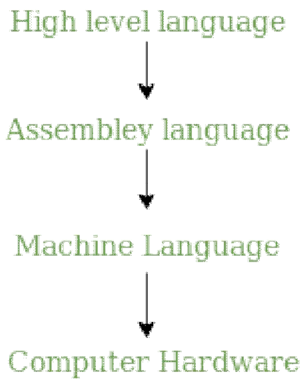
Introduction to Programming Languages

Hierarchy of Computer language –



There have been many programming language some of them are listed below:

C	**Python**	**C++**
C#	**R**	**Ruby**
COBOL	**ADA**	**Java**
Fortran	**BASIC**	**Altair BASIC**
True BASIC	**Visual BASIC**	**GW BASIC**
QBASIC	**PureBASIC**	**PASCAL**
Turbo Pascal	**GO**	**ALGOL**
LISP	**SCALA**	**Swift**
Rust	**Prolog**	**Reia**
Racket	**Scheme**	**Shimula**
Perl	**PHP**	**Java Script**
CoffeeScript	**VisualFoxPro**	**Babel**
Logo	**Lua**	**Smalltalk**
Matlab	**F**	**F#**
Dart	**Datalog**	**Dbase**
Haskell	**dylan**	**Julia**
Ksh	**metro**	**Mumps**
Nim	**OCaml**	**Pick**
TCL	**D**	**CPL**
Curry	**ActionScript**	**Erlang**
Clojure	**DarkBASCIC**	**Assembly**

Most Popular Programming Languages –

- C
- Python
- C++
- Java
- SCALA
- C#
- R
- Ruby
- Go
- Swift
- JavaScript

Characteristics of a programming Language –

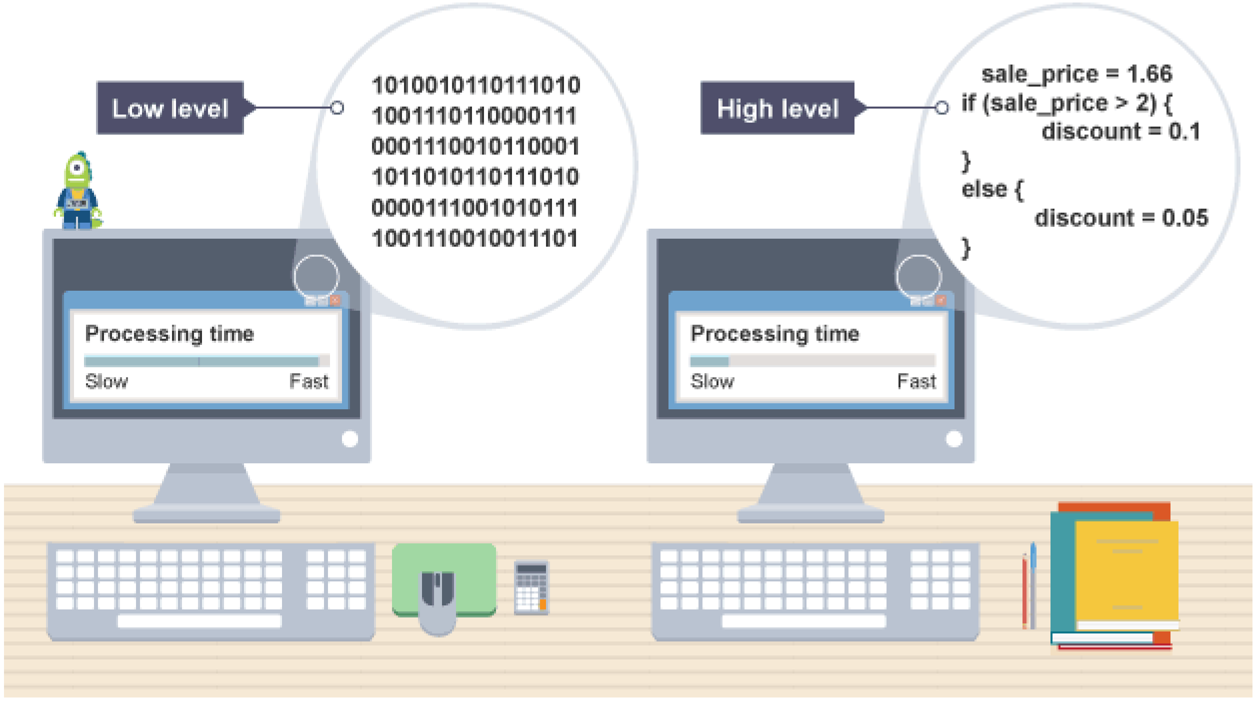
- A programming language must be simple, easy to learn and use, have good readability and human recognizable.
- A portable programming language is always preferred.
- Programming language's efficiency must be high so that it can be easily converted into a machine code and executed consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development.
- Necessary tools for development, debugging, testing, maintenance of a program must be provided by a programming language.
- A programming language should provide single environment known as Integrated Development Environment(IDE).
- A programming language must be consistent in terms of syntax.

A language is the main medium of communicating between the Computer systems and the most common are the programming languages. As we know a Computer only understands binary numbers that is 0 and 1 to perform various operations but the languages are developed for different types of work on a Computer. A language consists of all the instructions to make a request to the system for processing a task. From the first generation and now fourth generation of the Computers there were several programming languages used to communicate with the Computer.

TYPES OF COMPUTER LANGUAGES

There are mainly two types of computer languages :-

- Low Level Language
- High Level Language



High Level Languages

When we think about computer programmers, we are probably thinking about people who write in high-level programming languages.

High level languages are written in a form that is close to our human language, enabling to programmer to just focus on the problem being solved.

No particular knowledge of the hardware is needed as high level languages create programs that are portable and not tied to a particular computer or microchip.

These programmer friendly languages are called 'high level' as they are far removed from the machine code instructions understood by the computer.

Examples include: C++, Java, Pascal, Python, Visual Basic.

Advantages

- Easier to modify as it uses English like statements

- Easier/faster to write code as it uses English like statements
- Easier to debug during development due to English like statements
- Portable code – not designed to run on just one type of machine

---

Low Level Languages

Low level languages are used to write programs that relate to the specific architecture and hardware of a particular type of computer.

They are closer to the native language of a computer (binary), making them harder for programmers to understand.

Examples of low level language:

- Assembly Language
- Machine Code

Assembly Language

Few programmers write programs in low level assembly language, but it is still used for developing code for specialist hardware, such as device drivers.

It is easy distinguishable from a high level language as it contains few recognisable human words but plenty of mnemonic code.

Advantages

- Can make use of special hardware or special machine-dependent instructions (e.g. on the specific chip)
- Translated program requires less memory
- Write code that can be executed faster
- Total control over the code
- Can work directly on memory locations

Machine Code

Programmers rarely write in machine code (binary) as it is difficult to understand.

High-Level Languages	Low-Level Languages
High-Level Languages are easy to learn and understand.	Low-Level Languages are challenging to learn and understand.
They are executed slower than lower level	They execute with high speed.

languages because they require a translator program.	
They do not provide many facilities at the hardware level.	They are very close to the hardware and help to write a program at the hardware level.
For writing programs, hardware knowledge is not required.	For writing programs, hardware knowledge is a must.
The programs are easy to modify.	Modifying programs is difficult.
C++, Java, C#, Paython etc are examples of High-Level Languages.	Machine language and Assembly language are Low-Level Languages.

Introduction of Programming Concept

high-level language to work on the computer it must be translated into machine language. There are two kinds of translators – compilers and interpreters – and high-level languages are called either compiled languages or interpreted languages.

Compiler – a translation program that convert the programmer's entire high-level program, which is called the source code , into a machine language code, which is called the object code . This translation process is called COMPILATION.
.
Interpreter – a translation program that converts each program statement (line by line) into machine code just before the program statement is to be executed. Translation and execution occur immediately, one after another, one statement at a time.

Unlike the compiled languages, no object code is stored and there is no compilation. This means that in a program where one statement is executed several times, that statement is converted to machine language each time it is executed.

Compiled languages are better than interpreted languages as they can be executed faster and more efficiently once the object code has been obtained. On the other hand, interpreted languages
do not need to create object code and so are usually easier to develop – that is, to code and test

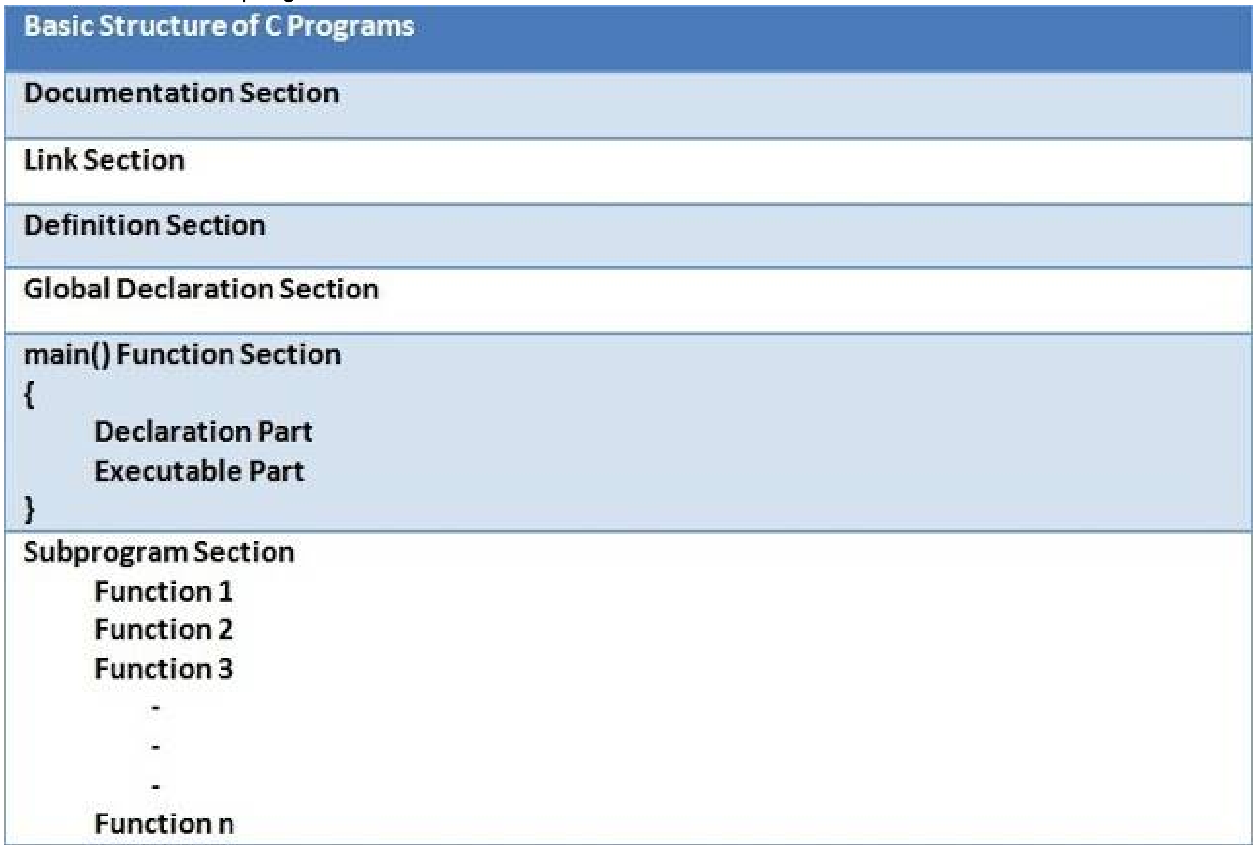
C Language Introduction

C is a procedural programming language. It was initially developed by Dennis Ritchie between 1969 and 1973. It was mainly developed as a system programming language to write operating system. The main features of C language include low-level access to memory, simple set of keywords, and clean style, these features make C language suitable for system programming like operating system or compiler development.
Many later languages have borrowed syntax/features directly or indirectly from C language. Like syntax of Java, PHP, JavaScript and many other languages is mainly based on C language. C++ is nearly a superset of C language (There are few programs that may compile in C, but not in C++).

Beginning with C programming:

1. **Structure of a C program**
   After the above discussion, we can formally assess the structure of a C program. By structure, it is meant that any program can be written in this structure only. Writing a C program in any other structure will hence lead to a Compilation Error.

The structure of a C program is as follows:



Basic Structure of C Programs

Documentation Section

Link Section

Definition Section

Global Declaration Section

```
main() Function Section
{
        Declaration Part
        Executable Part
}
```

```
Subprogram Section
        Function 1
        Function 2
        Function 3
                -
                -
                -
        Function n
```

The components of the above structure are:

1. **Header Files Inclusion**: The first and foremost component is the inclusion of the Header files in a C program.
   A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

   - **Some of C Header files**:

     1. stddef.h – Defines several useful types and macros.
     2. stdint.h – Defines exact width integer types.
     3. stdio.h – Defines core input and output functions
     4. stdlib.h – Defines numeric conversion functions, pseudo-random network generator, memory allocation
     5. string.h – Defines string handling functions
     6. math.h – Defines common mathematical functions
     7. conio.h – Defines console input output

   Syntax to include a header file in C:

   ```
   #include <(header_file_name).h>
   ```

2. **Main Method Declaration:** The next part of a C program is to declare the main() function. The syntax to declare the main function is:

   Syntax to Declare main method:

   ```
   void main()
   {

   }
   ```

3. **Variable Declaration:** The next part of any C program is the variable declaration. It refers to the variables that are to be used in the function. Please note that in C program, no variable can be used without being declared. Also in a C program, the variables are to be declared before any operation in the function.

   Example:

   ```
   void main()
   {
       int a;
   }
   ```

4. **Body:** Body of a function in C program, refers to the operations that are performed in the functions. It can be anything like manipulations, searching, sorting, printing, etc.

Example:

```
void main()
{
    int a=10;

    printf("%d", a);
}
```

- **Writing first program:**

Following is first program in C

```
#include <stdio.h>
void main()
{
    printf("the BCA");

}
```

C Character Set

As every language contains a set of characters used to construct words, statements etc., C language also has a set of characters which include **alphabets, digits** and **special symbols**. C language supports a total of 256 characters.

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters...

1. Alphabets
2. Digits
3. Special Symbols

Alphabets

C language supports all the alphabets from english language. Lower and upper case letters together supports 52 alphabets.

lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

Digits

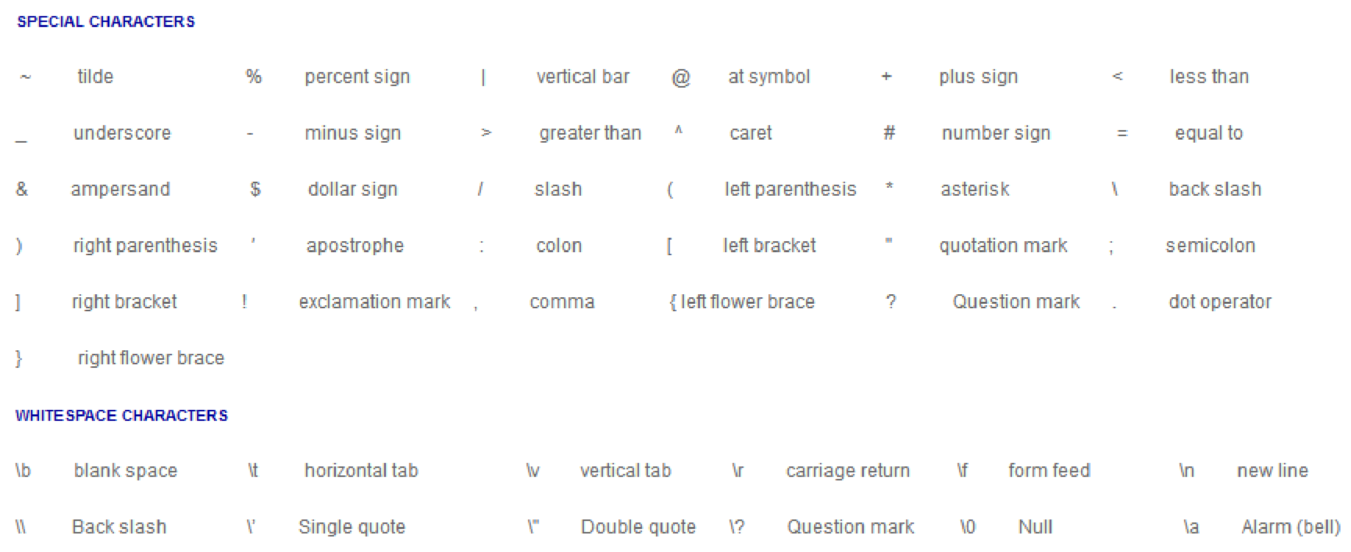
C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Special Symbols

C language supports rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, back spaces and other special symbols.

Special Symbols - **~ @ # $ % ^ & * ( ) _ - + = { } [ ] ; : ' " / ? . > , < \ | tab newline space NULL bell backspace verticaltab etc.,**



SPECIAL CHARACTERS

~	tilde	%	percent sign	|	vertical bar	@	at symbol	+	plus sign	<	less than
_	underscore	-	minus sign	>	greater than	^	caret	#	number sign	=	equal to
&	ampersand	$	dollar sign	/	slash	(	left parenthesis	*	asterisk	\	back slash
)	right parenthesis	'	apostrophe	:	colon	[	left bracket	"	quotation mark	;	semicolon
]	right bracket	!	exclamation mark	,	comma	{	left flower brace	?	Question mark	.	dot operator
}	right flower brace										

WHITESPACE CHARACTERS

\b	blank space	\t	horizontal tab	\v	vertical tab	\r	carriage return	\f	form feed	\n	new line
\\	Back slash	\'	Single quote	\"	Double quote	\?	Question mark	\0	Null	\a	Alarm (bell)

Execution Character Set

Certain ASCII characters are unprintable, which means they are not displayed on the screen or printer. Those characters perform other functions aside from displaying text. Examples are backspacing, moving to a newline, or ringing a bell.

They are used in output statements. Escape sequence usually consists of a backslash and a letter or a combination of digits. An escape sequence is considered as a single character but a valid character constant.

These are employed at the time of execution of the program. Execution characters set are always represented by a backslash (\) followed by a character. Note that each one of character constants represents one character, although they consist of two characters. These characters combinations are called as **escape sequence**.

Backslash character constants

Character	ASCII value	Escape Sequence	Result
Null	000	\0	Null
Alarm (bell)	007	\a	Beep Sound
Back space	008	\b	Moves previous position
Horizontal tab	009	\t	Moves next horizontal tab
New line	010	\n	Moves next Line
Vertical tab	011	\v	Moves next vertical tab
Form feed	012	\f	Moves initial position of next page
Carriage return	013	\r	Moves beginning of the line
Double quote	034	\"	Present Double quotes
Single quote	039	\'	Present Apostrophe
Question mark	063	\?	Present Question Mark
Back slash	092	\\	Present back slash
Octal number	\000		
Hexadecimal number	\x		

C tokens:

- Tokens are the smallest elements of a program, which are meaningful to the compiler.
- The following are the types of tokens: Keywords, Identifiers, Constant, Strings, Operators, etc.
- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual units in a C program are known as C tokens.

C tokens are of six types. They are,

1. Keywords	(eg: int, while),
2. Identifiers	(eg: main, total),
3. Constants	(eg: 10, 20),
4. Strings	(eg: “total”, “hello”),
5. Special symbols	(eg: (), {}),
6. Operators	(eg: +, /,-,*)

Let us begin with Keywords.

Keywords

Keywords are predefined, reserved words in C and each of which is associated with specific features. These words help us to use the functionality of C language. They have special meaning to the compilers.

There are total 32 keywords in C.

Auto	double	Int	Struct
Break	Else	Long	Switch
Case	Enum	register	Typedef
Char	Extern	Return	Union
continue	For	Signed	void
Do	If	Static	while
Default	Goto	Sizeof	volatile
Const	Float	Short	unsigned

Identifiers

Each program element in C programming is known as an identifier. They are used for naming of variables, functions, array etc. These are user-defined names which consist of alphabets, number, underscore '_'. Identifier's name should not be same or same as keywords. Keywords are not used as identifiers.

Rules for naming C identifiers −

- It must begin with alphabets or underscore.
- Only alphabets, numbers, underscore can be used, no other special characters, punctuations are allowed.
- It must not contain white-space.
- It should not be a keyword.
- It should be up to 31 characters long.

Strings

A string is an array of characters ended with a null character(\0). This null character indicates that string has ended. Strings are always enclosed with double quotes(" ").

Let us see how to declare String in C language −

- char string[20] = {'s','t','u','d','y', '\0'};
- char string[20] = "demo";
- char string [] = "demo";

Here is an example of tokens in C language,

Example

```
#include<stdio.h>
int main() {
   // using keyword char
   char a1 = 'H';
   int b = 8;
   float d = 5.6;
   // declaration of string
   char string[200] = "demodotcom";
   if(b<10)
      printf("Character Value : %c\n",a1);
   else
      printf("Float value : %f\n",d);
   printf("String Value : %s\n", string);
   return 0;
}
```

Output

```
Character Value : H
String Value : demodotcom
```

Hierarchy of operators , Operators Precedence in C.

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

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

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

Example

Try the following example to understand operator precedence in C −

```
#include <stdio.h>

int main() {

   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;

   e = (a + b) * c / d;       // ( 30 * 15 ) / 5
   printf("Value of (a + b) * c / d is : %d\n",  e );

   e = ((a + b) * c) / d;     // (30 * 15 ) / 5
   printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );

   e = (a + b) * (c / d);    // (30) * (15/5)
   printf("Value of (a + b) * (c / d) is  : %d\n",  e );

   e = a + (b * c) / d;      //  20 + (150/5)
   printf("Value of a + (b * c) / d is  : %d\n" ,  e );

   return 0;
}
```

When you compile and execute the above program, it produces the following result −

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50
```

C - Type Casting

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the **cast operator** as follows −

```
(type_name) expression
```

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation −

```
#include <stdio.h>

void main() {

   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character with an integer −

```
#include <stdio.h>

void main() {

   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   int sum;

   sum = i + c;
   printf("Value of sum : %d\n", sum );
}
```

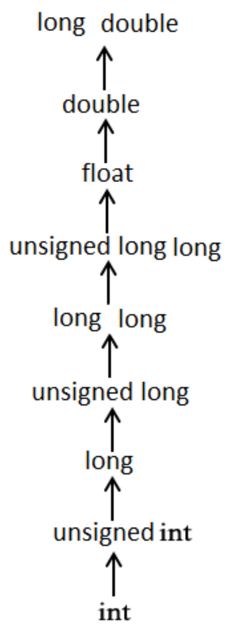
When the above code is compiled and executed, it produces the following result −

```
Value of sum : 116
```

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

Usual Arithmetic Conversion

The **usual arithmetic conversions** are implicitly performed to cast their values to a common type. The compiler first performs *integer promotion*; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy −



The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take the following example to understand the concept −

```
#include <stdio.h>

void main() {

   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   float sum;

   sum = i + c;
   printf("Value of sum : %f\n", sum );
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of sum : 116.000000
```

Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

C - Data Types

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see the basic types in the following section, where as other types will be covered in the upcoming chapters.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges −

Type	Storage size	Value range
Char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
Int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
Short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
Long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions *sizeof(type)* yields the storage size of the object or type in bytes. Given below is an example to get the size of various type on a machine using different constant defined in limits.h header file −

Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision −

Type	Storage size	Value range	Precision
Float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values −

The void Type

The void type specifies that no value is available. It is used in three kinds of situations −

Sr.No.	Types & Description
1	**Function returns as void** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

PRE-PROCESSORS IN C

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else** The alternative for #if.
8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error**

	Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

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

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

C program involves the following sections:

- Documentations (Documentation Section)
- Preprocessor Statements (Link Section)
- Global Declarations (Definition Section)
- The main() function
    - Local Declarations

  - Program Statements & Expressions
- User Defined Functions

Let's begin with a simple C program code.

Sample Code of C "Hello World" Program

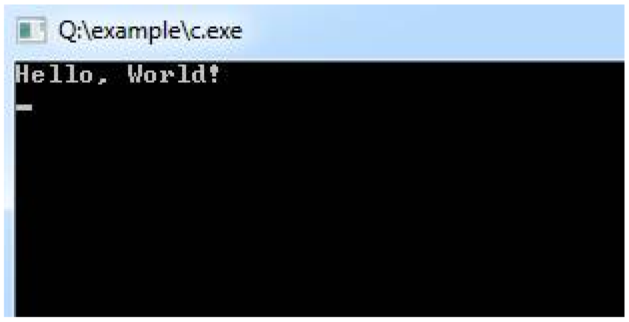
Example:

```
/* Author: www.w3schools.in
Date: 2018-04-28
Description:
Writes the words "Hello, World!" on the screen */
#include<stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

or in a different way

```
/* Author: www.w3schools.in
Date: 2013-11-15
Description:
Writes the words "Hello, World!" on the screen */
#include<stdio.h>
#include<conio.h>

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

Program Output:



```
Hello, World!
_
```

The above example has been used to print *Hello, World!* Text on the screen.

Documentation	Consists of comments, some description of the program, programmer name and any other useful points that can be referenced later.
Link	Provides instruction to the compiler to link function from the library function.
Definition	Consists of symbolic constants.
Global declaration	Consists of function declaration and global variables.

main( )
{
 	
}
	Every C program must have a `main()` function which is the starting point of the program execution.

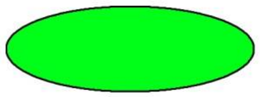
Subprograms	User defined functions.

An introduction to Flowcharts

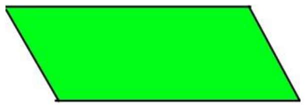
What is a Flowchart?
Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.
The process of drawing a flowchart for an algorithm is known as “flowcharting”.

Basic Symbols used in Flowchart Designs

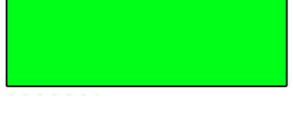
1. **Terminal:** The oval symbol indicates Start, Stop and Halt in a program’s logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



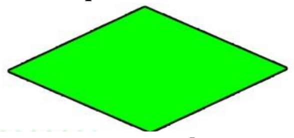
2. **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



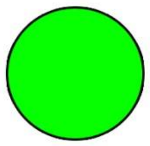
3. **Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.



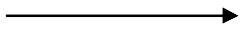
4. **Decision** Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



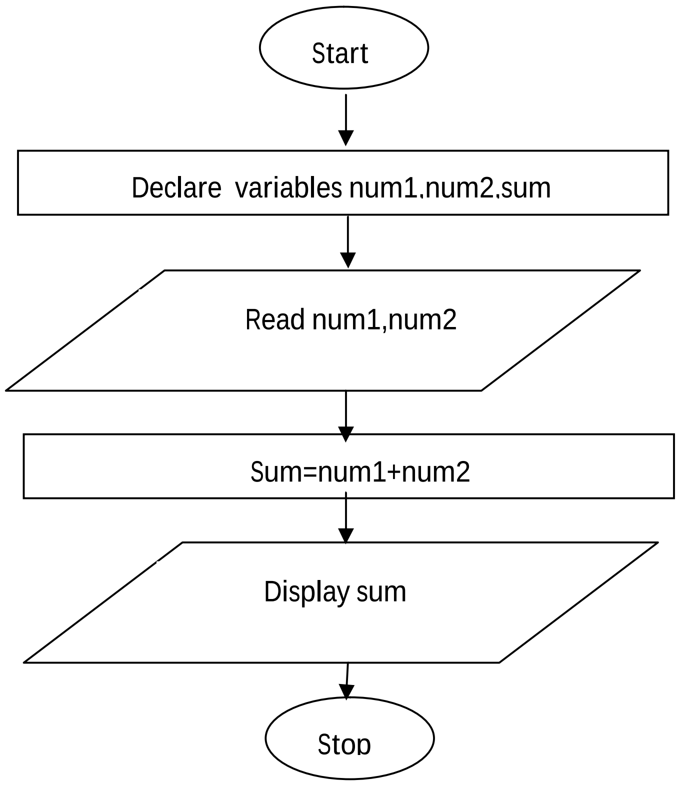
5. **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.



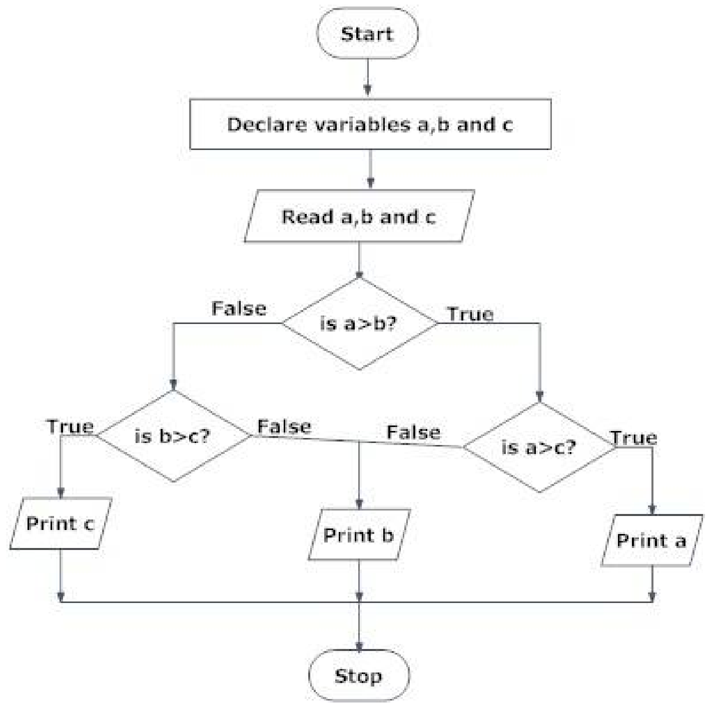
6. **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



Example : Draw a flowchart to input two numbers from user and display the sum of two numbers



Example : Draw a flowchart to input three numbers from user and display largest from those numbers



```
// C program to find largest of two numbers

#include <stdio.h>

void main()
{
    int num1, num2, largest;

    /*Input two numbers*/
    printf("Enter two numbers:\n");
    scanf("%d%d", &num1, &num2);

    /*check if a is greater than b*/
    if (num1 > num2)
        largest = num1;
    else
        largest = num2;

    /*Print the largest number*/
    printf("large number is =%d", largest);

    getch();
}
```

Output

```
Enter two numbers:
10 30
large number is =30
```

Format Specifier	Description
%d	Integer Format Specifier
%f	Float Format Specifier
%c	Character Format Specifier
%s	String Format Specifier
%u	Unsigned Integer Format Specifier
%ld	Long Int Format Specifier
%lf	Double

```
#include <stdio.h>
Void main()
{
   char ch = 'A';
   char str[20] = "googleindia.com";
   float flt = 10.234;
   int no = 150;
   double dbl = 20.123456;
   printf("Character is %c \n", ch);
   printf("String is %s \n" , str);
   printf("Float value is %f  \n", flt);     // printf("Float value is %.2f  \n", flt);
   printf("Integer value is %d \n" , no);
   printf("Double value is %lf \n", dbl);
   printf("Octal value is %o \n", no);
   printf("Hexadecimal value is %x \n", no);
   getch();
}
```

Output:

```
Character is A
String is googleindia.com
Float value is 10.234000       //10.23
Integer value is 150
Double value is 20.123456
```

```
Octal value is 226
Hexadecimal value is 96
```

Example 1: #define preprocessor

```c
#include <stdio.h>
#define PI 3.1415

int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%d", &radius);
    // Notice, the use of PI
    area = PI*radius*radius;
    printf("Area=%.2f",area);
    return 0;
}
```

Example 2: Using #define preprocessor

```c
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)

int main()
{
    int radius;
    float area;

    printf("Enter the radius: ");
    scanf("%d", &radius);
    area = circleArea(radius);
    printf("Area = %.2f", area);

    return 0;
}
```