Ch.1 Introduction To Computer Programming:

Computer is an electronic device that can be instructed (assigned tasks) to perform any logical or arithmetic tasks via computer programming.

Modern Computer systems have the ability to perform sophisticated or complex tasks by following some generalized sets of operations called **Programs**.

More sophisticated (complex) machines did specialized analog calculations in the early 20th Century.

The First ever digital electronic calculating machine were developed during World War 2.

Charles Babbage is considered as the Father of Computer.

The Evolution of Computer is divided into generations.

- **First Generation: -**
    - The First Generation of computers was considered between the year 1937 – 1946.
    - First electronic digital computer was built by John V. Atanasoff and Clifford Berry. Also known as **ABC.**
    - Then, Electronic Numerical Integrator and Computer (ENIAC) was built which was like 30 tons and it had up to 18000 Vacuum Tubes.
    - These Vacuum Tubes were used for processing.
    - The First-Generation computers were not having any operating system and it could only perform one task at a time.

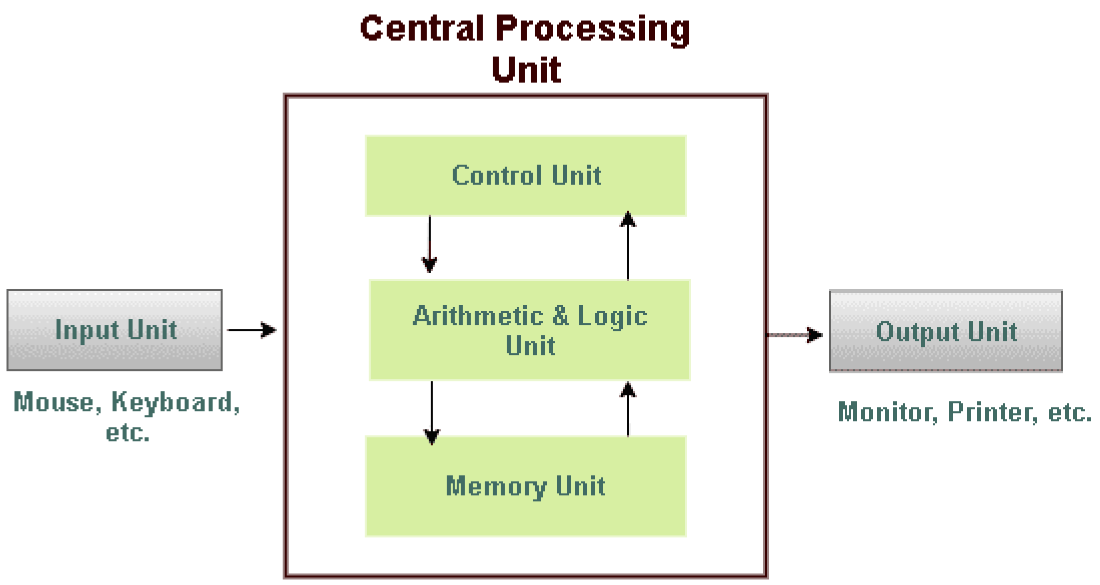
- **Second Generation: -**
    - The First computer system in second generation was introduced in 1951 for commercial purposes.
    - These computer generation was dominated by **International Business Machine (IBM).**
    - The computers of this generation were having the operating system and memory.
    - Many programming languages were developed and used in this time.

- **Third Generation: -**
    - The **Integrated Circuit (IC)** was developed in this generation which is started from 1963 – present.
    - Due to this, the computer systems became dominant and more trustworthy.
    - The computer system in this generation is faster, can perform multiple tasks at a same time.

Basic Block Diagram of Computer System

A Computer system performs some basic operations

- It accepts the data as a way of input.
- It stores the data.
- It processes the data as required by the user.
- It gives response (output) to the user for his input.
- It controls all the operations inside a computer.



Input Unit

All the data received by the computer goes through the input unit. The input unit comprises different devices like a mouse, keyboard, scanner, etc. In other words, each of these devices acts as a mediator between the users and the computer.

The data that is to be processed is put through the input unit. The computer accepts the raw data in binary form. It then processes the data and produces the desired output.

- Take the data to be processed by the user.
- Convert the given data into machine-readable form.
- And then, transmit the converted data into the main memory of the computer.

The sole purpose is to connect the user and the computer.

CPU – Central Processing Unit

Central Processing Unit or the CPU, is the brain of the computer. It works the same way a human brain works. As the brain controls all human activities, similarly the CPU controls all the tasks.

Moreover, the CPU conducts all the arithmetical and logical operations in the computer.

Now the CPU comprises of two units, namely – ALU (Arithmetic Logic Unit) and CU (Control Unit). Both of these units work in sync. The CPU processes the data as a whole.

ALU – Arithmetic Logic Unit

The Arithmetic Logic Unit is made of two terms, arithmetic and logic. There are two primary functions that this unit performs.

1. Data is inserted through the input unit into the primary memory. Performs the basic arithmetical operation on it. Like addition, subtraction, multiplication, and division. It performs all sorts of calculations required on the data. Then sends back data to the storage.
2. The unit is also responsible for performing logical operations like AND, OR, Equal to, Less than, etc. In addition to this it conducts merging, sorting, and selection of the given data.

CU – Control Unit

The control unit as the name suggests is the controller of all the activities/tasks and operations. All this is performed inside the computer.

The memory unit sends a set of instructions to the control unit. Then the control unit in turn converts those instructions. After that these instructions are converted to control signals.

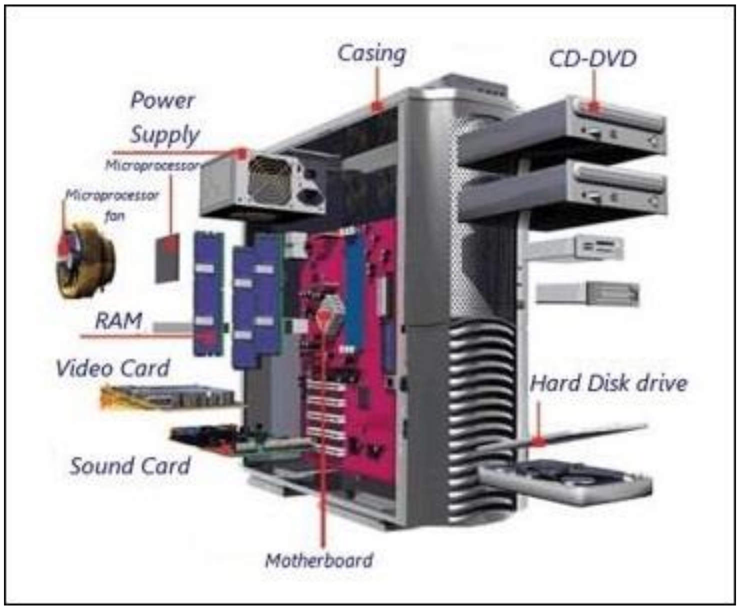
These control signals help in prioritizing and scheduling activities. Thus, the control unit coordinates the tasks inside the computer in sync with the input and output units.

Concept of Hardware

The term hardware refers to mechanical device that makes up computer.

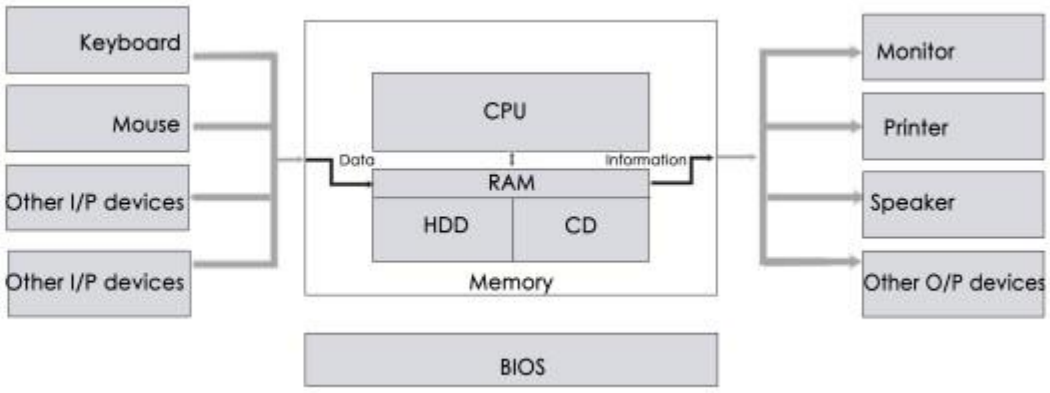
Computer hardware consists of interconnected electronic devices that we can use to control computer's operation, input and output.

Examples of hardware are CPU, keyboard, mouse, hard disk, etc.



Hardware Components

Computer hardware is a collection of several components working together. Some parts are essential and others are added advantages.



Concept of Software

A set of instructions that drives computer to do stipulated tasks is called a program. Software instructions are programmed in a computer language, translated into machine language, and executed by computer. Software can be categorized into two types −

- System software
- Application software

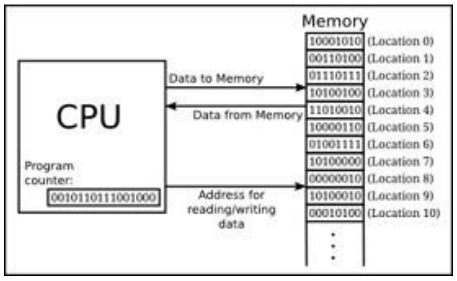
System Software

System software operates directly on hardware devices of computer. It provides a platform to run an application. It provides and supports user functionality. Examples of system software include operating systems such as Windows, Linux, Unix, etc.

Application Software

An application software is designed for benefit of users to perform one or more tasks. Examples of application software include Microsoft Word, Excel, PowerPoint, Oracle, etc.

Differences between Software and Hardware

Sr.No.	Software	Hardware
1	It is a collection of programs to bring computer hardware system into operation.	It includes physical components of computer system.
2	It includes numbers, alphabets, alphanumeric symbols, identifiers, keywords, etc.	It consists of electronic components like ICs, diodes, registers, crystals, boards, insulators, etc.
3	Software products evolve by adding new features to existing programs to support hardware.	Hardware design is based on architectural decisions to make it work over a range of environmental conditions and time.
4	It will vary as per computer and its built-in functions and programming language.	It is mostly constructed for all types of computer systems.
5	It is designed and developed by experienced programmers in high-level language.	The hardware can understand only low-level language or machine language.
6	It is represented in any high-level language such as BASIC, COBOL, C, C++, JAVA, etc.	The  Memory CPU Program counter: 0010110111001000 Data to Memory Data from Memory Address for reading/writing data 10001010 (Location 0) 00110100 (Location 1) 01110111 (Location 2) 10100100 (Location 3) 11010010 (Location 4) 10000110 (Location 5) 01001111 (Location 6) 10100000 (Location 7) 00000010 (Location 8) 10100010 (Location 9) 00010100 (Location 10) hardware works only on binary codes 1's and 0's.
7	The software is categorized as operating system, utilities, language processor, application software, etc.	The hardware consists of input devices, output devices, memory, etc.

Concept of Machine Level Language

Machine language is a low-level programming language made out of binary numbers or bits that can only be read by machines. It is also known as machine code or object code, in which instructions are executed directly by the CPU.

The only language that the computer understands is machine language. All programmes and programming languages, such as Swift and C++, produce or run programmes in machine language before they are run on a computer.

When a specific task, even the smallest process executes, machine language is transported to the system processor. Computers are only able to understand binary data as they are digital devices.

Computer programs are created in one or more programming languages (for example, Java, C++, or Visual Basic). The program code needs to be compiled through which the computer can understand it, as programming languages used to create computer programs cannot be understand by computer directly.

When the program' s code is compiled, it is converted into machine language, so that, the computer can understand it.

Uses of Machine Language

Common uses of machine language are discussed below:

- Machine language is a low-level language that machines understand but that humans can decipher using an assembler.
- A compiler plays an important role between humans and computers as it converts machine language into other code or language that is understandable by humans.
- Assembly language is dedicated to comprehending machine language since it is a rip-off of it.

Advantages & Disadvantages of Machine Level Language

Advantages		Disadvantages
Machine language makes fast and efficient use of the computer.		All operation codes have to be remembered
It requires no translator to translate the code. It is directly understood by the computer.		All memory addresses have to be remembered.
		It is hard to amend or find errors in a program written in the machine language.

Concept Of Assembly Language

An assembly language is a type of low-level programming language that is intended to communicate directly with a computer's hardware. Unlike machine language, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans.

An assembly language allows a software developer to code using words and expressions that can be easier to understand and interpret than the binary or hexadecimal data the computer stores and reads

Assembly languages often serve as intermediaries, allowing for developing more complex programming languages, which can offer further efficiency to a developer.

Assembly languages differ between hardware architectures. A computer's architecture includes its machine components, hardware design, processor and the relationships it has with other machines.

An assembler is a program that translates commands into machine code. The assembler gathers the instructions from the assembly language and translates each action into a series of electrical signals the machine can interpret.

Although the assembly languages are specific to their hardware, they typically run various operating systems, meaning an assembly language can often be compatible with any programming language.

Altogether, this assembly language looks like this:

```
1: MOV eax, 3
MOV ebx, 4
ADD eax, ebx, ecx
```

- **Where**, "1:" is the label which lets the computer know where to begin the operation,
- The "MOV" is the mnemonic command to move the number "3" into a part of the computer processor, which can function as a variable.
- "EAX," "EBX" and "ECX" are the variables. The first line of code loads "3" into the register "eax."
- The second line of code loads "4" into the register "ebx." Finally, the last line of code adds "eax" and "ebx" and stores the result of the addition, which is seven, in "ecx."

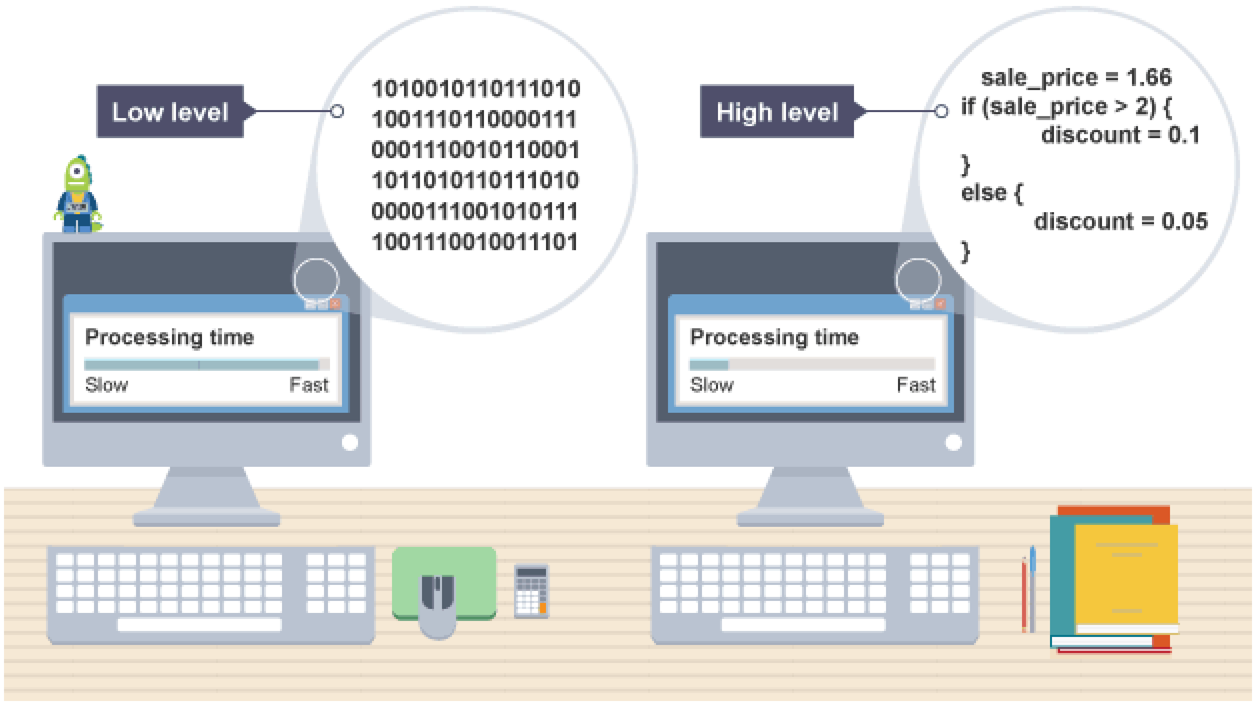
Advantages and Disadvantages of Assembly Language

Advantages	Disadvantages
Speed and control	Difficulty of learning
Low-level access	Lack of portability
Smaller code size	Time-consuming
Greater security	Error-prone
Access to specialized hardware	Limited functionality

Concept Of High-Level Language

High-level programming languages mean that languages of writing computer instructions in a way that is easily understandable and close to human language. High-level languages are created by developers so that programmers don't need to know highly difficult low level/machine language.

Programmers can easily learn high-level languages as it is very close to human language. Whereas, in **low level programming languages**, programming is done which is associated with the machine (or we can say Hardware).



High-level-Programming Languages are associated with human to understand, the way it is in syntax or style of code is easily understandable.

And the other case, low-level-programming languages are complex to learn because it is associated with machine language which every instruction, we pass will be in binary format like 0 or 1 (Hardware).

Difference between Low, Assembly and High-level Language

Low-level language	Assembly language
The machine-level language comes at the lowest level in the hierarchy, so it has zero abstraction level from the hardware.	The assembly language comes above the machine language means that it has less abstraction level from the hardware.
It cannot be easily understood by humans.	It is easy to read, write, and maintain.
The machine-level language is written in binary digits, i.e., 0 and 1.	The assembly language is written in simple English language, so it is easily understandable by the users.
It does not require any translator as the machine code is directly executed by the computer.	In assembly language, the assembler is used to convert the assembly code into machine code.
It is a first-generation programming language.	It is a second-generation programming language.

Low-level language	High-level language
It is a machine-friendly language, i.e., the computer understands the machine language, which is represented in 0 or 1.	It is a user-friendly language as this language is written in simple English words, which can be easily understood by humans.
The low-level language takes more time to execute.	It executes at a faster pace.
It requires the assembler to convert the assembly code into machine code.	It requires the compiler to convert the high-level language instructions into machine code.
The machine code cannot run on all machines, so it is not a portable language.	The high-level code can run all the platforms, so it is a portable language.
It is memory efficient.	It is less memory efficient.
Debugging and maintenance are not easier in a low-level language.	Debugging and maintenance are easier in a high-level language.

Compiler And Interpreter

Compiler

A compiler is a special program that translates a programming language's source code into machine code.

We generally write a computer program using a high-level language. A high-level language is one that is understandable by us, humans. This is called **source code**.

However, a computer does not understand high-level language. It only understands the program written in **0**'s and **1**'s in binary, called the **machine code**.

To convert source code into machine code, we use either a **compiler** or an **interpreter**.

Both compilers and interpreters are used to convert a program written in a high-level language into machine code understood by computers. Yet, both do have some differences too.

Interpreter Vs Compiler

INTERPRETER	COMPILER
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No object code is generated, hence are memory efficient.	Generates Object Code which further requires linking, hence requires more memory.
Programming languages like JavaScript, python, ruby use interpreters.	Programming languages like C, C++, Java use compilers.

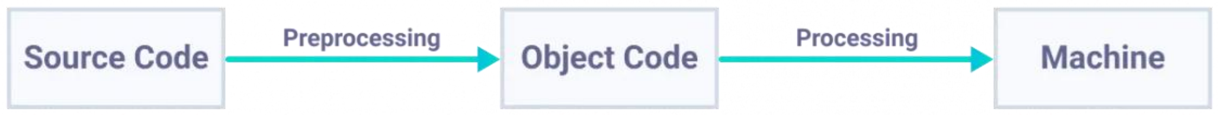


Figure: Compiler

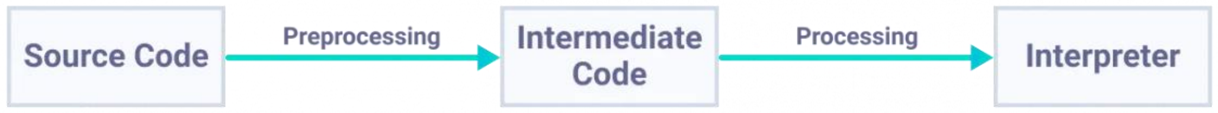


Figure: Interpreter

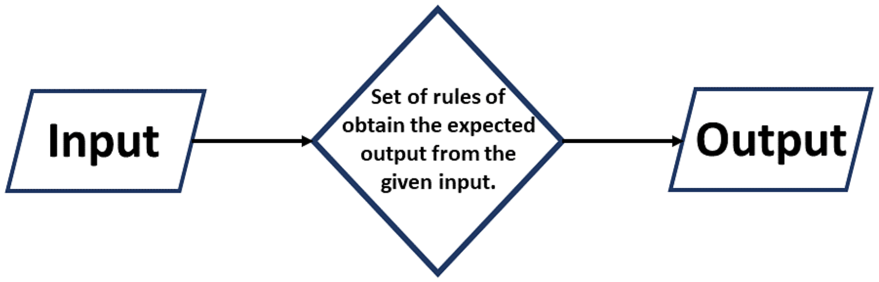
Fig. Compiler And Interpreter

What is Algorithm?

An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.

According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task.

It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudocode.


Input
Set of rules of obtain the expected output from the given input.
Output

Algorithm

- **Problem**: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.
- **Algorithm**: An algorithm is defined as a step-by-step process that will be designed for a problem.
- **Input**: After designing an algorithm, the algorithm is given the necessary and desired inputs.
- **Processing unit**: The input will be passed to the processing unit, producing the desired output.
- **Output**: The outcome or result of the program is referred to as the output.

A finite set of steps that must be followed to solve any problem is called an **algorithm.**

For Example,

Algorithm to add two numbers:

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
    sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Algorithm 2: Find the largest number among three numbers

```
Step 1: Start
Step 2: Declare variables a, b and c.
Step 3: Read variables a, b and c.
Step 4: If a > b
          If a > c
            Display a is the largest number.
          Else
            Display c is the largest number.
        Else
          If b > c
            Display b is the largest number.
          Else
            Display c is the greatest number.
Step 5: Stop
```

Advantages of algorithm

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program

Disadvantages of algorithm.

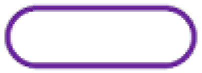
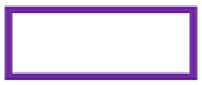
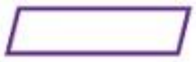
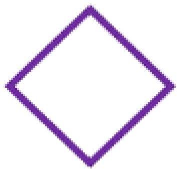
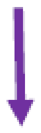
1. Writing algorithm takes a long time.
2. An Algorithm is not a computer program, it is rather a concept of how a program should be.

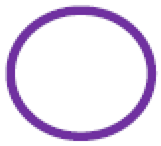
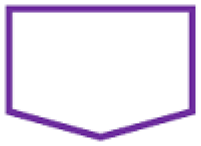
Flowchart

Flowchart is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart Symbols

Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.

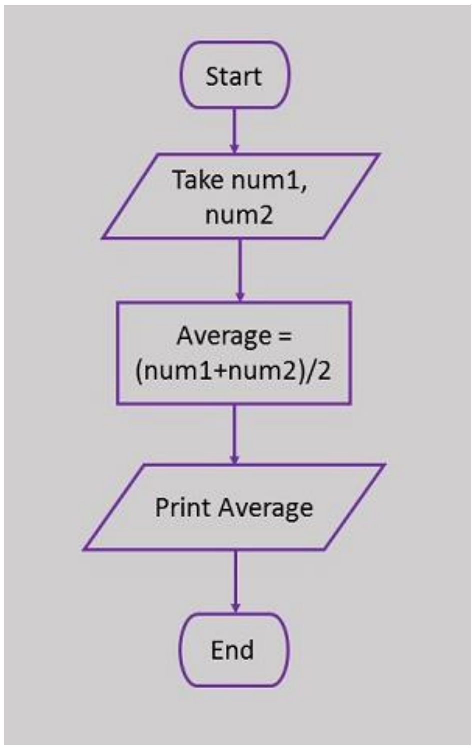
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Guidelines for Developing Flowcharts

These are some points to keep in mind while developing a flowchart −

- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers
- Off-page connectors are referenced using alphabets
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

Example of Flowchart: For calculating average of 2 numbers.



Advantages of Flowchart:

1. The Flowchart is an excellent way of communicating the logic of a program.
2. It is easy and efficient to analyze problem using flowchart.
3. During program development cycle, the flowchart plays the role of a guide or a blueprint. Which makes program development process easier.
4. After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
5. It helps the programmer to write the program code.
6. It is easy to convert the flowchart into any programming language code as it does not use any specific programming language concept.

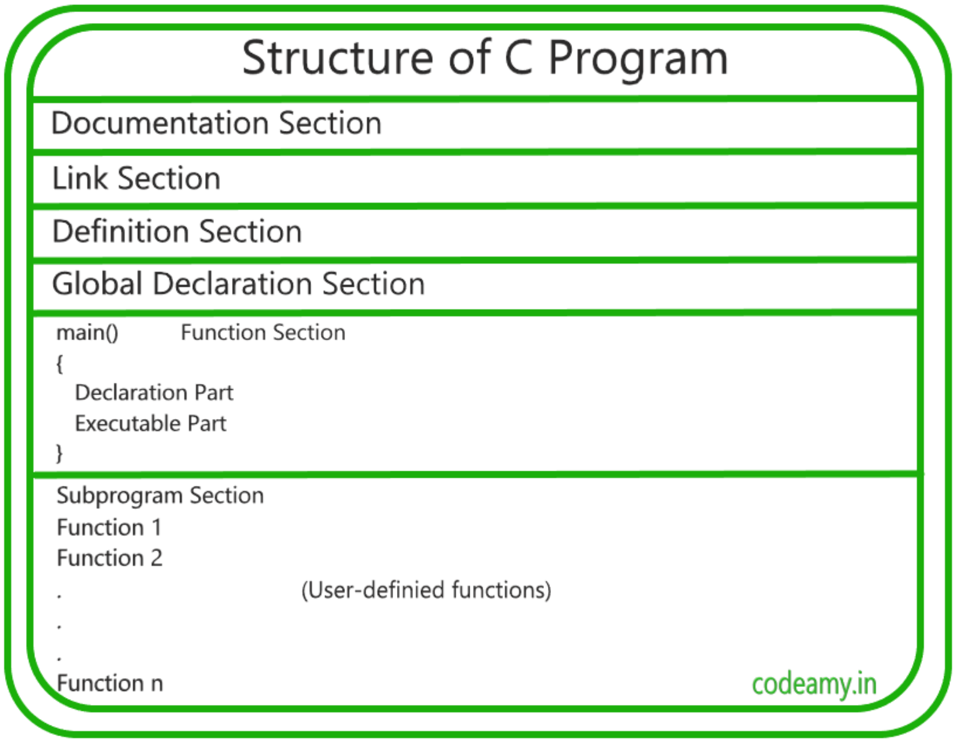
Disadvantage of flowchart

1. The flowchart can be complex when the logic of a program is quite complicated.
2. Drawing flowchart is a time-consuming task.
3. Difficult to alter the flowchart. Sometimes, the designer needs to redraw the complete flowchart to change the logic of the flowchart or to alter the flowchart.
4. Since it uses special sets of symbols for every action, it is quite a tedious task to develop a flowchart as it requires special tools to draw the necessary symbols.
5. In the case of a complex flowchart, other programmers might have a difficult time understanding the logic and process of the flowchart.
6. It is just a visualization of a program; it cannot function like an actual program.

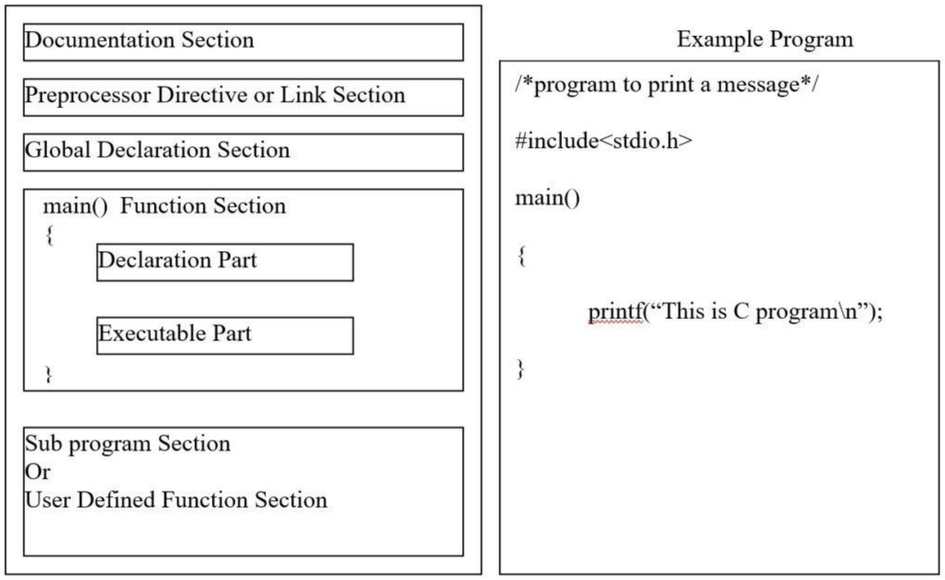
Structure of C Program

The structure of a C program can be divided into six sections, namely - Documentation, Link, Definition, Global Declaration, main() Function, and Subprograms.

The main() function is compulsory to include in every C program, whereas the rest are optional.



Example:



```
/*program to print a message*/

#include<stdio.h>

main()

{

        printf("This is C program\n");

}
```

Data Types and Size

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.

Data types in C are classified into 2 categories.

1. Primitive Data Type
2. Derived Data Type

Primitive Data Type:

Primitive Data Types are those data types which are provided by the language. These data types are also known as the in-built data types. It includes,

1. Int
2. Char
3. Float
4. Double

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals. The memory size of the basic data types may change according to 32 or 64-bit operating system.

Data Types	**Memory Size**	**Range**
char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
short	2 bytes	−32,768 to 32,767
signed short	2 bytes	−32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
int	2 bytes	−32,768 to 32,767

signed int	2 bytes	−32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
short int	2 bytes	−32,768 to 32,767
signed short int	2 bytes	−32,768 to 32,767
unsigned short int	2 bytes	0 to 65,535
long int	4 bytes	-2,147,483,648 to 2,147,483,647
signed long int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long int	4 bytes	0 to 4,294,967,295
float	4 bytes	
double	8 bytes	
long double	10 bytes	

Constants and Variables

Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

int player1 = 95;

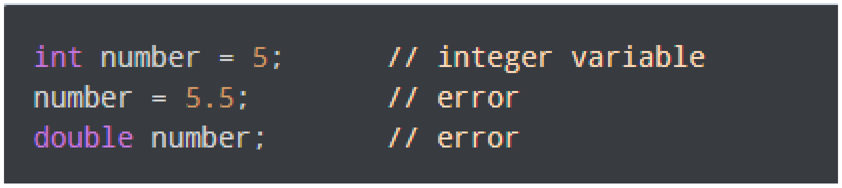
Here, player1 is a variable of int type. Here, the variable is assigned an integer value 95.

The value of a variable can be changed, hence the name variable.

Rules for naming a variable

1. A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore.
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

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



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

Here, the type of number variable is int.

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

By the way, to store the decimal values in C, you need to declare its type to either double or float.

Constants

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

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

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

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

Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C programming.
For example: newline(enter), tab, question mark etc.
In order to use these characters, escape sequences are used.

Escape Sequences

Escape Sequences	Character
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\0`	Null character

Operators in C

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions.

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

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

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language.

Assume variable **A** holds 10 and variable **B** holds 20.

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Relational Operators

The following table shows all the relational operators supported by C.

Assume variable A holds 10 and variable B holds 20.

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0.

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows:

p	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13,

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

The following table lists the assignment operators supported by the C language.

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2

|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc Operators ↦ sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

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

Type Conversion

In C programming, we can convert the value of one data type (int, float, double, etc.) to another. This process is known as **type conversion**. For example,

```c
#include <stdio.h>

int main() {

  int number = 34.78;
  printf("%d", number);

  return 0;
}

// Output: 34
```

Here, we are assigning the double value **34.78** to the integer variable number. In this case, the double value is automatically converted to integer value **34**.

This type of conversion is known as **implicit type conversion**.

In C, there are two types of type conversion:

1. **Implicit Conversion**
2. **Explicit Conversion**

Implicit Type Conversion In C

As mentioned earlier, in implicit type conversion, the value of one type is automatically converted to the value of another type. For example,

```c
#include<stdio.h>

int main() {

  // create a double variable
  double value = 4150.12;
  printf("Double Value: %.2lf\n", value);

  // convert double value to integer
  int number = value;
  printf("Integer Value: %d", number);

  return 0;
}
```

Output:

```
Double Value: 4150.12
Integer Value: 4150
```

The above example has a double variable with a value **4150.12**. Notice that we have assigned the double value to an integer variable.

Here, the C compiler automatically converts the double value **4150.12** to integer value **4150**.

Since the conversion is happening automatically, this type of conversion is called **Implicit type conversion.**

Example: Implicit Type Conversion

```c
#include<stdio.h>

int main() {

  // character variable
  char alphabet = 'a';
  printf("Character Value: %c\n", alphabet);

  // assign character value to integer variable
  int number = alphabet;
  printf("Integer Value: %d", number);

  return 0;
}
```

Output

```
Character Value: a
Integer Value: 97
```

Here, the C compiler automatically converts the character 'a' to integer 97. This is because, in C programming, characters are internally stored as integer values known as **ASCII Values**.

Note: ASCII (American Standard Code for Information Interchange) is the most common character encoding format for text data in computers and on the internet.

In standard ASCII-encoded data, there are unique values for 128 alphabetic, numeric or special additional characters and control codes.

ASCII code for,
A-Z is in the range between **65 to 90,**
a-z is in the range between **97 to 122**

Explicit Type Conversion In C

In explicit type conversion, we manually convert values of one data type to another type. For example,

```c
#include<stdio.h>

int main() {

  // create an integer variable
  int number = 35;
  printf("Integer Value: %d\n", number);

  // explicit type conversion
  double value = (double) number;
  printf("Double Value: %.2lf", value);

  return 0;
}
```

Output:

```
Integer Value: 35
Double Value: 35.00
```

We have created an integer variable named number with the value 35 in the above program. Notice the code,

```
// explicit type conversion
double value = (double) number;
```

Here,

- (double) - represents the data type to which number is to be converted
- number - value that is to be converted to double type.

Example of Explicit Type Conversion

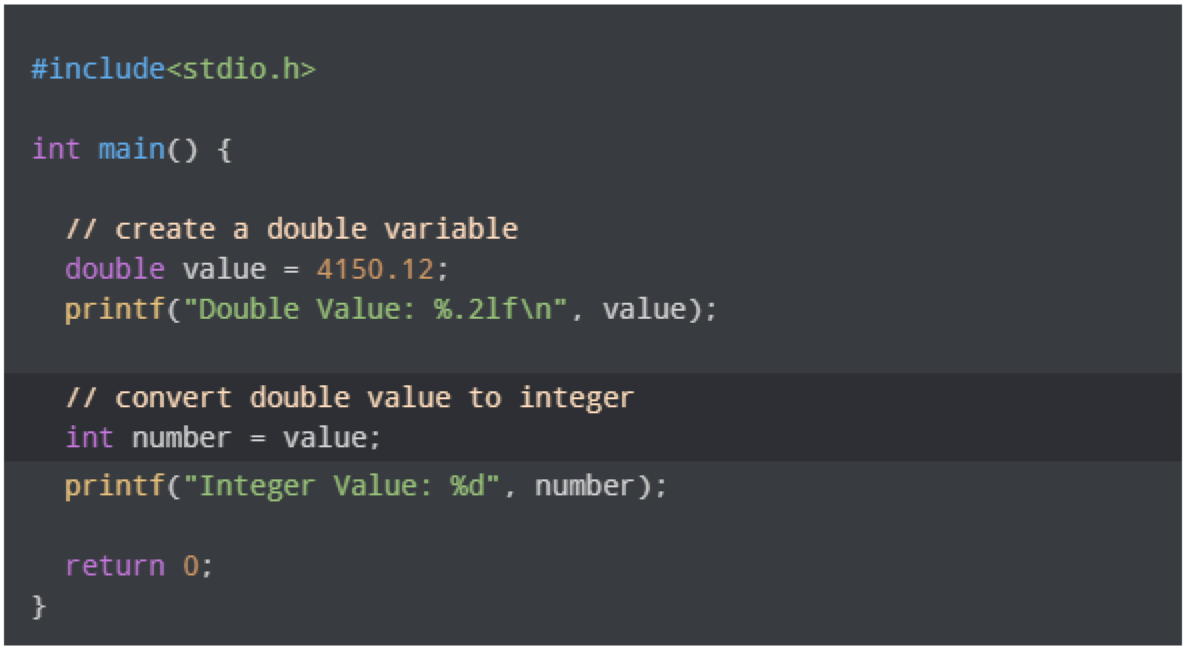
```
#include<stdio.h>

int main() {

  // create an integer variable
  int number = 97;
  printf("Integer Value: %d\n", number);

  // (char) converts number to character
  char alphabet = (char) number;
  printf("Character Value: %c", alphabet);

  return 0;
}
```

Output

```
Integer Value: 97
Character Value: a
```

Data Loss In Type Conversion

In our earlier examples, when we converted a double type value to an integer type, the data after decimal was lost.



```c
#include<stdio.h>

int main() {

  // create a double variable
  double value = 4150.12;
  printf("Double Value: %.2lf\n", value);

  // convert double value to integer
  int number = value;
  printf("Integer Value: %d", number);

  return 0;
}
```

Here, the data **4150.12** is converted to **4150**. In this conversion, data after the decimal, **.12** is lost.

This is because double is a larger data type (**8 bytes**) than int (**4 bytes**), and when we convert data from larger type to smaller, there will be data loss.

Similarly, there is a hierarchy of data types in C programming. Based on the hierarchy, if a higher data type is converted to lower type, data is lost, and if lower data type is converted to higher type, no data is lost.

- **data loss** - if **long double** type is converted to **double type**.
- **no data loss** - if **char** is converted to **int.**

Precedence & Order of Evaluation

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

Input/Output Functions

Output Functions

C programming language provides built-in functions to perform input/output operation.

The output operations are used to display data on user screen (output screen) or printer or any file. The c programming language provides the following built-in output functions...

1. **printf()**
2. **putchar()**
3. **puts()**
4. **fprintf()**

1. printf() function:

The printf() function is used to print string or data values or a combination of string and data values on the output screen.

The printf() function is built-in function defined in a header file called "**stdio.h**". When we want to use printf() function in our program we need to include the respective header file (stdio.h) using the **#include** statement.

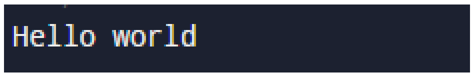
Syntax:

```
printf("message to be display!!!");
```

Example:

```
#include<stdio.h>

#include<conio.h>

void main(){

   printf("Hello World!!!");

}
```

Output:

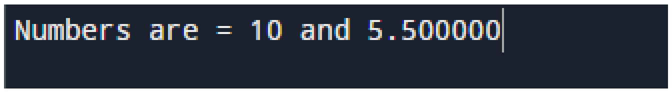


```
Hello world
```

Example:

```
#include<stdio.h>

#include<conio.h>

void main(){

   int i = 10;

   float x = 5.5;

   printf("Numbers are =%d and %f",i, x);

}
```

Output:



```
Numbers are = 10 and 5.500000
```

putchar() function

The putchar() function is used to display a single character on the output screen.

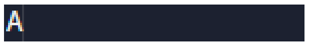
The putchar() functions prints the character which is passed as a parameter to it and returns the same character as a return value. This function is used to print only a single character.

To print multiple characters we need to write multiple times or use a looping statement. Consider the following example program.

Example

```
#include<stdio.h>

#include<conio.h>

void main(){

   char ch = 'A';

   putchar(ch);

}
```

Output:



```
A
```

puts() function

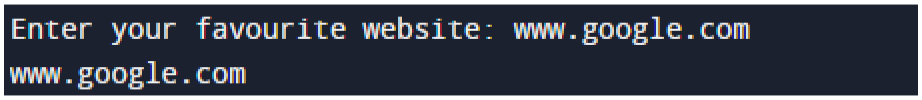
The puts() function is used to display a string on the output screen.

The puts() functions prints a string or sequence of characters till the newline. Consider the following example program.

Example

```
#include<stdio.h>

#include<conio.h>

void main(){

  char name[30];

  printf("\nEnter your favourite website: ");

  gets(name);

  puts(name);

}
```

Output:



```
Enter your favourite website: www.google.com
www.google.com
```

fprintf() function

The fprintf() function is used with the concept of files.

The fprintf() function is used to print a line into the file. When you want to use fprintf() function the file must be opened in writting mode.

Input Functions

The input operations are used to read user values (input) from the keyboard.

The input operations are used to read user values (input) from the keyboard.

1. **scanf()**
2. **getchar()**
3. **getch()**
4. **gets()**
5. **fscanf()**

scanf() function

The scanf() function is used to read multiple data values of different data types from the keyboard. The scanf() function is built-in function defined in a header file called "**stdio.h**".

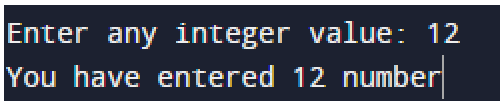
When we want to use scanf() function in our program, we need to include the respective header file (stdio.h) using **#include** statement. The scanf() function has the following syntax...

Syntax

```
scanf("format strings",&variableNames);
```

Example:

```
#include<stdio.h>

#include<conio.h>

void main(){

  int i;

  printf("\nEnter any integer value: ");

  scanf("%d",&i);

  printf("\nYou have entered %d number",i);

}
```



```
Enter any integer value: 12
You have entered 12 number
```

getchar() function

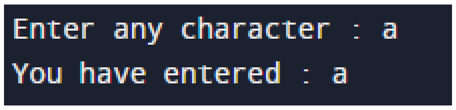
The getchar() function is used to read a character from the keyboard and return it to the program. This function is used to read a single character.

To read multiple characters we need to write multiple times or use a looping statement. Consider the following example program.

Example:

```
#include<stdio.h>

#include<conio.h>

void main(){

   char ch;

   printf("\nEnter any character : ");

   ch = getchar();

   printf("\nYou have entered : %c\n",ch);

}
```

Output:



```
Enter any character : a
You have entered : a
```

getch() function

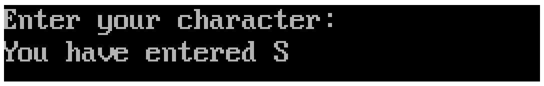
The getch() function is similar to getchar function. The getch() function is used to read a character from the keyboard and return it to the program.

This function is used to read a single character. To read multiple characters we need to write multiple times or use a looping statement. Consider the following example program.

Example

```
#include<stdio.h>

#include<conio.h>

void main(){

  char ch;

  printf("\nEnter your character : ");

  ch = getch();

  printf("\nYou have entered : %c",ch);

}
```

Output:



```
Enter your character:
You have entered S
```

gets() function

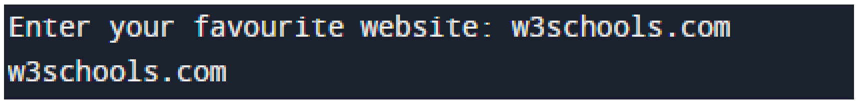
The gets() function is used to read a line of string and stores it into a character array.

The gets() function reads a line of string or sequence of characters till a newline symbol enters. Consider the following example program...

Example:

```
#include<stdio.h>

#include<conio.h>

void main(){

  char name[30];

  printf("\nEnter your favourite website: ");

  gets(name);

  printf("%s",name);

}
```

Output:



```
Enter your favourite website: w3schools.com
w3schools.com
```

fscanf() function

The fscanf() function is used with the concept of files. The fscanf() function is used to read data values from a file. When you want to use fscanf() function the file must be opened in reading mode.