

# C Programming Language Tutorial

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.

## 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

In this tutorial, all C programs are given with C compiler so that you can quickly change the C program code.

File: main.c

```
#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 the 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

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.

# How to install C

There are many compilers available for c and c++. You need to download any one. Here, we are going to use **Turbo C++**. It will work for both C and C++. To install the Turbo C software, you need to follow following steps.

1. Download Turbo C++
2. Create turboc directory inside c drive and extract the tc3.zip inside c:\turboc
3. Double click on install.exe file
4. Click on the tc application file located inside c:\TC\BIN to write the c program

## 1) Download Turbo C++ software

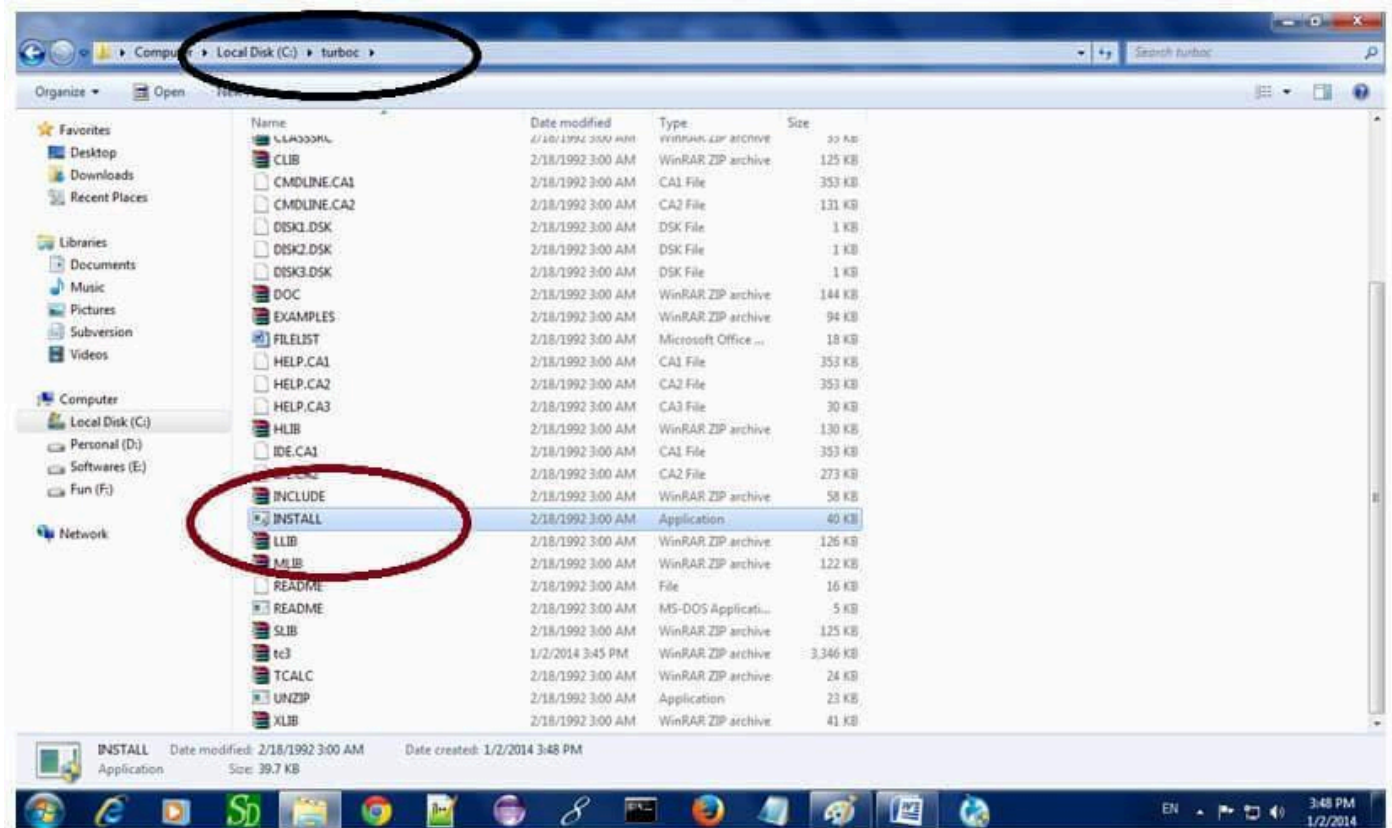
You can download turbo c++ from many sites. [download Turbo c++](#)

## 2) Create turboc directory in c drive and extract the tc3.zip

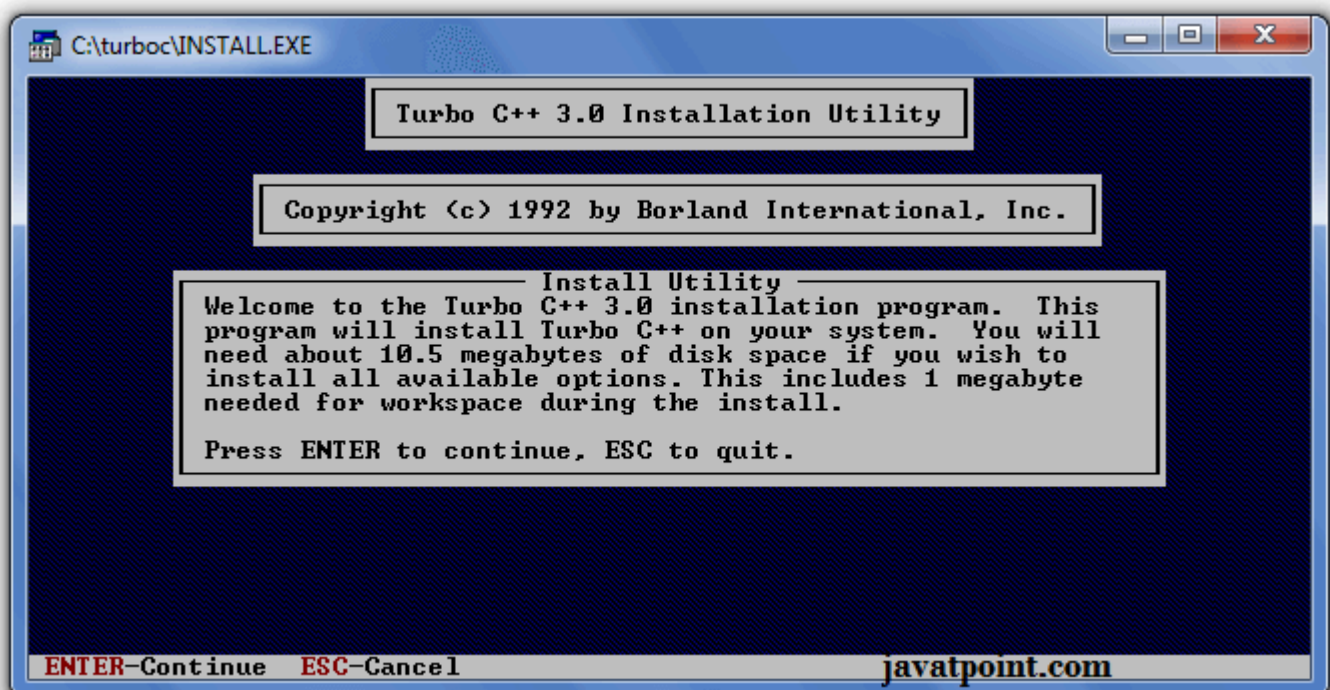
Now, you need to create a new directory turboc inside the c: drive. Now extract the tc3.zip file in c:\truboc directory.

## 3) Double click on the install.exe file and follow steps

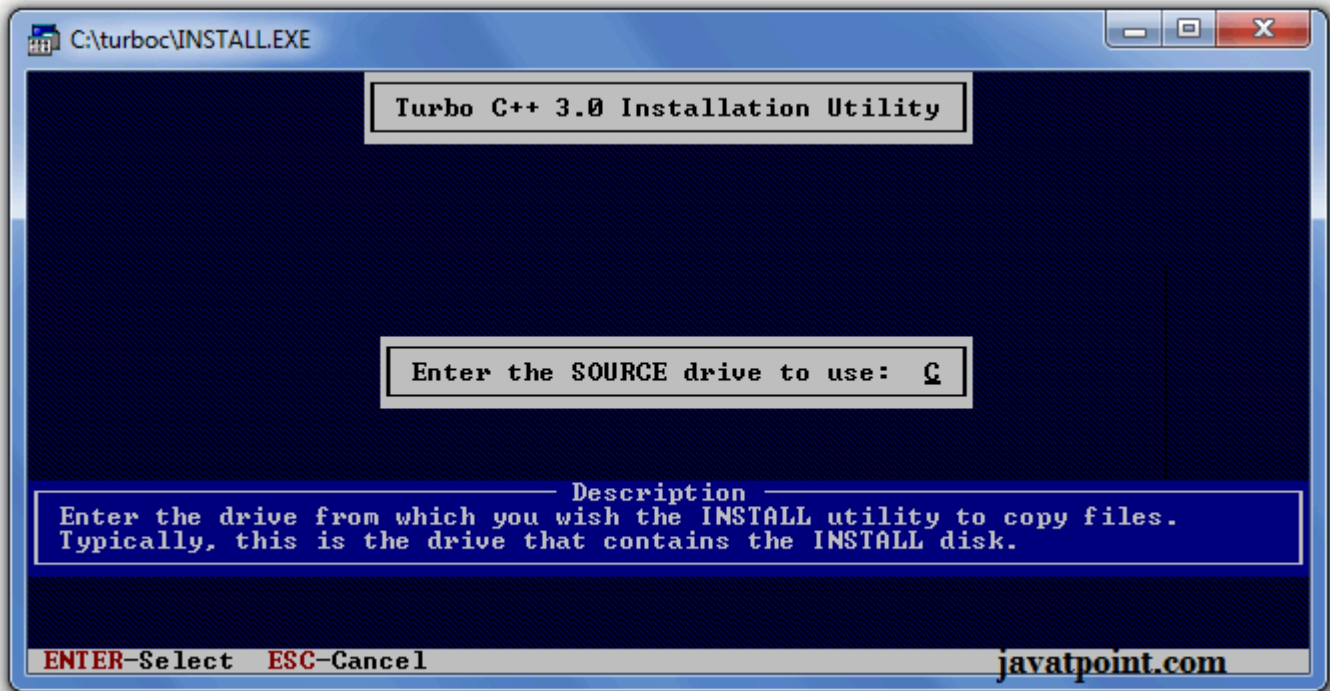
Now, click on the install icon located inside the c:\turboc



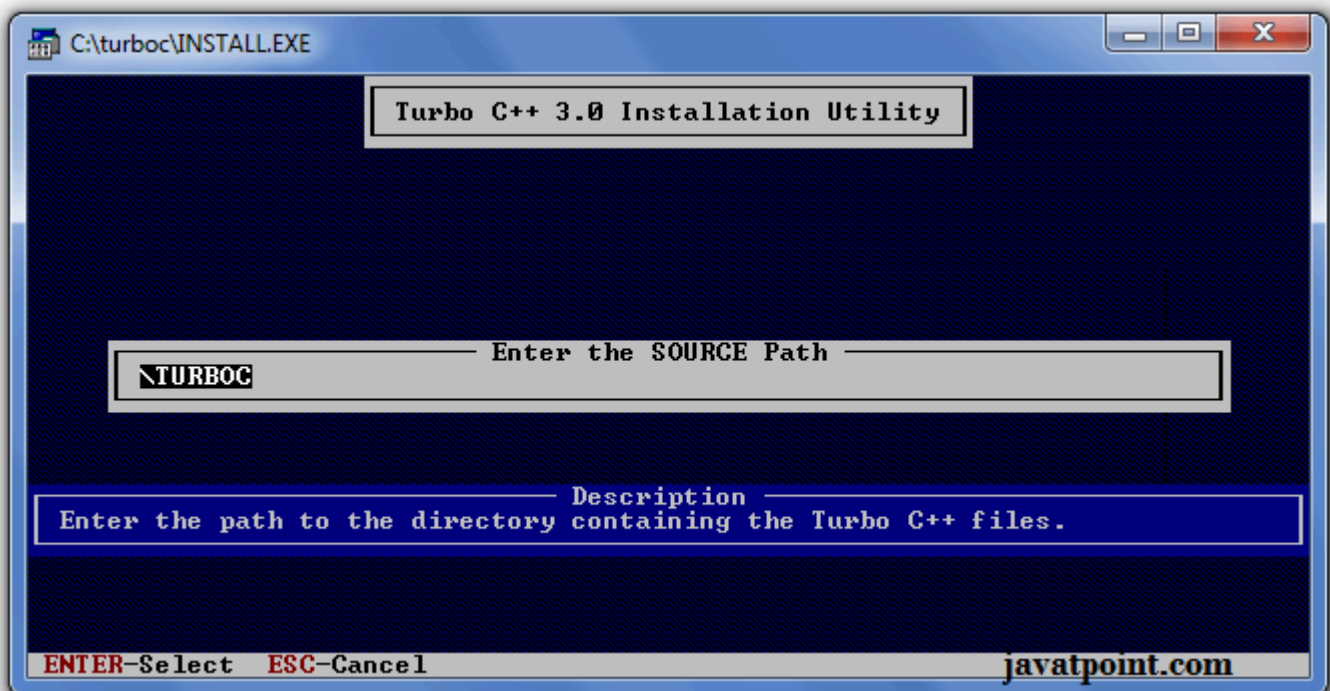
It will ask you to install c or not, press enter to install.



Change your drive to c, press c.

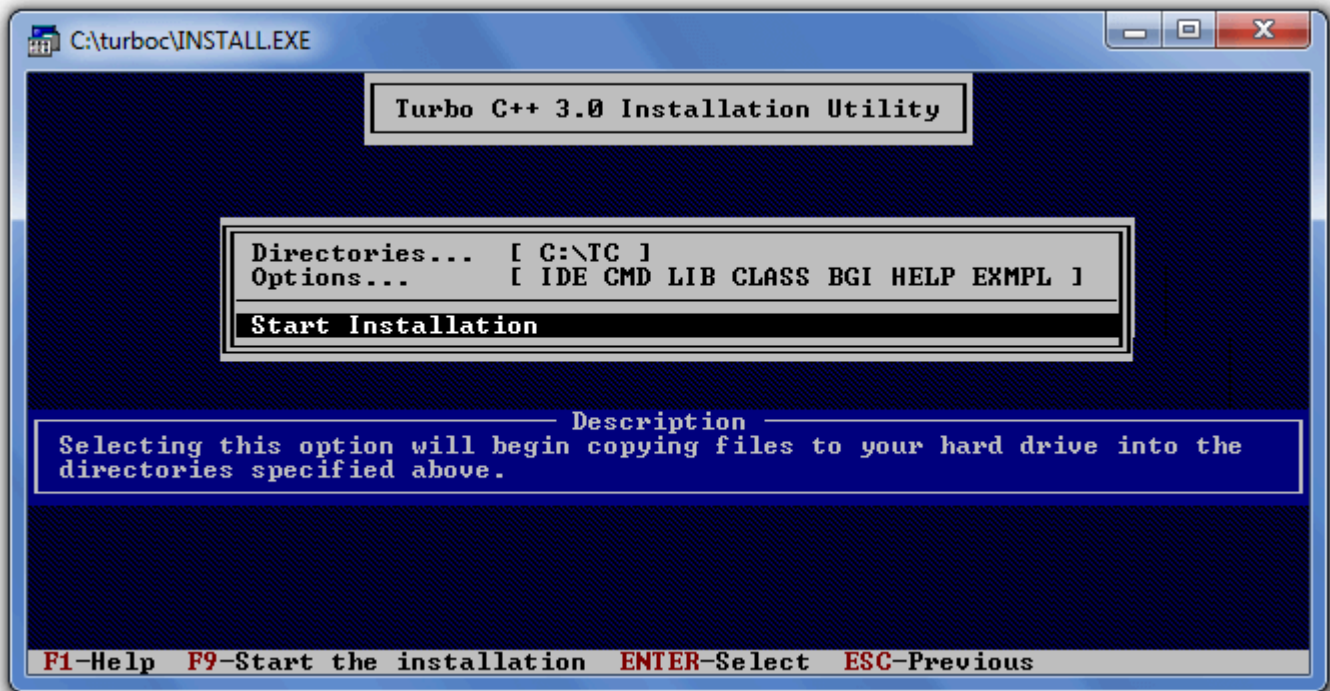


Press enter, it will look inside the c:\turboc directory for the required files.

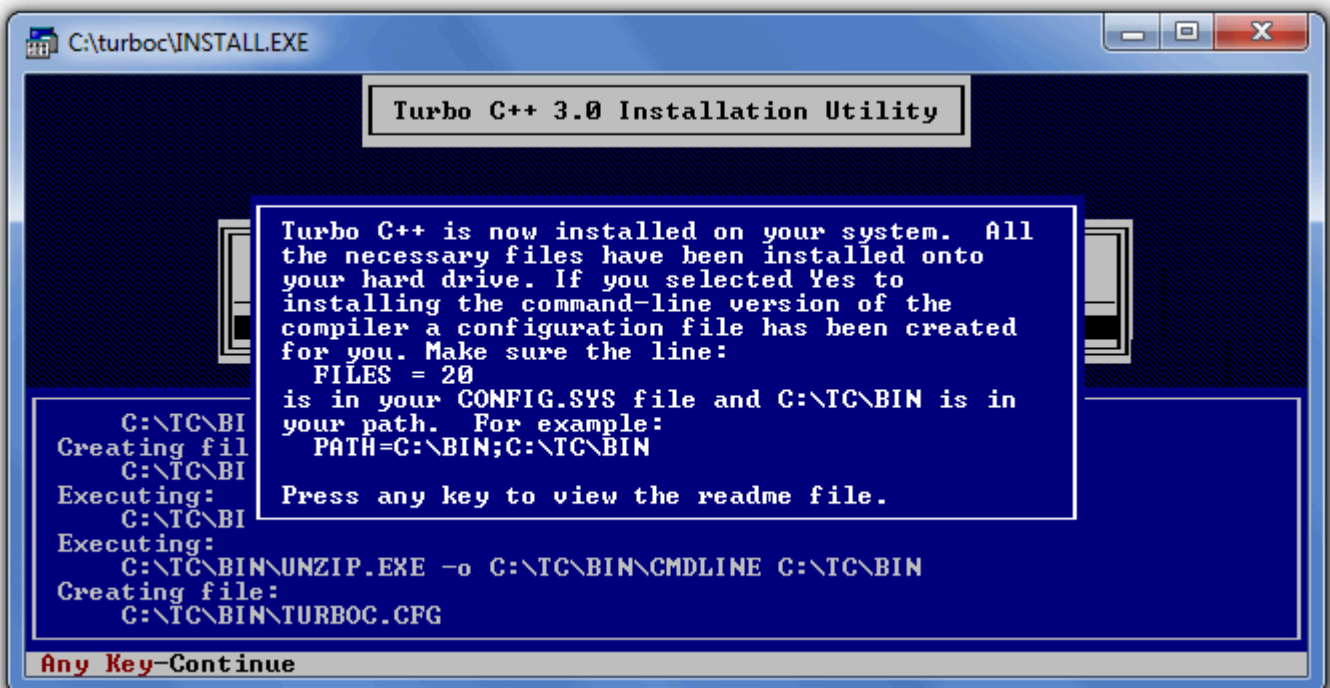


Select Start installation by the down arrow key then press enter.



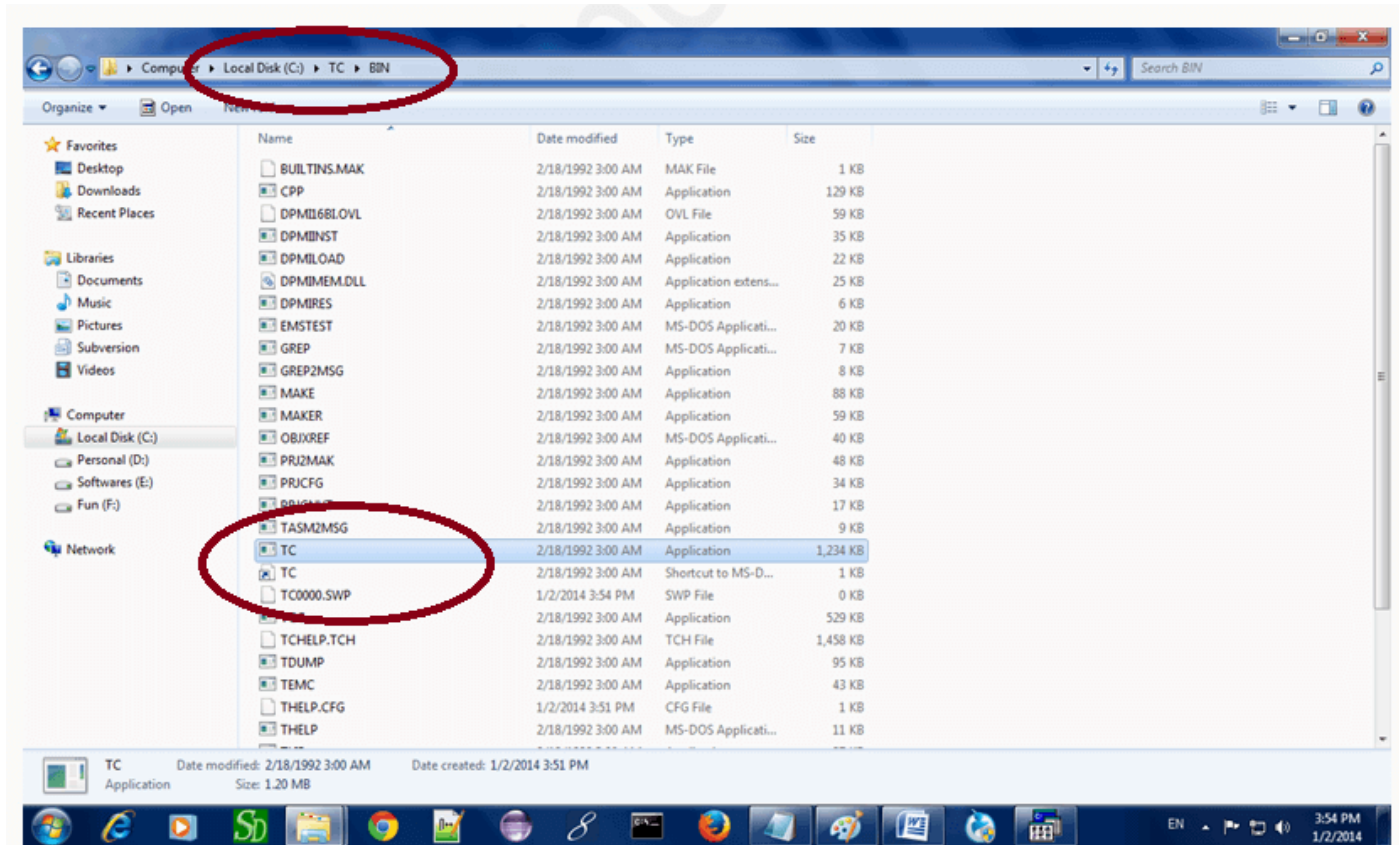


Now C is installed, press enter to read documentation or close the software.



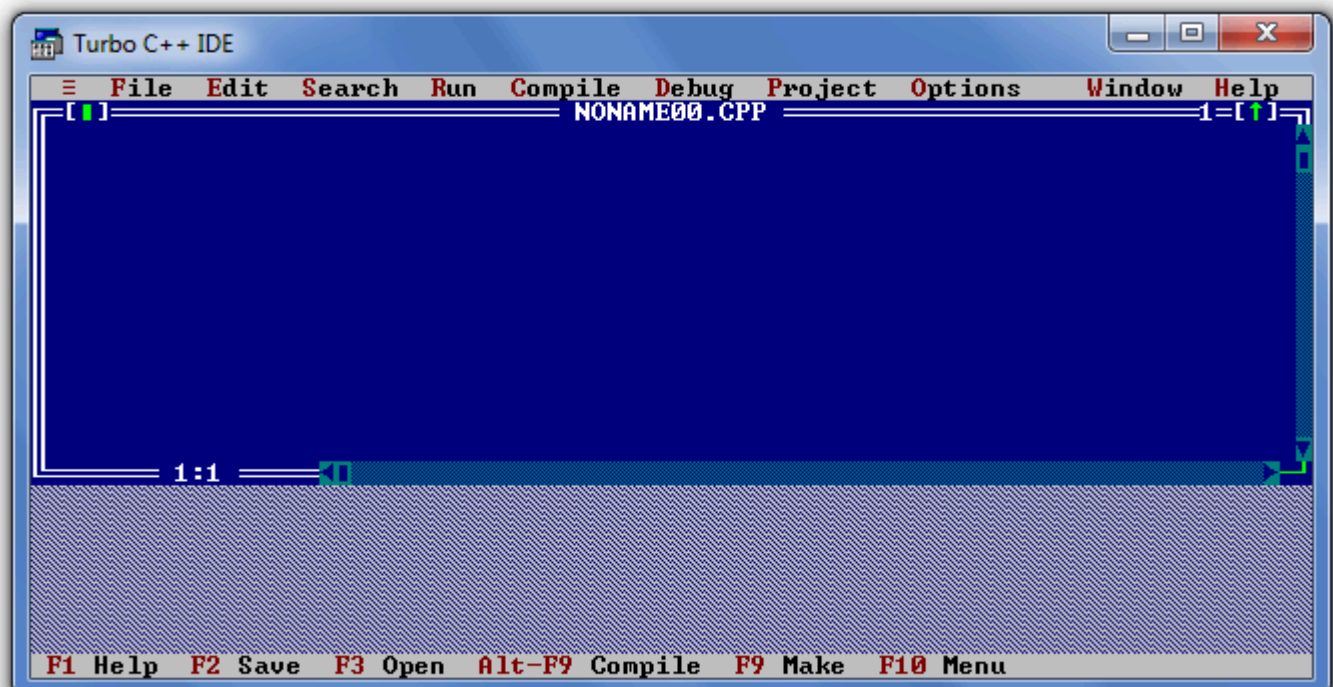
#### 4) Click on the tc application located inside c:\TC\BIN

Now double click on the tc icon located in c:\TC\BIN directory to write the c program.



In windows 7 or window 8, it will show a dialog block to ignore and close the application because fullscreen mode is not supported. Click on Ignore button.

Now it will showing following console.





# First C Program

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.

## How to compile and run the c program

There are 2 ways to compile and run the c program, by menu and by shortcut.

### By menu

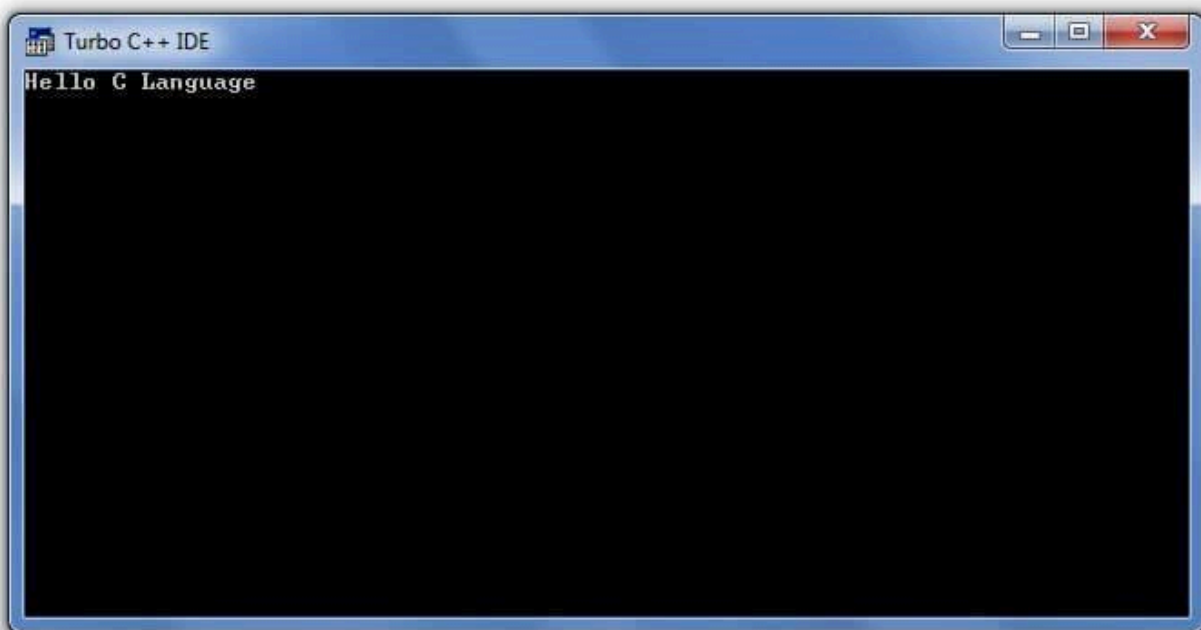
Now click on the **compile menu** then **compile sub menu** to compile the c program.

Then click on the **run menu** then **run sub menu** to run the c program.

### By shortcut

Or, press **ctrl+f9** keys compile and run the program directly.

You will see the following output on user screen.



You can view the user screen any time by pressing the **alt+f5** keys.

Now press **Esc** to return to the turbo c++ console.

# Compilation process in c

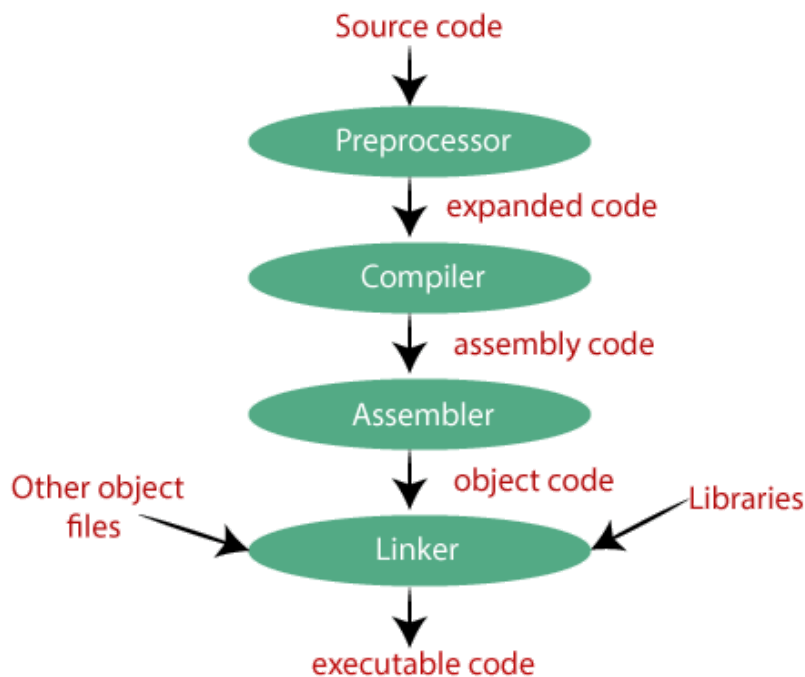
## What is a compilation?

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

The preprocessor takes the source code as an input, and it removes all the comments from the source code. The preprocessor takes the preprocessor directive and interprets it. For example, if `<stdio.h>`, the directive is available in the program, then the preprocessor interprets the directive and replace this directive with the content of the 'stdio.h' file.

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



- Preprocessor
- Compiler
- Assembler
- Linker

## Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

## Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

## Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj', and in UNIX, the extension is '.o'. If the name of the source file is '**hello.c**', then the name of the object file would be 'hello.obj'.

## Linker

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', **Let's understand through an example.**

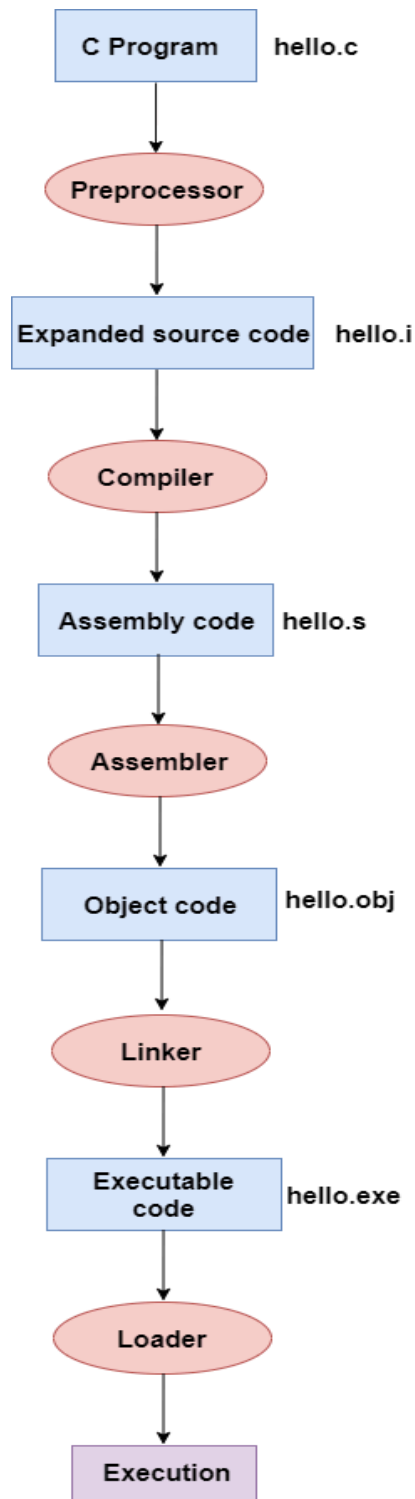
hello.c

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

Now, we will create a flow diagram of the above program:

In the above flow diagram, the following steps are taken to execute a program:

- Firstly, the input file, i.e., **hello.c**, is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i**.
- The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s**.
- This assembly code is then sent to the assembler, which converts the assembly code into object code.
- After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.



## printf() and scanf() in C

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

### printf() function

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

The syntax of printf() function is given below:

1. 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.

1. `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:

1. type variable\_list;

The example of declaring the variable is given below:

1. `int a;`
2. `float b;`
3. `char c;`

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

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

1. `int a=10,b=20; //declaring 2 variable of integer type`
2. `float f=20.8;`
3. `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:

1. `int a;`
2. `int _ab;`
3. `int a30;`

Invalid variable names:

1. `int 2;`
2. `int a b;`
3. `int long;`

## Types of Variables in C

There are many types of variables in c:

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

### Local Variable

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

It must be declared at the start of the block.

```
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.

## Automatic Variable

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

```
void main(){
    int x=10;//local variable (also automatic)
    auto int y=20;//automatic variable
}
```

## External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

*myfile.h*

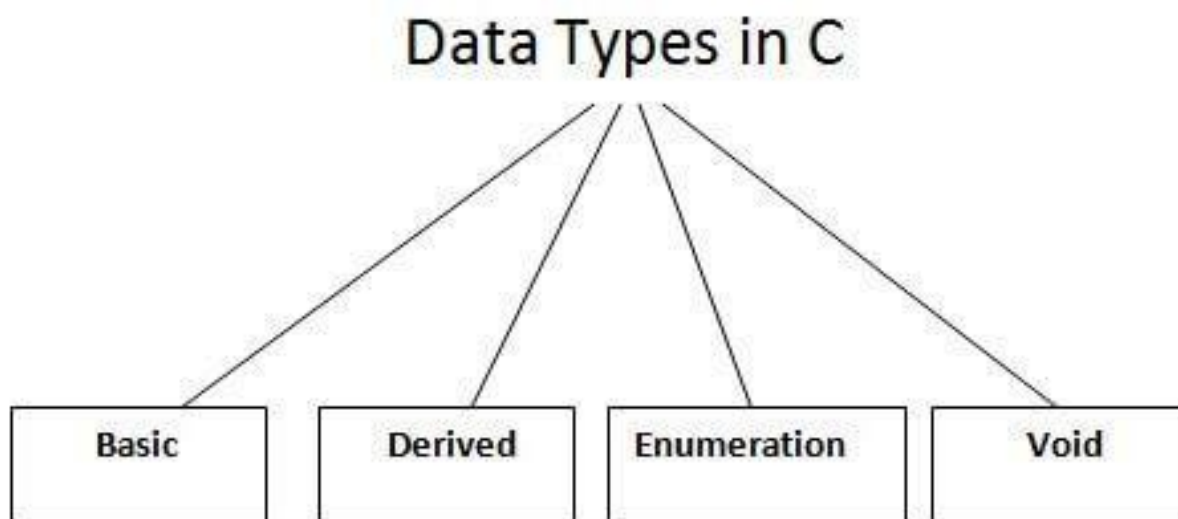
```
extern int x=10;//external variable (also global)
```

*program1.c*

```
#include "myfile.h"
#include <stdio.h>
void printValue(){
    printf("Global variable: %d", global_variable);
}
```

# 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
<b>short</b>	2 byte	–32,768 to 32,767
signed short	2 byte	–32,768 to 32,767
unsigned short	2 byte	0 to 65,535
<b>int</b>	2 byte	–32,768 to 32,767
signed int	2 byte	–32,768 to 32,767
unsigned int	2 byte	0 to 65,535
<b>short int</b>	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
<b>long int</b>	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
<b>float</b>	4 byte	
<b>double</b>	8 byte	
<b>long double</b>	10 byte	

## Keywords in C

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

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

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if

int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

## C Identifiers

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

1. 2sum (starts with a numerical digit)
2. **int** (reserved word)
3. **char** (reserved word)
4. 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.

```
int main()
{
    int a=10;
    int A=20;
    printf("Value of a is : %d",a);
    printf("\nValue of A is :%d",A);
    return 0;
}
```

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 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:

1. `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	<code>() [] -&gt; . ++ --</code>	Left to right
Unary	<code>+ - ! ~ ++ -- (type)* &amp; sizeof</code>	Right to left
Multiplicative	<code>* / %</code>	Left to right
Additive	<code>+ -</code>	Left to right
Shift	<code>&lt;&lt; &gt;&gt;</code>	Left to right
Relational	<code>&lt; &lt;= &gt; &gt;=</code>	Left to right
Equality	<code>== !=</code>	Left to right
Bitwise AND	<code>&amp;</code>	Left to right
Bitwise XOR	<code>^</code>	Left to right
Bitwise OR	<code> </code>	Left to right
Logical AND	<code>&amp;&amp;</code>	Left to right
Logical OR	<code>  </code>	Left to right
Conditional	<code>?:</code>	Right to left
Assignment	<code>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=  =</code>	Right to left
Comma	<code>,</code>	Left to right



# Comments in C

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
```

## 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:

1. `/*`
2. `code`
3. `to be commented`
4. `*/`

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 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.

Let's understand the format specifiers in detail through an example.

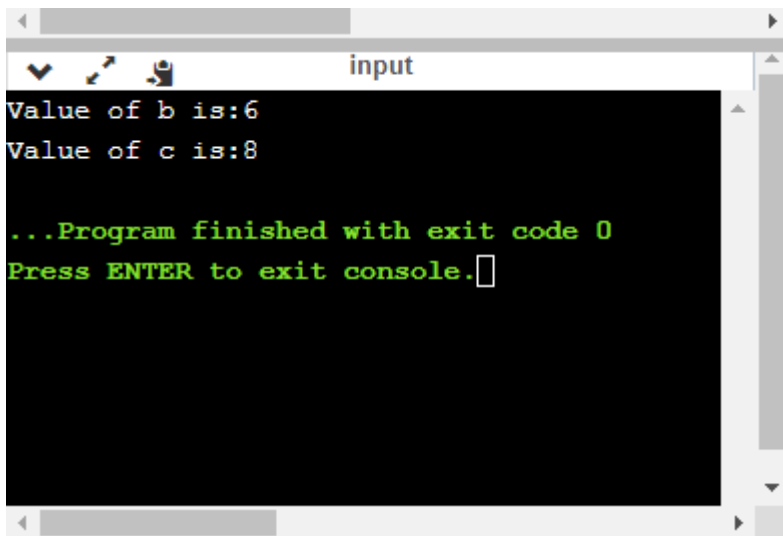
- %d

```
int main()
{
    int b=6;
    int c=8;
    printf("Value of b is:%d", b);
    printf("\nValue of c is:%d",c);

    return 0;
}
```

In the above code, we are printing the integer value of b and c by using the %d specifier.

Output



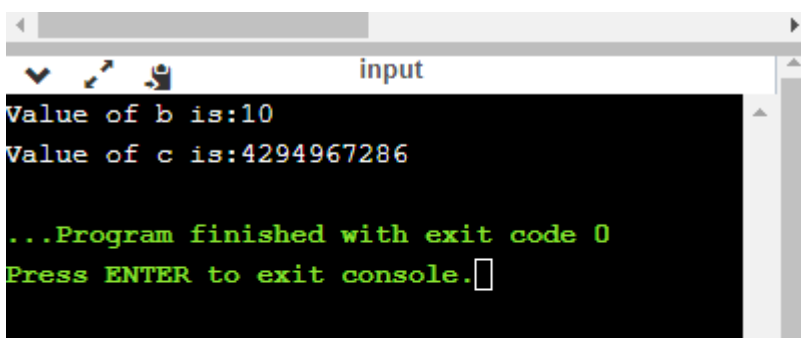
- %u

```
int main()
{
    int b=10;
    int c= -10;
    printf("Value of b is:%u", b);
    printf("\nValue of c is:%u",c);

    return 0;
}
```

In the above program, we are displaying the value of b and c by using an unsigned format specifier, i.e., %u. The value of b is positive, so %u specifier prints the exact value of b, but it does not print the value of c as c contains the negative value.

Output

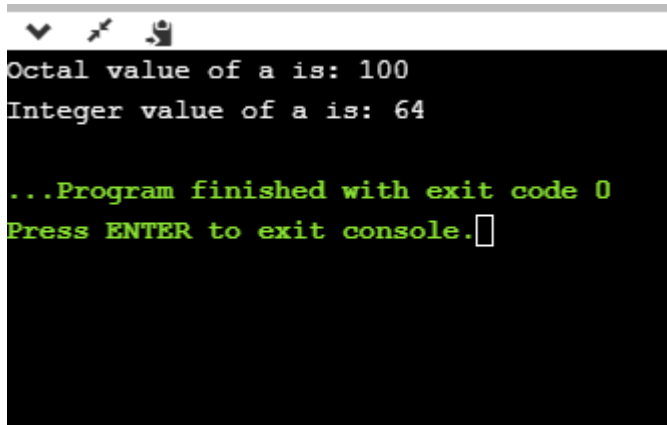


- %o

```
int main()
{
    int a=0100;
    printf("Octal value of a is: %o", a);
    printf("\nInteger value of a is: %d",a);
    return 0;
}
```

In the above code, we are displaying the octal value and integer value of a.

### Output



```
Octal value of a is: 100
Integer value of a is: 64

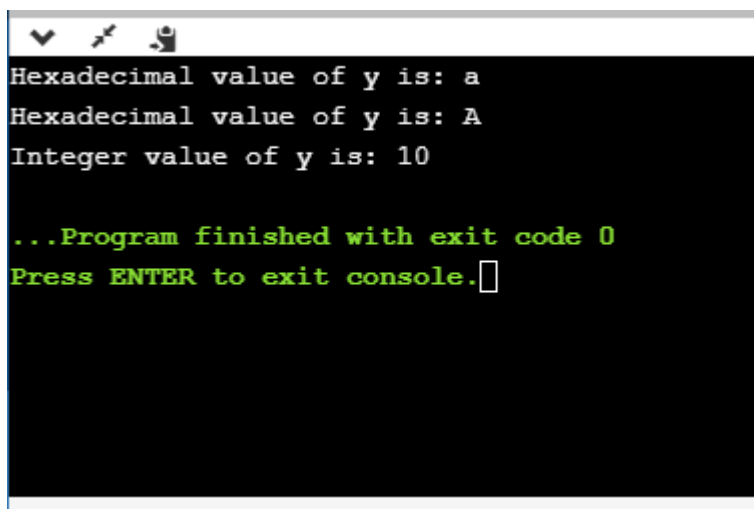
...Program finished with exit code 0
Press ENTER to exit console.
```

- %x and %X

```
int main()
{
    int y=0xA;
    printf("Hexadecimal value of y is: %x", y);
    printf("\nHexadecimal value of y is: %X",y);
    printf("\nInteger value of y is: %d",y);
    return 0;
}
```

In the above code, y contains the hexadecimal value 'A'. We display the hexadecimal value of y in two formats. We use %x and %X to print the hexadecimal value where %x displays the value in small letters, i.e., 'a' and %X displays the value in a capital letter, i.e., 'A'.

### Output



```
Hexadecimal value of y is: a
Hexadecimal value of y is: A
Integer value of y is: 10

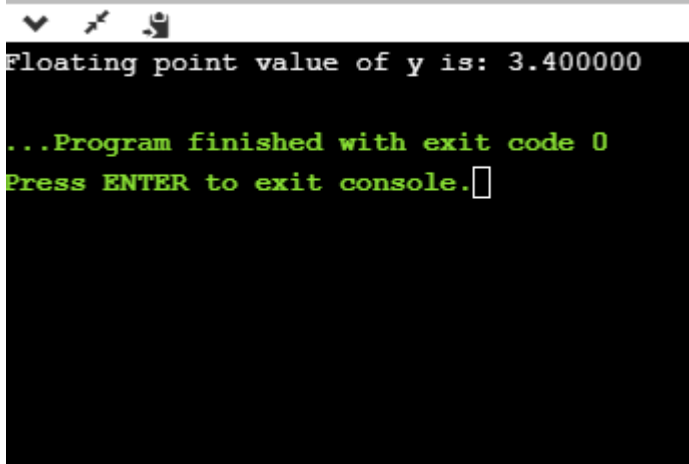
...Program finished with exit code 0
Press ENTER to exit console.
```

- %f

```
int main()
{
    float y=3.4;
    printf("Floating point value of y is: %f", y);
    return 0;
}
```

The above code prints the floating value of y.

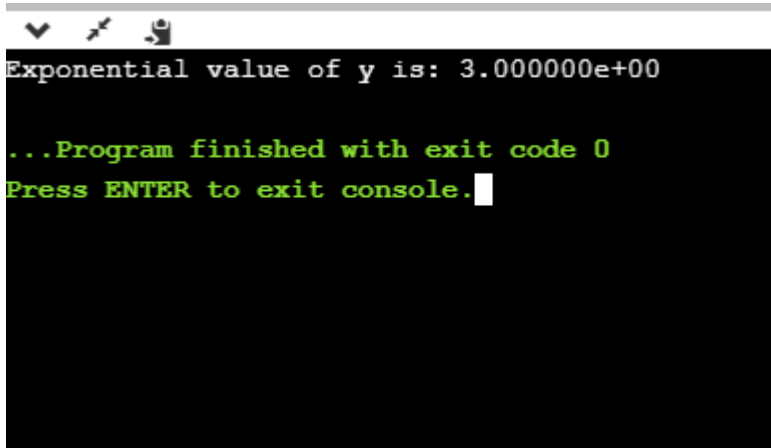
Output

A screenshot of a terminal window with a black background and green text. At the top, there are three small icons: a checkmark, a cursor, and a document. The main text in the terminal reads: "Floating point value of y is: 3.400000". Below this, it says "...Program finished with exit code 0" and "Press ENTER to exit console." followed by a small white square cursor.

- %e

```
int main()
{
    float y=3;
    printf("Exponential value of y is: %e", y);
    return 0;
}
```

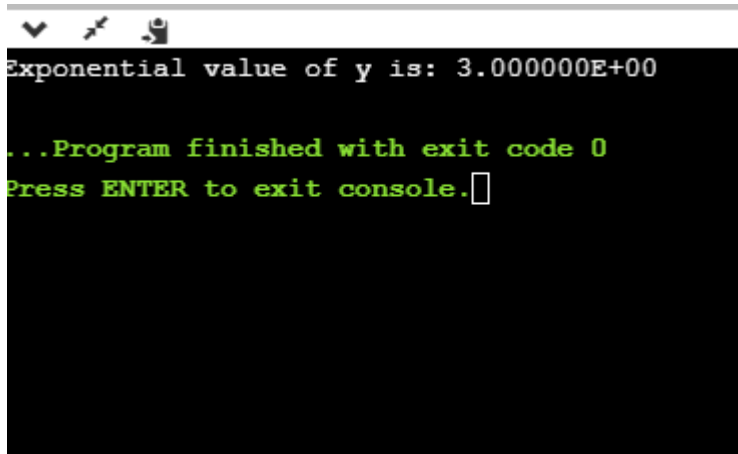
Output

A screenshot of a terminal window with a black background and green text. At the top, there are three small icons: a checkmark, a cursor, and a document. The main text in the terminal reads: "Exponential value of y is: 3.000000e+00". Below this, it says "...Program finished with exit code 0" and "Press ENTER to exit console." followed by a small white square cursor.

- %E

```
int main()
{
    float y=3;
    printf("Exponential value of y is: %E", y);
    return 0;
}
```

## Output



```
Exponential value of y is: 3.000000E+00

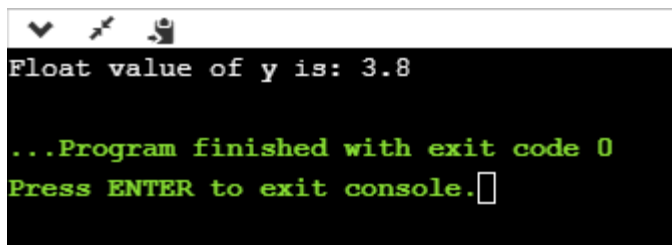
...Program finished with exit code 0
Press ENTER to exit console.
```

- %g

```
int main()
{
    float y=3.8;
    printf("Float value of y is: %g", y);
    return 0;
}
```

In the above code, we are displaying the floating value of y by using %g specifier. The %g specifier displays the output same as the input with a same precision.

## Output



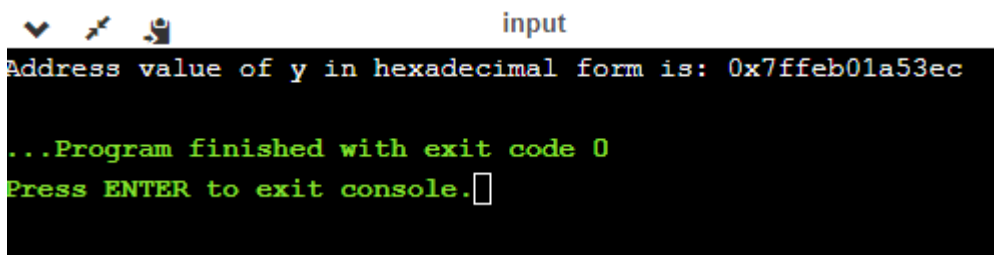
```
Float value of y is: 3.8

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

- %p

```
int main()
{
    int y=5;
    printf("Address value of y in hexadecimal form is: %p", &y);
    return 0;
}
```

## Output



```
input
Address value of y in hexadecimal form is: 0x7ffeb01a53ec

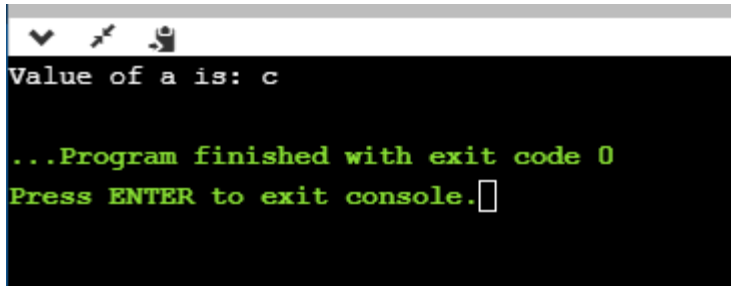
...Program finished with exit code 0
Press ENTER to exit console.
```



- %c

```
int main()
{
    char a='c';
    printf("Value of a is: %c", a);
    return 0;
}
```

Output

A terminal window with a black background and green text. The first line shows 'Value of a is: c'. The second line shows '...Program finished with exit code 0'. The third line shows 'Press ENTER to exit console.' followed by a cursor icon.

- %s

```
int main()
{
    printf("%s", "Edureka");
    return 0;
}
```

Output

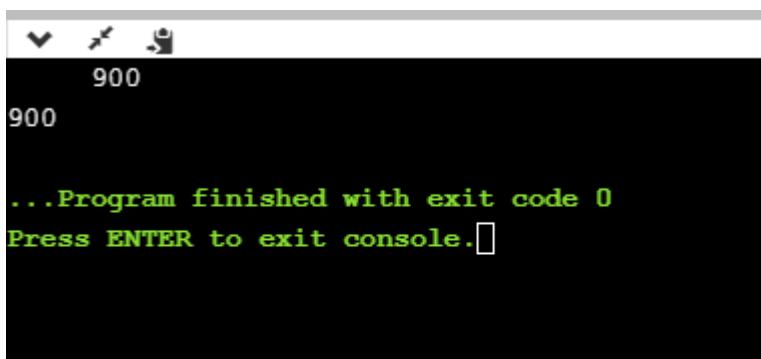
## Minimum Field Width Specifier

Suppose we want to display an output that occupies a minimum number of spaces on the screen. You can achieve this by displaying an integer number after the percent sign of the format specifier.

```
int main()
{
    int x=900;
    printf("%8d", x);
    printf("\n%-8d",x);
    return 0;
}
```

In the above program, %8d specifier displays the value after 8 spaces while %-8d specifier will make a value left-aligned.

Output

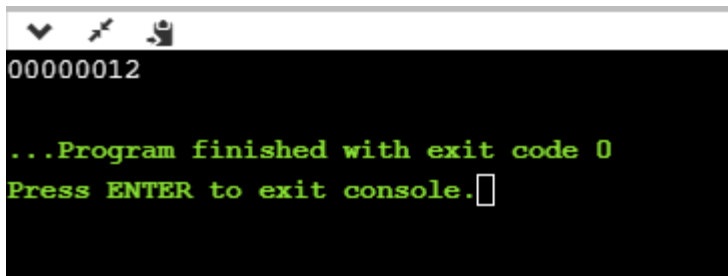
A terminal window with a black background and green text. The first line shows '900' preceded by 8 spaces. The second line shows '900'. The third line shows '...Program finished with exit code 0'. The fourth line shows 'Press ENTER to exit console.' followed by a cursor icon.

Now we will see how to fill the empty spaces. It is shown in the below code:

```
int main()
{
    int x=12;
    printf("%08d", x);
    return 0;
}
```

In the above program, %08d means that the empty space is filled with zeroes.

Output



```
00000012

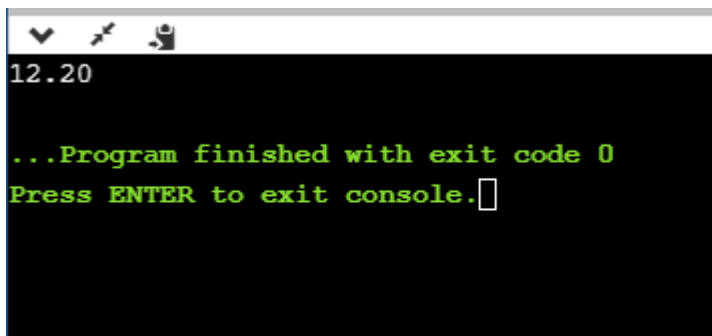
...Program finished with exit code 0
Press ENTER to exit console.
```

## Specifying Precision

We can specify the precision by using "." (Dot) operator which is followed by integer and format specifier.

```
int main()
{
    float x=12.2;
    printf("%.2f", x);
    return 0;
}
```

Output



```
12.20

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

## Escape Sequence in C

An *escape sequence* in the C programming language consists of a **backslash ()** and a character that stands in for a **special character** or **control sequence**. During the compilation process, the **C compiler** substitutes any escape sequences it comes across with the **relevant character** or **control sequence**. It enables the use of difficult-to-represent characters like newlines, tabs, quotations, and backslashes.

It is composed of two or more characters starting with backslash \. For example: \n represents new line.

# Regular Escape Sequences:

There are several escape sequence in C programming languages.

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

Now that we have a thorough understanding of each escape sequence in C,

## Alarm or Beep (\a):

The *alarm* or *beep escape sequence* (*a*) produces an audible alert or beep sound.

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

```
int main() {  
    printf("This is an alarm sound: \a");  
  
    return 0;  
}
```

Output:

```
This is an alarm sound:
```

## Backspace (\b):

The cursor can be advanced by one character with the *backspace escape key (b)*.

```
#include <stdio.h>

int main() {
    printf("Hello\b\b\bWorld!");

    return 0;
}
```

Output:

```
HelloWorld!
```

## Form Feed (\f):

The *form feed escape sequence (f)* is used to mimic a *page break* or advance to the next page.

```
#include <stdio.h>

int main() {
    printf("This is before the form feed.\fThis is after the form feed.");

    return 0;
}
```

Output:

```
This is before the form feed.
                        This is after the form feed.
```

## New Line (\n):

The *new line escape sequence (n)* is used to insert a newline character and move the cursor to the start of the following line.

```
#include <stdio.h>

int main() {
    printf("Line 1\nLine 2");

    return 0;
}
```

Output:

```
Line 1
Line 2
```

## Carriage Return (\r):

The *cursor can* be moved to the start of the current line by using the *carriage return escape sequence (r)*.

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

Output:

```
World!
```

## Tab (Horizontal) (\t):

The *tab escape sequence (t)* is used to insert a horizontal tab character and shift the cursor to the following tab stop.

```
#include <stdio.h>

int main() {
    printf("Name:\tJohn\tAge:\t25");

    return 0;
}
```

Output:

```
Name:   John   Age:   25
```

## Vertical Tab (\v):

The *vertical tab escape sequence (v)* is used to simulate a vertical tab or shift the mouse to the following vertical tab location.

```
#include <stdio.h>

int main() {
    printf("Hello\vWorld!");
    return 0;
}
```

Output:

```
Hello
World!
```

## Backslash (\):

A *backslash character* is inserted using the *backslash escape sequence (\)*.

```
#include <stdio.h>

int main() {
    printf("This is a backslash: \\Hello");
    return 0;
}
```

Output:

```
This is a backslash: \Hello
```

## Single Quote ('):

The *single quote escape sequence (')* is used to insert a single quote character.

```
#include <stdio.h>

int main() {
    printf("This is a single quote: \'Hello\'");
    return 0;
}
```

Output:

```
This is a single quote: 'Hello'
```

## Double Quote ("):

A *double quotation* character is inserted using the double *quote escape sequence* (`"`).

```
#include <stdio.h>

int main() {
    printf("This is a double quote: \"Hello\"");

    return 0;
}
```

Output:

```
This is a double quote: "Hello"
```

## Question Mark (?):

The *question mark escape sequence* (`?`) is used to insert a question mark character.

```
#include <stdio.h>

int main() {
    printf("This is a question mark: \?");

    return 0;
}
```

Output:

```
This is a question mark: ?
```

## Octal Number (\nnn):

The *character's octal value* can be inserted using the *octal number escape sequence* (`nnn`).

```
#include <stdio.h>

int main() {
    printf("This is an octal value: \101");

    return 0;
}
```

Output:

```
This is an octal value: A
```

## Hexadecimal Number (\xhh):

A *character's hexadecimal* value can be inserted using the *hexadecimal number escape sequence* (`xhh`).

```
#include <stdio.h>

int main() {
    printf("This is a hexadecimal value: \x41");

    return 0;
}
```

Output:

```
This is a hexadecimal value: A
```



## Null (\0):

The *null character*, denoted by "**0**", is inserted using the *null escape sequence (0)*.

```
#include <stdio.h>
int main() {
    char myString[] = "Hello\0World!";
    printf("String: %s", myString);
    return 0;
}
```

Output:

```
String: Hello
```

## ASCII value in C

### What is ASCII code?

The full form of ASCII is the **American Standard Code for information interchange**. It is a character encoding scheme used for electronics communication. Each character or a special character is represented by some ASCII code, and each ascii code occupies 7 bits in memory.

In C programming language, a character variable does not contain a character value itself rather the ascii value of the character variable. The ascii value represents the character variable in numbers, and each character variable is assigned with some number range from 0 to 127. For example, the ascii value of 'A' is 65.

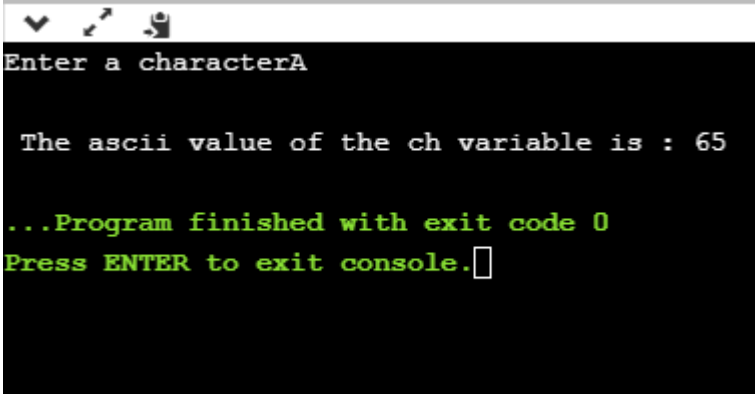
In the above example, we assign 'A' to the character variable whose ascii value is 65, so 65 will be stored in the character variable rather than 'A'.

Let's understand through an example.

We will create a program which will display the ascii value of the character variable.

```
#include <stdio.h>
int main()
{
    char ch; // variable declaration
    printf("Enter a character");
    scanf("%c",&ch); // user input
    printf("\n The ascii value of the ch variable is : %d", ch);
    return 0;
}
```

Output



```
Enter a characterA

The ascii value of the ch variable is : 65

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

Now, we will create a program which will display the ascii value of all the characters.

```
#include <stdio.h>
int main()
{
    int k; // variable declaration
    for(int k=0;k<=255;k++) // for loop from 0-255
    {
        printf("\nThe ascii value of %c is %d", k,k);
    }
    return 0;
}
```

## Constants in C

A **constant in C** is a value that doesn't change as the program runs. **Integers, floating-point numbers, characters, and strings** are just a few of the several types of constants that may be employed. When a constant has a value, it cannot be changed, unlike variables. They may be utilized in various operations and computations and serve as the program's representation of fixed values.

### Advantages of C Constants:

There are several advantages of C Constants. Some main advantages of C Constants are as follows:

### 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 Edureka" 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.

1. **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.

### Types of constant:

There are different types of Constants in C. Some of them are as follows:

#### Decimal Constant

A whole number represented in **base 10** is known as a **decimal constant**. It has digits that range from **0** to **9**. Declaring a **decimal constant** has a simple syntax that just requires the value to be written.

Example:

```
#include <stdio.h>

int main() {
    int decimal = 42;
    printf("The decimal constant is: %d\n", decimal);
    return 0;
}
```

Output:

```
The decimal constant is: 42
```

## Real or Floating-Point Constant:

A *fractional component* or *exponentiation* of a number is represented by a *real or floating-point constant*. It can be expressed with a decimal point, the letter "**E**", or the symbol "**e**" in exponential or decimal notation.

Example:

```
#include <stdio.h>

int main() {
    float real = 3.14;
    printf("The real constant is: %f\n", real);
    return 0;
}
```

Output:

```
The real constant is: 3.140000
```

## Octal Constant:

A *base 8* value is represented by an *octal constant*. It is prefixed with a '**0**' (**zero**) to show that it is an octal constant and has digits ranging from **0** to **7**.

Example:

```
#include <stdio.h>

int main() {
    int octal = 052; // Octal representation of decimal 42
    printf("The octal constant is: %o\n", octal);
    return 0;
}
```

Output:

```
The octal constant is: 52
```

## Hexadecimal Constant:

A *base-16* value is represented by a *hexadecimal constant*. It uses letters **A to F** (or **a to f**) and numbers **0 to 9** to represent values from **10** to **15**. It is prefixed with '**0x**' or '**0X**' to identify it as a hexadecimal constant.

Example:

```
#include <stdio.h>

int main() {
    int hexadecimal = 0x2A; // Hexadecimal representation of decimal 42
    printf("The hexadecimal constant is: %x\n", hexadecimal);
    return 0;
}
```

Output:

```
The hexadecimal constant is: 2a
```

## Character Constant

A *character constant* represents a *single character* that is enclosed in *single quotes*.

Example:

```
#include <stdio.h>

int main() {
    char character = 'A';
    printf("The character constant is: %c\n", character);
    return 0;
}
```

Output:

```
The character constant is: A
```

## String Constant:

A *series of characters* wrapped in *double quotes* is represented by a *string constant*. It is a character array that ends with the null character `\0`.

Example:

```
#include <stdio.h>

int main() {
    char string[] = "Hello, World!";
    printf("The string constant is: %s\n", string);
    return 0;
}
```

Output:

```
The string constant is: Hello, World!
```

## 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. It represents the value neither in fractional nor exponential part.

**It can be specified in the following three ways:**

### Decimal number (base 10)

It is defined by representing the digits between 0 to 9. For example, 45, 67, etc.

### Octal number (base 8)

It is defined as a number in which 0 is followed by digits such as 0,1,2,3,4,5,6,7. For example, 012, 034, 055, etc.

### Hexadecimal number (base 16)

It is defined as a number in which 0x or 0X is followed by the hexadecimal digits (i.e., digits from 0 to 9, alphabetical characters from (a-z) or (A-Z)).

**An integer literal is suffixed by following two sign qualifiers:**

**L or l:** It is a size qualifier that specifies the size of the integer type as long.

**U or u:** It is a sign qualifier that represents the type of the integer as unsigned. An unsigned qualifier contains only positive values.

**Note: The order of the qualifier is not considered, i.e., both *lu* and *ul* are the same.**

Let's look at a simple example of integer literal.

```
#include <stdio.h>
int main()
{
    const int a=23; // constant integer literal
    printf("Integer literal : %d", a);
    return 0;
}
```

Output

```
Integer literal : 23
```

### Float literal

It is a literal that contains only floating-point values or real numbers. These real numbers contain the number of parts such as integer part, real part, exponential part, and fractional part. The floating-point literal must be specified either in decimal or in exponential form. Let's understand these forms in brief.

### Decimal form

The decimal form must contain either decimal point, exponential part, or both. If it does not contain either of these, then the compiler will throw an error. The decimal notation can be prefixed either by '+' or '-' symbol that specifies the positive and negative numbers.

**Examples of float literal in decimal form are:**

1. 1.2, +9.0, -4.5

Let's see a simple example of float literal in decimal form.

```
#include <stdio.h>
int main()
{
    const float a=4.5; // constant float literal
    const float b=5.6; // constant float literal
    float sum;
    sum=a+b;
    printf("%f", sum);
    return 0;
}
```

## Output

```
10.100000
```

## Exponential form

The exponential form is useful when we want to represent the number, which is having a big magnitude. It contains two parts, i.e., mantissa and exponent. For example, the number is 2340000000000, and it can be expressed as 2.34e12 in an exponential form.

### Syntax of float literal in exponential form

1. `[+/-] <Mantissa> <e/E> [+/-] <Exponent>`

Examples of real literal in exponential notation are:

1. `+1e23, -9e2, +2e-25`

### Rules for creating an exponential notation

The following are the rules for creating a float literal in exponential notation:

- In exponential notation, the mantissa can be specified either in decimal or fractional form.
- An exponent can be written in both uppercase and lowercase, i.e., e and E.
- We can use both the signs, i.e., positive and negative, before the mantissa and exponent.
- Spaces are not allowed

## 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. Let's observe this scenario through an example.

```
#include <stdio.h>
int main()
{
    const char c='ak';
    printf("%c",c);
    return 0;
}
```

## 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= "Edureka";
```

```
String2= "family";
```

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

```
"Edureka " + "family"= Edureka family
```

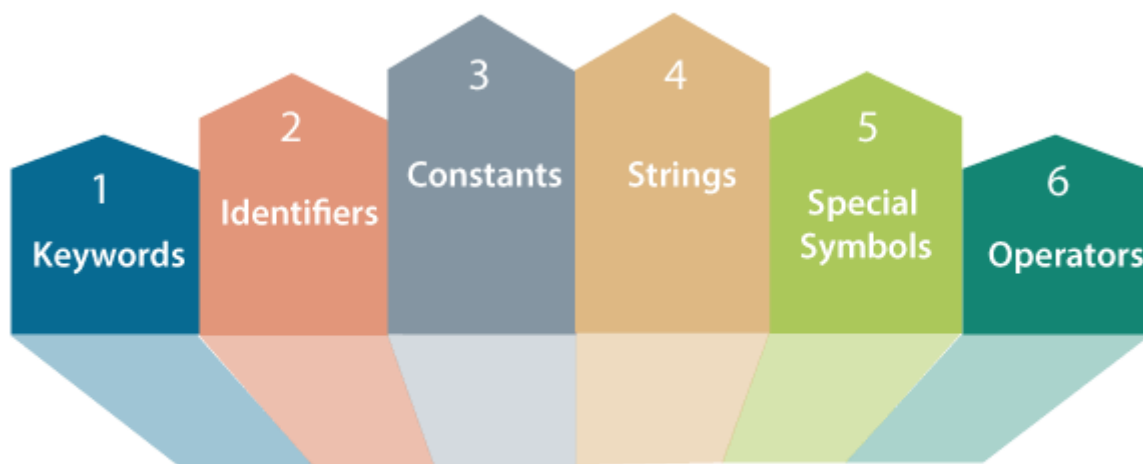
**Note: If we represent a single character, i.e., 'b', then this character will occupy a single byte as it is a character literal. And, if we represent the character within double quotes "b" then it will occupy more bytes as it is a string literal.**

# Tokens in C

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

## Classification of tokens in C

Tokens in C language can be divided into the following categories:



## Classification of C Tokens

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

# C Boolean

In C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false. Here, '0' represents false value, while '1' represents true value.

In C Boolean, '0' is stored as 0, and another integer is stored as 1. We do not require to use any header file to use the Boolean data type in C++, but in C, we have to use the header file, i.e., `stdbool.h`. If we do not use the header file, then the program will not compile.

## Syntax

1. **bool** variable\_name;

In the above syntax, **bool** is the data type of the variable, and **variable\_name** is the name of the variable.



Let's understand through an example.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    bool x=false; // variable initialization.
    if(x==true) // conditional statements
    {
        printf("The value of x is true");
    }
    else
        printf("The value of x is FALSE");
    return 0;
}
```

Output

```
The value of x is FALSE
```

## Boolean Array

Now, we create a bool type array. The Boolean array can contain either true or false value, and the values of the array can be accessed with the help of indexing.

Let's understand this scenario through an example.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    bool b[2]={true,false}; // Boolean type array
    for(int i=0;i<2;i++) // for loop
    {
        printf("%d",b[i]); // printf statement
    }
    return 0;
}
```

In the above code, we have declared a Boolean type array containing two values, i.e., true and false.

Output

```
1, 0,
```

## Boolean with Logical Operators

The Boolean type value is associated with logical operators. There are three types of logical operators in the C language:

**&&(AND Operator):** It is a logical operator that takes two operands. If the value of both the operands are true, then this operator returns true otherwise false

**||(OR Operator):** It is a logical operator that takes two operands. If the value of both the operands is false, then it returns false otherwise true.

**!(NOT Operator):** It is a NOT operator that takes one operand. If the value of the operand is false, then it returns true, and if the value of the operand is true, then it returns false.

Let's understand through an example.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    bool x=false;
    bool y=true;
    printf("The value of x&&y is %d", x&&y);
    printf("\nThe value of x||y is %d", x||y);
    printf("\nThe value of !x is %d", !x);
}
```

Output

```
The value of x&&y is 0
The value of x||y is 1
The value of !x is 1
```

## Static in C

Static is a keyword used in C programming language. It can be used with both variables and functions, i.e., we can declare a static variable and static function as well. An ordinary variable is limited to the scope in which it is defined, while the scope of the static variable is throughout the program.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    printf("%d",func());
    printf("\n%d",func());
    return 0;
}
int func()
{
    int count=0; // variable initialization
    count++; // incrementing counter variable

    return count;
}
```

Output

```
1
1
```

## Static variable

A static variable is a variable that persists its value across the various function calls.

Syntax

The syntax of a static variable is given below:

1. **static** data\_type variable\_name;

Let's look at a simple example of static variable.

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

    return 0;
}
int func()
{
    static int count=0;
    count++;
    return count;
}
```

Output

```
1
2
```

## Programming Errors in C

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

- Syntax error
- Run-time error
- Linker error
- Logical error
- Semantic error

Types of  
errors

1

Syntax error

2

Run-time error

3

Linker error

4

Logical error

5

Semantic error

# Conditional Operator in C

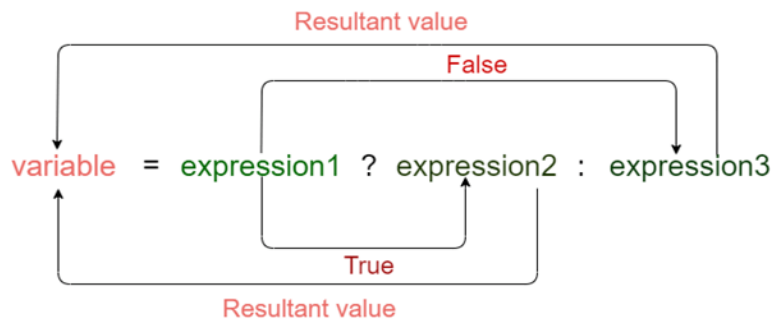
The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.  
As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement.

## Syntax of a conditional operator

1. Expression1? expression2: expression3;

The pictorial representation of the above syntax is shown below:



Meaning of the above syntax.

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

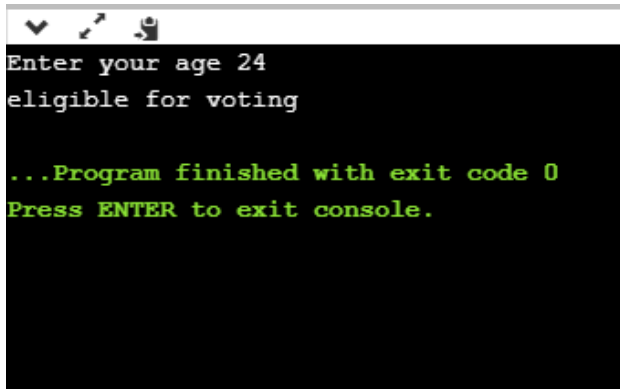
Let's understand the ternary or conditional operator through an example.

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

```
Enter your age 12
not eligible for voting

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

If we provide the age of user above 18, then the output would be:



```
Enter your age 24
eligible for voting

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

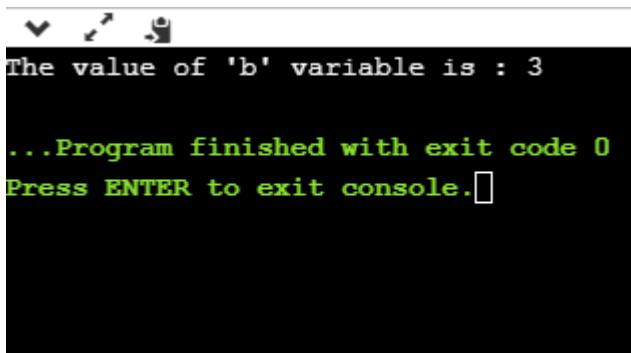
As we can observe from the above two outputs that if the condition is true, then the statement1 is executed; otherwise, statement2 will be executed.

Till now, we have observed that how conditional operator checks the condition and based on condition, it executes the statements. Now, we will see how a conditional operator is used to assign the value to a variable.

Let's understand this scenario through an example.

```
#include <stdio.h>
int main()
{
    int a=5,b; // variable declaration
    b=((a==5)?(3):(2)); // conditional operator
    printf("The value of 'b' variable is : %d",b);
    return 0;
}
```

Output



```
The value of 'b' variable is : 3

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

## Bitwise Operator in C

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

We have different types of bitwise operators in the C programming language. The following is the list of the bitwise operators:

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

Let's look at the truth table of the bitwise operators.

X	Y	X&Y	X Y	X^Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

## Bitwise AND operator

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

For example,

1. We have two variables a and b.
2. a = 6;
3. b = 4;
4. The binary representation of the above two variables are given below:
5. a = 0110
6. b = 0100
7. When we apply the bitwise AND operation in the above two variables, i.e., a&b, the output would be:
8. Result = 0100

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

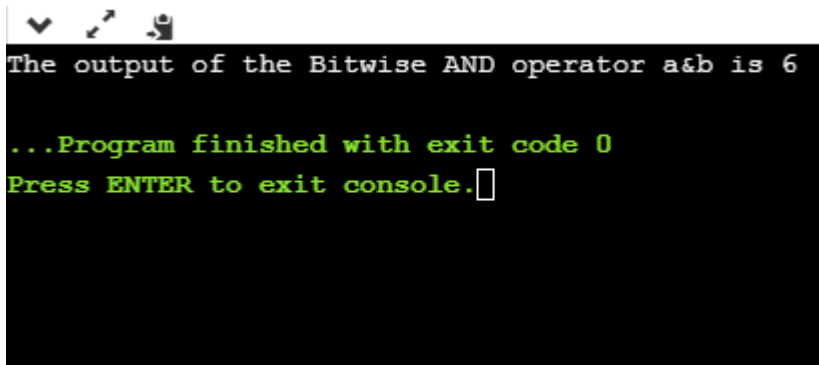
Let's understand the bitwise AND operator through the program.

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

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

**a AND b = 0110 && 1110 = 0110**

Output



## Bitwise OR operator

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

For example,

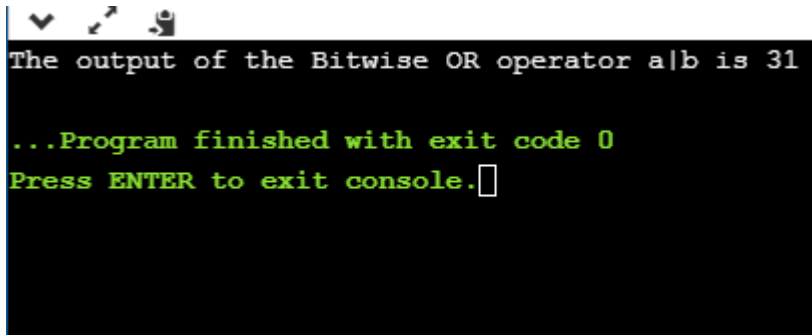
1. We consider two variables,
2. a = 23;
3. b = 10;
4. The binary representation of the above two variables would be:
5. a = 0001 0111
6. b = 0000 1010
7. When we apply the bitwise OR operator in the above two variables, i.e., a|b , then the output would be:
8. Result = 0001 1111

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

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

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

Output



## Bitwise exclusive OR operator

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

For example,

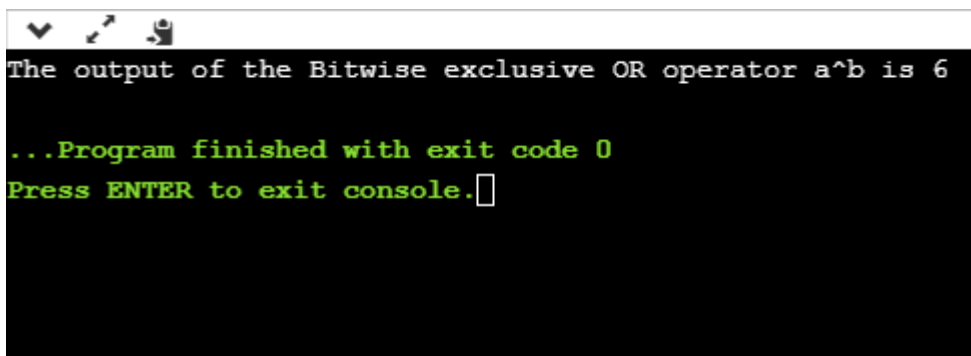
1. We consider two variables a and b,
2. a = 12;
3. b = 10;
4. The binary representation of the above two variables would be:
5. a = 0000 1100
6. b = 0000 1010
7. When we apply the bitwise exclusive OR operator in the above two variables ( $a \wedge b$ ), then the result would be:
8. Result = 0000 1110

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

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

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

Output



## Bitwise complement operator

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

For example,



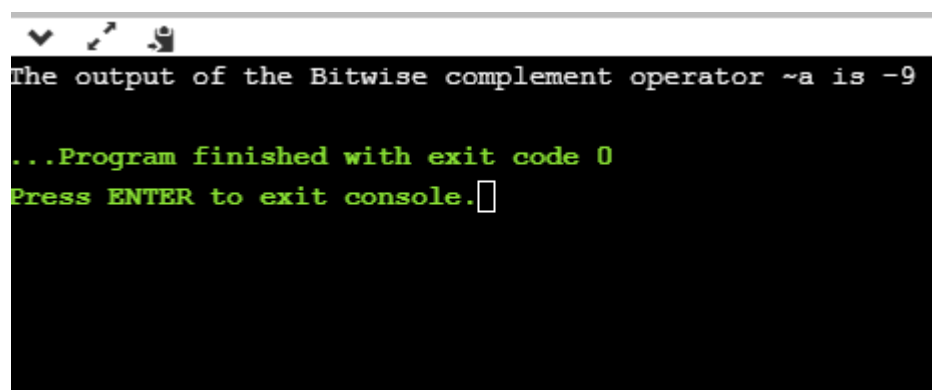
1. If we have a variable named 'a',
2. a = 8;
3. The binary representation of the above variable is given below:
4. a = 1000
5. When we apply the bitwise complement operator to the operand, then the output would be:
6. Result = 0111

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

Let's understand the complement operator through a program.

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

Output



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

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

## Bitwise shift operators

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

- Left-shift operator
- Right-shift operator

### Left-shift operator

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

Syntax of the left-shift operator is given below:

1. Operand << n

Where,

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

n is the number of bits to be shifted.

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

For example,

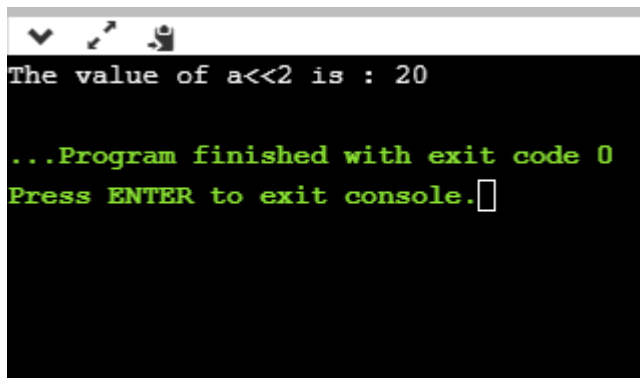
1. Suppose we have a statement:
2. `int a = 5;`
3. The binary representation of 'a' is given below:
4. a = 0101
5. If we want to left-shift the above representation by 2, then the statement would be:

6.  $a \ll 2$ ;
7.  $0101 \ll 2 = 00010100$

Let's understand through a program.

```
#include <stdio.h>
int main()
{
    int a=5; // variable initialization
    printf("The value of a<<2 is : %d ", a<<2);
    return 0;
}
```

Output



### Right-shift operator

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

Syntax of the right-shift operator is given below:

1. Operand  $\gg$  n;

Where,

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

N is the number of bits to be shifted.

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

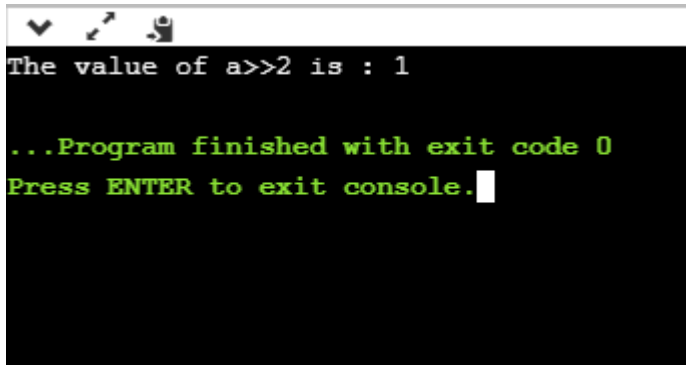
For example,

1. Suppose we have a statement,
2. `int a = 7;`
3. The binary representation of the above variable would be:
4.  $a = 0111$
5. If we want to right-shift the above representation by 2, then the statement would be:
6.  $a \gg 2$ ;
7.  $0000\ 0111 \gg 2 = 0000\ 0001$

Let's understand through a program.

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

## Output



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

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

## 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.

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

### Flowchart of if statement in C

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

```
#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. The syntax of the if-else statement is given below.

1. **if**(expression){
2. //code to be executed if condition is true
3. }**else**{
4. //code to be executed if condition is false
5. }

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

1. **#include**<stdio.h>
2. **int** main(){
3. **int** number=0;
4. printf("enter a number:");
5. scanf("%d",&number);
6. **if**(number%2==0){
7. printf("%d is even number",number);
8. }
9. **else**{
10. printf("%d is odd number",number);
11. }
12. **return** 0;
13. }

## 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.

1. `if(condition1){`
2. `//code to be executed if condition1 is true`
3. `}else if(condition2){`
4. `//code to be executed if condition2 is true`
5. `}`
6. `else if(condition3){`
7. `//code to be executed if condition3 is true`
8. `}`
9. `...`
10. `else{`
11. `//code to be executed if all the conditions are false`
12. `}`

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 ...
```

## 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 possible values of a single variable called switch variable. Here, We can define various statements in the multiple cases for the different values of a single variable.

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

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

default:
    code to be executed if all cases are not matched;
}
```

## Rules for switch statement in C language

1. The *switch expression* must be of an integer or character type.
2. The *case value* must be an integer or character constant.
3. The *case value* can be used only inside the switch statement.
4. The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

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

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

### C Program:

```
#include <stdio.h>

int main() {
    int num = 2;
    switch (num) {
        case 1:
            printf("Value is 1\n");
            break;
        case 2:
            printf("Value is 2\n");
            break;
        case 3:
            printf("Value is 3\n");
            break;
        default:
            printf("Value is not 1, 2, or 3\n");
            break;
    }
    return 0;
}
```

### Output

```
Value is 2
```

## Example of a switch statement in C

Let us see a simple example of a 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
```

## Break and Default keyword in Switch statement

Let us explain and define the *"break"* and *"default"* keywords in the context of the switch statement, along with example code and output.

### 1. Break Keyword:

The *"break"* keyword is used within the code block of each case to terminate the switch statement prematurely.

Example:

```
#include <stdio.h>
int main() {
    int num = 3;

    switch (num) {
        case 1:
            printf("Value is 1\n");
            break; // Exit the switch statement after executing this case block
        case 2:
            printf("Value is 2\n");
            break; // Exit the switch statement after executing this case block
        case 3:
            printf("Value is 3\n");
            break; // Exit the switch statement after executing this case block
        default:
            printf("Value is not 1, 2, or 3\n");
            break; // Exit the switch statement after executing the default case block
    }
    return 0;
}
```

Output

```
Value is 3
```

## 2. Default Keyword:

When none of the *case constants* match the *evaluated expression*, it operates as a *catch-all case*. If no matching case exists and a *"default" case exists*, the code block associated with the *"default"* case is run.

### Example:

Let's take a program to understand the use of the *default keyword* in C.

```
#include <stdio.h>
int main() {
    int num = 5;

    switch (num) {
        case 1:
            printf("Value is 1\n");
            break;
        case 2:
            printf("Value is 2\n");
            break;
        case 3:
            printf("Value is 3\n");
            break;
        default:
            printf("Value is not 1, 2, or 3\n");
            break;
    }

    return 0;
}
```

### Output

```
Value is not 1, 2, or 3
```

## C Switch statement is fall-through

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

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

```
#include<stdio.h>
int main(){
    int number=0;

    printf("enter a number:");
    scanf("%d",&number);

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

## Output

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

## Output

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

## Nested switch case statement

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

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

    int i = 10;
    int j = 20;

    switch(i) {

        case 10:
            printf("the value of i evaluated in outer switch: %d\n",i);
        case 20:
            switch(j) {
                case 20:
                    printf("The value of j evaluated in nested switch: %d\n",j);
            }
    }

    printf("Exact value of i is : %d\n", i);
    printf("Exact value of j is : %d\n", j);

    return 0;
}
```

## Output

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

## if-else vs switch

Let's summarize the above differences in a tabular form.

If-else	switch
---------	--------

<b>Definition</b>	Depending on the condition in the 'if' statement, 'if' and 'else' blocks are executed.	The user will decide which statement is to be executed.
<b>Expression</b>	It contains either logical or equality expression.	It contains a single expression which can be either a character or integer variable.
<b>Evaluation</b>	It evaluates all types of data, such as integer, floating-point, character or Boolean.	It evaluates either an integer, or character.
<b>Sequence of execution</b>	First, the condition is checked. If the condition is true then 'if' block is executed otherwise 'else' block	It executes one case after another till the break keyword is not found, or the default statement is executed.
<b>Default execution</b>	If the condition is not true, then by default, else block will be executed.	If the value does not match with any case, then by default, default statement is executed.
<b>Editing</b>	Editing is not easy in the 'if-else' statement.	Cases in a switch statement are easy to maintain and modify. Therefore, we can say that the removal or editing of any case will not interrupt the execution of other cases.
<b>Speed</b>	If there are multiple choices implemented through 'if-else', then the speed of the execution will be slow.	If we have multiple choices then the switch statement is the best option as the speed of the execution will be much higher than 'if-else'.

## C Loops

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

### 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:

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

## while loop in C

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

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

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

## for loop in C

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

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

1. `for(initialization;condition;incr/decr){`
2. `//code to be executed`
3. `}`

## do while loop in C

A **loop** is a programming control structure that allows you to execute a **block of code** indefinitely if a specific condition is met. Loops are used to execute repeating activities and boost programming performance. There are multiple loops in the C programming language, one of which is the **"do-while" loop**.

A **"do-while" loop** is a form of a **loop** in C that executes the code block first, followed by the condition. If the condition is **true**, the **loop** continues to run; else, it stops. However, whether the condition is originally **true**, it ensures that the code block is performed at least once.

## do while loop syntax

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

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

The components are divided into the following:

- The **do keyword** marks the beginning of the Loop.

- The **code block** within **curly braces {}** is the body of the loop, which contains the code you want to repeat.
- The **while keyword** is followed by a condition enclosed in parentheses (). After the code block has been run, this condition is verified. If the condition is **true**, the loop continues else, the **loop ends**.

## Working of do while Loop in C

Let us look at an example of how a **do-while loop** works in C. In this example, we will write a simple program that questions the user for a **password** and keeps asking until the right password is input.

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char password[] = "secret";
    char input[20];
    do {
        printf("Enter the password: ");
        scanf("%s", input);
    } while (strcmp(input, password) != 0);
    printf("Access granted!\n");
    return 0;
}
```

Output:

Let us walk through a possible scenario:

```
Enter the password: 123
Enter the password: abc
Enter the password: secret
Access Granted!
```

Explanation:

In this example, the user initially enters the wrong passwords, **"123"** and **"abc"**. The loop prompts the user until the correct password **"secret"** is entered. Once the correct password is provided, the loop terminates, and the **"Access granted!"** message is displayed.

## Example of do while loop in C:

Example 1:

Here is a simple example of a **"do-while" loop** in C that prints numbers from 1 to 5:

```
#include <stdio.h>
int main() {
    inti = 1;
    do {
        printf("%d\n", i);
        i++;
    } while (i<= 5);
    return 0;
}
```

Output:

```
1
2
3
4
5
```

## Example 2:

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

```
#include<stdio.h>
intmain(){
    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
```

## Example 3:

Let's take a program that prints the multiplication table of a given number N using a **do...while Loop**:

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

Output:

Let us say you enter the number 7 as input:

```
Please enter a number to generate its multiplication table: 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
```

```
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

The program calculates and prints the multiplication table for **7** from 1 to 10.

## Infinite do while loop

An *infinite loop* is a loop that runs indefinitely as its condition is always *true* or it lacks a terminating condition. Here is an example of an *infinite do...while loop* in C:

Example:

```
#include <stdio.h>
int main() {
    int i = 1;
    do {
        printf("Iteration %d\n", i);
        i++;
    } while (1); // Condition is always true

    return 0;
}
```

Output:

When you run the program, you will see that it continues printing "*Iteration x*", where x is the *iteration number* without stopping:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
... (and so on)
```

To interrupt an infinite loop like this, you generally use a *break statement* within the *loop* or some external condition you can control, such as *hitting* a specific key combination. In most desktop settings, the keyboard shortcut **Ctrl+C** can escape the Loop.

## Nested do while loop in C

In C, we take an example of a *nested do...while loop*. In this example, we will write a program that uses *nested do...while loops* to create a numerical pattern.

Example:

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



In this program, we use **nested do...while loops** to generate a pattern of numbers. The **outer loop** controls the number of rows, and the **inner loop** generates the numbers for each row.

### Output:

Let us say you input five as the number of rows:

```
Enter the number of rows: 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

### Explanation:

In this example, the program generates a pattern of numbers in a **triangular shape**. The **outer loop** iterates over the rows, and the **inner loop** iterates within each row, printing the numbers from 1 up to the current row number.

## Difference between while and do while Loop

Here is a tabular comparison between the while loop and the do-while Loop in C:

Aspect	while loop	do-while loop
Syntax	while (condition) { ... }	do { ... } while (condition);
Loop Body Execution	Condition is checked before execution.	The body is executed before the condition.
First Execution	The condition must be true initially.	The body is executed at least once.
Loop Execution	May execute zero or more times.	Will execute at least once.
Example	while (i< 5) { printf("%d\n", i); i++; }	do { printf("%d\n", i); i++; } while (i< 5);
Common Use Cases	When the loop may not run at all.	When you want the loop to run at least once.

## while loop in C

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

### Syntax of while loop in C language

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

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

## 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
```

### 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 Aisect");
    }
}
```

### Output

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

### Example 3

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

### Output

```
infinite loop
```

## Infinitive while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```
while(1){
    //statement
}
```

# for loop in C

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

## Syntax of for loop in C

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

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

## 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 1

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

### output

0 1 2 3 4

## Example 2

```
#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
```

## Example 3

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

## Output

```
infinite loop
```

## Example 1

```
#include<stdio.h>
void main ()
{
    int i=0,j=2;
    for(i = 0;i<5;i++,j=j+2)
    {
        printf("%d %d\n",i,j);
    }
}
```

## Output

```
0 2
1 4
2 6
3 8
4 10
```

## Loop body

The braces {} are used to define the scope of the loop. However, if the loop contains only one statement, then we don't need to use braces. A loop without a body is possible. The braces work as a block separator, i.e., the value variable declared inside for loop is valid only for that block and not outside. Consider the following example.

```
#include<stdio.h>
void main ()
{
    int i;
    for(i=0;i<10;i++)
    {
        int i = 20;
        printf("%d ",i);
    }
}
```

Output

```
20 20 20 20 20 20 20 20 20 20
```

## Infinite for loop in C

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

```
#include<stdio.h>
void main ()
{
    for(;;)
    {
        printf("welcome to Aisect");
    }
}
```

If you run this program, you will see above statement infinite times.

## 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

1. Outer\_loop
2. {
3. Inner\_loop
4. {
5. // inner loop statements.
6. }
7. // outer loop statements.
8. }

**Outer\_loop** and **Inner\_loop** are the valid loops that can be a 'for' loop, 'while' loop or 'do-while' loop.

### Nested for loop

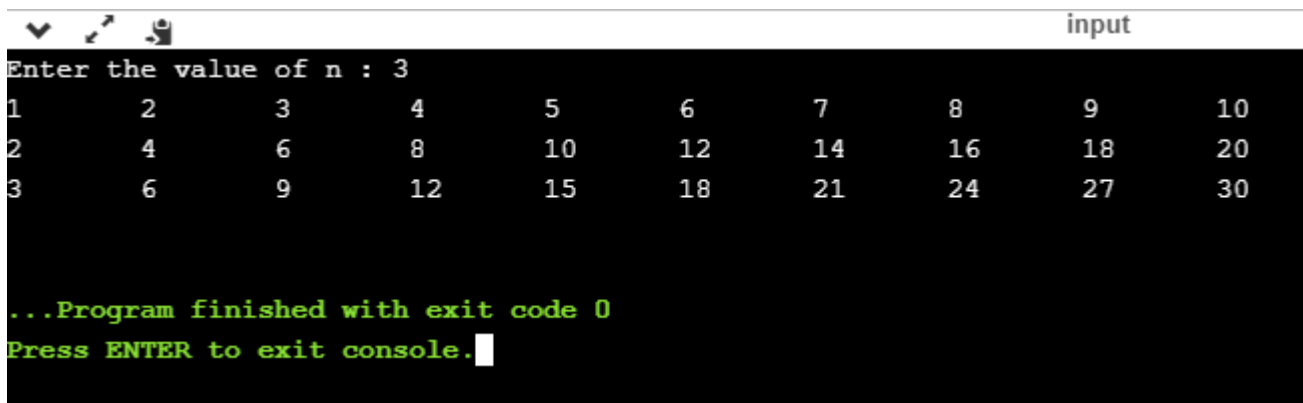
The nested for loop means any type of loop which is defined inside the 'for' loop.

```
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");
    }
}
```

Output:



```
input
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.

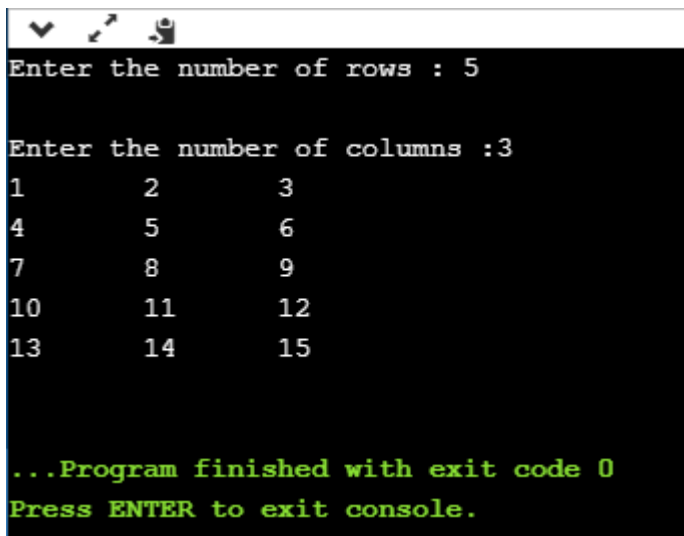
1. while(condition)
2. {
3. while(condition)
4. {
5. // inner loop statements.
6. }
7. // outer loop statements.
8. }



## 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");
    }
}
```

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

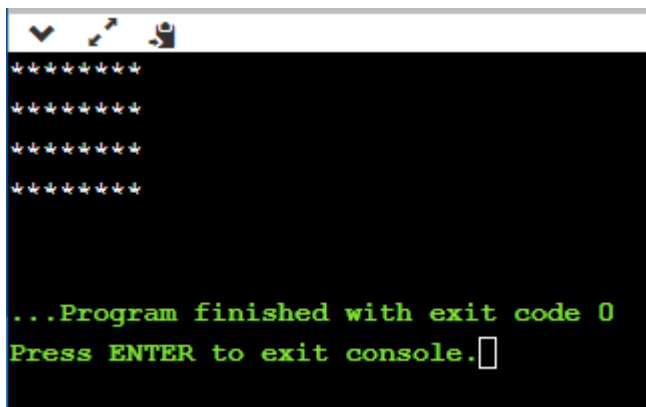
The nested do..while loop means any type of loop which is defined inside the 'do..while' loop.

1. **do**
2. {
3. **do**
4. {
5. // inner loop statements.
6. } **while**(condition);
7. // outer loop statements.
8. } **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:



## 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.

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:

```
1. for(;;)
2. {
3.     // body of the for loop.
4. }
```

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 Aisect");
    }
    return 0;
}
```

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

## 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:

```
1. while(1)
2. {
3.     // body of the loop..
4. }
```

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.

## 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.

1. **do**
2. {
3.     // body of the loop..
4. }**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.

## goto statement

We can also use the goto statement to define the infinite loop.

1. infinite\_loop;
2. // body statements.
3. **goto** infinite\_loop;

In the above code, the goto statement transfers the control to the infinite loop.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    char ch;
    while(1)
    {
        ch=getchar();
        if(ch=='\n')
        {
            break;
        }
        printf("hello");
    }
    return 0;
}
```

## Unintentional infinite loops

Sometimes the situation arises where unintentional infinite loops occur due to the bug in the code. If we are the beginners, then it becomes very difficult to trace them. Below are some measures to trace an unintentional infinite loop:

- We should examine the semicolons carefully. Sometimes we put the semicolon at the wrong place, which leads to the infinite loop.

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

In the above code, we put the semicolon after the condition of the while loop which leads to the infinite loop. Due to this semicolon, the internal body of the while loop will not execute.

- We should check the logical conditions carefully. Sometimes by mistake, we place the assignment operator (=) instead of a relational operator (==).

```
#include <stdio.h>
int main()
{
    char ch='n';
    while(ch='y')
    {
        printf("hello");
    }
    return 0;
}
```

In the above code, we use the assignment operator (ch='y') which leads to the execution of loop infinite number of times.

- We use the wrong loop condition which causes the loop to be executed indefinitely.

```
#include <stdio.h>
int main()
{
    for(int i=1;i>=1;i++)
    {
        printf("hello");
    }
    return 0;
}
```

The above code will execute the 'for loop' infinite number of times. As we put the condition (i>=1), which will always be true for every condition, it means that "hello" will be printed infinitely.

- We should be careful when we are using the **break** keyword in the nested loop because it will terminate the execution of the nearest loop, not the entire loop.

```
#include <stdio.h>
int main()
{
    while(1)
    {
        for(int i=1;i<=10;i++)
        {
            if(i%2==0)
            {
                break;
            }
        }
    }
    return 0;
}
```

In the above code, the while loop will be executed an infinite number of times as we use the break keyword in an inner loop. This break keyword will bring the control out of the inner loop, not from the outer loop.

- We should be very careful when we are using the floating-point value inside the loop as we cannot underestimate the floating-point errors.

```

#include <stdio.h>
int main()
{
    float x = 3.0;
    while (x != 4.0) {
        printf("x = %f\n", x);
        x += 0.1;
    }
    return 0;
}

```

## C 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:

1. //loop or switch case
2. break;

### 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
```

## 2. Nested Loops:

The **break statement** can be applied in **nested loops** to break out of the **inner** and **outer** loops simultaneously when a specific condition is met. It allows you to stop processing further iterations in **multiple loops** at once.

### Syntax:

It has the following syntax:

```
for (outer_loop_initialization; outer_loop_condition; outer_loop_increment) {  
    // Code block of the outer loop  
    for (inner_loop_initialization; inner_loop_condition; inner_loop_increment) {  
        // Code block of the inner loop  
        if (some_condition) {  
            break; // Exit both inner and outer loops if this condition is met  
        }  
        // Rest of the inner loop's code  
    }  
    // Rest of the outer loop's code  
}
```

### Example:

Let's take an example to understand the use of the **break statement** in **nested loops** in C:

```
// Using break in nested loops  
#include <stdio.h>  
int main() {  
    for (inti = 1; i<= 3; i++) {  
        for (int j = 1; j <= 3; j++) {  
            if (i == 2 && j == 2) {  
                break; // Exit both loops when i=2 and j=2  
            }  
            printf("(%d, %d) ", i, j);  
        }  
        printf("\n");  
        return 0;  
    }  
}
```

### Output

```
(1, 1) (1, 2) (1, 3)
```

## 3. Infinite Loops:

An **infinite loop** runs continuously unless terminated by a **"break" statement** or another condition within the **loop**. In **infinite loops**, the **"break" statement** is typically used to give a means to leave the loop based on specified criteria.

### Syntax:

It has the following syntax:

```
while (1) { // Infinite loop using a true condition  
    // Code block inside the loop  
    if (some_condition) {  
        break; // Exit the loop if this condition is met  
    }  
    // Rest of the loop's code  
}
```

### Example:

Let's take an example to understand the use of the **break statement** in *infinite loops* in C:

```
// Using break in an infinite loop
#include <stdio.h>
int main() {
    int number;
    while (1) {
        printf("Enter a number (0 to exit): ");
        scanf("%d", &number);
        if (number == 0) {
            break; // Exit the loop when the user enters 0
        }
        printf("You entered: %d\n", number);
    }
    return 0;
}
```

### Output

```
Enter a number (0 to exit): 7
You entered: 7
Enter a number (0 to exit): 5
You entered: 5
Enter a number (0 to exit): 0
```

## 4. Switch Case:

The **"break" statement** is used in a **"switch" statement** to exit the switch block after a particular case is executed. Without the **"break" statement**, the program would continue executing the code for all the subsequent cases, potentially leading to unexpected behavior.

### Syntax:

It has the following syntax:

```
switch (expression) {
    case value1:
        // Code block for case value1
        break; // Exit the switch block after executing this case
    case value2:
        // Code block for case value2
        break; // Exit the switch block after executing this case
        // More cases...
    default:
        // Code block for the default case
        break; // Exit the switch block after executing this case
}
```

### Example:

Let's take an example to understand the use of the **break statement** in the **switch case** in C:

```
// Using break in a switch statement
#include <stdio.h>
int main() {
    int choice;
    printf("Menu:\n");
    printf("1. Option 1\n");
    printf("2. Option 2\n");
    printf("3. Quit\n");
    printf("Enter your choice: ");
}
```



```

scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("You selected Option 1\n");
        break;
    case 2:
        printf("You selected Option 2\n");
        break;
    case 3:
        printf("Quitting the menu...\n");
        break;
    default:
        printf("Invalid choice. Try one more time.\n");
        break;
}
return 0;
}

```

#### Output (Example 1):

```

Menu:
Option 1
Option 2
Quit
Enter your choice: 2
You selected Option 2

```

#### Output (Example 2):

```

Menu:
Option 1
Option 2
Quit
Enter your choice: 4
Invalid choice. Try one more time.

```

## 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:

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

### Continue statement example 1

```

#include<stdio.h>
void main ()
{
    int i = 0;
    while(i!=10)
    {
        printf("%d", i);
        continue;
        i++;
    }
}

```

## Output

infinite loop

## 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 continue statement with inner loop

In such case, C continue statement continues only inner loop, but not outer loop.

```
#include<stdio.h>
int main(){
    int i=1,j=1;//initializing a local variable
    for(i=1;i<=3;i++){
        for(j=1;j<=3;j++){
            if(i==2 && j==2){
                continue;//will continue loop of j only
            }
            printf("%d %d\n",i,j);
        }
    }//end of for loop
    return 0;
}
```

## Output

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

As you can see, 2 2 is not printed on the console because inner loop is continued at i==2 and j==2.

# 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. Syntax:

1. label:
2. `//some part of the code;`
3. `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
```

## When should we use goto?

The only condition in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time. Consider the following example.

```
#include <stdio.h>
int main()
{
    int i, j, k;
    for(i=0;i<10;i++)
    {
        for(j=0;j<5;j++)
        {
            for(k=0;k<3;k++)
            {
                printf("%d %d %d\n",i,j,k);
                if(j == 3)
                {
                    goto out;
                }
            }
        }
    }
}
```

```

        out:
        printf("came out of the loop");
    }
0 0 0
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
0 3 0
came out of the loop

```

## Type Casting in C

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:

1. (type)value;

**Note: It is always recommended to convert the lower value to higher for avoiding data loss.**

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
```

# Types of Type Casting

Type casting may be done in several ways in C, including implicit and explicit casting.

## Implicit Casting:

When the compiler **automatically transforms** the data from one type to another, it is referred to as **implicit casting** or **automated type conversion**.

**Implicit casting** doesn't require any special syntax because the compiler takes care of it.

**Example:**

```
#include <stdio.h>

int main() {
    int num1 = 10;
    float num2 = 5.5;
    float result;
    result = num1 + num2; // Implicit cast from int to float
    printf("The result is: %f\n", result);
    return 0;
}
```

**Output:**

The result is: 15.500000

**Explanation:**

In the example above, the **integer value num1** is added to the **float** variable **num2** after being implicitly cast to a float. The **result** variable contains a **float** value representing the outcome.

## Explicit Casting:

Using the **cast operator, explicit casting** entails explicitly changing one data type to another. It needs explicit instructions in the code and allows the programmer control over type conversion.

**Syntax:**

1. (type) expression

Here, **expression** is the value or variable that is to be **cast**, and **(type)** stands for the required **data type**.

### Example:

```
#include <stdio.h>

int main() {

    float num1 = 15.6;

    int num2;

    num2 = (int) num1; // Explicit cast from float to int

    printf("The result is: %d\n", num2);

    return 0;

}
```

### Output:

The result is: 15

**Explanation:** This example uses the *(int)* syntax to explicitly cast the **float** variable **num1** to an integer. After truncating the **fractional portion**, we get the integer value and that value is assigned to the **num2 variable**.

## Narrowing Conversion:

When a **value** is converted to a **data type** with a **narrower range** or **precision**, it is said to be **narrowing transformed**, which may result in data loss. If it is not handled appropriately, it may lead to unexpected behavior and needs explicit casting.

### Example:

```
#include <stdio.h>

int main() {

    double num1 = 1234.56789;

    int num2;

    num2 = (int) num1; // Narrowing conversion from double to int

    printf("The result is: %d\n", num2);

    return 0;

}
```

### Output:

The result is: 1234

### Explanation:

This example **narrows** the conversion by explicitly casting the **double value num1** to an **integer**. The **integer value** is sent to the **num2 variable**, and the fractional portion is eliminated.

## Widening Conversion:

When a value is converted to a **data type** with a wider range or greater accuracy, this process is called **widening conversion**. Since it occurs implicitly, casting is not necessary.

### Example:

```
#include <stdio.h>

int main() {
    int num1 = 10;
    double num2;
    num2 = num1; // Widening conversion from int to double
    printf("The result is: %f\n", num2);
    return 0;
}
```

### Output:

The result is: 10.000000

### Explanation:

In this example, the **assignment** to the **num2 variable** results in an **implicit conversion** of the **integer value num1** to a **double**. A **double** with the same numerical value as the initial integer is the outcome.

# C Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

## Advantage of functions in C

There are the following advantages of C functions.

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

## Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.



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

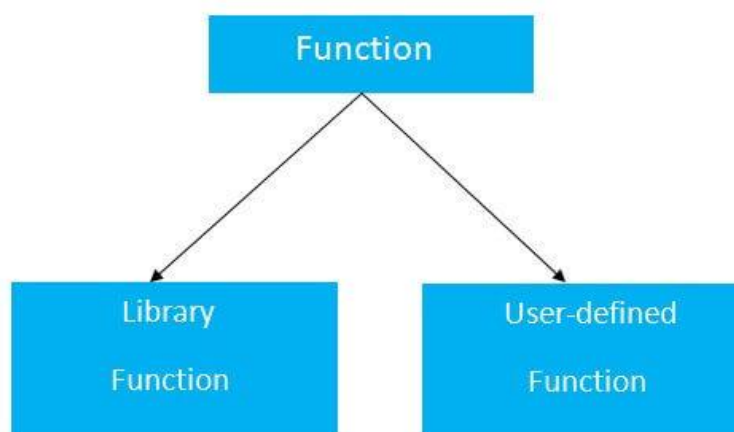
The syntax of creating function in c language is given below:

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.

```
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

```
#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

## Example for Function without argument and with return value

### Example 1

```
#include<stdio.h>

int sum();

void main()
{
    int result;

    printf("\nGoing to calculate the sum of two numbers:");

    result = sum();

    printf("%d",result);
}

int sum()
{
    int a,b;

    printf("\nEnter two numbers");

    scanf("%d %d",&a,&b);

    return a+b;
}
```

## Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

### Example 2: program to calculate the area of the square

```
#include<stdio.h>

int sum();

void main()

{

    printf("Going to calculate the area of the square\n");

    float area = square();

    printf("The area of the square: %f\n",area);

}

int square()

{

    float side;

    printf("Enter the length of the side in meters: ");

    scanf("%f",&side);

    return side * side;

}
```

### Output

Going to calculate the area of the square

Enter the length of the side in meters: 10

The area of the square: 100.000000

## Example for Function with argument and without return value

### Example 1

```
#include<stdio.h>

void sum(int, int);

void main()
{
    int a,b,result;

    printf("\nGoing to calculate the sum of two numbers:");

    printf("\nEnter two numbers:");

    scanf("%d %d",&a,&b);

    sum(a,b);
}

void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

### Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

### Example 2: program to calculate the average of five numbers.

```
#include<stdio.h>

void average(int, int, int, int, int);

void main()
{
```

```

int a,b,c,d,e;

printf("\nGoing to calculate the average of five numbers:");

printf("\nEnter five numbers:");

scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);

average(a,b,c,d,e);

}

void average(int a, int b, int c, int d, int e)
{
    float avg;

    avg = (a+b+c+d+e)/5;

    printf("The average of given five numbers : %f",avg);

}

```

## Output

```

Going to calculate the average of five numbers:
Enter five numbers:10
20
30
40
50

The average of given five numbers : 30.000000

```

## Example for Function with argument and with return value

### Example 1

```

#include<stdio.h>

int sum(int, int);

void main()
{
    int a,b,result;

    printf("\nGoing to calculate the sum of two numbers:");

```

```

    printf("\nEnter two numbers:");

    scanf("%d %d",&a,&b);

    result = sum(a,b);

    printf("\nThe sum is : %d",result);
}

int sum(int a, int b)
{
    return a+b;
}

```

### Output

```

Going to calculate the sum of two numbers:
Enter two numbers:10
20
The sum is : 30

```

### Example 2: Program to check whether a number is even or odd

```

#include<stdio.h>

int even_odd(int);

void main()
{
    int n,flag=0;

    printf("\nGoing to check whether a number is even or odd");
    printf("\nEnter the number: ");
    scanf("%d",&n);

    flag = even_odd(n);

    if(flag == 0)
    {
        printf("\nThe number is odd");
    }
}

```



```
else
{
    printf("\nThe number is even");
}
}

int even_odd(int n)
{
    if(n%2 == 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

### Output

Going to check whether a number is even or odd

Enter the number: 100

The number is even

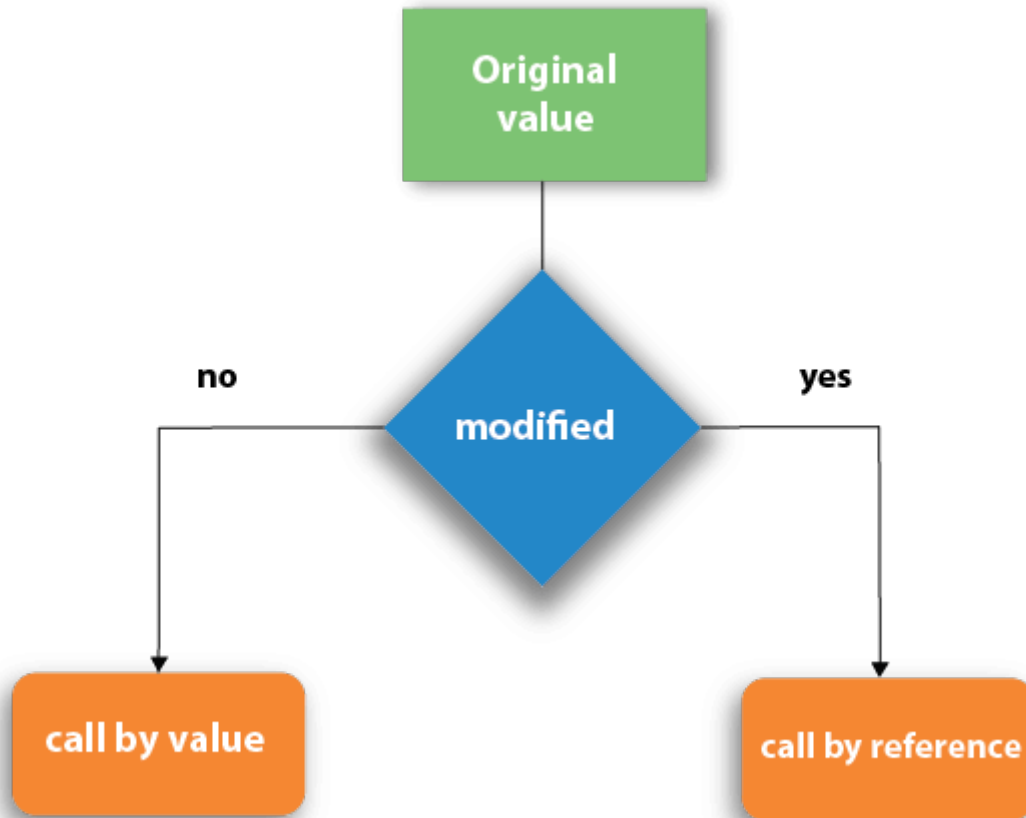
## C Library Functions

Library functions are the inbuilt functions in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. The list of mostly used header files is given in the following table.

S N	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

# Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



Let's understand call by value and call by reference in c language one by one.

---

## Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

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

```
#include<stdio.h>

void change(int num) {

    printf("Before adding value inside function num=%d \n",num);

    num=num+100;

    printf("After adding value inside function num=%d \n", num);

}

int main() {

    int x=100;

    printf("Before function call x=%d \n", x);

    change(x); //passing value in function

    printf("After function call x=%d \n", x);

    return 0;

}
```

### Output

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

---

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

```
#include <stdio.h>

void swap(int , int); //prototype of the function

int main()

{

    int a = 10;

    int b = 20;
```

```

    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a
    and b in main

    swap(a,b);

    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual
    parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
}

void swap (int a, int b)
{
    int temp;

    temp = a;

    a=b;

    b=temp;

    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a =
    20, b = 10
}

```

## Output

```

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

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

```

## Call by reference in C

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

Consider the following example for the call by reference.

```
#include<stdio.h>

void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}

int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x); //passing reference in function
    printf("After function call x=%d \n", x);
    return 0;
}
```

## Output

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

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

```
#include <stdio.h>

void swap(int *, int *); //prototype of the function

int main()
{
    int a = 10;
    int b = 20;

    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a
    and b in main
    swap(&a,&b);
}
```

```

    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual
parameters do change in call by reference, a = 10, b = 20
}

void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;

    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a
= 20, b = 10
}

```

## Output

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

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

## Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.

3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location
---	--	---

## Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls.

In the following example, recursion is used to calculate the factorial of a number.

```
#include <stdio.h>

int fact (int);

int main()
{
    int n,f;

    printf("Enter the number whose factorial you want to calculate?");

    scanf("%d",&n);

    f = fact(n);

    printf("factorial = %d",f);
}

int fact(int n)
{
    if (n==0)
    {
        return 0;
    }

    else if ( n == 1)
    {
```



```
    return 1;
}
else
{
    return n*fact(n-1);
}
}
```

### Output

Enter the number whose factorial you want to calculate?5

factorial = 120

We can understand the above program of the recursive method call by the figure given below:

```
return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1
```

$1 * 2 * 3 * 4 * 5 = 120$

**Fig: Recursion**

# C Array

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.

## 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 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.

1. `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.

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

80	60	70	85	75
----	----	----	----	----

marks[0]   marks[1]   marks[2]   marks[3]   marks[4]

### **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.

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

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

1. `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

## C Array Example: Sorting an array

In the following program, we are using bubble sort method to sort the array in ascending order.

```
#include<stdio.h>

void main ()
{
    int i, j, temp;

    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};

    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]);
    }
}
```

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

```
#include<stdio.h>

void main ()
{
    int arr[100],i,n,largest,sec_largest;
    printf("Enter the size of the array?");
    scanf("%d",&n);
    printf("Enter the elements of the array?");
    for(i = 0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0;i<n;i++)
    {
        if(arr[i]>largest)
        {
            sec_largest = largest;
            largest = arr[i];
        }
        else if (arr[i]>sec_largest && arr[i]!=largest)
        {
            sec_largest=arr[i];
        }
    }
    printf("largest = %d, second largest = %d",largest,sec_largest);

}
```

# 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.

## Declaration of two dimensional Array in C

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

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

Consider the following example.

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

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

## Initialization of 2D Array in C

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

1. `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
```

## C 2D array example: Storing elements in a matrix and printing it.

```
#include <stdio.h>

void main ()
{
    int arr[3][3],i,j;
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
```



```

        printf("Enter a[%d][%d]: ",i,j);

        scanf("%d",&arr[i][j]);

    }

}

printf("\n printing the elements ....\n");

for(i=0;i<3;i++)
{
    printf("\n");

    for (j=0;j<3;j++)
    {
        printf("%d\t",arr[i][j]);

    }

}

}

```

## Output

Enter a[0][0]: 56

Enter a[0][1]: 10

Enter a[0][2]: 30

Enter a[1][0]: 34

Enter a[1][1]: 21

Enter a[1][2]: 34

Enter a[2][0]: 45

Enter a[2][1]: 56

Enter a[2][2]: 78

printing the elements ....

56    10    30

34    21    34

45    56    78

# Return an Array in C

## What is an Array?

An array is a type of data structure that stores a fixed-size of a homogeneous collection of data. In short, we can say that array is a collection of variables of the same type.

For example, if we want to declare 'n' number of variables, n1, n2...n., if we create all these variables individually, then it becomes a very tedious task. In such a case, we create an array of variables having the same type. Each element of an array can be accessed using an index of the element.

Let's first see how to pass a single-dimensional array to a function.

Passing array to a function

```
#include <stdio.h>

void getarray(int arr[])
{
    printf("Elements of array are : ");
    for(int i=0;i<5;i++)
    {
        printf("%d ", arr[i]);
    }
}

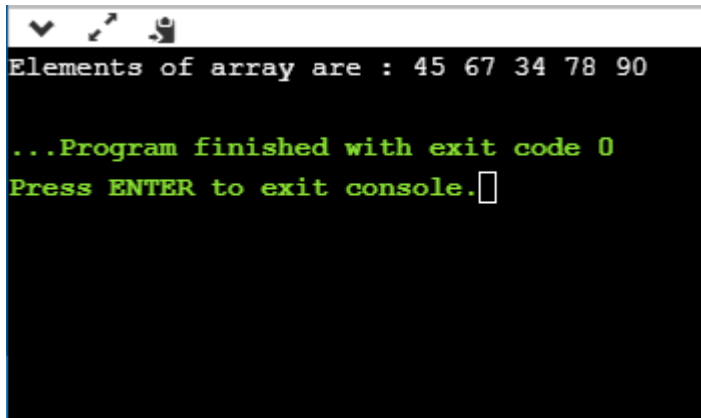
int main()
{
    int arr[5]={45,67,34,78,90};

    getarray(arr);

    return 0;
}
```

In the above program, we have first created the array **arr[]** and then we pass this array to the function **getarray()**. The **getarray()** function prints all the elements of the array **arr[]**.

## Output



```
Elements of array are : 45 67 34 78 90

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

## Passing array to a function as a pointer

Now, we will see how to pass an array to a function as a pointer.

```
#include <stdio.h>

void printarray(char *arr)
{
    printf("Elements of array are : ");
    for(int i=0;i<5;i++)
    {
        printf("%c ", arr[i]);
    }
}

int main()
{
    char arr[5]={'A','B','C','D','E'};
    printarray(arr);
    return 0;
}
```

In the above code, we have passed the array to the function as a pointer. The function **printarray()** prints the elements of an array.

## Output

```
✓ ↗ 📄  
Elements of array are : A B C D E  
  
...Program finished with exit code 0  
Press ENTER to exit console.□
```

Note: From the above examples, we observe that array is passed to a function as a reference which means that array also persist outside the function.

## How to return an array from a function

### Returning pointer pointing to the array

```
#include <stdio.h>  
  
int *getarray()  
{  
    int arr[5];  
    printf("Enter the elements in an array : ");  
    for(int i=0;i<5;i++)  
    {  
        scanf("%d", &arr[i]);  
    }  
    return arr;  
}  
  
int main()  
{  
    int *n;  
    n=getarray();  
    printf("\nElements of array :");  
    for(int i=0;i<5;i++)  
    {
```

```

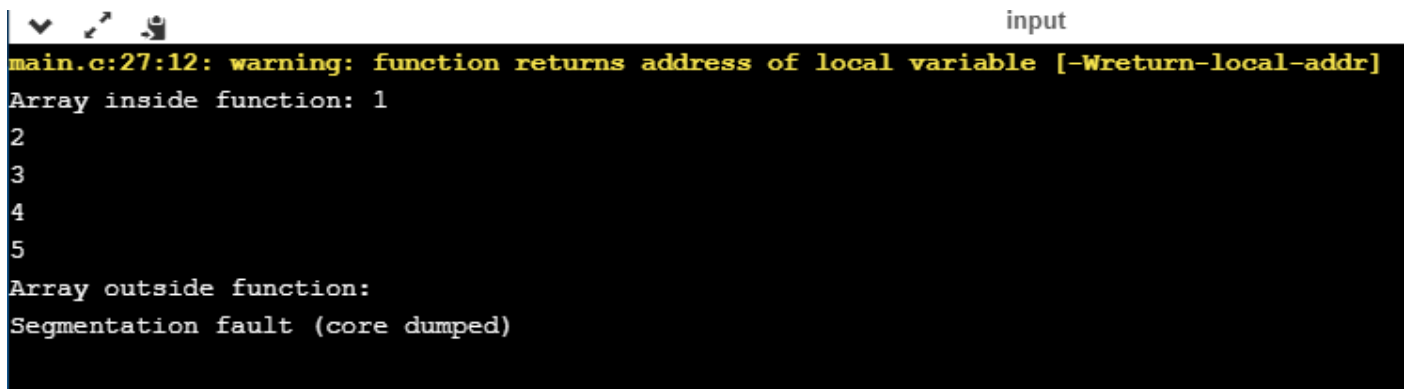
        printf("%d", n[i]);
    }

    return 0;
}

```

In the above program, **getarray()** function returns a variable 'arr'. It returns a local variable, but it is an illegal memory location to be returned, which is allocated within a function in the stack. Since the program control comes back to the **main()** function, and all the variables in a stack are freed. Therefore, we can say that this program is returning memory location, which is already de-allocated, so the output of the program is a **segmentation fault**.

## Output



```

input
main.c:27:12: warning: function returns address of local variable [-Wreturn-local-addr]
Array inside function: 1
2
3
4
5
Array outside function:
Segmentation fault (core dumped)

```

# 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.

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

1. 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:

1. return\_type function(type arrayname[])

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

### Second way:

1. return\_type function(type arrayname[SIZE])

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

### Third way:

1. 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 Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. `int n = 10;`
2. `int* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.`

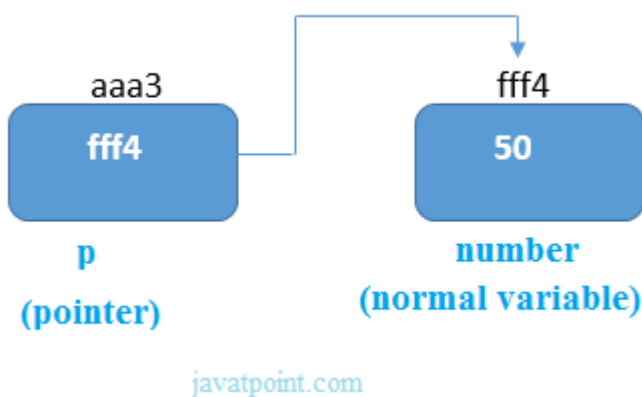
## Declaring a pointer

The pointer in c language can be declared using \* (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. `int *a; // pointer to int`
2. `char *c; // pointer to char`

## Pointer Example

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of \* (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>

int main(){

    int number=50;

    int *p;

    p=&number;//stores the address of number variable

    printf("Address of p variable is %x \n",p); // p contains the address of the number
    therefore printing p gives the address of number.

    printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a
    pointer therefore if we print *p, we will get the value stored at the address contained
    by p.

    return 0;

}
```

## Output

Address of number variable is fff4

Address of p variable is fff4

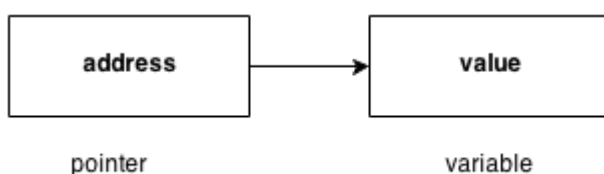
Value of p variable is 50

## Pointer to array

1. `int arr[10];`
2. `int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.`

## Pointer to a function

1. `void show (int);`
2. `void(*p)(int) = &display; // Pointer p is pointing to the address of a function`



## Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.



- 2) We can **return multiple values from a function** using the pointer.
- 3) It makes you able to **access any memory location** in the computer's memory.

## Usage of pointer

There are many applications of pointers in c language.

### 1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

### 2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

## Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>

int main(){
    int number=50;
    printf("value of number is %d, address of number is %u",number,&number);
    return 0;
}
```

### Output

value of number is 50, address of number is fff4

## NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

## Pointer Program to swap two numbers without using the 3rd variable.

```
#include<stdio.h>

int main(){

    int a=10,b=20,*p1=&a,*p2=&b;


    printf("Before swap: *p1=%d *p2=%d",*p1,*p2);

    *p1=*p1+*p2;

    *p2=*p1-*p2;

    *p1=*p1-*p2;

    printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);


    return 0;

}
```

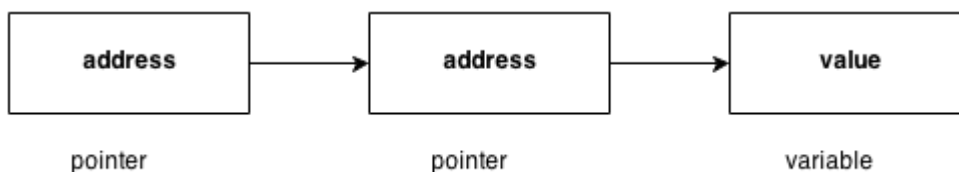
### Output

Before swap: \*p1=10 \*p2=20

After swap: \*p1=20 \*p2=10

## C Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



The syntax of declaring a double pointer is given below.

1. `int **p; // pointer to a pointer which is pointing to an integer.`

Consider the following example.

```
#include<stdio.h>

void main ()
{
    int a = 10;

    int *p;

    int **pp;

    p = &a; // pointer p is pointing to the address of a

    pp = &p; // pointer pp is a double pointer pointing to the address of pointer p

    printf("address of a: %x\n",p); // Address of a will be printed

    printf("address of p: %x\n",pp); // Address of p will be printed

    printf("value stored at p: %d\n",*p); // value stored at the address contained by p i.e. 10
    will be printed

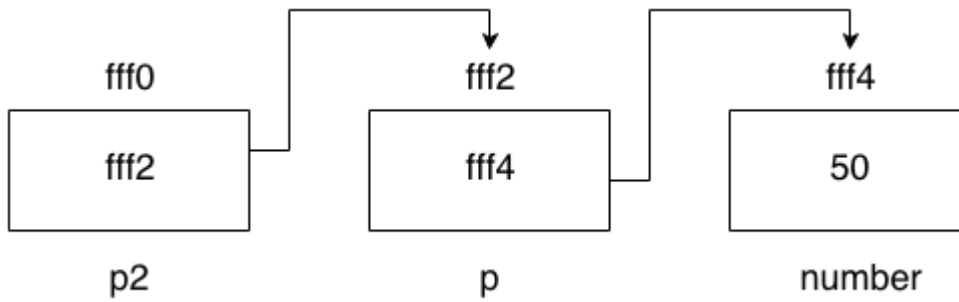
    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the
    pointer stored at pp
}
```

## Output

```
address of a: d26a8734
address of p: d26a8738
value stored at p: 10
value stored at pp: 10
```

## C double pointer example

Let's see an example where one pointer points to the address of another pointer.



As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

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

```
int main(){
```

```
    int number=50;
```

```
    int *p;//pointer to int
```

```
    int **p2;//pointer to pointer
```

```
    p=&number;//stores the address of number variable
```

```
    p2=&p;
```

```
    printf("Address of number variable is %x \n",&number);
```

```
    printf("Address of p variable is %x \n",p);
```

```
    printf("Value of *p variable is %d \n",*p);
```

```
    printf("Address of p2 variable is %x \n",p2);
```

```
    printf("Value of **p2 variable is %d \n",*p);
```

```
    return 0;
```

```
}
```

## Output

Address of number variable is fff4

Address of p variable is fff4

Value of \*p variable is 50

Address of p2 variable is fff2

Value of \*\*p variable is 50

# Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc.

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Let's see the example of incrementing pointer variable on 64-bit architecture.

```
#include<stdio.h>

int main(){
    int number=50;

    int *p;//pointer to int

    p=&number;//stores the address of number variable

    printf("Address of p variable is %u \n",p);

    p=p+1;

    printf("After increment: Address of p variable is %u \n",p); // in our case, p will get
    incremented by 4 bytes.

    return 0;
}
```

## Output

Address of p variable is 3214864300

After increment: Address of p variable is 3214864304

## Traversing an array by using pointer

```
#include<stdio.h>

void main ()
{
    int arr[5] = {1, 2, 3, 4, 5};
```

```

int *p = arr;

int i;

printf("printing array elements...\n");

for(i = 0; i < 5; i++)
{
    printf("%d ",*(p+i));
}
}

```

### Output

```

printing array elements...
1 2 3 4 5

```

## Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

1.  $\text{new\_address} = \text{current\_address} - i * \text{size\_of}(\text{data type})$

### 32-bit

For 32-bit int variable, it will be decremented by 4 bytes.

### 64-bit

For 64-bit int variable, it will be decremented by 8 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```

#include <stdio.h>

void main(){

    int number=50;

    int *p;//pointer to int

    p=&number;//stores the address of number variable

    printf("Address of p variable is %u \n",p);

    p=p-1;
}

```

```
printf("After decrement: Address of p variable is %u \n",p); // P will now point to the  
immediate previous location.
```

```
}
```

## Output

Address of p variable is 3214864300

After decrement: Address of p variable is 3214864296

## C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

1.  $\text{new\_address} = \text{current\_address} + (\text{number} * \text{size\_of}(\text{data type}))$

### 32-bit

For 32-bit int variable, it will add  $2 * \text{number}$ .

### 64-bit

For 64-bit int variable, it will add  $4 * \text{number}$ .

Let's see the example of adding value to pointer variable on 64-bit architecture.

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

```
int main(){
```

```
    int number=50;
```

```
    int *p;//pointer to int
```

```
    p=&number;//stores the address of number variable
```

```
    printf("Address of p variable is %u \n",p);
```

```
    p=p+3; //adding 3 to pointer variable
```

```
    printf("After adding 3: Address of p variable is %u \n",p);
```

```
    return 0;
```

```
}
```

## Output

Address of p variable is 3214864300

After adding 3: Address of p variable is 3214864312

# C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

1.  $\text{new\_address} = \text{current\_address} - (\text{number} * \text{size\_of}(\text{data type}))$

## 32-bit

For 32-bit int variable, it will subtract  $2 * \text{number}$ .

## 64-bit

For 64-bit int variable, it will subtract  $4 * \text{number}$ .

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```
#include<stdio.h>

int main(){
    int number=50;
    int *p;//pointer to int
    p=&number;//stores the address of number variable
    printf("Address of p variable is %u \n",p);
    p=p-3; //subtracting 3 from pointer variable
    printf("After subtracting 3: Address of p variable is %u \n",p);
    return 0;
}
```

## Output

Address of p variable is 3214864300

After subtracting 3: Address of p variable is 3214864288

You can see after subtracting 3 from the pointer variable, it is 12 ( $4*3$ ) less than the previous address value.

If two pointers are of the same type,

1.  $\text{Address2} - \text{Address1} = (\text{Subtraction of two addresses}) / \text{size of data type which pointer points}$



Consider the following example to subtract one pointer from an another.

```
#include<stdio.h>

void main ()
{
    int i = 100;
    int *p = &i;
    int *temp;
    temp = p;
    p = p + 3;
    printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
}
```

### Output

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

## Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.

memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

<b>malloc()</b>	allocates single block of requested memory.
<b>calloc()</b>	allocates multiple block of requested memory.
<b>realloc()</b>	reallocates the memory occupied by malloc() or calloc() functions.
<b>free()</b>	frees the dynamically allocated memory.

## malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

1. `ptr=(cast-type*)malloc(byte-size)`

Let's see the example of malloc() function.

```
#include<stdio.h>

#include<stdlib.h>

int main(){

    int n,i,*ptr,sum=0;

    printf("Enter number of elements: ");

    scanf("%d",&n);

    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc

    if(ptr==NULL)
```

```

{
    printf("Sorry! unable to allocate memory");
    exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}

```

### Output

```

Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30

```

## calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

1. ptr=(cast-type\*)calloc(number, byte-size)

Let's see the example of calloc() function.

```

#include<stdio.h>

#include<stdlib.h>

int main(){
    int n,i,*ptr,sum=0;

    printf("Enter number of elements: ");

    scanf("%d",&n);

    ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");

        exit(0);
    }

    printf("Enter elements of array: ");

    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);

        sum+=*(ptr+i);
    }

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

    free(ptr);

    return 0;
}

```

## Output

```

Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30

```

## realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, new-size)

## free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. free(ptr)

## C Strings

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences.

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

1. By char array
2. By string literal

## String Example in C

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

1. `#include<stdio.h>`
2. `#include <string.h>`
3. `int main(){`
4. `char ch[11]={ 's', 'm', 'a', 'r', 't', 'p', 'o', 'i', 'n', 't', '\0'};`
5. `char ch2[11]="smartpoint";`
- 6.
7. `printf("Char Array Value is: %s\n", ch);`

```
8.  printf("String Literal Value is: %s\n", ch2);
9.  return 0;
10.}
```

## Output

Char Array Value is: smartpoint

String Literal Value is: smartpoint

## Traversing String

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

Hence, there are two ways to traverse a string.

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

Let's discuss each one of them.

### Using the length of string

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

```
1. #include<stdio.h>
2. void main ()
3. {
4.     char s[11] = "smartpoint";
5.     int i = 0;
6.     int count = 0;
7.     while(i<11)
8.     {
9.         if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
10.        {
11.            count ++;
12.        }
```

```
13.     i++;
14. }
15. printf("The number of vowels %d",count);
16.}
```

## Output

The number of vowels 4

## Using the null character

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

```
1. #include<stdio.h>
2. void main ()
3. {
4.     char s[11] = "smartpoint";
5.     int i = 0;
6.     int count = 0;
7.     while(s[i] != NULL)
8.     {
9.         if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
10.        {
11.            count ++;
12.        }
13.        i++;
14.    }
15.    printf("The number of vowels %d",count);
16.}
```

## Output

The number of vowels 4

# Accepting string as the input

Till now, we have used `scanf` to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

```
#include<stdio.h>

void main ()
{
    char s[20];

    printf("Enter the string?");

    scanf("%s",s);

    printf("You entered %s",s);
}
```

## Output

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

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

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `char s[20];`
5. `printf("Enter the string?");`
6. `scanf("%[^\n]s",s);`
7. `printf("You entered %s",s);`
8. `}`

## Output

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



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

## Some important points

However, there are the following points which must be noticed while entering the strings by using scanf.

- The compiler doesn't perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data.
- Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

## Pointers with strings

We have used pointers with the array, functions, and primitive data types so far. However, pointers can be used to point to the strings. There are various advantages of using pointers to point strings. Let us consider the following example to access the string via the pointer.

```
#include<stdio.h>

void main ()
{
    char s[11] = "smartpoint";

    char *p = s; // pointer p is pointing to string s.

    printf("%s",p); // the string smartpoint is printed if we print p.
}
```

### Output

smartpoint

For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.

```
#include<stdio.h>

void main ()
{
    char *p = "hello smartpoint";
    printf("String p: %s\n",p);

    char *q;
    printf("copying the content of p into q...\n");

    q = p;
    printf("String q: %s\n",q);
}
```

### Output

String p: hello smartpoint

copying the content of p into q...

String q: hello smartpoint

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

```
#include<stdio.h>

void main ()
{
    char *p = "hello smartpoint";
    printf("Before assigning: %s\n",p);

    p = "hello";
    printf("After assigning: %s\n",p);
}
```

### Output

Before assigning: hello smartpoint

After assigning: hello

# C gets() and puts() functions

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

## C gets() function

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

### Declaration

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

### Reading string using gets()

```
#include<stdio.h>

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

### Output

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

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

```
#include<stdio.h>

void main()
{
    char str[20];

    printf("Enter the string? ");

    fgets(str, 20, stdin);

    printf("%s", str);
}
```

## Output

```
Enter the string? smartpointis the best website
smartpoint is the b
```

## C puts() function

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console.

### Declaration

1. **int** puts(**char**[])

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

```
#include<stdio.h>

#include <string.h>

int main(){
    char name[50];

    printf("Enter your name: ");

    gets(name); //reads string from user

    printf("Your name is: ");

    puts(name); //displays string

    return 0;
}
```

Output:

Enter your name: Sonoo Jaiswal

Your name is: Sonoo Jaiswal

## C String Length: strlen() function

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

```
#include<stdio.h>

#include <string.h>

int main(){
    char ch[20]={ 's', 'm', 'a', 'r', 't', 'p', 'o', 'i', 'n', 't', '\0' };
    printf("Length of string is: %d",strlen(ch));
    return 0;
}
```

Output:

Length of string is: 10

## C Copy String: strcpy()

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

```
#include<stdio.h>

#include <string.h>

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

Output:

Value of second string is: smartpoint

# Strcat() function in C

Programmers may easily **concatenate** two strings using the **strcat() function** in C, which is a potent tool for manipulating strings. This function accepts **two input strings** and appends the **second string's** content to the end of the **first string**.

```
#include<stdio.h>

#include <string.h>

int main(){

    char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};

    char ch2[10]={'c', '\0'};

    strcat(ch,ch2);

    printf("Value of first string is: %s",ch);

    return 0;

}
```

## Output

```
Value of first string is: helloc
```

# Strcmp() function in C

The **C standard library** includes the **strcmp() function** for **string comparison**. In C programming, it is frequently used to **compare** two strings and is a component of the **<string.h> header**. A connection between the two strings is shown by the function's **return value**, which is an **integer value**.

```
#include<stdio.h>

#include <string.h>

int main(){

    char str1[20],str2[20];

    printf("Enter 1st string: ");

    gets(str1);//reads string from console

    printf("Enter 2nd string: ");

    gets(str2);
```

```
if(strcmp(str1,str2)==0)

    printf("Strings are equal");

else

    printf("Strings are not equal");

return 0;

}
```

### Output

Enter 1st string: hello

Enter 2nd string: hello

Strings are equal

## C Reverse String: strrev()

The **C standard library** does not contain the **strrev() function**, which is not a **standard library function**. The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```
#include<stdio.h>

#include <string.h>

int main(){

    char str[20];

    printf("Enter string: ");

    gets(str);//reads string from console

    printf("String is: %s",str);

    printf("\nReverse String is: %s",strrev(str));

    return 0;

}
```

### Output

Enter string: smartpoint

String is: smartpoint

Reverse String is: tnioptrams

# Strlwr() function in C

Working with strings is a regular activity while programming in the **C programming language**. It is frequently necessary to transform strings to **lowercase** or **uppercase** to ensure **uniform processing** and **comparisons**.

## Syntax:

It has the following syntax:

1. **char\*** strlwr(**char\*** str);

**str**: It is a reference to the string that must be lowercase and has a null ending.

## Value Returned:

The same pointer **str** is returned by the **strlwr() method**, which now points to the **altered lowercase text**.

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

```
#include<stdio.h>

#include <string.h>

int main(){

    char str[20];

    printf("Enter string: ");

    gets(str);//reads string from console

    printf("String is: %s",str);

    printf("\nLower String is: %s",strlwr(str));

    return 0;

}
```

## Output

Enter string: SMARTpoint

String is: SMARTpoint

Lower String is: smartpoint



```
#include <stdio.h>

#include <string.h>

int main() {

    char str[20];

    printf("Enter string: ");

    fgets(str, sizeof(str), stdin);

}
```

### Output

Enter string: HELLOworld

String is: HELLOworld

Lower String is: helloworld

## C String Uppercase:strupr()

In C programming, the `strupr()` method is used to make all the characters in a string uppercase. The **<string.h> header file** contains the ***strupr()* function**, and it is a component of the **C Standard Library**.

```
#include<stdio.h>

#include <string.h>

int main(){

    char str[20];

    printf("Enter string: ");

    gets(str);//reads string from console

    printf("String is: %s",str);

    printf("\nUpper String is: %s",strupr(str));

    return 0;

}
```

### Output

Enter string: smartpoint

String is: smartpoint

Upper String is: SMARTPOINT

# C Structure

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

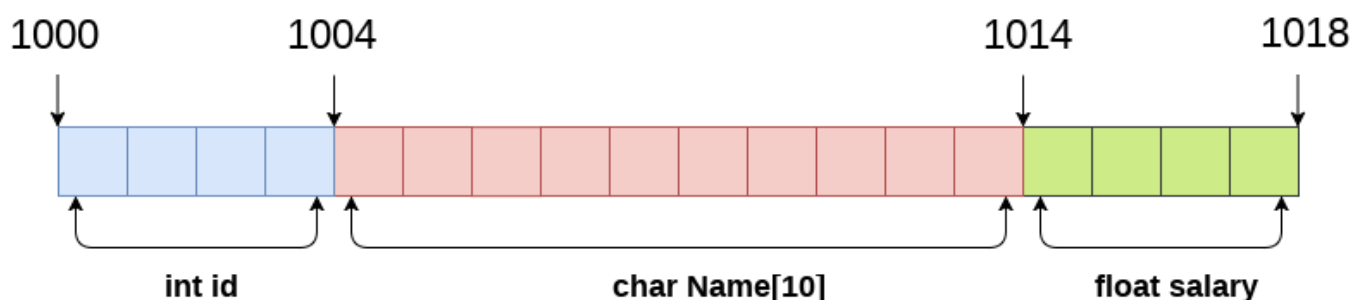
The **,struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

1. **struct** structure\_name
2. {
3.   data\_type member1;
4.   data\_type member2;
5.   .
6.   .
7.   data\_type memeberN;
8. };

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

1. **struct** employee
2. { **int** id;
3.   **char** name[20];
4.   **float** salary;
5. };


The following image shows the memory allocation of the structure employee that is defined in the above example.



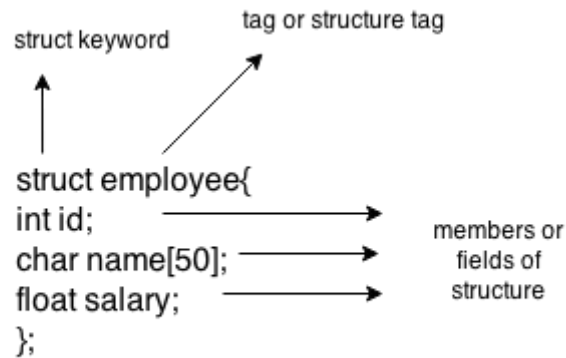
```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

**sizeof (emp) = 4 + 10 + 4 = 18 bytes**

**where;**  
**sizeof (int) = 4 byte**  
**sizeof (char) = 1 byte**  
**sizeof (float) = 4 byte**

 → 1 byte

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:



## Declaring structure variable

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

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

### 1st way:

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

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

Now write given code inside the main() function.

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

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

## 2nd way:

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

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

## Which approach is good

If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

## Accessing members of the structure

There are two ways to access structure members:

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

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

1. p1.id

## C Structure example

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

```
#include<stdio.h>

#include <string.h>

struct employee
{ int id;
  char name[50];
}e1; //declaring e1 variable for structure

int main()
```

```

{
    //store first employee information
    e1.id=101;
    strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
    //printing first employee information
    printf( "employee 1 id : %d\n", e1.id);
    printf( "employee 1 name : %s\n", e1.name);
    return 0;
}

```

### Output:

```

employee 1 id : 101
employee 1 name : Sonoo Jaiswal

```

Let's see another example of the structure in C language to store many employees information.

```

#include<stdio.h>
#include <string.h>
struct employee
{
    int id;
    char name[50];
    float salary;
}e1,e2; //declaring e1 and e2 variables for structure
int main( )
{
    //store first employee information
    e1.id=101;
    strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
    e1.salary=56000;

    //store second employee information

```

```
e2.id=102;

strcpy(e2.name, "James Bond");

e2.salary=126000;


//printing first employee information

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


//printing second employee information

printf( "employee 2 id : %d\n", e2.id);
printf( "employee 2 name : %s\n", e2.name);
printf( "employee 2 salary : %f\n", e2.salary);

return 0;

}
```

#### Output:

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

# C Array of Structures

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

```
#include<stdio.h>

struct student
{
    char name[20];
    int id;
    float marks;
};

void main()
{
    struct student s1,s2,s3;
    int dummy;

    printf("Enter the name, id, and marks of student 1 ");
    scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
    scanf("%c",&dummy);

    printf("Enter the name, id, and marks of student 2 ");
    scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
    scanf("%c",&dummy);

    printf("Enter the name, id, and marks of student 3 ");
    scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
    scanf("%c",&dummy);

    printf("Printing the details....\n");
    printf("%s %d %f\n",s1.name,s1.id,s1.marks);
    printf("%s %d %f\n",s2.name,s2.id,s2.marks);
    printf("%s %d %f\n",s3.name,s3.id,s3.marks);
}
```

**Output**

Enter the name, id, and marks of student 1 James 90 90

Enter the name, id, and marks of student 2 Adoms 90 90

Enter the name, id, and marks of student 3 Nick 90 90

Printing the details....

James 90 90.000000

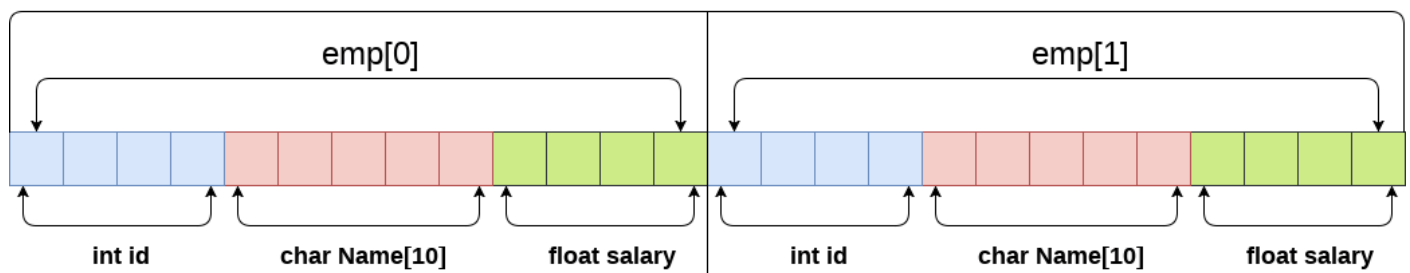
Adoms 90 90.000000

Nick 90 90.000000

## Array of Structures in C

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

### Array of structures



```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

`sizeof (emp) = 4 + 5 + 4 = 13 bytes`

`sizeof (emp[2]) = 26 bytes`

Let's see an example of an array of structures that stores information of 5 students and prints it.

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

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

```
struct student{
```

```
    int rollno;
```

```
    char name[10];
```



```
};  
  
int main(){  
    int i;  
    struct student st[5];  
    printf("Enter Records of 5 students");  
    for(i=0;i<5;i++){  
        printf("\nEnter Rollno:");  
        scanf("%d",&st[i].rollno);  
        printf("\nEnter Name:");  
        scanf("%s",&st[i].name);  
    }  
    printf("\nStudent Information List:");  
    for(i=0;i<5;i++){  
        printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);  
    }  
    return 0;  
}
```

### Output:

```
Enter Records of 5 students  
Enter Rollno:1  
Enter Name:Sonoo  
Enter Rollno:2  
Enter Name:Ratan  
Enter Rollno:3  
Enter Name:Vimal  
Enter Rollno:4  
Enter Name:James  
Enter Rollno:5  
Enter Name:Sarfraz
```

Student Information List:

Rollno:1, Name:Sonoo

Rollno:2, Name:Ratan

Rollno:3, Name:Vimal

Rollno:4, Name:James

Rollno:5, Name:Sarfraz

## Nested Structure in C

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure.

```
#include<stdio.h>

struct address
{
    char city[20];
    int pin;
    char phone[14];
};

struct employee
{
    char name[20];
    struct address add;
};

void main ()
{
    struct employee emp;

    printf("Enter employee information?\n");
    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
    printf("Printing the employee information....\n");
```

```
printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);  
}
```

## Output

Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890

# Union in C

**Union** can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

**Let's understand this through an example.**

1. **struct** abc
2. {
3.   **int** a;
4.   **char** b;
5. }

**Let's have a look at the pictorial representation of the memory allocation.**

The below figure shows the pictorial representation of the structure. The structure has two members; i.e., one is of integer type, and the another one is of character type. Since 1 block is equal to 1 byte; therefore, 'a' variable will be allocated 4 blocks of memory while 'b' variable will be allocated 1 block of memory.

The below figure shows the pictorial representation of union members. Both the variables are sharing the same memory location and having the same initial address.

In union, members will share the memory location. If we try to make changes in any of the member then it will be reflected to the other member as well. Let's understand this concept through an example.

```
union abc
{
    int a;
    char b;
}var;

int main()
{
    var.a = 66;

    printf("\n a = %d", var.a);

    printf("\n b = %d", var.b);
}
```

In the above code, union has two members, i.e., 'a' and 'b'. The 'var' is a variable of union abc type. In the **main()** method, we assign the 66 to 'a' variable, so var.a will print 66 on the screen. Since both 'a' and 'b' share the memory location, **var.b** will print '**B**' (ascii code of 66).

## Deciding the size of the union

The size of the union is based on the size of the largest member of the union.

Let's understand through an example.

```
union abc{

    int a;

    char b;

    float c;

    double d;

};

int main()

{

    printf("Size of union abc is %d", sizeof(union abc));

    return 0;

}
```

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. Since the double variable occupies the largest memory among all the four variables, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

## Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

```
#include <stdio.h>

union abc
{
    int a;
    char b;
};

int main()
{
    union abc *ptr; // pointer variable declaration
    union abc var;
    var.a= 90;
    ptr = &var;
    printf("The value of a is : %d", ptr->a);
    return 0;
}
```

Let's see how can we access the members of the structure.

```
int main()
{
    struct store book;
    book.title = "C programming";
    book.author = "Paulo Cohelo";
    book.number_pages = 190;
    book.price = 205;
    printf("Size is : %ld bytes", sizeof(book));
    return 0;
}
```

We can save lots of space if we use unions.

```
#include <stdio.h>

struct store
{
    double price;
    union
    {
        struct{
            char *title;
            char *author;
            int number_pages;
        } book;

        struct {
            int color;
            int size;
            char *design;
        } shirt;
    }item;
};

int main()
{
    struct store s;
    s.item.book.title = "C programming";
    s.item.book.author = "John";
    s.item.book.number_pages = 189;
    printf("Size is %ld", sizeof(s));
    return 0;
}
```

# C Preprocessor Directives

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

## C #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

1. #include <filename>
2. #include "filename"

The **#include <filename>** tells the compiler to look for the directory where system header files are held. In UNIX, it is \usr\include directory.

The **#include "filename"** tells the compiler to look in the current directory from where program is running.

## #include directive example

Let's see a simple example of #include directive. In this program, we are including stdio.h file because printf() function is defined in this file.

```
#include<stdio.h>

int main(){

    printf("Hello C");

    return 0;

}
```

Output:

Hello C



# C #define

The C programming language's **#define preprocessor directive** provides a strong and flexible tool for declaring **constants** and producing **macros**.

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

## Syntax:

1. #define token value

Let's see an example of #define to define a constant.

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

```
#define PI 3.14
```

```
main() {  
  
    printf("%f",PI);  
  
}
```

## Output:

```
3.140000
```

Let's see an example of #define to create a macro.

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

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

```
void main() {  
  
    printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));  
  
}
```

## Output:

```
Minimum between 10 and 20 is: 10
```

# C #undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax:

1. #undef token

Let's see a simple example to define and undefine a constant.

```
#include <stdio.h>

#define PI 3.14

#undef PI

main() {
    printf("%f",PI);
}
```

Output:

Compile Time Error: 'PI' undeclared

The #undef directive is used to define the preprocessor constant to a limited scope so that you can declare constant again.

Let's see an example where we are defining and undefining number variable. But before being undefined, it was used by square variable.

```
#include <stdio.h>

#define number 15

int square=number*number;

#undef number

main() {
    printf("%d",square);
}
```

Output:

225

# C #ifdef

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

1. `#ifdef MACRO`
2. `//code`
3. `#endif`

Syntax with #else:

1. `#ifdef MACRO`
2. `//successful code`
3. `#else`
4. `//else code`
5. `#endif`

## C #ifdef example

Let's see a simple example to use #ifdef preprocessor directive.

```
#include <stdio.h>

#include <conio.h>

#define NOINPUT

void main() {

    int a=0;

    #ifdef NOINPUT

        a=2;

    #else

        printf("Enter a:");

        scanf("%d", &a);

    #endif

    printf("Value of a: %d\n", a);

    getch();
```

```
}
```

Output:

```
Value of a: 2
```

But, if you don't define NOINPUT, it will ask user to enter a number.

```
#include <stdio.h>

#include <conio.h>

void main() {

    int a=0;

    #ifdef NOINPUT

    a=2;

    #else

    printf("Enter a:");

    scanf("%d", &a);

    #endif


    printf("Value of a: %d\n", a);

    getch();

}
```

Output:

```
Enter a:5
```

```
Value of a: 5
```

# C #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

1. `#ifndef MACRO`
2. `//code`
3. `#endif`

Syntax with #else:

1. `#ifndef MACRO`
2. `//successful code`
3. `#else`
4. `//else code`
5. `#endif`

## C #ifndef example

Let's see a simple example to use #ifndef preprocessor directive.

```
#include <stdio.h>

#include <conio.h>

#define INPUT

void main() {

    int a=0;

    #ifndef INPUT

        a=2;

    #else

        printf("Enter a:");

        scanf("%d", &a);

    #endif

    printf("Value of a: %d\n", a);

    getch();
```

```
}
```

Output:

Enter a:5

Value of a: 5

But, if you don't define INPUT, it will execute the code of #ifndef.

```
#include <stdio.h>

#include <conio.h>

void main() {

    int a=0;

    #ifndef INPUT

    a=2;

    #else

    printf("Enter a:");

    scanf("%d", &a);

    #endif

    printf("Value of a: %d\n", a);

    getch();

}
```

Output:

Value of a: 2

## C #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

1. #if expression
2. //code
3. #endif

Syntax with #else:

1. `#if expression`
2. `//if code`
3. `#else`
4. `//else code`
5. `#endif`

Syntax with #elif and #else:

1. `#if expression`
2. `//if code`
3. `#elif expression`
4. `//elif code`
5. `#else`
6. `//else code`
7. `#endif`

## C #if example

Let's see a simple example to use #if preprocessor directive.

```
#include <stdio.h>

#include <conio.h>

#define NUMBER 0

void main() {

    #if (NUMBER==0)

        printf("Value of Number is: %d",NUMBER);

    #endif

    getch();

}
```

Output:

Value of Number is: 0

Let's see another example to understand the #if directive clearly.

```
#include <stdio.h>

#include <conio.h>

#define NUMBER 1

void main() {

clrscr();

#if (NUMBER==0)

printf("1 Value of Number is: %d",NUMBER);

#endif


#if (NUMBER==1)

printf("2 Value of Number is: %d",NUMBER);

#endif

getch();

}
```

Output:

```
2 Value of Number is: 1
```

## C #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

1. #if expression
2. //if code
3. #else
4. //else code
5. #endif

Syntax with #elif:

1. #if expression
2. //if code



3. `#elif` expression
4. `//elif` code
5. `#else`
6. `//else` code
7. `#endif`

## C #else example

Let's see a simple example to use `#else` preprocessor directive.

```
#include <stdio.h>

#include <conio.h>

#define NUMBER 1

void main() {

    #if NUMBER==0

    printf("Value of Number is: %d",NUMBER);

    #else

    print("Value of Number is non-zero");

    #endif

    getch();

}
```

Output:

Value of Number is non-zero

-----END OF C PROGRAMMING LANGUAGE-----