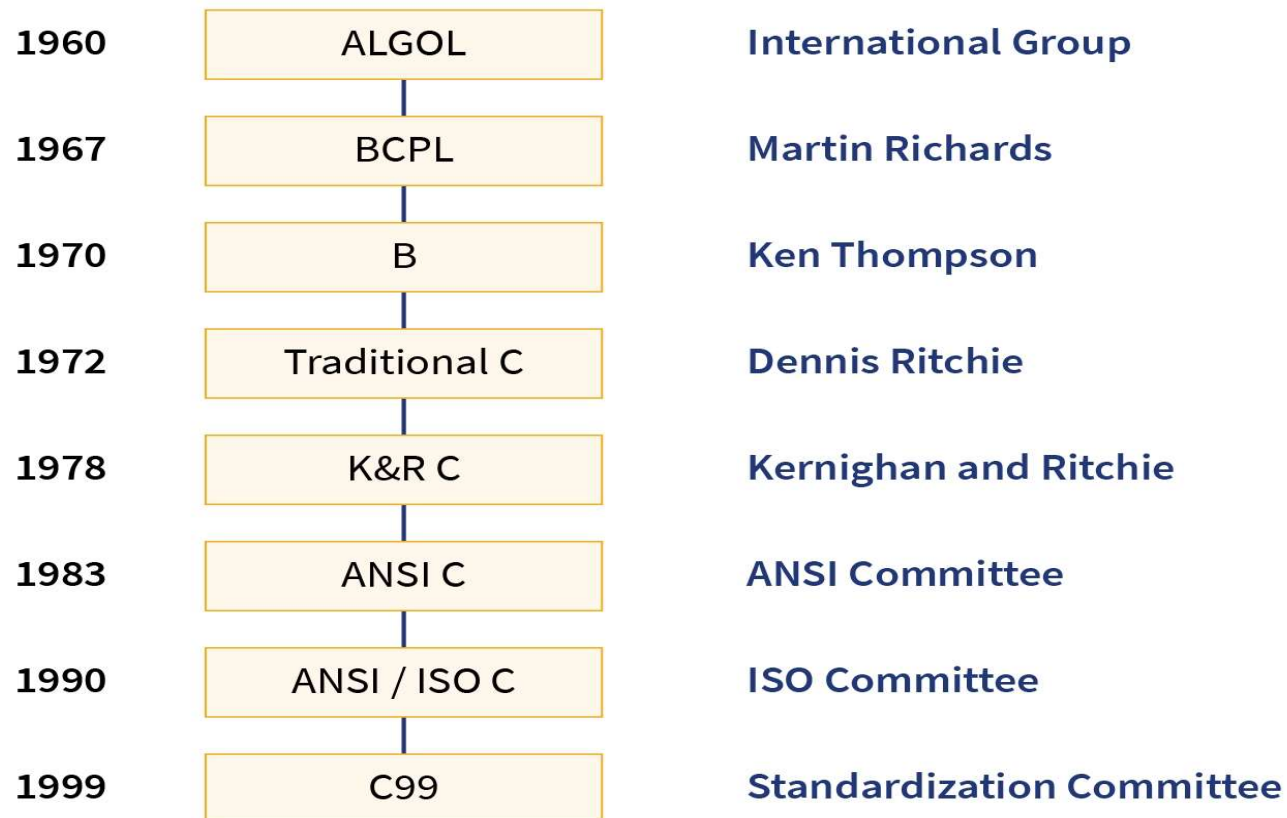




# History of C language

# History of C language



# History of C language



- **C programming is a general-purpose, procedural, imperative computer programming language.**
- **It was developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system.**
- **C is the most widely used computer language.**

# Why to Learn C Programming?

- ▶ **C programming** language is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Software Development Domain.
- ▶ Following are the key advantages of learning C Programming:
  - ▶ **Easy to learn**
  - ▶ **Structured language**
  - ▶ **It produces efficient programs**
  - ▶ **It can handle low-level activities**
  - ▶ **It can be compiled on a variety of computer platforms**

# Facts about C



- ▶ C was invented to write an operating system called UNIX.
- ▶ C is a successor of B language which was introduced around the early 1970s.
- ▶ The language was formalized in 1988 by the American National Standard Institute (ANSI).
- ▶ The UNIX OS was totally written in C.
- ▶ Today C is the most widely used and popular System Programming Language.
- ▶ Most of the state-of-the-art software have been implemented using C.
- ▶ Today's most popular Linux OS and RDBMS MySQL have been written in C.

# Structure of a C Program

## Hello World Example

A C program basically consists of the following parts –

- ▶ Preprocessor Commands
- ▶ Functions
- ▶ Variables
- ▶ Statements & Expressions
- ▶ Comments

# Structure of a C Program

## Hello World Example

```
#include <stdio.h>
int main()
{
    /* my first program in C */
    printf("Hello, World! \n");
    return 0;
}
```



Thank you



# Why C is So Important?

- ⌘ C is worlds most popular programming lang.
- ⌘ C important to build programming skills
- ⌘ Most popular databases like **MySQL**, **Oracle** is written in C
- ⌘ Most popular for hardware programming, hence almost all **device drivers** are written in C
- ⌘ **UNIX** is developed in C
- ⌘ Covers basic features of all programming language
- ⌘ Used in campus recruitment process



# Most Important Features of C?



## Simple and Efficient:

- C language is known for its simple and easily comprehensible syntax, making it an excellent choice for beginners.
- The simplicity of C facilitates efficient application development and redesign.



## Fast:

- Being a statically typed programming language and compiler-based, C exhibits faster execution compared to dynamic languages.
- Its focus on essential features enhances speed, avoiding the overhead associated with more feature-rich languages.



# Most Important Features of C?



## Portability:

- C programs are machine-independent, allowing code to run on various systems with minimal or no machine-specific modifications.
- This portability feature makes C versatile for deployment on different platforms.



## Extensibility:

- C enables quick and easy extension of programs. Existing code can be altered to add new features, functionalities, and operations without extensive modifications, providing flexibility to programmers.



# Most Important Features of C?



## Function-Rich Libraries:

- C comes with an extensive set of libraries that include built-in functions, simplifying coding for programmers.
- The availability of a vast array of functions allows the development of diverse programs and applications.



## Dynamic Memory Management:

- C supports dynamic memory management (DMA), allowing the allocation and management of data structure size during runtime.
- Functions like `malloc()`, `calloc()`, `realloc()`, and `free()` facilitate effective memory utilization.



# Most Important Features of C?



## **Modularity With Structured Appr. :**

- C's modularity, coupled with its structured nature, enables breaking code into different parts using functions.
- These functions can be stored as libraries for future use, enhancing code organization and reusability.
- This structured approach enhances code readability, reduces errors, and facilitates future maintenance.



# Most Important Features of C?

## ⌘ Mid-Level Programming Language:

- Originally developed for low-level programming, C has evolved to support high-level programming features, making it a mid-level language.
- It combines the advantages of both low and high-level languages, allowing direct hardware manipulation.



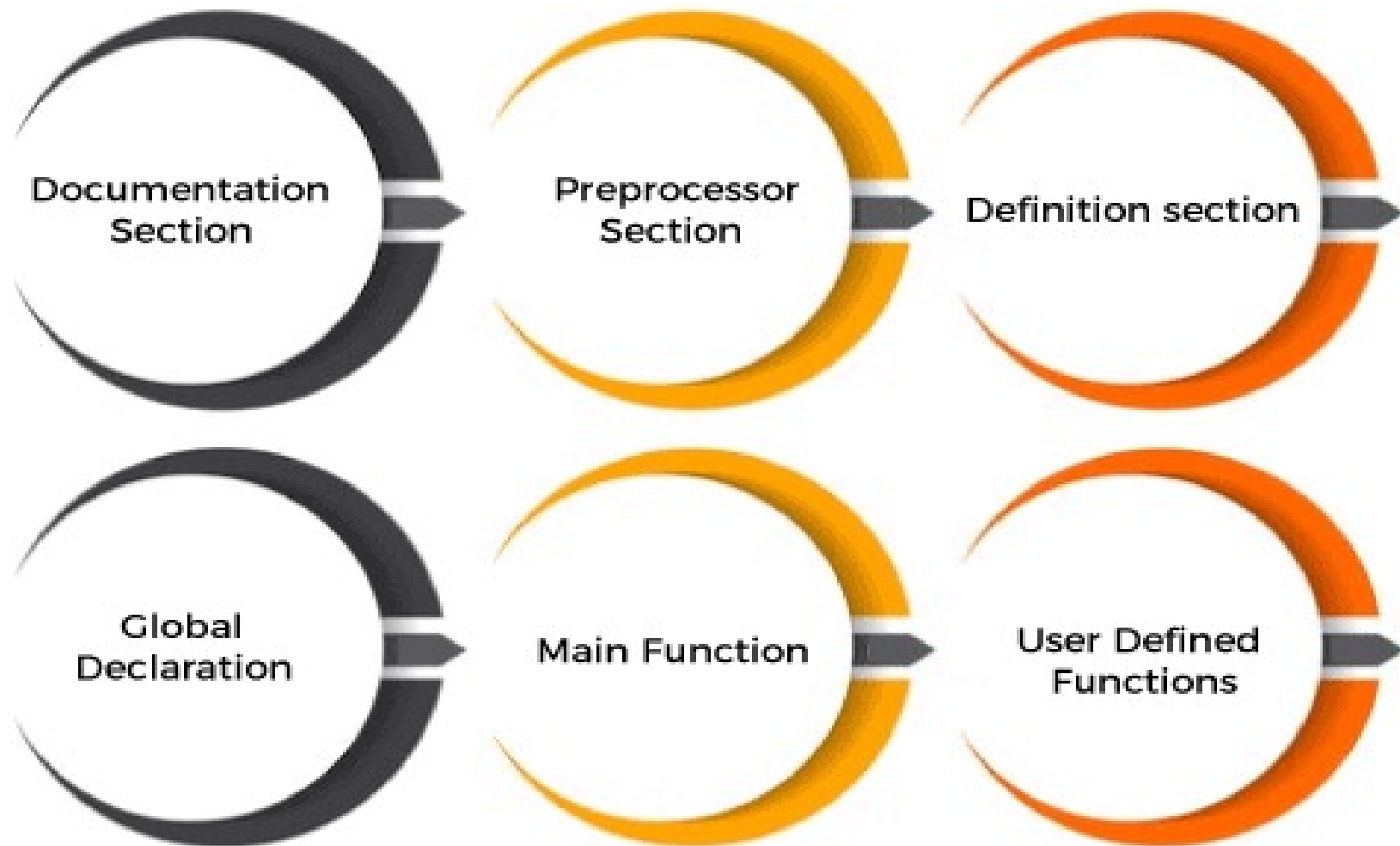
# Basic Structure of C Program

- ⌘ 1. Documentation section
- ⌘ 2. Preprocessor section/Header Files
- ⌘ 3. Definition section
- ⌘ 4. Global declaration
- ⌘ 5. Main function
- ⌘ 6. User defined functions



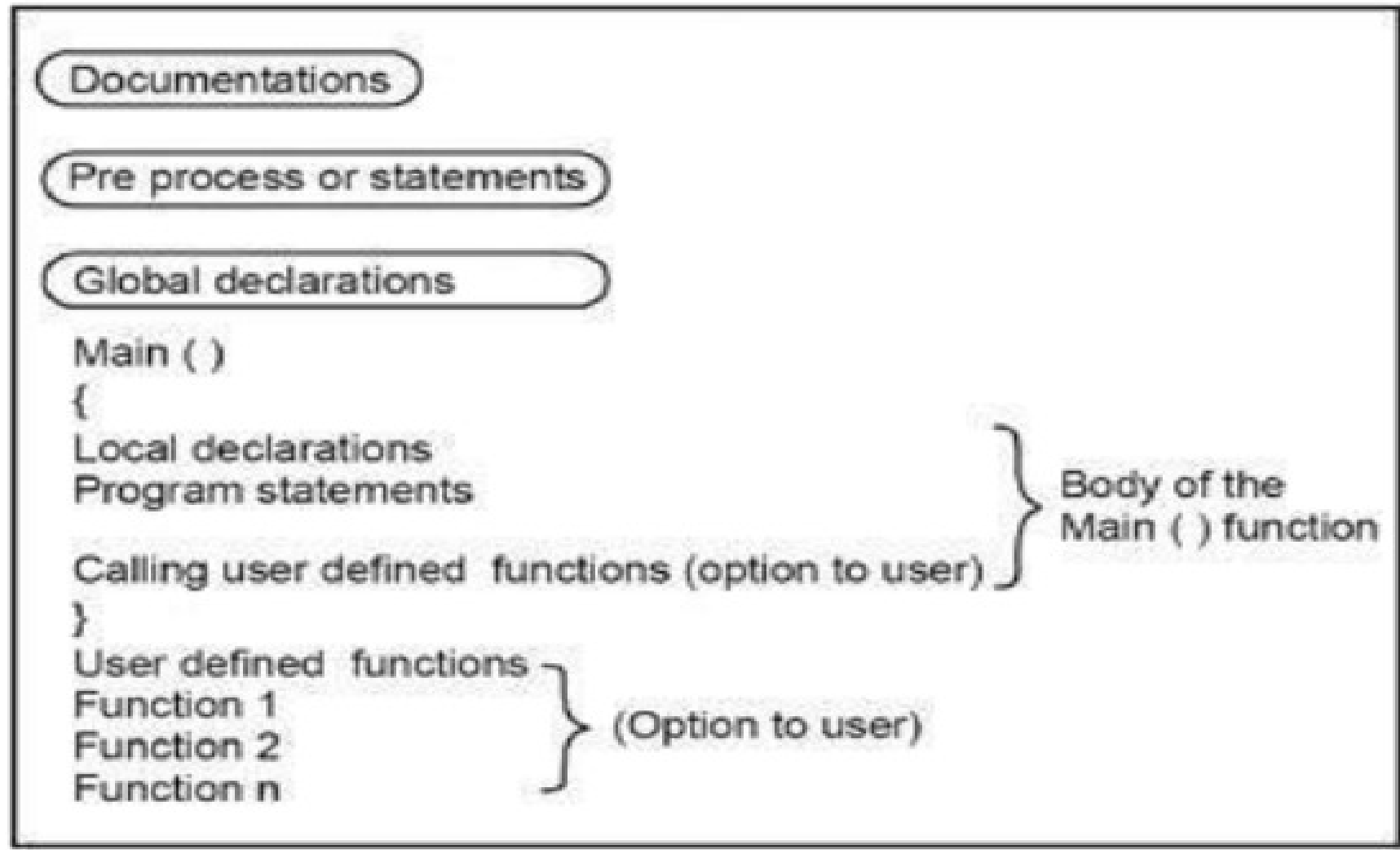
# Basic Structure of C Program

## Sections of a C Program





# Basic Structure of C Program



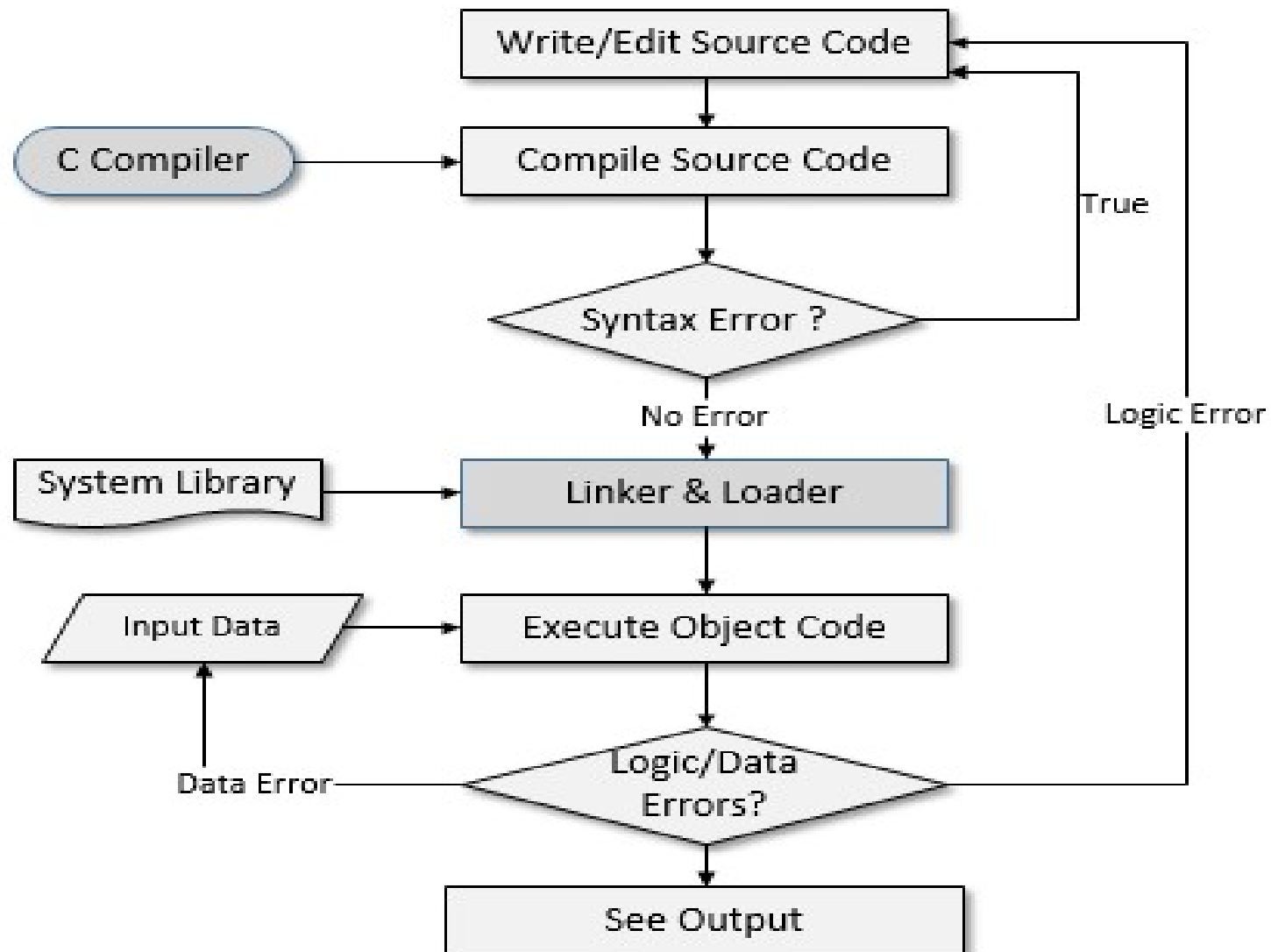
# Execution of C Program

⌘ Executing of C program involves series of steps.

- 1. Writing/Creating Program
- 2. Compiling the Program
- 3. Linking the Program with C lib functions
- 4. Executing the Program



# Process of Execution of C Prog.



# Process of Execution of C Prog.

✂ Type `gcc c -o [program_name].exe [program_name].c`

✂ For Ex.

My program name is hello.c

My output file name is hello.exe

Then the command will be

**`gcc -o hello.exe hello.c`**



# Process of Execution of C Prog.

Administrator: Command Prompt

Microsoft Windows [Version 10.0.19041.388]  
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd C:\Source\Programs

C:\Source\Programs>gcc c -o hello\_world.exe hello\_world.c

C:\Source\Programs>hello\_world.exe

wikiHow



# Character Set

Characters in C are grouped into 4 categories:

- ⌘ **Letters** a-z A-Z
- ⌘ **Digits** 0-9
- ⌘ **Special Characters** ; @, #, &, () <>
- ⌘ **White Spaces**



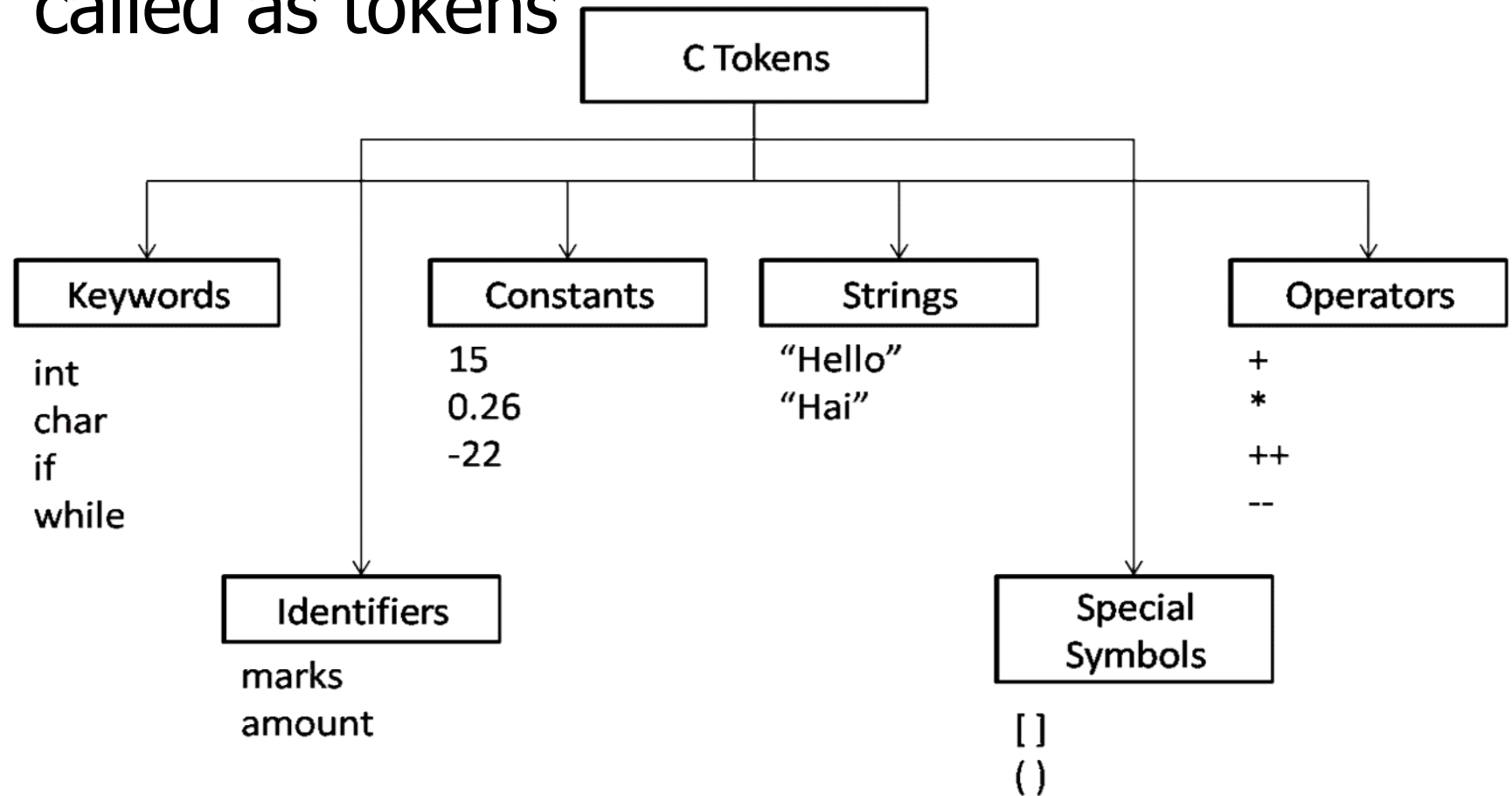
# Character Set

Types	Character Set
Uppercase Alphabets	A, B, C, ... Y, Z
Lowercase Alphabets	a, b, c, ... y, z
Digits	0, 1, 2, 3, ... 9
Special Symbols	~ ' ! @ # % ^ & * ( ) _ - + =   \ { } [ ] : ; " ' < > , . ? /
White spaces	Single space, tab, new line.



# Tokens in C

⌘ Individual words or punctuation marks are called as tokens





# Tokens in C

## ⌘ Keywords

← These are reserved words of the C language. For example `int`, `float`, `if`, `else`, `for`, `while` etc.

## ⌘ Identifiers

← An Identifier is a sequence of letters and digits, but must start with a letter. Underscore ( `_` ) is treated as a letter. Identifiers are case sensitive. Identifiers are used to name variables, functions etc.

← Valid: `Root`, `_getchar`, `__sin`, `x1`, `x2`, `x3`, `x_1`, `If`

← Invalid: `324`, `short`, `price$`, `My Name`

## ⌘ Constants

← Constants like `13`, `'a'`, `1.3e-5` etc.



# 32 Keywords in C

**auto**

**break**

**case**

**char**

**const**

**continue**

**default**

**do**

**double**

**else**

**enum**

**extern**

**float**

**for**

**goto**

**if**

**int**

**long**

**register**

**return**

**short**

**signed**

**sizeof**

**static**

**struct**

**switch**

**typedef**

**union**

**unsigned**

**void**

**volatile**

**while**



# Identifier Vs Keywords

Sr. No	Identifier	Keyword
1	An identifier is a unique name created by the programmer to define a variable, structure, class, or function.	Keywords are restricted words with a definite, predetermined meaning. Any programme statement may be defined with the aid of keywords.
2	A keyword always starts in lowercase.	The initial character in the identification can be capitalized, lowercase, or start with an underscore.
3	It can only have alphabetical characters.	It can have numbers, alphabetical characters, and underscores.
4	Examples of keywords are double, int, auto, char, break, and many others.	Examples of IDs are test, count1, high speed, etc.



# Tokens in C

## ⌘ String Literals

← A sequence of characters enclosed in double quotes as "...". For example "13" is a string literal and not number 13. 'a' and "a" are different.

## ⌘ Operators

← Arithmetic operators like +, -, \*, / , % etc.

← Logical operators like ||, &&, ! etc. and so on.

## ⌘ White Spaces

← Spaces, new lines, tabs, comments ( A sequence of characters enclosed in /\* and \*/ ) etc. These are used to separate the adjacent identifiers, keywords and constants.



# Basic Data Types

## ⌘ Integral/Integer Types

← Integers are stored in various sizes. They can be signed or unsigned.

### ← Example

Suppose an integer is represented by a byte (8 bits). Leftmost bit is sign bit. If the sign bit is 0, the number is treated as positive.

Bit pattern 01001011 = 75 (decimal).

The largest positive number is 01111111 =  $2^7 - 1 = 127$ .

Negative numbers are stored as two's complement or as one's complement.

-75 = 10110100 (one's complement).

-75 = 10110101 (two's complement).



# Basic Data Types

## ⌘ Integral Types

- ← `char`                      Stored as 8 bits. Unsigned 0 to 255.  
Signed -128 to 127.
- ← `short int`                Stored as 16 bits. Unsigned 0 to 65535.  
Signed -32768 to 32767.
- ← `int`                        Same as either short or long int.
- ← `long int`                Stored as 32 bits. Unsigned 0 to 4294967295.  
Signed -2147483648 to 2147483647



# Basic Data Types

## ⌘ Floating Point Numbers

- ← Floating point numbers are rational numbers. Always signed numbers.
- ← **float**    Approximate precision of 6 decimal digits .
  - Typically stored in 4 bytes with 24 bits of signed mantissa and 8 bits of signed exponent.
- ← **double**    Approximate precision of 14 decimal digits.
  - Typically stored in 8 bytes with 56 bits of signed mantissa and 8 bits of signed exponent.
- ← One should check the file limits.h to what is implemented on a particular machine.



# Constants

## ⌘ Numerical Constants

- ← Constants like 12, 253 are stored as `int` type. No decimal point.
- ← 12L or 12l are stored as `long int`.
- ← 12U or 12u are stored as `unsigned int`.
- ← 12UL or 12ul are stored as `unsigned long int`.
- ← Numbers with a decimal point (12.34) are stored as `double`.
- ← Numbers with exponent ( $12e-3 = 12 \times 10^{-3}$ ) are stored as `double`.
- ← 12.34f or 1.234e1f are stored as `float`.
- ← These are not valid constants:  
25,000    7.1e 4                      \$200    2.3e-3.4 etc.





# Constants

## ⌘ Character and string constants

← `'c'` , a single character in single quotes are stored as char.

Some special character are represented as two characters in single quotes.

`'\n'` = newline, `'\t'` = tab, `'\\'` = backlash, `'\"'` = double quotes.

Char constants also can be written in terms of their ASCII code.

`'\060'` = `'0'` (Decimal code is 48).

← A sequence of characters enclosed in double quotes is called a string constant or string literal. For example

`"Charu"`

`"A"`

`"3/9"`

`"x = 5"`



# Variables

## ⌘ Naming a Variable/Identifier

- ← Must be a valid identifier.
- ← Must not be a keyword
- ← Names are case sensitive.
- ← Variables are identified by only first 32 characters.
- ← Library commonly uses names beginning with \_.
- ← Naming Styles: Uppercase style and Underscore style
- ← `lowerLimit`    `lower_limit`
- ← `incomeTax`            `income_tax`



# Declarations

## ⌘ Declaring a Variable

← Each variable used must be declared.

← A form of a declaration statement is

**data-type var1, var2, ...;**

← Declaration announces the data type of a variable and allocates appropriate memory location. No initial value (like 0 for integers) should be assumed.

← It is possible to assign an **initial value** to a variable in the declaration itself like below:

**data-type var = expression;**

← **Examples**

```
int sum = 0;
```

```
char newLine = '\n';
```

```
float epsilon = 1.0e-6;
```



# Global and Local Variables

## ⌘ Global Variables

← These variables are declared outside all functions.

← Life time of a global variable is the entire execution period of the program.

← Can be accessed by any function defined below the declaration, in a file.

```
/* Compute Area and Perimeter of a
circle */
#include <stdio.h>
float pi = 3.14159; /* Global */

main() {
    float rad;      /* Local */

    printf( "Enter the radius " );
    scanf("%f" , &rad);

    if ( rad > 0.0 ) {
        float area = pi * rad * rad;
        float peri = 2 * pi * rad;

        printf( "Area = %f\n" , area );
        printf( "Peri = %f\n" , peri );
    }
    else
        printf( "Negative radius\n");

    printf( "Area = %f\n" , area );
}
```



# Global and Local Variables

## ❖ Local Variables

- ❖ These variables are declared inside some functions.
- ❖ Life time of a local variable is the entire execution period of the function in which it is defined.
- ❖ Cannot be accessed by any other function.
- ❖ In general variables declared inside a block are accessible only in that block.

```
/* Compute Area and Perimeter of a
circle */
#include <stdio.h>
float pi = 3.14159; /* Global */

main() {
    float rad;      /* Local */

    printf( "Enter the radius " );
    scanf("%f" , &rad);

    if ( rad > 0.0 ) {
        float area = pi * rad * rad;
        float peri = 2 * pi * rad;

        printf( "Area = %f\n" , area );
        printf( "Peri = %f\n" , peri );
    }
    else
        printf( "Negative radius\n");

    printf( "Area = %f\n" , area );
}
```



# Storage Class

- ❖ Every variable in C has two properties:
  1. Type
  2. Storage class
- ❖ Among them, the type refers to the data type of the variable, and storage classes in C determine the **scope**, **lifetime**, and **visibility** of the variable.
- ❖ Storage class in C is used to find the lifetime, visibility, memory location, and initial value of a variable.



# Storage Class

❖ There are four different types of storage classes that we use in the C language:

1. Automatic Storage Class
2. External Storage Class
3. Static Storage Class
4. Register Storage Class



# Storage Class: Automatic

- ❖ Automatic variables are allocated memory automatically at runtime.
- ❖ The **scope & visibility** 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.



# Storage Class: Automatic

```
include <stdio.h>
int main( )
{
    auto int i = 11;
    {
        auto int i = 22;
        {
            auto int i = 33;
            printf("%d", i);
        }
        printf("%d", i);
    }
    printf("%d", i);
}
```

The output of the Program:

33 22 11



# Storage Class: Static

- ❖ The keyword used to define static variable is **static**.
- ❖ 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.



# Storage Class: Static

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

The output of the Program:

1 0

2 1



# Storage Class: Register

- ❖ 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 **0**.



# Storage Class: Register

- ❖ The register keyword is used for the variable which should be stored in the CPU register.
- ❖ However, it is compilers 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 Class: Register

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

The output of the Program:

0



# Storage Class: External

- ❖ The external storage class in C is an intriguing feature that allows extending a program's functionality by sharing variables across multiple source files.
- ❖ By allowing variables initialized in one file to be utilized in another, the 'extern' keyword in C encourages modularity and improves the organization and maintenance of the code.
- ❖ It makes it simpler to collaborate and construct ambitious, complex projects since it promotes communication across code segments.



# Storage Class: External

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block





# Symbolic Constants in C

- ❖ Identifiers are used to represent fixed values in programs using symbolic constants in the C programming language.
- ❖ These parameters are frequently used to increase the readability, maintainability, and modifiability of code, which may be numerical or not.
- ❖ The C language's **"#define"** command is used to declare symbolic constants.



# Symbolic Constants in C

❖ Syntax:

Syntax for defining a symbolic constant in C:

```
#define MAX_V 100
```

```
#define PI 3.1428
```



# The *const* keyword

- ❖ Variables can be declared as constants by using the "*const*" keyword before the datatype of the variable.
- ❖ The constant variables can be initialized once only.
- ❖ The default value of constant variables are *zero*.



# The *const* keyword

```
#include <stdio.h>
int main() {
    const int a;
    const int b = 12;
    printf("The default value of variable
a : %d", a);
    printf(" The value of variable b :
%d", b);
    return 0;
}
```



# Declaring a Variable as Volatile

- ❖ In C, the volatile keyword is used to indicate to the compiler that a variable's value may change unexpectedly, so it should not rely on the value being cached in a register or optimized away.
- ❖ When a variable is declared as volatile, the compiler must generate code to read and write the variable's value from memory each time it is used, rather than relying on a cached value in a register.



# Declaring a Variable as Volatile

The syntax for using the volatile keyword in C is as follows:

```
volatile    data_type    variable_name;
```



# Constants Vs Variables

Sr. No	Constants	Variables
1	A constant does not change its value as the equation is solved.	A variable, on the other hand, changes its value depending on the equation.
2	Constants are usually written in numbers(whether fractions, integers, decimals or real numbers).	Variables are written as letters or symbols.
3	Constants usually represent the known values in an equation or expression.	Variables, on the other hand, represent unknown values.
4	Constants have fixed face values.	Variables do not have a fixed face value..



# Course Name: Fundamentals of Programming Language



**Prof. Mangesh Hajare,**  
Assistant Professor In Comp. Engg,  
Army Institute of Technology, Pune.



# UNIT 1: Program Design Tools

# Lecture Outline

- Algorithms
- Flowcharts
- Pseudo-codes
- Implementation of algorithms.

# Algorithm

- In our day-to-day life we perform activities by following certain sequence of steps.
- Examples of activities include getting ready for school, making breakfast, riding a bicycle, wearing a tie, solving a puzzle etc.
- To complete each activity, we follow a sequence of steps.
- Suppose following are the steps required for an activity '**riding a bicycle**':
  - 1) remove the bicycle from the stand,
  - 2) sit on the seat of the bicycle
  - 3) start peddling,
  - 4) stop on reaching the destination.

## What is Algorithm?

*“An algorithm is the list of instructions and rules that a computer needs to do to complete a task.”*

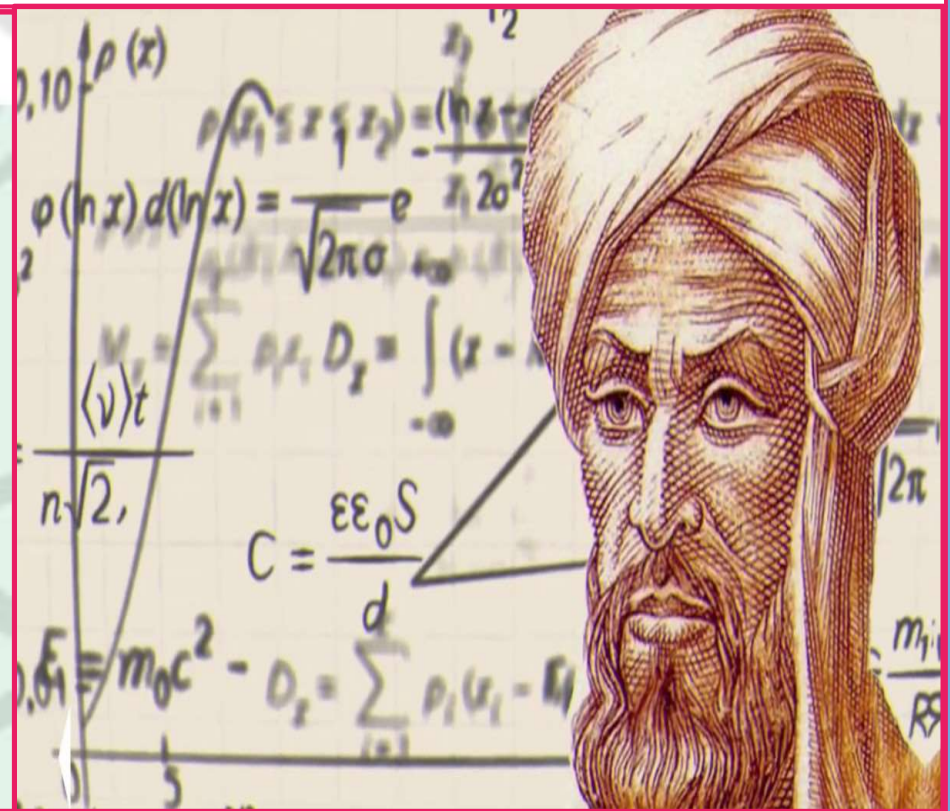
*“An algorithm is a set of precise instructions for solving a problem or accomplishing a task ”*

*“ The meaning of algorithm is a set of steps that are followed in order to solve a mathematical problem or to complete a computer process. ”*

# Origin of Algorithm



The origin of the term Algorithm is traced to Persian astronomer and mathematician, Abu Abdullah Muhammad ibn Musa Al-Khwarizmi (c. 850 AD) as the Latin translation of Al-Khwarizmi was called 'Algorithmi'.



## Why do we need an Algorithm?

- A programmer writes a program to instruct the computer to do certain tasks as desired.
- The computer then follows the steps written in the program code.
- Therefore, the programmer first prepares a roadmap of the program to be written, before actually writing the code.
- Without a roadmap, the programmer may not be able to clearly visualize the instructions to be written and may end up developing a program which may not work as expected.

## Why do we need an Algorithm?

- Writing an algorithm is mostly considered as a first step to programming.
- Once we have an algorithm to solve a problem, we can write the computer program for giving instructions to the computer in high level language.
- If the algorithm is correct, computer will run the program correctly, every time.
- So, the purpose of using an algorithm **is to increase the reliability, accuracy and efficiency** of obtaining solutions.

## Algorithm: Examples

- Write an algorithm to find the number is even or odd.

Step 1 :           Input number as A.

Step 2:           Check if  $A \% 2 == 0$   
if " true" Print "A is Even Number"  
else Print "A is Odd Number"

Step 3:           End.



## Characteristics of a good algorithm

- **Input:** the algorithm receives some input (zero or more)
- **Output:** the algorithm produces some output.(atleast one)
- **Precision or Unambiguous:** the steps are precisely stated or defined.
- **Uniqueness or Definiteness :** results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Finiteness:** the algorithm always stops after a finite number of steps. Must be terminated after finite number of steps.

# Representation of Algorithm

- Algorithms are mainly represented by two ways
  1. **Pseudocode**
  2. **Flowcharts**

# What is Pseudocode?

- Pseudocode is an informal high-level representation of the actual code that shows how an algorithm or a computer program works in plain English.
- Many times algorithms are written in simple English this is known as Pseudocode.
- It simplifies program development by classifying into two parts:
  - 1. Logic Design**
  - 2. Coding**
- It is intended for human reading and cannot be executed directly by the computer.
- No specific standard for writing a pseudocode exists.

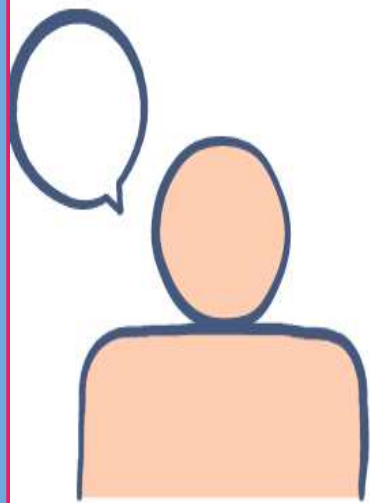
# What is Pseudocode?

- The word “pseudo” means “**not real**” so “pseudocode” means “not real code”.
- In pseudocode, you don't have to think about semi-colons, curly braces, the syntax for arrow function and other core language principles.
- Pseudocode acts as the bridge between your brain and computer's code executor.
- It allows you to plan instructions which follow a logical pattern, without including all of the technical or programming details.

## What is Pseudocode?

- In pseudocode, you don't have to think about **semi-colons, curly braces**, the **syntax for arrow function** and other core language principles.
- When you're writing code in a programming language, you'll have to battle with **strict syntax and rigid coding patterns**.
- Pseudocode acts as the bridge between your brain and computer's code executor.
- It allows you to plan instructions which follow a logical pattern, without including all of the technical or programming details.

# What is Pseudocode?



Idea



Pseudocode



```
/* Hello World.java
*/

public class Hello World
{
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Actual Code



## Example Pseudocode

```
1  set i to 0
2  for each i from 0 to 9
3      if i is odd
4          print i
5  end for loop
```








# Flowcharts

- A flowchart is a visual representation of an algorithm.
- A flowchart is a diagram made up of boxes, diamonds and other shapes, connected by arrows.
- Each shape represents a step of the solution process and the arrow represents the order or link among the steps.
- There are standardized symbols to draw flowcharts.





# Flowcharts Symbols

Flowchart symbol	Function	Description
	Start/End	Also called "Terminator" symbol. It indicates where the flow starts and ends.
	Process	Also called "Action Symbol," it represents a process, action, or a single step.
	Decision	A decision or branching point, usually a yes/no or true/false question is asked, and based on the answer, the path gets split into two branches.
	Input/Output	Also called data symbol, this parallelogram shape is used to input or output data
	Arrow	Connector to show order of flow between shapes.



# Flowcharts Symbols

## Algorithm

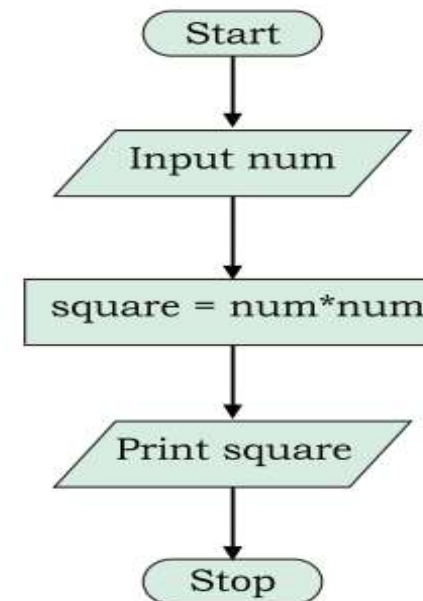
Algorithm to find square of a number.

Step 1: Input a number and store it to num

Step 2: Compute  $\text{num} * \text{num}$  and store it in square

Step 3: Print square

## Flowchart





# Implementation of Algorithm

- Implementation of algorithm that has been properly designed in top down fashion.
- If an algorithm is properly designed then the path of execution should flow in straight line from top to bottom.
- Programs implemented in this way are much easier to understand and debug.
- Even much more easier to modify also.
- Number of points that should be focused while implementing the algorithm.



# Implementation of Algorithm

- Use of modularity
- Choice of variable names.
- Documentation of program.
- Debugging a program.
- Program testing

How well did you like this lesson?



Students, drag the icon!



Pear Deck Interactive Slide  
Do not remove this bar