C Language

C programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. C is the most widely used computer language.

```
1. #include <stdio.h>
2. int main(){
3. printf("Hello C Language");
4. return 0;
5. }
```

#include <stdio.h> includes the **standard input output** library functions. The printf() function is d--efined in stdio.h .

int main() The **main() function is the entry point of every program** in c language.**printf()** The printf() function is **used to print data** on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

printf() and scanf() in C

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:

```
printf("format string",argument_list);
```

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The **scanf() function** is used for input. It reads the input data from the console.

scanf("format string",argument_list);

Program to print cube of given number

Let's see a simple example of c language that gets input from the user and prints the cube of the given number.

```
#include<stdio.h>
int main(){
int number;
printf("enter a number:");
scanf("%d",&number);
printf("cube of number is:%d ",number*number*number);
return 0;
}
```

Output

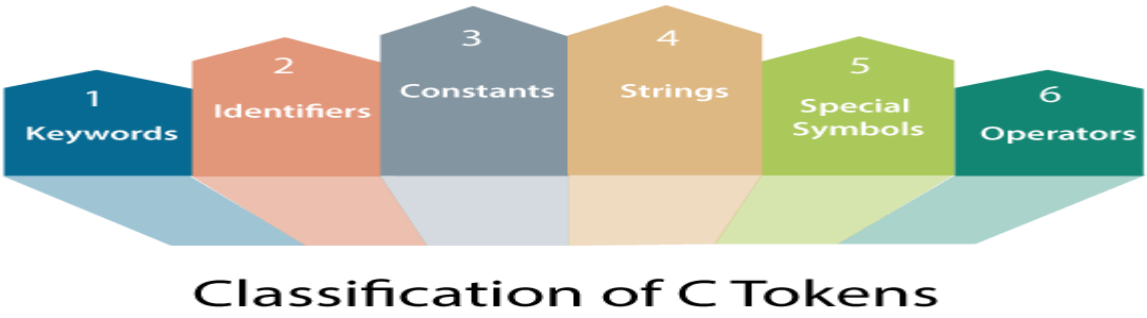
```
enter a number:5
cube of number is:125
```

The **scanf("%d",&number)** statement reads integer number from the console and stores the given value in number variable.

The **printf("cube of number is:%d ",number*number*number)** statement prints the cube of number on the console.

Tokens in C

Tokens in C is the most important element to be used in creating a program in C. We can define the token as the smallest individual element in C. For `example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C. Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language



Classification of C Tokens

- Keywords in C
- Identifiers in C
- Strings in C
- Operators in C
- Constant in C
- Special Characters in C

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:

type variable_list;

The example of declaring the variable is given below:

```
1. int a;
2. float b;
3. 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:	Invalid variable names:
`int a;`	`int 2;`
`int _ab;`	`int a b;`
`int a30;`	`int long;`

Types of Variables in C

There are many types of variables in c:local variable

1. global variable
2. static variable
3. automatic variable
4. external 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.

```
1. void function1(){
2. int x=10;//local variable
3. }
```

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.

```
1. int value=20;//global variable
2. void function1(){
3. int x=10;//local variable
4. }
```

Static Variable

A variable that is declared with the static keyword is called static variable.

It retains its value between multiple function calls.

```
1. void function1(){
2. int x=10;//local variable
3. static int y=10;//static variable
4. x=x+1;
5. y=y+1;
6. printf("%d,%d",x,y);
7. }
```

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

Data Types	Memory Size	Range
Char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
Short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
Int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
Float	4 byte	
Double	8 byte	

long double	10	byte	

Keywords in C

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

auto	break	Case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

C Identifiers

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.

- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers

total, sum, average, _m _, sum_1, etc.

.

Operators in C

is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:

Unary Operator

A unary operator is an operator applied to the single operand. For example: increment operator (++), decrement operator (--), sizeof, (type)*.

Binary Operator

The binary operator is an operator applied between two operands. The following is the list of the binary operators:

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Conditional Operators
- Assignment Operator
- Misc Operator

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

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	Example
==	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.

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	Example
&&	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) is 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.

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 is equivalent 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 & 2
^=	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

Bitwise Operator in C

The bitwise operators are the operators used to perform the operations on the data at the bit-level. When we perform the bitwise operations, then it is also known as bit-level programming. It consists of two digits, either 0 or 1. It is mainly used in numerical computations to make the calculations faster.

We have different types of bitwise operators in the C programming language.

Operator	Meaning of operator
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
~	One's complement operator (unary operator)
<<	Left shift operator
>>	Right shift operator

Conditional Operator in C

The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends

upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

As conditional operator works on three operands, so it is also known as the ternary operator.

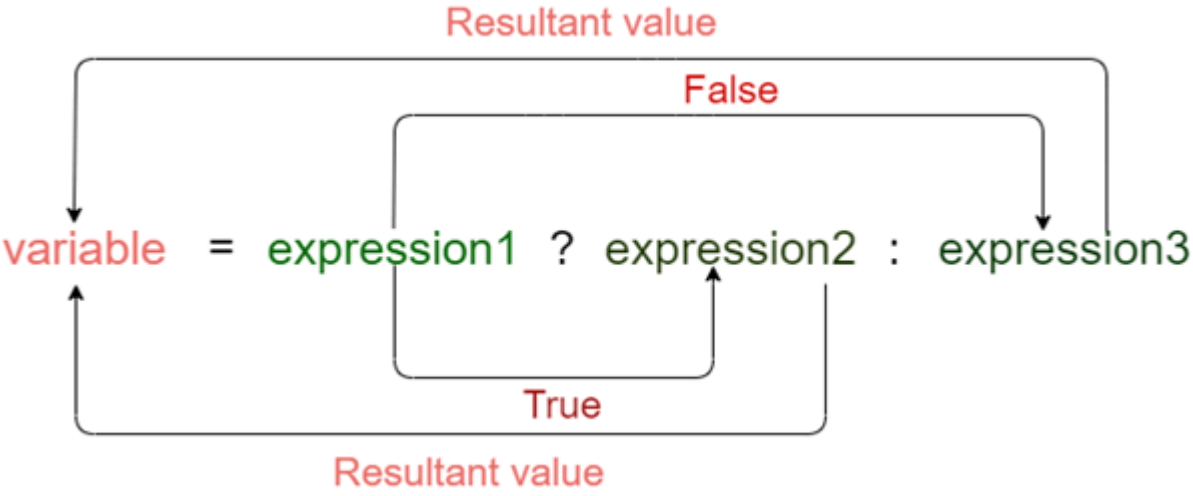
The behavior of the conditional operator is similar to the 'if-else

' statement as 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

```
Expression1? expression2: expression3;
```

The pictorial representation of the above syntax is shown below:



Meaning of the above syntax.

- In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
- If the expression1 results into a true value, then the expression2 will execute.
- The expression2 is said to be true only when it returns a non-zero value.
- If the expression1 returns false value then the expression3 will execute.
- The expression3 is said to be false only when it returns zero value.

Let's understand the ternary or conditional operator through an example.

```c
#include <stdio.h>
int main()
{
    int age;  // variable declaration
    printf("Enter your age");
    scanf("%d",&age);   // taking user input for age variable
    (age>=18)? (printf("eligible for voting")) : (printf("not eligible for voti
ng"));  // conditional operator
    return 0;
}
```

Constants in C

A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed.

There are two ways of declaring constant:

- Using const keyword
- Using #define pre-processor

CONTROL STATEMENT

if else Statement

The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

There are the following variants of if statement in C language.

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

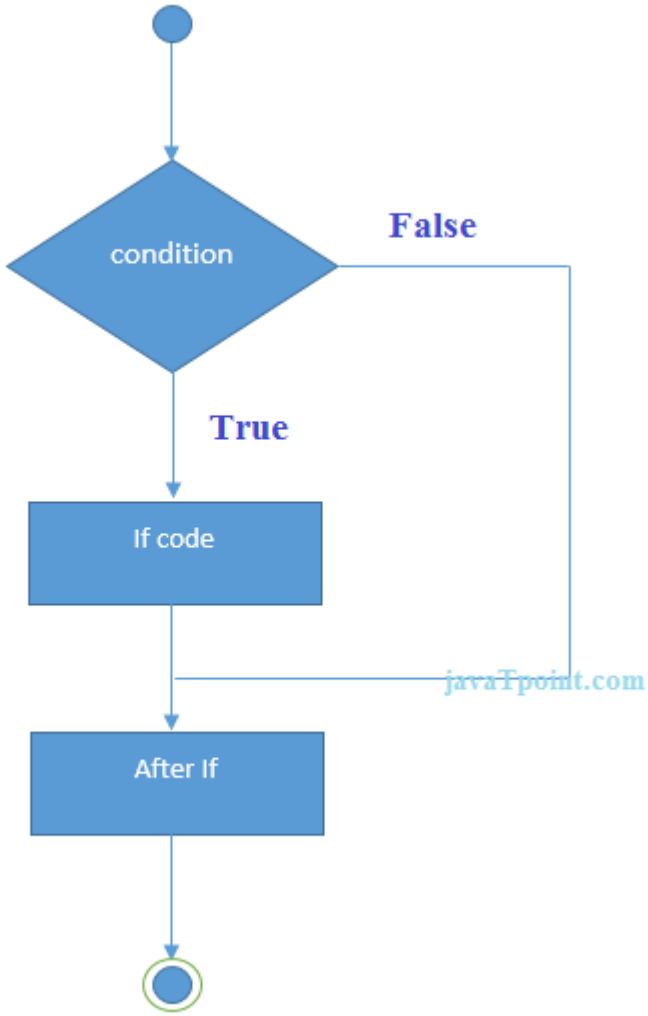
- Nested if

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.

```
1. if(expression){
2. //code to be executed
3. }
```

Flowchart of if statement in C



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

```
1. #include<stdio.h>
2. int main(){
3. int number=0;
4. printf("Enter a number:");
5. scanf("%d",&number);
6. if(number%2==0){
7. printf("%d is even number",number);
8. }
9. return 0;
10. }
```

Output

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

Program to find the largest number of the three.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a, b, c;
5.      printf("Enter three numbers?");
6.     scanf("%d %d %d",&a,&b,&c);
7.     if(a>b && a>c)
8.     {
9.         printf("%d is largest",a);
10.    }
11.    if(b>a  && b > c)
12.    {
13.        printf("%d is largest",b);
14.    }
15.    if(c>a && c>b)
16.    {
17.        printf("%d is largest",c);
18.    }
19.    if(a == b && a == c)
20.    {
21.        printf("All are equal");
22.    }
23. }
```

Output

Enter three numbers?

12 23 34

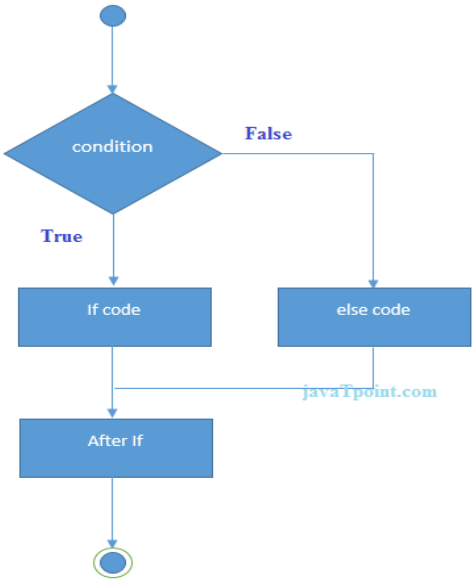
34 is largest

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.

```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);
}
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.

```c
#include <stdio.h>
int main()
{
    int age;
    printf("Enter your age?");
    scanf("%d",&age);
    if(age>=18)
    {
        printf("You are eligible to vote...");
    }
```

```
11.    else
12.    {
13.        printf("Sorry ... you can't vote");
14.    }
15. }
```

Output

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

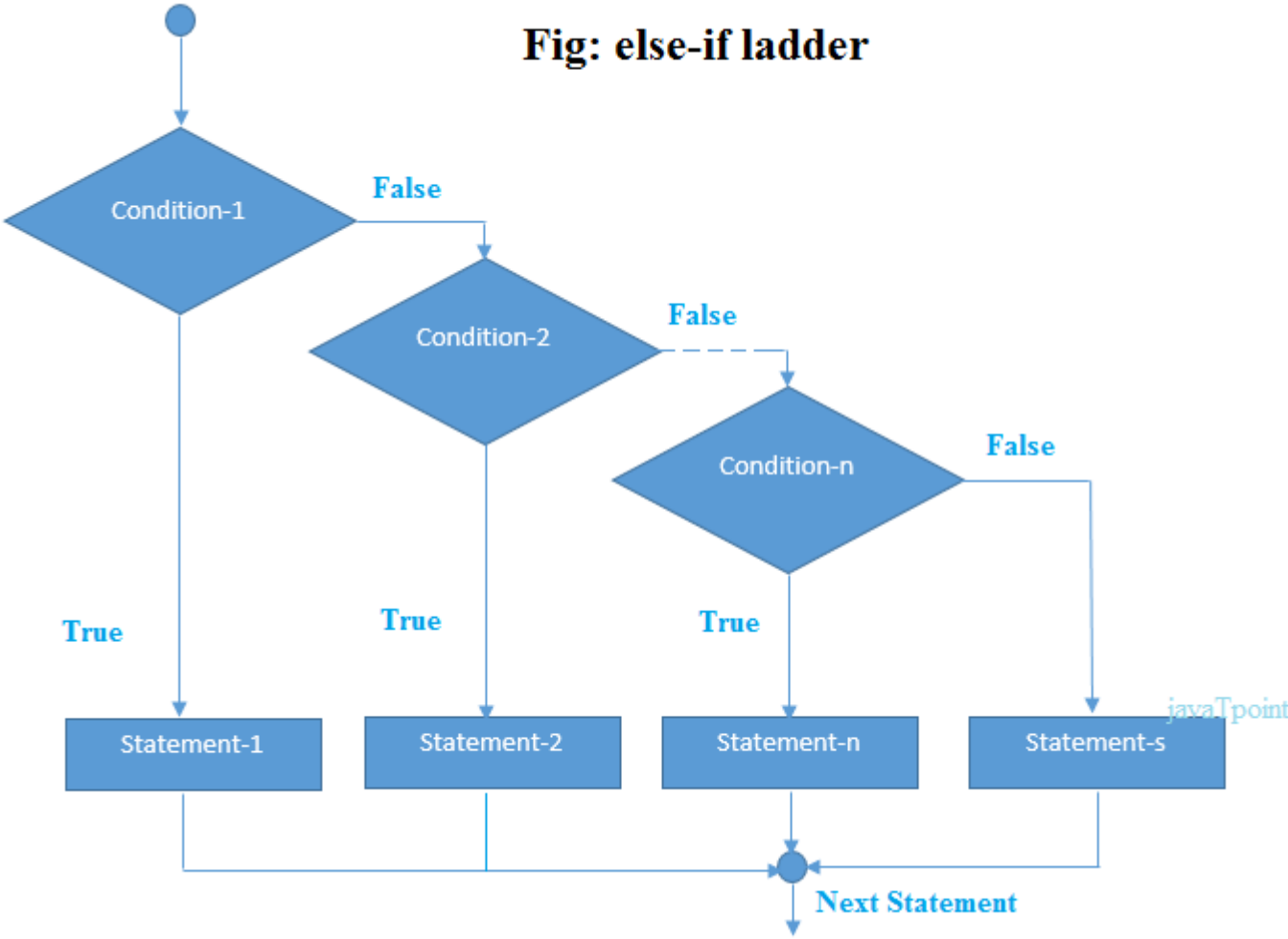
If else-if ladder Statement

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.

```
1. if(condition1){
2. //code to be executed if condition1 is true
3. }else if(condition2){
4. //code to be executed if condition2 is true
5. }
6. else if(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. else{
```

```
11. //code to be executed if all the conditions are false
12. }
```

Flowchart of else-if ladder statement in C



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

```
1. #include<stdio.h>
2. int main(){
3. int number=0;
4. printf("enter a number:");
5. scanf("%d",&number);
6. if(number==10){
7. 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
```

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 ...");
    }
```

```
11.    else if (marks > 60 && marks <= 85)
12.    {
13.        printf("You scored grade B + ...");
14.    }
15.    else if (marks > 40 && marks <= 60)
16.    {
17.        printf("You scored grade B ...");
18.    }
19.    else if (marks > 30 && marks <= 40)
20.    {
21.        printf("You scored grade C ...");
22.    }
23.    else
24.    {
25.        printf("Sorry you are fail ...");
26.    }
27. }
```

Output

```
Enter your marks?10
Sorry you are fail ...
Enter your marks?40
You scored grade C ...
Enter your marks?90
Congrats ! you scored grade A ...
```

C Switch 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 possibles 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.

```
1. int x,y,z;
2. char a,b;
3. float f;
```

C Loops

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language. In this part of the tutorial, we are going to learn all the aspects of C loops.

Types of C Loops

There are three types of loops in C language

that is given below:

1. do while
2. while
3. for

do-while loop in C

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

The syntax of do-while loop in c language

is given below:

```
1. do{
2. //code to be executed
3. }while(condition);
```

Flowchart and Example of do-while loop

while loop in C

The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

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

```
1. while(condition){
2. //code to be executed
3. }
```

Flowchart and Example of while loop

for loop in C

The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.

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

```
1. for(initialization;condition;incr/decr){
2. //code to be executed
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 1

```
#include<stdio.h>
#include<stdlib.h>
void main ()
{
    char c;
    int choice,dummy;
    do{
    printf("\n1. Print Hello\n2. Print Javatpoint\n3. Exit\n");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1 :
        printf("Hello");
        break;
        case 2:
        printf("Javatpoint");
        break;
        case 3:
        exit(0);
        break;
        default:
        printf("please enter valid choice");
    }
    printf("do you want to enter more?");
```

```
25.     scanf("%d",&dummy);
26.     scanf("%c",&c);
27.     }while(c=='y');
28. }
```

Output

```
1. Print Hello
2. Print Javatpoint
3. Exit
1
Hello
do you want to enter more?
y

1. Print Hello
2. Print Javatpoint
3. Exit
2
Javatpoint
do you want to enter more?
n
```

Flowchart of do while loop

do while example

There is given the simple program of c language do while loop where we are printing the table of 1.

```
1. #include<stdio.h>
2. int main(){
```

```
3. int i=1;
4. do{
5. printf("%d \n",i);
6. i++;
7. }while(i<=10);
8. return 0;
9. }
```

Output

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

Program to print table for the given number using do while loop

```
1. #include<stdio.h>
2. int main(){
3. int i=1,number=0;
4. printf("Enter a number: ");
5. scanf("%d",&number);
6. do{
7. printf("%d \n",(number*i));
8. i++;
```

```
9.  }while(i<=10);
10. return 0;
11. }
```

Output

```
Enter a number: 5
5
10
15
20
25
30
35
40
45
50
Enter a number: 10
10
20
30
40
50
60
70
80
90
100
```

Infinitive do while loop

The do-while loop will run infinite times if we pass any non-zero value as the conditional expression.

```
1. do{
2. //statement
3. }while(1);
```

while loop in C

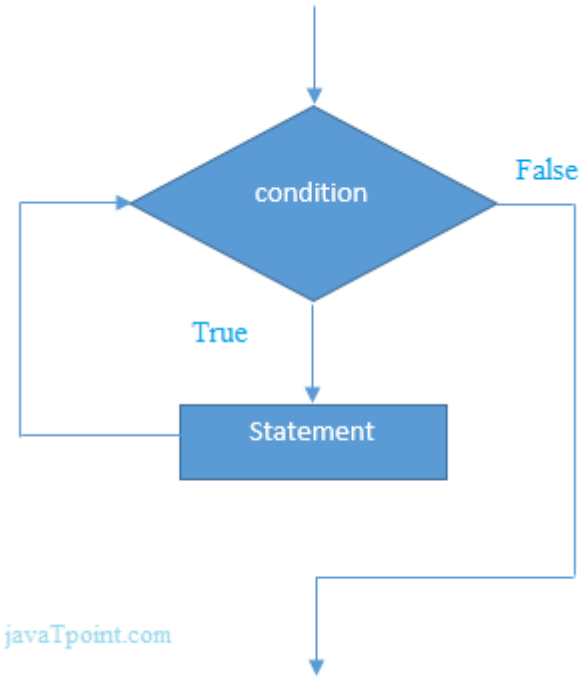
While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known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

Syntax of while loop in C language

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

```
1. while(condition){
2. //code to be executed
3. }
```

Flowchart of while loop in C



Example of the while loop in C language

Let's see the simple program of while loop that prints table of 1.

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

Output

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

Program to print table for the given number using while loop in C

```
#include<stdio.h>
int main(){
int i=1,number=0,b=9;
printf("Enter a number: ");
```

```
5. scanf("%d",&number);
6. while(i<=10){
7. printf("%d \n",(number*i));
8. i++;
9. }
10. return 0;
11. }
```

Output

```
Enter a number: 50
50
100
150
200
250
300
350
400
450
500
Enter a number: 100
100
200
300
400
500
600
700
800
900
1000
```

Properties of while loop

- A conditional expression is used to check the condition. The statements defined inside the while loop will repeatedly execute until the given condition fails.
- The condition will be true if it returns 0. The condition will be false if it returns any non-zero number.
- In while loop, the condition expression is compulsory.
- Running a while loop without a body is possible.
- We can have more than one conditional expression in while loop.
- If the loop body contains only one statement, then the braces are optional.

for loop in C

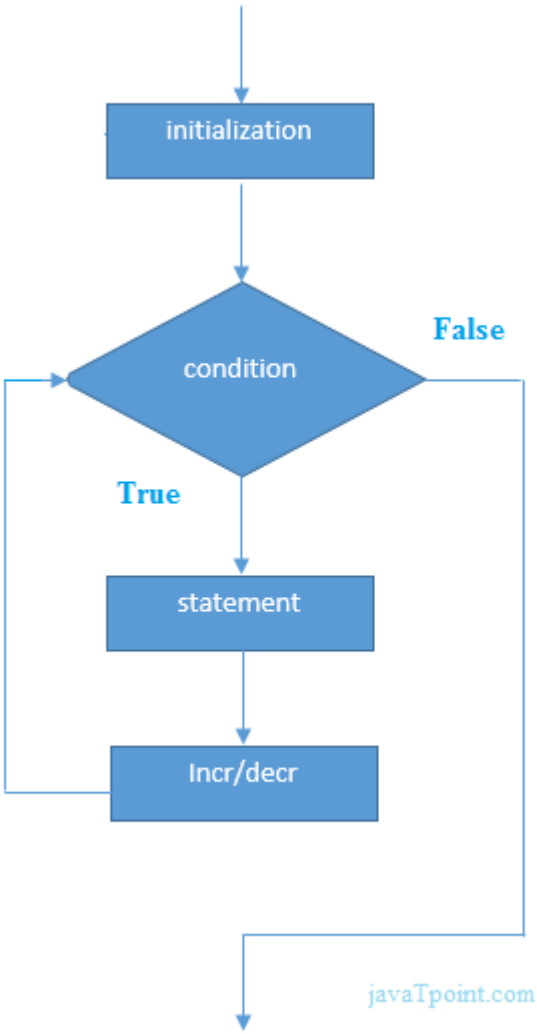
The **for loop in C language** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

Syntax of for loop in C

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

```
for(Expression 1; Expression 2; Expression 3){
//code to be executed
}
```

Flowchart of for loop in C



C for loop Examples

Let's see the simple program of for loop that prints table of 1.

```
1. #include<stdio.h>
2. int main(){
3. int i=0;
4. for(i=1;i<=10;i++){
5. printf("%d \n",i);
6. }
7. return 0;
8. }
```

Output

Play Videox

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

C Program: Print table for the given number using C for loop

```
1. #include<stdio.h>
2. int main(){
3. int i=1,number=0;
4. printf("Enter a number: ");
5. scanf("%d",&number);
6. for(i=1;i<=10;i++){
7. printf("%d \n",(number*i));
8. }
9. return 0;
10. }
```

C Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say

that the collection of functions creates a program. The function is also known as *procedure*or *subroutine*in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

-

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body

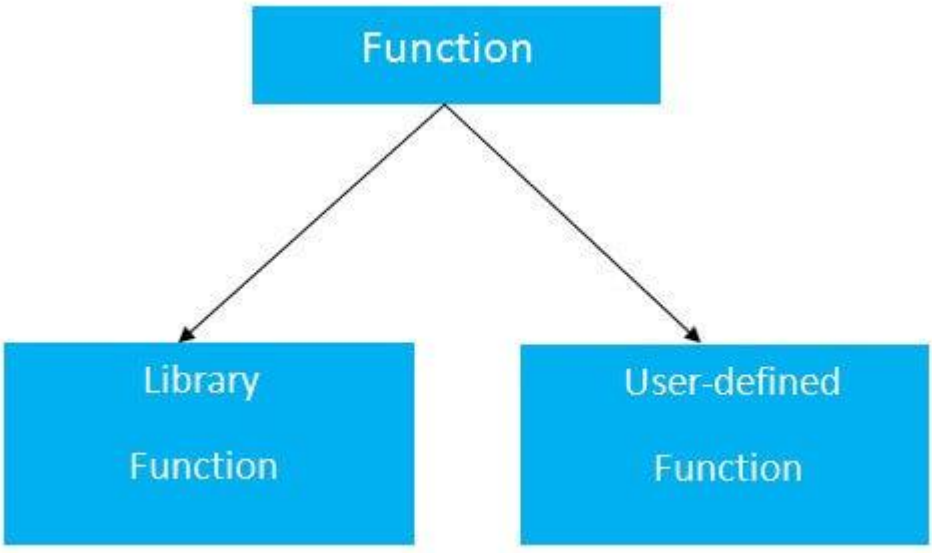
The syntax of creating function in c language is given below

```
return_type function_name(data_type parameter...)
{
  //code to be executed
}
```

Types of Functions

There are two types of functions in C programming:

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

```
1. void hello(){
2. printf("hello c");
3. }
```

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

```
1. int get(){
```

```
2. return 10;
3. }
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```
1. float get(){
2. return 10.2;
3. }
```

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value

```
1. void printName()
2. {
3.    printf("Javatpoint");
4. }
```

Output

```
Hello Javatpoint
```

Example 2

```
1. #include<stdio.h>
2. void sum();
3. void main()
4. {
5.     printf("\nGoing to calculate the sum of two numbers:");
6.     sum();
7. }
8. void sum()
9. {
10.     int a,b;
11.     printf("\nEnter two numbers");
12.     scanf("%d %d",&a,&b);
13.     printf("The sum is %d",a+b);
14. }
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example for Function without argument and with return value

Example 1

```
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();
    printf("%d",result);
}
int sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the area of the square

```c
#include<stdio.h>
int square();
void main()
{
    printf("Going to calculate the area of the square\n");
    float area = square();
    printf("The area of the square: %f\n",area);
}
int square()
{
    float side;
    printf("Enter the length of the side in meters: ");
    scanf("%f",&side);
    return side * side;
}
```

Output

```
Going to calculate the area of the square
Enter the length of the side in meters: 10
The area of the square: 100.000000
```

Example for Function with argument and without return value

Example 1

```
#include<stdio.h>
void sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    sum(a,b);
}
void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.

- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
1. #include<stdio.h>
2. void change(int num) {
3.     printf("Before adding value inside function num=%d \n",num);
4.     num=num+100;
5.     printf("After adding value inside function num=%d \n", num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(x);//passing value in function
11.    printf("After function call x=%d \n", x);
12. return 0;
13. }
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```c
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
    swap(a,b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
}
void swap (int a, int b)
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.

- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

Stay

```
1. #include<stdio.h>
2. void change(int *num) {
3.     printf("Before adding value inside function num=%d \n",*num);
4.     (*num) += 100;
5.     printf("After adding value inside function num=%d \n", *num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(&x);//passing reference in function
11.    printf("After function call x=%d \n", x);
12. return 0;
13. }
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
```

```
After function call x=200
```

Call by reference Example: Swapping the values of the two variables

```
1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // p
   rinting the value of a and b in main
8.     swap(&a,&b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The v
   alues of actual parameters do change in call by reference, a = 10, b = 20
10. }
11. void swap (int *a, int *b)
12. {
13.     int temp;
14.     temp = *a;
15.     *a=*b;
16.     *b=temp;
17.     printf("After swapping values in function a = %d, b = %d\n",*a,*b); //
   Formal parameters, a = 20, b = 10
18. }
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed i function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function outside of the function also. The va the actual parameters do change by c the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are cr the same memory location

Storage Classes in C

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, value between multiple functions ca
register	Register	Garbage Value	Local	Within the function

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.

  The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Example 1

```
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c); // printing initial default value of automatic var iables a, b, and c.
return 0;
}
```

Output:

```
garbage garbage garbage
```

Example 2

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

```
4. int a = 10,i;
5. printf("%d ",++a);
6. {
7. int a = 20;
8. for (i=0;i<3;i++)
9. {
10. printf("%d ",a); // 20 will be printed 3 times since it is the local value of a

11. }
12. }
13. printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
14. }
```

Output:

```
11 20 20 20 11
```

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

```
1. #include<stdio.h>
2. static char c;
3. static int i;
```

```
4. static float f;
5. static char s[100];
6. void main ()
7. {
8. printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will b
   e printed.
9. }
```

Output:


```
0 0 0.000000 (null)
```

Example 2

```
1. #include<stdio.h>
2. void sum()
3. {
4. static int a = 10;
5. static int b = 24;
6. printf("%d %d \n",a,b);
7. a++;
8. b++;
9. }
10. void main()
11. {
12. int i;
13. for(i = 0; i< 3; i++)
14. {
15. sum(); // The static variables holds their value between multiple function
    calls.
16. }
17. }
```

Output:

```
10 24
11 25
12 26
```

C Array

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

Advantage of C 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.

Declaration of C 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.

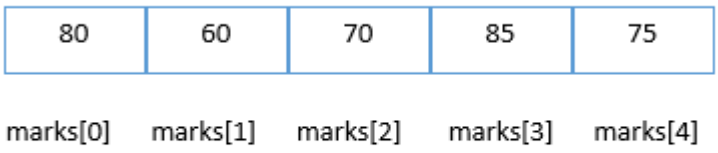
```
int marks[5];
```

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

Initialization of C 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.

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



Initialization of Array

C 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;
```

```
10. //traversal of array
11. for(i=0;i<5;i++){
12. printf("%d \n",marks[i]);
13. }//end of for loop
14. return 0;
15. }
```

Output

```
80
60
70
85
75
```

C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

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

In such case, there is **no requirement to define the size**. So it may also be written as the following code.

```
#include<stdio.h>
1. void main ()
2. {
3.     int i, j,temp;
4.     int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
5.     for(i = 0; i<10; i++)
6.     {
7.         for(j = i+1; j<10; j++)
8.         {
```

```
            if(a[j] > a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n",a[i]);
    }
}
```

Program to print the largest and second largest element of the array.

```
#include<stdio.h>
void main ()
{
    int arr[100],i,n,largest,sec_largest;
    printf("Enter the size of the array?");
    scanf("%d",&n);
    printf("Enter the elements of the array?");
    for(i = 0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0;i<n;i++)
    {
```

```
16.         if(arr[i]>largest)
17.         {
18.             sec_largest = largest;
19.             largest = arr[i];
20.         }
21.         else if (arr[i]>sec_largest && arr[i]!=largest)
22.         {
23.             sec_largest=arr[i];
24.         }
25.     }
26.     printf("largest = %d, second largest = %d",largest,sec_largest);
27.
28. }
```

Two Dimensional Array in C

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. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of two dimensional Array in C

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

```
1. data_type array_name[rows][columns];
```

Consider the following example.

```
1. int twodimen[4][3];
```

Here, 4 is the number of rows, and 3 is the number of columns.

Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

```
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

Two-dimensional array example in C

```
#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;
}
```

C 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 array 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.

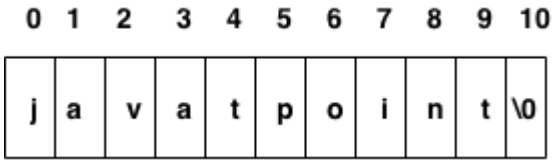
1. By char array

2. By string literal

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

1. **char** ch[10]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

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



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

1. **char** ch[]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

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

1. **char** ch[]="javatpoint";

In such 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.

- We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
- The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

String Example in C

Let's see a simple example where a string is declared and being printed. The '%s' is used as a format specifier for the string in c language.

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   char ch2[11]="javatpoint";

   printf("Char Array Value is: %s\n", ch);
   printf("String Literal Value is: %s\n", ch2);
 return 0;
}
```

Output

```
Char Array Value is: javatpoint
String Literal Value is: javatpoint
```

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] = "javatpoint";
    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 4
```

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] = "javatpoint";
    int i = 0;
```

```
6.      int count = 0;
7.      while(s[i] != NULL)
8.      {
9.          if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
10.         {
11.             count ++;
12.         }
13.         i++;
14.     }
15.     printf("The number of vowels %d",count);
16. }
17.
```

C 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.

C 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.

Declaration

```
1. char[] gets(char[]);
```

Reading string using gets()

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

```
void main ()
{
   char s[30];
   printf("Enter the string? ");
   gets(s);
   printf("You entered %s",s);
}
```

Output

```
Enter the string?
javatpoint is the best
You entered javatpoint is the best
```

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read. Consider the following example.

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

Output

```
Enter the string? javatpoint is the best website
javatpoint is the b
```

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. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

Declaration

1. **Int-** puts(**char**[])

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

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4. char name[50];
5. printf("Enter your name: ");
6. gets(name); //reads string from user
7. printf("Your name is: ");
8. puts(name);  //displays string
9. return 0;
10. }
```

Output:

```
Enter your name: Sonoo Jaiswal
Your name is: Sonoo Jaiswal
```

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.

C String Functions

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

C String Length: strlen() function

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

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4. char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
5.    printf("Length of string is: %d",strlen(ch));
6.  return 0;
7. }
```

Output:

```
Length of string is: 10
```

C 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]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   char ch2[20];
   strcpy(ch2,ch);
   printf("Value of second string is: %s",ch2);
 return 0;
}
```

Output:

```
Value of second string is: javatpoint
```

C 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
```

C 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: hello
Enter 2nd string: hello
Strings are equal
```

C 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: ");
```

```
6.    gets(str);//reads string from console
7.    printf("String is: %s",str);
8.    printf("\nReverse String is: %s",strrev(str));
9.   return 0;
10. }
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Reverse String is: tnioptavaj
```

C String Lowercase: strlwr()

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
1.  #include<stdio.h>
2.  #include <string.h>
3.  int main(){
4.    char str[20];
5.    printf("Enter string: ");
6.    gets(str);//reads string from console
7.    printf("String is: %s",str);
8.    printf("\nLower String is: %s",strlwr(str));
9.   return 0;
10. }
```

Output:

```
Enter string: JAVATpoint
String is: JAVATpoint
Lower String is: javatpoint
```

C 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: javatpoint
String is: javatpoint
Upper String is: JAVATPOINT
```

C String strstr()

The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax:

```
char *strstr(const char *string, const char *match)
```

String strstr() parameters

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

String strstr() example

```c
#include<stdio.h>
#include <string.h>
int main(){
  char str[100]="this is javatpoint with c and java";
  char *sub;
  sub=strstr(str,"java");
  printf("\nSubstring is: %s",sub);
 return 0;
}
```

Output:

```
javatpoint with c and java
```

What is Structure

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures ca; simulate the use of classes and templates as it can store various information

The **,struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

```c
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeberN;
```

```
};
```

Let's see the example to define a structure for an entity employee in c.

```
struct employee
{   int id;
    char name[20];
    float salary;
};
```

The following image shows the memory allocation of the structure employee that is defined in the above example.Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:

Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{   int id;
    char name[50];
    float salary;
};
```

Now write given code inside the main() function.

```
1. struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++

and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
1. struct employee
2. {   int id;
3.     char name[50];
4.     float salary;
5. }e1,e2;
```

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by. (member) operator.

```
1. p1.id
```

C Structure example

Let's see a simple example of structure in C language.

```c
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
```

Let's see another example of the structure in C language

to store many employees information.

```c
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;  //declaring e1 and e2 variables for structure
int main( )
```

```
{
  //store first employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  e1.salary=56000;

 //store second employee information
  e2.id=102;
  strcpy(e2.name, "James Bond");
  e2.salary=126000;

  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  printf( "employee 1 salary : %f\n", e1.salary);

  //printing second employee information
  printf( "employee 2 id : %d\n", e2.id);
  printf( "employee 2 name : %s\n", e2.name);
  printf( "employee 2 salary : %f\n", e2.salary);
  return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

C Array of Structures

Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
1. #include<stdio.h>
2. struct student
3. {
4.     char name[20];
5.     int id;
6.     float marks;
7. };
8. void main()
9. {
10.     struct student s1,s2,s3;
11.     int dummy;
12.     printf("Enter the name, id, and marks of student 1 ");
13.     scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
14.     scanf("%c",&dummy);
15.     printf("Enter the name, id, and marks of student 2 ");
16.     scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
17.     scanf("%c",&dummy);
18.     printf("Enter the name, id, and marks of student 3 ");
19.     scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
20.     scanf("%c",&dummy);
21.     printf("Printing the details....\n");
22.     printf("%s %d %f\n",s1.name,s1.id,s1.marks);
```

```
23.     printf("%s %d %f\n",s2.name,s2.id,s2.marks);
24.     printf("%s %d %f\n",s3.name,s3.id,s3.marks);
25. }
```

Output

```
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adoms 90 90
Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adoms 90 90.000000
Nick 90 90.000000
```

Introduction on Constructor in C

A Constructor in C is used in the memory management of C++programming.
It allows built-in data types like int, float and user-defined data types such as class. Constructor in Object-oriented programming initializes the variable of a user-defined data type. Constructor helps in the creation of an object. The name of the constructor is the same as the name of the object but it has no return type. A Constructor is executed automatically when an object or special member is created. It allocates the memory for the new object created and it can be overloaded.

example :

```
// class with Constructor
class integer
{
int a, b;
```

```
public:
integer (void);
// declaration of Constructor
};
integer :: integer (void)
// constructor defined
{
a = 0, b = 0;
}
```

Uses of the Constructor

Below are some uses of the constructor.

1. It is a special method that holds the same name as the class name and initializes the

object whenever it is created. So it is simple and easy to execute.

2. It is mainly used for memory management. They are used to initialize and remove memory

block when it is no longer required by having New and Delete options as specified by

the programmer

3. The compiler creates a default constructor whenever the object is created. When you

didn’t declare the constructor the compiler would create a one. It is useful because

the object and function in the program knows that the object exists

4. A constructor for an object is created when an instance is an object that is declared. A class can have multiple constructors for different situations. Constructor overloading increases the versatility of the class by having many constructors in an individual class.

TYPES OF CONSTRUCTOR

1. Default Constructor

A default constructor has no parameter or the present parameter has default values.

If no user-defined constructor is present in class the compiler creates a new one

if needed and that is called as default constructor. This is an inline public member

of the class.

2. Parameterized Constructors

The constructor that can accept the arguments is called Parameterized constructors.

It can specify the argument whenever it is needed.

3. Copy Constructor

It is used to initialize and declare one object from another object

DESTRUCTOR

What is a destructor?

Destructor is an instance member function which is invoked automatically whenever an

object is going to be destroyed. Meaning, a destructor is the last function that is

going to be called before an object is destroyed.

Destructor is also a special member function like constructor. Destructor destroys

the class objects created by constructor.

Destructor has the same name as their class name preceded by a tiled (~) symbol.

It is not possible to define more than one destructor.

The destructor is only one way to destroy the object create by constructor.

Hence destructor can-not be overloaded.

Destructor neither requires any argument nor returns any value.

It is automatically called when object goes out of scope.

Destructor release memory space occupied by the objects created by constructor.

In destructor, objects are destroyed in the reverse of an object creation.

Object Oriented Programming

1. *Class*
2. *Objects*
3. *Encapsulation*
4. *Abstraction*
5. *Polymorphism*
6. *Inheritance*
7. *Dynamic Binding*
8. *Message Passing*

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage

range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviours.

Object: An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
class person
{
   char name[20];
   int id;
public:
   void getdetails(){}
};

int main()
{
  person p1; // p1 is a object
}
```

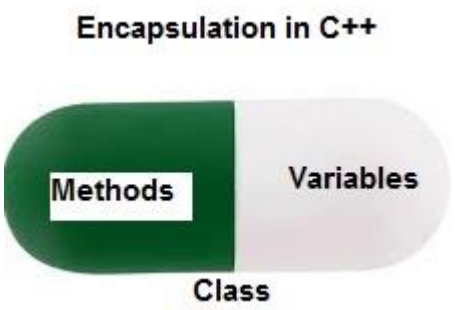
Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.



Encapsulation in C++

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes*: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- *Abstraction in Header files*: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Polymorphism: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

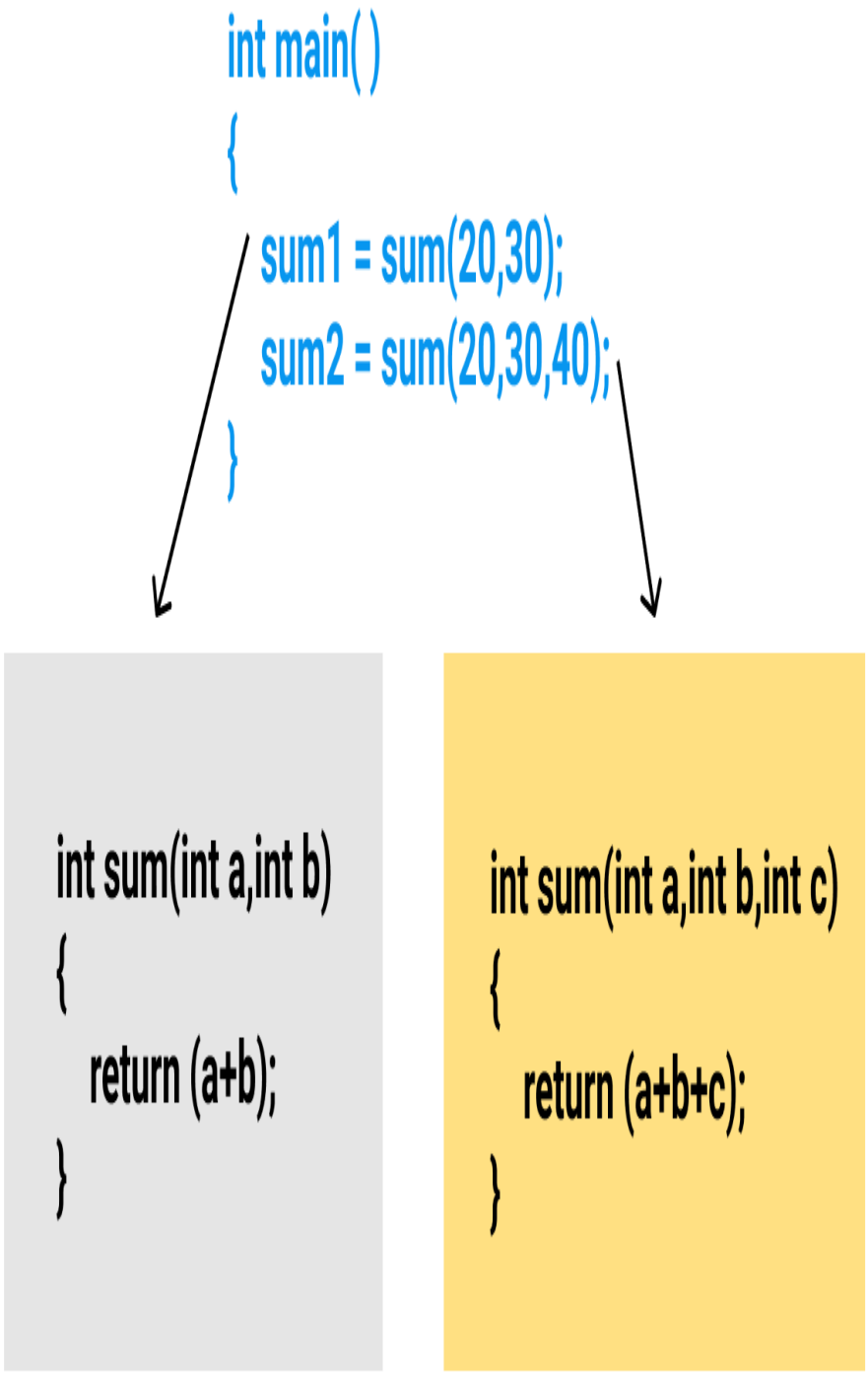
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks.
  Polymorphism is extensively used in implementing inheritance.

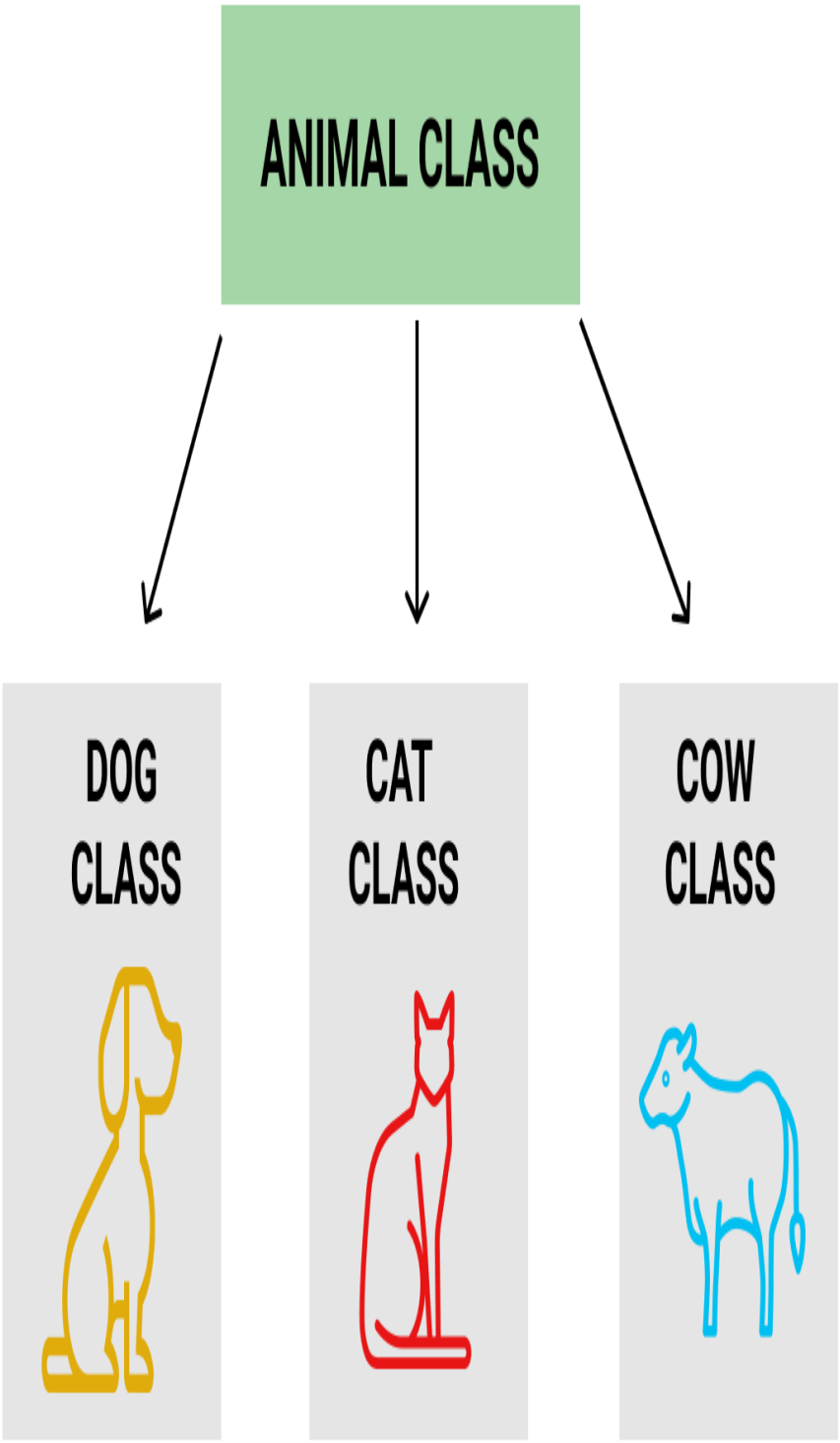
Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.


int main( )
{
sum1 = sum(20,30);
sum2 = sum(20,30,40);
}
int sum(int a,int b)
{
return (a+b);
}
int sum(int a,int b,int c)
{
return (a+b+c);
}

Inheritance: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**:The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Function Overloading in C++

1. Let's consider a situation: Say you wanted to create a program that would return the area of the rectangle. Say if I provided length(say x) and breadth(say y) you would return x*y. You would probably write this code:

```
#include <iostream>int area(int x, int y) {  return x * y;}
int main() {  int areaOfRect = area(5, 2);  std::cout << "Area of
rectangle with length 5 and breadth 2 is " << areaOfRect << std::endl;
return 0;}
```

1. So far so good. Until your supervisor tells you that people also want to calculate the area of a square using the same function. Now, you want to calculate the area based on how many arguments are passed to the function. Say if I was provided just x (just 1 argument) you would assume that I'm calculating the area of a square and return x*x. But if I was providing x and y (so 2 argument), you would return x*y. You would write the code something similar to this:

```
#include <iostream>int area(int x) {  return x * x;}
int area(int x, int y) {  return x * y;}
int main() {  int areaOfSquare = area(5);  int areaOfRect = area(5, 2);
std::cout << "Area of square with side length 5 is " << areaOfSquare <<
std::endl;  std::cout << "Area of rectangle with length 5 and breadth 2 is
" << areaOfRect << std::endl;  return 0;}
```

Function overriding in C++

Say instead of writing functions, I wanted to instead define an OOP based solution. First, we need to have a mapping of how we want to design these structures.

One solution is : say I have a Rectangle class. Since we know that a Square is a Rectangle, we can create a subclass Square off of class Rectangle.

```
#include <iostream>class Rectangle {  private:      int length, breadth;
public:      Rectangle() {        length = 0, breadth = 0;      }
     Rectangle(int l, int b) {        length = l;
       breadth = b;
     }
     int area() {        return length * breadth;      }
     void helloFromRectangle() {        std::cout << "Hello from Rectangle"
<< std::endl;      }
};
class Square: public Rectangle {  private:      int side;  public:      Square()
{        side = 0;      }
     Square(int s) {        side = s;
     }
     int area() {        return side * side;      }
};
int main() {  Rectangle rect(5, 2);  Square square(5);  std::cout << "Area
of rectangle is " << rect.area() << std::endl;  std::cout << "Area of square
is " << square.area() << std::endl;  // child class inherits all functions as is
except the overridden function  rect.helloFromRectangle();
square.helloFromRectangle();  return 0;}
```

Output:

```
Area of rectangle is 10Area of square is 25Hello from Rectangle
Hello from Rectangle
```

We know that child classes inherit all public data and methods from the parent class. So the Square class also inherits the area function. However, we redefine the area function to be side * side. This redefinition of the area function is an example of function overriding. Basically, C++ knows that you specifically redefined the area function for the Square class. This overriding of function is a type of runtime polymorphism.

C++ distinguishes the function call at runtime instead of knowing it during compilation because the function signature is similar at compile time.

Function Overloading vs Function Overriding in C++

As we've seen before, function overloading and function overriding play a similar role. They both provide an abstraction over the interface so that the end user doesn't have to think much about the context and simply pass in the arguments. However, there are subtle differences in the two approaches.

Function Overloading	Function Overriding
Function overloading can be used in normal functions as well as in classes (for eg: constructor overloading is a classic example where you would vary the number/type of arguments for different initialisations)	Function overriding is applicable exclusively to an inherited class (or in other words a subclass)
Function overloading is resolved at compile time	Function overriding is resolved at run time.
Overloaded functions are in same scope	Overridden functions are in different scopes
Overloaded functions have different function signatures	Overridden functions have same function signatures

Different Types of Inheritance

OOPs support the six different types of inheritance as given below :

1.Single inheritance

2.Multi-level inheritance

3.Multiple inheritance

4.Multipath inheritance

5.Hierarchical Inheritance

6.Hybrid Inheritance

Single inheritance

In this inheritance, a derived class is created from a single base class.

In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behavior of the parent class A.

The syntax for Single Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
```

```
 }
}
```

Multi-level inheritance

In this inheritance, a derived class is created from another derived class.

In the given example, class c inherits the properties and behavior of class B and class B inherits the properties and behavior of class B. So, here A is the parent class of B and class B is the parent class of C. So, here class C implicitly inherits the properties and behavior of class A along with Class B i.e there is a multilevel of inheritance.

The syntax for Multi-level Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}
```

```
//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : B
{
 public void fooC()
 {
 //TO DO:
 }
}
```

Multiple inheritance

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F#, etc., and Java Language.

In the given example, class c inherits the properties and behavior of class B and class A at the same level. So, here A and Class B both are the parent classes for Class C.

The syntax for Multiple Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Base Class
class B
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A, B
```

```
{
 public void fooC()
 {
 //TO DO:
 }
}
```

Multipath inheritance

In this inheritance, a derived class is created from other derived classes and the same base class of other derived classes. This inheritance is not supported by .NET Languages like C#, F#, etc.

In the given example, class D inherits the properties and behavior of class C and class B as well as Class A. Both class C and class B inherit the Class A. So, Class A is the parent for Class B and Class C as well as Class D. So it's making it a Multipath inheritance.

The syntax for Multipath Inheritance

```
//Base Class
class A
{
 public void fooA()
```

```
 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
```

```
class D : B, A, C
{
 public void fooD()
 {
 //TO DO:
 }
}
```

Hierarchical Inheritance

In this inheritance, more than one derived class is created from a single base class and further child classes act as parent classes for more than one child class.

In the given example, class A has two children class B and class D. Further, class B and class C both are having two children - class D and E; class F and G respectively.

The syntax for Hierarchical Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
```

```
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : C
```

```
{
 public void fooD()
 {
 //TO DO:
 }
}

//Derived Class
class E : C
{
 public void fooE()
 {
 //TO DO:
 }
}

//Derived Class
class F : B
{
 public void fooF()
 {
 //TO DO:
 }
}
```

```
//Derived Class
class G :B
{
 public void fooG()
 {
 //TO DO:
 }
}
```

Hybrid inheritance

This is a combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance Hierarchical and Multipath inheritance, or Hierarchical, Multilevel and Multiple inheritances.

Since .NET Languages like C#, F#, etc. do not support multiple and multipath inheritance. Hence hybrid inheritance with a combination of multiple or multipath inheritances is not supported by .NET Languages.

The syntax for Hybrid Inheritance

```
//Base Class
```

```
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Base Class
class F
{
 public void fooF()
 {
 //TO DO:
 }
}

//Derived Class
class B : A, F
{
 public void fooB()
 {
 //TO DO:
 }
```

```
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : C
{
 public void fooD()
 {
 //TO DO:
 }
}

//Derived Class
class E : C
{
 public void fooE()
```

```
{
//TO DO:
}
}
```

Advantages of Inheritance

1.Reduce code redundancy.

2.Provides better code reusabilities.

3.Reduces source code size and improves code readability.

4.The code is easy to manage and divided into parent and child classes.

5.Supports code extensibility by overriding the base class functionality within child classes.

6.Code usability will enhance the reliability eventually where the base class code will always be tested and debugged against the issues.

Disadvantages of Inheritance

1.In Inheritance base class and child class, both are tightly coupled. Hence If you change the code of the parent class, it will affect all the child classes.

2.In a class hierarchy, many data members remain unused and the memory allocated to them is not utilized. Hence it affects the performance of your program if you have not implemented inheritance correctly.

3.Inheritance increases the coupling between the base class and the derived class. any small change in the base class will directly affect all the child classes which are extended to the parent class.