C Programming Language

It can be defined by the following ways:

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

1) C as a mother language

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

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

2) C as a system programming language

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

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

3) C as a procedural language

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

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

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

4) C as a structured programming language

C is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller structural blocks each of which handles a particular responsibility. These structural blocks are –

- Decision making blocks like if-else-elseif, switch-cases,
- Repetitive blocks like For-loop, While-loop, Do-while loop etc
- subroutines/procedures - functions

The program which solves the entire problem is a collection of such structural blocks. Even a bigger structural block like a function can have smaller inner structural blocks like decisions and loops.

Structure means to break a program into parts or blocks so that it may be easy to understand.Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if else, switch) and repetition (while, dso-while and for), block structures, and subroutines.

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.

Features of C Language

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

1. Simple
2. Mid-level programming language
3. structured programming language
4. Rich Library
5. Memory Management
6. Fast Speed
7. Pointers
8. Recursion
9. 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) 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.

---

3) 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.

---

4) Rich Library

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

---

5) 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.

---

6) Speed

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

---

7) 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.

---

8) 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.

---

9) Extensible

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

Compilation process in c

What is a compilation?

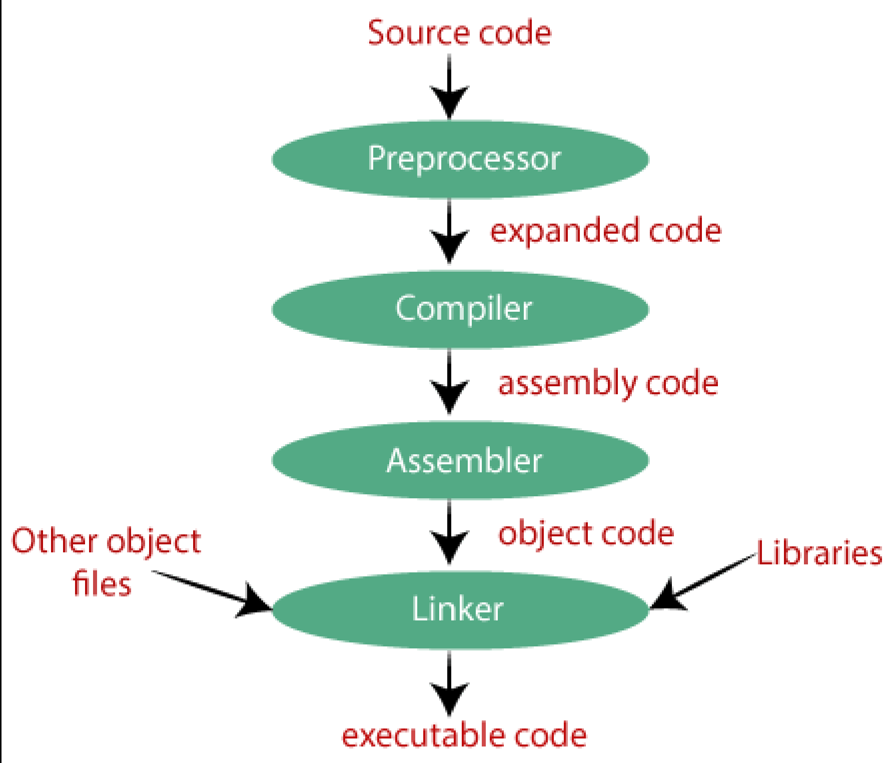
The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the errors, and if the source code is error-free, then it generates the object code.

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 **#include<stdio.h>** 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

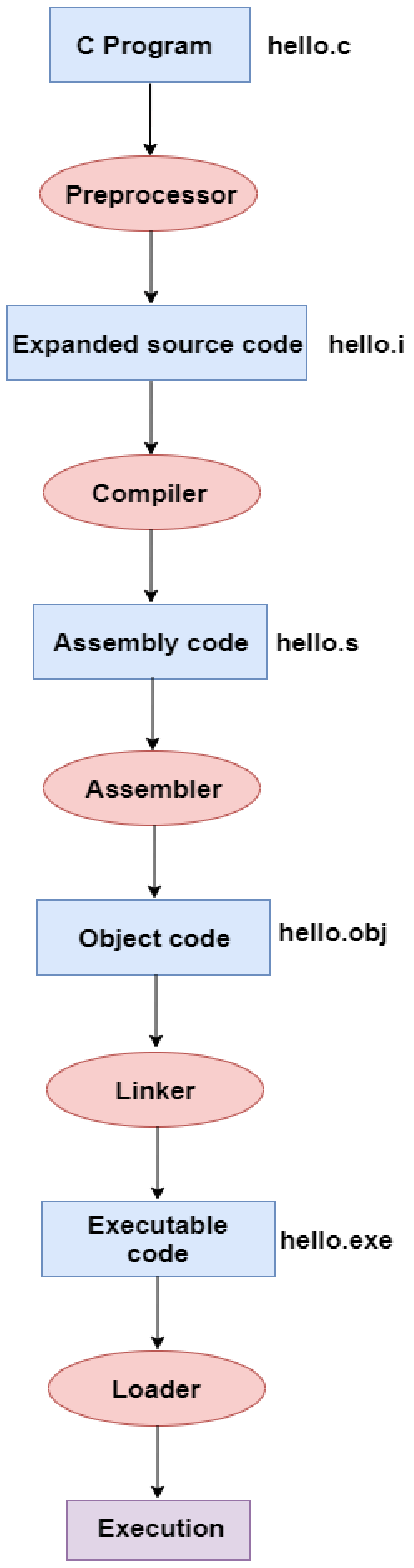
Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. 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', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.

Let's understand through an example.

hello.c

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

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


C Program
hello.c
Preprocessor
Expanded source code
hello.i
Compiler
Assembly code
hello.s
Assembler
Object code
hello.obj
Linker
Executable code
hello.exe
Loader
Execution

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:

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

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

scanf() function

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

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

Program to print cube of given number

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

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int num,cube;
printf("enter a number:");
scanf("%d",&num);
cube=num*num*num
printf("cube of number is:%d ",cube);
getch();
}
```

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.

Syntax to declare a variable:

```
datatype variable_list;
```

The example of declaring the variable is given below:

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

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

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

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

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace and special character is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.
- It can be of any length. However, you may run into problems in some compilers if the variable name is longer than 31 characters.
- Variable names are case sensitive.

Note: You should always try to give meaningful names to variables. For example: **firstName** is a better variable name than **fn**.

C is a strongly typed language. This means that the variable type cannot be changed once it is declared. For example:

```
int number = 5;      // integer variable
number = 5.5;        // error
double number;       // error
```

Here, the type of `number` variable is `int`. You cannot assign a floating-point (decimal) value `5.5` to this variable. Also, you cannot redefine the data type of the variable to `double.`

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 is accessible only in the block in which it has been declared.

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", x);
}
```

Constants

As the name suggests the name constants is given to such variables or values in C programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any types of constants like integer, float, octal, hexadecimal, character constants etc.

If you want to define a variable whose value cannot be changed, you can use the `const` keyword. This will create a constant. For example,

```
const double PI = 3.14;
```

Notice, we have added keyword `const`.
Here, `PI` is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;
PI = 2.9; //Error
```

You can also define a constant using the `#define` preprocessor directive.

```
#include <stdio.h>

// Constants
#define val 10
#define floatVal 4.5
#define charVal 'G'

int main()
{
    printf("Integer Constant: %d\n", val);
    printf("Floating point Constant: %f\n", floatVal);
    printf("Character Constant: %c\n", charVal);

    return 0;
}
```

Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: `1`, `2.5`, `'c'` etc.
Here, `1`, `2.5` and `'c'` are literals. Why? You cannot assign different values to these terms.

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)

For example:

```
Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc
```

In C programming, octal starts with a `0`, and hexadecimal starts with a `0x`.

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

```
-2.0

0.0000234

-0.22E-5
```

Note: `E-5 = 10`$^{-5}$

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: `'a'`, `'m'`, `'F'`, `'2'`, `'}'` etc.

4. Escape Sequences(Non-graphic characters)

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C programming.

Non-graphic characters are those characters which cannot be typed directly from keyboard. They can be stored by using escape sequence.

Escape sequence is back slash(\) followed by one or more character.

For example: newline(enter), tab etc.

In order to use these characters, escape sequences are used.

Escape Sequences

Escape Sequences	**Character**
`\b`	Backspace
`\n`	Newline
`\t`	Horizontal tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\0`	Null character

For example: `\n` is used for a newline. The backslash `\` causes escape from the normal way the characters are handled by the compiler.

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

```
"good"                  //string constant

""                     //null string constant

"      "               //string constant of six white space

"x"                    //string constant having a single character.

"Earth is round\n"         //prints string with a newline
```

Character set

Set of valid characters which a language can recognize is known as character set. C character set includes alphabets, digits and some special characters.

Alphabets

```
Uppercase: A B C ................................... X Y Z

Lowercase: a b c ...................................... x y z
```

C accepts both lowercase and uppercase alphabets.

Digits

```
0 1 2 3 4 5 6 7 8 9
```

Special Characters

*,&, #,!,…. etc

White space Characters

Blank space, newline, horizontal tab etc.

C Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example,

```
int money;
```

Here, `int` is a keyword that indicates `money` is a variable of type `int` (integer).

As C is a case sensitive language, all keywords must be written in lowercase.

There are only 32 reserved words (keywords) in the C language.

c keywords

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

C Identifiers

C identifiers represent the names given to different components in the C program, for example, variables, functions, arrays, structures, unions, labels, etc.

Rules for naming identifiers:

- An identifier can have alphabets, digits, and underscore.
- An identifier can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace and special character is allowed within the identifier.
- An identifier must not be any reserved word or keyword, e.g. int, float, etc.
- It can be of any length. However, you may run into problems in some compilers if the variable name is longer than 31 characters.
- Identifiers are case sensitive.

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.

C Programming Operators:

An operator is a symbol that is used to perform some operations on a value or a variable. For example: `+` is an operator to perform addition. C has a wide range of operators to perform various operations.

Types of operators:

1. Unary operator: An operator that works on single operand is called as unary operator. For eg, ++,--

2. Binary operator: An operator that works on two operands is called as binary operator. For eg, +,-,*,/ etc

3. Ternary operator: An operator that works on three operands is called as ternary operator. For eg, ?:(Ternary operator)

C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc/Other Operators

C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division

Operator	Meaning of Operator
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

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

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

    return 0;
}
```

Output

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

The operators `+`, `-` and `*` computes addition, subtraction, and multiplication respectively as you might have expected.
In normal calculation, `9/4 = 2.25`. However, the output is `2` in the program.
It is because both the variables `a` and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer `2` instead of `2.25`.
The modulo operator `%` computes the remainder. When `a=9` is divided by `b=4`, the remainder is `1`. The `%` operator can only be used with integers.
Suppose `a = 5.0`, `b = 2.0`, `c = 5` and `d = 2`. Then in C programming,

```
// Either one of the operands is a floating-point number

a/b = 2.5

a/d = 2.5

c/b = 2.5



// Both operands are integers

c/d = 2
```

C Increment and Decrement Operators

C programming has two operators increment ++ and decrement -- to change the value of an operand by 1.
Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

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

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

    return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000
```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`.

a++	postfix increment	follows USE THAN CHANGE
++a	prefix increment	follows CHANGE THAN USE
a--	postfix decrement	follows USE THAN CHANGE
--a	prefix decrement	follows CHANGE THAN USE

EXAMPLE:

```
#include <stdio.h>
int main()
 {
   int var1 = 5, var2 = 5,c,d;

//var1 is firstly assigned to c and then incremented
   c=var1++;
//var2 is firstly incremented and then assigned to d
   d=++var2;

   printf("%d\n", var1);
   printf("%d\n", var2);
   printf("%d\n", c);
   printf("%d\n", d);


   return 0;
}
```

OUTPUT:

```
6
6
5
6
```

C Assignment Operators

An assignment operator is used for assigning a value to a variable.

The most common assignment operator is =

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0. Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1
<=	Less than or equal to	`5 <= 3` is evaluated to 0

Example: Relational Operators

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

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

    return 0;
}
```

Output

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

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

Example: Logical Operators

```
// Working of logical operators

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

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

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

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

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

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

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

    return 0;
}
```

Output

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

Explanation of logical operator program

- `(a == b) && (c > 5)` evaluates to 1 because both operands `(a == b)` and `(c > b)` is 1 (true).
- `(a == b) && (c < b)` evaluates to 0 because operand `(c < b)` is 0 (false).
- `(a == b) || (c < b)` evaluates to 1 because `(a = b)` is 1 (true).
- `(a != b) || (c < b)` evaluates to 0 because both operand `(a != b)` and `(c < b)` are 0 (false).
- `!(a != b)` evaluates to 1 because operand `(a != b)` is 0 (false). Hence, !(a != b) is 1 (true).
- `!(a == b)` evaluates to 0 because `(a == b)` is 1 (true). Hence, `!(a == b)` is 0 (false).

C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)



Bit Operation of 12 and 25

  00001100
```

```
& 00011001

  ________

  00001000  = 8 (In decimal)
```

Example #1: Bitwise AND

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a&b);
    return 0;
}
```

Output

```
Output = 8
```

Bitwise OR operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)



Bitwise OR Operation of 12 and 25

  00001100

| 00011001

  ________

  00011101  = 29 (In decimal)
```

Example #2: Bitwise OR

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a|b);
    return 0;
}
```

Output

```
Output = 29
```

Bitwise XOR (exclusive OR) operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)



Bitwise XOR Operation of 12 and 25

  00001100

^ 00011001

  ________

  00010101  = 21 (In decimal)
```

Example #3: Bitwise XOR

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a^b);
    return 0;
}
```

Output

```
Output = 21
```

Bitwise complement operator ~

Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. It is denoted by ~.

```
35 = 00100011 (In Binary)



Bitwise complement Operation of 35

~ 00100011

  ________

  11011100  = 220 (In decimal)
```

Twist in bitwise complement operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer `n`, bitwise complement of `n` will be `-(n+1)`. To understand this, you should have the knowledge of 2's complement.

2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

```
 Decimal         Binary           2's complement

   0            00000000           -(11111111+1) = -00000000 = -0(decimal)

   1            00000001           -(11111110+1) = -11111111 = -256(decimal)

   12           00001100           -(11110011+1) = -11110100 = -244(decimal)

   220          11011100           -(00100011+1) = -00100100 = -36(decimal)


Note: Overflow is ignored while computing 2's complement.
```

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

Bitwise complement of any number N is -(N+1). Here's how:

```
bitwise complement of N = ~N (represented in 2's complement form)

2'complement of ~N= -(~(~N)+1) = -(N+1)
```

Example #4: Bitwise complement

```
#include <stdio.h>
int main()
{
    printf("Output = %d\n",~35);
    printf("Output = %d\n",~-12);
    return 0;
}
```

Output

```
Output = -36
Output = 11
```

Shift Operators in C programming

There are two shift operators in C programming:

- Right shift operator
- Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

```
212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)
```

Left Shift Operator

Left shift operator shifts all bits towards left by a certain number of specified bits. The bit positions that have been vacated by the left shift operator are filled with 0. The symbol of the left shift operator is <<.

```
212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 = 11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)
```

Example #5: Shift Operators

```
#include <stdio.h>
int main()
{
    int num=212, i;

        printf("Right shift by 0: %d\n", num>>0);
        printf("Right shift by 1: %d\n", num>>1);
        printf("Right shift by 2: %d\n", num>>2);
        printf("\n");

        num=212;
        printf("Left shift by 0: %d\n", num<<0);
        printf("Left shift by 1: %d\n", num<<1);
        printf("Left shift by 2: %d\n", num<<2);
     return 0;
}
```

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

Other Operators:

Comma Operator

Comma operators are used to link related expressions together. For example:

int a, c = 5, d;

The sizeof operator

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Example : sizeof Operator

```
#include <stdio.h>
void main()
{
   int a;
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
   printf("Size of char=%lu byte\n",sizeof(d));
   }
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y