

C<sub>t-1</sub>

---

# Introduction to C Programming

# HISTORY

# History of 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.

Language	Year	Developed By
Algol(Algorithmic language)	1960	International Group
BCPL(Basic combined programming Language)	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie

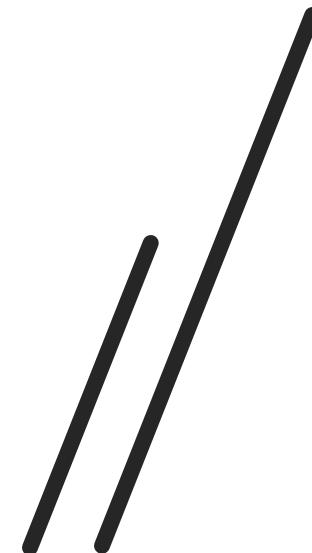
# Features of C language

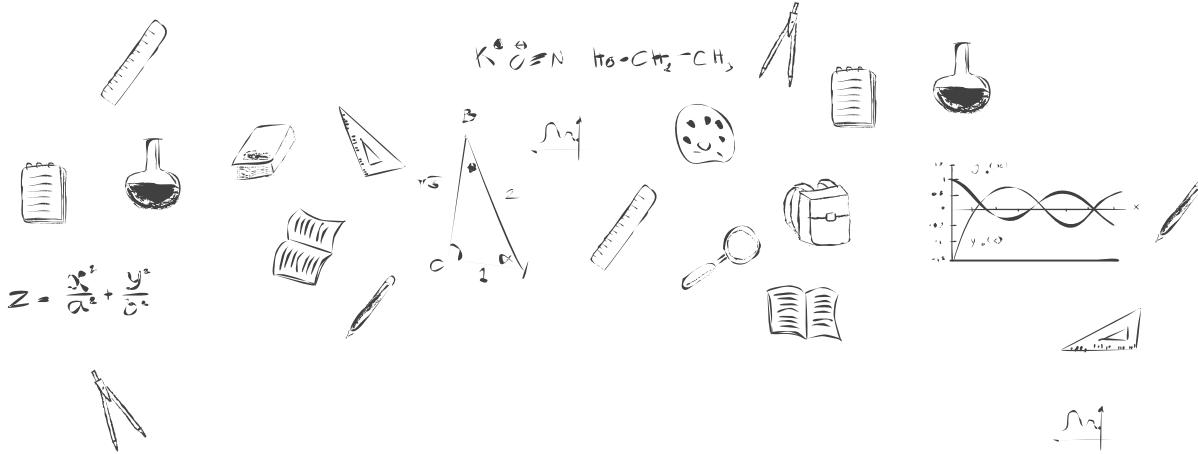
# Features of C language

- Middle level language
- It has rich set of Operators
- Program execution are fast and efficient
- small language(32 Keywords)
- Varieties of data types are available
- Separate compilation of functions is possible and such functions can be called by any C program
- Structured language because of its predefine structure and due to presence of control structure.
- Free form of language.
- Case-sensitive language.
- Procedural language.

# C Programming

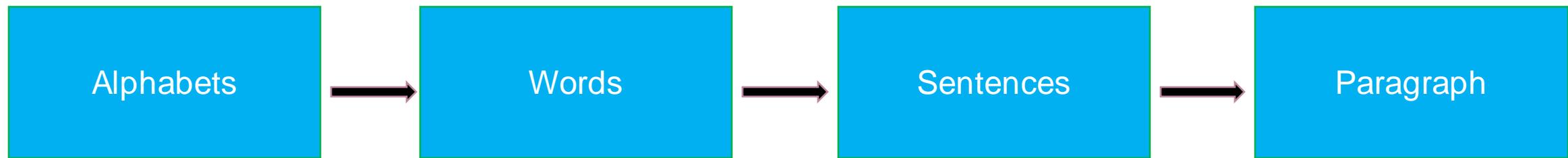
**Introduction to C  
Programming**



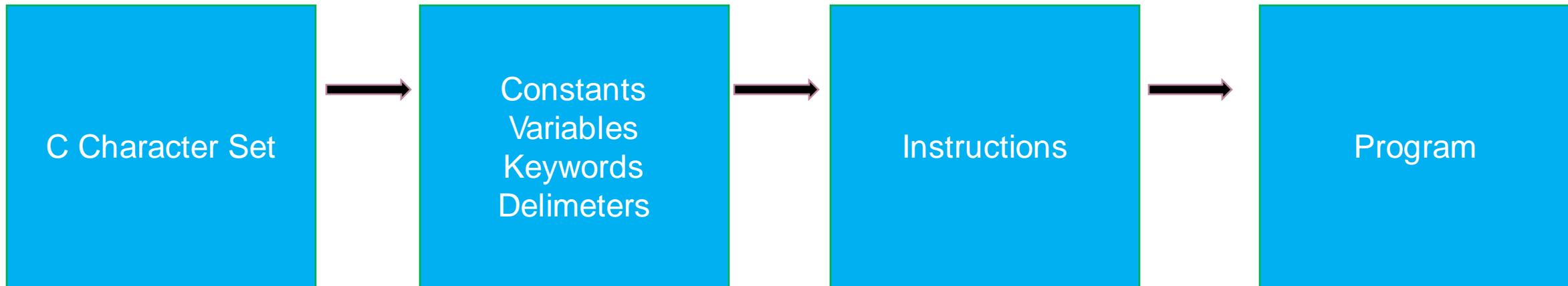


# Steps to learn C Programming

# Steps in learning English language:



# Steps in learning C language:



# The C Character Set

Alphabets

a.....z

A.....Z

Digits

0.....9

Special Symbols

@ # \$ % ^ & \* ( ) , . ‘ “ \

# C Programming



## Keywords

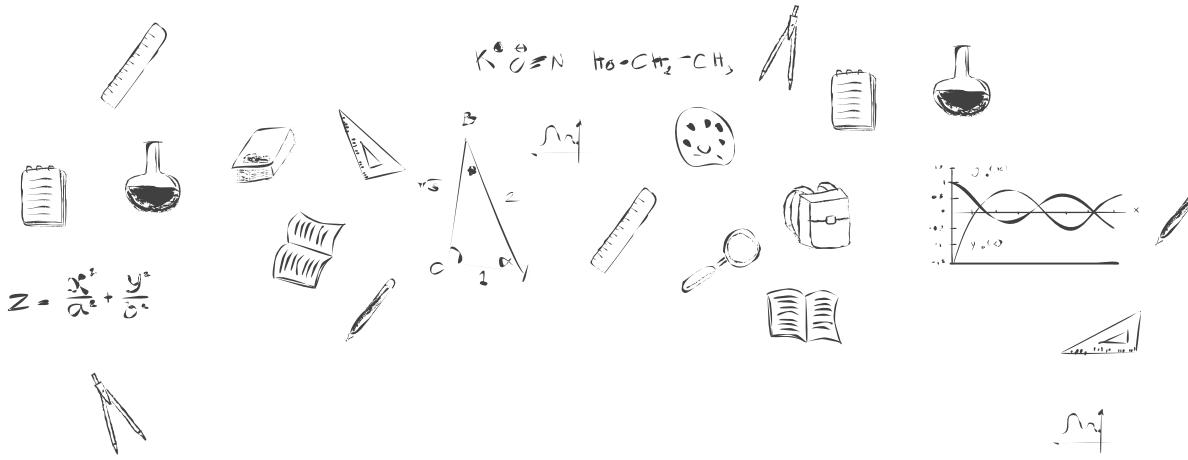
# **Keywords:**

The keywords are also called ‘Reserved words’.

Keywords are the words whose meaning has already been explained to the C compiler.

The keywords cannot be used as variable names.

There are only 32 keywords available in C.



# List of keywords

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

# C Programming

## Constants

# **Constant/Literals:**

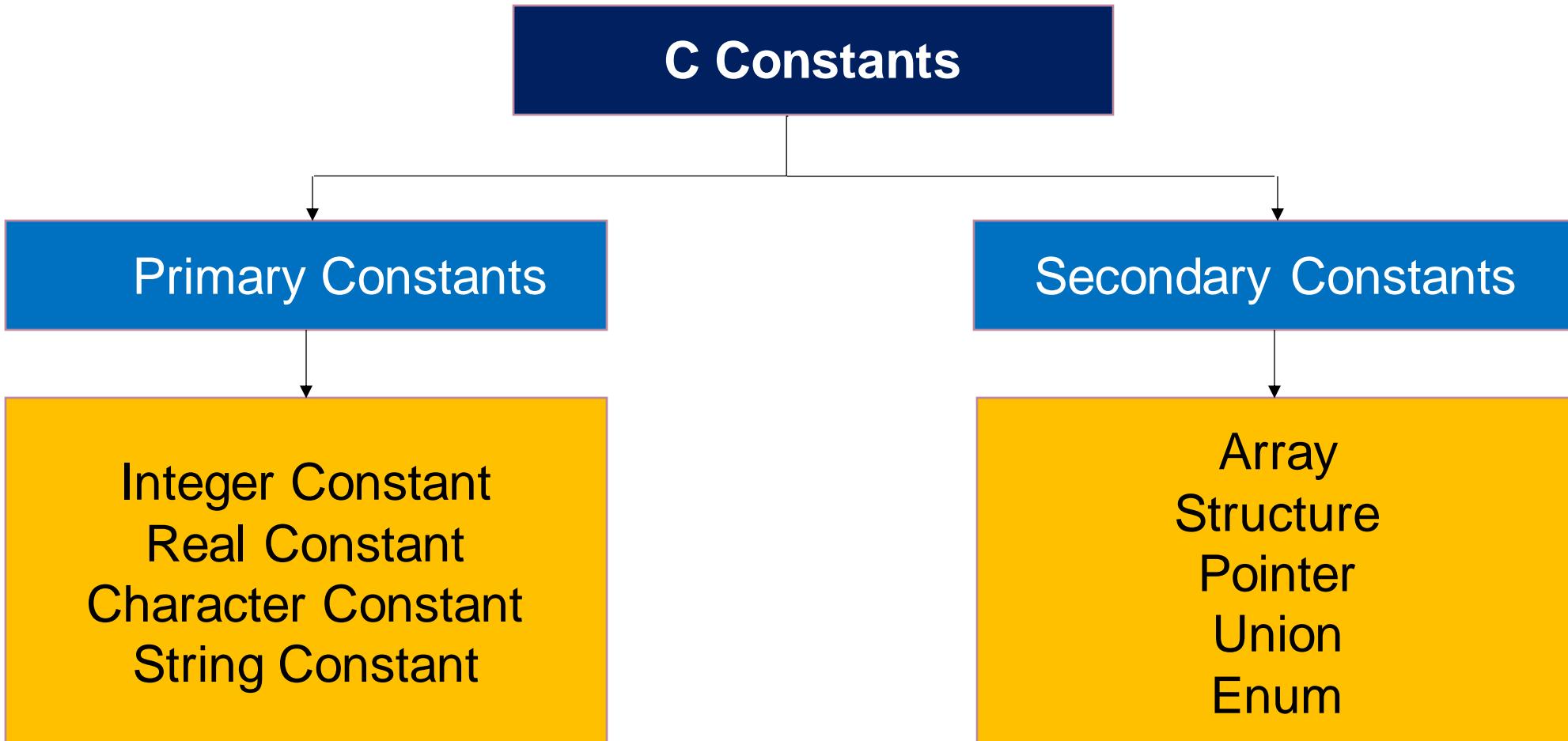
A constant is an entity that doesn't change.

Examples: 3,764,3.5,'a' etc.

# **Types of C Constants:**

C constants can be divided into two major categories:

- (a) Primary Constants
- (b) Secondary Constants



# C Programming



## Integer Constants

# Integer Constants:

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. It can be either positive or negative.

4. If no sign precedes an integer constant it is assumed to be positive.
5. No commas or blanks are allowed within an integer constant.
6. The allowable range for integer constants is **-32768 to 32767**.

Example: 426, +782, -80, 0

These are further divided into three types depending on the number systems they belong to.

They are:

- i. Decimal integer constants
- ii. Octal integer constants
- iii. Hexadecimal integer constants

A **decimal integer constant** is characterized by the following properties

- i. It is a sequence of one or more digits ([0...9], the symbols of decimal number system).
- ii. It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- iii. Commas and blank spaces are not permitted.
- iv. It should not have a period as part of it.

Some examples of valid decimal integer constants:

456

-123

Some examples of invalid decimal integer constants:

4.56 - Decimal point is not permissible

1,23 - Commas are not permitted

An **octal integer constant** is characterized by the following properties:

- It is a sequence of one or more digits ([0...7], symbols of octal number system).
- It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the digit 0.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid octal integer constants:

0456

-0123

+0123

Some examples of invalid octal integer constants:

04.56 - Decimal point is not permissible

04,56 - Commas are not permitted

x34 - x is not permissible symbol

568 - 8 is not a permissible symbol

An **hexadecimal integer constant** is characterized by the following properties

- It is a sequence of one or more symbols ([0...9][A....F], the symbols of Hexadecimal number system).
- It may have an optional + or - sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the symbols 0X or 0x.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid hexadecimal integer constants:

0x456

-0x123

0x56A 0

0XB78

Some examples of invalid hexadecimal integer constants:

0x4.56 - Decimal point is not permissible

0x4,56 - Commas are not permitted.

# C Programming



## Real Constants

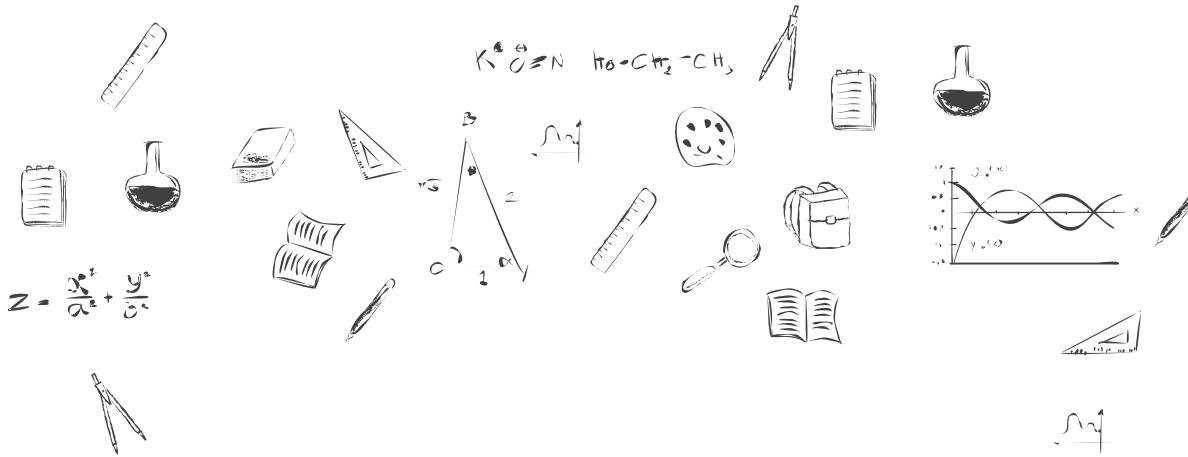
## Real Constants:

Real constants are often called Floating Point constants. These constants must have a decimal point.

The real constants could be written in two forms.

(a). Fractional or Floating Point form

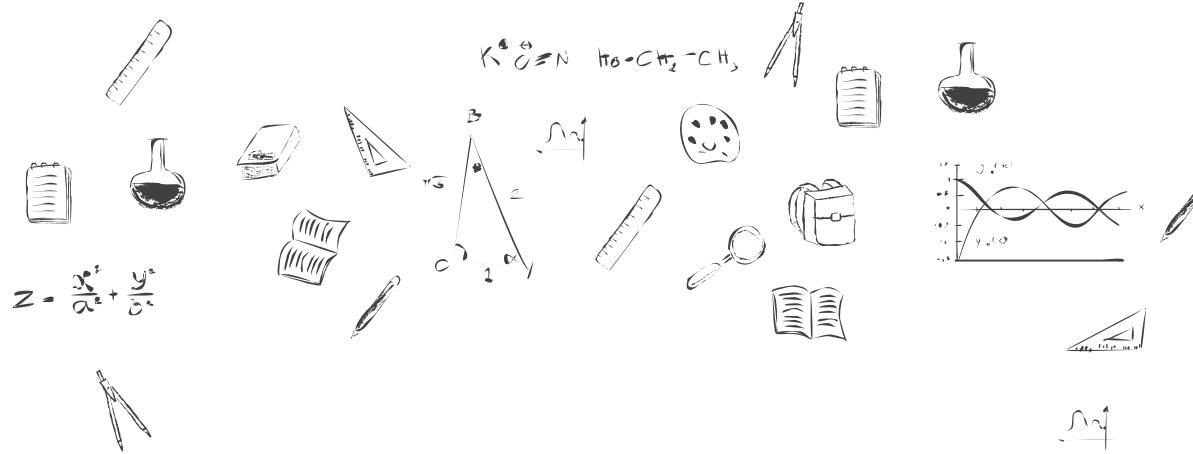
(b). Exponential form



# Fractional or Floating Point form

1. A floating point constant must have at least one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive.
5. No commas or blanks are allowed within a real constant.

**Example: +325.34, 426.0, -32.76,**



# Exponential form

**3.4 e^12**

1. The mantissa part and the exponential part should be separated by a letter e.
2. The mantissa part may have a positive or negative sign.
3. Default sign of mantissa part is positive.

4. The exponent must have at least one digit, which must be a positive or negative integer.
5. Default sign of exponent is positive.
6. Range of real constants expressed in exponential form is -3.4e^38 to 3.4e^38.

Example: **+3.2e^-5, 4.1e^8, -0.2e^+3, -3.2e^-5**

# C Programming



## Character Constants

A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.

Both the inverted commas should point to the left. For example, 'A' is a valid character constant whereas 'A' is not.

The maximum length of a character constant can be 1 character.

Examples:

'A'

'c'

'5'

'='

Ex.: 'A'

'T'

'5'

'='

# String Constants

A string constant is a sequence of alphanumeric characters enclosed in double quotes whose maximum length is 255 characters

Following are the examples of valid string constants:

- “My name is Krishna”
- “Bible”
- “Salary is 18000.00”

Following are the examples of invalid string constants:

My name is Krishna - Character are not enclosed in double quotation marks.

“My name is Krishna - Closing double quotation mark is missing.

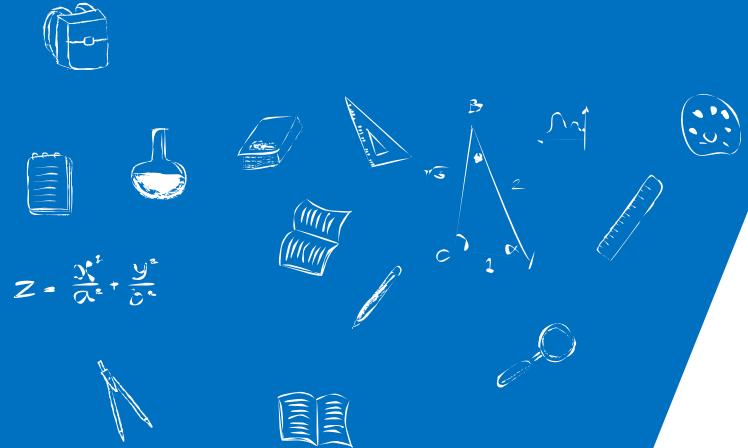
‘My name is Krishna’ - Characters are not enclosed in double quotation marks

# Delimiters

This is symbol that has syntactic meaning and has got significance. These will not specify any operation to result in a value. C language delimiters list is given below

Symbol	Name	Meaning
#	Hash	Pre-processor directive
,	comma	Variable delimiter to separate variable
:	colon	label delimiter
;	Semicolon	statement delimiter
( )	parenthesis	used for expressions
{ }	curly braces	used for blocking of statements
[ ]	square braces	used along with arrays

# C Programming



# Variable

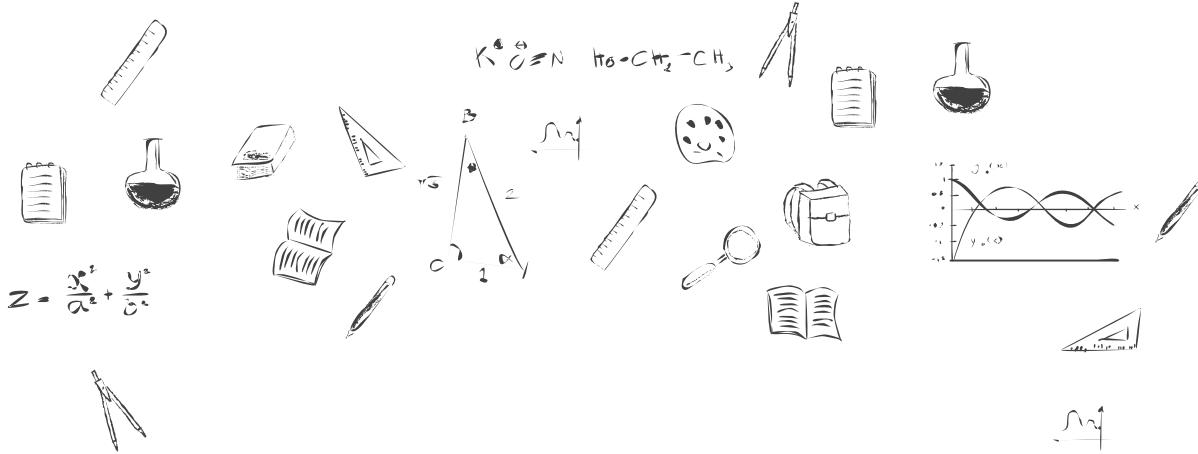
## **Variable/Identifiers:**

An entity that may change during program execution is called a variable.

Variable names are names given to locations in memory.

These locations can contain integer, real or character constants.

The types of variables that it can support depend on the types of constants that it can handle. This is because a particular type of variable can hold only the same type of constant.



# Rules to Declare Variable

1. A variable name is any combination of 1 to 31 alphabets, digits or Underscores.

2. The first character in the variable name must be an alphabet or underscore but it should not be a digit.

Valid variable names: Sum, a1, \_temp, TOTAL

Invalid variable names: 5a, #s

3. No commas or blanks are allowed within a variable name.

Invalid variables:    Total,number    sum of numbers

valid Variables:    Total\_number    sum\_of\_numbers

4. No special symbol other than an underscore can be used in a variable name.

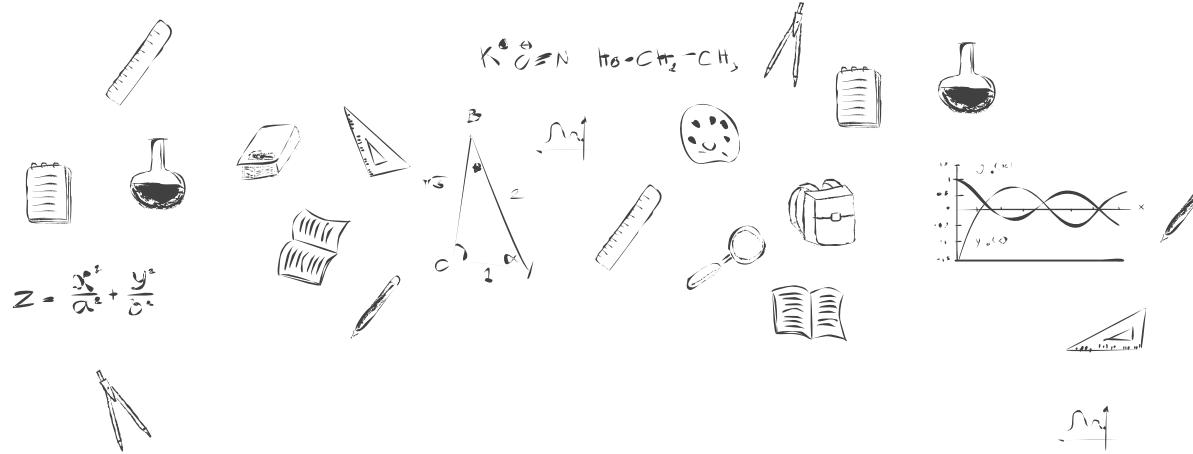
Valid variable Names: Total\_number sum\_of\_numbers

Invalid variable Names: Total#number sum@of-numbers

5. keywords cannot be used as variable names.

Valid variable Names: INT, LONG

Invalid variable Names: int, float, void etc.



# Syntax for Variable Declaration

Data type variable\_name;

Examples:

int a;

float b;

char c;

double d;



# **Programming for Problem Solving**

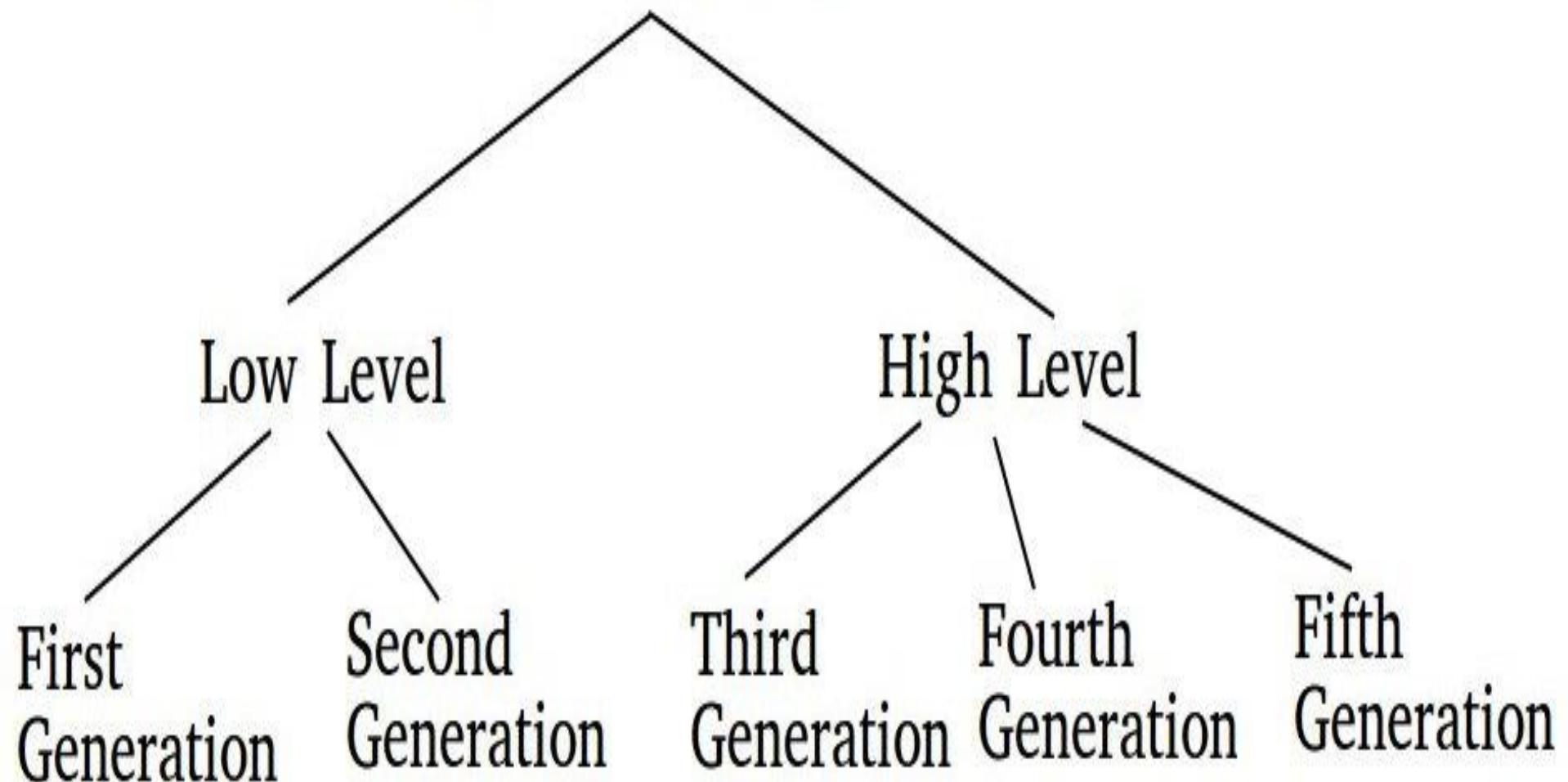
# **(21CSS101J)**

Evolution of  
Programming

&

Languages

# Programming Languages



# **First Generation Language :**

- The first generation languages are also called machine languages/ 1G language. This language is machine dependent. The machine language statements are written in binary code (0/1 form) because the computer can understand only binary language.

## **Advantages :**

1. Fast & efficient as statements are directly written in binary language.
2. No translator is required.

## **Disadvantages :**

1. Difficult to learn binary codes.
2. Difficult to understand – both programs & where the error occurred.

OPERAND  
OPERATOR

OPERAND	OPERATOR
00	ADDITION
01	SUBTRACTION
10	MULTIPLICATION
11	DIVISION

# Second Generation Language :

- The second generation languages are also called assembler languages/ 2G language. Assembly language contains human readable notations that can be further converted to machine language using an assembler.
- **Assembler** – converts assembly level instructions to machine level instructions.
- Programmers can write the code using symbolic instruction codes that are meaningful abbreviations of mnemonics. It is also known as low level language.

## **Advantages :**

1. Reading and writing programs became easier .
2. Programs became simple, efficient and less complex.
3. It is easier to understand as compared to machine language.
2. Modifications are easy.
3. Correction & location of errors are easy.

## **Disadvantages :**

1. Assembler is required.
2. This language is architecture/machine dependent , different instruction set for different machines.

OPERAND	OPERATOR	NAMES
00	ADDITION	ADD()
01	SUBTRACTION	SUB()
10	MULTIPLICATION	MUL()
11	DIVISION	DIV()

- Names were given to numerical opcodes.
- Assembler translates from assembly to machine language.
- As the language became easier, programmer could perform more complex tasks increasing the size & complexity of program.

# Third Generation Language :

- The third generation is also called procedural language /3 GL. It consists of use of series of English-like words that human can understand easily, to write instructions. Its also called High Level Programming Language. For execution, program in this language needs to be translated into machine language using Compiler/ Interpreter. Example of this type of languages are : C, PASCAL, FORTRAN, COBOL etc.

## **Advantages :**

1. Use of English-like words makes it human understandable language.
2. Lesser number of lines of code as compared to above 2 languages.
3. Same code can be copied to other machine & executed on that machine by using compiler specific to that machine.

## **Disadvantages :**

1. Compiler/ interpreter is needed.
2. Different compilers are needed for different machines.

# Fourth Generation Language :

- The **fourth-generation languages**, or **4GL**, are languages that consist of statements similar to statements in a human language. Fourth generation languages are commonly used in database programming and scripts examples include [Perl](#), [PHP](#), [Python](#), [Ruby](#), and [SQL](#).
- These languages are also human friendly to understand.

## **Advantages :**

1. Easy to understand & learn.
2. Less time required for application creation.
3. It is less prone to errors.

## **Disadvantages :**

1. Memory consumption is high.
2. Has poor control over Hardware.
3. Less flexible.

# **Fifth Generation Language :**

- The fifth generation languages are also called as 5GL. It is based on the concept of artificial intelligence. It uses the concept that rather than solving a problem algorithmically, an application can be built to solve it based on some constraints, i.e., we make computers learn to solve any problem. Parallel Processing & superconductors are used for this type of language to make real artificial intelligence.
- Example : PROLOG, LISP etc.

## **Advantages :**

1. Machines can make decisions.
2. Programmer effort reduces to solve a problem.
3. Easier than 3GL or 4GL to learn and use.

## **Disadvantages :**

1. Complex and long code.
2. More resources are required & they are expensive too.

# Flow of C Program

# Flow of C Program

- The C program follows many steps in execution.
- To understand the flow of C program well, let us see a simple program first.

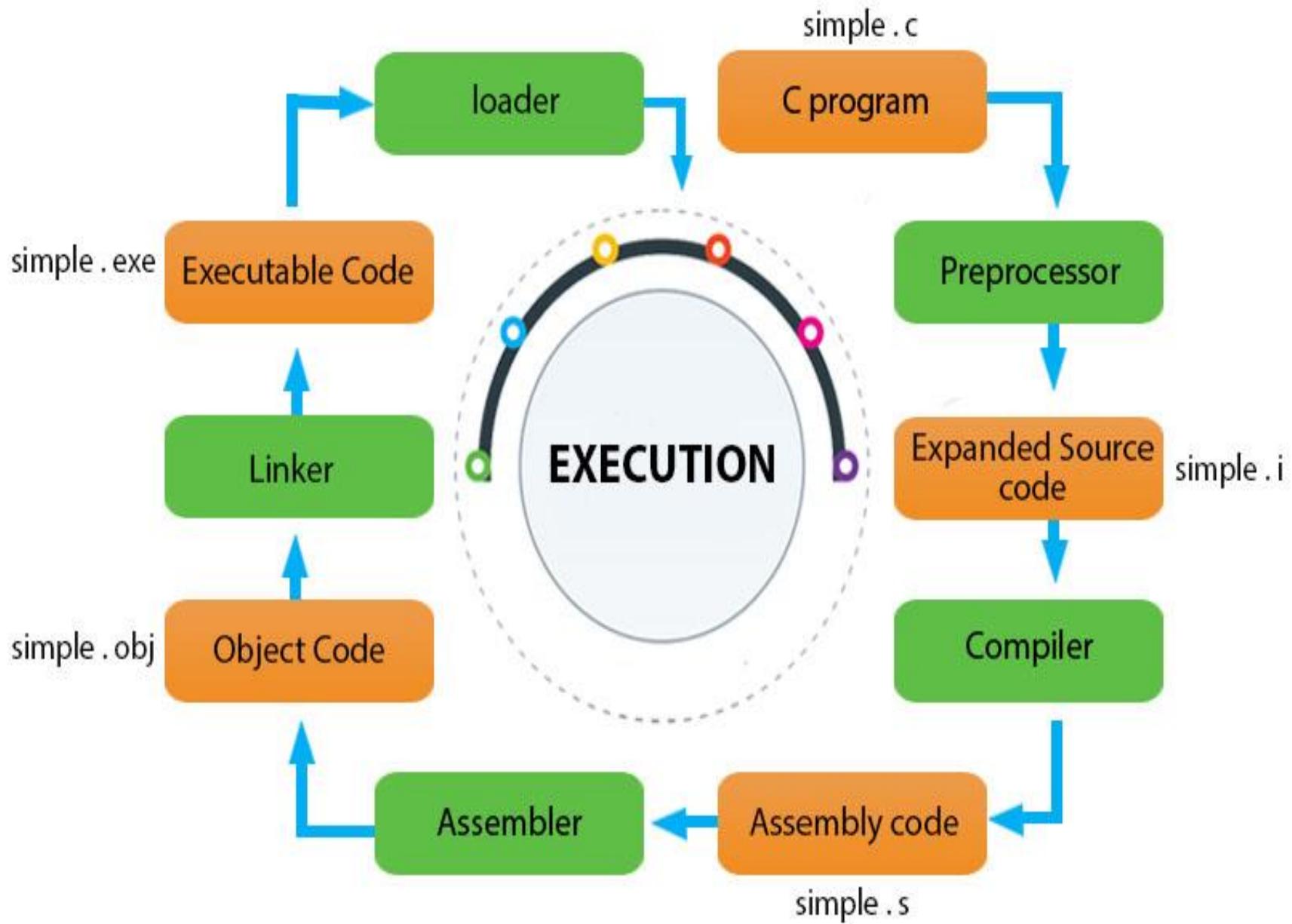
File: simple.c

```
#include <stdio.h>
void main()
{
    printf("Hello C Language");
    getch();
}
```

# Execution Flow

- Let's try to understand the flow of above program by the figure given below.
- 1) C program (source code) is sent to preprocessor first. The preprocessor is responsible to convert preprocessor directives into their respective values. The preprocessor generates an expanded source code.

- 2) Expanded source code is sent to compiler which compiles the code and converts it into assembly code.
- 3) The assembly code is sent to assembler which assembles the code and converts it into object code. Now a simple.obj file is generated.
- 4) The object code is sent to linker which links one or more object files generated by a compiler and combines into a single executable code. A simple.exe file is generated.
- 5) The executable code is sent to loader which loads it into memory and then it is executed. After execution, output is sent to console.



# Preprocessor

The preprocessor is responsible to convert preprocessor directives into their respective values.

The preprocessor generates an expanded source code.

# **COMPILER**

Compiler is a translator program, which converts the high level language program (i.e., source code) into its corresponding machine level language program (i.e. object code).

## High-level

```
int main()
{
    int x, y, z;
    x = 1;
    y = 2;
    z = x+y;
    return 0;
}
```

```
00001001001011100110011001101001011011000110010100001001
01010111001001100101001100010010111001100011001000100000
01100011011011110110110101110000011010010110110001100101
00110110010101100011011101000110100101101111011011100000
01110100001000100000101000001001001011100110000101101100
101000000100100101110011001110110110001101111011000100110
01101110000010100000100100101110011101000111100101110000
1001011011100010110000100011011001110101011011100110
00001001001011100111000001110010011011110110001100001001
1001011011100011101000010100000100100100001001000110101
01010101000100010001100100000001100000000101000001001
01010111001101110000001011000010110100110001001100100011
0000101000001001000100010000011000000001010000010010001111
001100100000001100001000010100000100101101011011110111
001100000000010100000100101110011011101000010000000100101
011001110000000101101001100011000001011101000010100000
0010110000100101011100110000000010100000100101110011
1100010110110010010101100110000000010101001100100011
0010000001011011001001010110011000000010110100110010
000000000101000001001011010001100000001011010010
010111010010110000100101010111001100010000101000001001
1111001100000010110000100101011011100110001001011000010
01110100001000000010010101111001100000010110001011011
1000010111010000101000001001011010101111011101100010
00001010000010010110001000100000001011100100110000110001100011110111
000000000101000101110010011000010100000100111010000001010
0000100101110010011000101011100100011011101000010100001001011101000000110
01100110010100110000010100001001110011000011011001010111010000001001000000
011011010100110000101101001011100010110000100110001100100100110001001011010110
110101100001011010010110111000001010000010010110011001001011011100111010000001001
00100010010001101000011001110100010000000101000010001110101010100101001001000000011
0010001011100011100000101110001100010010000001010000100011101010101001010010010000001010
```

# INTERPRETER

Interpreter is a translator program, which converts the high-level language program (i.e., source code) into its corresponding machine level language program (i.e. object code).

It converts source code into object code, statement-by-statement, i.e., first it reads a sentence and then translates it into machine level language, and this process is repeated till the last sentence.

# **COMPILER vs INTERPRETER**

Compiler reads the source code and then converts the whole program into machine language at a time.

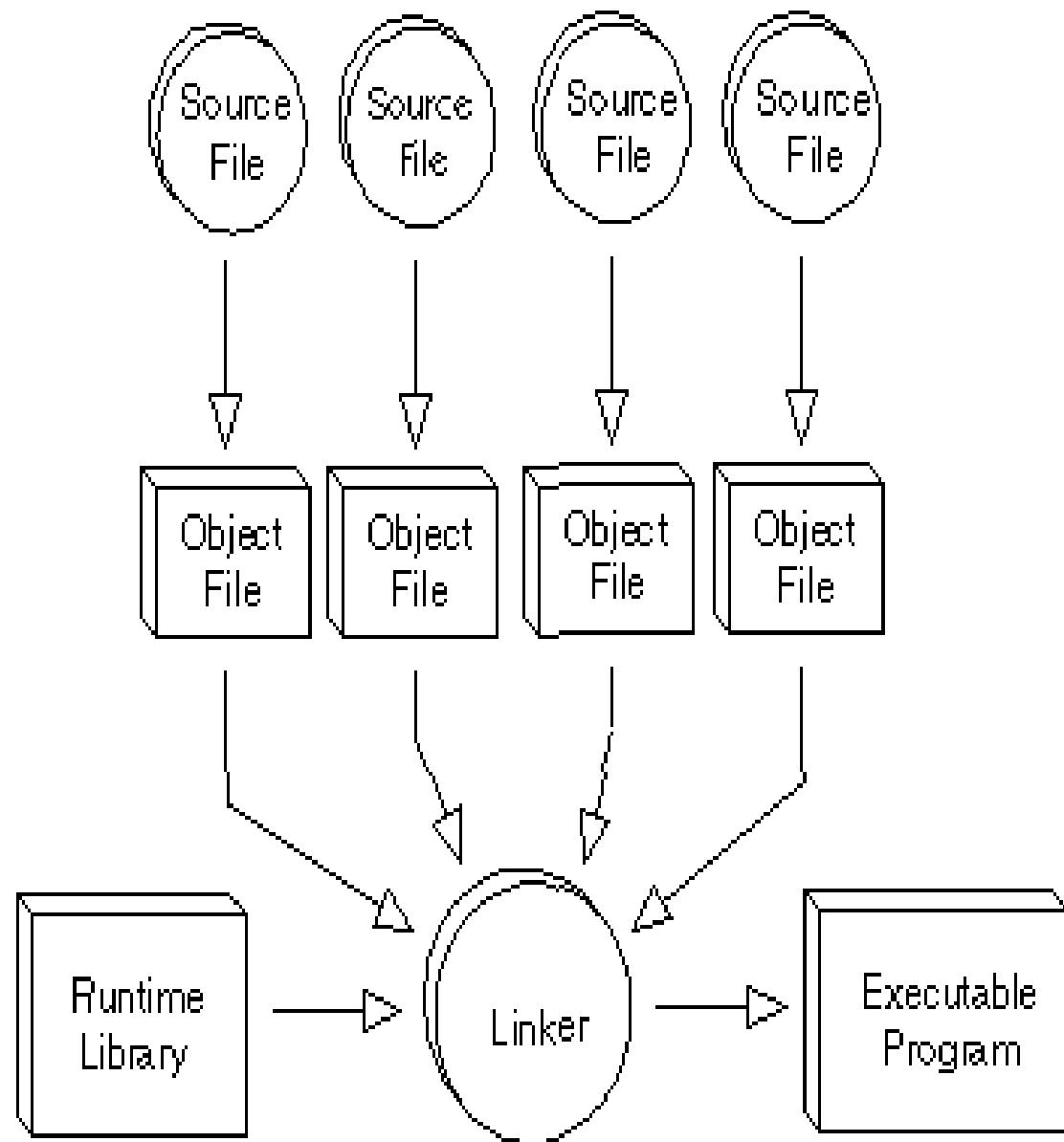
Interpreter converts source code into object code, instructions by- instructions i.e., first it read a instruction and then translates it into machine level language, and this process is repeated till the last instruction.

# Assembler

Assembler is a translator program, which converts the assembly language program (i.e., source code) into its corresponding machine level language program.

# Linker

A linker or link editor is a program that takes one or more object files generated by a compiler and combines them into a single executable program.



# Loader

Loader is the part of an operating system that is responsible for loading programs from executables (i.e., executable files) into memory, preparing them for execution and then executing them.

# Structure of a ‘C’ Program

# Structure of a ‘C’ Program

The Complete structure of C program is The basic components of a C program are:

- Preprocessor Statements
- Global Declarations
- main()
- pair of braces { }
- declarations and statements
- user defined functions

# Preprocessor Statements:

- These statements begin with # symbol.
- They are called preprocessor directives.
- These statements direct the C preprocessor to include header files and also symbolic constants in to C program.

Some of the preprocessor statements are

- `#include<stdio.h>`: for the standard input/output functions.
- `#include<test.h>` : for file inclusion of header file Test.
- `#define NULL 0`: for defining symbolic constant `NULL = 0` etc.

# Global Declarations

- Variables or functions whose existence is known in the main() function and other user defined functions are called global variables (or functions) and their declarations is called global declaration. This declaration should be made before main().

# main()

- As the name itself indicates it is the main function of every C program.
- Execution of C program starts from main () .
- No C program is executed without main() function.
- It should be written in lowercase letters and should not be terminated by a semicolon.
- It calls other Library functions user defined functions.
- There must be one and only one main() function in every C program.

# Braces

- Every C program uses a pair of curly braces {},.
- The left brace indicates beginning of main() function.
- On the other hand, the right brace indicates end of the main() function.
- The braces can also be used to indicate the beginning and end of user-defined functions and compound statements.

# Declarations

- It is part of C program where all the variables, arrays, functions etc., used in the C program are declared and may be initialized with their basic data types.
- Statements: These are instructions to the specific operations. They may be input-output statements, arithmetic statements, control statements and other statements. They are also including comments.

# User-defined functions

- These are subprograms.
- Generally, a subprogram is a function, and they contain a set of statements to perform a specific task.
- These are written by the user; hence the name is user-defined functions.
- They may be written before or after the main() function.

- It is word which tells to the compiler that what types of data can be stored in a particular variable.

### Data types

PRIMARY /built-in data types  
(Integer,real ,character)

Secondary/Derived data types  
(array,pointer)

User defined data types  
(structure,union,enumeration)

#### **Integers:**

Keyword	Int
Range	-32768 to +32767
Bytes reserved	2 bytes= 16 bits
Format specifier	%d

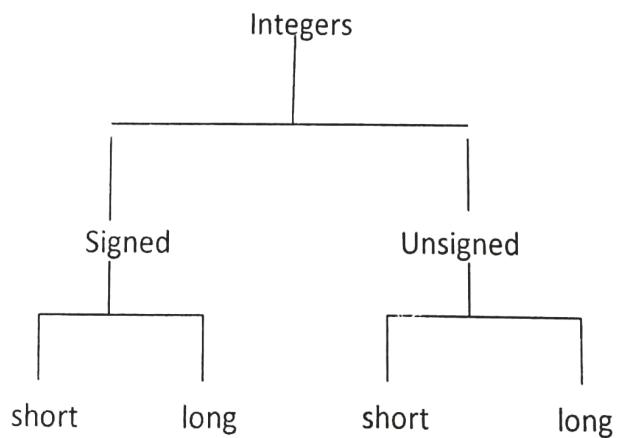
#### **Real(float):**

Keyword	Float
Range	$3.4 \times 10^{-38}$ to $3.4 \times 10^{38}$
Bytes reserved	4 bytes=32 bits
Format specifier	%f or %e

#### **Character:**

Keyword	Char
Range	-128 to 127
Bytes reserved	1 byte=8 bits
Format specifier	%c

#### **Further subdivisions of data types:**



**Integer signed short:**

Keyword	int
Range	-32768 to +32767
Bytes reserved	2 bytes= 16 bits
Format specifier	%d

**Integer signed long:**

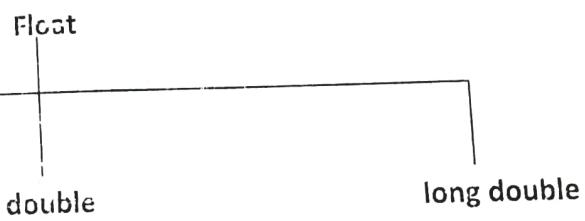
Keyword	long int/long
Range	-2147483648 to +2147483647
Bytes reserved	4 bytes
Format specifier	%ld

**Integer unsigned short:**

Keyword	unsigned int
Range	0 to 65535
Bytes reserved	2 bytes
Format specifier	%u

**Integer unsigned long:**

Keyword	unsigned long int/ Unsigned int
Range	0 to 4294967295
Bytes reserved	4 bytes
Format specifier	%lu

**float:**

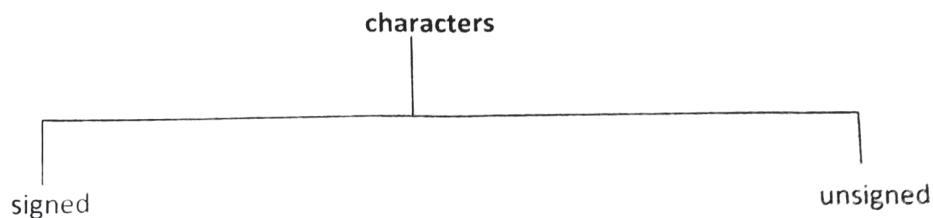
Keyword	float
Range	$3.4 \times 10^{-38}$ to $3.4 \times 10^{38}$
Bytes reserved	4 bytes=32 bits
Format specifier	%f or %e

**double:**

Keyword	double
Range	$1.7 \times 10^{-308}$ to $1.7 \times 10^{308}$
Bytes reserved	8 bytes
Format specifier	%lf

**long double:**

Keyword	long double
Range	$1.7 \times 10^{-4932}$ to $1.7 \times 10^{4932}$
Bytes reserved	10 bytes
Format specifier	%lf

**signed:**

Keyword	signed char
Range	-128 to 127
Bytes reserved	1 byte=8 bits
Format specifier	%c

**Unsigned:**

Keyword	Unsigned char
Range	0 to 255
Bytes reserved	1 byte=8 bits
Format specifier	%c

## Storage class

- Storage class is word that tells to the compiler storage location, default value, scope and life time of a variable.
- There are following storage classes which can be used in a C Program
  - auto
  - register
  - static
  - extern

**Automatic storage class**

KEYWORD	Auto
DEFAULT VALUE	Garbage value
STORAGE	RAM(memory)
SCOPE OF A VARIABLE	Within the block or function in which it is declared
LIFE OF VARIABLE	Till the control is within the block or function in which it is declared
DECLARATION	auto type-of-variable variable list.
Ex:	auto int l; auto float j; auto char k;

# C Programming

Translators

In c Programming

# **COMPILER**

Compiler is a translator program, which converts the high level language program (i.e., source code) into its corresponding machine level language program (i.e. object code).

## High-level

```
00001001001011100110011001101001011011000110010100001001  
01010111001001100101001100010010111001100011001000100000  
011000110110111101101101110000011010010110110001100101  
0011011001010110001101110100011010010110111101101110000  
01110100001000100000101000001001001011100110000101101100  
10100000100100101110011001110110110001101111011000100110  
01101110000010100000100100101110011101000111100101110000  
1001011011100010110000100011011001110101011011100110  
00001001001011100111000001110010011011110110001100001001  
1001011011100011101000010100000100100100001001000110101  
0101010100010001100100000001100000000101000001001  
01010111001101110000001011000010110100110001001000110011  
000010100001001001000100100011010100000101001001001111  
00110010000000110001000010100000100101101011011110111  
001100000000010100000100101110011011101000010000000100101  
011001110000001011010011001000110000001011101000010100000  
0010110000100101011100110000000010100000100101110011  
1100010110110010010110001100111000000101101001100100011  
0010000001011011001001011000110011100000010110100110010  
000000000101000001001011011000110010000100000010110110010  
01011101001011000010010101101111001100010000101000001001  
1111001100000010110000100101011011100110001001011000010  
01110100001000000010010101101111001100000010110001011011  
100001011101000010100000100101101011110111011001100010  
0000101000001001011000100010000000101110010011000011001111011  
00000000010100010111001001100001100010011101000001010000010010  
000010010111001001100101011100110011110100110010100001010001001000110  
01100110010011000100111010000010010010111001100111101001100100000100000001010  
01101101011000010110100101110001011000010111001001100011001100010010000010010110110  
1101011000010110100101101110001011000010111001001100011001100010100110001001011010110  
00100010010001110100001101000011001110100010000000101000010001110100101010100101001000000011  
001000101110001110000010111000110001001000000101000010001110100101010010100100000001010
```

```
int main()  
{  
    int x, y, z;  
  
    x = 1;  
    y = 2;  
    z = x+y;  
  
    return 0;  
}
```

# INTERPRETER

Interpreter is a translator program, which converts the high-level language program (i.e., source code) into its corresponding machine level language program (i.e. object code).

It converts source code into object code, statement-by-statement, i.e., first it reads a sentence and then translates it into machine level language, and this process is repeated till the last sentence.

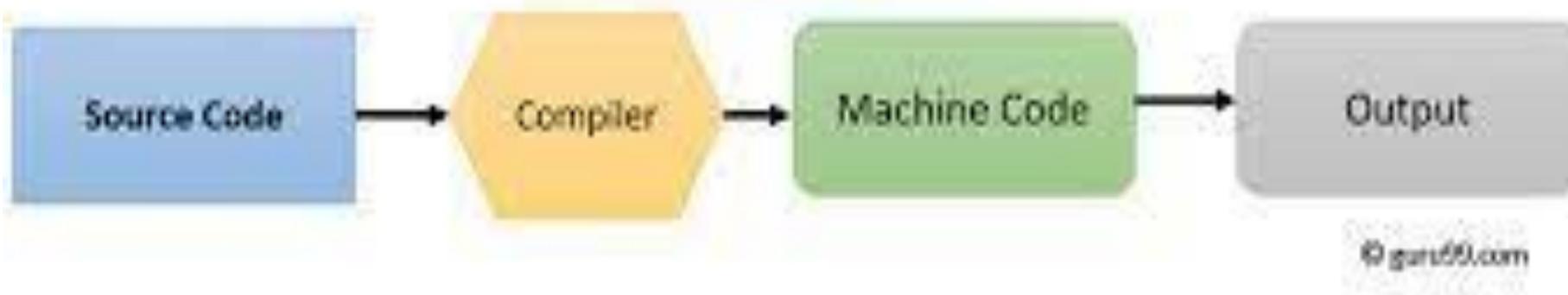
# **COMPILER vs INTERPRETER**

Compiler reads the source code and then converts the whole program into machine language at a time.

Interpreter converts source code into object code, instructions by-instructions i.e., first it read a instruction and then translates it into machine level language, and this process is repeated till the last instruction.

<b>COMPARISON</b>	<b>COMPILER</b>	<b>INTERPRETER</b>
<b>Input</b>	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
<b>Output</b>	It generates intermediate object code.	It does not produce any intermediate object code.
<b>Working mechanism</b>	The compilation is done before execution.	Compilation and execution take place simultaneously.
<b>Speed</b>	Comparatively faster	Slower
<b>Memory</b>	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
<b>Errors</b>	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
<b>Error detection</b>	Difficult	Easier comparatively
<b>Pertaining Programming languages</b>	C, C++, C#, Scala, typescript uses compiler.	PHP, Perl, Python, Ruby uses an interpreter.

### How Compiler Works



### How Interpreter Works



# Assembler

Assembler is a translator program, which converts the assembly language program (i.e., source code) into its corresponding machine level language program.

# Local Variable

- The variables which are declared within a function are known as local variable and they can not be accessed from outside of that function.

# Local Variable

The variables declared inside a block are automatic or local variables. The local variables exist only inside the block in which it is declared.

Let's take an example.

```
#include <stdio.h>
int main(void)
{
    for (int i = 0; i < 5; ++i)
    {
        printf("C programming");
    }
    // Error: i is not declared at this point
    printf("%d", i);
    return 0;
}
```

When you run the above program, you will get an error undeclared identifier i. It's because i is declared inside the for loop block. Outside of the block, it's undeclared.

Let's take another example.

```
int main()
{
    int n1; // n1 is a local variable to
            main()
}
void func()
{
    int n2; // n2 is a local variable to
            func()
}
```

In the above example, *n1* is local to *main()* and *n2* is local to *func()*. This means you cannot access the *n1* variable inside *func()* as it only exists inside *main()*. Similarly, you cannot access the *n2* variable inside *main()* as it only exists inside *func()*.

# Global Variable

- The variables which are declared out of all functions are known as global variables and they can be accessed from anywhere in the program.
- Global variables are static by default.
- Storage classes auto and register are not allowed in the declaration of global variables . It is only allowed for local variables.

# Global Variable

## Example 1: Global Variable

```
#include <stdio.h>
void display();
int n = 5; // global variable
int main()
{
    ++
n;
display();
return 0;
}
void display()
{
    ++
n;
printf("n = %d", n);
}
```

## Output

n = 7

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword extern is used in file2 to indicate that the external variable is declared in another file.

# Storage Class

- Storage class is word that tells to the compiler storage location, default value, scope and life time of a variable.
- There are following storage classes that can be used in a C program
  - automatic
  - external
  - static
  - Register

# Important points :

- **automic** and **register** storage class are allowed in the declaration of local variables only i.e it is not allowed in the declaration of global variables .
- **auto** storage class is the default storage class for all local variables.
- Static and extern storage class are allowed in the declaration of local variables and global variables both.
- **static** storage class is the default storage class for all Global variables.

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

# auto

- The **auto** storage class is the default storage class for all local variables.

```
{  
    int mount;  
    auto int month;  
}
```

- The example above defines two variables within the same storage class. 'auto' can only be used within functions, i.e., local variables.

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

### **Example 1:**

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

### **Output:**

garbage garbage garbage

## Example 2:

```
#include <stdio.h>
int main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
}
}
printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

## Output:

11 20 20 20 11

Example 3:

```
#include <stdio.h>
int main( )
{
auto int j = 1;
{
auto int j= 2;
{
auto int j = 3;
printf( " %d ", j);
}
printf( "\t %d ",j);
}
printf( "%d\n", j);
}
```

OUTPUT:  
3 2 1

## Example 4:

```
#include <stdio.h>
int main(void)
{
for (int i = 0; i < 5; ++i)
{
printf("C programming");
}
// Error: i is not declared at this point
printf("%d", i);
return 0;
}
```

When you run the above program, you will get an error undeclared identifier *i*. It's because *i* is declared inside the for loop block. Outside of the block, it's undeclared.

## Example 5:

```
int main()
{
    int n1; // n1 is a local variable to main()
}
void func()
{
    int n2; // n2 is a local variable to func()
}
```

Explanation: In the above example, *n1* is local to `main()` and *n2* is local to `func()`. This means you cannot access the *n1* variable inside `func()` as it only exists inside `main()`. Similarly, you cannot access the *n2* variable inside `main()` as it only exists inside `func()`.

Example 6:

```
#include<stdio.h>
void sum()
{
auto int a = 10;
auto int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i < 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

**Output:**

10 24 10 24 10 24

# Static

- The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
- The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
- In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

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

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

## Example 1:

```
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f ",c,i,f); // the initial default value of c, i, and f will be printed.
}
```

## Output

0 0 0.000000

Example 2:

```
#include<stdio.h>
void sum()
{
static int a = 10;
static int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i < 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

**Output:**

10 24 11 25 12 26

### Example 3 :

```
#include <stdio.h>
void func(void);
static int count = 5; /* global variable */
main()
{
    while(count--)
    {
        func();
    }
    return 0;
}
void func( void )
{
    static int i = 5; /* local static variable */
    i++;
    printf("i is %d and count is %d\n", i, count);
}
```

Output:

i is 6 and count is 4  
i is 7 and count is 3  
i is 8 and count is 2  
i is 9 and count is 1  
i is 10 and count is 0

#### Example 4:

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

#### Output

6 11

During the first function call, the value of *c* is initialized to 1. Its value is increased by 5.  
Now, the value of *c* is 6, which is printed on the screen.

During the second function call, *c* is not initialized to 1 again. It's because *c* is a static variable. The value *c* is increased by 5. Now, its value will be 11, which is printed on the screen.

# External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.
- Extern is used to increase scope of variable.

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

### **Example 1:**

```
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

### **Output:**

**error**

## **Example 2:**

```
#include <stdio.h>
int a;
int main()
{
// variable a is defined globally, the memory will not be allocated to a
Extern int a;
printf("%d",a);
}
```

## **Output**

0

### **Example 3(a):**

```
#include <stdio.h>
int a;
int main()
{
// we can not initialize the external variable within any block or
    method i.e.,We can only initialize the extern variable globally.
extern int a = 0;
printf("%d",a);
}
```

### **Output:**

compile time error

## Example 3(b):

```
#include <stdio.h>
extern int a = 20;
int main()
{
printf("%d",a);
}
```

Output:20

## Example 3 (c):

```
#include <stdio.h>
extern int a ;
int a= 20;
int main()
{
printf("%d",a);
}
```

Output:20

#### Example 4:

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

#### Explanation:

- In the above C program, if `extern int i` is removed, there will be an error “*Undefined symbol ‘i’*” because the variable `i` is defined after being used in `printf`. The `extern` specifier instructs the compiler that variable `i` has been defined and is declared here.
- If you change `extern int i;` to `extern int i = 5;` you will get an error “*variable i is initialized more than once*”.

### **Example 5:**

```
#include <stdio.h>
int main()
{
extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the program or not.
printf("%d",a);
}
int a = 20;
```

### **Output**

20

## Example 6:

```
extern int a;  
int a = 10;  
#include <stdio.h>  
int main()  
{  
    printf("%d",a);  
}  
int a = 20; // compiler will show an error at this line
```

Output:  
compile time error

# Register

- The register keyword is used to declare register variables.
- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is garbage value.
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler's choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local(within the block or function in which it is declared)	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

- Register variables were supposed to be faster than local variables.
- However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.
- Unless you are working on embedded systems where you know how to optimize code for the given application, there is no use of register variables.

- The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{  
register int miles;  
}
```

- The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it might be stored in a register depending on hardware and implementation restrictions.

## **Example 1:**

```
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial defa
ult value of a is garbage.
printf("%d",a);
}
```

## **Output:**

Garbage

## **Example 2:**

```
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable because register do not have address.
}
```

## **Output:**

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~
```

A well-documented program is a good practice as a programmer. It makes a program more readable and error finding become easier. One important part of good documentation is Comments.

- In computer programming, a comment is a programmer-readable explanation or annotation in the source code of a computer program
- Comments are statements that are not executed by the compiler and interpreter.

**In C there are two types of comments :**

1. Single line comment
2. Multi-line comment

## Comments

//

**Single line comment**

/\*

**Multi-line comment**

dg

### Single line Comment

Represented as // double forward slash

It is used to denote a single line comment. It applies comment to a single line only.

for example:

```
// single line comment
```

## **Example:**

```
// C program to illustrate

// use of multi-line comment

#include <stdio.h>

int main(void)

{

    // Single line Welcome user comment

    printf("Welcome to XYZ");

    return 0;

}
```

## **Output:**

Welcome to XYZ

## **Multi-line comment**

Represented as */\* any\_text \*/* start with forward slash and asterisk /\*) and end with asterisk and forward slash (\*).

It is used to denote multi-line comment. It can apply comment to more than a single line. It is referred to as C-Style comment as it was introduced in C programming.

```
/*Comment starts
continues
continues
.
.
.
```

Comment ends\*/

**Example:**

```
/* C program to illustrate  
use of  
multi-line comment */  
  
#include <stdio.h>  
  
int main(void)  
{  
  
    /* Multi-line Welcome user comment  
written to demonstrate comments  
in C/C++ */  
  
    printf("Welcome to XYZ");  
  
    return 0;  
}
```

**Output:**

Welcome to XYZ

**Comment at End of Code Line**

You can also create a comment that displays at the end of a line of code. But generally its a better practice to put the comment before the line of code.

**Example:** This example goes same for C and C++ as the style of commenting remains same for both the language.

```
int age; // age of the person
```

## **When and Why to use Comments in programming?**

1. Comments are a way to make a code more readable by providing more description.
2. Comments can include a description of an algorithm to make code understandable.
3. Comments can be helpful for one's own self too if code is to be reused after a long gap.

# C Programming



# Algorithm

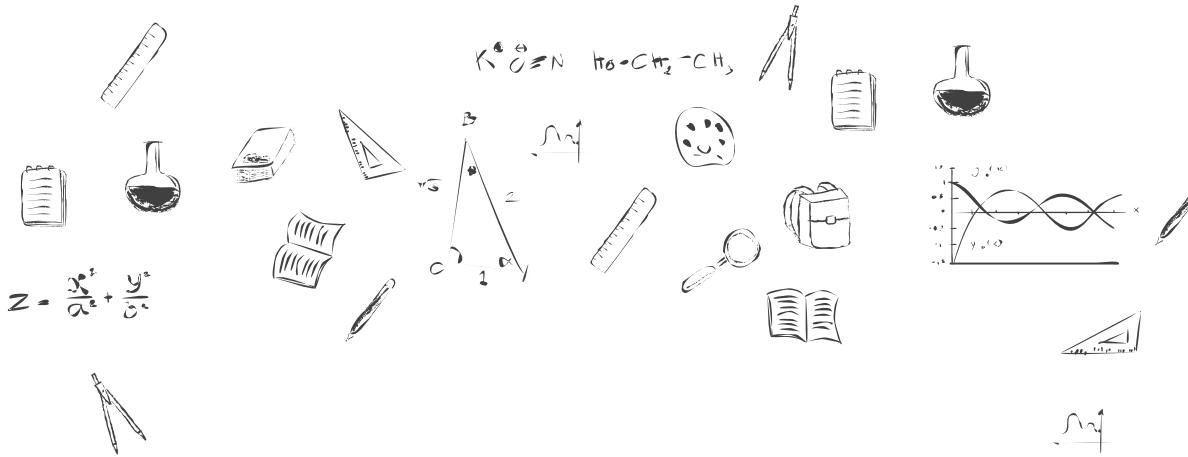
An algorithm is a set of computational steps that transform input into the output.

or

A set of sequential steps usually written in Ordinary Language to solve a given problem is called Algorithm.

or

An algorithm can be defined as “a complete, unambiguous, finite number of logical steps for solving a specific problem “



# Steps involved in algorithm development

**Step1:** Identification of input: For an algorithm, there are quantities to be supplied called input and these are fed externally. The input is to be identified first for any specified problem.

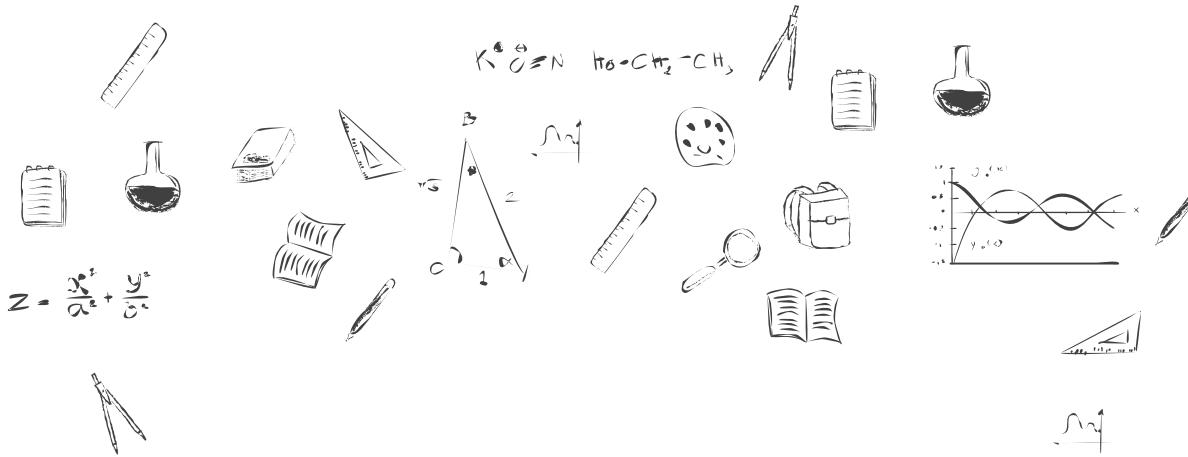
**Step2:** Identification of output: From an algorithm, at least one quantity is produced, called for any specified problem.

**Step3:** Identification the processing operations : All the calculations to be performed in order to lead to output from the input are to be identified in an orderly manner.

**Step4:** Processing Definiteness : The instructions composing the algorithm must be clear and there should not be any ambiguity in them.

**Step5:** Processing Finiteness : If we go through the algorithm, then for all cases, the algorithm should terminate after a finite number of steps.

**Step6:** Possessing Effectiveness : The instructions in the algorithm must be sufficiently basic and in practice they can be carried out easily.

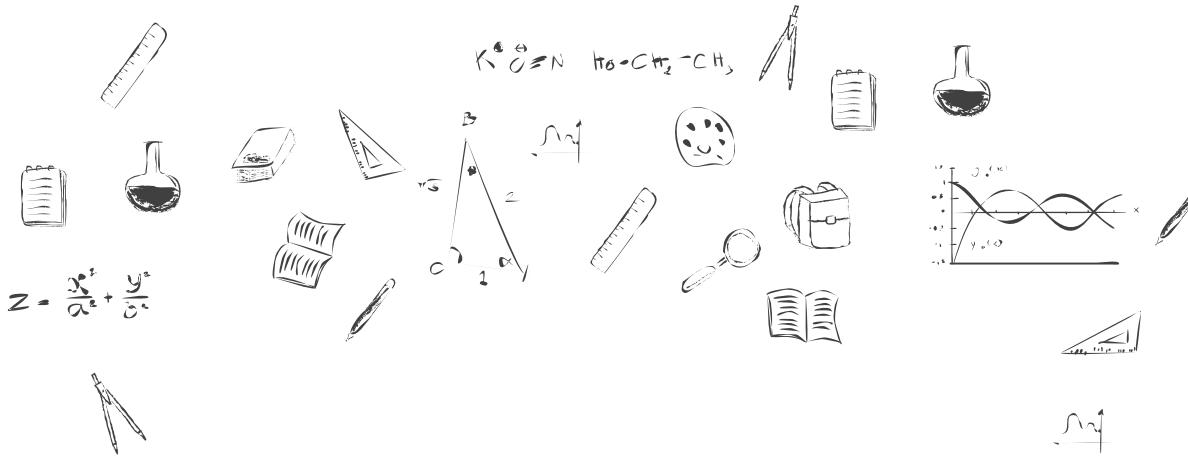


# Properties of algorithm

# An algorithm must possess the following properties

1. Finiteness: An algorithm must terminate in a finite number of steps
2. Definiteness: Each step of the algorithm must be precisely and unambiguously stated
3. Effectiveness: Each step must be effective, in the sense that it should be primitive easily convertible into program statement can be performed exactly in a finite amount of time.

4. Generality: The algorithm must be complete in itself so that it can be used to solve problems of a specific type for any input data.
5. Input/output: Each algorithm must take zero, one or more quantities as input data produce one or more output values. An algorithm can be written in English like sentences or in any standard representation sometimes, algorithm written in English like languages are called Pseudo Code.



# Examples

1. Suppose we want to find the average of three numbers, the algorithm is as follows:

Step 1: Read the numbers a, b, c

Step 2: Compute the sum of a, b and c

Step 3: Divide the sum by 3

Step 4: Store the result in variable d

Step 5: Print the value of d

Step 6: End of the program

2. Write an algorithm to calculate the simple interest using the formula. Simple interest =  $P \times N \times R / 100$ . Where P is principle Amount, N is the number of years and R is the rate of interest.

Step 1: Read the three input quantities' P, N and R.

Step 2 : Calculate simple interest as

Simple interest =  $P \times N \times R / 100$

Step 3: Print simple interest.

Step 4: Stop.

3. Write an algorithm to find the area of the triangle. Let b, c be the sides of the triangle ABC and A the included angle between the given sides.

Step 1: Input the given elements of the triangle namely sides b, c and angle between the sides A.

Step 2: Area =  $(1/2) * b * C * \sin A$

Step 3: Output the Area

Step 4: Stop.

4. Write an algorithm to find the largest of three numbers X, Y,Z.

Step 1: Read the numbers X,Y,Z.

Step 2: if ( $X > Y$ )

Big = X

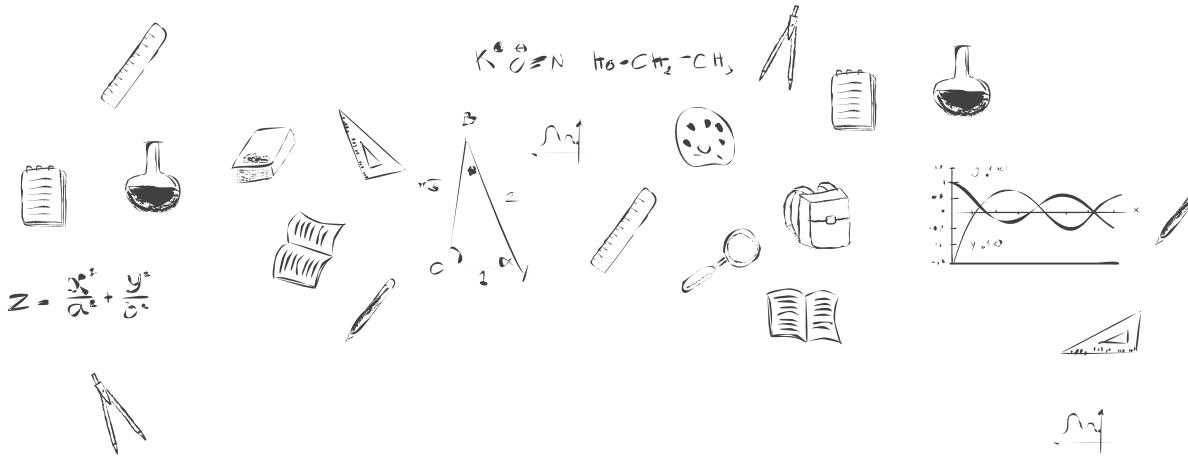
else BIG = Y

Step 3 : if ( $BIG < Z$ )

Step 4: Big = Z

Step 5: Print the largest number i.e. Big

Step 6: Stop.



# Algorithm vs Flowchart

# Algorithm

1. A method of representing the step-by-step logical procedure for solving a problem.
2. It contains step-by-step English descriptions, each step representing a particular operation leading to solution of problem.
3. These are particularly useful for small problems
4. For complex programs, algorithms prove to be Inadequate

# Flowchart

1. Flowchart is diagrammatic representation of an algorithm. It is constructed using different types of boxes and symbols.
2. The flowchart employs a series of blocks and arrows, each of which represents a particular step in an algorithm.
3. These are useful for detailed representations of complicated programs.
4. For complex programs, Flowcharts prove to be adequate.

# C Programming

## K.P.Singh



# Pseudo Code

The Pseudo code is neither an algorithm nor a program.

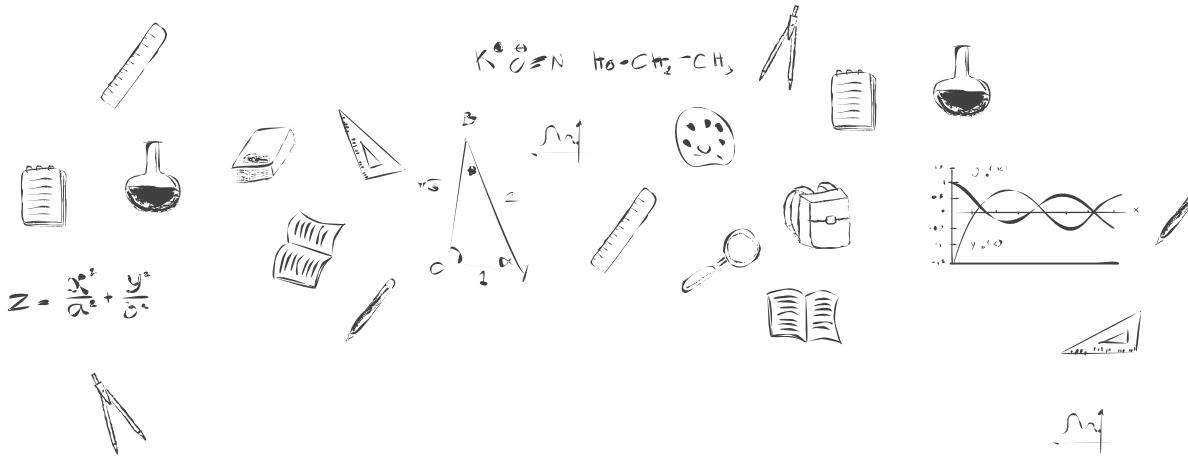
It is an abstract form of a program.

It consists of English like statements which perform the specific operations.

It is defined for an algorithm.

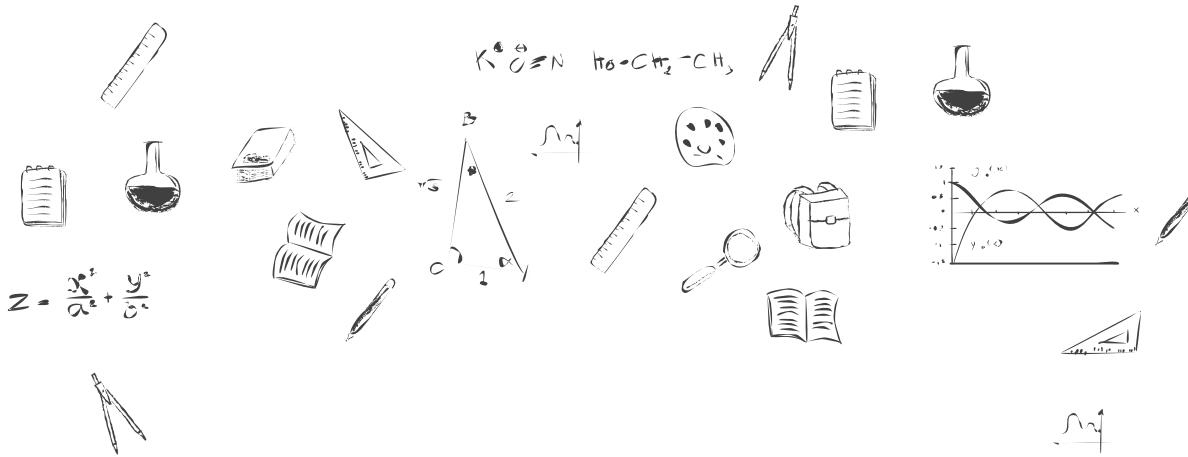
It does not use any graphical representation.

In pseudo code, the program is represented in terms of words and phrases, but the syntax of program is not strictly followed.



# Advantages

1. Easy to read
2. Easy to understand
3. Easy to modify



# Examples

Example: Write a pseudo code to perform the basic arithmetic operations.

Read n1, n2

Sum = n1 + n2

Diff = n1 – n2

Mult = n1 \* n2

Quot = n1/n2

Print sum, diff, mult, quot

End.

# Operator

# Operators in C

Unary operator



Binary operator



Ternary operator



Operator	Type
<code>++</code> , <code>--</code>	Unary operator
<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code>	Arithmetic operator
<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> , <code>==</code> , <code>!=</code>	Relational operator
<code>&amp;&amp;</code> , <code>  </code> , <code>!</code>	Logical operator
<code>&amp;</code> , <code> </code> , <code>&lt;&lt;</code> , <code>&gt;&gt;</code> , <code>~</code> , <code>^</code>	Bitwise operator
<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>	Assignment operator
<code>?:</code>	Ternary or conditional operator

# Operator

- 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.
  - Unary operator
  - Arithmetic Operators
  - Relational Operators
  - Bitwise Operator
  - Logical Operators
  - Ternary or Conditional Operators
  - Assignment Operator
  - Comma Operator

Category	Operator	Associativity
Unary operator	+ - ! ~ ++ -- (type)* & sizeof()	Right to left
Arithmetic Operators	* / %	Left to right
	+ -	Left to right
Bitwise Shift Operators	>> <<	Left to right
Relational Operators	< <= > >=	Left to right
	== !=	Left to right
Bitwise Operator	&(Bitwise AND)	Left to right
	^(Bitwise XOR)	Left to right
Logical Operators	&&(Logical AND)	Left to right
	(Logical OR)	Left to right
Ternary or Conditional Operators	?:	Right to left
Assignment Operator	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma Operator	,	Left to right

# Points to note:

- Bitwise not( $\sim$ ) is a Unary operator
- Logical not( $!$ ) is a Unary operator

# **Unary operator**

- **Unary operator** are operators that act upon a single operand to produce a new value.
- **Types of unary operators:**
  - unary minus(-)
  - increment(++)
  - decrement(- -)
  - NOT(!)
  - Address of operator(&)
  - sizeof()

# unary minus

- The minus operator changes the sign of its argument. A positive number becomes negative, and a negative number becomes positive.

```
int a = 10;
```

```
int b = -a; // b = -10
```

- unary minus is different from subtraction operator, as subtraction requires two operands.

# increment

- It is used to increment the value of the variable by 1. The increment can be done in two ways:
- **prefix increment**

In this method, the operator precedes the operand (e.g., `++a`). The value of operand will be altered *before* it is used.

```
int a = 1;  
int b = ++a; // b = 2
```

- **postfix increment**

In this method, the operator follows the operand (e.g., `a++`). The value operand will be altered *after* it is used.

```
int a = 1;  
int b = a++; // b = 1  
int c = a; // c = 2
```

# decrement

- It is used to decrement the value of the variable by 1. The decrement can be done in two ways:
- **prefix decrement**  
In this method, the operator precedes the operand (e.g., `--a`). The value of operand will be altered *before* it is used.

```
int a = 1;  
int b = --a; // b = 0
```

- **postfix decrement**  
In this method, the operator follows the operand (e.g., `a--`). The value of operand will be altered *after* it is used.

```
int a = 1;  
int b = a--; // b = 1  
int c = a; // c = 0
```

# **NOT(!)**

- It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. If  $x$  is true, then  $\neg x$  is false. If  $x$  is false, then  $\neg x$  is true.

# Addressof operator(&)

- It gives an address of a variable. It is used to return the memory address of a variable. These addresses returned by the address-of operator are known as pointers because they “point” to the variable in memory.& gives an address on variable n

```
int a;
```

```
int *ptr;
```

```
ptr = &a; // address of a is copied to the location ptr.
```

# sizeof()

- This operator returns the size of its operand, in bytes. The *sizeof* operator always precedes its operand. The operand is an expression, or it may be a cast.

```
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

# Points to Remember

- Pre increment have highest priority.
- Post increment have lowest priority than all the operator even from Assignment operator.

# C Programming

## Increment operators

# Increment Operators:

Increment Operators are used to increase the value of the variable by one.

Increment operators are used on a single operand or variable, That's why, it is called unary operator.

\* The priority of unary operators are higher than the other operators.

# Syntax:

`++` // increment operator

## Examples:

`a++;`

`++a;`

`* x= 4++;` // gives error, because 4 is constant  
`*y= ++5;` // gives error, because 5 is constant

# Type of Increment Operators

1. Pre Increment
2. Post Increment

# Pre Increment Operator (++variable):

Syntax:

```
++ variable;
```

# Pre Increment Operator:

b = ++a;

a=a+1;

b=a;

```
void main()
{
    int a,b;
    a=3;
    clrscr();
    b=++a;
    printf("%d%d",a,b);
    getch();
}
```

# **Post Increment Operator (variable++):**

Syntax:

**variable++;**

# **Post Increment Operator:**

b = a++;

b=a;

a=a+1;

```
void main()
{
    int a,b;
    a=3;
    clrscr();
    b=a++;
    printf("%d%d",a,b);
    getch();
}
```

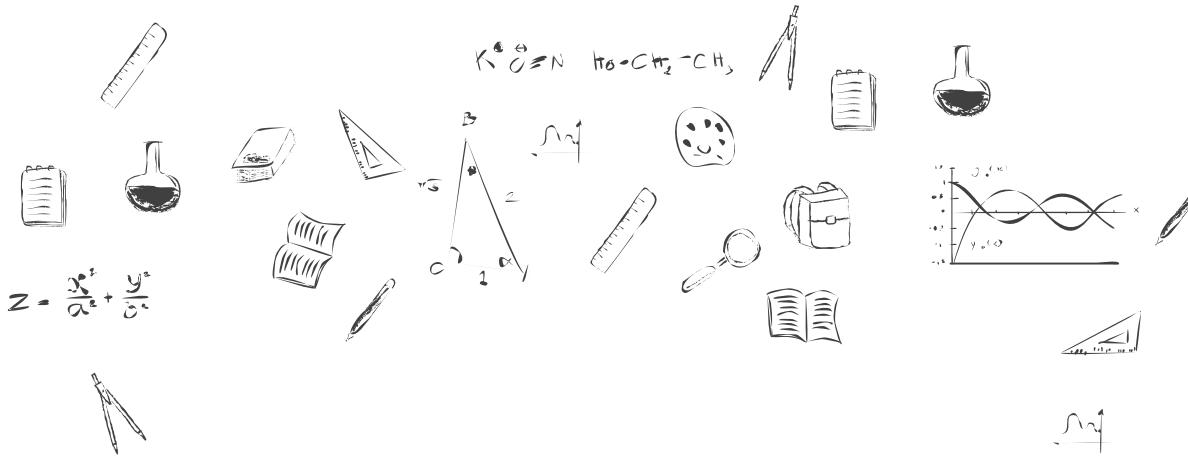
# C Programming



## Arithmetic Operators

Category	Operator	Associativity
Unary operator	+ - ! ~ ++ -- (type)* & sizeof()	Right to left
Arithmetic Operators	* / %	Left to right
	+ -	Left to right
Bitwise Shift Operators	>> <<	Left to right
Relational Operators	< <= > >=	Left to right
	== !=	Left to right
Bitwise Operator	&(Bitwise AND)	Left to right
	^(Bitwise XOR)	Left to right
	(Bitwise OR)	
Logical Operators	&&(Logical AND)	Left to right
	(Logical OR)	Left to right
Ternary or Conditional Operators	: ?	Right to left
Assignment Operator	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma Operator	,	Left to right

1. Addition	+
2. Subtraction	-
3. Multiplication	*
4. Division	/
5. Modulus	%



# Precedence Rule for Arithmetic Operators

# Precedence Rule for Arithmetic Operators:

1<sup>st</sup> Priority: \* , /, %

2<sup>nd</sup> Priority: +, -

# Addition (+):

operand1+operand2;

Example:

S=10+20;

# **Subtraction (-):**

operand1- operand2;

**Example:**

S=20-10;

# Multiplication (\*):

operand1 \* operand2;

Example:

M=5\*10;

# Division (/):

operand1 / operand2;

## Example:

10/2;      answer: 5

7/3      answer: 2

3/7      answer: 0

# **Operation**

**int/int**

**=**

**int**

**int/float**

**=**

**float**

**float/int**

**=**

**float**

**float/float**

**=**

**float**

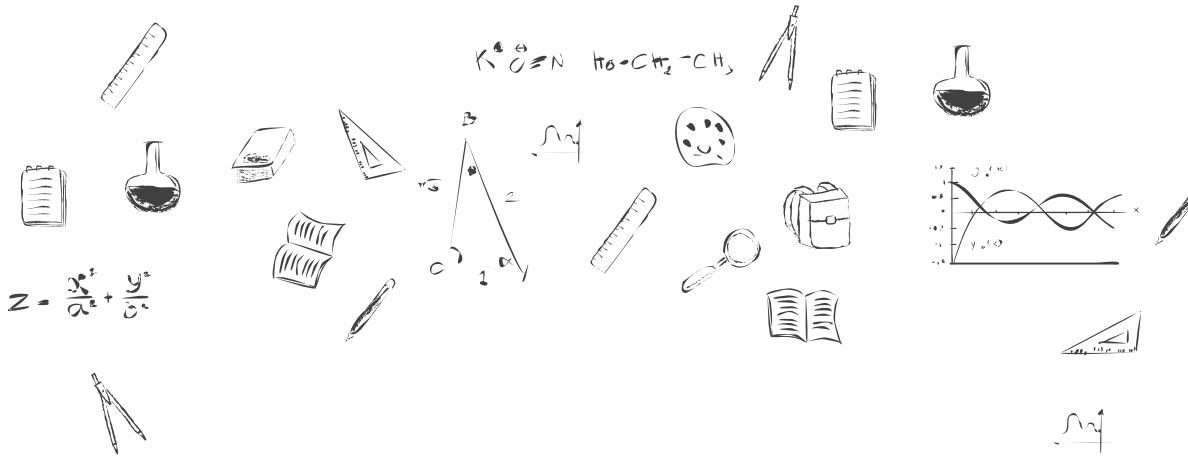
# **Result**

# Modulus Operator(%):

operand1 % operand2;

Example:

a=10%2;



$$z = \frac{x^2}{a^2} + \frac{y^2}{b^2}$$

# Arithmetic Instructions

# **Arithmetic Instructions:**

Arithmetic instructions are used to perform arithmetic operations between constants and variables.

```
n= 2*3/4+5-6%3;
```

```
n= 2*3/4+5-6%3;  
n=6/4+5-6%3;          // 2*3  
n=1+5-6%3;            // 6/4  
n=1+5-0;              // 6%3  
n=6-0;                // 1+5  
n=6;
```

# C Programming

## Relational operators

Category	Operator	Associativity
Unary operator	+ - ! ~ ++ -- (type)* & sizeof()	Right to left
Arithmetic Operators	* / %	Left to right
	+ -	Left to right
Bitwise Shift Operators	>> <<	Left to right
Relational Operators	< <= > >=	Left to right
	== !=	Left to right
Bitwise Operator	&(Bitwise AND)	Left to right
	^(Bitwise XOR)	Left to right
	(Bitwise OR)	
Logical Operators	&&(Logical AND)	Left to right
	(Logical OR)	Left to right
Ternary or Conditional Operators	: ?	Right to left
Assignment Operator	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma Operator	,	Left to right

# Relational Operators:

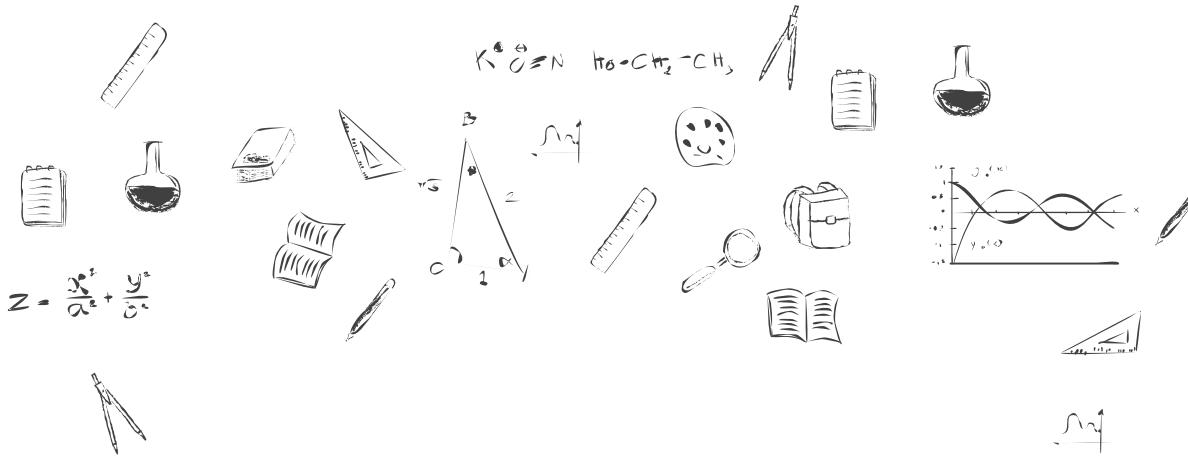
A relational operator checks the relationship between two operands.

If relation is true, then it returns 1 and if the relation is false, therefore, it returns value 0.

Relational operators are used in decision making and loops.

# Type of Relational Operators

1. Less than ( < )
2. Less than or equal to ( <= )
3. Greater than ( > )
4. Greater than or equal to ( >= )
5. Not equal to ( != )
6. Comparison Operator ( == )



# Precedence Rule for Relational Operators

# Precedence Rule for Relational Operators:

1<sup>st</sup> Priority:      <    <=    >    >=

2<sup>nd</sup> Priority:      ==    !=

## Less than ( < ):

if     **x<y** then this expression will be true.

## Less than ( < ):

```
void main()
{
    int a,b,c;
    a=4<5;          // value of a = 1
    b=14<5;         // value of b = 0
    c=5<5;          // value of c = 0
    printf("%d%d%d",a,b,c);
    getch();
}
```

## Less than ( < ):

```
void main()
{
    int a,b,c,x;
    x=4;
    a=x<5;           // value of a = 1
    b=5<x;           // value of b = 0
    c=x<4;           // value of c = 0
    printf("%d%d%d",a,b,c);
    getch();
}
```

## Less than or equal to ( $\leq$ ):

if  $x \leq y$  then this expression will be true.

## Less than or equal to ( $\leq$ ):

```
void main()
{
    int a,b,c;
    a=(4<=5);      // value of a = 1
    b=(14<=5);     // value of b = 0
    c=(5<=5);      // value of c = 1
    printf("%d%d%d",a,b,c);
    getch();
}
```

## Less than or equal to ( <= ):

```
void main()
{
    int a,b,c,x;
    x=4;
    a=(x<=5);      // value of a = 1
    b=(5<=x);      // value of b = 0
    c=(x<=4);      // value of c = 1
    printf("%d%d%d",a,b,c);
    getch();
}
```

## **Greater than ( > ):**

if  $x>y$  then this expression will be true.

## Greater than ( > ):

```
void main()
{
    int a,b,c;
    a= 4>5;          // value of a = 0
    b= 14>5;         // value of b = 1
    c= 5>5;          // value of c = 0
    printf("%d%d%d",a,b,c);
    getch();
}
```

## Greater than ( > ):

```
void main()
{
    int a,b,c,x;
    x=4;
    a=x>5;          // value of a = 0
    b=5>x;          // value of b = 1
    c=x>4;          // value of c = 0
    printf("%d%d%d",a,b,c);
    getch();
}
```

## **Greater than or equal to( $\geq$ ):**

if  $x \geq y$  then this expression will be true.

## Greater than or equal to( $\geq$ ):

```
void main()
{
    int a,b,c;
    a= (4 $\geq$ 5);           // value of a = 0
    b= (14 $\geq$ 5);         // value of b = 1
    c= (5 $\geq$ 5);           // value of c = 1
    printf("%d%d%d",a,b,c);
    getch();
}
```

## Greater than or equal to( $\geq$ ):

```
void main()
{
    int a,b,c,x;
    x=4;
    a=(x>=5);      // value of a = 0
    b=(5>=x);      // value of b = 1
    c=(x>=4);      // value of c = 1
    printf("%d%d%d",a,b,c);
    getch();
}
```

# **Comparison Operator or double equals to ( == ):**

if  $x==y$  then this expression will be true.

# Comparison Operator or double equals to ( == ):

```
void main()
{
    int a, b, c;
    a= (4==5);           // value of a = 0
    b= (14==5);          // value of b = 0
    c= (5==5);           // value of c = 1
    printf("%d%d%d",a,b,c);
    getch();
}
```

## Comparison Operator or double equals to ( == ):

```
void main()
{
    int a,b,c,x;
    x=4;
    a=(x==5);      // value of a = 0
    b=(5==x);      // value of b = 0
    c=(x==4);      // value of c = 1
    printf("%d%d%d",a,b,c);
    getch();
}
```

## **Not equal to ( != ):**

if  $x \neq y$  then this expression will be true.

## Not equal to ( != ):

```
void main()
{
    int a, b, c;
    a= (4!=5);          // value of a = 1
    b= (14!=5);         // value of b = 1
    c= (5!=5);          // value of c = 0
    printf("%d%d%d",a,b,c);
    getch();
}
```

## Not equal to ( != ):

```
void main()
{
    int a,b,c,x;
    x=4;
    a=(x!=5);      // value of a = 1
    b=(5!=x);      // value of b = 1
    c=(x!=4);      // value of c = 0
    printf("%d%d%d",a,b,c);
    getch();
}
```

**Example:**

```
void main()
{
    int x;
    x=3>4;
    printf("%d",x);
    getch();
}
```

Output: 0

**Example:**

```
void main()
{
    int x;
    x=3<=4;
    printf("%d",x);
    getch();
}
```

Output: 1

**Example:**

```
void main()
{
    int x;
    x=5>4>3;
    printf("%d",x);
    getch();
}
```

Output:

<b>this expression</b>	<b>is true if</b>
<code>x == y</code>	x is equal to y
<code>x != y</code>	x is not equal to y
<code>x &lt; y</code>	x is less than y
<code>x &gt; y</code>	x is greater than y
<code>x &lt;= y</code>	x is less than or equal to y
<code>x &gt;= y</code>	x is greater than or equal to y

# Bitwise Operator in C

Category	Operator	Associativity
Unary operator	+ - ! ~ ++ -- (type)* & sizeof()	Right to left
Arithmetic Operators	* / %	Left to right
	+ -	Left to right
Bitwise Shift Operators	>> <<	Left to right
Relational Operators	< <= > >=	Left to right
	== !=	Left to right
Bitwise Operator	&(Bitwise AND)	Left to right
	^(Bitwise XOR)	Left to right
Logical Operators	&&(Logical AND)	Left to right
	(Logical OR)	Left to right
Ternary or Conditional Operators	?:	Right to left
Assignment Operator	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma Operator	,	Left to right

# Bitwise Operator

- 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

# 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,**

- We have two variables a and b.

a =6;

b=4;

The binary representation of the above two variables are given below:

a = 0110

b = 0100

When we apply the bitwise AND operation in the above two variables, i.e., a&b, the output would be:

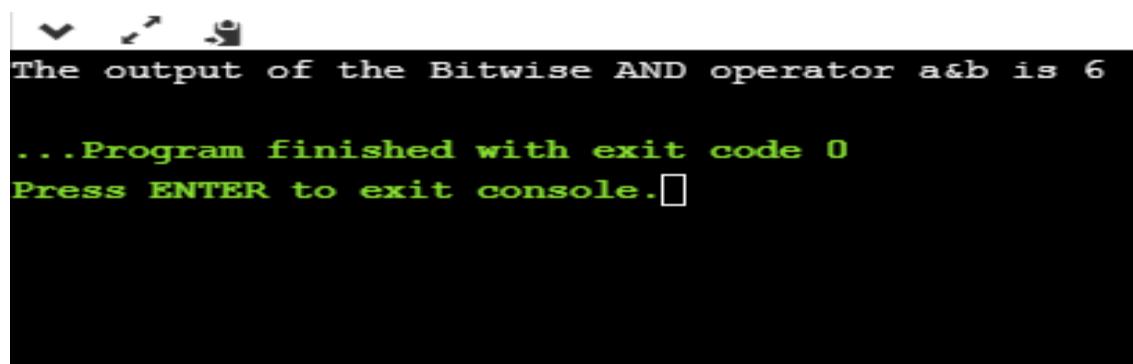
Result = 0100

# Program using Bitwise AND operator

```
#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;
}
```

a AND b = 0110 && 1110 = 0110

Output:



```
The output of the Bitwise AND operator a&b is 6
...Program finished with exit code 0
Press ENTER to exit console.
```

# 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,**

- We consider two variables,

a = 23;

b = 10;

The binary representation of the above two variables would be:

a = 0001 0111

b = 0000 1010

When we apply the bitwise OR operator in the above two variables, i.e., a|b , then the output would be:

Result = 0001 1111

# Program using Bitwise OR operator

```
#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;
}
```

a = 0001 0111 and b = 0000 1010

OUTPUT:

```
          ^   ~   $ 
The output of the Bitwise OR operator a|b is 31

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

# 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,**

- We consider two variables a and b,

a = 12;

b = 10;

The binary representation of the above two variables would be:

a = 0000 1100

b = 0000 1010

When we apply the bitwise exclusive OR operator in the above two variables (a^b), then the result would be:

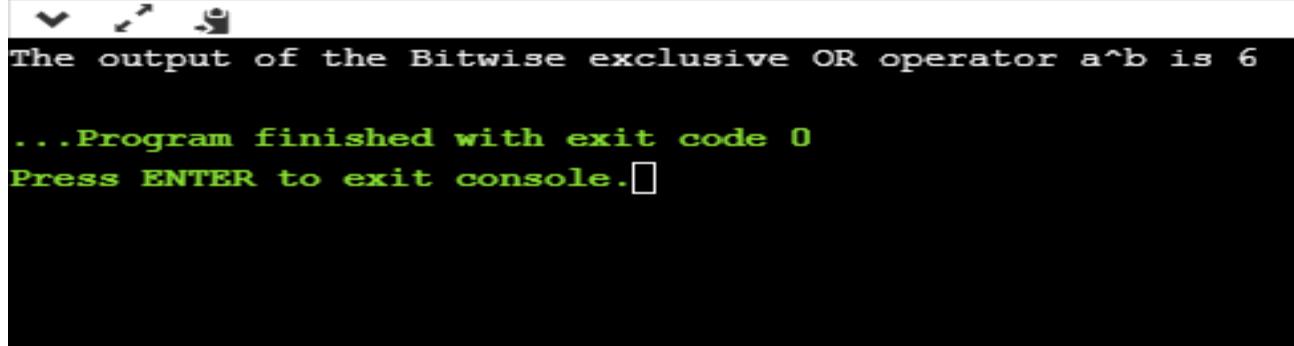
Result = 0000 0110

# Program using Bitwise exclusive OR operator

```
#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;
}
```

a = 0000 1100 and b = 0000 1010

Output:



```
The output of the Bitwise exclusive OR operator a^b is 6
...Program finished with exit code 0
Press ENTER to exit console.□
```

# Bitwise complement operator

- The bitwise complement operator is also known as one's complement operator. It is represented by the symbol tilde ( $\sim$ ). 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,

- If we have a variable named 'a',

$a = 8;$

The binary representation of the above variable is given below:

$a = 1000$

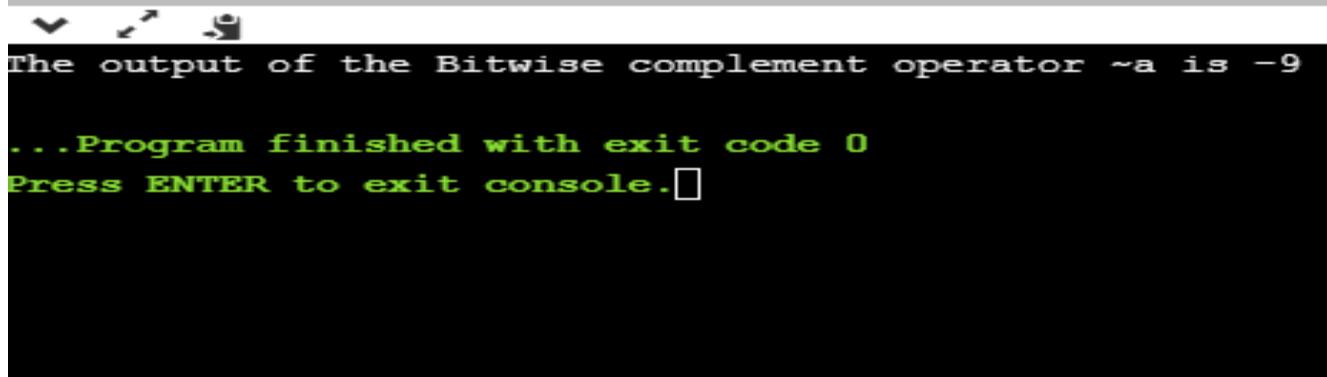
When we apply the bitwise complement operator to the operand, then the output would be:

Result = 0111

# Program using Bitwise complement operator

```
#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.

A screenshot of a terminal window showing the execution of a C program. The program declares an integer variable 'a' with the value 8, then prints its bitwise complement using the printf function. The output is '-9'. After the output, it shows the standard message "...Program finished with exit code 0" and "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:**

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,

- Suppose we have a statement:

```
int a = 5;
```

The binary representation of 'a' is given below:

```
a = 0101
```

If we want to left-shift the above representation by 2, then the statement would be:

```
a << 2;
```

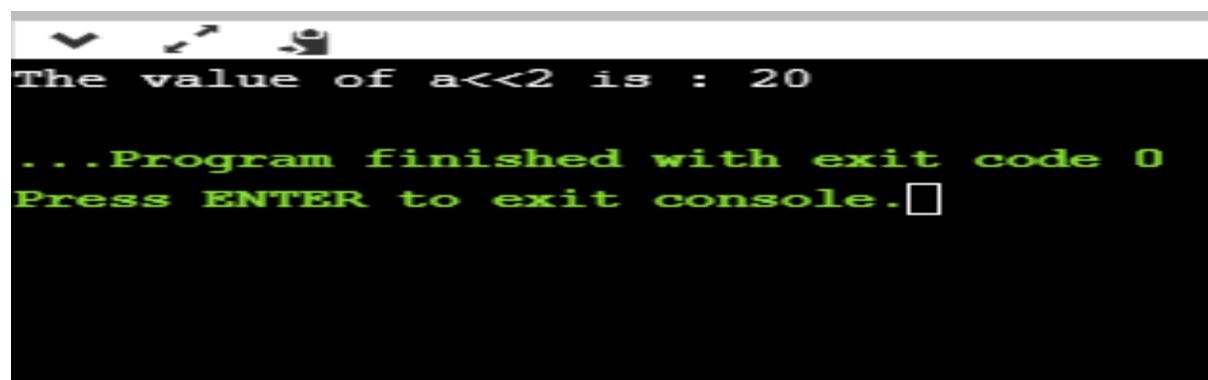
```
0101<<2 = 010100
```

# Program using Left-shift operator

```
#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:**

Operand >> 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,

Suppose we have a statement,

```
int a = 7;
```

The binary representation of the above variable would be:

```
a = 0111
```

If we want to right

shift the above representation by 2, then the statement would be:

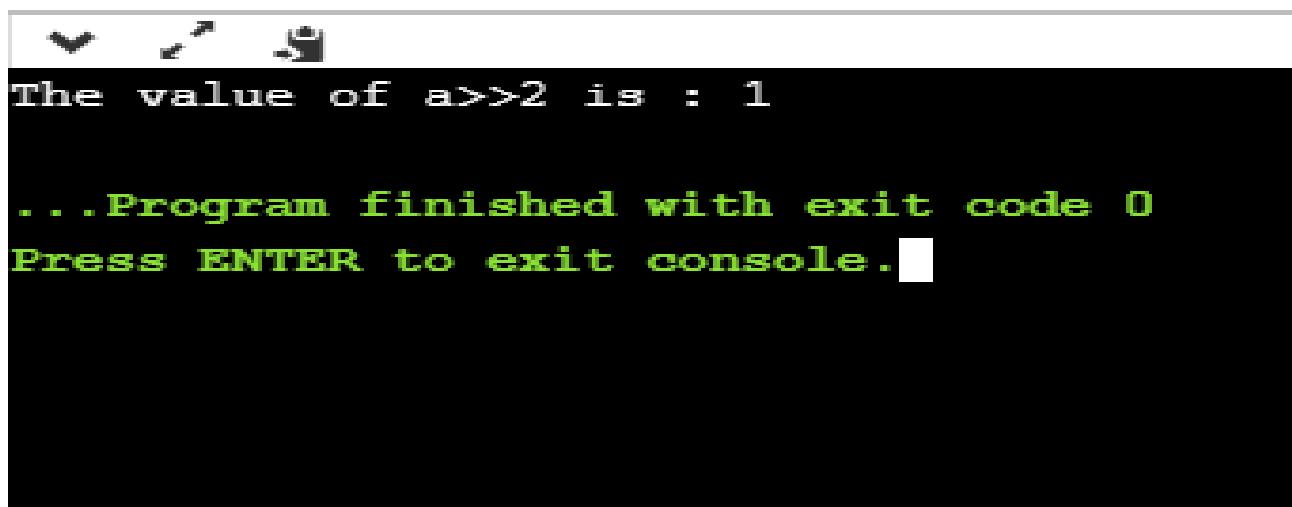
```
a>>2;
```

```
0000 0111 >> 2 = 0000 0001
```

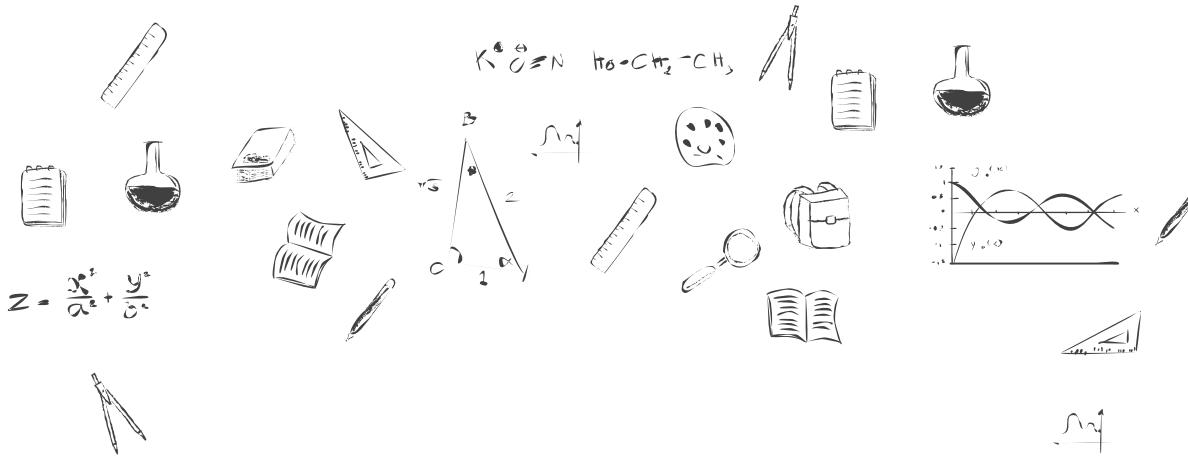
# Program using Right-shift operator

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

Output:



The screenshot shows a terminal window with a black background and white text. At the top, there are three small icons: a red circle with a white checkmark, a green circle with a white checkmark, and a blue circle with a white checkmark. Below these icons, the text "The value of a>>2 is : 1" is displayed. At the bottom of the window, there is a message in green text: "... Program finished with exit code 0". Below that, there is another message in green text: "Press ENTER to exit console." followed by a small white square icon.



# Logical Operator

Category	Operator	Associativity
Unary operator	+ - ! ~ ++ -- (type)* & sizeof()	Right to left
Arithmetic Operators	* / %	Left to right
	+ -	Left to right
Bitwise Shift Operators	>> <<	Left to right
Relational Operators	< <= > >=	Left to right
	== !=	Left to right
Bitwise Operator	&(Bitwise AND)	Left to right
	^(Bitwise XOR)	Left to right
	(Bitwise OR)	
Logical Operators	&&(Logical AND)	Left to right
	(Logical OR)	Left to right
Ternary or Conditional Operators	: ?	Right to left
Assignment Operator	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma Operator	,	Left to right

# C Programming

## Logical Operator

The Logical operators are used to perform logical operations on the given expressions.

Logical Operators are used with binary variables.

They are mainly used in conditional statements and loops for evaluating a condition.

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.

Logical operators are used to combine and evaluate two or more conditions.

# Types of Logical Operators:

1. Logical AND (`&&`)
2. Logical OR (`||`)
3. Logical NOT (`!`)

# Logical AND (&&):

A	B	A&&B
0	0	0
1	0	0
0	1	0
1	1	1

# Logical OR (||):

A	B	A  B
0	0	0
1	0	1
0	1	1
1	1	1

# Logical NOT (!):

A	!A
0	1
1	0

# C Programming

## Logical Instructions

The Logical operators are used to perform logical operations on the given expressions.

These expressions are known as logical instructions in C Programming.

```
void main()
{
    int a;
    clrscr();
    a=2||3&&4||0&&5||!7&&6;
    printf("%d",a);
    getch();
}
```

# Precedence Rule for Logical Operators:

1. Logical NOT ( ! )
2. Logical AND ( && )
3. Logical OR ( || )

```
void main()
{
    int a;
    clrscr();
    a=2||3&&4||0&&5||!7&&6;
    printf("%d",a);
    getch();
}
```

```
a=2 || 3&&4 || 0&&5 || !7&&6;  
a=2 || 3&&4 || 0&&5 || 0 &&6;    // !7 = 0  
a=2 || 1 || 0&&5 || 0 &&6;        // 3&&4 = 1  
a=2 || 1 || 0 || 0 &&6;           // 0&&5 = 0  
a=2 || 1 || 0 || 0                // 0&&6 = 0  
a=1 || 0 || 0                     // 2||1 = 1  
a=1 || 0                         // 1 || 0 = 1  
a=1                           // 1 || 0 = 1
```

**Example:**

```
void main()
{
int y,x=5;
clrscr();
y=!x>4
printf("%d",y);
getch();
}
```

OUTPUT: 0

**Example:**

```
void main()
{
int y,x=5;
clrscr();
y=x>4 && x<10
printf("%d",y);
getch();
}
```

OUTPUT: 1

**Example:**

```
void main()
{
int y,x=5;
clrscr();
y=x<4|| x<10
printf("%d",y);
getch();
}
```

OUTPUT: 1

# Conditional Operator in C

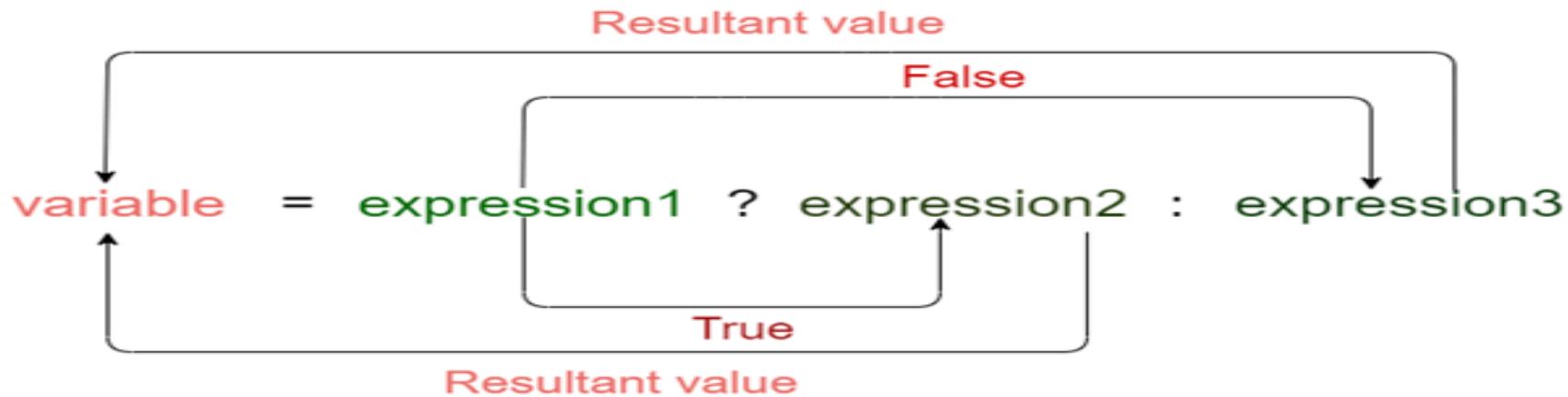
Category	Operator	Associativity
Unary operator	+ - ! ~ ++ -- (type)* & sizeof()	Right to left
Arithmetic Operators	* / %	Left to right
	+ -	Left to right
Bitwise Shift Operators	>> <<	Left to right
Relational Operators	< <= > >=	Left to right
	== !=	Left to right
Bitwise Operator	&(Bitwise AND)	Left to right
	^(Bitwise XOR)	Left to right
Logical Operators	&&(Logical AND)	Left to right
	(Logical OR)	Left to right
Ternary or Conditional Operators	?:	Right to left
Assignment Operator	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma Operator	,	Left to right

# Conditional Operator in C

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

# Syntax of a conditional operator

Expression1? expression2: expression3;



## 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.

# Example 1 of Conditional Operator

```
#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;
}
```

# Explanation of example 1

- In the given code, we are taking input as the 'age' of the user. After taking input, we have applied the condition by using a conditional operator. In this condition, we are checking the age of the user. If the age of the user is greater than or equal to 18, then the statement1 will execute, i.e., (printf("eligible for voting")) otherwise, statement2 will execute, i.e., (printf("not eligible for voting")).

# Output of example 1

```
v ~ ^P & Enter your age 24  
eligible for voting  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

# Example 2 of Conditional Operator

```
#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;
}
```

# Explanation of example 2

- In the above code, we have declared two variables, i.e., 'a' and 'b', and assign 5 value to the 'a' variable. After the declaration, we are assigning value to the 'b' variable by using the conditional operator. If the value of 'a' is equal to 5 then 'b' is assigned with a 3 value otherwise 2.

# Output of example 2

---

```
  ▼ ▷ ⌂
The value of 'b' variable is : 3
...
...Program finished with exit code 0
Press ENTER to exit console.□
```

## **Assignment Operator**

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of the variable on the left otherwise the compiler will raise an error.

Different types of assignment operators are shown below:

- “=”: This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.

For example:

```
a = 10;  
b = 20;  
ch = 'y';
```

- “+=”: This operator is combination of ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

Example:

(a += b) can be written as (a = a + b)

If initially value stored in a is 5. Then (a += 6) = 11.

- “-=”: This operator is combination of ‘-’ and ‘=’ operators. This operator first subtracts the current value of the variable on left from the value on the right and then assigns the result to the variable on the left.

Example:

- (a -= b) can be written as (a = a - b)

If initially value stored in a is 8. Then (a -= 6) = 2.

- “\*=”: This operator is combination of ‘\*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

Example:

(a \*= b) can be written as (a = a \* b)

If initially value stored in a is 5. Then (a \*= 6) = 30.

- “/=”: This operator is combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.

Example:

(a /= b) can be written as (a = a / b)

If initially value stored in a is 6. Then (a /= 2) = 3.

# Comma Operator

Category	Operator	Associativity
Unary operator	+ - ! ~ ++ -- (type)* & sizeof()	Right to left
Arithmetic Operators	* / %	Left to right
	+ -	Left to right
Bitwise Shift Operators	>> <<	Left to right
Relational Operators	< <= > >=	Left to right
	== !=	Left to right
Bitwise Operator	&(Bitwise AND)	Left to right
	^(Bitwise XOR)	Left to right
Logical Operators	&&(Logical AND)	Left to right
	(Logical OR)	Left to right
Ternary or Conditional Operators	?:	Right to left
Assignment Operator	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma Operator	,	Left to right

# Comma as Separator

- In C programming language, **comma (,) works as a separator and an operator too and its behaviour is little different according to the place where it is used.**

## 1) Comma (,) as separator

While declaration multiple variables and providing multiple arguments in a function, comma works as a separator.

### Example:

- `int a,b,c;`
- In this statement, **comma is a separator and tells to the compiler that these (a, b, and c) are three different variables.**

# Comma as operator

## 2) Comma (,) as an operator

Sometimes we assign multiple values to a variable using comma, in that case comma is known as operator.

**Example:**

- `a = 10,20,30;`
- `b = (10,20,30);`

In the first statement, value of **a** will be 10, because **assignment operator (=) has more priority than comma (,)**, thus 10 will be assigned to the variable **a**.

In the second statement, value of **b** will be 30, because **10, 20, 30 are enclosed in braces, and braces has more priority than assignment (=) operator**. When multiple values are given with comma operator within the braces, then right most value is considered as result of the expression. Thus, 30 will be assigned to the variable **b**.

## Example:

```
#include <stdio.h>
int main()
{
int a,b;
a = 10,20,30;
b = (10,20,30);
//printing the values
printf("a= %d, b= %d\n",a,b);
return 0;
}
```

Output

a= 10, b= 30

# NOTE

- Try not to confuse between comma as a separator and comma as an operator. Sample example:  
`int a = 4, 3;`
- This will generate an error as comma in this case acts as a separator as declaration takes place. So the error less code will be as follows:  
`int a;  
a = 4,3;`  
Now the value stored in a will be 4.
- Also, the following is valid,  
`int a =(4, 3);`  
here , 3 is stored in a.

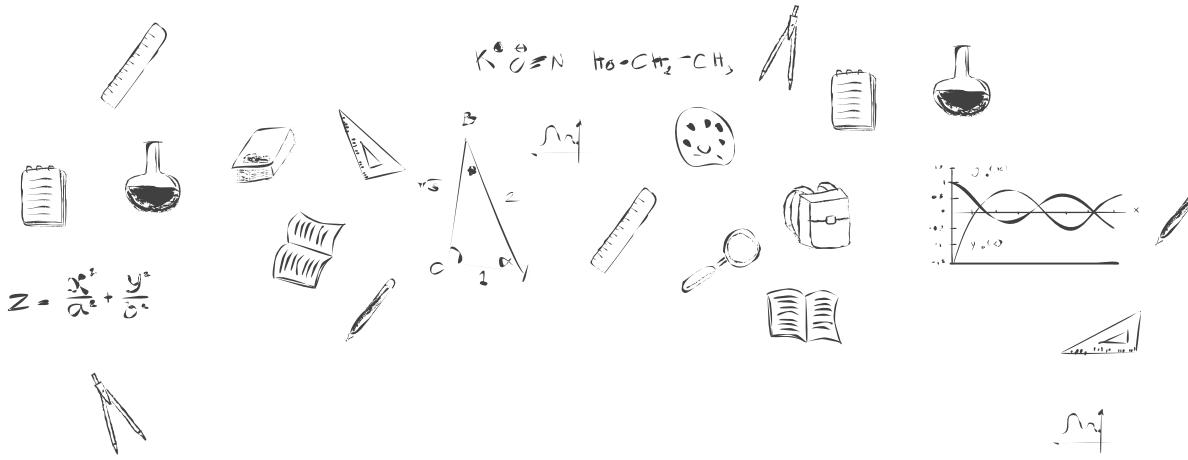
Ct-2

---

# Control Flow Statement

- This statement is a part of program that can be executed.
- A control flow statement enable us to specify the order in which the various instruction in a program are to be executed by the computer.

- Following are the various types of control statement in C:
  - i) Selection Statement(Conditional Statement):if and switch
  - ii) Iteration Statement(Loop Statement):for,while,do-while
  - iii) Jump Statement:Break,Continue,goto,return



# IF Statement

# C Programming

K.P.Singh

## If Statement

## **Syntax:**

The if statement is used to express conditional expressions. If the given condition is true then it will execute the statements otherwise skip the statement

```
If (condition)  
statement-1;
```

OR

```
If (condition)
{
    statement-1;
    statement-2;
    statement-3;
    .....
    .....
STATEMENT-N
}
```

The expression is evaluated and if the expression is true the statements will be executed. If the expression is false the statements are skipped and execution continues with the next statements.

```
void main()
{
    int a=5;
    if(a<10)
    {
        printf("Hello");
    }
    getch();
}
```

Output: Hello

```
void main()
{
    int a=5;
    if(a<10)
    {
        printf("\n Hello");
        printf("\n XYZ");
        printf("\n www.gmail.com");
    }
    getch();
}
```

Output:   Hello  
         XYZ  
         www.gmail.com

```
void main()
{
    int a=50;
    if(a<10)
    {
        printf("\n Hello");
        printf("\n Kunwar");
        printf("\n www.gmail.com");
    }
    getch();
}
```

Output:

---

C Programming

# **Example : C Program to Find the Largest Number Among Three Numbers(Using if Statement)**

```
#include <stdio.h>
int main()
{
double n1, n2, n3;
printf("Enter three different numbers: ");
scanf("%lf %lf %lf", &n1, &n2, &n3);

// if n1 is greater than both n2 and n3, n1 is the largest
if (n1 >= n2 && n1 >= n3)
    printf("%.2f is the largest number.", n1);

// if n2 is greater than both n1 and n3, n2 is the largest
if (n2 >= n1 && n2 >= n3) printf("%.2f is the largest number.", n2);

// if n3 is greater than both n1 and n2, n3 is the largest
if (n3 >= n1 && n3 >= n2)
    printf("%.2f is the largest number.", n3);

return 0;
}
```

# Output

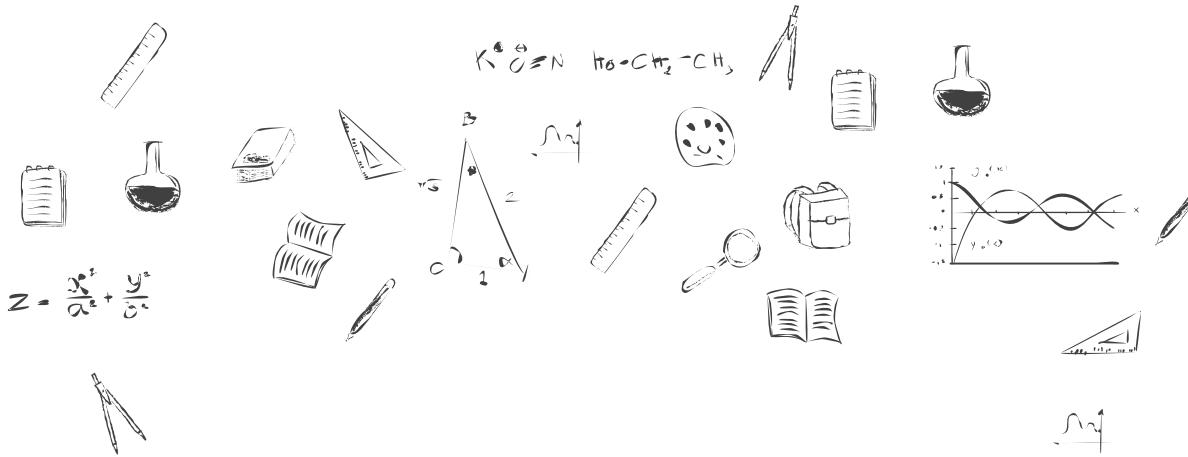
Enter three numbers: -4.5

3.9

5.6

5.60 is the largest number.

117



# IF –else Statement

# C Programming

K.P.Singh

## If-else Statement

# Syntax:

```
if(condition)
{
    statement(s);//code to be executed if condition is true
}
else
{
    statement(s);//code to be executed if condition is false
}
```

```
void main()
{
    int a=5;
    if(a<10)
    {
        printf("Hello");
    }
    else
    {
        printf("XYZ");
    }
    getch();
}
```

Output: Hello

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

```
#include<stdio.h>
int main()
{
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number%2==0)
{
printf("%d is even number",number);
}
else
{
printf("%d is odd number",number);
}
return 0;
}
```

Output:

enter a number:4

4 is even number

enter a number:5

5 is odd number

## **Program to check whether a person is eligible to vote or not.**

```
#include <stdio.h>
#include <conio.h>
void 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");
    }
    getch();
}
```

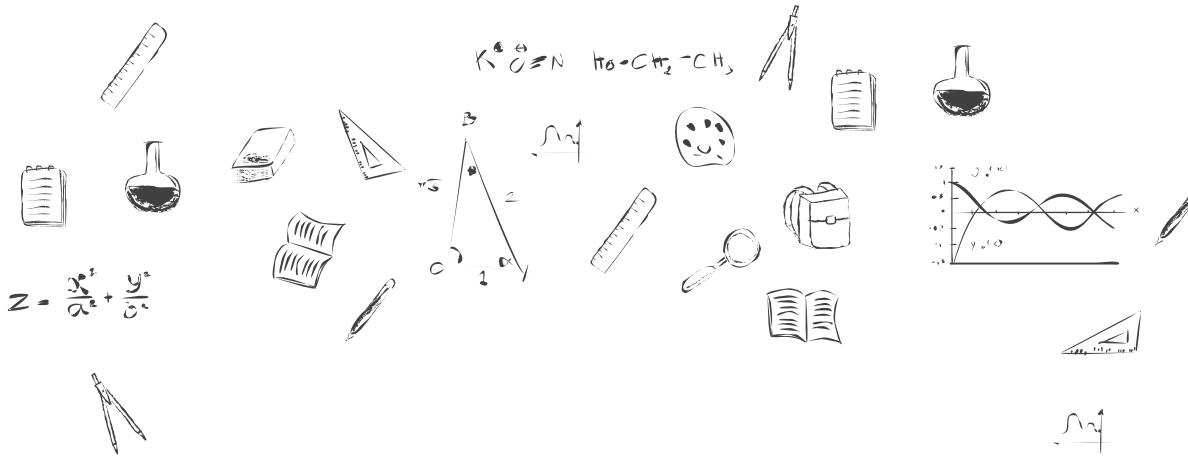
# Output

Enter your age?18

You are eligible to vote...

Enter your age?13

Sorry ... you can't vote



# Nested If-else Statement

# C Programming

## Nested If-else Statement

When multiple conditions or decisions are required, then, nested if-else is used.

Nesting means when an if else statement is present inside the body of another “if” or “else” then this is called nested if else.

# Syntax:

```
if(condition 1)
{
    if(condition 2)
        statement 1; //statement 1 to be executed if condition 1 and condition 2 is true
    else
        statement 2; //statement 2 to be executed if condition 1 is true and condition 2 is false

}
else
{
    if(condition 3)
        statement 3;//statement 3 to be executed if condition 1 is false and condition 3 is true
    else
        statement 4;// statement 4 to be executed if condition 1 and condition 3 is false
}
```

# **Example : C Program to Find the Largest Number Among Three Numbers(Using Nested if-else Statement)**

```
#include <stdio.h>
int main()
{
    double n1, n2, n3;
    printf("Enter three numbers: ");
    scanf("%lf %lf %lf", &n1, &n2, &n3);
    if (n1 >= n2)
    {
        if (n1 >= n3)
            printf("%.2lf is the largest number.", n1);
        else
            printf("%.2lf is the largest number.", n3);
    }
    else
    {
        if (n2 >= n3)
            printf("%.2lf is the largest number.", n2);
        else
            printf("%.2lf is the largest number.", n3);
    }
    return 0;
}
```

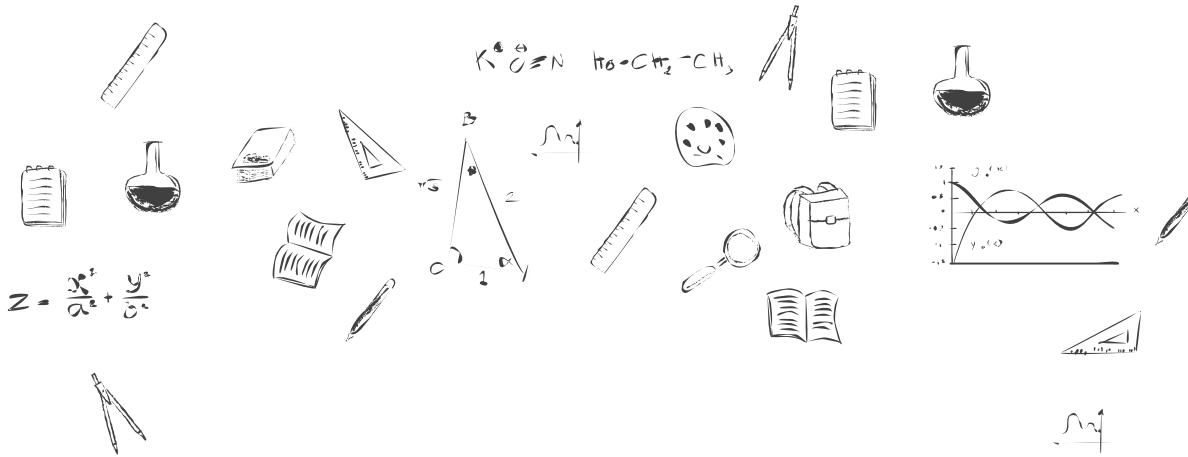
# output

Enter three numbers: -4.5

3.9

5.6

5.60 is the largest number.



# Else-If Ladder Statement

# C Programming

## Else-If Ladder

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.

## Syntax:

```
if(condition 1)
statement 1;//code to be executed if condition1 is true
else if(condition 2)
statement 2; //code to be executed if condition2 is true
else if(condition 3)
statement 3; //code to be executed if condition3 is true
else
statement 4; //code to be executed if condition4 is true
```

**Example:** The marks obtained by a student in a subject is input through keyboard.

Marks above 85: grade A

Marks between 61-85: grade B+

Marks above 41-60: grade B

Marks above 31-40: grade C

Marks less than 31: Fail

Write a C program to check the grade of student in the given subject

```
#include <stdio.h>
#include <conio.h>
void 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 ...");
}
getch();
```

# 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 ...

# **Example : C Program to Find the Largest Number Among Three Numbers(Using else-if )**

```
#include <stdio.h>
int main()
{
    double n1, n2, n3;
    printf("Enter three numbers: ");
    scanf("%lf %lf %lf", &n1, &n2, &n3);

    // if n1 is greater than both n2 and n3, n1 is the largest
    if (n1 >= n2 && n1 >= n3)
        printf("%.2lf is the largest number.", n1);

    // if n2 is greater than both n1 and n3, n2 is the largest
    else if (n2 >= n1 && n2 >= n3)
        printf("%.2lf is the largest number.", n2);

    // if both above conditions are false, n3 is the largest
    else
        printf("%.2lf is the largest number.", n3); return 0; }
```

# output

Enter three numbers: -4.5

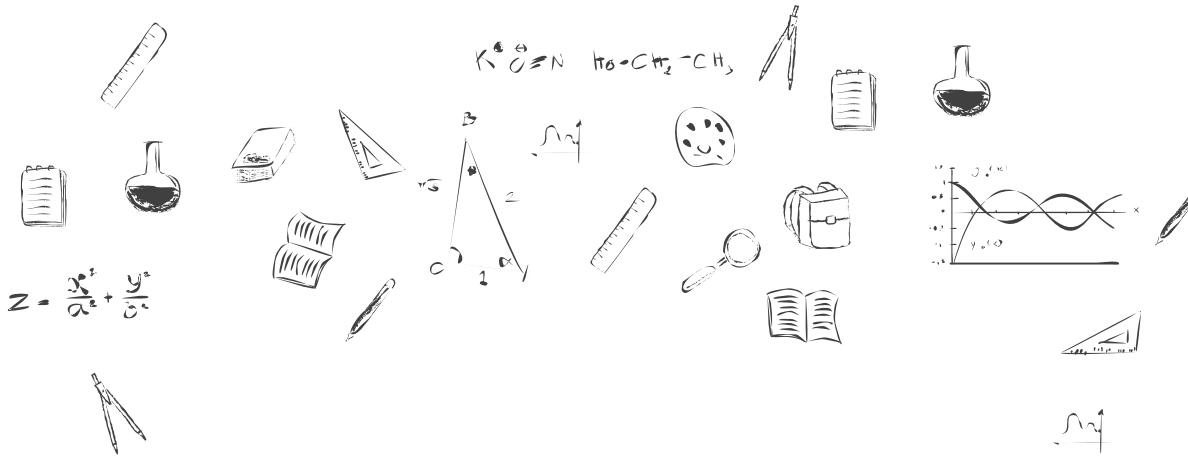
3.9

5.6

5.60 is the largest number.

# C Programming

## The Loops



# Introduction to Loops

## The Loops:

The Loops are used to execute the block of code multiple times according to the condition given in the loop.

It means it executes the same code several times so it saves code and also helps to access the elements of an array.

The loop will iterate until the last condition becomes false. it also executes the code until condition is false.

It executes a block of statements number of times until the condition becomes false

# Why do we need loops?

```
void main()
{
    clrscr();
    printf("\n Hello World");
    getch();
}
```

C programming has three types of loops:

- 1. for loop**
- 2. while loop**
- 3. do...while loop**

## **Types of Loops:**

Depending upon the conditional statement in a program, a loop is classified into two types:

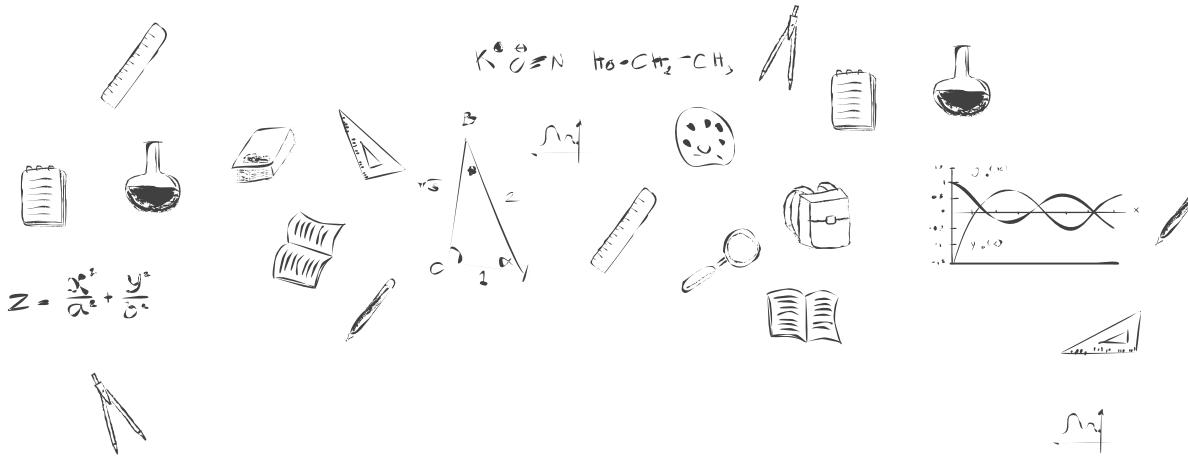
- 1. Entry controlled loop**
- 2. Exit controlled loop**

## **1. Entry controlled loop:**

In an entry controlled loop, a condition is checked before executing the body of a loop i.e. for & while loop. It is also called as a pre-checking loop.

## **2. Exit controlled loop:**

In an exit controlled loop, a condition is checked after executing the body of a loop i.e. do- while loop. It is also called as a post-checking loop.



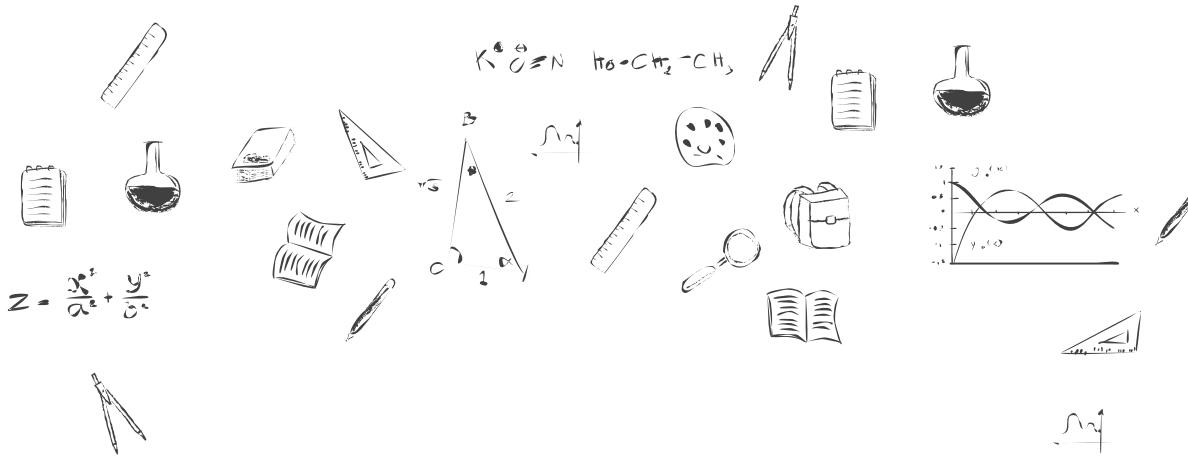
# for loop

## The **for** loop:

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

## Syntax of for loop:

```
for(initialization ; condition ; increment/decrement)
{
    statement(s);
}
```



# while loop

## The while loop:

It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop.

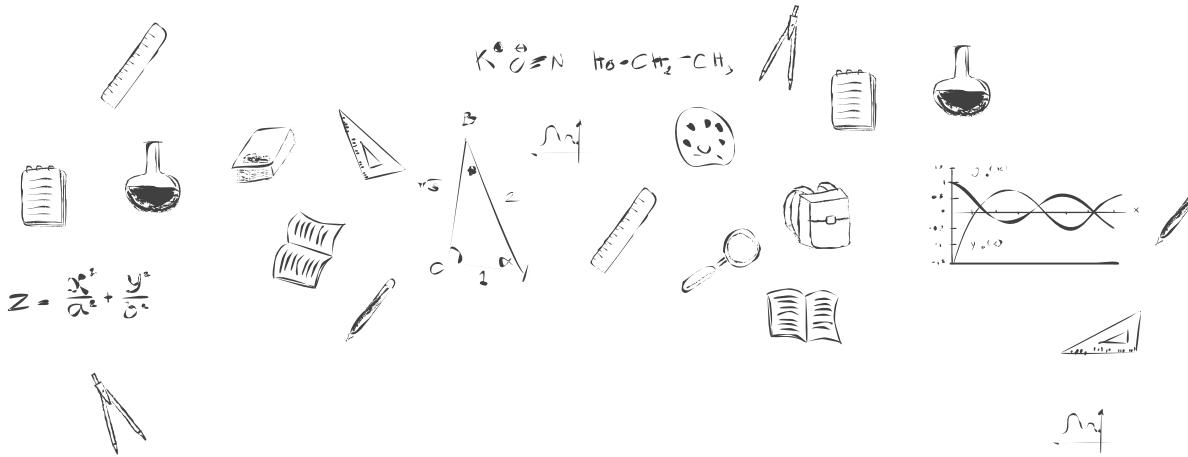
If condition is true then and only then the statements of a loop is executed.

After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false.

Once the condition becomes false, the control goes out of the loop.

## Syntax of while loop:

```
initialization;  
while(condition)  
{  
    statement(s);  
    increment/decrement;  
}
```



# do-while loop

## The do-while loop:

A do-while loop is similar to the while loop except that the condition is always checked after the body of a loop. It is also called an exit-controlled loop.

In a while loop, the body is executed if and only if the condition is true and if the condition is not true, then the body of a loop will not be executed, not even once.

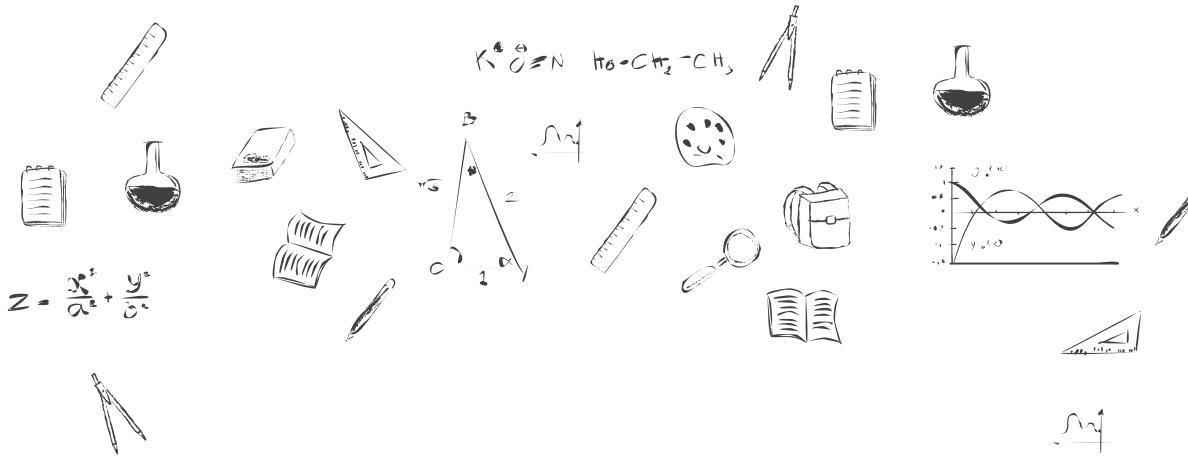
In some cases, we have to execute a body of the loop at least once even if the condition is false. This can be achieved by using a do-while loop.

In the do-while loop, the body of a loop is always executed at least once.

After the body is executed, then it checks the condition. If the condition is true, then it will again execute the body of a loop otherwise control will be out from the loop.

## Syntax of do-while loop:

```
initialization;  
do  
{  
    statements;  
    increment/decrement;  
} while (condition);
```



# While vs do-while loop

## **while loop**

This loop is also known as entry controlled loop.

The while loop checks the condition first and then execute the statements.

The condition is specified at the beginning of the loop.

The body is executed only if a certain condition is true and it terminates if condition is false.

## **do-while loop**

This loop is also known as exit controlled loop.

The do-while loop executes the statements before evaluating the condition.

The condition is specified after the body of the loop.

The body is always executed at least once even if condition is false.

# Which loop to Select?

1. If pre-test is required, use a while or for a loop.
2. If post-test is required, use a do-while loop.

## **for vs while loop:**

The for loop is used where we already know about the number of times loop needs to be executed.

The While loop is used where we do not know how many times loop needs to be executed.

# **Infinite or Indefinite Loop**

## Infinite or Indefinite Loop:

The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times.

The loop that does not stop executing and processes the statements number of times is called as an infinite loop.

An infinite loop is also called as an "Endless loop."

Following are some characteristics of an infinite loop:

1. No termination condition is specified.
2. The specified conditions never meet.

## **When to use an infinite loop:**

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

In the following situations, this type of loop can be used:

1. Menu driven programs
2. All the operating systems run in an infinite loop as it does not exit after performing some task. It comes out of an infinite loop only when the user manually shuts down the system.
3. All the servers run in an infinite loop as the server responds to all the client requests. It comes out of an indefinite loop only when the administrator shuts down the server manually.

4. All the games also run in an infinite loop. The game will accept the user requests until the user exits from the game.

All the operating systems run in an infinite loop as it does not exist after performing some task. It comes out of an infinite loop only when the user manually shuts down the system.

All the servers run in an infinite loop as the server responds to all the client requests. It comes out of an indefinite loop only when the administrator shuts down the server manually.

All the games also run in an infinite loop. The game will accept the user requests until the user exits from the game.

# Infinite loop with for loop:

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

# Infinite loop with while loop:

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

# Infinite loop with do-while loop:

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

# Programming Questions based on Loop

42. Write a C program for counting of 1 to 10.
43. Write a C program for reverse counting of 1 to 10.
44. Write a C program for table of any number.
45. Write a C program to print the sum of first 50 integer values.
46. Write a C program to print the sum of the series:

$1+2+3+4+5+6+\dots$ .....n terms

47. Write a C program to print the sum of the series:

$2^2+4^2+6^2+8^2+\dots$ .....n terms

48. Write a C program to print the sum of the series:

$1^3+3^3+5^3+\dots$ ..... n terms

49. Write a C program to print the sum of the series:

$1^{1/2}+3^{1/2}+5^{1/2}+\dots$ ..... n terms

50. Write a C program to print the sum of the series:

$-1+2-3+4-5+\dots$ ..... n terms

51. Write a C program to print the sum of the series:

$1^2+2^{1/2}+3^2+4^{1/2}+\dots$ ..... nterms

52. Write a C program to print even number between 1 to 100 using while loop.

53. Write a C program to print the multiplication table of the number entered by the user.

The table should get displayed in the following form.

7 \* 1 = 7

7 \* 2 = 14

7 \* 3 = 21

7 \* 4 = 28

7 \* 5 = 35

.

.

7 \* 10 = 70

54. Write a C program to print reverse of a number.

55. Write a C program to check the given number is palindrome or not.

56. Write a C program to find whether a given number is armstrong or not.

57. Write a C program to find whether a given number is prime or not.

58. Write a C program to print prime number between 1 to 100.

59. Write a program to calculate the Fibonacci series from 1 to n

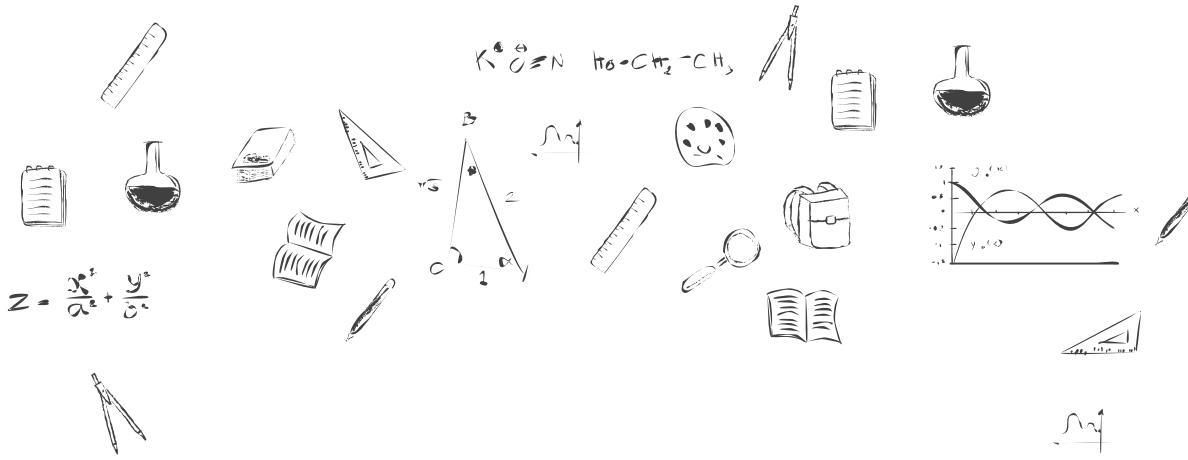
0 1 1 2 3 5 8 13.....n

60. Write a C program to find the factorial of any number entered through the keyboard

61. Write a C program for counting of 1 to 10 using do-while loop.

# C Programming

## The Loops



# Nesting of Loops

We can create nested loops by placing one loop inside another loop. The nested loops are also known as “loop inside loop”.

We can put any type of loop in another type. for example we can write a for loop inside while loop and while inside another while loop etc.

The most common is the nesting for loops.

# Why we need nested for loops?

Nested for loops are used to store and access values from matrix or 2-dimensional arrays.

## Syntax of nested for loop:

```
for ( initialization; condition; increment )
{
    for (initialization; condition; increment )
    {
        // statement of inside loop
    }
    // statement of outside loop
}
```

# Syntax of nested while loop:

```
while(condition)
{
    while(condition)
    {
        statement(s);
    }
    statement(s);
}
```

## Syntax of nested do-while loop:

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

# Programming Questions based on Nested Loop

62. Write a C program to print the sum of the following pattern:

```
*
```

```
* *
```

```
* * *
```

```
* * * *
```

```
* * * * *
```

63. Write a C program to print the sum of the following pattern:

```
1
```

```
2 2
```

```
3 3 3
```

```
4 4 4 4
```

```
5 5 5 5 5
```

64. Write a C program to print the sum of the following pattern:

```
1
```

```
2 3
```

```
4 5 6
```

```
7 8 9 10
```

```
11 12 13 14 15
```

65. Write a C program to print the sum of the following pattern:

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

66. Write a C program to print the sum of the following pattern:

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

67. Write a C program to print the sum of the following pattern:

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

68. Write a C program to print the sum of the following pattern:

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

69. Write a C program to print the sum of the following pattern:

```
1  
2 2  
3 3 3  
4 4 4 4  
5 5 5 5 5
```

70. Write a C program to print the sum of the following pattern:

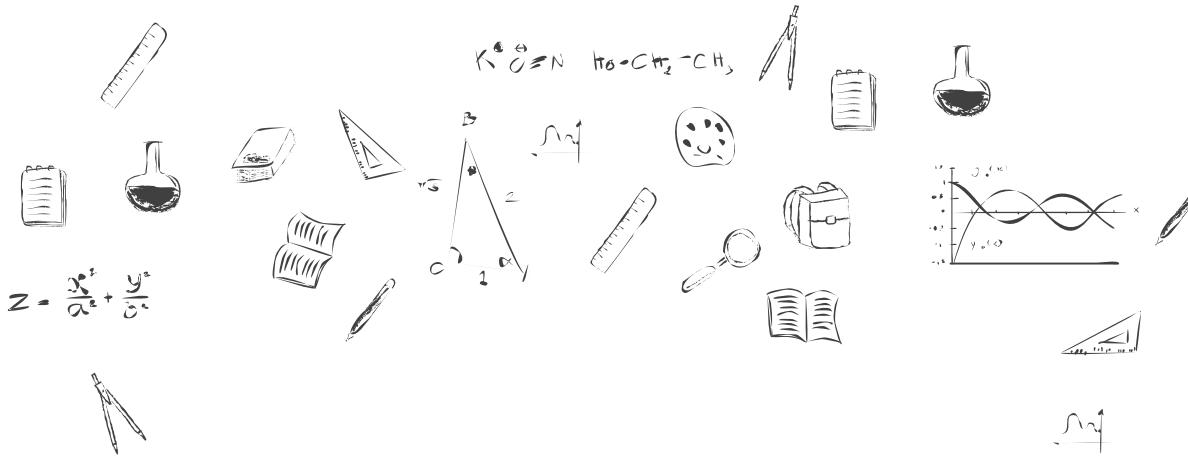
```
1  
2 3  
4 5 6  
7 8 9 10  
11 12 13 14 15
```

71. Write a C program to print the sum of the following pattern:

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

# C Programming

## **Break,Continue , goto Statement**



# Break statement

## **When we use break statement?**

Basically break statements are mostly used in two situations .

1. when we are not sure about the actual number of iterations for the loop.
2. We want to terminate the loop based on some condition.

## The break statement:

The break is a keyword in C Programming.

The break is used to terminate the execution of loop body and transfers the control to the next statement following the loop.

It can also be used to terminate a case in the switch statement. Whenever it is encountered in switch-case block, the control comes out of the switch-case.

If we are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

The break statement is almost always used with if...else statement inside the loop.

## Syntax:

The syntax for a break statement in C is as follows –

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

# **When we use break statement?**

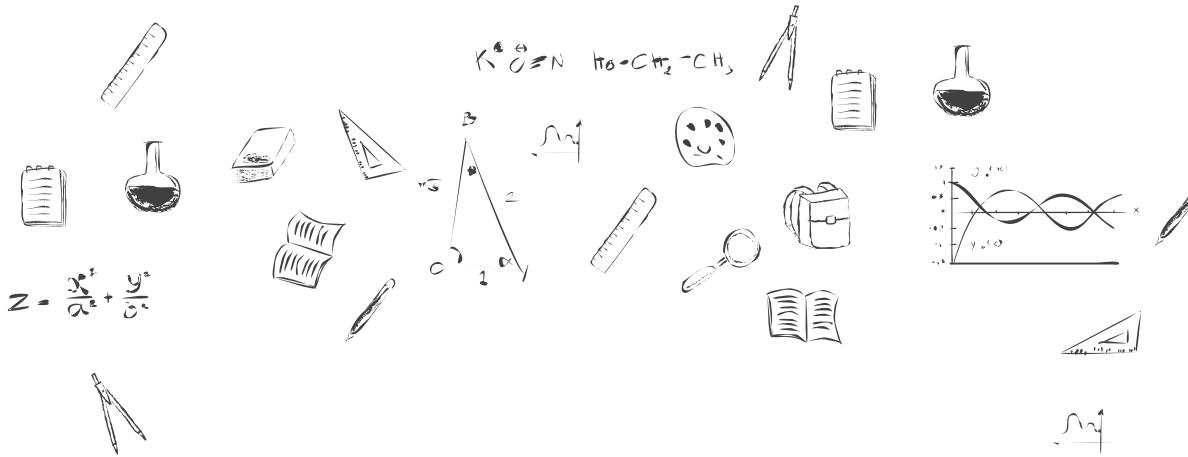
1. when we are not sure about the actual number of iterations for the loop.
2. We want to terminate the loop based on some condition.

## 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);
    getch();
}
```

## Output:

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



# Continue statement

## The continue statement:

The continue is also a keyword in C Programming.

The continue statement skips the current iteration of the loop and control automatically passes to the beginning of the loop.

The continue statement is always used with if...else statement inside the loop.

## Syntax:

The syntax for a continue statement in C is as follows –

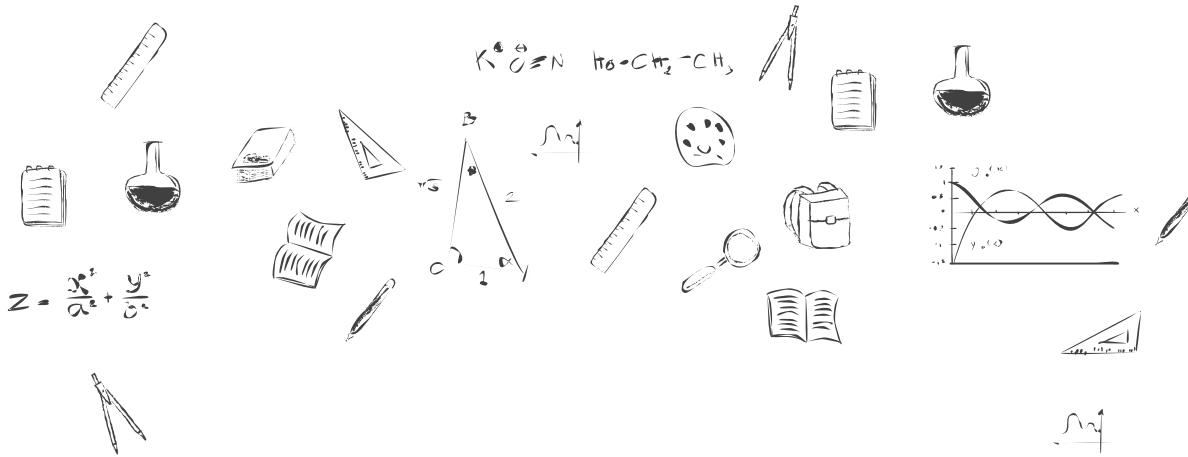
```
//loop statements  
continue;  
//some lines of the code which is to be skipped
```

## Example:

```
#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
```



# goto statement

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

# Syntax:

**label:**

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

```
#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
```

In real life we are often faced few situations where we are required to make a choice between a number of alternatives.

For example, which course to enrol or which city to visit or which school to join.

C provides a special control instruction that allows us to manage such cases effectively rather than using nested if-else or else-if ladder statements.

# Rules for switch Case:

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. Case labels always end with a colon ( : ). Each of these cases is associated with a block.
5. You are not allowed to have duplicate cases

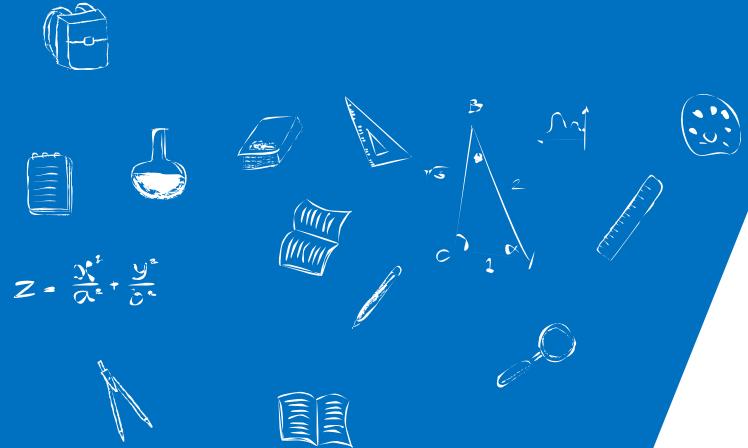
The default statement is optional. Even if the switch case statement do not have a default statement, it would run without any problem.

The expression is evaluated once and compared with the values of each case label.

When a match is found, the program executes the statements following that case, and all subsequent case and default statements as well.

If a case match is NOT found, then the default statement is executed, and the control goes out of the switch block.

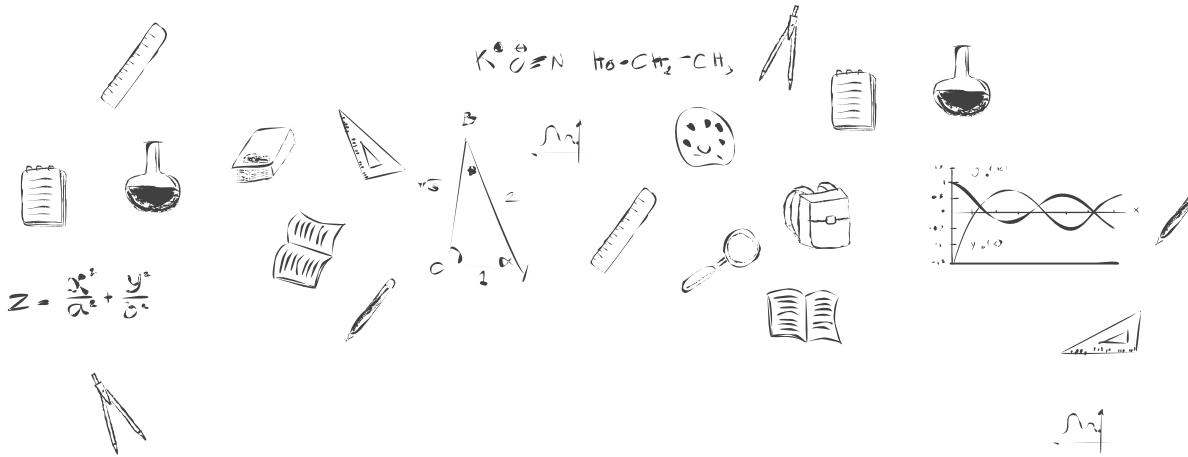
# C Programming



## Switch Case in C Programming

## **Switch Case:**

The switch case control statement allows us to make a decision from the number of choices.



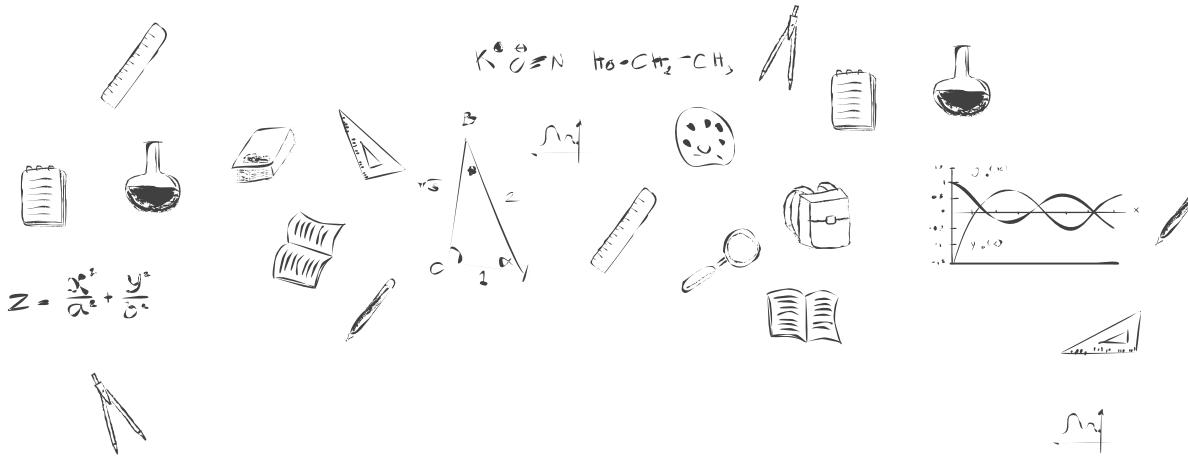
# Syntax of Switch Case

```
switch ( expression )
{
    case constant 1 :
        statement(s); //code to be executed;

    case constant 2 :
        statement(s); //code to be executed;

    case constant 3 :
        statement(s); //code to be executed;

    default :
        statement(s); //code to be executed;
}
```



# Break Statement

The break statement when used in a switch takes the control outside the switch.

The break statement in switch case is optional.

When a break statement is encountered, the switch terminates, and the flow of control jumps to the next line following the switch statement.

If there is no break statement found in the case, all the cases will be executed after the matched case. It is known as fall through the state of C switch statement.

It is not necessary to put these cases in ascending order. In fact, you can put these cases in any order as you want.

```
switch ( integer expression )
{
    case constant 3 :
        statement(s); //code to be executed;
        break; //optional

    case constant 1 :
        statement(s); //code to be executed;
        break; //optional

    case constant 2 :
        statement(s); //code to be executed;
        break; //optional

    default :
        statement(s);
}
```

## FACTS RELATED TO SWITCH

1

You are not allowed to add duplicate cases.

```
int main() {
    int x = 1;
    switch(x)
    {
        case 1: printf("x is 1");
                  break;
        case 1: printf("x is 1");
                  break;
        case 2: printf("x is 2");
                  break;
    }
}
```

Output:

```
prog.c:9:6: error: duplicate case value
    case 1: printf("x is 1");
              ^
prog.c:7:6: error: previously used here
    case 1: printf("x is 1");
```

## FACTS RELATED TO SWITCH

2

Only those expressions are allowed in switch which results in an integral constant value.

ALLOWED

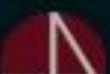
```
int main() {
    int a = 1, b = 2, c = 3;
    switch(a+b*c)
    {
        case 1: printf("choice 1");
        break;
        case 2: printf("choice 2");
        break;
        default: printf("default");
        break;
    }
}
```

NOT ALLOWED

```
int main() {
    float a = 1.15, b = 2.0, c = 3.0;
    switch(a+b*c)
    {
        case 1: printf("choice 1");
        break;
        case 2: printf("choice 2");
        break;
        default: printf("default");
        break;
    }
}
```

Output:

default



## FACTS RELATED TO SWITCH

③

Float value is not allowed as a constant value in **case label**. Only integer constants/constant expressions are allowed in case label.

NOT ALLOWED

```
int main() {
    float x = 3.14;
    switch(x)
    {
        case 3.14: printf("x is 3.14");
                    break;
        case 1.1: printf("x is 1.14");
                    break;
        case 2: printf("x is 2");
                    break;
    }
}
```

```
prog.c:7:6: error: case label does not reduce to an integer constant
    case 3.14: printf("x is 3.14");
                 ^
prog.c:9:6: error: case label does not reduce to an integer constant
    case 1.1: printf("x is 1.14");
```

ALLOWED

```
int main()
{
    int x = 23;
    switch(x)
    {
        case 3+3: printf("choice 1");
                    break;
        case 3+4*5: printf("choice 2");
                     break;
        default: printf("default");
                   break;
    }
}
```



## FACTS RELATED TO SWITCH

③

Float value is not allowed as a constant value in **case label**. Only integer constants/constant expressions are allowed in case label.

### NOT ALLOWED

```
int main() {
    float x = 3.14;
    switch(x)
    {
        case 3.14: printf("x is 3.14");
                    break;
        case 1.1: printf("x is 1.14");
                    break;
        case 2: printf("x is 2");
                    break;
    }
}
```

```
prog.c:7:6: error: case label does not reduce to an integer constant
    case 3.14: printf("x is 3.14");
                 ^
prog.c:9:6: error: case label does not reduce to an integer constant
    case 1.1: printf("x is 1.14");
```

### ALLOWED

```
int main() {
    int x = 23;
    switch(x)
    {
        case 3+3: printf("choice 1");
                    break;
        case 3+4*5: printf("choice 2");
                     break;
        default: printf("default");
                   break;
    }
}
```



## FACTS RELATED TO SWITCH

4

Variable expressions are not allowed in case labels. Although macros are allowed.

```
int main() {
    int x = 2, y = 2, z = 23;
    switch(x)
    {
        case y: printf("Number is 2");
        break;
        case z: printf("Number is 23");
        break;
        default: printf("default case");
        break;
    }
}
```

```
prog.c:7:6: error: case label does not reduce to an integer constant
    case y: printf("Number is 2");
               ^
prog.c:9:6: error: case label does not reduce to an integer constant
    case z: printf("Number is 23");
```

## FACTS RELATED TO SWITCH

4

Variable expressions are not allowed in case labels. **Although macros are allowed.**

```
#include <stdio.h>
#define y 2
#define z 23
int main() {
    int x = 2;
    switch(x)
    {
        case y: printf("Number is 2");
                   break;
        case z: printf("Number is 23");
                   break;
        default: printf("default case");
                   break;
    }
}
```

## FACTS RELATED TO SWITCH

⑤

- Default can be placed anywhere inside switch. It will still get evaluated if no match is found.

```
int main() {
    int x = 2;
    switch(x)
    {
        default: printf("default case");
        break;
        case 1: printf("Number is 1");
        break;
        case 2: printf("Number is 2");
        break;
    }
}
```

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

```
int x,y,z;  
char a,b;  
float f;
```

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

## Example:

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

## Example:

```
#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
```

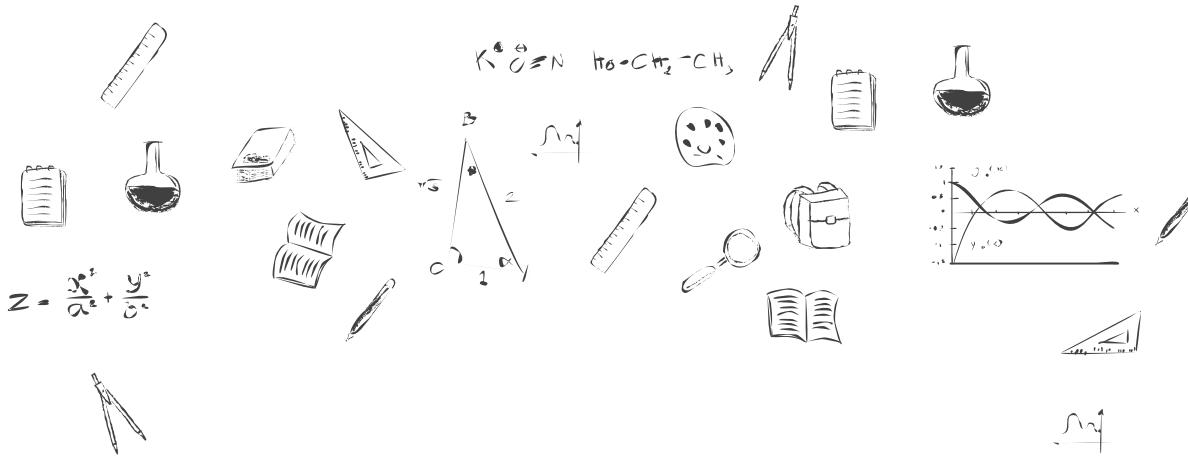
### Output

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

# C Programming

## Array

### In c Programming



# Introduction to Array

# Array:

An array is a collection of similar elements or similar data types i.e integer, real or character etc.

Subscripted variable is another name of an array.

We can't group different data types in array.

Array is called as homogeneous data type.

These similar elements could be all ints, or all floats, or all chars, etc. But array of characters is called a ‘string’, whereas an array of ints or floats is called simply an array.

Remember that all elements of any given array must be of the same type. i.e. we cannot have an array of 10 numbers, of which 5 are ints and 5 are floats.

# Declaration of C Array

## Syntax:

Datatype array\_name[size];

i.e.            int a[5];

An array needs to be declared just like other variables so that the compiler will know what kind of an array and how large an array we want.

We are using a statement here:

```
int a[5] ;
```

Here, int specifies the type of the variable, just as it does with ordinary variables and the word ‘a’ specifies the name of the variable.

The 5 in square brackets however is new. The number 5 tells how many elements of the type int will be in our array.

This number is often called the ‘dimension’ of the array. The square bracket ( [ ] ) tells the compiler that we are dealing with an array.

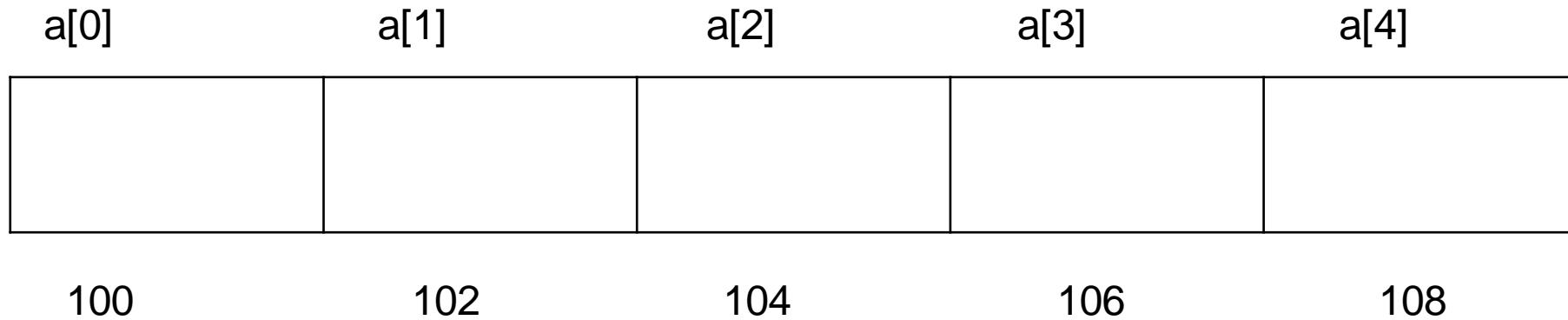
When we declare an array of size of 5 here. It will stored in memory like this.

These are the indexes and as we know that indexes start from 0 in array.

The number given below are addresses of the elements of the array. array consist of contiguous memory locations.

# Example:

```
int a[5];
```



# Initialization of C Array

# Method 1

It is possible to initialize an array during declaration.

For example,

```
int mark[5]={20,30,40,50,60};
```

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

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

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

marks[0]	marks[1]	marks[2]	marks[3]	marks[4]
20	30	40	50	60

# Example

```
#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

# Method 2

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index.

Consider the following example.

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

## Example:

#include<stdio.h>	<b>Output</b>
int main(){	80
int i=0;	60
int marks[5];//declaration of array	70
marks[0]=80;//initialization of array	85
marks[1]=60;	75
marks[2]=70;	
marks[3]=85;	
marks[4]=75;	
//traversal of array	
for(i=0;i<5;i++)	
{	
printf("%d \n",marks[i]);	
return 0;	
}	

## Method 3:Dynamic Initialization of array(Example)

```
#include<stdio.h>
int main(){
int i=0, marks[5];
Printf("\nEnter marks\n")
for(i=0;i<5;i++)
{
scanf("%d \n",&marks[i]); //dynamic
initialization of array
}
for(i=0;i<5;i++)
{
printf("Output is:%d \n",marks[i]);
}
return 0;
}
```

<b>Output</b>
Enter marks
20
30
40
50
60
Output is: 20
30
40
50
60

# Change Value of Array elements

```
int mark[5] = {19, 10, 8, 17, 9}
```

```
// make the value of the third element to -1
```

```
mark[2] = -1;
```

```
// make the value of the fifth element to 0
```

```
mark[4] = 0;
```

# Input and Output Array Elements

Here's how you can take input from the user and store it in an array element.

```
// take input and store it in the 3rd element  
scanf("%d", &mark[2]);
```

```
// take input and store it in the ith element  
scanf("%d", &mark[i-1]);
```

Here's how you can print an individual element of an array.

```
// print the first element of the array
```

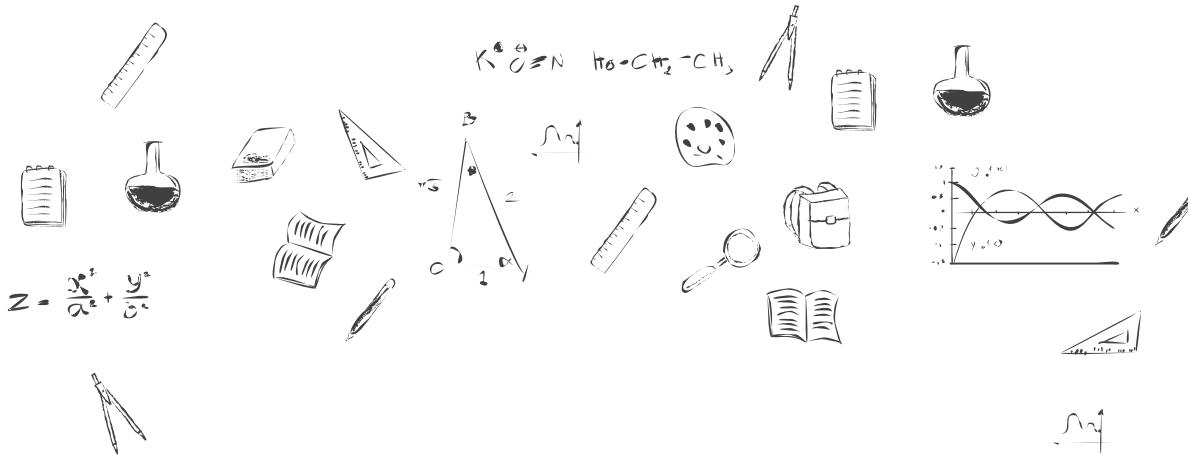
```
printf("%d", mark[0]);
```

```
// print the third element of the array
```

```
printf("%d", mark[2]);
```

```
// print ith element of the array
```

```
printf("%d", mark[i-1]);
```



# Why do we need an array?

# Why do we need an array?

Consider a case where we need to find out the average of 100 integer numbers entered by user.

In C, we have two ways to do this:

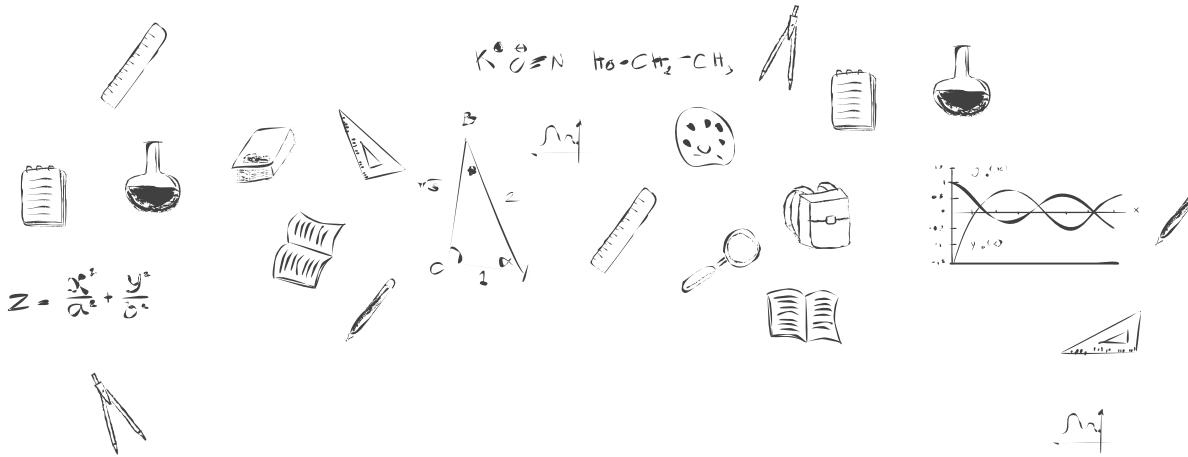
1. Declare 100 variables with int data type and then perform 100 scanf() operations to store the entered values in the variables and then at last calculate the average of them.

2. Have a single integer array to store all the values, loop the array to store all the entered values in array and later calculate the average.

Obviously the second solution is better, it is convenient to store same data types in one single variable and later access them using array index.

# How to access element of an array in C?

You can use array subscript (or index) to access any element stored in array. Subscript starts with 0, which means arr[0] represents the first element in the array arr.



# Why do we need an array?

# Why do we need an array?

Let's Consider a situation where we need to store five integer numbers and print them.

In C programming, we have two ways to do this:

In the first approach, we can declare 5 variables with int data type and then use 5%d as a format specifiers in scanf() function to store the entered values in the variables and then at last print the value of variables.

While, In the second approach, we can have a single integer array to store all the values, loop the array to store all the entered values in array and later print the values of each element.

Obviously the second solution is better, it is convenient to store same data types in one single variable.

```
void main()
{
    int a,b,c,d,e;
    clrscr()
    print("\n Enter the values:");
    scanf("%d%d%d%d%d",&a,&b,&c,&d,&e);
    printf(" The values are %d%d%d%d%d",a,b,c,d,e);
    getch();
}
```

```
Void main()
{
    int a[5],i;
    clrscr()
    print("\n Enter the values:");
    for(i=0;i<5;i++)
        scanf("%d",&a[i]);
    printf(" The values in array a are ");
    for(i=0;i<5;i++)
        printf("%d",a[i]);
    getch();
}
```

## Example 1: Array Input/Output

```
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>
int main()
{
    int values[5];
    printf("Enter 5 integers: ");

    // taking input and storing it in an array
    for(int i = 0; i < 5; ++i)
    {
        scanf("%d", &values[i]);
    }
    printf("Displaying integers: ");
    // printing elements of an array
    for(int i = 0; i < 5; ++i)
    {
        printf("%d\n", values[i]);
    }
    return 0;
}
```

Enter 5 integers: 1

-3

34

0

3

Displaying integers:

1

-3

34

0

3

## Example 2: Calculate Average

```
// Program to find the average of n numbers using arrays
#include <stdio.h>
int main()
{
int marks[10], i, n, sum = 0, average;
printf("Enter number of elements: ");
scanf("%d", &n);
for(i=0; i<n; ++i)
{
printf("Enter number%d: ", i+1);
scanf("%d", &marks[i]);
// adding integers entered by the user to the sum variable
sum += marks[i];
}
average = sum/n;
printf("Average = %d", average);
return 0;
}
```

Enter n: 5  
Enter number1: 45  
Enter number2: 35  
Enter number3: 38  
Enter number4: 31  
Enter number5: 49  
Average = 39

### **Example 3: 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);
}
```

## **Null Pointer**

A null pointer is a pointer which points nothing.

Some uses of the null pointer are:

- a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
- b) To pass a null pointer to a function argument when we don't want to pass any valid memory address.
- c) To check for null pointer before accessing any pointer variable. So that, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.

**Let's look at the situations where we need to use the null pointer.**

- **When we do not assign any memory address to the pointer variable.**

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

```
int main()
```

```
{
```

```
    int *ptr;
```

```
printf("Address: %d", ptr); // printing the value of ptr.  
  
printf("Value: %d", *ptr); // dereferencing the illegal  
pointer  
  
return 0;  
  
}
```

In the above code, we declare the pointer variable `*ptr`, but it does not contain the address of any variable. The dereferencing of the uninitialized pointer variable will show the compile-time error as it does not point any variable. According to the stack memory concept, the local variables of a function are stored in the stack, and if the variable does not contain any value, then it shows the garbage value. The above program shows some unpredictable results and causes the program to crash. Therefore, we can say that keeping an uninitialized pointer in a program can cause serious harm to the computer.

## Example

```
#include <stdio.h>  
int main() {  
    int *ptr= NULL;//initialize the pointer as null.
```

```
    printf("The value of pointer is %u",ptr);
    return 0;
}
```

## Output

The value of pointer is 0.

## What is difference between uninitialized(wild) pointer and null pointer.

A Pointer in C that has not been initialized till its first use is known as uninitialized pointer. Uninitialized pointer points to some random memory location. NULL pointer in C is a pointer which is pointing to nothing or the base address of the segment.

For Example:

```
int *ptr1 = NULL;
int *ptr2;
```

Here ptr1 is a NULL pointer whereas ptr2 is an uninitialized(wild) pointer.

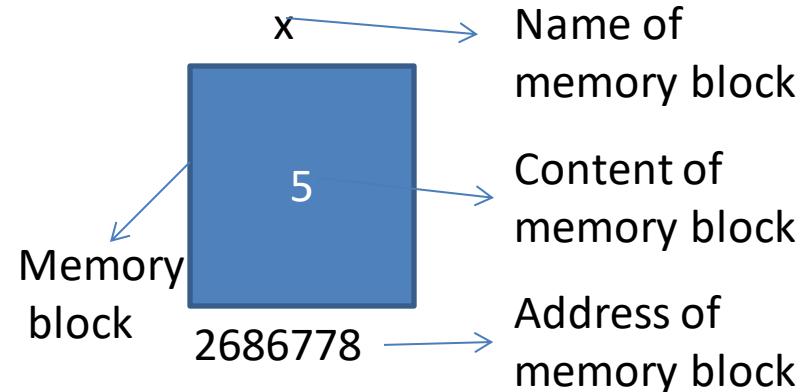
# Address of operator

- & is known as address of operator
- It is an unary operator.
- Operand must be the name of variable.
- & operator gives address of variable.
- **& is also known as referencing operator.**
- If you have a variable *x* in your program, *&x* will give you its address in the memory.
- We have used address numerous times while using the *scanf()* function.
- *scanf("%d", &x);* Here, the value entered by the user is stored in the address of *x* variable.

# Address of operator

- Let's take a working example.

```
#include <stdio.h>
int main()
{
    int x = 5;
    printf("x: %d\n", x);
    // Notice the use of & before x
    printf("address of x: %u", &x);
    return 0;
}
```



- Output**

x: 5

address of x: 2686778

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

# Indirection Operator or dereferencing operator

- \* is indirection pointer.
- It is also known as dereferencing pointer.
- It is an Unary operator
- It takes address as an argument
- \* returns the Content/Container whose address is its argument

# Indirection Operator

- Let's take a working example.

```
#include <stdio.h>
int main()
{
    int x = 5;
    printf("%d\n",x);
    printf("%u",&x);
    printf("%u",*&x);
    return 0;
}
```

- Output**

```
5
2686778
5
```

- 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.

# Pointer Syntax

- Here is how we can declare pointers.

```
int* p;
```

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

- You can also declare pointers in these ways.

```
int *p1;
```

```
int * p2;
```

- Let's take another example of declaring pointers.

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

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

# Assigning addresses to Pointers

- Let's take an example.

```
int* pc, c;
```

```
c = 5;
```

```
pc = &c;
```

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

# Get Value of Thing Pointed by Pointers

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

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

- Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.
- Note:** In the above example, `pc` is a pointer, not `*pc`. You cannot and should not do something like `*pc = &c;`
- By the way, `*` is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

# Changing Value Pointed by Pointers

- Let's take an example.

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

- We have assigned the address of *c* to the *pc* pointer.
- Then, we changed the value of *c* to 1. Since *pc* and the address of *c* is the same, *\*pc* gives us 1.
- Let's take another example.

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

- We have assigned the address of *c* to the *pc* pointer.
- Then, we changed *\*pc* to 1 using *\*pc = 1;*. Since *pc* and the address of *c* is the same, *c* will be equal to 1.

- Let's take one more example.

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

- Initially, the address of *c* is assigned to the *pc* pointer using *pc = &c;*. Since *c* is 5, *\*pc* gives us 5.
- Then, the address of *d* is assigned to the *pc* pointer using *pc = &d;*. Since *d* is -15, *\*pc* gives us -15.

```
int main()
{
    int x = 5,*j;
    j=&x;
    printf("%d %u\n",x,j);
    printf("%d %u",*j,&x);
    printf("%u",*&j);
    return 0;
}
```

Output:  
5 2048  
5 2048  
2048

# Example

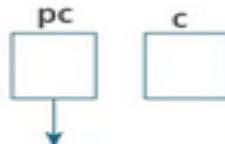
```
#include <stdio.h>
int main()
{
    int* pc, c;
    c = 22;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c); // 22
    pc = &c;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 22
    c = 11;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 11
    *pc = 2;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c); // 2
    return 0;
}
```

## Output

Address of c: 2686784  
Value of c: 22  
Address of pointer pc: 2686784  
Content of pointer pc: 22  
Address of pointer pc: 2686784  
Content of pointer pc: 11  
Address of c: 2686784  
Value of c: 2

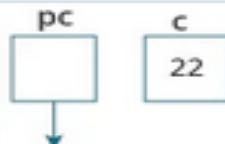
## Explanation of the program

1. `int* pc, c;`



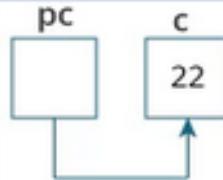
Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created. Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

2. `c = 22;`



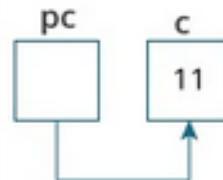
This assigns 22 to the variable `c`. That is, 22 is stored in the memory location of variable `c`.

3. `pc = &c;`



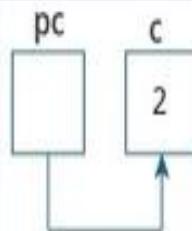
This assigns the address of variable `c` to the pointer `pc`.

4. `c = 11;`



This assigns 11 to variable `c`.

5. `*pc = 2;`



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

# C Pointers & Arrays with Examples

- Traditionally, we access the array elements using its index, but this method can be eliminated by using pointers. Pointers make it easy to access each array element.

```
#include <stdio.h>
int main()
{
    int a[5]={1,2,3,4,5}; //array initialization
    int *p; //pointer declaration
    /*the ptr points to the first element of the array*/
    p=a; /*We can also type simply ptr==&a[0] */
    printf("Printing the array elements using pointer\n");
    for(int i=0;i<5;i++) //loop for traversing array elements
    {
        printf("\n%x",*p); //printing array elements
        p++; //incrementing to the next element, you can also
              write p=p+1 }
    }
    return 0;
}
```

Output:  
1  
2  
3  
4  
5

```
#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

- Pointer + n = pointer + size of (type of pointer)\*n

- Adding a particular number to a pointer will move the pointer location to the value obtained by an addition operation. Suppose p is a pointer that currently points to the memory location 0 if we perform following addition operation,  $p+1$  then it will execute in this manner:

Adding a particular number to a pointer will move the pointer location to the value obtained by an addition operation. Suppose  $p$  is a pointer that currently points to the memory location 0 if we perform following addition operation,  $p+1$  then it will execute in this manner:



#### Pointer Addition/Increment

Since  $p$  currently points to the location 0 after adding 1, the value will become 1, and hence the pointer will point to the memory location 1.

# C Function Pointer

# Declaration of a function pointer

- **Syntax of function pointer**

return type (\*ptr\_name)(type1, type2...);

- For example:

- **int (\*ip) (int);**

In the above declaration, \*ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

- **float (\*fp) (float);**

In the above declaration, \*fp is a pointer that points to a function that returns a float value and accepts a float value as an argument.

- **float (\*fp) (int , int);** // Declaration of a function pointer.
- **float func( int , int );** // Declaration of function.
- **fp = func;** // Assigning address of func to the fp pointer.
- In the above declaration, '**fp**' pointer contains the address of the '**func**' function.
- Note: Declaration of a function is necessary before assigning the address of a function to the function pointer.

# Calling a function through a function pointer

- We already know how to call a function in the usual way. Now, we will see how to call a function using a function pointer.

- Suppose we declare a function as given below:

```
float func(int , int); // Declaration of a function.
```

- Calling an above function using a usual way is given below:

```
result = func(a , b); // Calling a function using usual ways.
```

- Calling a function using a function pointer is given below:

```
result = (*fp)( a , b); // Calling a function using function pointer.
```

Or

```
result = fp(a , b); // Calling a function using function pointer, and indirection operator can be removed.
```

- The effect of calling a function by its name or function pointer is the same. If we are using the function pointer, we can omit the indirection operator as we did in the second case. Still, we use the indirection operator as it makes it clear to the user that we are using a function pointer.

# Example

```
#include <stdio.h>
int add(int,int);
int main()
{
    int a,b;
    int (*ip)(int,int);
    int result;
    printf("Enter the values of a and b : ");
    scanf("%d %d",&a,&b);
    ip=add;
    result=(*ip)(a,b);
    printf("Value after addition is : %d",result)
    ;
    return 0;
}
int add(int a,int b)
{
    int c=a+b;
    return c;
}
```

Output:

```
V / / $ Enter the values of a and b : 4 55 Value after addition is : 59 ...Program finished with exit code 0 Press ENTER to exit console. ]
```

## What is the Size of a Pointer in C?

In straightforward terms, a pointer is a variable that stores the address of another variable. They are declared with the general syntax as follows:

*datatype \*variable\_name;*

Here, the data type is that of the variable whose address the pointer will store.

But then what will be the size of the pointer in C?

A possible guess is that the size will be the same as that of the variable whose address the pointer is storing, **but it's not**.

The size of the pointer in C is not fixed, and it depends on certain factors. So before we find out what the actual size of a pointer in C is, let's discuss these factors.

## **Factors on Which the Size of a Pointer in C Depend**

As mentioned above, the size of the pointer in C is not fixed. Instead, it depends upon factors like the CPU architecture and the processor's word size, to be more specific.

Thus, the word size is the main determining factor in finding the size of the pointer. Now, since the word size is the same for a particular computer system, the size of the pointer in C will also be the same, irrespective of the variable's data type whose address it's storing.

**For example**, the size of a char pointer in a 32-bit processor is 4 bytes, while the size of a char pointer in a 16-bit processor is 2 bytes. If the system is 64-bit, size of pointer is 8 bytes.

To understand this point better, let us see the size of a pointer in C of different data types with the help of some examples. But, before we go to the examples, we must first learn how to print the size of the pointer in C.

## **How to Print the Size of a Pointer in C?**

As we already know, the syntax of the `sizeof( )` operator, printing the size of pointer in C, won't be complicated at all.

The basic syntax of the `sizeof( )` operator is **`sizeof(data/data type/expression)`**.

We can easily determine that to find the size of the pointer in C, the operand must be a pointer.

Thus, the syntax is **`sizeof(pointer variable)`**

Now that we know how to find the size of the pointer in C let us see some examples.

# Size of a Pointer in C of Different Data Types

## 1. Size of Character Pointer

The code to find the size of a character pointer in C is as follows:

```
#include<stdio.h>
int main()
{
    char c='A';
    char *ptr=&c;
    printf("The size of the character pointer is %d bytes",sizeof(ptr));
    return 0;
}
```

Output:

The size of the character pointer is 8 bytes.

**Note:** This code is executed on a 64-bit processor.

## 2. Size of Double Pointer in C

As we already know, the size of pointer in C is machine-dependent, so the size of the double-pointer should be the same as that of the character-pointer. Let us confirm that with the following code.

```
#include<stdio.h>
int main()
{
    double x=3.14;
    double *ptr=&x;
    printf("The size of the double pointer is %d bytes",sizeof(ptr));
    return 0;
}
```

Output:

The size of the double pointer is 8 bytes

**Note:** This code is executed on a 64-bit processor.

## What is the size of void pointer in C

The size of void pointer varies system to system. If the system is 16-bit, size of void pointer is 2 bytes. If the system is 32-bit, size of void pointer is 4 bytes. If the system is 64-bit, size of void pointer is 8 bytes.

Here is an example to find the size of void pointer in C language,

Example

```
#include <stdio.h>

int main()
{
    void *ptr;
    printf("The size of pointer value : %d", sizeof(ptr));
    return 0;
}
```

Output

The size of pointer value : 8

## void pointer in C

Till now, we have studied that the address assigned to a pointer should be of the same type as specified in the pointer declaration. For example, if we declare the int pointer, then this int pointer cannot point to the float variable or some other type of variable, i.e., it can point to only int type variable. To overcome this problem, we use a pointer to void. A pointer to void means a generic pointer that can point to any data type. We can assign the address of any data type to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

### **Syntax of void pointer**

1. void \*pointer name;

### **Declaration of the void pointer is given below:**

1. void \*ptr;

In the above declaration, the void is the type of the pointer, and 'ptr' is the name of the pointer.

### **Let us consider some examples:**

int i=9; // integer variable initialization.

int \*p; // integer pointer declaration.

```
float *fp;      // floating pointer declaration.  
void *ptr;      // void pointer declaration.  
  
p=fp;          // incorrect.  
  
fp=&i;          // incorrect  
  
ptr=p;          // correct  
  
ptr=fp;          // correct  
  
ptr=&i;          // correct
```

## Size of the void pointer in C

The size of void pointer varies system to system. If the system is 16-bit, size of void pointer is 2 bytes. If the system is 32-bit, size of void pointer is 4 bytes. If the system is 64-bit, size of void pointer is 8 bytes.

Here is an example to find the size of void pointer in C language,

### Example

```
#include <stdio.h>  
int main() {  
    void *ptr;  
    printf("The size of pointer value : %d", sizeof(ptr));  
    return 0;
```

}

## Output

The size of pointer value : 8

In the above example, a void type pointer variable is created and by using sizeof() function, the size of void pointer is found out.

**Let's look at the below example:**

```
#include <stdio.h>

int main()
{
    void *ptr = NULL; //void pointer
    int *p = NULL;// integer pointer
    char *cp = NULL;//character pointer
    float *fp = NULL;//float pointer
    //size of void pointer
    printf("size of void pointer = %d\n\n",sizeof(ptr));
    //size of integer pointer
    printf("size of integer pointer = %d\n\n",sizeof(p));
    //size of character pointer
```

```
    printf("size of character pointer = %d\n\n",sizeof(cp))
);

//size of float pointer

printf("size of float pointer = %d\n\n",sizeof(fp));

return 0;

}
```

## Output

```
size of void pointer = 8

size of integer pointer = 8

size of character pointer = 8

size of float pointer = 8
```

## Advantages of void pointer

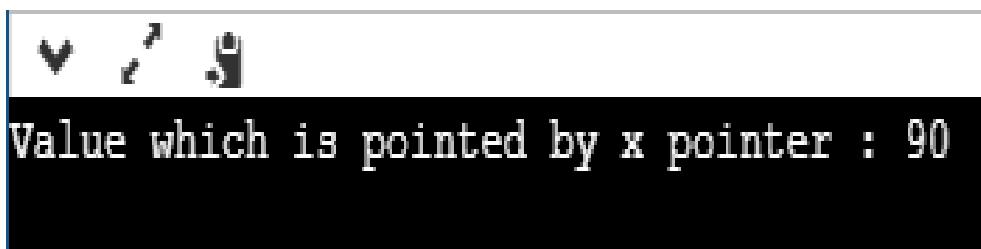
Following are the advantages of a void pointer:

- The malloc() and calloc() function return the void pointer, so these functions can be used to allocate the memory of any data type.

```
#include <stdio.h>
#include<malloc.h>
int main()
{
    int a=90;
    int *x = (int*)malloc(sizeof(int)) ;
    x=&a;
    printf("Value which is pointed by x pointer : %d",*x);

    return 0;
}
```

## Output



- The void pointer in C can also be used to implement the generic functions in C.

## **It has some limitations –**

- 1) It can't be used as dereferenced.
- 2) Pointer arithmetic is not possible with void pointer due to its concrete size.

### **1) Dereferencing a void pointer in C**

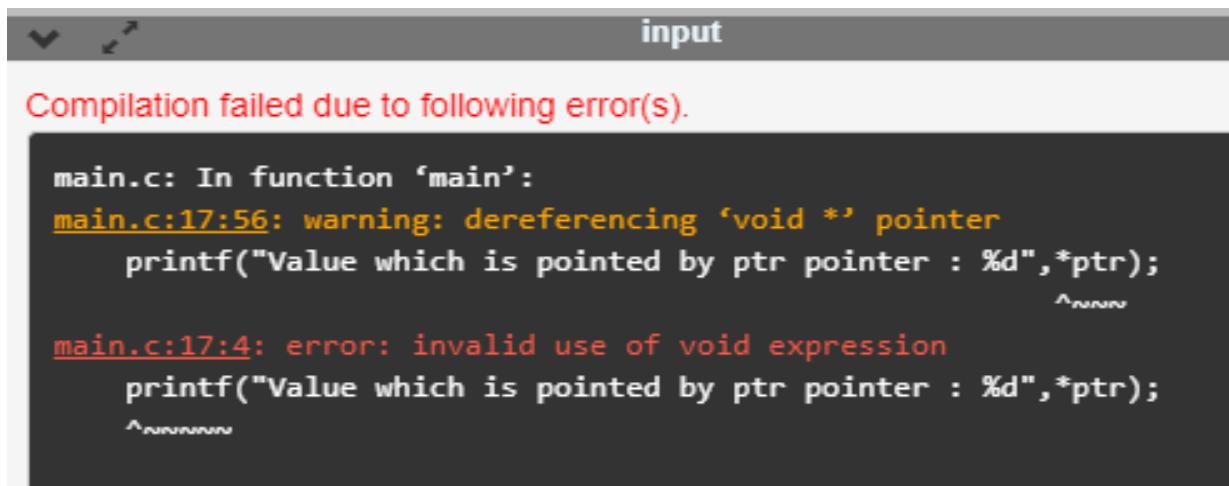
The void pointer in C cannot be dereferenced directly.  
Let's see the below example.

```
#include <stdio.h>

int main()
{
    int a=90;
    void *ptr;
    ptr=&a;
    printf("Value which is pointed by ptr pointer : %d",*ptr);
    return 0;
}
```

In the above code, `*ptr` is a void pointer which is pointing to the integer variable '`a`'. As we already know that the void pointer cannot be dereferenced, so the above code will give the compile-time error because we are printing the value of the variable pointed by the pointer '`ptr`' directly.

## Output



The screenshot shows a terminal window with the title bar labeled "input". Inside the window, the text "Compilation failed due to following error(s)." is displayed in red. Below it, two error messages are shown in white text on a black background:

```
main.c: In function 'main':  
main.c:17:56: warning: dereferencing 'void **' pointer  
    printf("Value which is pointed by ptr pointer : %d", *ptr);  
                           ^~~~~~  
  
main.c:17:4: error: invalid use of void expression  
    printf("Value which is pointed by ptr pointer : %d", *ptr);  
   ^~~~~~
```

Now, we rewrite the above code to remove the error.

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

```
int main()
```

```
{
```

```
    int a=90;
```

```
    void *ptr;
```

```
    ptr=&a;
```

```
    printf("Value which is pointed by ptr pointer : %d",*(  
int*)ptr);  
  
    return 0;  
  
}
```

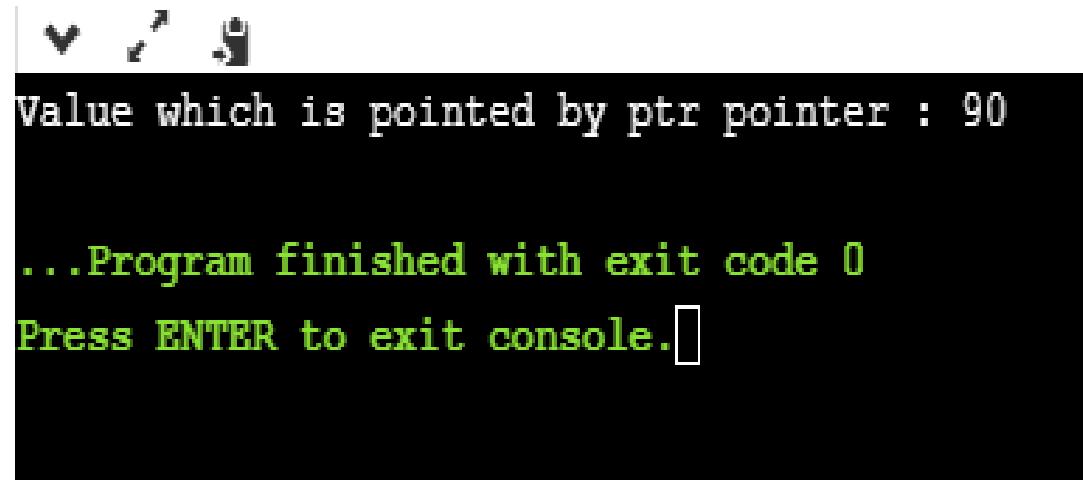
In the above code, we typecast the void pointer to the integer pointer by using the statement given below:

**(int\*)ptr;**

Then, we print the value of the variable which is pointed by the void pointer 'ptr' by using the statement given below:

**\*(int\*)ptr;**

## Output



```
Vim [File] [Edit] [Search]  
Value which is pointed by ptr pointer : 90  
...Program finished with exit code 0  
Press ENTER to exit console.[]
```

## **2) Arithmetic operation on void pointers**

We cannot apply the arithmetic operations on void pointers in C directly. We need to apply the proper typecasting so that we can perform the arithmetic operations on the void pointers.

**Let's see the below example:**

```
#include<stdio.h>

int main()
{
    float a[4]={6.1,2.3,7.8,9.0};
    void *ptr;
    ptr=a;
    for(int i=0;i<4;i++)
    {
        printf("%f,*ptr);
        ptr=ptr+1;      // Incorrect.
```

```
} }
```

The above code shows the compile-time error that "**invalid use of void expression**" as we cannot apply the arithmetic operations on void pointer directly, i.e., `ptr=ptr+1`.

**Let's rewrite the above code to remove the error.**

```
#include<stdio.h>

int main()

{
    float a[4]={6.1,2.3,7.8,9.0};

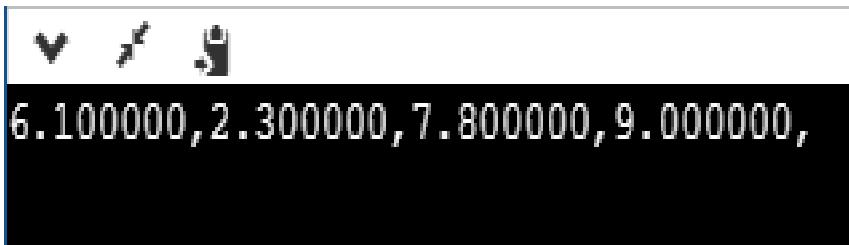
    void *ptr;

    ptr=a;

    for(int i=0;i<4;i++)
    {
        printf("%f,*((float*)ptr+i));
    }
}
```

The above code runs successfully as we applied the proper casting to the void pointer, i.e., `(float*)ptr` and then we apply the arithmetic operation, i.e., `*((float*)ptr+i)`.

## Output



```
6.100000, 2.300000, 7.800000, 9.000000,
```

## Why we use void pointers?

We use void pointers because of its reusability. Void pointers can store the object of any type, and we can retrieve the object of any type by using the indirection operator with proper typecasting.

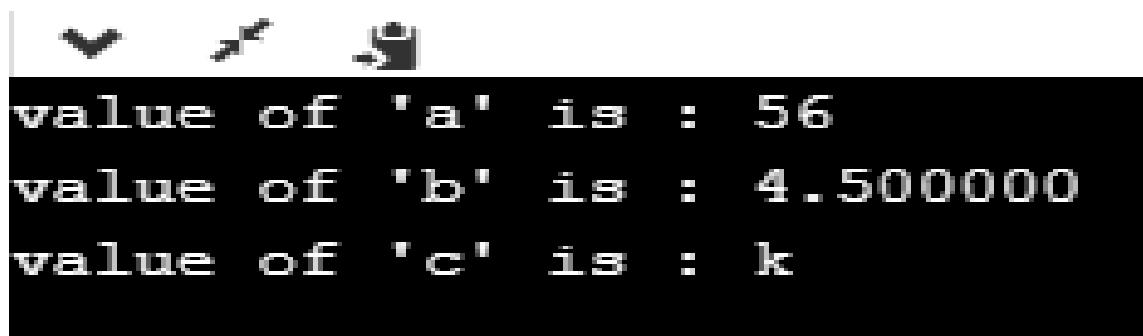
## Let's understand through an example.

```
#include<stdio.h>

int main()
{
    int a=56; // initialization of a integer variable 'a'.
    float b=4.5; // initialization of a float variable 'b'.
```

```
char c='k'; // initialization of a char variable 'c'.  
  
void *ptr; // declaration of void pointer.  
  
// assigning the address of variable 'a'.  
  
ptr=&a;  
  
printf("value of 'a' is : %d",*((int*)ptr));  
  
// assigning the address of variable 'b'.  
  
ptr=&b;  
  
printf("\nvalue of 'b' is : %f",*((float*)ptr));  
  
// assigning the address of variable 'c'.  
  
ptr=&c;  
  
printf("\nvalue of 'c' is : %c",*((char*)ptr));  
  
return 0;  
  
}
```

## Output



```
value of 'a' is : 56  
value of 'b' is : 4.500000  
value of 'c' is : k
```

## **What is the size of void pointer in C**

The size of void pointer varies system to system. If the system is 16-bit, size of void pointer is 2 bytes. If the system is 32-bit, size of void pointer is 4 bytes. If the system is 64-bit, size of void pointer is 8 bytes.

Here is an example to find the size of void pointer in C language,

Example

```
#include <stdio.h>

int main()
{
    void *ptr;
    printf("The size of pointer value : %d", sizeof(ptr));
    return 0;
}
```

Output

The size of pointer value : 8

# **String**

- C has a few built-in data types.  
They are int, short, long, float, double, long double and char.
- As you see, there is no built-in string or str (short for string) data type in C language.
- Hence, to display a String in C, you need to make use of a character array.

# How to declare a string?

- Here's how you can declare strings:

**char c[5];**

- Array index starts from 0, so it will be represented as in the figure given below.

c[0]    c[1]    c[2]    c[3]    c[4]



# How to initialize a string?

- There are two ways to initialize a string in c language.
  - By char array
  - By string literal

# Initializing string by char array in C language.

- **Syntax:**

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

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

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

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

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

# Initializing string by the string literal in C language

- Syntax:  
**char c[5] = "abcd";**
- In such case, '\0' will be appended at the end of the string by the compiler.
- While declaring string, size is not mandatory. So we can write the above **Syntax** as given below:  
**char c[] = "abcd";**
- As we know, array index starts from 0, so it will be represented as in the figure given below.

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

# Summarize: How to initialize strings?

- You can initialize strings in a number of ways.

- **char c[] = "abcd";**
- **char c[5] = "abcd";**
- **char c[] = {'a', 'b', 'c', 'd', '\0'};**
- **char c[5] = {'a', 'b', 'c', 'd', '\0'};**

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

c[0]

c[1]

c[2]

c[3]

c[4]

a

b

c

d

\0

# Example:

- `char c[5] = "abcde"; //wrong initialization`
- Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters. This is bad and you should never do this.

# Assigning Values to Strings

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

**char c[100];**

**c = "C programming"; // Error! array type is not assignable.**

# String Example in C

The '%s' is used as a format specifier for the string in c language.

```
#include<stdio.h>
#include <string.h>
int main()
{
    char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
    char ch2[11]="javatpoint";
    printf("Char Array Value is: %s\n", ch);
    printf("String Literal Value is: %s\n", ch2);
    return 0;
}
```

## Output

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

# Reading string value from user in C programming language

- We can read a string value from the user during the program execution. We use the following two methods...
  - Using scanf() method - reads single word
  - Using gets() method - reads a line of text

# **Reading String using scanf() method**

# Read String from the user using scanf function

- You can use the scanf() function to read a string.
- The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).
- Using scanf() method we can read only one word of string. We use %s to represent string in scanf() and printf() methods.

# Example: scanf() to read a string

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

## Output

Enter name: Dennis Ritchie  
Your name is Dennis.

# Explanation of previous Example

- Even though *Dennis Ritchie* was entered in the above program, only "*Dennis*" was stored in the *name* string. It's because there was a space after *Dennis*.
- Also notice that we have used the code *name* instead of `&name` with `scanf()`.
- `scanf("%s", name);` This is because *name* is a char array, and we know that array names decay to pointers in C.
- Thus, the *name* in `scanf()` already points to the address of the first element in the string, which is why we don't need to use `&`.

# **Reading String using gets() method**

# Read String from the user using 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.

# Examples of reading string value using gets() method

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char name[50];
    printf("Please enter your name : ");
    gets(name);
    printf("Hello! %s , welcome to btech smart class !!", name);
    return 0;
}
```

Output:

xyz

Hello! xyz , welcome to btech smart class !!

# Displaying string value on console in C programming language

- We can display a string value on the console after the program execution. We use the following two methods...
  - Using printf() method //Already discussed
  - Using puts() method

# Puts() function

# puts() function

- The puts() function is very much similar to printf() function.
- The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function.
- The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.
- **Declaration**  
int puts(char[])

# Example

```
#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: arpit saxena  
Your name is: arpit saxena

# **getchar() function**

- The **getchar function** is part of the `<stdio.h>` header file in C.
- It is used when single character input is required from the user.
- The `getchar()` takes the following form  
`Variable_name=getchar();`

## **Read a single character using the getchar() function**

Let's consider a program to take a single using the getchar() function in C.

### **Program.c**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char c;
    printf("\n Enter a character \n");
    c = getchar(); // get a single character
    printf(" You have passed ");
    putchar(c); // print a single character using putchar
    getch();
}
```

### **Output**

Enter a character

A

You have passed A

## **Read n characters from the user using getchar() function**

Let's consider a program to read n characters using the getchar() function in C.

### **Getchar.c**

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
int main()
{
    char ch;
    printf(" Enter a character ( If we want to exit
        press #) \n");
    while (ch != '#') /* accept the number till the
        user does not enter the # to exit from the l
        oop. */
    {
        ch = getchar();
        printf("\n We have entered the character :
        ");
        putchar(ch); // print a single character
        printf("\n");
    }
    return 0;
}
```

## **Output**

Enter a character ( If we want to exit.. press #) A

We have entered the character: A

We have entered the character:

B

We have entered the character: B

We have entered the character:

C

We have entered the character: C

We have entered the character:

# **strlen()**

- The **strlen()** function calculates the length of a given string. The **strlen()** function is defined in **string.h** header file. It doesn't count null character '\0'.
- **Declaration**

```
int strlen(const char *str)
```

**str:** It represents the string variable whose length we have to find.

# Example 1: C strlen() function

```
#include <stdio.h>
#include <string.h>
int main()
{
char a[20]="Program";
char b[20]={'P','r','o','g','r','a','m','\0'};
// using the %zu format specifier to print size_t
printf("Length of string a = %zu \n",strlen(a));
printf("Length of string b = %zu \n",strlen(b));
return 0;
}
```

## Output:

Length of string a = 7  
Length of string b = 7

**Note that the strlen() function doesn't count the null character \0 while calculating the length.**

# Example 2: C strlen() function

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

int main( )
{
    int len;
    char array[20]="fresh2refresh.com" ;

    len = strlen(array);

    printf ( "\String length = %d \n" , len ) ;
    return 0;
}
```

## Output:

string length = 17

**strcat( )**

- `strcat( )` function in C language concatenates two given strings. It concatenates source string at the end of destination string.
- Syntax for `strcat( )` function is given below.  
**`char * strcat ( char * destination, const char * source );`**
- Example:  
`strcat ( str2, str1 );` – str1 is concatenated at the end of str2.  
`strcat ( str1, str2 );` – str2 is concatenated at the end of str1.
- As you know, each string in C is ended up with null character ('\0').
- In `strcat( )` operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after `strcat( )` operation.

# Example 1: C strcat() function

```
#include <stdio.h>
#include <string.h>
int main( )
{
    char source[ ] = " fresh2refresh";
    char target[ ]= " C tutorial" ;
    printf ( "\nSource string = %s", source ) ;
    printf ( "\nTarget string = %s", target ) ;
    strcat ( target, source ) ;
    printf ( "\nTarget string after strcat( ) = %s",
    target ) ;
}
```

## Output:

Source string = fresh2refresh  
Target string = C tutorial  
Target string after strcat( ) = C tutorial fresh2refresh

# Example 2: C strcat() function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[100] = "This is ", str2[] = "programiz.com";

    // concatenates str1 and str2
    // the resultant string is stored in str1.
    strcat(str1, str2);
    puts(str1);
    puts(str2);
    return 0;
}
```

## Output:

This is programiz.com programiz.com

# **strcmp()**

- strcmp( ) function in C compares two given strings and returns zero if they are same.
- Syntax for strcmp( ) function is given below.  
**int strcmp ( const char \* str1, const char \* str2 );**
- strcmp( ) function is case sensitive. i.e, “A” and “a” are treated as different characters.

## • **strcmp() Parameters**

The function takes two parameters:

**str1** - a string

**str2** - a string

## • **Return Value from strcmp()**

Return Value	Remarks
0	if strings are equal
>0	if the first non-matching character in <i>str1</i> is greater (in ASCII) than that of <i>str2</i> i.e If length of string1 > string2, it returns > 0 value.
<0	if the first non-matching character in <i>str1</i> is lower (in ASCII) than that of <i>str2</i> i.e If length of string1 < string2, it returns < 0

# Example 1: C strcmp() function

```
#include <stdio.h>
#include <string.h>
int main()
{
char str1[] = "abcd",
str2[] = "abCd",
str3[] = "abcd";
int result;
// comparing strings str1 and str2
result = strcmp(str1, str2);
printf("strcmp(str1, str2) = %d\n", result);
// comparing strings str1 and str3
result = strcmp(str1, str3);
printf("strcmp(str1, str3) = %d\n", result);
return 0;
}
```

## Output

strcmp(str1, str2) = 1  
strcmp(str1, str3) = 0

In the program,  
strings str1 and str2 are not  
equal. Hence, the result is a non-  
zero integer.

strings str1 and str3 are equal.  
Hence, the result is 0.

# Example 2: C strcmp() function

```
#include<stdio.h>
#include <string.h>
int main()
{
    char str1[20],str2[20];
    printf("Enter 1st string: ");
    gets(str1);//reads string from consol
    e
    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

**strrev()**

- `strrev( )` function reverses a given string in C language.
- **Syntax**

`char *strrev(char *string);`

**or**

`strrev (string)`

- `strrev( )` function is non standard function which may not available in standard library in C.

# Example: C strrev() function

```
#include<stdio.h>
main ( )
{
    char a[50] ;
    clrscr( );
    printf ("enter a string\n");
    gets (a);
    strrev (a);
    printf("\nreversed string = %s",a)
    getch ( );
}
```

## Output

```
enter a string
Hello
Reverse string = olleH
```

**atoi()**

- The atoi() function in C takes a string (which represents an integer) as an argument and returns its value of type int. So basically the function is used to convert a string argument to an integer.
- Syntax of Atoi() function  
**int atoi(const char \*str)**
- Parameters
  - str – This is the string representation of an integral number.
  - Return Value
  - This function returns the converted integral number as an int value. If no valid conversion could be performed, it returns zero.

# Example: C atoi() function

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    int val;
    char string1[20] = "99";
    val = atoi(string1);
    printf("String value = %s\n", string1);
    printf("Integer value = %d\n", val);
    char string2[] = "Javatpoint";
    val = atoi(string2);
    printf("String value = %s\n", string2);
    printf("Integer value = %d\n", val);
    return (0);
}
```

## Output:

String value = 99  
Integer value = 99  
String value = Javatpoint  
Integer value = 0

# **sscanf()**

- In C, sscanf() is used to read formatted data. It works much like scanf() but the data will be read from a string instead of the console.
- Syntax

sscanf(String\_Name, “format specifier”,&variable)

```
/* sscanf example */
#include <stdio.h>
int main ()
{
    char buffer[30] = "Helloworld 5 ";
    char name [20];
    int age;
    sscanf (buffer,"%s %d",name,&age);
    printf ("Name : %s \n Age : %d \n",name,age);
    return 0;
}
```

### **Output:**

Name : Helloworld  
Age : 5

# Reading one item of the same type

```
/* sscanf example */  
#include<stdio.h>  
int main()  
{  
    char buffer[] = "Hello";  
    char store_value[10];  
    int total_read;  
    total_read = sscanf(buffer, "%s", store_value);  
    printf("Value in buffer: %s",store_value);  
    printf("\nTotal items read: %d",total_read);  
    return 0;  
}
```

**Output:**

Value in buffer: Hello  
Total items read: 1

# Reading multiple items of the same type

```
/* sscanf example */  
#include<stdio.h>  
int main()  
{  
    char* buffer = "Hello World";  
    char store_hello[10], store_world[10];  
    int total_read;  
    total_read = sscanf(buffer, "%s %s" , store_hello, store_world);  
    printf("Value in first variable: %s",store_hello);  
    printf("\nValue in second variable: %s",store_world);  
    printf("\nTotal items read: %d",total_read);  
    return 0;  
}
```

## Output:

Value in first variable: Hello  
Value in second variable: World  
Total items read: 2

# Reading multiple items of different types

```
/* sscanf example */  
#include<stdio.h>  
int main()  
{  
    char* buffer = "Hello 20";  
    char store_string[10];  
    int store_integer;  
    int total_read;  
    total_read = sscanf(buffer, "%s %d", store_string, &store_integer);  
    printf("String value in buffer: %s", store_string);  
    printf("\nInteger value in buffer: %d", store_integer);  
    printf("\nTotal items read: %d", total_read);  
    return 0;  
}
```

## Output:

String value in buffer: Hello  
Integer value in buffer: 20  
Total items read: 2

**sprintf()**

- In C, **sprintf()** is used to send formatted data. Instead of printing on console, sprintf() function stores the output on string(character array ) that is specified in sprintf.
- Syntax  
`sprintf(String_Name, “format specifier”,variable)`

```
/* sprintf example */
#include <stdio.h>
int main()
{
    char buffer[50];
    int a = 10, b = 20, c;
    c = a + b;
    sprintf(buffer, "Sum of %d and %d is %d", a, b, c);

    // The string "sum of 10 and 20 is 30" is stored
    // into buffer instead of printing on stdout
    printf("%s", buffer);
    return 0;
}
```

## Output

Sum of 10 and 20 is 30

```
/* sprintf example */
#include <stdio.h>
int main()
{
    float num = 9.34;
    printf("I'm a float, look: %f\n", num);
    char output[50]; //for storing the converted string
    sprintf(output, "%f", num);
    printf("Look, I'm now a string: %s", output);
```

}

Output:

I'm a float, look: 9.340000

Look, I'm now a string: 9.340000

```
/* sprintf example */
#include <stdio.h>
int main()
{
    char output[50];
    int num1 = 3, num2 = 5, ans;
    ans = num1 * num2;
    sprintf(output, "%d multiplied by %d is %d", num1, num2, ans);
    printf("%s", output);
    return 0;
}
```

Output:

3 multiplied by 5 is 15

# **sprintf returns the length of the converted string, as shown below:**

```
/* sprintf example */
#include <stdio.h>
int main()
{
    int num = 3003;
    int length;
    char output[50];//for storing the converted string
    length = sprintf(output, "%d", num);
    printf("The converted string is %s and its length is %d.", output, length);
}
```

**Output:**

The converted string is 3003 and its length is 4.

# **FUNCTION**

# Function

- Function is a part of program or set of statements which can be compiled standalone and perform a special task in our program.
- 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.

# Function vs Program

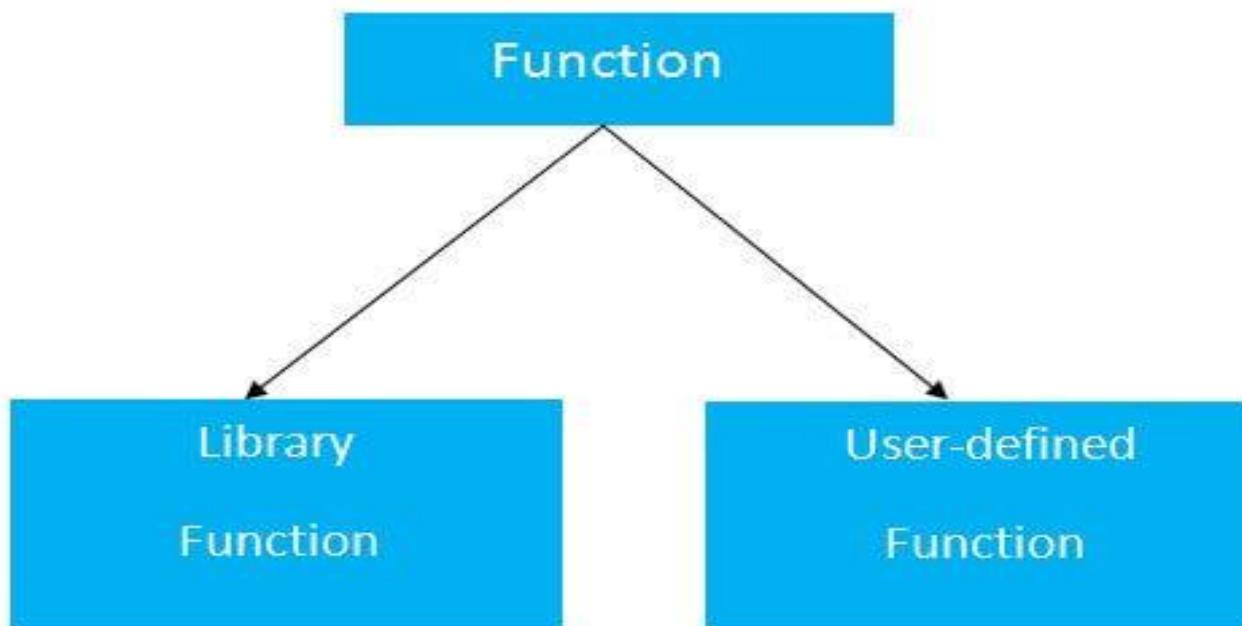
- A Program must have atleast one function.
- If there is only one function then it must be main().
- There is no limit of number of functions in a program.
- While control of program is handled by main function.
- Any function can call any function.
- Every function returns control to the main function after completing its task.

# 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.

# Types of Functions

- There are two types of functions in C programming:



- **Library Functions or Built-in Functions:** are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
- **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.

# Function Aspects

There are three aspects of a C function.

- **Function prototyping or 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 or body:** 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.

# Function Prototyping or Declaration

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts :

return\_type function\_name( parameter list );

Void

int

char

float

double

(All

data

types)

Valid identifier

i)Number of parameter  
ii)Type of parameter

# Function Call

a) If return type is void

function name(list of parameters)

example: add(a,b);

 Actual parameter

a) If return type is not void

variable name=function name(list of parameters)

example: r=add(a,b);

 Actual parameter

# Function Definition/body

```
return_type function_name (argument list)
{
    function body;
}
```

Example:

```
int add(int x,int y)
{
    int z;
    z=x+y;
    return (z);
}
```

 Formal parameter

# 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;
}
```

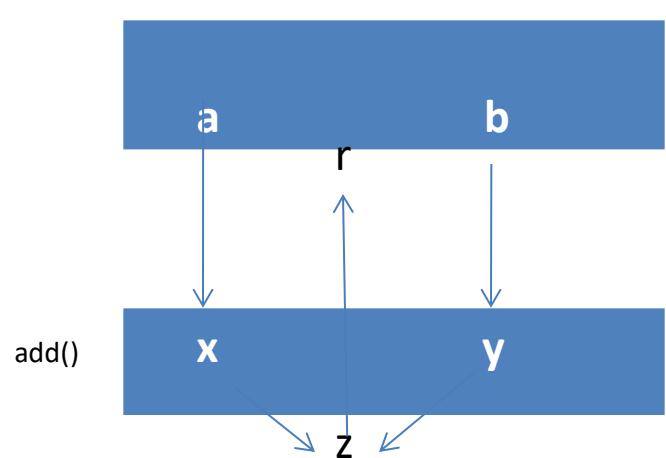
# Example of function

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



Called function

→ Calling function



# Output

Going to calculate the sum of two numbers:

Enter two numbers:10

20 The sum is : 30

## **Different aspects of function calling(classification of functions based on return type and parameters)**

- 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.
- 1)function without arguments and without return value
  - 2)function without arguments and with return value
  - 3)function with arguments and without return value
  - 4)function with arguments and with return value

**1)Function without arguments and  
without return value**

# **Example 1 for Function without argument and return value**

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

- **Output**  
Hello Javatpoint

# **Example 2 for Function without argument and return value**

```
//Program to add two given number

#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

## **2) Function without arguments and with return value**

# Example 1 for Function without argument and with return value

//Program to add two given number

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

- **Output**

Going to calculate the sum of two numbers: Enter two numbers 10  
24  
The sum is 34

# Example 2 for Function without argument and with return value

//Program to calculate area of square

```
#include<stdio.h>
int square();
void main()
{
    float area;
    printf("Going to calculate the area of the square\n");
    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);
    z= side * side;
    return z;
}
```

- **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
```

### **3) Function with arguments and without return value**

# **Example 1 for Function with argument and without return value**

## //Program to add two given number

```
#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 x, int y)
{
    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 for Function with argument and without return value**

## //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 3 for Function with argument and without return value**

//to print fibonacci series

```
#include<stdio.h>
#include<conio.h>
void fibonacci(int);
void main()
{
int number;
printf("Enter the number of elements:");
scanf("%d",&n);
fibonacci(n);
getch();
}
void fibonacci(int x)
{
int a=0,b=1,c,i;
printf("\n%d %d",a,b);//printing 0 and 1
for(i=1;i<=x-2;i++) //loop goes to x-2 because 0 and 1 are already printed
{
c=a+b;
printf(" %d",c);
a=b;
b=c;
}
}
```

#### OUTPUT:

Enter the number of elements:15  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

## **4)Function with arguments and with return value**

# **Example 1 for Function with argument and with return value**

## //Program to add two given number

```
#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 for Function with argument and with return value**

# //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

# **Example 3 for Function with argument and with return value**

**//to find Factorial of a number using recursion**

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

void main()
{
    int n;
    long f;
    printf("\nEnter a number: ");
    scanf("%d",&n);
    f = fact(n);
    printf("\nFactorial of %d is : %ld\n", n, f);
    return 0;
}
```

```
int fact(int x)
{
    if (x == 0)
        return 1;
    else
        return(x * fact(x-1));
}
```

Output :  
Enter a Number : 5  
The Factorial of 5 is : 120

# Program To Swap Two Numbers using Function

When we call a function and pass the actual value it's called as **Call by Value** method.

If we pass the reference or the address of the variable while calling the function, then it's called **Call by Reference**.

```

#include<stdio.h>

void swap(int, int);

int main()
{
    int a, b;

    printf("Enter values for a and b\n");
    scanf("%d%d", &a, &b);

    printf("\n\nBefore swapping: a = %d and b = %d\n", a, b);

    swap(a, b);

    printf("\n: a = %d and b = %d\n", a, b);
    return 0;
}

void swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;

    printf("\nAfter swapping: a = %d and b = %d\n", x, y);
}

```

## Call by Value method.

### **Output 1:**

Enter values for a and b

20

50

Before swapping: a = 20 and b = 50  
 : a = 20 and b = 50

After swapping: a = 50 and b = 20

### **Output 2:**

X=20 y=50

Temp= 20

X=50

Y=20

# Pointer as function argument in C

Pointer as a function parameter is used to hold addresses of arguments passed during function call.

This is also known as **call by reference**.

When a function is called by reference any change made to the reference variable will effect the original variable.

## call by reference method

```
#include <stdio.h>

void swap(int *x, int *y);

int main()
{
int a = 20,b = 50;
printf("a = %d\n", a);
printf("b = %d\n\n", b);
printf("\n\nBefore swapping: a = %d and b = %d\n", a, b);
swap(&a, &b); //passing address of m and n to the swap function
printf("\nAfter swapping: a = %d and b = %d\n", a,b);
return 0;
}

/* pointer 'a' and 'b' holds and points to the address of 'm' and 'n' */

void swap(int *x, int *y)
{
int temp;
temp = *x; //temp=20
*x = *y;// a=50
*y = temp;//b=20
printf("\nAfter swapping: a = %d and b = %d\n", a,b);
}
```

Output:

a = 20

b = 50

Before swapping: a = 20 and  
b = 50

After swapping: a = 50 and  
b = 20

After swapping: a = 50 and  
b = 20

**Example to print fibonacci series  
without function**

```
#include<stdio.h>
int main()
{
    int a=0,b=1,c,i,n;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    printf("\n%d %d ",a,b);//printing 0 and 1
    for(i=1;i<n-2;++i)// loop goes to n-2 because 0 and 1 are already printed
    {
        c=a+b;
        printf(" %d",c);
        a=b;
        b=c;
    }
    return 0;
}
```

#### OUTPUT:

Enter the number of elements:15  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

# **Example to print fibonacci series with function**

```
#include<stdio.h>
#include<conio.h>
void fibonacci(int);
void main()
{
int number;
printf("Enter the number of elements:");
scanf("%d",&n);
fibonacci(n);
getch();
}
void fibonacci(int x)
{
int a=0,b=1,c,i;
printf("\n%d %d",a,b);//printing 0 and 1
for(i=1;i<=x-2;i++) //loop goes to x-2 because 0 and 1 are already printed
{
c=a+b;
printf(" %d",c);
a=b;
b=c;
}
}
```

#### OUTPUT:

Enter the number of elements:15  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

**Example to find Factorial of a  
number without using function**

```
#include <stdio.h>
#include <conio.h>
void main ()
{
    int n;
    long f=1;
    clrscr();
    printf ("\n Enter a Number : ");
    scanf ("%d",&n);
    for (i=1;i<=n;i++)
    {
        f=f*i;
    }
    printf ("\n The Factorial of %d is : %ld",n,f);
    getch();
}
```

Output:

Enter a Number : 5

The Factorial of 5 is : 120

**Example to find Factorial of a  
number using function**

```
#include <stdio.h>
#include <conio.h>
long fact(int);
void main ()
{
    int n;
    long f;
    clrscr();
    printf ("\n Enter a Number : ");
    scanf ("%d",&n);
    f=fact(n);
    printf ("\n The Factorial of %d is : %ld",n,f);
    getch();
}
long fact(int x)
{
    int i;
    long f=1;
    for (i=1;i<=n;i++)
        f=f*i;
    return(f);
}
```

Output :  
Enter a Number : 5  
The Factorial of 5 is : 120