CONTENTS

What is C language 6

1) C as a mother language 6

2) C as a system programming language 7

3) C as a procedural language 7

4) C as a structured programming language 7

5) C as a mid-level programming language 7

C Program 8

History of C Language 8

Features of C Language 9

Features of C Language 9

1) Simple 10

2) Machine Independent or Portable 10

3) Mid-level programming language 10

4) Structured programming language 10

5) Rich Library 10

6) Memory Management 11

7) Speed 11

8) Pointer 11

9) Recursion 11

10) Extensible 11

First C programe 12

What is Compilation? 13

Printf() and scanf() in C Language 14

Program to print cube of given number 14

Program to print sum of 2 numbers 15

Variables in c 16

Rules for defining variables 16

Types of Variables in C 17

Local Variable 17

Global Variable 17

Static Variable 18

Data types in C Language 18

Data Types in C 18

Basic Data Types 19

Keywords in C Language 21

C Language Identifier 22
Rules for constructing C identifiers 22
Types of identifiers 23
Differences between Keyword and Identifier 23
C Language Operators 24
Precedence of Operators in C 24
Comments in C Language 26
Single Line Comments 26
Multi-Line Comments 27
C Language Format Specifier 27
Constants in C language 29
List of Constants in C 29
2 ways to define constant in C 30
1) C const keyword 30
2) C #define preprocessor 31
What are literals? 31
Types of literals 31
Integer literal 32
Float literal 32
Character literal 32
String literal 32
C Language control statement 33
C if else Statement 33
If Statement 34
Program to find the largest number of the three. 36
If-else Statement 36
Program to check whether a person is eligible to vote or not. 38
If else-if ladder Statement 39
Program to calculate the grade of the student according to the specified marks. 41
C Switch Statement 42
Rules for switch statement in C language 43
Switch case example 2 46
Nested switch case statement 47
C Language loops 48
Why use loops in C language? 48
Advantage of loops in C 48
Types of C Loops 48

do-while loop in C 49
while loop in C 52
Example of the while loop in C language 53
Program to print table for the given number using while loop in C 54
Properties of while loop 54
for loop in C 56
Flowchart of for loop in C 56
C for loop Examples 57
C Program: Print table for the given number using C for loop 57
Nested Loops in C 60
Infinite Loop in C 65
What is infinite loop? 65
When to use an infinite loop 65
For loop 66
while loop 67
C Language break statement 69
Syntax: 69
Flowchart of break in c 69
Example 70
C continue statement 70
Syntax: 70
Continue statement example 2 71
C Goto Statement 71
goto example 72
Type casting in C language 73
Type Casting example 73
C language Array 74
Properties of Array 74
Advantage of C Array 74
Disadvantage of C Array 75
Declaration of C Array 75
Initialization of C Array 75
C array example 76
C Array- 76
Declaration with Initialization 76
Two Dimensional Array in C 77
Declaration of two dimensional Array in C 77

Initialization of 2D Array in C 78
Two-dimensional array example in C 78
Passing Array to Function in C 79
Methods to declare a function that receives an array as an argument 79
C language passing an array to function example 80
C function to sort the array 81
C LANGUAGE FUNCTIONS 83
C Functions 83
Advantage of functions in C 83
Function Aspects 83
Types of Functions 84
Return Value 85
Different aspects of function calling 86
Example for Function without argument and return value 86
C Pointers 88
Address in C 88
C Pointers 89
Pointer Syntax 89
Assigning addresses to Pointers 90
Get Value of Thing Pointed by Pointers 90
Changing Value Pointed by Pointers 91
Example: Working of Pointers 92
Common mistakes when working with pointers 94
Relationship Between Arrays and Pointers 95
Relationship Between Arrays and Pointers 96
Example 1: Pointers and Arrays 97
Example 2: Arrays and Pointers 98
C Pass Addresses and Pointers 99
Example: Pass Addresses to Functions 99
Example 2: Passing Pointers to Functions 101
Example: Largest Element in an array 101
C Programming Strings 103
How to declare a string? 104
How to initialize strings? 104
Assigning Values to Strings 105
Read String from the user 105
Example 1: scanf() to read a string 106

How to read a line of text? 106
Example 2: fgets() and puts() 107
Passing Strings to Functions 107
Example 3: Passing string to a Function 108
Strings and Pointers 108
Example 4: Strings and Pointers 109
Commonly Used String Functions 109
gets() and puts() 110
Program to count vowels, consonants, etc. 111

C LANGUAGE COURSE

WHAT IS C LANGUAGE

C language Tutorial with programming approach for beginners and professionals, helps you to understand the C language tutorial easily. Our C tutorial explains each topic with programs.

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

1) C AS A MOTHER LANGUAGE

C language is considered as the mother language of all the modern programming languages because **most of the compilers, JVMs, Kernels, etc. are written in C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

It provides the core concepts like the array, strings, functions, file handling, etc. that are being used in many languages like C++, Java, C#, etc.

2) C AS A SYSTEM PROGRAMMING LANGUAGE

A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

3) C AS A PROCEDURAL LANGUAGE

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

4) C AS A STRUCTURED PROGRAMMING LANGUAGE

A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

In the C language, we break the program into parts using functions. It makes the program easier to understand and modify.

5) C AS A MID-LEVEL PROGRAMMING LANGUAGE

C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

A **Low-level language** is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

A **High-Level language** is not specific to one machine, i.e., machine independent. It is easy to understand.

C PROGRAM

```
#include <stdio.h>
int main() {
printf("Hello C Programming\n");
return 0;
}
```

HISTORY OF C LANGUAGE

History of C language is interesting to know. Here we are going to discuss a brief history of the c language.

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Dennis Ritchie is known as the **founder of the c language**.

It was developed to overcome the problems of previous languages such as B, BCPL, etc.

initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

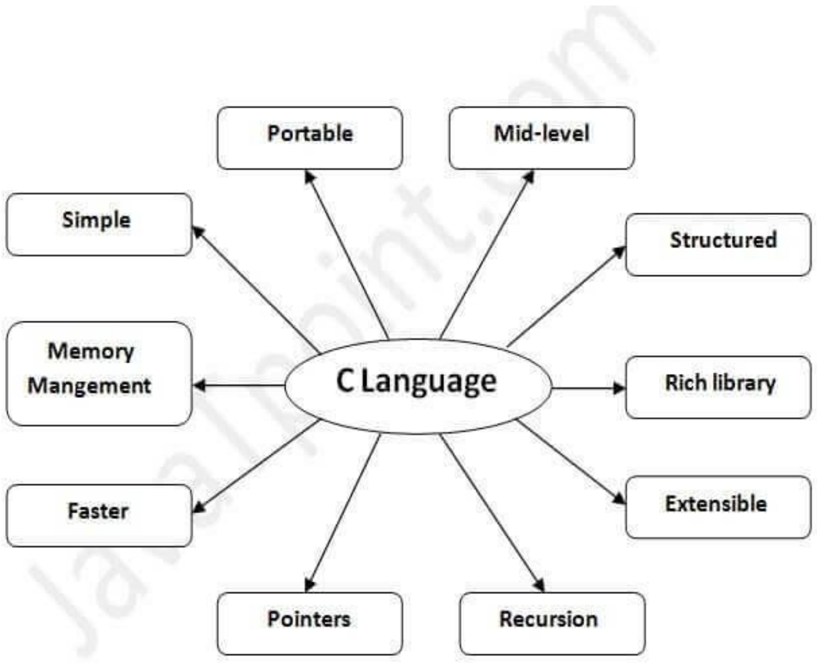
Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson

Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

FEATURES OF C LANGUAGE

FEATURES OF C LANGUAGE



C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library

6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

1) SIMPLE

C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc.

2) MACHINE INDEPENDENT OR PORTABLE

Unlike assembly language, c programs **can be executed on different machines** with some machine specific changes. Therefore, C is a machine independent language.

3) MID-LEVEL PROGRAMMING LANGUAGE

Although, C is **intended to do low-level programming**. It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.

4) STRUCTURED PROGRAMMING LANGUAGE

C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.

5) RICH LIBRARY

C **provides a lot of inbuilt functions** that make the development fast.

6) MEMORY MANAGEMENT

It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

7) SPEED

The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) POINTER

C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array**, etc.

9) RECURSION

In C, we **can call the function within the function**. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) EXTENSIBLE

C language is extensible because it **can easily adopt new features**.

FIRST C PROGRAME

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

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

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

int main() The **main() function is the entry point of every program** in c language.

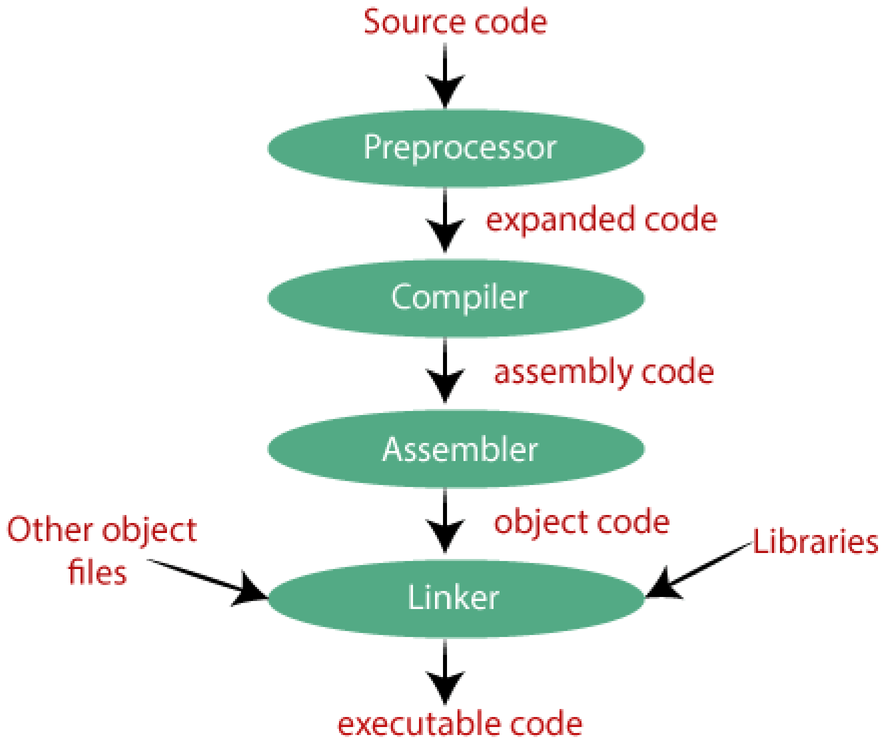
printf() The printf() function is **used to print data** on the console.

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.

WHAT IS COMPILATION?

The following are the phases through which our program passes before being transformed into an executable form:

- **Preprocessor**
- **Compiler**
- **Assembler**
- **Linker**



PRINTF() AND SCANF() IN C LANGUAGE

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.

PROGRAM TO PRINT SUM OF 2 NUMBERS

Let's see a simple example of input and output in C language that prints addition of 2 numbers.

```
#include<stdio.h>
int main(){
int x=0,y=0,result=0;

printf("enter first number:");
scanf("%d",&x);
printf("enter second number:");
scanf("%d",&y);

result=x+y;
printf("sum of 2 numbers:%d ",result);

return 0;
}
```

Output

```
enter first number:9
enter second number:9
sum of 2 numbers:18
```

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:

```
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
3. static 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
}
```

STATIC VARIABLE

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

It retains its value between multiple function calls.

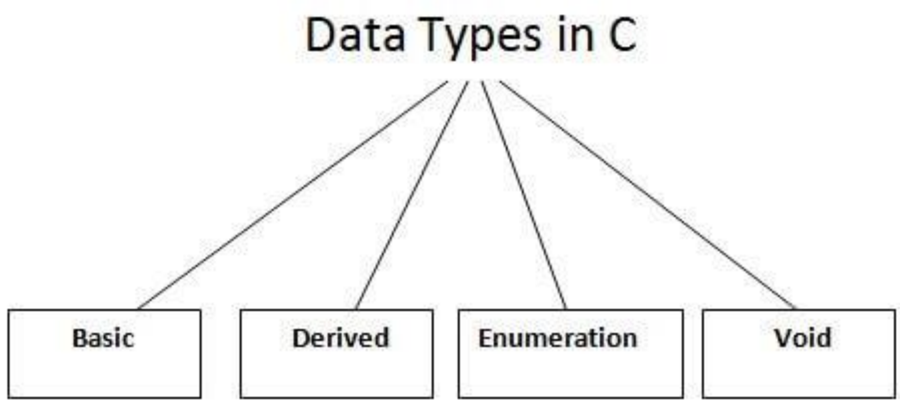
```
void function1(){
int x=10;//local variable
static int y=10;//static variable
x=x+1;
y=y+1;
printf("%d,%d",x,y);
}
```

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.

DATA TYPES IN C LANGUAGE

DATA TYPES IN C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

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 LANGUAGE

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

We will learn about all the C language keywords later.

C LANGUAGE IDENTIFIER

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. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumerical characters that represent the identifiers.

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

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

Example of invalid identifiers

```
2sum (starts with a numerical digit)
int (reserved word)
char (reserved word)
m+n (special character, i.e., '+')
```

TYPES OF IDENTIFIERS

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

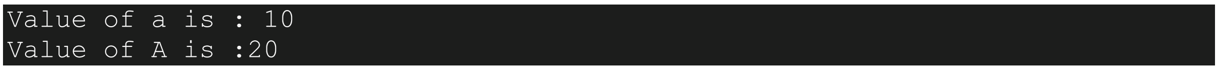
If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

DIFFERENCES BETWEEN KEYWORD AND IDENTIFIER

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Let's understand through an example.

Output



```
Value of a is : 10
Value of A is :20
```

The above output shows that the values of both the variables, 'a' and 'A' are different. Therefore, we conclude that the identifiers are case sensitive.

C LANGUAGE OPERATORS

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

There are following types of operators to perform different types of operations in C language.

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

PRECEDENCE OF OPERATORS IN C

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

Let's understand the precedence by the example given below:

```
int value=10+20*10;
```

The value variable will contain **210** because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C operators is given below:

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right

Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

COMMENTS IN C LANGUAGE

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

SINGLE LINE COMMENTS

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

```
#include<stdio.h>
int main(){
   //printing information
   printf("Hello C");
return 0;
}
```

Output:

```
Hello C
```

Even you can place the comment after the statement. For example:

```
printf("Hello C");//printing information
```

MULTI-LINE COMMENTS

Multi-Line comments are represented by slash asterisk * ... *\. It can occupy many lines of code, but it can't be nested. Syntax:

```
/*
code
to be commented
*/
```

Let's see an example of a multi-Line comment in C.

```
#include<stdio.h>
int main(){
    /*printing information
     Multi-Line Comment*/
    printf("Hello C");
return 0;
}
```

Output:

```
Hello C
```

C LANGUAGE 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 the variable can hold both positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 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 precision, i.e., the value after the decimal in input would be exactly the same as the value in 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.

CONSTANTS IN C LANGUAGE

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.

There are different types of constants in C programming.

LIST OF CONSTANTS IN C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in javatpoint" etc.

2 WAYS TO DEFINE CONSTANT IN C

There are two ways to define constant in C programming.

1. const keyword
2. #define preprocessor

1) C CONST KEYWORD

The const keyword is used to define constant in C programming.

```
const float PI=3.14;
```

Now, the value of PI variable can't be changed.

```
#include<stdio.h>
int main(){
    const float PI=3.14;
   printf("The value of PI is: %f",PI);
    return 0;
}
```

Output:

```
The value of PI is: 3.140000
```

If you try to change the the value of PI, it will render compile time error.

```
#include<stdio.h>
int main(){
const float PI=3.14;
PI=4.5;
printf("The value of PI is: %f",PI);
   return 0;
}
```

Output:

```
Compile Time Error: Cannot modify a const object
```

2) C #DEFINE PREPROCESSOR

The #define preprocessor is also used to define constant. We will learn about #define preprocessor directive later.

WHAT ARE LITERALS?

Literals are the constant values assigned to the constant variables. We can say that the literals represent the fixed values that cannot be modified. It also contains memory but does not have references as variables. For example, const int =10; is a constant integer expression in which 10 is an integer literal.

TYPES OF LITERALS

There are four types of literals that exist in C programming:

- **Integer literal**
- **Float literal**
- **Character literal**
- **String literal**

INTEGER LITERAL

It is a numeric literal that represents only integer type values.

FLOAT LITERAL

It is a literal that contains only floating-point values or real numbers.

CHARACTER LITERAL

A character literal contains a single character enclosed within single quotes. If multiple characters are assigned to the variable, then we need to create a character array. If we try to store more than one character in a variable, then the warning of a **multi-character character constant** will be generated.

STRING LITERAL

A string literal represents multiple characters enclosed within double-quotes. It contains an additional character, i.e., '\0' (null character), which gets automatically inserted. This null character specifies the termination of the string. We can use the '+' symbol to concatenate two strings.

For example,

String1= "Disha";

String2= "family";

To concatenate the above two strings, we use '+' operator, as shown in the below statement:

"Disha " + "family"= Disha family

DISHA.DEHUROAD2021@GMAIL.COM

MAJJINDUCK@GMAIL.COM

C LANGUAGE CONTROL STATEMENT

C 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.

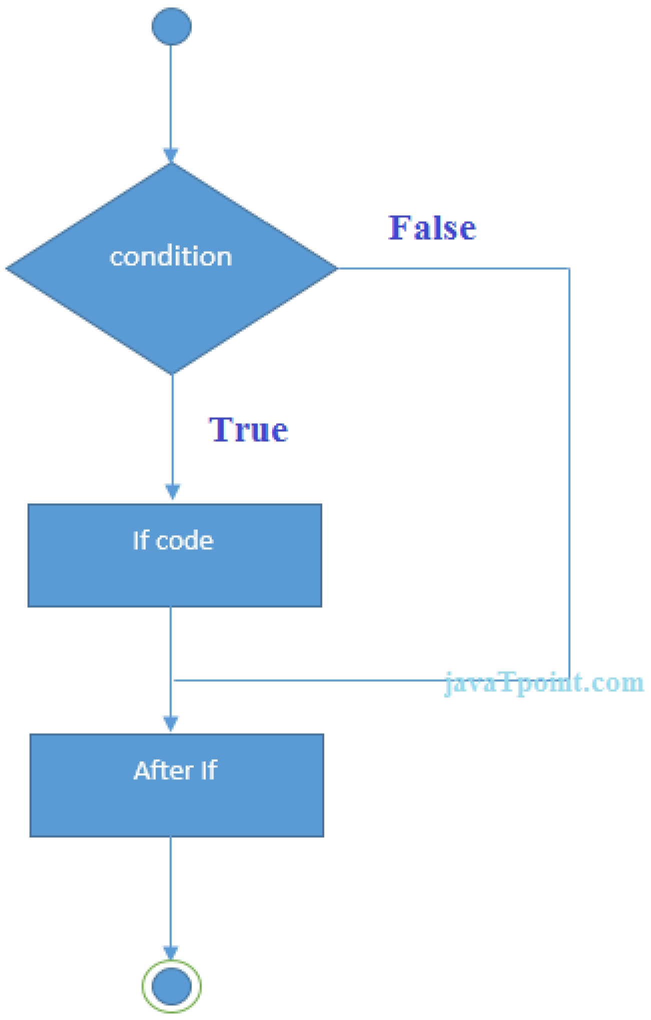
- If statement
- If-else statement
- If else-if ladder
- Nested if

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.

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

Flowchart of if statement in C



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

```
#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
```

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

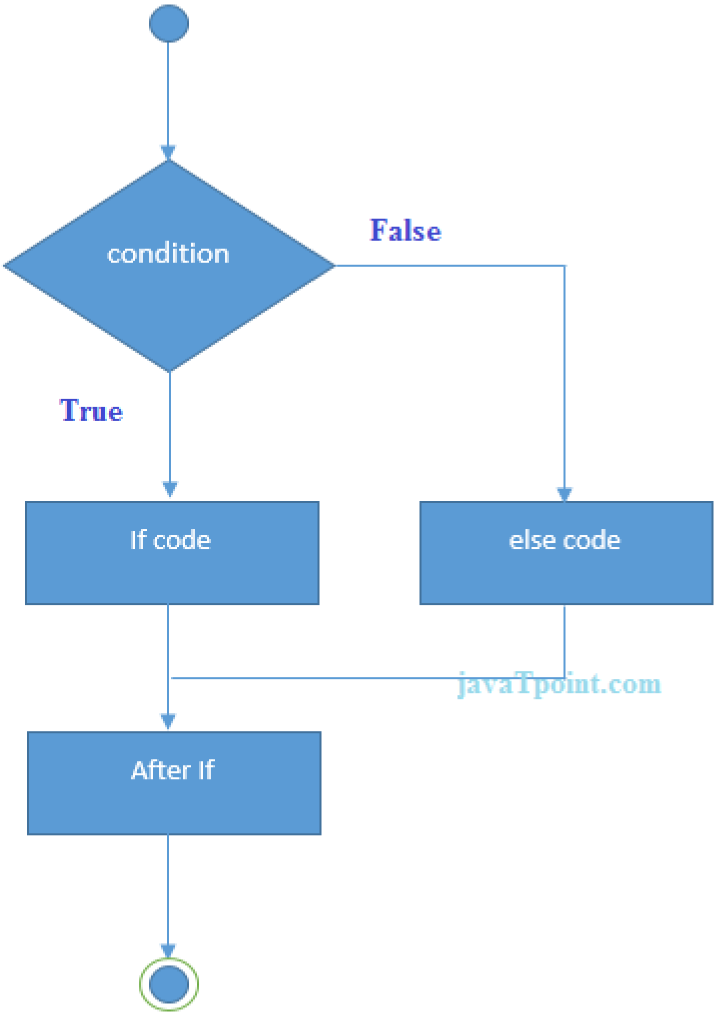
```
Enter three numbers?
12 23 34
34 is largest
```

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

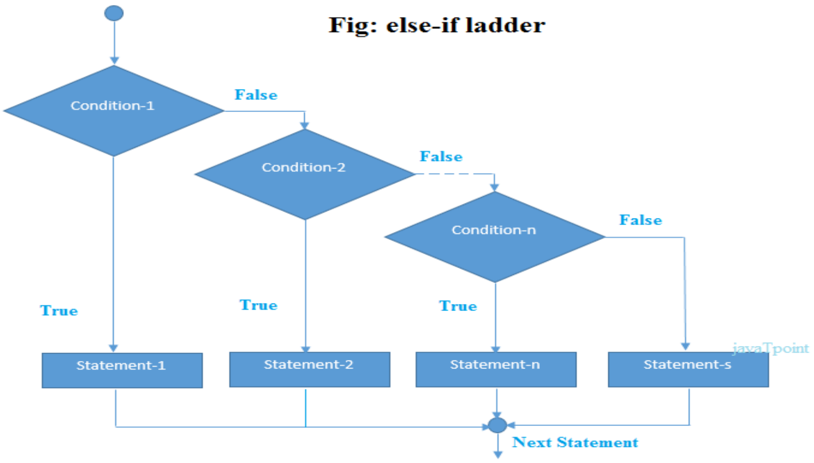


Fig: else-if ladder

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

```
#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;
}
```

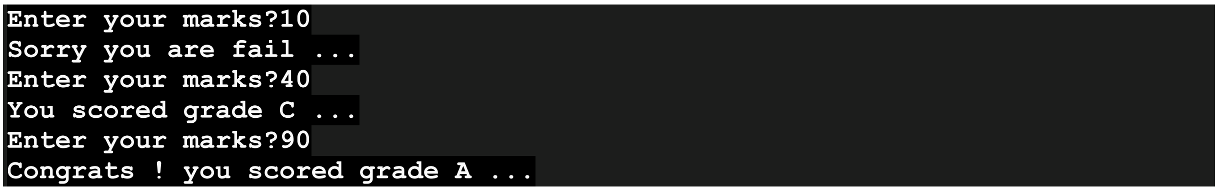
Output

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

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?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.

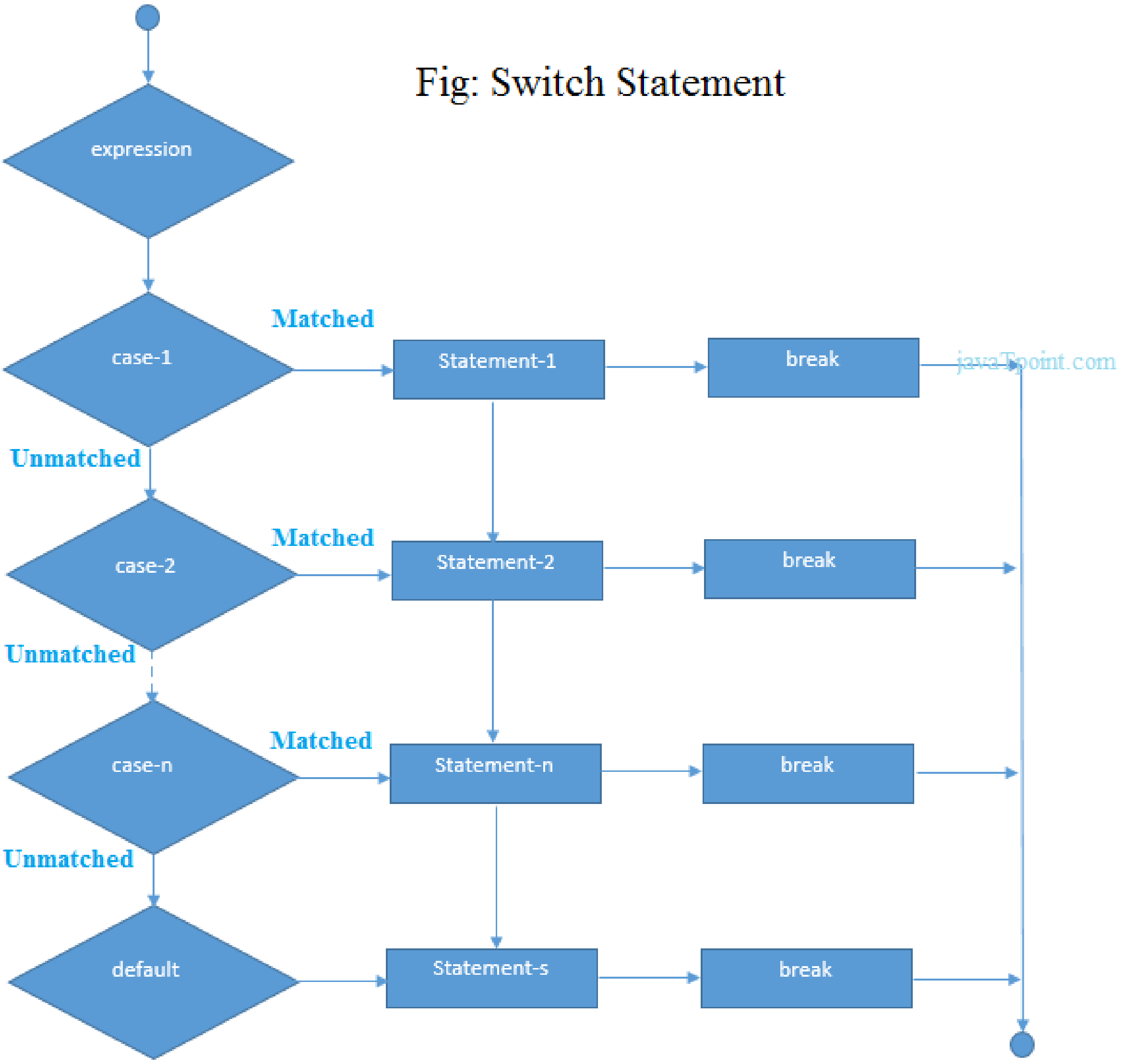
4) The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

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

```
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



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
```

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
```

C LANGUAGE 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.

WHY USE LOOPS IN C LANGUAGE?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

ADVANTAGE OF LOOPS IN C

1) It provides code reusability.

2) Using loops, we do not need to write the same code again and again.

3) Using loops, we can traverse over the elements of data structures (array or linked lists).

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:

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

EXAMPLE 1

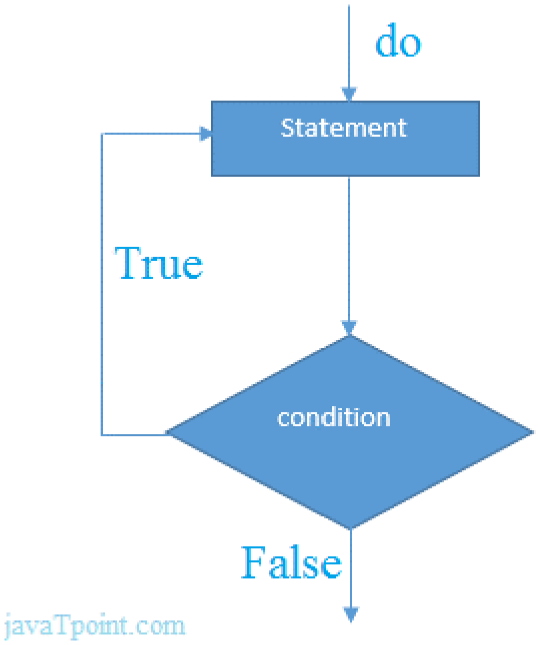
```
#include<stdio.h>
#include<stdlib.h>
void main ()
{
    char c;
   int choice,dummy;
    do{
   printf("\n1. Print Hello\n2. Print Javatpoint\n3. Exit\n");
    scanf("%d",&choice);
   switch(choice)
    {
      case 1 :
       printf("Hello");
      break;
       case 2:
      printf("Javatpoint");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

```
    scanf("%d",&dummy);
    scanf("%c",&c);
    }while(c=='y');
}
```

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.

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

OUTPUT

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

PROGRAM TO PRINT TABLE FOR THE GIVEN NUMBER USING DO WHILE LOOP

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

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
```

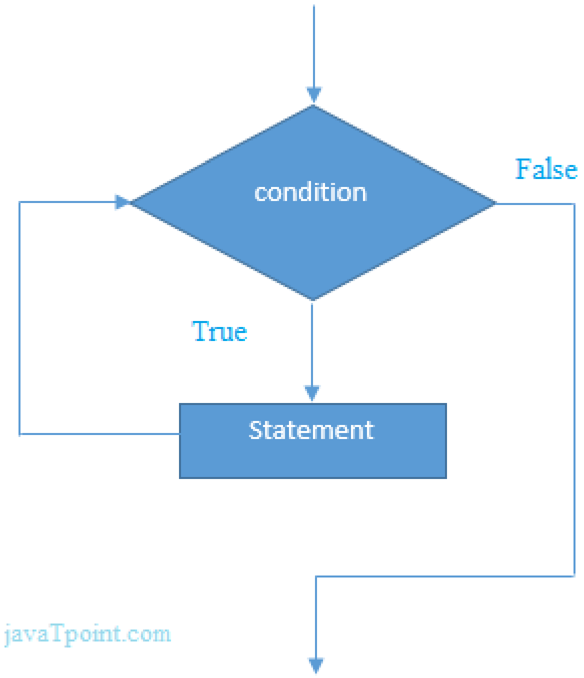
WHILE LOOP IN C

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:

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

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: ");
scanf("%d",&number);
while(i<=10){
printf("%d \n",(number*i));
i++;
}
return 0;
}
```

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.

EXAMPLE 1

```
#include<stdio.h>
void main ()
{
    int j = 1;
    while(j+=2,j<=10)
    {
        printf("%d ",j);
    }
    printf("%d",j);
}
```

OUTPUT

```
3 5 7 9 11
```

EXAMPLE 2

```
#include<stdio.h>
void main ()
{
    while()
    {
        printf("hello Javatpoint");
    }
}
```

OUTPUT

```
compile time error: while loop can't be empty
```

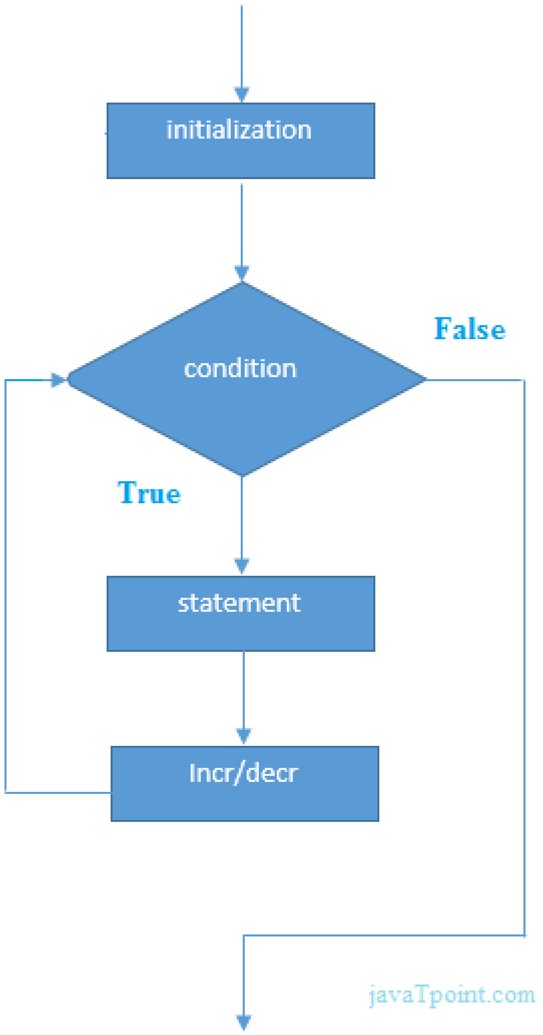
FOR LOOP IN C

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:

```
for(initialization;condition;incr/decr){
//code to be executed
}
```

FLOWCHART OF FOR LOOP IN C

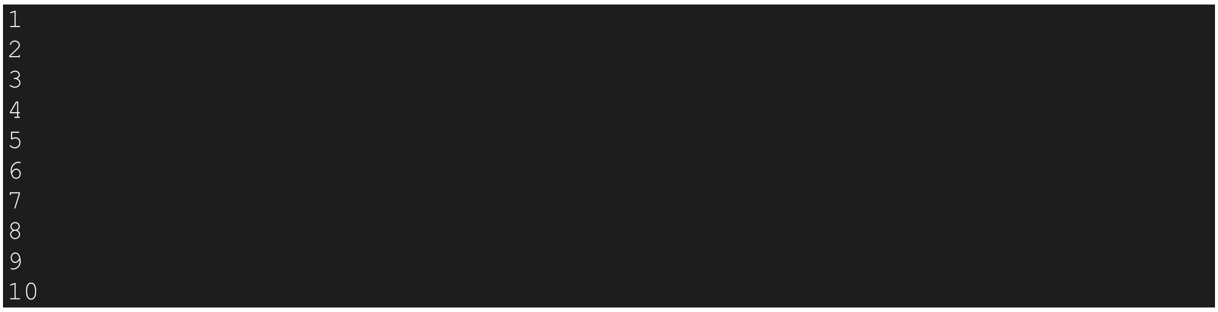


C FOR LOOP EXAMPLES

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

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

Output

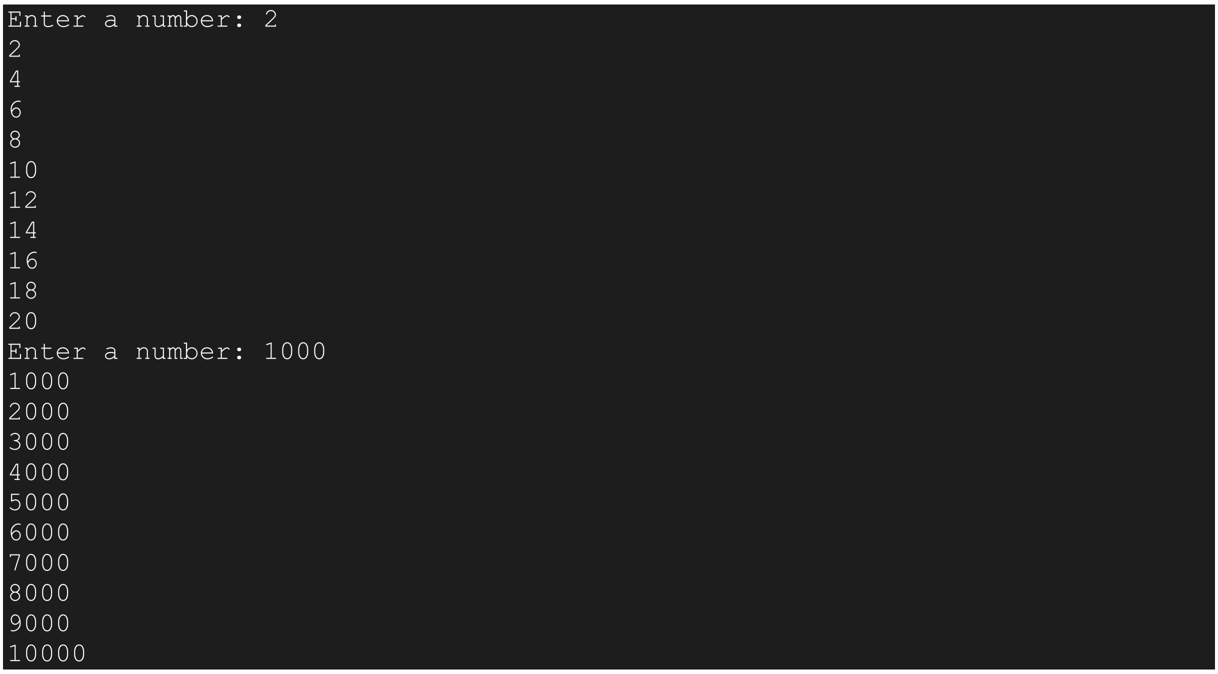


```
1
2
3
4
5
6
7
8
9
10
```

C PROGRAM: PRINT TABLE FOR THE GIVEN NUMBER USING C FOR LOOP

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

Output

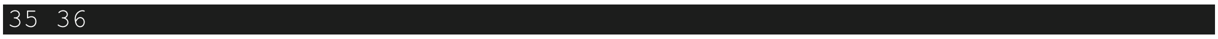


```
Enter a number: 2
2
4
6
8
10
12
14
16
18
20
Enter a number: 1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

Example 1

```
#include <stdio.h>
int main()
{
    int a,b,c;
    for(a=0,b=12,c=23;a<2;a++)
    {
        printf("%d ",a+b+c);
    }
}
```

Output

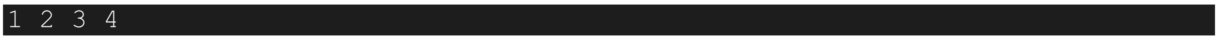


```
35 36
```

Example 2

```
#include <stdio.h>
int main()
{
    int i=1;
    for(;i<5;i++)
    {
        printf("%d ",i);
    }
}
```

Output

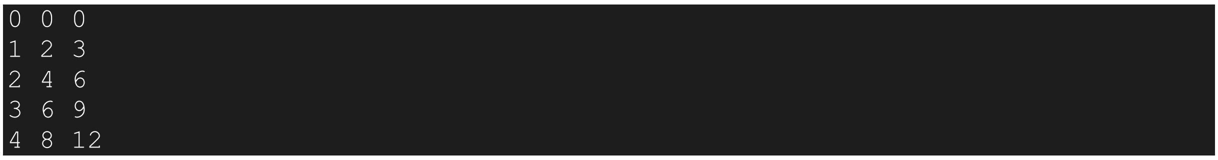


```
1 2 3 4
```

Example 3

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

Output



```
0 0 0
1 2 3
2 4 6
3 6 9
4 8 12
```

NESTED LOOPS IN C

C supports nesting of loops in C. **Nesting of loops** is the feature in C that allows the looping of statements inside another loop. Let's observe an example of nesting loops in C.

Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. You can define any type of loop inside another loop; for example, you can define '**while**' loop inside a '**for**' loop.

Syntax of Nested loop

```
Outer_loop
{
   Inner_loop
  {
         // inner loop statements.
  }
       // outer loop statements.
}
```

Outer_loop and **Inner_loop** are the valid loops that can be a 'for' loop, 'while' loop or 'do-while' loop.

Nested for loop

The nested for loop means any type of loop which is defined inside the 'for' loop.

```
for (initialization; condition; update)
{
   for(initialization; condition; update)
   {
        // inner loop statements.
   }
   // outer loop statements.
}
```

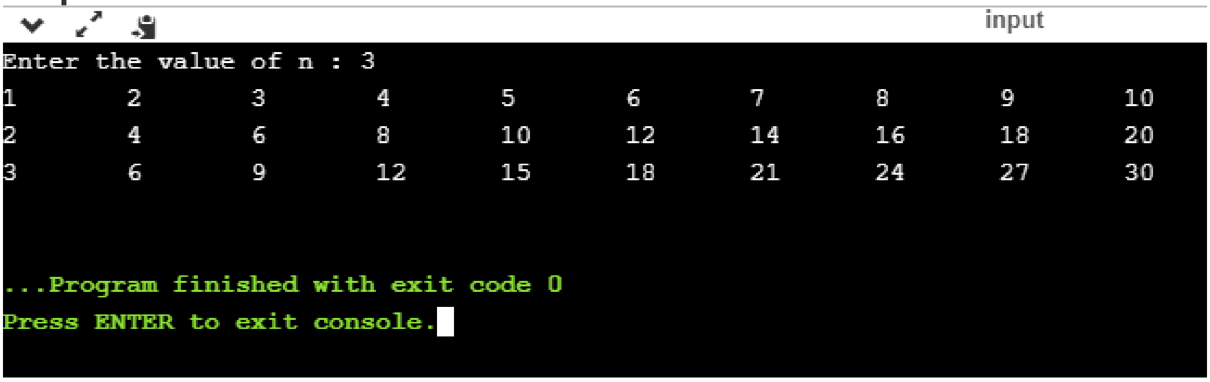
Example of nested for loop

```
#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:



```
Enter the value of n : 3
1         2         3         4         5         6         7         8         9         10
2         4         6         8         10        12        14        16        18        20
3         6         9         12        15        18        21        24        27        30


...Program finished with exit code 0
Press ENTER to exit console.
```

Nested while loop

The nested while loop means any type of loop which is defined inside the 'while' loop.

```
while(condition)
{
   while(condition)
  {
       // inner loop statements.
  }
// outer loop statements.
}
```

Example of nested while loop

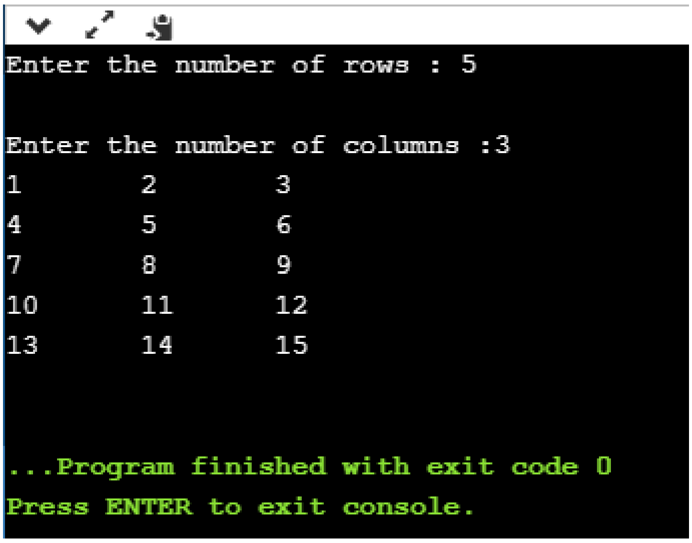
```
#include <stdio.h>
int main()
{
  int rows;  // variable declaration
  int columns; // variable declaration
  int k=1; // variable initialization
  printf("Enter the number of rows :");  // input the number of rows.
  scanf("%d",&rows);
  printf("\nEnter the number of columns :"); // input the number of columns.
  scanf("%d",&columns);
    int a[rows][columns]; //2d array declaration
    int i=1;
  while(i<=rows) // outer loop
  {
     int j=1;
    while(j<=columns)  // inner loop
     {
       printf("%d\t",k);  // printing the value of k.
        k++;   // increment counter
       j++;
     }
    i++;
```

```
        printf("\n");
    }
}
```

Explanation of the above code.

- We have created the 2d array, i.e., int a[rows][columns].
- The program initializes the 'i' variable by 1.
- Now, control moves to the while loop, and this loop checks whether the condition is true, then the program control moves to the inner loop.
- After the execution of the inner loop, the control moves to the update of the outer loop, i.e., i++.
- After incrementing the value of 'i', the condition (i<=rows) is checked.
- If the condition is true, the control then again moves to the inner loop.
- This process continues until the condition of the outer loop is true.

Output:



```
Enter the number of rows : 5

Enter the number of columns :3
1       2       3
4       5       6
7       8       9
10      11      12
13      14      15



...Program finished with exit code 0
Press ENTER to exit console.
```

Nested do while loop

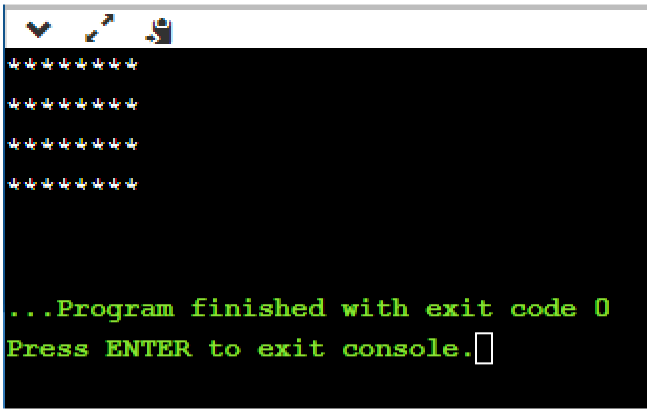
The nested do..while loop means any type of loop which is defined inside the 'do..while' loop.

```
do
{
  do
 {
     // inner loop statements.
  }while(condition);
// outer loop statements.
}while(condition);
```

Example of nested do while loop.

```
#include <stdio.h>
int main()
{
 /*printing the pattern
     ********
    ********
     ********
    ******** */
int i=1;
do          // outer loop
{
   int j=1;
    do       // inner loop
  {
      printf("*");
     j++;
   }while(j<=8);
   printf("\n");
    i++;
    }while(i<=4);
}
```

Output:



```
********
********
********
********


...Program finished with exit code 0
Press ENTER to exit console.
```

Explanation of the above code.

- First, we initialize the outer loop counter variable, i.e., 'i' by 1.
- As we know that the do..while loop executes once without checking the condition, so the inner loop is executed without checking the condition in the outer loop.
- After the execution of the inner loop, the control moves to the update of the i++.
- When the loop counter value is incremented, the condition is checked. If the condition in the outer loop is true, then the inner loop is executed.
- This process will continue until the condition in the outer loop is true.

INFINITE LOOP IN C

WHAT IS INFINITE LOOP?

An infinite loop is a looping construct that does not terminate the loop and executes the loop forever. It is also called an **indefinite** loop or an **endless** loop. It either produces a continuous output or no output.

WHEN TO USE AN INFINITE LOOP

An infinite loop is useful for those applications that accept the user input and generate the output continuously until the user exits from the application manually. In the following situations, this type of loop can be used:

- All the operating systems run in an infinite loop as it does not exist after performing some task. It comes out of an infinite loop only when the user manually shuts down the system.
- All the servers run in an infinite loop as the server responds to all the client requests. It comes out of an indefinite loop only when the administrator shuts down the server manually.
- All the games also run in an infinite loop. The game will accept the user requests until the user exits from the game.

We can create an infinite loop through various loop structures. The following are the loop structures through which we will define the infinite loop:

- for loop
- while loop
- do-while loop
- go to statement
- C macros

FOR LOOP

Let's see the **infinite 'for'** loop. The following is the definition for the **infinite** for loop:

```
for(; ;)
{
   // body of the for loop.
}
```

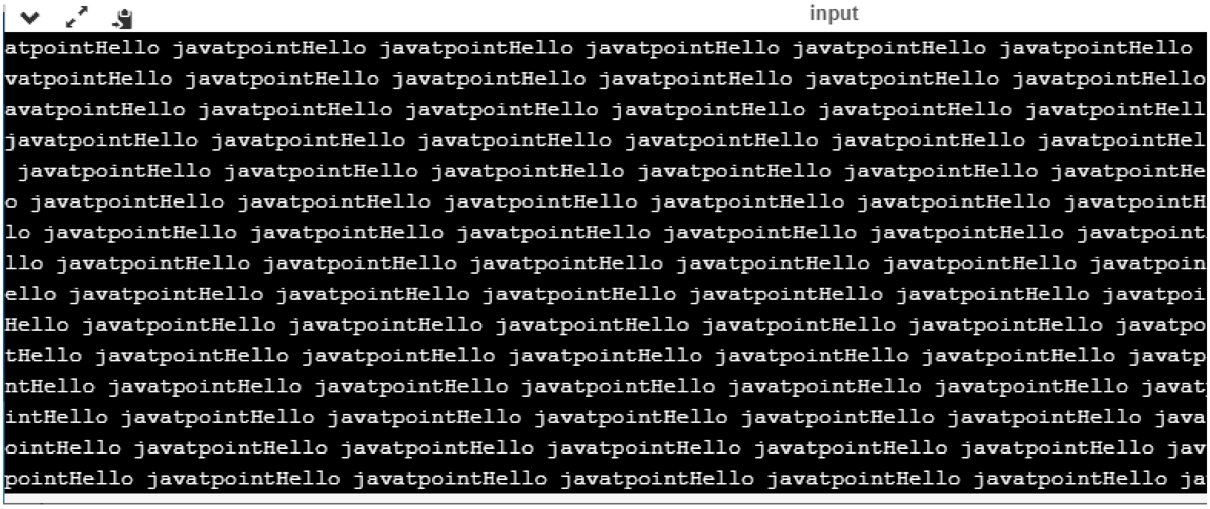
As we know that all the parts of the **'for' loop** are optional, and in the above for loop, we have not mentioned any condition; so, this loop will execute infinite times.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
for(;;)
{
printf("Hello javatpoint");
}
return 0;
}
```

In the above code, we run the 'for' loop infinite times, so **"Hello javatpoint"** will be displayed infinitely.

Output



```
atpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello
vatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello
avatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHell
javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHel
 javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHe
o javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointH
lo javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpoint
llo javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpoin
ello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpoi
Hello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpo
tHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javatp
ntHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello javat
intHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello java
ointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello jav
pointHello javatpointHello javatpointHello javatpointHello javatpointHello javatpointHello ja
```

WHILE LOOP

Now, we will see how to create an infinite loop using a while loop. The following is the definition for the infinite while loop:

```
while(1)
{
  // body of the loop..
}
```

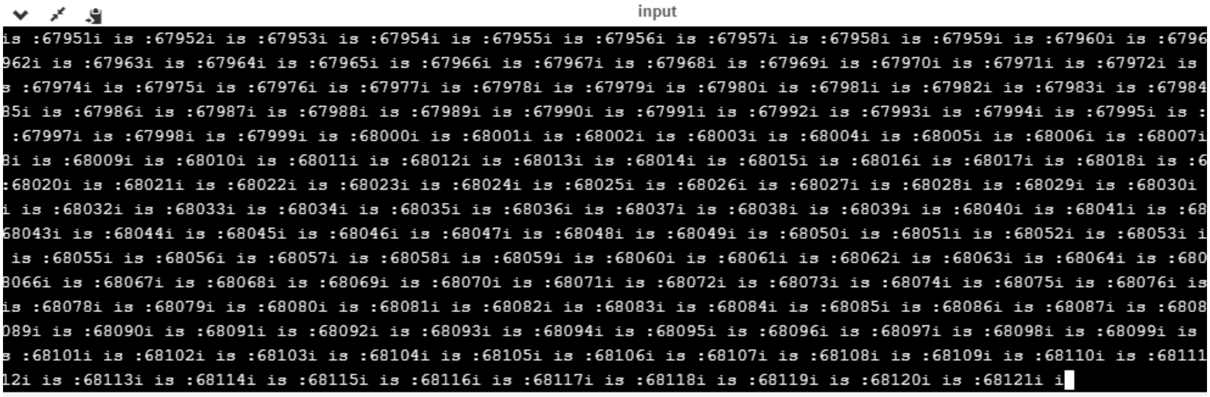
In the above while loop, we put '1' inside the loop condition. As we know that any non-zero integer represents the true condition while '0' represents the false condition.

Let's look at a simple example.

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

In the above code, we have defined a while loop, which runs infinite times as it does not contain any condition. The value of 'i' will be updated an infinite number of times.

Output



```
is :67951i is :67952i is :67953i is :67954i is :67955i is :67956i is :67957i is :67958i is :67959i is :67960i is :6796
962i is :67963i is :67964i is :67965i is :67966i is :67967i is :67968i is :67969i is :67970i is :67971i is :67972i is
s :67974i is :67975i is :67976i is :67977i is :67978i is :67979i is :67980i is :67981i is :67982i is :67983i is :67984
85i is :67986i is :67987i is :67988i is :67989i is :67990i is :67991i is :67992i is :67993i is :67994i is :67995i is :
 :67997i is :67998i is :67999i is :68000i is :68001i is :68002i is :68003i is :68004i is :68005i is :68006i is :68007i
8i is :68009i is :68010i is :68011i is :68012i is :68013i is :68014i is :68015i is :68016i is :68017i is :68018i is :6
:68020i is :68021i is :68022i is :68023i is :68024i is :68025i is :68026i is :68027i is :68028i is :68029i is :68030i
i is :68032i is :68033i is :68034i is :68035i is :68036i is :68037i is :68038i is :68039i is :68040i is :68041i is :68
68043i is :68044i is :68045i is :68046i is :68047i is :68048i is :68049i is :68050i is :68051i is :68052i is :68053i i
 is :68055i is :68056i is :68057i is :68058i is :68059i is :68060i is :68061i is :68062i is :68063i is :68064i is :680
8066i is :68067i is :68068i is :68069i is :68070i is :68071i is :68072i is :68073i is :68074i is :68075i is :68076i is
is :68078i is :68079i is :68080i is :68081i is :68082i is :68083i is :68084i is :68085i is :68086i is :68087i is :6808
089i is :68090i is :68091i is :68092i is :68093i is :68094i is :68095i is :68096i is :68097i is :68098i is :68099i is
s :68101i is :68102i is :68103i is :68104i is :68105i is :68106i is :68107i is :68108i is :68109i is :68110i is :68111
12i is :68113i is :68114i is :68115i is :68116i is :68117i is :68118i is :68119i is :68120i is :68121i i
```

do..while loop

The **do..while** loop can also be used to create the infinite loop. The following is the syntax to create the infinite **do..while** loop.

```
do
{
   // body of the loop..
}while(1);
```

The above do..while loop represents the infinite condition as we provide the '1' value inside the loop condition. As we already know that non-zero integer represents the true condition, so this loop will run infinite times.

C LANGUAGE BREAK STATEMENT

The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. The break statement in C can be used in the following two scenarios:

1. With switch case
2. With loop

SYNTAX:

```
//loop or switch case
break;
```

FLOWCHART OF BREAK IN C

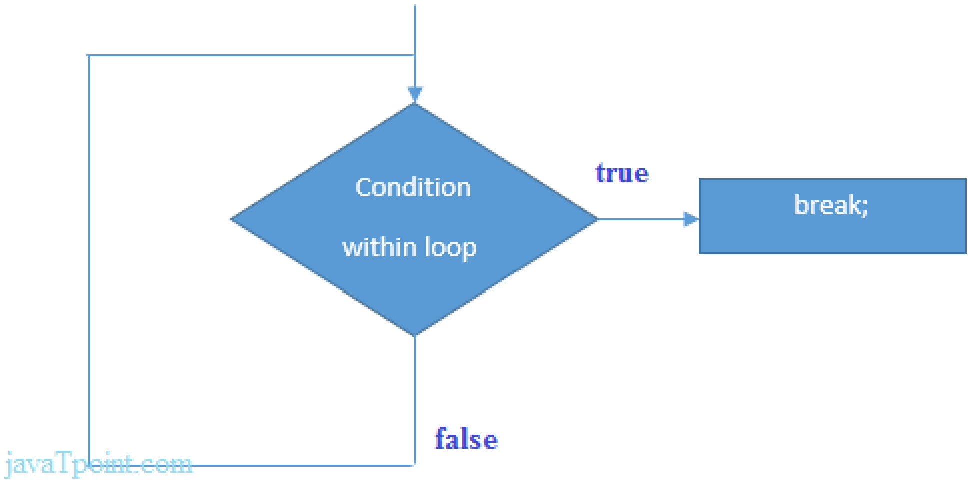
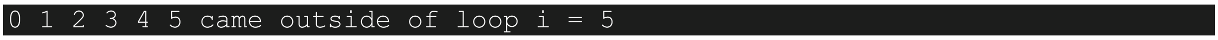


Figure: Flowchart of break statement

EXAMPLE

```
#include<stdio.h>
#include<stdlib.h>
void main ()
{
    int i;
   for(i = 0; i<10; i++)
   {
      printf("%d ",i);
       if(i == 5)
      break;
   }
   printf("came outside of loop i = %d",i);


}
```

Output



```
0 1 2 3 4 5 came outside of loop i = 5
```

C CONTINUE STATEMENT

The **continue statement** in C language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.

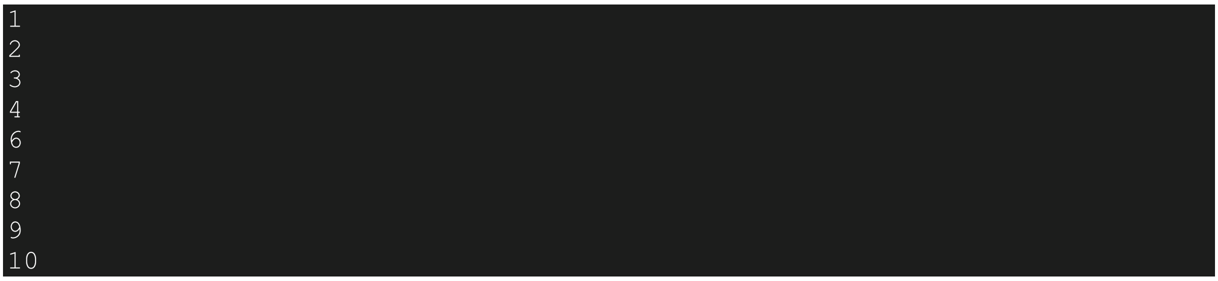
SYNTAX:

```
//loop statements
continue;
//some lines of the code which is to be skipped
```

CONTINUE STATEMENT EXAMPLE 2

```
#include<stdio.h>
int main(){
int i=1;//initializing a local variable
//starting a loop from 1 to 10
for(i=1;i<=10;i++){
if(i==5){//if value of i is equal to 5, it will continue the loop
continue;
}
printf("%d \n",i);
}//end of for loop
return 0;
}
```

Output



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

As you can see, 5 is not printed on the console because loop is continued at i==5.

C GOTO STATEMENT

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statment can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complecated.

Syntax:

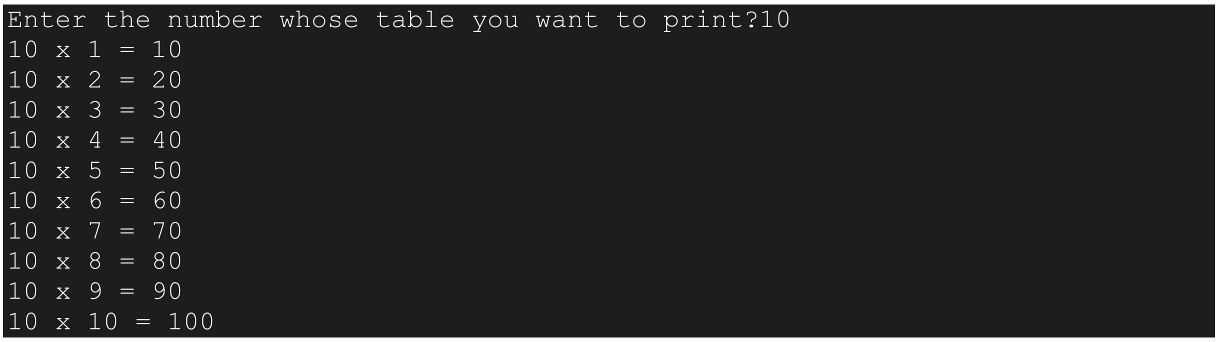
```
label:
//some part of the code;
goto label;
```

GOTO EXAMPLE

Let's see a simple example to use goto statement in C language.

```
#include <stdio.h>
int main()
{
  int num,i=1;
  printf("Enter the number whose table you want to print?");
  scanf("%d",&num);
  table:
  printf("%d x %d = %d\n",num,i,num*i);
  i++;
  if(i<=10)
  goto table;
}
```

Output:



```
Enter the number whose table you want to print?10
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
```

TYPE CASTING IN C LANGUAGE

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

```
(type)value;
```

Without Type Casting:

```
int f= 9/4;
printf("f : %d\n", f );//Output: 2
```

With Type Casting:

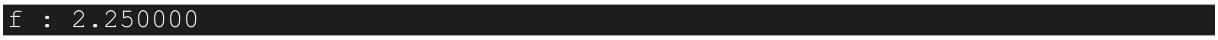
```
float f=(float) 9/4;
printf("f : %f\n", f );//Output: 2.250000
```

TYPE CASTING EXAMPLE

Let's see a simple example to cast int value into the float.

```
#include<stdio.h>
int main(){
float f= (float)9/4;
printf("f : %f\n", f );
return 0;
}
```

Output:



```
f : 2.250000
```

C LANGUAGE 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.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

PROPERTIES OF 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 = 4 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.

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.

DISADVANTAGE OF C 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.

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.

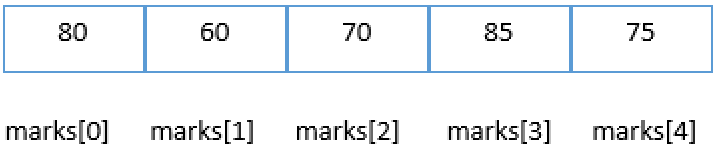
```
int marks[5];
```

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;
//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
```

C ARRAY- DECLARATION WITH INITIALIZATION

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

```
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.

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

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

```
#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
```

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.

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

Consider the following example.

```
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;
}
```

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
```

PASSING ARRAY TO FUNCTION IN C

In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

Consider the following syntax to pass an array to the function.

```
functionname(arrayname);//passing array
```

METHODS TO DECLARE A FUNCTION THAT RECEIVES AN ARRAY AS AN ARGUMENT

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way:

```
return_type function(type arrayname[])
```

Declaring blank subscript notation [] is the widely used technique.

Second way:

```
return_type function(type arrayname[SIZE])
```

Optionally, we can define size in subscript notation [].

Third way:

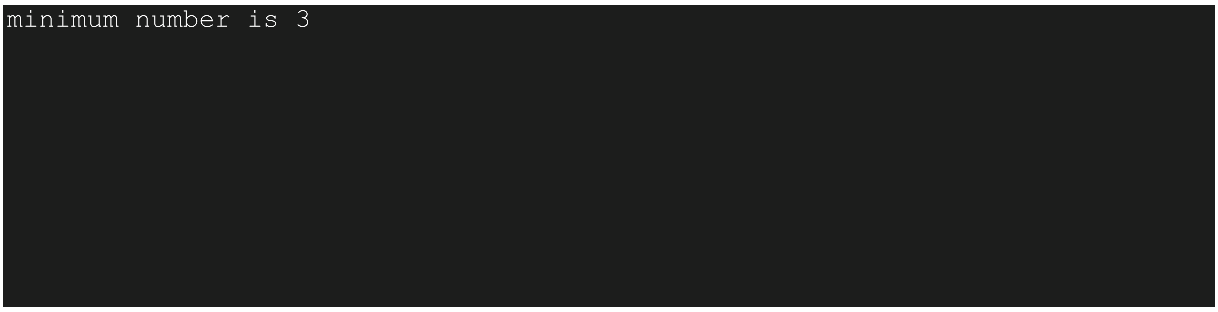
```
return_type function(type *arrayname)
```

You can also use the concept of a pointer. In pointer chapter, we will learn about it.

C LANGUAGE PASSING AN ARRAY TO FUNCTION EXAMPLE

```
#include<stdio.h>
int minarray(int arr[],int size){
int min=arr[0];
int i=0;
for(i=1;i<size;i++){
if(min>arr[i]){
min=arr[i];
}
}//end of for
return min;
}//end of function

int main(){
int i=0,min=0;
int numbers[]={4,5,7,3,8,9};//declaration of array

min=minarray(numbers,6);//passing array with size
printf("minimum number is %d \n",min);
return 0;
}
```

Output

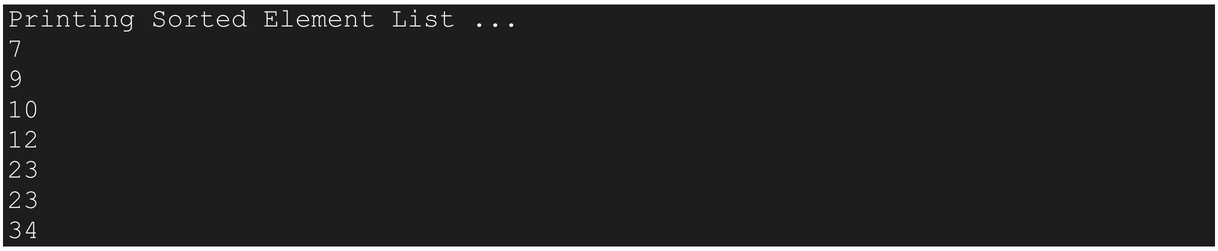


```
minimum number is 3
```

C FUNCTION TO SORT THE ARRAY

```
#include<stdio.h>
void Bubble_Sort(int[]);
void main ()
{
    int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    Bubble_Sort(arr);
}
void Bubble_Sort(int a[]) //array a[] points to arr.
{
int i, j,temp;
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            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]);
    }
}
```

Output



```
Printing Sorted Element List ...
7
9
10
12
23
23
34
```

```
44
78
101
```

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 PROCEDUREor SUBROUTINEin 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;}

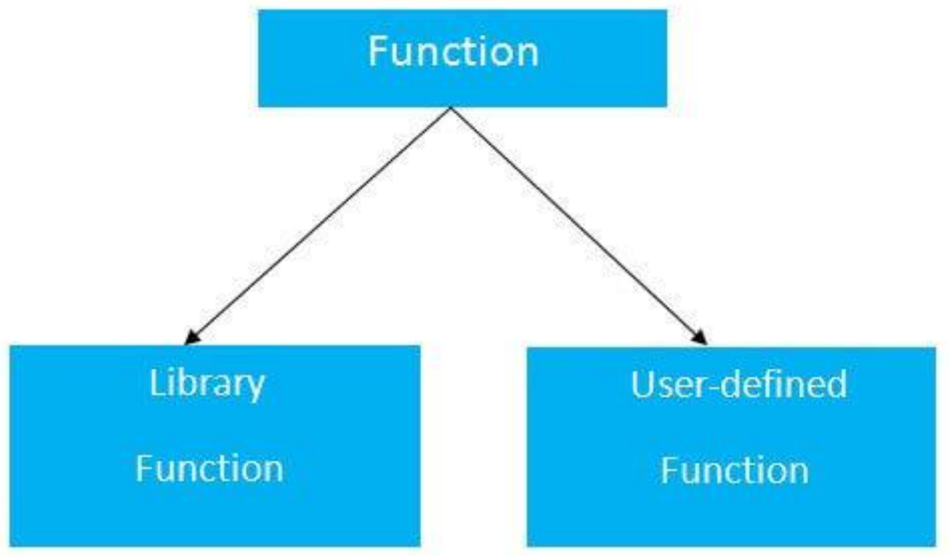
The syntax of creating function in c language is given below:

```
1. return_type function_name(data_type parameter...){
2. //code to be executed
3. }
```

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:

```
void hello(){
printf("hello c");
}
```

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:

```
int get(){
return 10;
}
```

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.

```c
float get(){
return 10.2;
}
```

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
- function with arguments and with return value

EXAMPLE FOR FUNCTION WITHOUT ARGUMENT AND RETURN VALUE

Example 1

```c
#include<stdio.h>
void printName();
void main ()
{
    printf("Hello ");
    printName();
}
void printName()
{
    printf("Javatpoint");
}
```

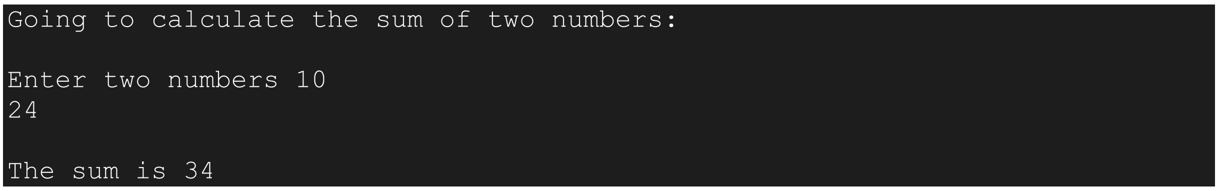
Output

```
Hello Javatpoint
```

Example 2

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

Output



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

Enter two numbers 10
24

The sum is 34
```

C POINTERS

In this tutorial, you'll learn about pointers; what pointers are, how do you use them and the common mistakes you might face when working with them with the help of examples.

Pointers are powerful features of C and C++ programming. Before we learn pointers, let's learn about addresses in C programming.

ADDRESS IN C

If you have a variable `var` in your program, `&var` will give you its address in the memory.

We have used address numerous times while using the `scanf()` function.

```
scanf("%d", &var);
```

Here, the value entered by the user is stored in the address of `var` variable. Let's take a working example.

```
#include <stdio.h>
int main()
{
  int var = 5;
  printf("var: %d\n", var);

  // Notice the use of & before var
  printf("address of var: %p", &var);
  return 0;
}
```

Output

```
var: 5
```

```
address of var: 2686778
```

Note: You will probably get a different address when you run the above code.

C POINTERS

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

POINTER SYNTAX

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer `p` of `int` type.

You can also declare pointers in these ways.

```
int *p1;
int * p2;
```

Let's take another example of declaring pointers.

```
int* p1, p2;
```

Here, we have declared a pointer `p1` and a normal variable `p2`.

ASSIGNING ADDRESSES TO POINTERS

Let's take an example.

```
int* pc, c;
c = 5;
pc = &c;
```

Here, 5 is assigned to the `c` variable. And, the address of `c` is assigned to the `pc` pointer.

GET VALUE OF THING POINTED BY POINTERS

To get the value of the thing pointed by the pointers, we use the `*` operator. For example:

```
int* pc, c;
c = 5;
pc = &c;
printf("%d", *pc);   // Output: 5
```

Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.

Note: In the above example, `pc` is a pointer, not `*pc`. You cannot and should not do something like `*pc = &c`;

By the way, `*` is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

CHANGING VALUE POINTED BY POINTERS

Let's take an example.

```
int* pc, c;
c = 5;
pc = &c;
c = 1;
printf("%d", c);    // Output: 1
printf("%d", *pc);  // Ouptut: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed the value of `c` to 1. Since `pc` and the address of `c` is the same, `*pc` gives us 1.

Let's take another example.

```
int* pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc);  // Ouptut: 1
printf("%d", c);    // Output: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed `*pc` to 1 using `*pc = 1;`. Since `pc` and the address of `c` is the same, `c` will be equal to 1.

Let's take one more example.

```
int* pc, c, d;
c = 5;
d = -15;

pc = &c; printf("%d", *pc); // Output: 5
pc = &d; printf("%d", *pc); // Ouptut: -15
```

Initially, the address of `c` is assigned to the `pc` pointer using `pc = &c;`.

Since `c` is 5, `*pc` gives us 5.

Then, the address of `d` is assigned to the `pc` pointer using `pc = &d;`. Since `d` is -15, `*pc` gives us -15.

EXAMPLE: WORKING OF POINTERS

Let's take a working example.

```
#include <stdio.h>
int main()
{
   int* pc, c;

   c = 22;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c);  // 22

   pc = &c;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 22

   c = 11;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 11

   *pc = 2;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c); // 2
   return 0;
}
```

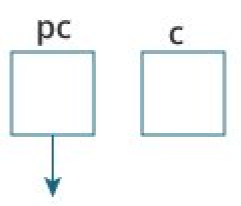
Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11
```

```
Address of c: 2686784
Value of c: 2
```

Explanation of the program

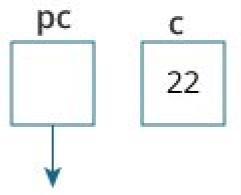
1. `int* pc, c;`

   Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created.

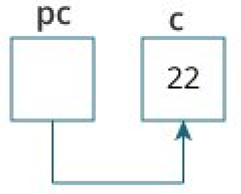
   Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

2. `c = 22;`

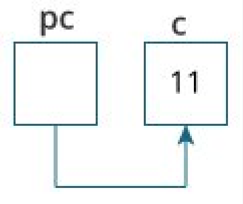
   This assigns 22 to the variable `c`. That is, 22 is stored in the memory location of variable `c`.

3. `pc = &c;`

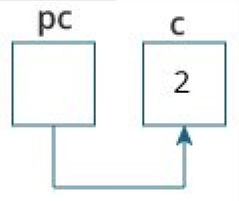
   This assigns the address of variable `c` to the pointer `pc`.

4. `c = 11;`

   This assigns 11 to variable `c`.

5. `*pc = 2;`

   This change the value at the memory location pointed by the pointer `pc` to 2.

COMMON MISTAKES WHEN WORKING WITH POINTERS

Suppose, you want pointer `pc` to point to the address of `c`. Then,

```
int c, *pc;

// pc is address but c is not
pc = c;  // Error

// &c is address but *pc is not
*pc = &c;  // Error

// both &c and pc are addresses
pc = &c;  // Not an error

// both c and *pc values
*pc = c;  // Not an error
```

Here's an example of pointer syntax beginners often find confusing.

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

```
int main() {
   int c = 5;
   int *p = &c;

   printf("%d", *p);  // 5
   return 0;
}
```

Why didn't we get an error when using `int *p = &c;`?

It's because

```
int *p = &c;
```

is equivalent to

```
int *p:
p = &c;
```

In both cases, we are creating a pointer `p` (not `*p`) and assigning `&c` to it.

To avoid this confusion, we can use the statement like this:

```
int* p = &c;
```

RELATIONSHIP BETWEEN ARRAYS AND POINTERS

In this tutorial, you'll learn about the relationship between arrays and pointers in C programming. You will also learn to access array elements using pointers.

Before you learn about the relationship between arrays and pointers, be sure to check these two topics:

- C Arrays

- [C Pointers]

RELATIONSHIP BETWEEN ARRAYS AND POINTERS

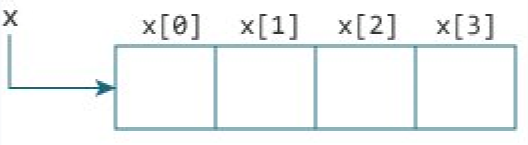
An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
#include <stdio.h>
int main() {
   int x[4];
   int i;

   for(i = 0; i < 4; ++i) {
      printf("&x[%d] = %p\n", i, &x[i]);
   }

   printf("Address of array x: %p", x);

   return 0;
}
```

Output

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

There is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler). Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.



Relation between Arrays and Pointers

From the above example, it is clear that `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.
Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

EXAMPLE 1: POINTERS AND ARRAYS

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

  int i, x[6], sum = 0;

  printf("Enter 6 numbers: ");

  for(i = 0; i < 6; ++i) {
  // Equivalent to scanf("%d", &x[i]);
      scanf("%d", x+i);

  // Equivalent to sum += x[i]
      sum += *(x+i);
  }

  printf("Sum = %d", sum);

  return 0;
}
```

When you run the program, the output will be:

```
Enter 6 numbers: 2
3
4
```

```
4
12
4
Sum = 29
```

Here, we have declared an array `x` of 6 elements. To access elements of the array, we have used pointers.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that **pointers and arrays are not the same**.

EXAMPLE 2: ARRAYS AND POINTERS

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

  int x[5] = {1, 2, 3, 4, 5};
  int* ptr;

  // ptr is assigned the address of the third element
  ptr = &x[2];

  printf("*ptr = %d \n", *ptr);   // 3
  printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
  printf("*(ptr-1) = %d", *(ptr-1));  // 2

  return 0;
}
```

When you run the program, the output will be:

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

In this example, `&x[2]`, the address of the third element, is assigned to the `ptr` pointer. Hence, `3` was displayed when we printed `*ptr`. And, printing `*(ptr+1)` gives us the fourth element. Similarly, printing `*(ptr-1)` gives us the second element.

C Pass Addresses and Pointers

In this tutorial, you'll learn to pass addresses and pointers as arguments to functions with the help of examples.

In C programming, it is also possible to pass addresses as arguments to functions.

To accept these addresses in the function definition, we can use pointers. It's because pointers are used to store addresses. Let's take an example:

Example: Pass Addresses to Functions

```
#include <stdio.h>
void swap(int *n1, int *n2);


int main()
{
    int num1 = 5, num2 = 10;
    // address of num1 and num2 is passed
    swap( &num1, &num2);

    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

When you run the program, the output will be:

```
num1 = 10
num2 = 5
```

The address of `num1` and `num2` are passed to the `swap()` function using `swap(&num1, &num2);`.

Pointers `n1` and `n2` accept these arguments in the function definition.

```
void swap(int* n1, int* n2) {
    ... ..
}
```

When `*n1` and `*n2` are changed inside the `swap()` function, `num1` and `num2` inside the `main()` function are also changed.

Inside the `swap()` function, `*n1` and `*n2` swapped. Hence, `num1` and `num2` are also swapped.

Notice that `swap()` is not returning anything; its return type is `void`.

Example 2: Passing Pointers to Functions

```
#include <stdio.h>

void addOne(int* ptr) {
  (*ptr)++; // adding 1 to *ptr
}

int main()
{
  int* p, i = 10;
  p = &i;
  addOne(p);

  printf("%d", *p); // 11
  return 0;
}
```

Here, the value stored at `p`, `*p`, is 10 initially.

We then passed the pointer `p` to the `addOne()` function. The `ptr` pointer gets this address in the `addOne()` function.

Inside the function, we increased the value stored at `ptr` by 1 using `(*ptr)++;`. Since `ptr` and `p` pointers both have the same address, `*p` inside `main()` is also 11.

EXAMPLE: LARGEST ELEMENT IN AN ARRAY

```
#include <stdio.h>
int main() {
  int n;
  double arr[100];
  printf("Enter the number of elements (1 to 100): ");
  scanf("%d", &n);

  for (int i = 0; i < n; ++i) {
    printf("Enter number%d: ", i + 1);
    scanf("%lf", &arr[i]);
  }

  // storing the largest number to arr[0]
  for (int i = 1; i < n; ++i) {
    if (arr[0] < arr[i]) {
      arr[0] = arr[i];
    }
  }

  printf("Largest element = %.2lf", arr[0]);

  return 0;
}
```

Output

```
Enter the number of elements (1 to 100): 5
Enter number1: 34.5
Enter number2: 2.4
Enter number3: -35.5
Enter number4: 38.7
Enter number5: 24.5
Largest element = 38.70
```

This program takes `n` number of elements from the user and stores it in the `arr` array.

To find the largest element,

- the first two elements of array are checked and the largest of these two elements are placed in `arr[0]`
- the first and third elements are checked and largest of these two elements is placed in `arr[0]`.

- this process continues until the first and last elements are checked
- the largest number will be stored in the `arr[0]` position

```
// storing the largest number at arr[0]
for (int i = 1; i < n; ++i) {
  if (arr[0] < arr[i]) {
    arr[0] = arr[i];
  }
}
```

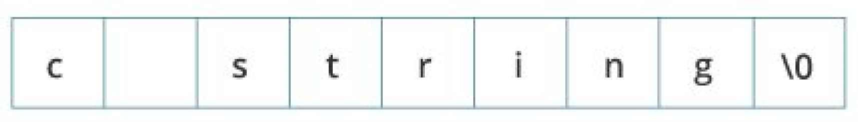
C PROGRAMMING STRINGS

In this tutorial, you'll learn about strings in C programming. You'll learn to declare them, initialize them and use them for various I/O operations with the help of examples.

In C programming, a string is a sequence of characters terminated with a null character `\0`. For example:

```
char c[] = "c string";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character `\0` at the end by default.

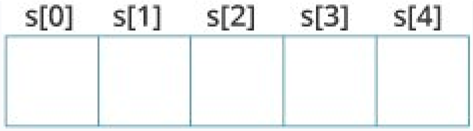


Memory Diagram

HOW TO DECLARE A STRING?

Here's how you can declare strings:

```
char s[5];
```



String Declaration in C

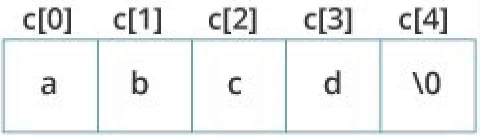
Here, we have declared a string of 5 characters.

HOW TO INITIALIZE STRINGS?

You can initialize strings in a number of ways.

```
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};
```

```c
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

String Initialization in C

Let's take another example:

```c
char c[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is `'\0'`) to a `char` array having 5 characters. This is bad and you should never do this.

ASSIGNING VALUES TO STRINGS

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```c
char c[100];
c = "C programming";  // Error! array type is not assignable.
```

> **Note:** Use the strcpy() function to copy the string instead.

READ STRING FROM THE USER

You can use the `scanf()` function to read a string.

The `scanf()` function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

EXAMPLE 1: SCANF() TO READ A STRING

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Even though `Dennis Ritchie` was entered in the above program, only `"Dennis"` was stored in the `name` string. It's because there was a space after `Dennis`.

Also notice that we have used the code `name` instead of `&name` with `scanf()`.

```
scanf("%s", name);
```

This is because `name` is a `char` array, and we know that array names decay to pointers in C.

Thus, the `name` in `scanf()` already points to the address of the first element in the string, which is why we don't need to use `&`.

HOW TO READ A LINE OF TEXT?

You can use the `fgets()` function to read a line of string. And, you can use `puts()` to display the string.

EXAMPLE 2: FGETS() AND PUTS()

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);  // read string
    printf("Name: ");
    puts(name);    // display string
    return 0;
}
```

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Here, we have used `fgets()` function to read a string from the user.

`fgets(name, sizeof(name), stdlin); // read string`

The `sizeof(name)` results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the `name` string.

To print the string, we have used `puts(name);`.

> **Note:** The `gets()` function can also be to take input from the user. However, it is removed from the C standard.
>
> It's because `gets()` allows you to input any length of characters. Hence, there might be a buffer overflow.

PASSING STRINGS TO FUNCTIONS

Strings can be passed to a function in a similar way as arrays. Learn more about passing arrays to a function.

EXAMPLE 3: PASSING STRING TO A FUNCTION

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str);     // Passing string to a function.
    return 0;
}
void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

STRINGS AND POINTERS

Similar like arrays, string names are "decayed" to pointers. Hence, you can use pointers to manipulate elements of the string. We recommended you to check C Arrays and Pointers before you check this example.

EXAMPLE 4: STRINGS AND POINTERS

```
#include <stdio.h>

int main(void) {
  char name[] = "Harry Potter";

  printf("%c", *name);     // Output: H
  printf("%c", *(name+1));   // Output: a
  printf("%c", *(name+7));   // Output: o

  char *namePtr;

  namePtr = name;
  printf("%c", *namePtr);     // Output: H
  printf("%c", *(namePtr+1));   // Output: a
  printf("%c", *(namePtr+7));   // Output: o
}
```

COMMONLY USED STRING FUNCTIONS

- **strlen()** - calculates the length of a string
- **strcpy()** - copies a string to another
- **strcmp()** - compares two strings
- **strcat()** - concatenates two strings

Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	computes string's length

Function	Work of Function
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

Strings handling functions are defined under `"string.h"` header file.

```
#include <string.h>
```

Note: You have to include the code below to run string handling functions.

GETS() AND PUTS()

Functions gets() and puts() are two string functions to take string input from the user and display it respectively as mentioned in the previous chapter.

```
#include<stdio.h>

int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);     //Function to read string from user.
    printf("Name: ");
    puts(name);    //Function to display string.
    return 0;
}
```

Note: Though, `gets()` and `puts()` function handle strings, both these functions are defined in `"stdio.h"` header file.

Program to count vowels, consonants, etc.

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

  char line[150];
  int vowels, consonant, digit, space;

  // initialize all variables to 0
  vowels = consonant = digit = space = 0;

  // get full line of string input
  printf("Enter a line of string: ");
  fgets(line, sizeof(line), stdin);

  // loop through each character of the string
  for (int i = 0; line[i] != '\0'; ++i) {

    // convert character to lowercase
    line[i] = strlwr(line[i]);

    // check if the character is a vowel
    if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||
        line[i] == 'o' || line[i] == 'u') {

      // increment value of vowels by 1
      ++vowels;
    }

    // if it is not a vowel and if it is an alphabet, it is a consonant
    else if ((line[i] >= 'a' && line[i] <= 'z')) {
      ++consonant;
    }

    // check if the character is a digit
    else if (line[i] >= '0' && line[i] <= '9') {
      ++digit;
    }

    // check if the character is an empty space
    else if (line[i] == ' ') {
      ++space;
    }
  }

  printf("Vowels: %d", vowels);
  printf("\nConsonants: %d", consonant);
  printf("\nDigits: %d", digit);
  printf("\nWhite spaces: %d", space);

  return 0;
}
```

Output

```
Enter a line of string: C++ 20 is the latest version of C++ yet.
Vowels: 9
Consonants: 16
Digits: 2
White spaces: 8
```

Here, the string entered by the user is stored in the `line` variable.

Initially, the variables `vowel`, `consonant`, `digit,` and `space` are initialized to **0**.

Then, a `for` loop is used to iterate over the characters of the string. In each iteration, we:

- convert the character to lowercase using the `tolower()` function

- check whether the character is a vowel, a consonant, a digit, or an empty space. Suppose, the character is a consonant. Then, the `consonant` variable is increased by **1**.

When the loop ends, the number of vowels, consonants, digits, and white spaces are stored in variables `vowel`, `consonant`, `digit,` and `space` respectively.