INTRODUCTION TO C

History of C Language

- The base or father of programming languages is 'ALGOL.' It was first introduced in 1960.
- 'ALGOL' was used on a large basis in European countries. 'ALGOL' introduced the concept of structured programming to the developer community.
- In 1967, a new computer programming language was announced called as 'BCPL' which stands for Basic Combined Programming Language. BCPL was designed and developed by Martin Richards, especially for writing system software. This was the era of programming languages.
- in 1970 a new programming language called 'B' was introduced by Ken Thompson that contained multiple features of 'BCPL.' This programming language was created using UNIX operating system at AT&T and Bell Laboratories. Both the 'BCPL' and 'B' were system programming languages.

- In 1972, a great computer scientist Dennis Ritchie created a new programming language called 'C' at the Bell Laboratories. It was created from 'ALGOL', 'BCPL' and 'B' programming languages. 'C' programming language contains all the features of these languages and many more additional concepts that make it unique from other languages.

Elements of C Programming Language

As every language has some basic geometrical rules and elements, similarly C language has some elements and rules for building a program which has some meaning.

Character Set: In Real world to communicate with people we use language like Hindi English Urdu extra which is constructed and Defined by some characters, words extra. Similarly in C programming language we have various characters to communicate with the computer in order to produce a meaningful program and can produce an output.

Character Set In C Language

Type	Set
Lowercase	a-z
Uppercase	A-Z
Digits	0-9
special characters	!,@,#,$,%
White space	space, tab, and new lines

Keywords:

They are 32 Keywords in C language whose meaning has already being defined.

32 keywords in C language

Auto	do	goto	signed	unsigned	break
void	else	int	case	static	double
sizeof	enum	long	struct	char	if
while	const	extern	register	continue	volatile
defualt	for	typeodef	float	short	return
union	const				

Data types:

Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. Let us briefly describe them one by one:
Following are the examples of some very common data types used in C:

- **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

- **int:** As the name suggests, an int variable is used to store an integer.
- **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Different data types also have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler.

Data types	Storage size	Value range
char	1 byte	a-z, A-z, special characters (-128 to 127)
Unsigned char	1 byte	a-z, A-z, special character(0 to 255)
int	2-4 bytes	-32768 to 32767 or -214748348 to 2147483647
unsigned int	2-4 bytes	0 to 65535 or 0 to 4294967295
short	2 bytes	-32768 to 32767
unsigned short	2 bytes	0 to 65535
long	4 bytes	-2147483648 to 2147483647
unsigned long	4 bytes	0 to 4294967295
float	4 bytes	1.2E-38 to 3.4E+38 6 decimal place
double	8 bytes	2.3E-308 to 1.7E+308 15 decimal place
long double	10 bytes	3.4E-4932 to 1.1E+4932 19 decimal place

VARIABLES IN C

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

The example of declaring the variable is given below:

```
int a;
float b;
char c;
```

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

```
int a=10,b=20;//declaring 2 variable of integer type
float f=20.8;
char c='A';
```

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Valid variable names:

```
int a;
int _ab;
```

```
int a30;
```

Invalid variable names:

```
int 2;
int a b;
int long;
```

Types of Variables in C

There are many types of variables in c:

1. local variable
2. global variable

Local Variable

A variable that is declared inside the function or block is called a local variable.

It must be declared at the start of the block.

```
void function1()
{
int x=10;//local variable
}
```

You must have to initialize the local variable before it is used.

Global Variable

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.

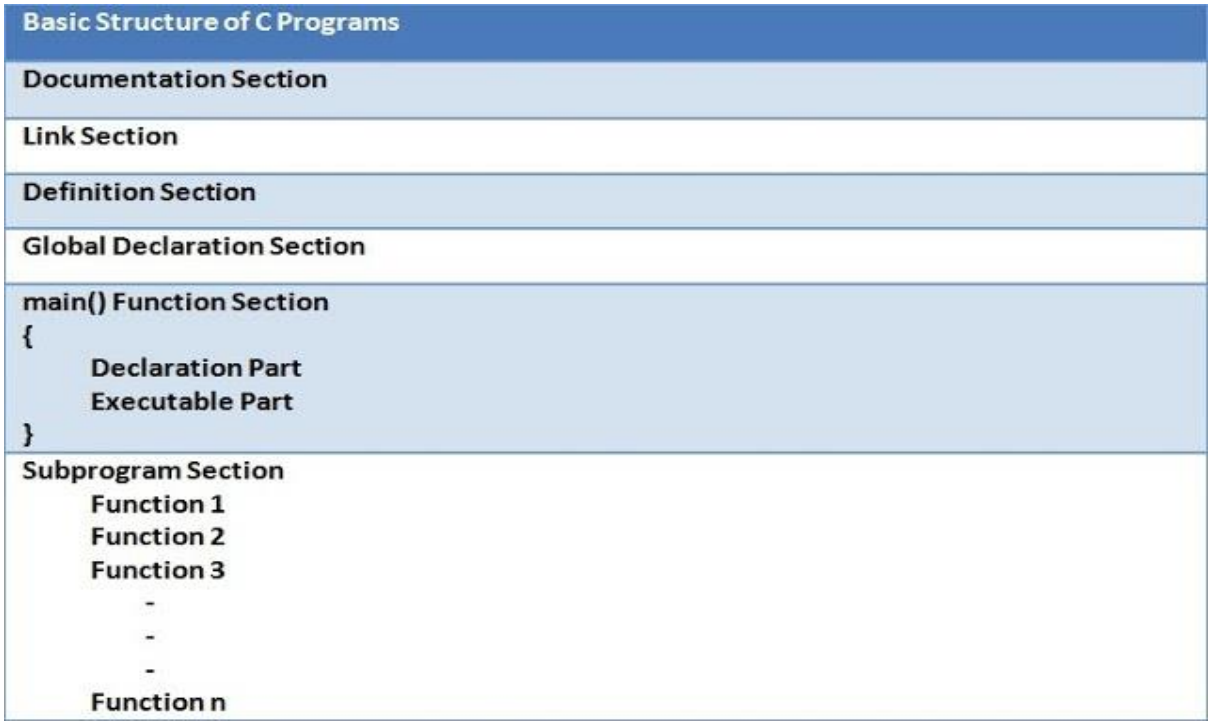
It must be declared at the start of the block.

```
int value=20;//global variable
void function1()
```

```
{
int x=10;//local variable
}
```

BASIC STRUCTURE OF C PROGRAM

Any C program is consists of 6 main sections. Below you will find brief explanation of each of them.



Basic Structure of C Program

Documentation Section

This section consists of comment lines which include the name of programmer, the author and other details like time and date of writing the program. Documentation section helps anyone to get an overview of the program.

The documentation section contains a set of comment including the name of the program other necessary details. Comments are ignored by compiler and are used to provide documentation to people who reads that code. Comments are be giving in C programming in two different ways:

1. Single Line Comment
2. Multi Line Comment

```
// This is single line comment

/*This is

multi line

comment

*/
```

Link Section

The link section consists of the header files of the functions that are used in the program. It provides instructions to the compiler to link functions from the system library.

The link section consists of header files while contains function prototype of Standard Library functions which can be used in program. Header file also consists of macro declaration. Example:

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

In the above code, stdio.h is a header file which is included using the preprocessing directive #include. Learn more about header files in C programming.

Definition Section

All the symbolic constants are written in definition section. Macros are known as symbolic constants.

The definition section defines all symbolic constants. A symbolic constant is a constant value given to a name which can't be changed in program. Example:

```
#define PI 3.14
```

In above code, the PI is a constant whole value is 3.14

Global Declaration Section

The global variables that can be used anywhere in the program are declared in global declaration section. This section also declares the user defined functions.

```
int a=10;
```

main() Function Section

The main () function section is the most important section of any C program.
The compiler start executing C program from **main()** function.
The **main()** function is mandatory in C programming. It has two parts:
Declaration Part – All the variables that are later used in the executable part are declared in this part.

Executable Part – This part contains the statements that are to be executed by the compiler

```
main()
  {
      ----------
      ----------
        }
```

Subprogram Section

The subprogram section contains all the user defined functions that are used to perform a specific task. These user defined functions are called in the main() function.

C - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then −

Show Examples

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

```c
//Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;
```

```
    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n",c);
    return 0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);

    return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
```

```
--d = 99.500000
```

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then −

Show Examples

Operator	Description	Exampl
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

```c
// Working of relational operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;
```

```
    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

Show Examples

Operator	Description	Exampl

&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) i true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

```c
// Working of logical operators

#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);

    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);

    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);

    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);

    result = !(a != b);
    printf("!(a != b) is %d \n", result);

    result = !(a == b);
    printf("!(a == b) is %d \n", result);

    return 0;
}
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
```

```
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Assignment Operators

The following table lists the assignment operators supported by the C language −

Show Examples

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 i equivale to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C &
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

```c
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
    printf("c = %d\n", c);
    c -= a;     // c is 5
    printf("c = %d\n", c);
    c *= a;     // c is 25
    printf("c = %d\n", c);
    c /= a;     // c is 5
```

```
    printf("c = %d\n", c);
    c %= a;     // c = 0
    printf("c = %d\n", c);

    return 0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Show Examples

Operator	Description	Examp
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), 0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i. 0000 1111

Bitwise AND operator

Bitwise AND operator is denoted by the single ampersand sign (&). Two integer operands are written on both sides of the (&) operator. If the corresponding bits of both the operands are 1, then the output of the bitwise AND operation is 1; otherwise, the output would be 0.

For example,

We have two variables a and b.

a =6;

b=4;

The binary representation of the above two variables are given below:

a = 0110

b = 0100

When we apply the bitwise AND operation in the above two variables, i.e., a&b, the output would be:

Result = 0100

As we can observe from the above result that bits of both the variables are compared one by one. If the bit of both the variables is 1 then the output would be 1, otherwise 0.

Let's understand the bitwise AND operator through the program.

```
#include <stdio.h>
int main()
{
  int a=6, b=14;  // variable declarations
  printf("The output of the Bitwise AND operator a&b is %d",a&b);
  return 0;
}
```

In the above code, we have created two variables, i.e., 'a' and 'b'. The values of 'a' and 'b' are 6 and 14 respectively. The binary value of 'a' and 'b' are 0110 and 1110, respectively. When we apply the AND operator between these two variables,

a AND b = 0110 && 1110 = 0110

Output : The output of the Bitwise AND operator a&b is 6

Bitwise OR operator

The bitwise OR operator is represented by a single vertical sign (|). Two integer operands are written on both sides of the (|) symbol. If the bit value of any of the operand is 1, then the output would be 1, otherwise 0.

For example,

We consider two variables,

a = 23;

b = 10;

The binary representation of the above two variables would be:

a = 0001 0111

b = 0000 1010

When we apply the bitwise OR operator in the above two variables, i.e., a|b , then t

he output would be:

Result = 0001 1111

As we can observe from the above result that the bits of both the operands are compared one by one; if the value of either bit is 1, then the output would be 1 otherwise 0.

Let's understand the bitwise OR operator through a program.

```
#include <stdio.h>
int main()
{
  int a=23,b=10;  // variable declarations
  printf("The output of the Bitwise OR operator a|b is %d",a|b);
  return 0;
}
```

Output :The output of the Bitwise OR operator a|b is 31

Bitwise exclusive OR operator

Bitwise exclusive OR operator is denoted by (^) symbol. Two operands are written on both sides of the exclusive OR operator. If the corresponding bit of any of the operand is 1 then the output would be 1, otherwise 0.

For example,

We consider two variables a and b,

a = 12;

b = 10;

The binary representation of the above two variables would be:

a = 0000 1100

b = 0000 1010

When we apply the bitwise exclusive OR operator in the above two variables (a^b), t

hen the result would be:

Result = 0000 1110

As we can observe from the above result that the bits of both the operands are compared one by one; if the corresponding bit value of any of the operand is 1, then the output would be 1 otherwise 0.

Let's understand the bitwise exclusive OR operator through a program.

```
#include <stdio.h>
int main()
{
  int a=12,b=10;  // variable declarations
  printf("The output of the Bitwise exclusive OR operator a^b is %d",a^b);
  return 0;
}
```

Output : The output of the Bitwise exclusive OR operator a^b is 6

Bitwise complement operator

The bitwise complement operator is also known as one's complement operator. It is represented by the symbol tilde (~). It takes only one operand or variable and performs complement operation on an operand. When we apply the complement operation on any bits, then 0 becomes 1 and 1 becomes 0.

For example,

If we have a variable named 'a',

a = 8;

The binary representation of the above variable is given below:

a = 1000

When we apply the bitwise complement operator to the operand, then the output wo uld be:

Result = 0111

As we can observe from the above result that if the bit is 1, then it gets changed to 0 else 1.

Let's understand the complement operator through a program.

```
#include <stdio.h>
int main()
{
  int a=8;  // variable declarations
  printf("The output of the Bitwise complement operator ~a is %d",~a);
  return 0;
}
```

Output : The output of the Bitwise complement operator ~a is :-9

Bitwise shift operators

Two types of bitwise shift operators exist in C programming. The bitwise shift operators will shift the bits either on the left-side or right-side. Therefore, we can say that the bitwise shift operator is divided into two categories:

- Left-shift operator
- Right-shift operator

Left-shift operator

It is an operator that shifts the number of bits to the left-side.

Syntax of the left-shift operator is given below:

Operand << n

Where,

Operand is an integer expression on which we apply the left-shift operation.

n is the number of bits to be shifted.

In the case of Left-shift operator, 'n' bits will be shifted on the left-side. The 'n' bits on the left side will be popped out, and 'n' bits on the right-side are filled with 0.

For example,

Suppose we have a statement:

```
int a = 5;
```

The binary representation of 'a' is given below:

a = 0101

If we want to left-

shift the above representation by 2, then the statement would be:

a << 2;

0101<<2 = 00010100

Let's understand through a program.

```
#include <stdio.h>
int main()
{
   int a=5; // variable initialization
   printf(": %d ", a<<2);
```

```
    return 0;  The value of a<<2 is
}
```

Output: The value of a<<2 is :20

Right-shift operator

It is an operator that shifts the number of bits to the right side.

Syntax of the right-shift operator is given below:

Operand >> n;

Where,

Operand is an integer expression on which we apply the right-shift operation.

N is the number of bits to be shifted.

In the case of the right-shift operator, 'n' bits will be shifted on the right-side. The 'n' bits on the right-side will be popped out, and 'n' bits on the left-side are filled with 0.

For example,

Suppose we have a statement,

```
int a = 7;
```

The binary representation of the above variable would be:

a = 0111

If we want to right-

shift the above representation by 2, then the statement would be:

a>>2;

0000 0111 >> 2 = 0000 0001

Let's understand through a program.

```
#include <stdio.h>
int main()
{
    int a=7; // variable initialization
    printf("The value of a>>2 is : %d ", a>>2);
    return 0;
}
```

Output: The value of a>>2 is : 1

C Format Specifier

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character.

The commonly used format specifiers in printf() function are:

Format specifier	Description
%d or %i	It is used to print the signed integer value where signed integer means that variable can hold both positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer me the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value alway with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal Value always starts with a 0x value. In this, alphabetical characters are pri small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alph characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints values after '.'.
%e/%E	It is used for scientific notation. It is also known as Mantissa or Exponent.
%g	It is used to print the decimal floating-point values, and it uses the fixed pre i.e., the value after the decimal in input would be exactly the same as the v the output.
%p	It is used to print the address in a hexadecimal form.

%c	It is used to print the unsigned character.
%s	It is used to print the strings.
%ld	It is used to print the long-signed integer value.

Example

```
#include <stdio.h>

main() {

  char ch = 'B';

  printf("%c\n", ch); //printing character data

  //print decimal or integer data with d and i

  int x = 45, y = 90;

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

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

  float f = 12.67;

  printf("%f\n", f); //print float value

  printf("%e\n", f); //print in scientific notation

  int a = 67;

  printf("%o\n", a); //print in octal format

  printf("%x\n", a); //print in hex format

  char str[] = "Hello World";

  printf("%s\n", str);

  printf("%20s\n", str); //shift to the right 20 characters including the string

  printf("%-20s\n", str); //left align

  printf("%20.5s\n", str); //shift to the right 20 characters including the string, and print string up to 5 character
```

```
  printf("%-20.5s\n", str); //left align and print string up to 5 character

}
```

Output

```
B
45
90
12.670000
1.267000e+001
103
43
Hello World
Hello World
Hello World
Hello
Hello
```

C Input and Output

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

`scanf()` and `printf()` functions

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf(), which are used to display output on screen and to take input from user respectively.

```
#include<stdio.h>

void main()
{
   // defining a variable
   int i;
   /*
      displaying message on the screen
      asking the user to input a value
   */
   printf("Please enter a value...");
   /*
      reading the value entered by the user
   */
```

```
   scanf("%d", &i);
   /*
      displaying the number as output
   */
   printf( "\nYou entered: %d", i);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

You must be wondering what is the purpose of %d inside the scanf() or printf() functions. It is known as **format string** and this informs the scanf() function, what type of input to expect and in printf() it is used to give a heads up to the compiler, what type of output to expect.

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

We can also **limit the number of digits or characters** that can be input or output, by adding a number with the format string specifier, like "%1d" or "%3s", the first one means a single numeric digit and the second one means 3 characters, hence if you try to input 42, while scanf() has "%1d", it will take only 4 as input. Same is the case for output.

In C Language, computer monitor, printer etc output devices are treated as files and the same process is followed to write output to these devices as would have been followed to write the output to a file.

NOTE : printf() function returns the number of characters printed by it, and scanf() returns the number of characters read by it.

`getchar()` & `putchar()` functions

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The putchar() function displays the character passed to

it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use putchar() method in a loop.

```
#include <stdio.h>

void main( )
{
   int c;
   printf("Enter a character");
   /*
      Take a character as input and
      store it in variable c
   */
   c = getchar();
   /*

      display the character stored

      in variable c

   */

   putchar(c);

}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

`gets()` & `puts()` functions

The gets() function reads a line from **stdin**(standard input) into the buffer pointed to str pointer, until either a terminating newline or EOF (end of file) occurs. The puts() function writes the string str and a trailing newline to **stdout**.

str → This is the pointer to an array of char where the C string is stored.

```
#include<stdio.h>

int main()
{
   /* character array of length 100 */
   char str[100];
   printf("Enter a string");
   gets( str );
   puts( str );
```

```
    return 0;
}
```

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Escape Sequence in C

An escape sequence in C language is a sequence of characters,It is composed of two or more characters starting with backslash \. For example: \n represents new line.

List of Escape Sequences in C

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number

\xhh	hexadecimal number
\0	Null

Example

```
#include<stdio.h>
int main()
  {
    int number=50;

printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\"");
   return 0;
      }
```

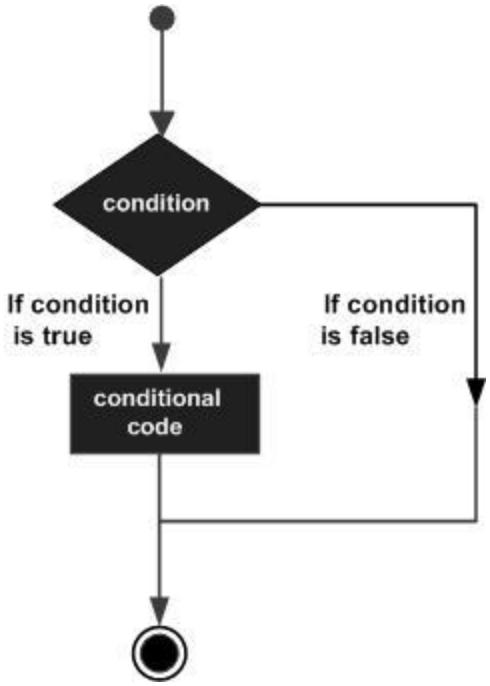
Output:

```
You
are
learning
'c' language
"Do you know C language"
```

Decision Control Statements

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 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 mor statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes whe 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).

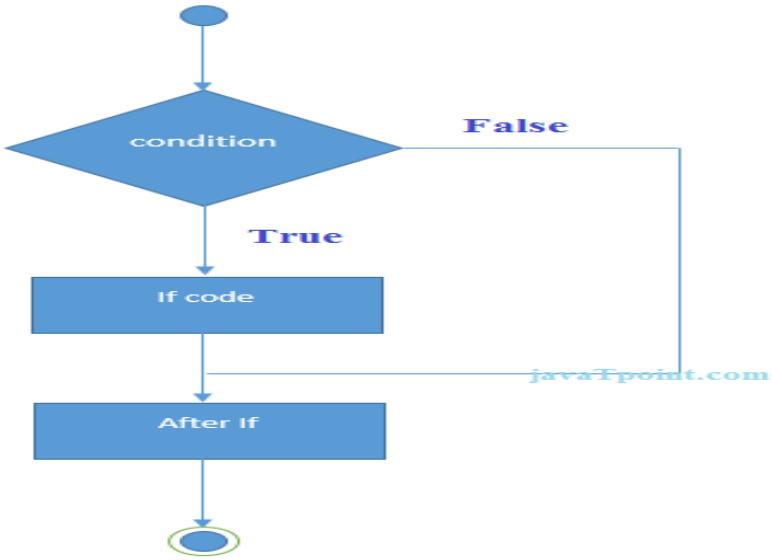
- If statement
- If-else statement
- If else-if ladder
- Nested if

If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

```
if(expression)
    {
//code to be executed
            }
```

Flowchart of if statement in C



Let's see a simple example of C language if statement.

```c
#include<stdio.h>
int main(){
int number=0;
printf("Enter a number:");
scanf("%d",&number);
if(number%2==0){
printf("%d is even number",number);
}
return 0;
}
```

Output

```
Enter a number:4
4 is even number
enter a number:5
```

```c
//Program to find the largest number of the three.
#include <stdio.h>
int main()
{
   int a, b, c;
    printf("Enter three numbers?");
   scanf("%d %d %d",&a,&b,&c);
   if(a>b && a>c)
   {
      printf("%d is largest",a);
   }
   if(b>a  && b > c)
   {
      printf("%d is largest",b);
   }
   if(c>a && c>b)
   {
      printf("%d is largest",c);
   }
   if(a == b && a == c)
   {
      printf("All are equal");
   }
```

```
}
```

Output

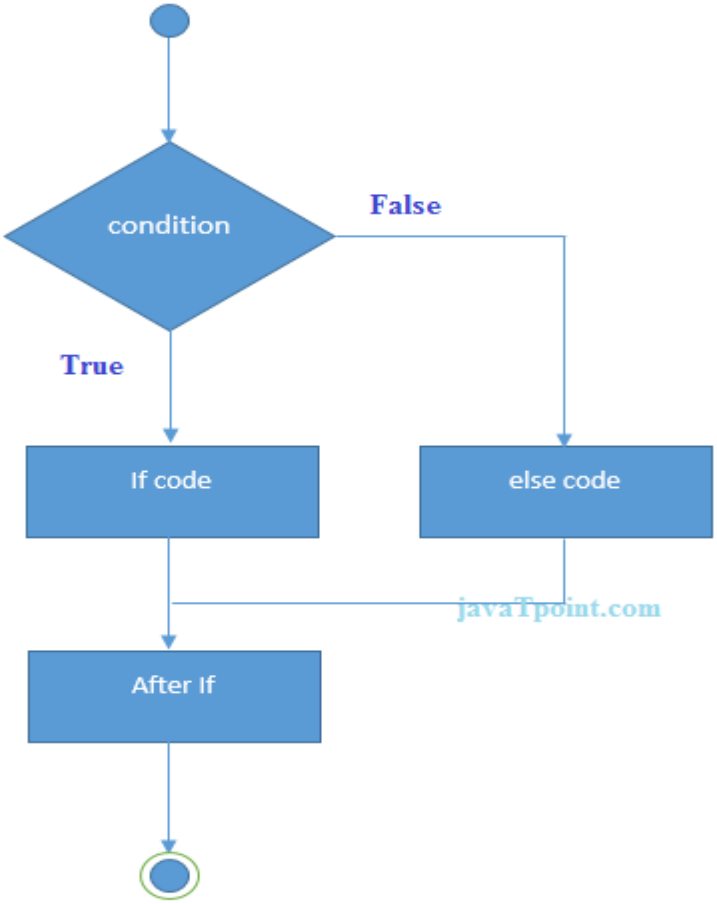
```
Enter three numbers?
12 23 34
34 is largest
```

If-else Statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simiulteneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition. The syntax of the if-else statement is given below.

```
if(expression)
 {
  //code to be executed if condition is true
    }
 else
 {
//code to be executed if condition is false
}
```

Flowchart of the if-else statement in C



Let's see the simple example to check whether a number is even or odd using if-else statement in C language.

```
#include<stdio.h>
int main()
 {
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number%2==0)
{
printf("%d is even number",number);
}
else
{
printf("%d is odd number",number);
}
```

```
return 0;
}
```

Output

```
enter a number:4
4 is even number
enter a number:5
5 is odd number
```

```
//Program to check whether a person is eligible to vote or not.
#include <stdio.h>
int main()
{
   int age;
   printf("Enter your age?");
   scanf("%d",&age);
   if(age>=18)
   {
      printf("You are eligible to vote...");
   }
   else
   {
      printf("Sorry ... you can't vote");
   }
}
```

Output

```
Enter your age?18
You are eligible to vote...
Enter your age?13
Sorry ... you can't vote
```

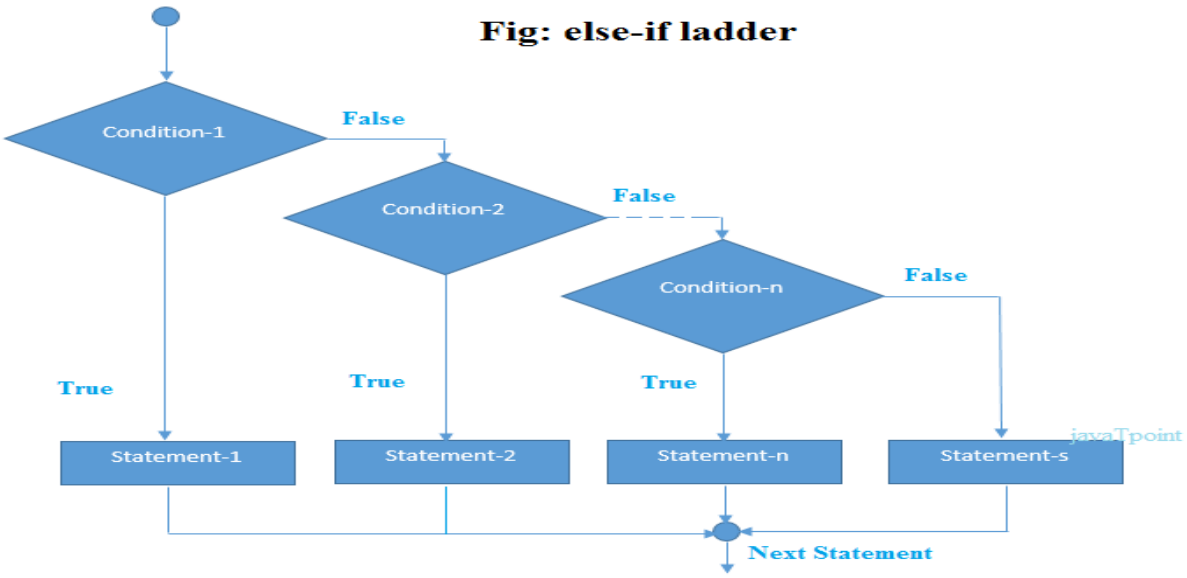
If else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks

possible. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

```
if(condition1)
 {
//code to be executed if condition1 is true
}
else if(condition2)
 {
//code to be executed if condition2 is true
}
else if(condition3)
{
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Flowchart of else-if ladder statement in C



The example of an if-else-if statement in C language is given below.

```c
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number==10){
printf("number is equals to 10");
}
else if(number==50){
printf("number is equal to 50");
}
else if(number==100){
printf("number is equal to 100");    }
else{
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Output

```
enter a number:4
number is not equal to 10, 50 or 100
enter a number:50
number is equal to 50
```

```c
//Program to calculate the grade of the student according to the specified marks.
#include <stdio.h>
int main()
{
   int marks;
   printf("Enter your marks?");
   scanf("%d",&marks);
   if(marks > 85 && marks <= 100)
   {
      printf("Congrats ! you scored grade A ...");
   }
   else if (marks > 60 && marks <= 85)
```

```
    {
       printf("You scored grade B + ...");
    }
    else if (marks > 40 && marks <= 60)
    {
       printf("You scored grade B ...");
    }
    else if (marks > 30 && marks <= 40)
    {
       printf("You scored grade C ...");
    }
    else
    {
       printf("Sorry you are fail ...");
    }
}
```

Output:

```
Enter your marks?50

You scored grade B ...
```

C Switch Case Statement

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable. Here, We can define various statements in the multiple cases for the different values of a single variable.

The syntax of switch statement in C language is given below:

```
switch(expression)
  {
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
```

```
 break;  //optional
.....
   default:
 code to be executed if all cases are not matched;
}
```

Rules for switch statement in C language

1) The *switch expression* must be of an integer or character type.

2) The *case value* must be an integer or character constant.

3) The *case value* can be used only inside the switch statement.

4) The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

Let's try to understand it by the examples. We are assuming that there are following variables.

```
int x,y,z;
char a,b;
float f;
```

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f);	case 3:	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a':	case x;
switch(a+b-2)		case 1+2:	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Flowchart of switch statement in C

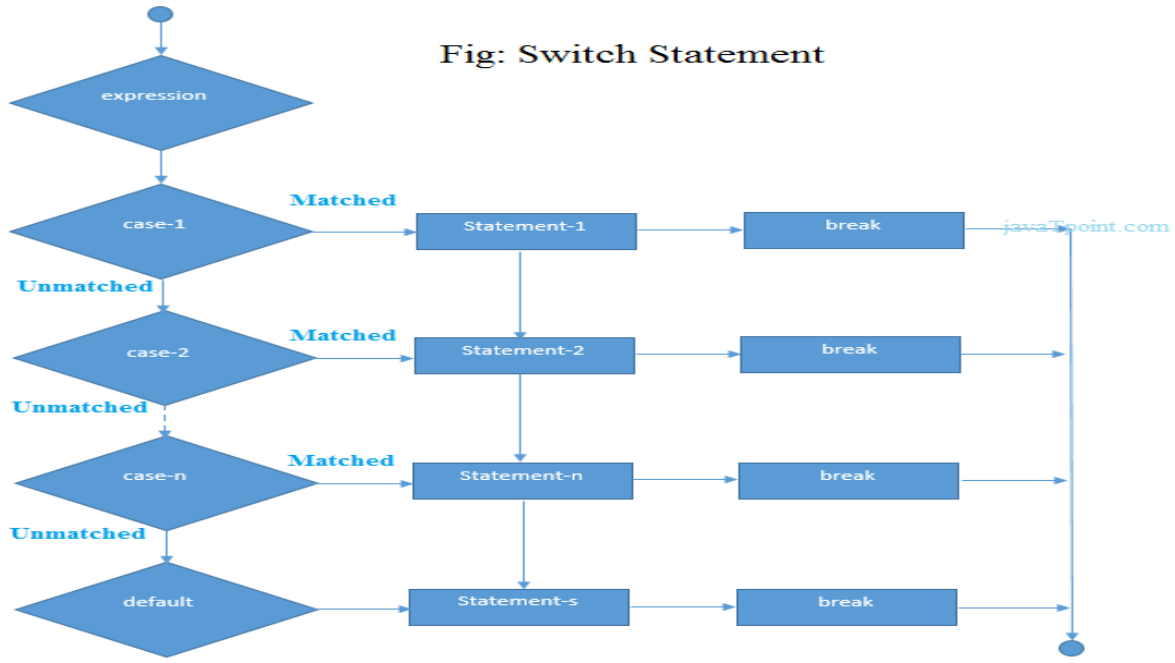


Fig: Switch Statement

Functioning of switch case statement

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

//Let's see a simple example of c language switch statement.

```
#include<stdio.h>
int main()
{
int number=0;
printf("enter a number:");
scanf("%d",&number);
switch(number){
case 10:    printf("number is equals to 10");
break;
case 50:
printf("number is equal to 50");
```

```
break;
case 100:
printf("number is equal to 100");
break;
default:
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Output

```
enter a number:4
number is not equal to 10, 50 or 100
enter a number:50
number is equal to 50
```

//Switch case example 2

```
#include <stdio.h>
int main()
{
   int x = 10, y = 5;
   switch(x>y && x+y>0)
   {
      case 1:
      printf("hi");
      break;
      case 0:
      printf("bye");
      break;
      default:
      printf(" Hello bye ");
           }
       }
```

Output

```
hi
```

C Switch statement is fall-through

In C language, the switch statement is fall through; it means if you don't use a break statement in the switch case, all the cases after the matching case will be executed.

Let's try to understand the fall through state of switch statement by the example given below.

```
#include<stdio.h>
int main()
{
int number=0;
printf("enter a number:");
scanf("%d",&number);
switch(number){
case 10:
printf("number is equal to 10\n");
case 50:
printf("number is equal to 50\n");
case 100:
printf("number is equal to 100\n");
default:
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Output

```
enter a number:10
number is equal to 10
number is equal to 50
number is equal to 100
number is not equal to 10, 50 or 100
```

Nested switch case statement

We can use as many switch statement as we want inside a switch statement. Such type of statements is called nested switch case statements. Consider the following example.

```
#include <stdio.h>
int main ()
  {
 int i = 10;
 int j = 20;
 switch(i)
{
    case 10:
      printf("the value of i evaluated in outer switch: %d\n",i);
    case 20:
      switch(j)
       {
        case 20:
          printf("The value of j evaluated in nested switch: %d\n",j);
      }
 }
  printf("Exact value of i is : %d\n", i );
  printf("Exact value of j is : %d\n", j );
  return 0;
}
```

Output

```
the value of i evaluated in outer switch: 10
The value of j evaluated in nested switch: 20
Exact value of i is : 10
Exact value of j is : 20
```

LOOPS IN C

In programming, a loop is used to repeat a block of code until the specified condition is met.

C programming has three types of loops:

1.for loop

2.while loop

3.do...while loop

1.for loop:

The **for loop in C language** is used to iterate the statements or a part of the program several times.

The syntax of the `for` loop is:

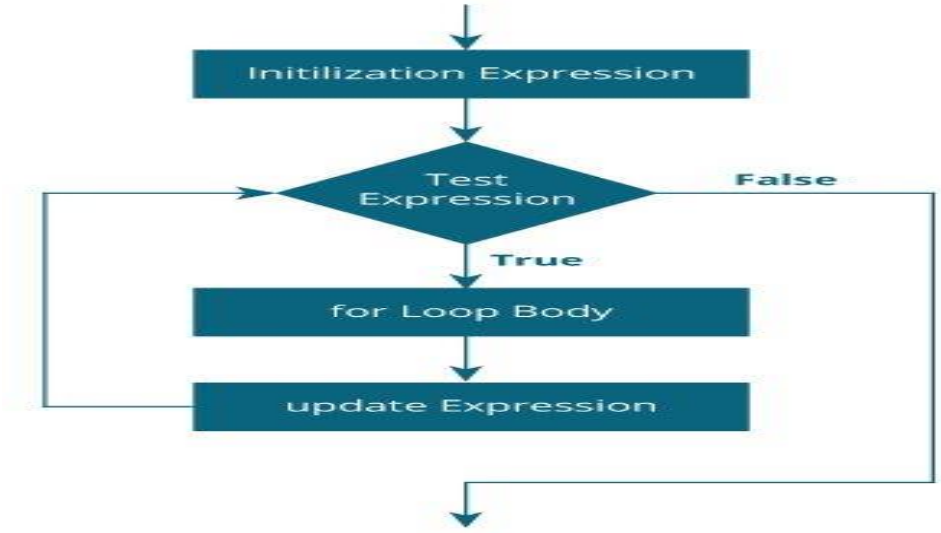
```
for (initializationStatement; testExpression; updateStatement)
{
   // statements inside the body of loop
}
```

How for loop works?

- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the `for` loop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of the `for` loop are executed, and the update expression is updated.
- Again the test expression is evaluated.

This process goes on until the test expression is false. When the test expression is false, the loop terminates.

for loop Flowchart



Example 1: for loop

```
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
  int i;
  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

1. `i` is initialized to 1.
2. The test expression `i < 11` is evaluated. Since 1 less than 11 is true, the body of `for` loop is executed. This will print the **1** (value of `i`) on the screen.
3. The update statement `++i` is executed. Now, the value of `i` will be 2. Again, the test expression is evaluated to true, and the body of `for` loop is executed. This will print **2** (value of `i`) on the screen.
4. Again, the update statement `++i` is executed and the test expression `i < 11` is evaluated. This process goes on until `i` becomes 11.
5. When `i` becomes 11, `i < 11` will be false, and the `for` loop terminates.

Example 2: for loop

```
// Program to calculate the sum of first n natural numbers
#include <stdio.h>
int main()
{
   int num, count, sum = 0;
   printf("Enter a positive integer: ");
   scanf("%d", &num);
   // for loop terminates when num is less than count
   for(count = 1; count <= num; ++count)
   {
      sum += count;
   }
   printf("Sum = %d", sum);
   return 0;
}
```

Output

```
Enter a positive integer: 10
Sum = 55
```

NESTED FOR LOOPS IN C

C programming allows to use one loop inside another loop.

Syntax

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

```
for( init; condition; increment )
 {

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

Example:

```
#include <stdio.h>
int main()
{
  int n;// variable declaration
  printf("Enter the value of n :");
  // Displaying the n tables.
  for(int i=1;i<=n;i++)  // outer loop
  {
     for(int j=1;j<=10;j++)  // inner loop
     {
        printf("%d\t",(i*j)); // printing the value.
     }
     printf("\n");
  }
```

Explanation of the above code

- First, the 'i' variable is initialized to 1 and then program control passes to the i<=n.
- The program control checks whether the condition 'i<=n' is true or not.
- If the condition is true, then the program control passes to the inner loop.
- The inner loop will get executed until the condition is true.
- After the execution of the inner loop, the control moves back to the update of the outer loop, i.e., i++.
- After incrementing the value of the loop counter, the condition is checked again, i.e., i<=n.
- If the condition is true, then the inner loop will be executed again.
- This process will continue until the condition of the outer loop is true.

Output:

EXAMPLE -2

```
#include <stdio.h>
int main()
{
  for (int i=0; i<2; i++)
  {
          for (int j=0; j<4; j++)
          {
            printf("%d, %d\n",i ,j);
          }
  }
  return 0;
}
```

Output:

```
0, 0
0, 1
0, 2
0, 3
1, 0
1, 1
1, 2
```

1, 3

2.while loop in C

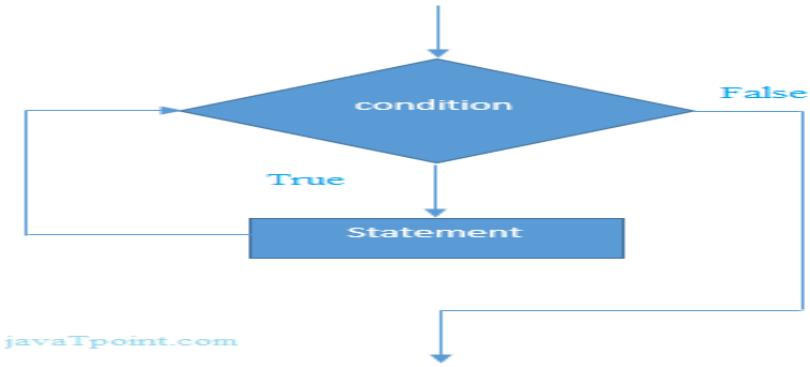
While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

Syntax of while loop in C language

The syntax of while loop in c language is given below:

```
while(condition){
//code to be executed
}
```

Flowchart of while loop in C



Example:

```
#include<stdio.h>
int main()
 {
int i=1;
while(i<=10){
printf("%d \n",i);
i++;
}
return 0;
```

```
}
```

Output:

Example:

```
#include<stdio.h>
int main()
   {
int i=1,number=0,b=9;
printf("Enter a number: ");
scanf("%d",&number);
while(i<=10){
printf("%d \n",(number*i));
i++;
}
return 0;
}
```

Output:

3.do while loop in C

The do while loop is a post tested loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

do while loop syntax

The syntax of the C language do-while loop is given below:

```
do{
//code to be executed
}while(condition);
```

Example

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

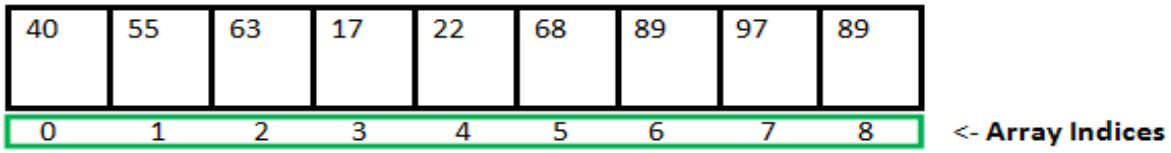
```c
int main () {

  /* local variable definition */
  int a = 10;

  /* do loop execution */
  do {
    printf("value of a: %d\n", a);
    a = a + 1;
  }while( a < 20 );

  return 0;
}
```

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

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

Arrays

An array is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type.



Array Length = 9
First Index = 0
Last Index = 8

Why do we need arrays?

We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

Properties of an Array

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e.,

  int = 2 bytes.

- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantages of an Array

1) Code Optimization: Less code to the access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of an Array

1) **Fixed Size**: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Array types:

There are mainly three types of the arrays:

One Dimensional (1D) Array
Multidimensional Array

One Dimensional Array:

The number of subscript or index determines the dimensions of the array. An array of one dimension is known as a one-dimensional array or 1-D array

Declaration of an Array

We can declare an array in the c language in the following way.

```
1.data_type array_name[array_size];
```

Now, let us see the example to declare the array.

```
2. int marks[5];
```

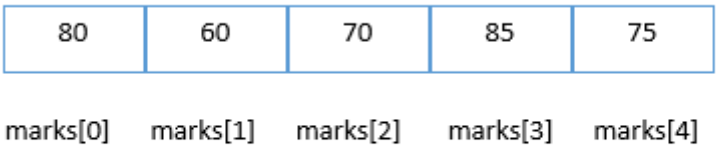
Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

Initialization of an Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index.

Consider the following example.

```
//int marks[5];
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```



Initialization of Array

Example:

```
#include<stdio.h>
int main()
 {
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
//traversal of array
for(i=0;i<5;i++)
 {
   printf("%d \n",marks[i]);
}//end of for loop
return 0;
}
```

Output:

80

60

70

85

75

Array Declaration with Initialization

We can initialize the array at the time of declaration

1. int marks[5]={20,30,40,50,60};

In another case, there is no requirement to define the size. So it may also be declare in the below format

2. int marks[]={20,30,40,50,60};

Let's see the C program to declare and initialize the array in C.

Example:

```
#include<stdio.h>
int main(){
int i=0;
int marks[5]={20,30,40,50,60};//declaration and initialization of array
 //traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
}
return 0;
}
```

Output :20 30 40 50 60

Example:

```
#include<stdio.h>
int main()
 {
    int marks[5],i;
    printf("\n ENTER MARKS:");
    for(i=0;i<5;i++)
```

```
    {
        scanf("%d",&marks[i]);
    }
printf("\n ENTERED MARKS:");
for(i=0;i<5;i++)
{
printf("%d \n",marks[i]);
}
return 0;
}
```

Output:

```
ENTER MARKS:90
63
72
45
54
ENTERED MARKS:90
63
72
45
54
```

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.

Declaration of two dimensional Array

The syntax to declare the 2D array is given below.

1. data_type array_name[rows][columns];

Consider the following example.

2. int x [3][3];

Here, the first 3 is the number of rows, and next 3 is the number of columns.

	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]

Example:

```
#include<stdio.h>
int main(){
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
for(i=0;i<4;i++)
  {
 for(j=0;j<3;j++)
        {
  printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
       }//end of j
  }//end of i
return 0;
}
```

Output:

```
arr[0] [0] = 1

arr[0] [1] = 2

arr[0] [2] = 3

arr[1] [0] = 2

arr[1] [1] = 3
```

arr[1] [2] = 4

arr[2] [0] = 3

arr[2] [1] = 4

arr[2] [2] = 5

arr[3] [0] = 4

arr[3] [1] = 5

arr[3] [2] = 6

Example:

```
#include <stdio.h>
void main ()
{
   int arr[3][3],i,j;
   for (i=0;i<3;i++)
   {
      for (j=0;j<3;j++)
      {
         printf("Enter a[%d][%d]: ",i,j);
         scanf("%d",&arr[i][j]);
      }
   }
   printf("\n printing the elements ....\n");
   for(i=0;i<3;i++)
   {
      printf("\n");
      for (j=0;j<3;j++)
      {
         printf("%d\t",arr[i][j]);
      }
   }
}
```

Output:

Enter a[0][0]: 10

Enter a[0][1]: 20

Enter a[0][2]: 30

Enter a[1][0]: 40

Enter a[1][1]: 50

Enter a[1][2]: 60

Enter a[2][0]: 70

Enter a[2][1]: 80

Enter a[2][2]: 90

printing the elements ....

10 20 30

40 50 60

70 80 90

Matrix addition

```
#include <stdio.h>

int main()
{
  int m, n, c, d, first[10][10], second[10][10], sum[10][10];

  printf("Enter the number of rows and columns of matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter the elements of first matrix\n");

  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
```

```
      scanf("%d", &first[c][d]);

  printf("Enter the elements of second matrix\n");

  for (c = 0; c < m; c++)
    for (d = 0 ; d < n; d++)
      scanf("%d", &second[c][d]);

  printf("Sum of entered matrices:-\n");

  for (c = 0; c < m; c++) {
    for (d = 0 ; d < n; d++) {
      sum[c][d] = first[c][d] + second[c][d];
      printf("%d\t", sum[c][d]);
    }
    printf("\n");
  }

  return 0;
}
```

Output:

```
 Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
1
2
3
4
Enter the elements of second matrix
1
2
3
3

Sum of entered matrices:-
2       4
6       7
```

Strings

- The string can be defined as the one-dimensional array of characters terminated by a null ('\0').
- The character array or the string is used to manipulate text such as word or sentences.
- Each character in the String occupies one byte of memory, and the last character must always be' \0'.
- The termination character ('\0') is important in a string since it is the only way to identify where the string ends.
- When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring string by char array in C language.

```
char ch[10]={'b', 'h', 'a', 'v', 'm', 'a', 'n', 'y', 'u', 'n', '\0'};
```

As we know, array index starts from 0, so it will be represented as in the figure given below.

0 1 2 3 4 5 6 7 8 9 10

b	h	a	v	m	a	n	y	u	n	\0

While declaring string, size is not mandatory. So we can write the above code as given below:

```
char ch[]={'b', 'h', 'a', 'v', 'm', 'a', 'n', 'y', 'u', 'n', '\0'};
```

We can also define the **string by the string literal** in C language.

For example:

```
char ch[]="bhavmanyun";
```

In the above case, '\0' will be appended at the end of the string by the compiler.

Difference between char array and string literal

There are two main differences between char array and literal.

- In character Array We need to add the null character '\0' at the end of the array by ourself
- In string literal we don't need to add null character the compiler itself appended at the end of the string.

Example:

```
#include<stdio.h>
#include <string.h>
 int main()
   {
  char ch[11]={'b', 'h', 'a', 'v', 'm', 'a', 'n', 'y', 'u', 'n', '\0'};
  char ch2[11]="bhavmanyun";
  printf("Char Array Value is: %s\n", ch);
  printf("String Literal Value is: %s\n", ch2);
 return 0;
}
```

Output :

Char Array Value is: bhavmanyun

String Literal Value is: bhavmanyun

Traversing String

- Traversing the string is one of the most important aspects in any of the programming languages.
- We may need to manipulate a very large text which can be done by traversing the text.
- Traversing string is somewhat different from the traversing an integer array.
- We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string
- By using the null character.

Let's discuss each one of them.

Using the length of string

Let's see an example of counting the number of vowels in a string.

```
#include<stdio.h>
void main ()
{
   char s[11] = "Bhavmanyun";
   int i = 0;
   int count = 0;
   while(i<11)
   {
      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
      {
         count ++;
      }
      i++;
   }
   printf("The number of vowels %d",count);
}
```

Output: The number of vowels 3

Using the null character

Let's see the same example of counting the number of vowels by using the null character.

```
#include<stdio.h>

void main ()
{
   char s[11] = "Bhavmanyun";
   int i = 0;
   int count = 0;
```

```
   while(s[i] !=NULL)
   {
      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
      {
         count ++;
      }
      i++;
   }
   printf("The number of vowels %d",count);
   getch();
}
```

Output: The number of vowels 3

Accepting string as the input

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario.

Consider the below code which stores the string while space is encountered.

```
#include<stdio.h>
void main ()
{
   char s[20];
   printf("Enter the string?");
   scanf("%s",s);
   printf("You entered %s",s);
}
```

Output:

```
Enter the string?ant
You entered ant
```

- It is clear from the output that, the above code will not work for space separated strings.
- To make this code working for the space separated strings, the minor changed required in the scanf function, i.e., instead of writing scanf("%s",s), we must write: scanf("%[^\n]s",s) which instructs the compiler to store the string s while the new line (\n) is encountered.

Let's consider the following example to store the space-separated strings.

```
#include<stdio.h>
void main ()
{
   char s[20];
   printf("Enter the string?");
   scanf("%[^\n]s",s);
   printf("You entered %s",s);
}
```

Output:

```
Enter the string?siddartha
You entered siddartha
```

- Here we must also notice that we do not need to use address of (&) operator in scanf to store a string
- Since string s is an array of characters and the name of the array, i.e., s indicates the base address of the string (character array) therefore we need not use & with it.

gets() and puts() functions

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

gets() function

- The gets() function enables the user to enter some characters followed by the enter key.
- All the characters entered by the user get stored in a character array.
- The null character is added to the array to make it a string.
- The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Example:

```
//Reading string using gets()
#include<stdio.h>
void main ()
{
   char s[30];
   printf("Enter the string? ");
   gets(s);
   printf("You entered %s",s);
}
```

Output:

```
Enter the string? cse
You entered cse
```

C puts() function

- The puts() function is very much similar to printf() function.
- The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function.

Let's see an example to read a string using gets() and print it on the console using puts().

```
#include<stdio.h>
#include <string.h>
int main()
{
char name[50];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name);  //displays string
return 0;
}
```

Output:

Enter your name: anand

Your name is: anand

C String Functions

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

String Length: strlen() function

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

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

```
#include <string.h>
int main()
   {
char ch[20]={'s', 'i', 'd', 'd', 'a', 'r', 't', 'h', 'a', 't', '\0'};
  printf("Length of string is: %d",strlen(ch));
 return 0;
}
```

Output:

Length of string is: 10

Copy String: strcpy()

The strcpy(destination, source) function copies the source string in destination.

```
#include<stdio.h>
#include <string.h>
 int main()
    {
  char ch[20]={'s', 'i', 'd', 'd', 'a', 'r', 't', 'h', 'a', '\0'};
  char ch2[20];
  strcpy(ch2,ch);
  printf("Value of second string is: %s",ch2);
 return 0;
}
```

Output:

Value of second string is: siddartha

String Concatenation: strcat()

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

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

```
  {
  char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
  char ch2[10]={'c', '\0'};
  strcat(ch,ch2);
  printf("Value of first string is: %s",ch);
 return 0;
}
```

Output: Value of first string is: helloc

Compare String: strcmp()

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using *gets()* function which reads string from the console.

```
#include<stdio.h>
#include <string.h>
int main()
{
  char str1[20],str2[20];
  printf("Enter 1st string: ");
  gets(str1);//reads string from console
  printf("Enter 2nd string: ");
  gets(str2);
  if(strcmp(str1,str2)==0)
     printf("Strings are equal");
  else
     printf("Strings are not equal");
 return 0;
}
```

Output:

```
 Enter 1st string: ant
Enter 2nd string: ant
Strings are equal
```

Reverse String: strrev()

The strrev(string) function returns reverse of the given string.

Let's see a simple example of strrev() function.

```
#include<stdio.h>
#include <string.h>
int main()
       {
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
 printf("\nReverse String is: %s",strrev(str));
 return 0;
}
```

Output:

```
Enter string: ant
Reverse String is: tna
```

String Lowercase: strlwr()

The strlwr(string) function returns string characters in lowercase.

Let's see a simple example of strlwr() function.

```
#include<stdio.h>
#include <string.h>
int main()
    {
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nLower String is: %s",strlwr(str));
 return 0;
}
```

Output:

```
Enter string:ANT
Lower String is: ant
```

String Uppercase: strupr()

The strupr(string) function returns string characters in uppercase.

Let's see a simple example of strupr() function.

```
#include<stdio.h>
#include <string.h>
int main()
  {
 char str[20];
 printf("Enter string: ");
 gets(str);//reads string from console
 printf("String is: %s",str);
 printf("\nUpper String is: %s",strupr(str));
return 0;
   }
```

Output:

```
Enter string:ant
String is:ant
Lower String is: ANT
```