/*WAP to calculate Simple Interest*/
#include<stdio.h>
#include<conio.h>
void main()
{
int t;
float p,r,SI;
clrscr();
printf("Enter the Principal->");
scanf("%f",&p);
printf("Enter the rate->");
scanf("%f",&r);
printf("Enter the time->");
scanf("%d",&t);
SI=(p*r*t)/100;
printf("simple interest is %f",SI);
getch();
}
```



```
/*WAP to print larger among two numbers*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
clrscr();
printf("Enter the first number->");
scanf("%d",&a);
printf("Enter the second number->");
scanf("%d",&b);
if(a>b)
{
printf("First number is greater");
}
else if(b>a)
{
printf("Second number is greater");
}
else
{
printf("Both numbers are equal");
}
getch();
}
```

```
Enter the first number->23
Enter the second number->43
Second number is greater
```

```
//WAP to print largest and smallest in an array
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],max,min,i,size;
clrscr();
printf("Enter the size of array");
scanf("%d",&size);
for(i=0;i<size;i++)
{
scanf("%d",&a[i]);
}
max=a[0];
for(i=1;i<size;i++)
{
if(max<a[i])
max=a[i];
}
printf("Max element is %d",max);
min=a[0];
for(i=0;i<size;i++)
{
if(min>a[i])
min=a[i];
}
printf("\nMin element is %d",min);
getch();
}
```



```
//wap to find a^b
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,num=1,temp;
clrscr();
printf("enter the value of a and b");
scanf("%d%d",&a,&b);
temp=a;
while(b!=0)
{
num=a*num;
b--;
}
printf("result=%d",num);
getch();
}
```

```
//WAP to check if number is prime or not
#include<stdio.h>
#include<conio.h>
void main()
{
int num,flag,i;
clrscr();
printf("Enter the number->");
scanf("%d",&num);
for(i=2;i<num;i++)
{
if(num%i==0)
{
flag=1;
}
}
if(flag==1)
{
printf("Number is not prime");
}
else
{
printf("Number is prime");
}
getch();
}
```

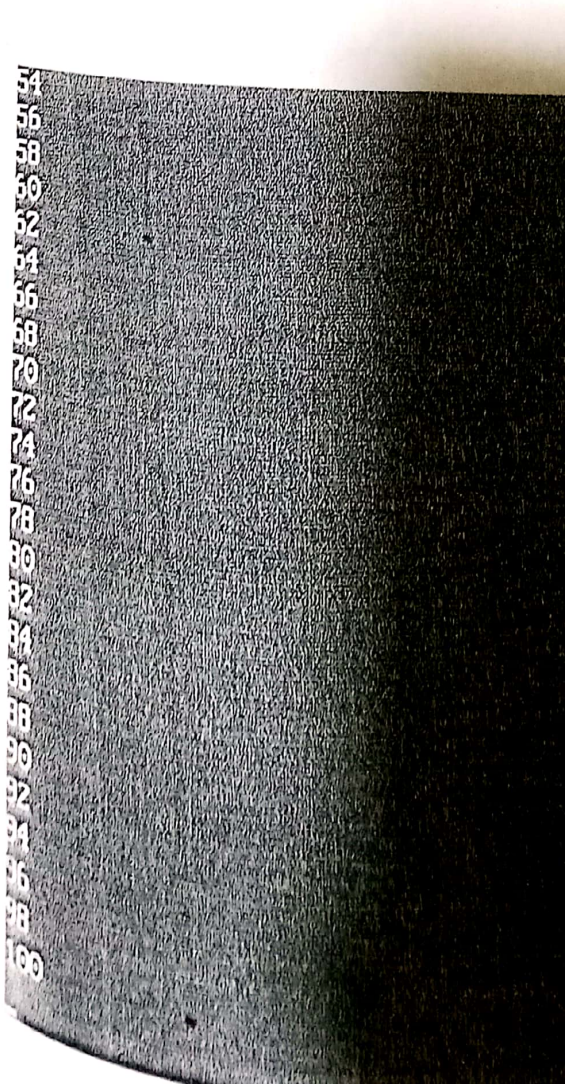
```
Enter the number->5
Number is prime_
```

```
//wap to print reverse of a number entered by user
#include<stdio.h>
#include<conio.h>
void main()
{
int rem,num,revno=0;
clrscr();
printf("Enter the number");
scanf("%d",&num);
while(num!=0)
{
rem=num%10;
revno=revno*10+rem;
num=num/10;
}
printf("reverse no is %d",revno);
getch();
}
```

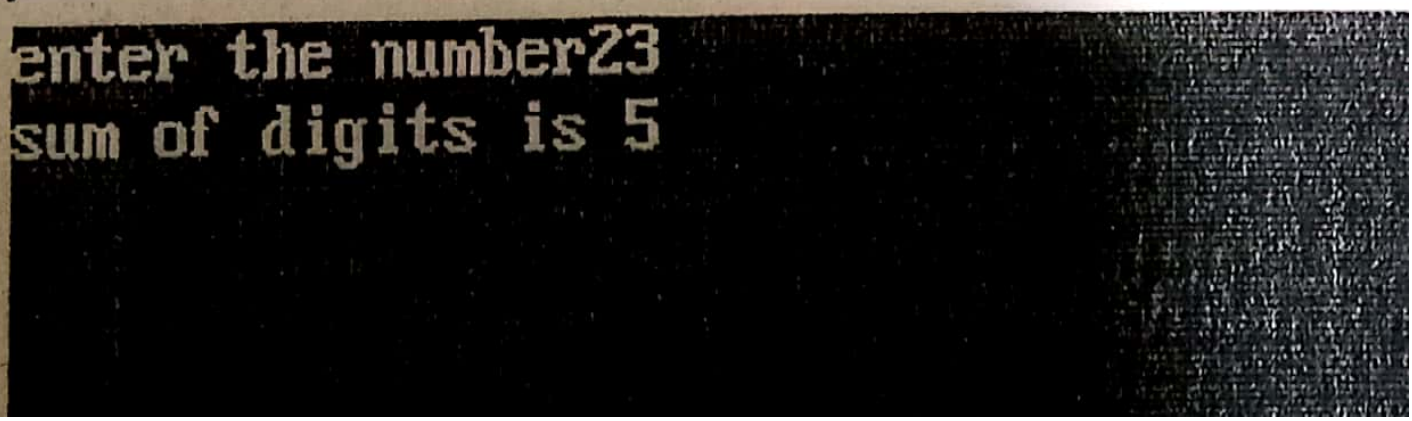
```
//wap to calculate factorial of a number
#include<stdio.h>
#include<conio.h>
void main()
{
int fact=1,num;
clrscr();
printf("enter the num->");
scanf("%d",&num);
while(num>0)
{
fact=num*fact;
num--;
}
printf("Factorial of a num is %d",fact);
getch();
}
```

```
enter the num->6
Factorial of a num is 720
```

```
//WAP to print even numbers from 1 to 100
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
clrscr();
for(i=1;i<=100;i++)
{
if(i%2==0)
{
printf("%d\n",i);
}
}
getch();
}
```



```
//wap to print sum of digits of a number
#include<stdio.h>
#include<conio.h>
void main()
{
int sum=0,num,r;
clrscr();
printf("enter the number");
scanf("%d",&num);
while(num>0)
{
r=num%10;
sum=sum+r;
num=num/10;
}
printf("sum of digits is %d",sum);
getch();
}
```



```
//WAP to print 1 to 10 using loop

#include<stdio.h>

#include<conio.h>

void main()

{

int i;

clrscr();

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

{

printf("%d\n",i);

}

getch();

}
```

Switch Statement in C

Switch case statements are a substitute for long if statements that compare a variable to several integral values

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Switch is a control statement that allows a value to change control of execution.

Syntax:

```
switch (n)
{
   case 1: // code to be executed if n = 1;
      break;
   case 2: // code to be executed if n = 2;
      break;
   default: // code to be executed if n doesn't match any cases
}
```

1. The expression provided in the switch should result in a **constant value** otherwise it would not be valid.

   Valid expressions for switch:

2. // Constant expressions allowed
3. switch(1+2+23)

   switch(1*2+3%4)

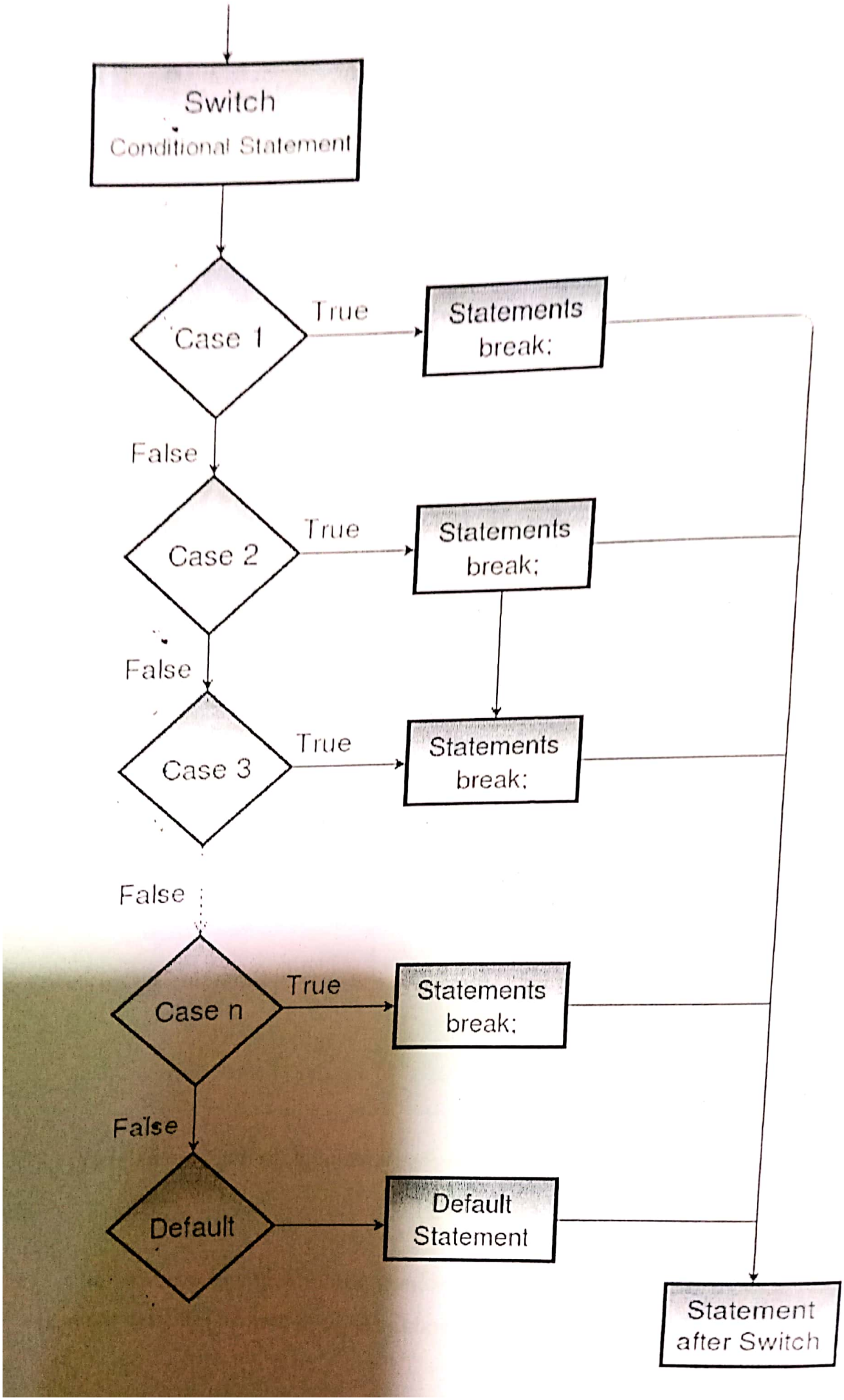
   Invalid switch expressions for switch:

   ```
   // Variable expression not allowed
   switch(ab+cd)
   switch(a+b+c)
   ```

4. Duplicate case values are not allowed.
5. The default statement is optional.Even if the switch case statement do not have a default statement,

   it would run without any problem.
6. The break statement is used inside the switch to terminate a statement sequence. When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
7. The break statement is optional. If omitted, execution will continue on into the next case. The flow of control will fall through to subsequent cases until a break is reached.
8. Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.

Flowchart



```c
/**
 * C program to print day of week using switch case
 */

#include <stdio.h>

int main()
{
    int week;

    /* Input week number from user */
    printf("Enter week number(1-7): ");
    scanf("%d", &week);

    switch(week)
    {
        case 1:
            printf("Monday");
            break;
        case 2:
            printf("Tuesday");
            break;
        case 3:
            printf("Wednesday");
            break;
        case 4:
            printf("Thursday");
            break;
        case 5:
            printf("Friday");
            break;
        case 6:
            printf("Saturday");
            break;
        case 7:
            printf("Sunday");
            break;
        default:
            printf("Invalid input! Please enter week number between 1-7.");
    }

    return 0;
}
```

Output

```
Enter the week no(1-7):1
Monday
```

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined –

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

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

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

```
#include <stdio.h>

int main () {
```

```
int  var = 20;   /* actual variable declaration */
int  *ip;        /* pointer variable declaration */

ip = &var;  /* store address of var in pointer variable*/

printf("Address of var variable: %x\n", &var  );

/* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip );

/* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );

return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

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

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

```
/**
 * C program to swap two numbers using call by value
 */

#include <stdio.h>
void swap(int num1, int num2);
int main()
{
    int n1, n2;

    /* Input two integers from user */
    printf("Enter two numbers: ");
    scanf("%d%d", &n1, &n2);

    /* Print value of n1 and n2 in before swapping */
    printf("In Main values before swapping: %d %d\n\n", n1, n2);

    /* Function call to swap n1 and n2 */
    swap(n1, n2);
    printf("In Main values after swapping: %d %d", n1, n2);

    return 0;
}
void swap(int num1, int num2)
{
    int temp;

    printf("In Function values before swapping: %d %d\n", num1, num2);

    temp = num1;
    num1 = num2;
    num2 = temp;

    printf("In Function values after swapping: %d %d\n\n", num1, num2);
}
```

Output -

```
Enter two numbers: 10 20
In Main values before swapping: 10 20
In Function values before swapping: 10 20
In Function values after swapping: 20 10
In Main values after swapping: 10 20
```

```
* C program to swap two numbers using call by reference
*/

#include <stdio.h>
void swap(int * num1, int * num2);
int main()
{
    int n1, n2;

    printf("Enter two numbers: ");
    scanf("%d%d", &n1, &n2);

    printf("In Main values before swapping: %d %d\n\n", n1, n2);

    /*
     * &n1 - & evaluate memory address of n1
     * &n2 - & evaluate memory address of n2
     */
    swap(&n1, &n2);

    printf("In Main values after swapping: %d %d", n1, n2);

    return 0;
}
void swap(int * num1, int * num2)
{
    int temp;

    printf("In Function values before swapping: %d %d\n", *num1, *num2);

    temp  = *num1;
    *num1 = *num2;
    *num2 = temp;

    printf("In Function values after swapping: %d %d\n\n", *num1, *num2);
}
```

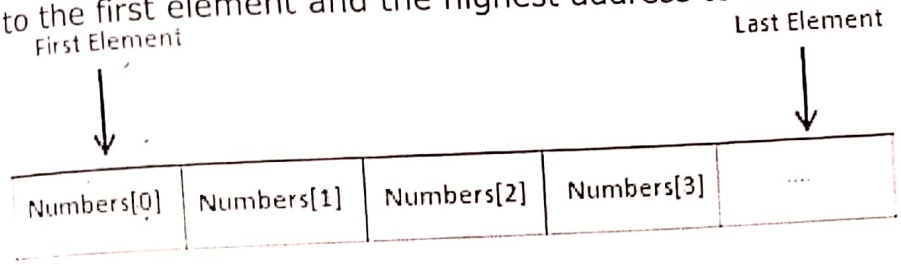
Output -

```
Enter two numbers: 10 20
In Main values before swapping: 10 20
In Function values before swapping: 10 20
In Function values after swapping: 20 10
In Main values after swapping: 20 10
```

C – Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement –

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

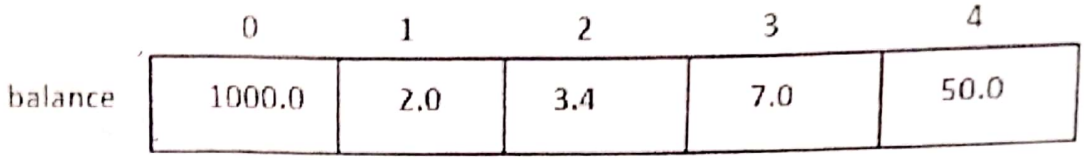
```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

Following is an example to assign a single element of the array –

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. A arrays have 0 as the index of their first element which is also called the base inde and the last index of an array will be total size of the array minus 1. Shown belo is the pictorial representation of the array we discussed above –



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays

```
#include <stdio.h>

int main () {

 int n[ 10 ]; /* n is an array of 10 integers */

  int i,j;

  /* initialize elements of array n to 0 */

  for ( i = 0; i < 10; i++ ) {

     n[ i ] = i + 100; /* set element at location i to i + 100 */

  }

      for (j = 0; j < 10; j++ ) /* output each array element's value */
      printf("Element[%d] = %d\n", j, n[j] );

return 0;}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C - Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program –

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
   int mount;
   auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
   register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared. In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

 /* function declaration */

void func(void);

static int count = 5; /* global variable */

main() {

   while(count--) {

      func();

   }

   return 0;

}

/* function definition */

void func( void ) {

   static int i = 5; /* local static variable */

   i++;

   printf("i is %d and count is %d\n", i, count);

}
```

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

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to

provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;

extern void write_extern();

main() {

   count = 5;

   write_extern();

}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {

   printf("count is %d\n", count);

}
```

Here, *extern* is being used to declare *count* in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows –

```
$gcc main.c support.c
```

It will produce the executable program **a.out**. When this program is executed, it produces the following result –

```
count is 5
```