

## Introduction to C language:

### What is Programming?

Computer programming is a medium for us to communicate with computers just like we use Hindi or English to communicate with each other, programming is a way for us to deliver our instructions to the computer.

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

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

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

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

### Features of C Language

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

1. Simple

2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

#### 1) Simple

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

#### 2) Machine Independent or Portable

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

#### 3) Mid-level programming language

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

#### 4) Structured programming language

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

#### 5) Rich Library

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

#### 6) Memory Management

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

## 7) Speed

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

## 8) Pointer

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

## 9) Recursion

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

## 10) Extensible

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

## First C 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:

```
1. #include <stdio.h>
2. main(){
3. printf("WELCOME TO PDS CLASS");
4. return 0;
5. }
```

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

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

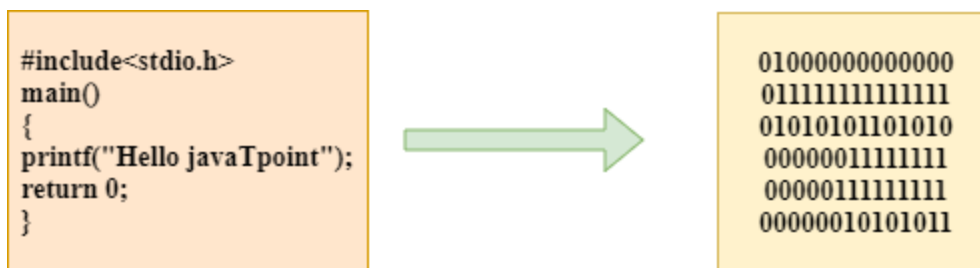
**printf()** The printf() function is **used to print data** on the console.

**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.

## 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.

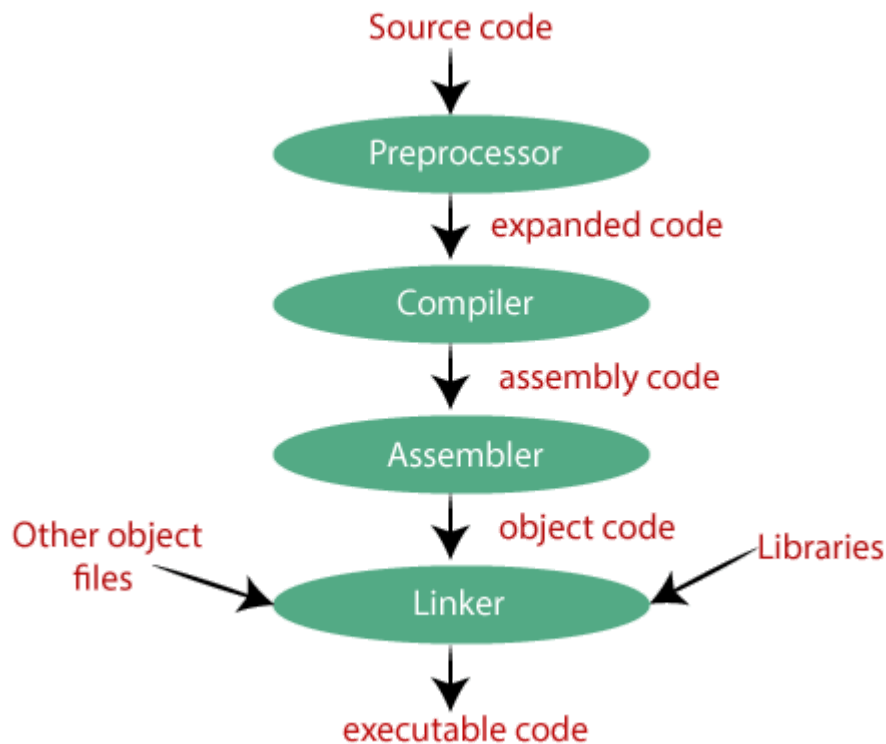


The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

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

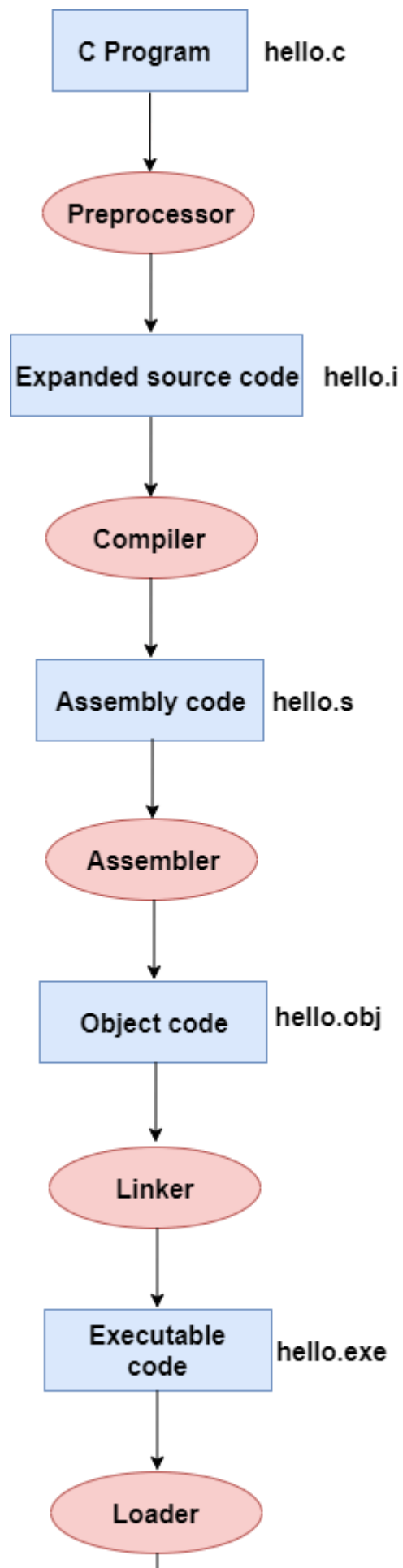
Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. 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**

1. #include <stdio.h>
2. **int** main()
3. {
4.     printf("Hello javaTpoint");
5.     **return** 0;
6. }

**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.

## **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.

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

3. }

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

### Global Variable

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

It must be declared at the start of the block.

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

### Static Variable

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

It retains its value between multiple function calls.

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

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

### 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**.

1. **void** main(){

2. **int** x=10;//local variable (also automatic)
3. auto **int** y=20;//automatic variable
4. }

### 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*

1. **extern int** x=10;//external variable (also global)
- program1.c*

1. #include "myfile.h"
2. #include <stdio.h>
3. **void** printValue(){
4.     printf("Global variable: %d", global\_variable);
5. }

## Data Types in C :

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

Following are the examples of some very common data types used in C:

- **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
- **int:** As the name suggests, an int variable is used to store an integer.
- **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Different data types also have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler. Below is list of ranges along with the memory requirement and format specifiers on 32 bit gcc compiler.

DATA TYPE	MEMORY	RANGE	FORMAT
-----------	--------	-------	--------

	(BYTES)		SPECIFIER
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long	4	0 to 4,294,967,295	%lu

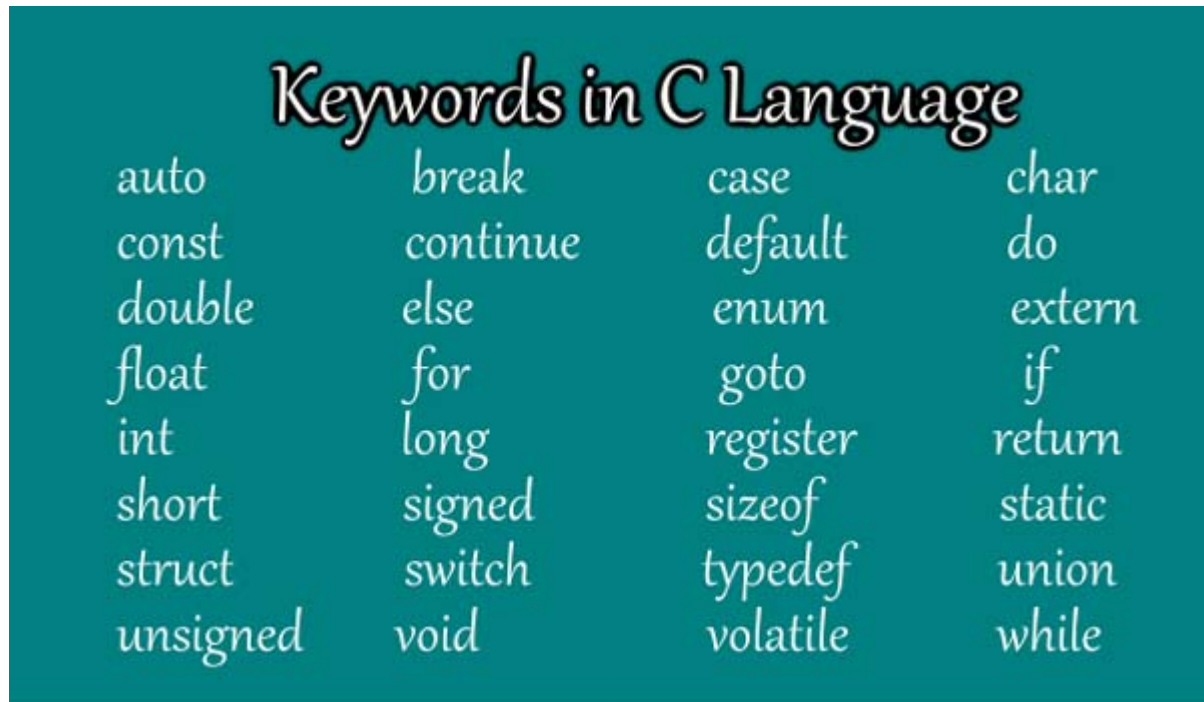
int			
long long int	8	$-(2^{63})$ to $(2^{63})-1$	%lld
unsigned long		0 to	
long int	8	18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned			
char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%Lf

---

## 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

We will learn about all the C language keywords later.

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

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent

various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumerical characters that represent the identifiers.

#### Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

#### Example of valid identifiers

1. total, sum, average, \_m\_, sum\_1, etc.

#### Example of invalid identifiers

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.

```
1. int main()
2. {
3.     int a=10;
4.     int A=20;
5.     printf("Value of a is : %d",a);
6.     printf("\nValue of A is :%d",A);
7.     return 0;
8. }
```

### 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 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
%	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 (constant or variable) 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

```
// 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--. Visit this page to learn more about how increment and decrement operators work when used as postfix.

### **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

### **Example 3: Assignment Operators**

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

```

c = a;    // c is 5
printf("c = %d\n", c);
c += a;   // c is 10
printf("c = %d\n", c);
c -= a;   // c is 5
printf("c = %d\n", c);
c *= a;   // c is 25
printf("c = %d\n", c);
c /= a;   // c is 5
printf("c = %d\n", c);
c %= a;   // c = 0
printf("c = %d\n", c);

return 0;
}

```

## Output

```

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

```

## 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 4: Relational Operators

```
// 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 && (d>5)) equals to 0.
	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equal

## Example 5: 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  $(c > 5)$  is 1 (true).
- $(a == b) \&\& (c < b)$  evaluates to 0 because operand  $(c < b)$  is 0 (false).
- $(a == b) \parallel (c < b)$  evaluates to 1 because  $(a == b)$  is 1 (true).
- $(a != b) \parallel (c < b)$  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
$\wedge$	Bitwise exclusive OR
$\sim$	Bitwise complement
$\ll$	Shift left

>>

Shift right

Visit [bitwise operator in C](#) to learn more.

## 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 6: sizeof Operator

```
#include <stdio.h>
int 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));

    return 0;
```

```
}
```

## Output

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

Other operators such as ternary operator `?:`, reference operator `&`, dereference operator `*` and member selection operator `->` will be discussed in later tutorials.

## Constants in C:

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

There are different types of constants in C programming.

List of Constants in C

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

## 2 ways to define constant in C

There are two ways to define constant in C programming.

1. `const` keyword
2. `#define` preprocessor

### 1) C `const` keyword

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

1. **`const float PI=3.14;`**

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

1. `#include<stdio.h>`
2. **`int main(){`**
3. **`const float PI=3.14;`**
4. `printf("The value of PI is: %f",PI);`
5. **`return 0;`**
6. **`}`**

#### **Output:**

The value of PI is: 3.140000

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

1. `#include<stdio.h>`
2. **`int main(){`**
3. **`const float PI=3.14;`**
4. `PI=4.5;`
5. `printf("The value of PI is: %f",PI);`
6. **`return 0;`**
7. **`}`**

#### **Output:**

Compile Time Error: Cannot modify a const object

## 2) C #define preprocessor

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

## 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



## Syntax error

Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified programming language. These mistakes are generally made by beginners only because they are new to the language. These errors can be easily debugged or corrected.

### For example:

1. If we want to declare the variable of type integer,
2. **int** a; // this is the correct form
3. Int a; // this is an incorrect form.

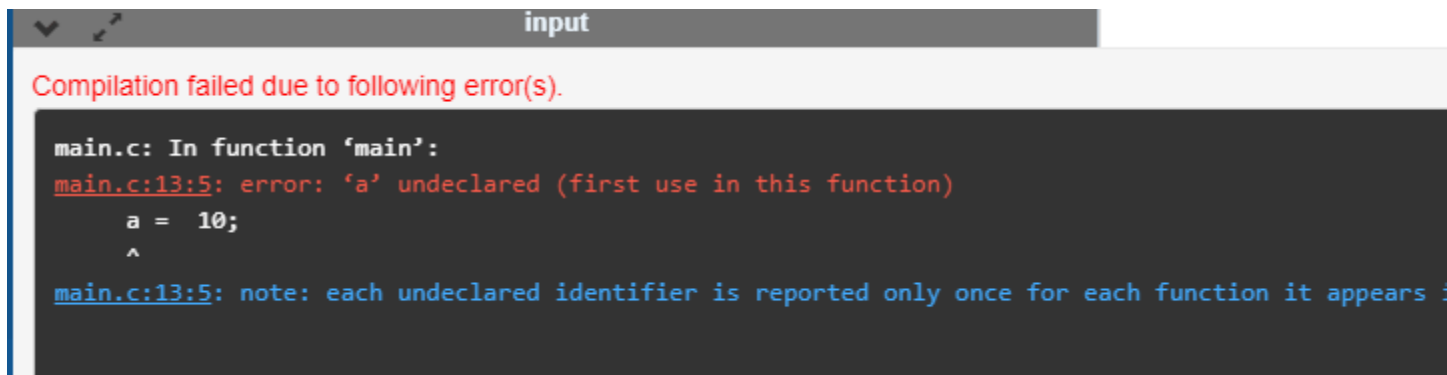
Commonly occurred syntax errors are:

- If we miss the parenthesis ( ) while writing the code.
- Displaying the value of a variable without its declaration.
- If we miss the semicolon ( ; ) at the end of the statement.

### Let's understand through an example.

1. #include <stdio.h>
2. **int** main()
3. {
4.     a = 10;
5.     printf("The value of a is : %d", a);
6.     **return** 0;
7. }

### Output

A screenshot of a code editor window titled 'input'. It shows a compilation error message in red text: 'Compilation failed due to following error(s)'. Below this, the error details are shown in a dark background with light text: 'main.c: In function 'main':', 'main.c:13:5: error: 'a' undeclared (first use in this function)', and a code snippet 'a = 10;' with a caret under the 'a'. A blue note at the bottom states: 'main.c:13:5: note: each undeclared identifier is reported only once for each function it appears in'.

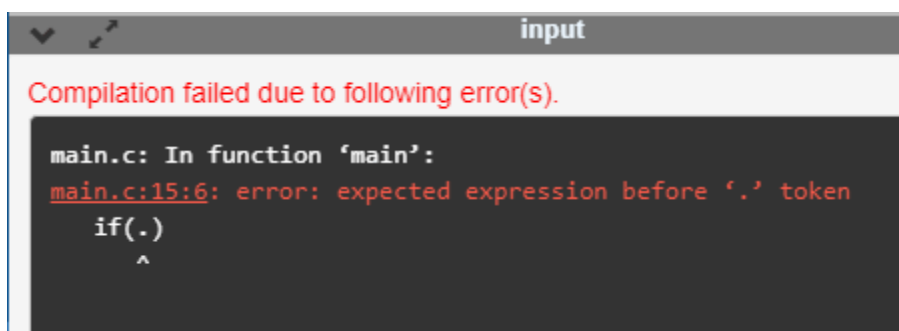
In the above output, we observe that the code throws the error that 'a' is undeclared. This error is nothing but the syntax error only.

There can be another possibility in which the syntax error can exist, i.e., if we make mistakes in the basic construct. Let's understand this scenario through an example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int a=2;`
5. `if(.) // syntax error`
6.
7. `printf("a is greater than 1");`
8. `return 0;`
9. `}`

In the above code, we put the (.) instead of condition in 'if', so this generates the syntax error as shown in the below screenshot.

## Output

A screenshot of a code editor window titled 'input'. It shows a compilation error message in red text: 'Compilation failed due to following error(s)'. Below this, the error details are shown in a dark background with light text: 'main.c: In function 'main':', 'main.c:15:6: error: expected expression before '.' token', and a code snippet 'if(.)' with a caret under the period.

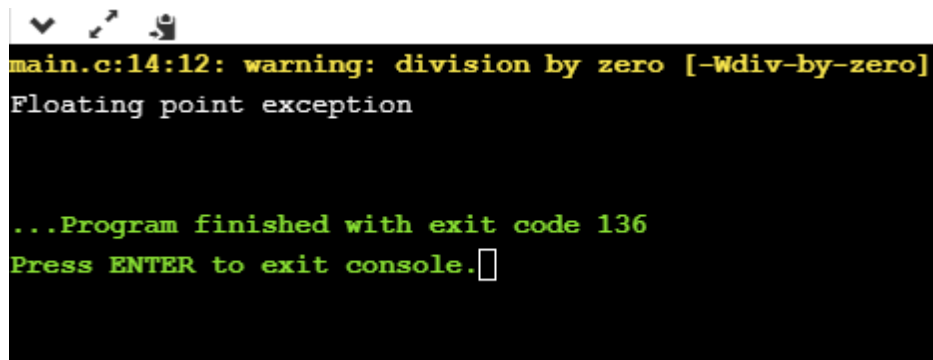
## Run-time error

Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

**Let's understand through an example.**

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=2;
5.     int b=2/0;
6.     printf("The value of b is : %d", b);
7.     return 0;
8. }
```

## Output



```
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
Floating point exception

...Program finished with exit code 136
Press ENTER to exit console.
```

In the above output, we observe that the code shows the run-time error, i.e., division by zero.

## Linker error

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. For example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function



in **func.c** file, so it generates two object files, i.e., **main.o** and **func.o**. At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown. The most common linker error that occurs is that we use **Main()** instead of **main()**.

**Let's understand through a simple example.**

```
1. #include <stdio.h>
2. int Main()
3. {
4.     int a=78;
5.     printf("The value of a is : %d", a);
6.     return 0;
7. }
```

**Output**

```
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Logical error

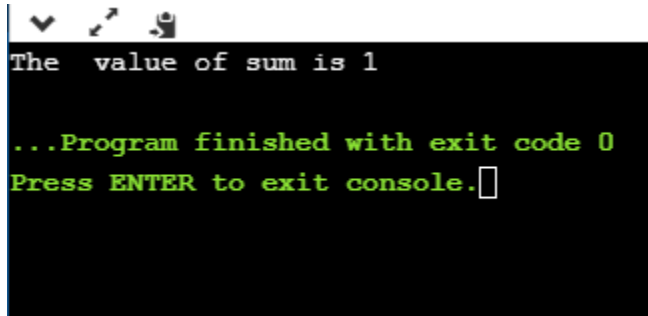
The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors. These types of mistakes are mainly done by beginners. The occurrence of these errors mainly depends upon the logical thinking of the developer. If the programmers sound logically good, then there will be fewer chances of these errors.

**Let's understand through an example.**

```
1. #include <stdio.h>
2. int main()
3. {
4.     int sum=0; // variable initialization
5.     int k=1;
6.     for(int i=1;i<=10;i++); // logical error, as we put the semicolon after loop
7.     {
8.         sum=sum+k;
9.         k++;
10.    }
```

```
11.printf("The value of sum is %d", sum);
12. return 0;
13.}
```

## Output



```
The value of sum is 1

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

In the above code, we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon (;) after the for loop, so the inner statements of the for loop will not execute. This produces the wrong output.

## Semantic error

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

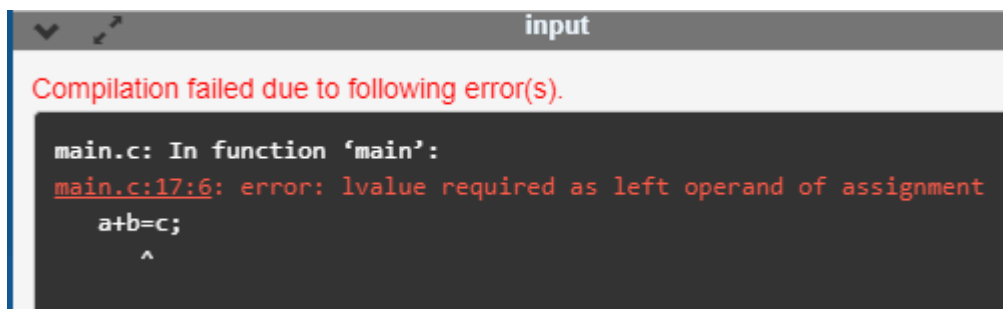
- Use of a un-initialized variable.  
int i;  
i=i+2;
- Type compatibility  
int b = "lakshmibala";
- Errors in expressions  
int a, b, c;  
a+b = c;
- Array index out of bound  
int a[10];  
a[10] = 34;

**Let's understand through an example.**

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a,b,c;
5.     a=2;
6.     b=3;
7.     c=1;
8.     a+b=c; // semantic error
9.     return 0;
10.}
```

In the above code, we use the statement **a+b =c**, which is incorrect as we cannot use the two operands on the left-side.

## Output



```
input
Compilation failed due to following error(s).
main.c: In function 'main':
main.c:17:6: error: lvalue required as left operand of assignment
a+b=c;
    ^
```

## Operator Precedence and Associativity in C :

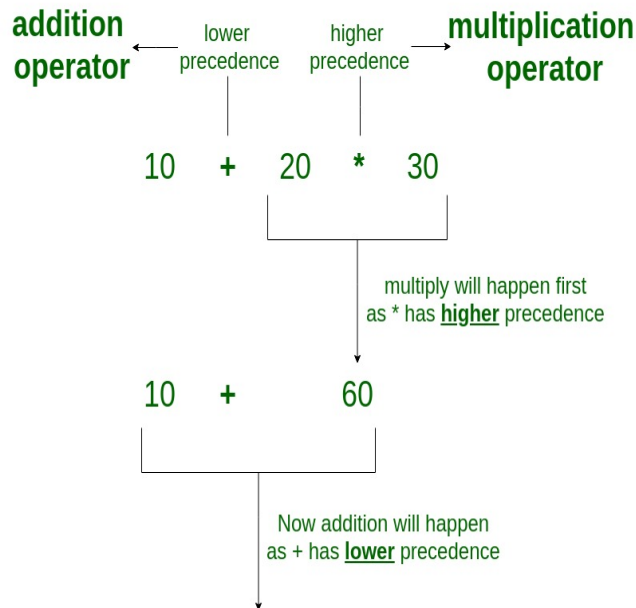
Last Updated: 08-07-2019

**Operator precedence** determines which operator is performed first in an expression with more than one operators with different precedence.

**For example:** Solve

```
10 + 20 * 30
```

# Operator Precedence



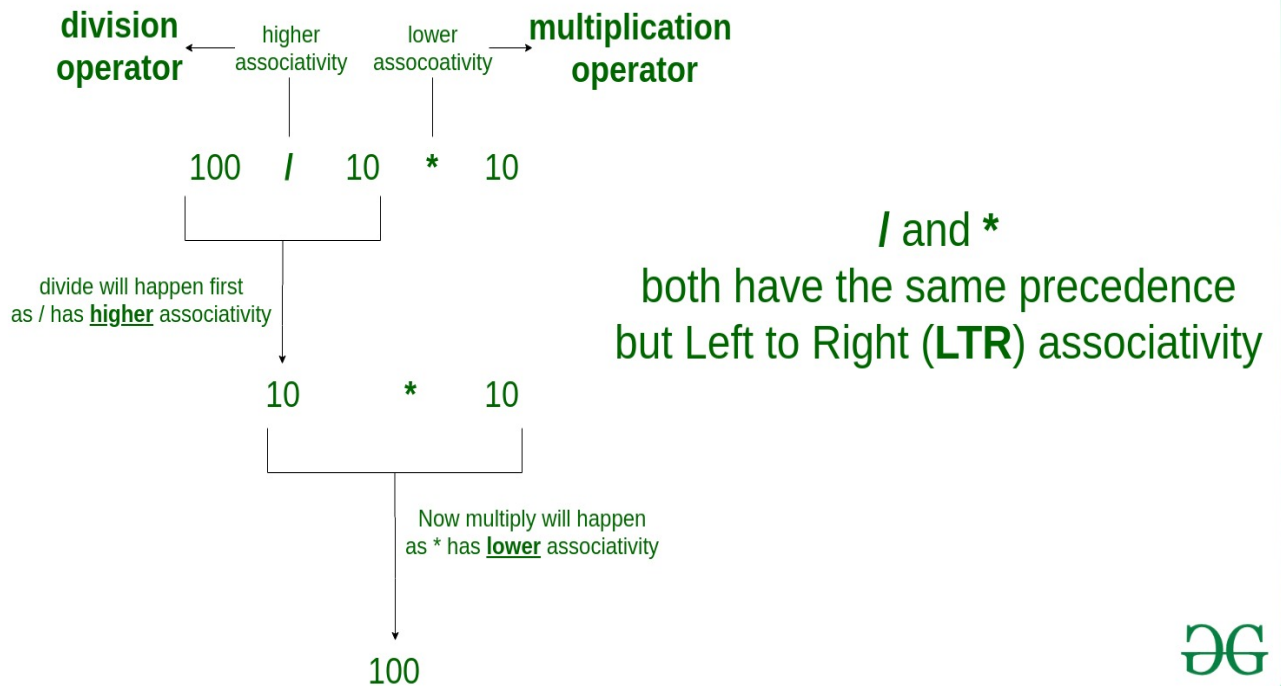
**$10 + 20 * 30$**  is calculated as  **$10 + (20 * 30)$**   
and not as  **$(10 + 20) * 30$**

**Operators Associativity** is used when two operators of same precedence appear in an expression. Associativity can be either **Left to Right** or **Right to Left**.

**For example:** '\*' and '/' have same precedence and their associativity is **Left to Right**, so the expression " $100 / 10 * 10$ " is treated as " $(100 / 10) * 10$ ".

ADVERTISING

# Operator Associativity

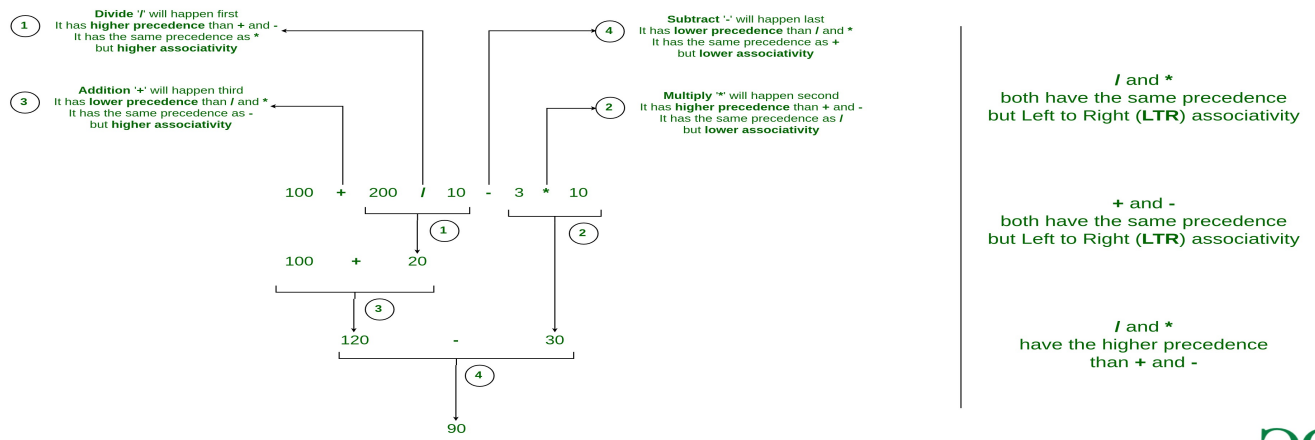


*Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets*

**For example:** Solve

$$100 + 200 / 10 - 3 * 10$$

## Operator Precedence and Associativity



### 1) Associativity is only used when there are two or more operators of same precedence.

The point to note is associativity doesn't define the order in which operands of a single operator are evaluated. For example, consider the following program, associativity of the + operator is left to right, but it doesn't mean f1() is always called before f2(). The output of the following program is in-fact compiler dependent. See [this](#) for details.

// Associativity is not used in the below program.

// Output is compiler dependent.

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

```
int x = 0;
```

```
int f1()
```

```
{
```

```
    x = 5;
```

```
    return x;
```

```

}

int f2()
{
    x = 10;

    return x;
}

int main()
{
    int p = f1() + f2();

    printf("%d ", x);

    return 0;
}

```

## **2) All operators with the same precedence have same associativity**

This is necessary, otherwise, there won't be any way for the compiler to decide evaluation order of expressions which have two operators of same precedence and different associativity. For example + and – have the same associativity.

## **3) Precedence and associativity of postfix ++ and prefix ++ are different**

Precedence of postfix ++ is more than prefix ++, their associativity is also different. Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left. See [this](#) for examples.

## **4) Comma has the least precedence among all operators and should be used carefully**

For example consider the following program, the output is 1.

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

```

int main()
{
    int a;

    a = 1, 2, 3; // Evaluated as (a = 1), 2, 3
}

```

```

printf("%d", a);

return 0;

}

```

### 5) There is no chaining of comparison operators in C

In Python, expression like “c > b > a” is treated as “c > b and b > a”, but this type of chaining doesn’t happen in C. For example consider the following program. The output of following program is “FALSE”.

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

```

int main()
{
    int a = 10, b = 20, c = 30;

    // (c > b > a) is treated as ((c > b) > a), associativity of '>'
    // is left to right. Therefore the value becomes ((30 > 20) > 10)
    // which becomes (1 > 20)
    if (c > b > a)
        printf("TRUE");
    else
        printf("FALSE");

    return 0;
}

```

Please see the following precedence and associativity table for reference.

OPERATOR	DESCRIPTION	ASSOCIATIVITY
----------	-------------	---------------



	Parentheses (function call) (see Note 1)	
( )	Brackets (array subscript)	
[ ]	Member selection via object name	
.	Member selection via pointer	
->	Postfix increment/decrement (see Note 2)	
++ —		left-to-right
	Prefix increment/decrement	
	Unary plus/minus	
	Logical negation/bitwise complement	
++ —		
+ —	Cast (convert value to temporary value of <i>type</i> )	
! ~		
( <i>type</i> )	Dereference	
*	Address (of operand)	
&	Determine size in bytes on this implementation	
sizeof		right-to-left
* / %	Multiplication/division/modulus	left-to-right

+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right

? :	Ternary conditional	right-to-left
=	Assignment	
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^=  =	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

It is good to know precedence and associativity rules, but the best thing is to use brackets, especially for less commonly used operators (operators other than +, -, \*.. etc). Brackets increase the readability of the code as the reader doesn't have to see the table to find out the order.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## **Expression Evaluation in C :**

In c language expression evaluation is mainly depends on priority and associativity. An expression is a sequence of operands and operators that reduces to a single value. For example, the expression, 10+15 reduces to the value of 25.

An expression is a combination of variables constants and operators written

according to the syntax of C language. every expression results in some value of a certain type that can be assigned to a variable.

### **Priority**

This represents the evaluation of expression starts from "what" operator.

### **Associativity**

It represents which operator should be evaluated first if an expression is containing more than one operator with same priority.

<b>Operator</b>	<b>Priority</b>	<b>Associativity</b>
{}, (), []	1	Left to right
++, --, !	2	Right to left
*, /, %	3	Left to right
+, -	4	Left to right
<, <=, >, >=, ==, !=	5	Left to right
&&	6	Left to right
	7	Left to right
?:	8	Right to left
=, +=, -=, *=, /=, %=	9	Right to left

---

### Example 1:

10 - 3 % 8 + 6 / 4

10 - 3 + 6 / 4

10 - 3 + 1

7 + 1

8

### Example 2:

17 - 8 / 4 \* 2 + 3 - ++a

17 - 8 / 4 \* 2 + 3 - 6

17 - 2 \* 2 + 3 - 6

17 - 4 + 3 - 6

13 + 4 - 6

16 - 6

10

### Types of Expression Evaluation in C

Based on the operators and operators used in the expression, they are divided into several types. Types of Expression Evaluation in C are:

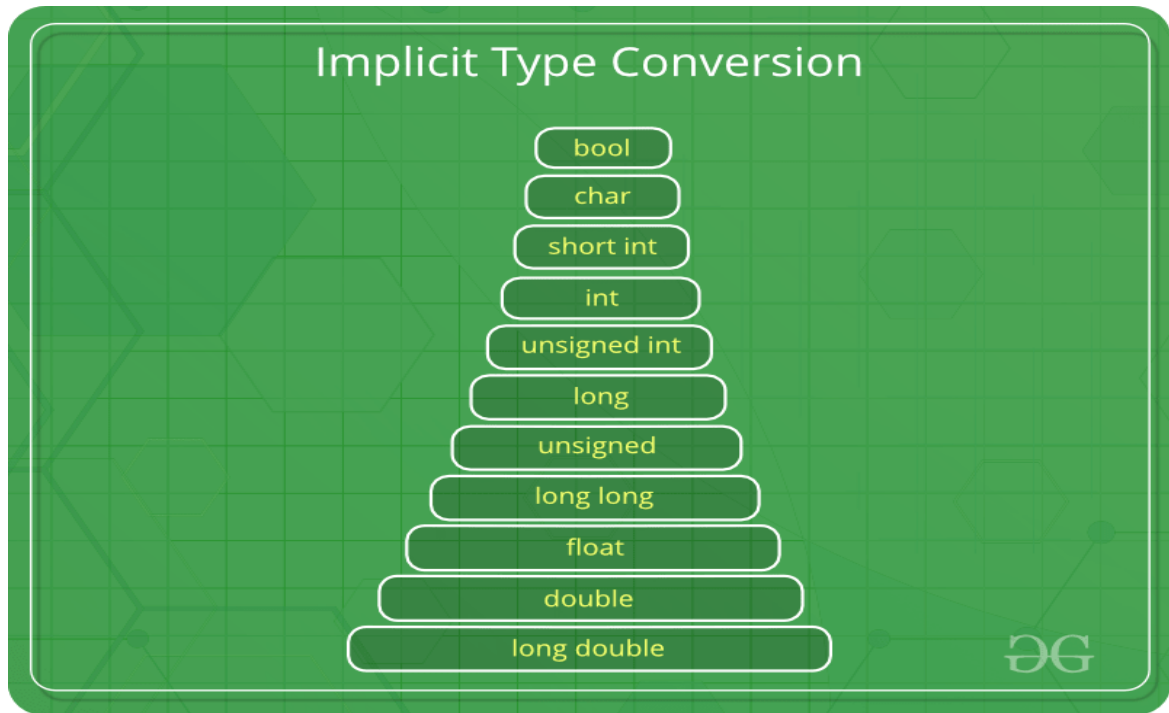
- **Integer expressions** – expressions which contains integers and operators
- **Real expressions** – expressions which contains floating point values and operators
- **Arithmetic expressions** – expressions which contain operands and arithmetic operators
- **Mixed mode arithmetic expressions** – expressions which contain both integer and real operands
- **Relational expressions** – expressions which contain relational operators and operands
- **Logical expressions** – expressions which contain logical operators and operands
- **Assignment expressions and so on...** – expressions which contain assignment operators and operands

## **Type Conversion in C**

Last Updated: 25-11-2020

A type cast is basically a conversion from one type to another. There are two types of type conversion:

### 1. **Implicit Type Conversion :**



Also known as ‘automatic type conversion’.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loss of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

- 
- **bool -> char -> short int -> int ->**
- **unsigned int -> long -> unsigned ->**
- **long long -> float -> double -> long double**

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

### **Example of Type Implicit Conversion:**

// An example of implicit conversion

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

```
int main()
```

```
{
```

```
int x = 10; // integer x

char y = 'a'; // character c

// y implicitly converted to int. ASCII

// value of 'a' is 97

x = x + y;

// x is implicitly converted to float

float z = x + 1.0;

printf("x = %d, z = %f", x, z);

return 0;

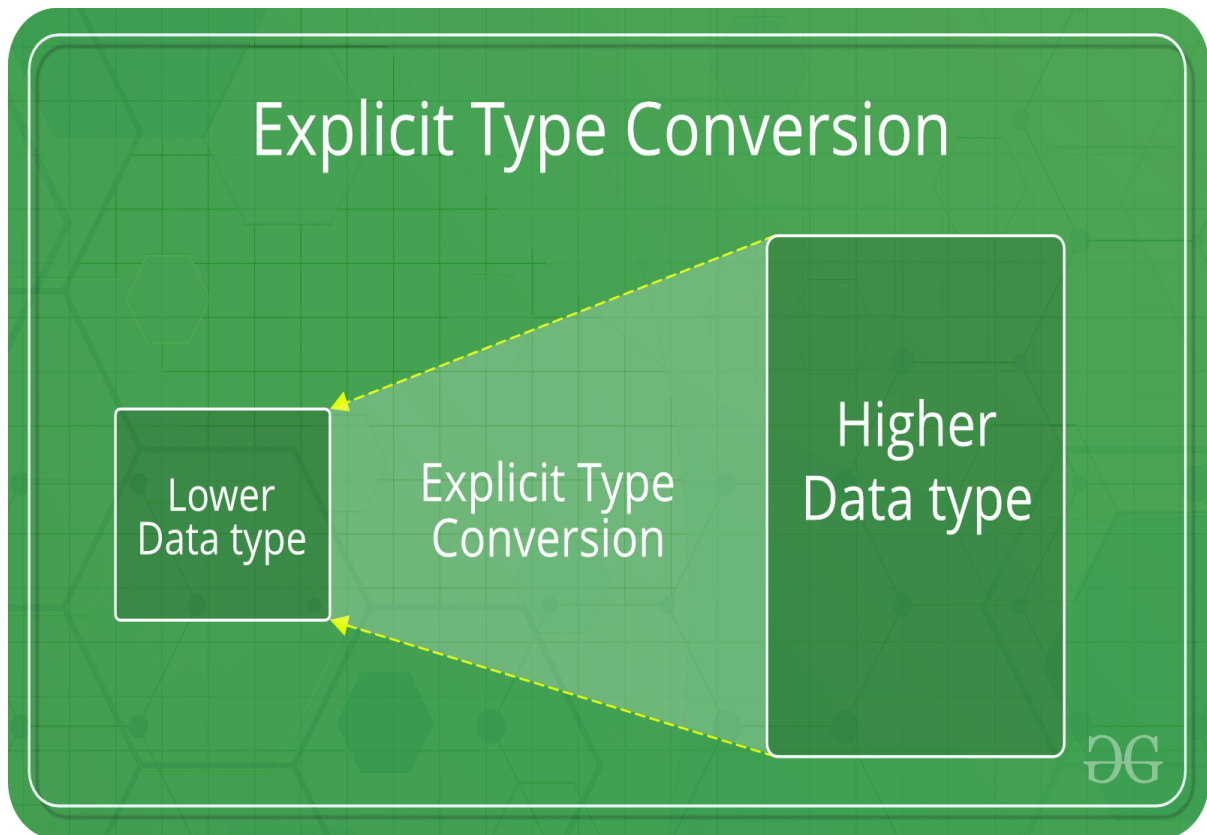
}
```

Output:

```
x = 107, z = 108.000000
```

## 2. Explicit Type Conversion:





This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type.

The syntax in C:

```
(type) expression
```

Type indicated the data type to which the final result is converted.

// C program to demonstrate explicit type casting

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

```
int main()
```

```
{
```

```
    double x = 1.2;
```

```
    // Explicit conversion from double to int
```

```

    int sum = (int)x + 1;

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

    return 0;
}

```

Output:

```
sum = 2
```

### Advantages of Type Conversion :

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps us to compute expressions containing variables of different data types.

## C Input and Output functions:

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

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

In this tutorial, we will learn about such functions, which can be used in our program to take input from user and to output the result on screen.

All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.

## scanf() and printf() functions

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

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

```
void main()
```

```
{
```

```
    // defining a variable
```

```

int i;

/*
    displaying message on the screen
    asking the user to input a value
*/

printf("Please enter a value...");

/*
    reading the value entered by the user
*/

scanf("%d", &i);

/*
    displaying the number as output
*/

printf( "\nYou entered: %d", i);
}

```

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

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

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number
%c	To scan or print a character

%s	To scan or print a character string. The scanning ends at whitespace.

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

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

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

```
int i = printf("studytonight");
```

In this program printf("studytonight"); will return 12 as result, which will be stored in the variable i, because studytonight has 12 characters.