

---

# Problem Solving Through Programming C

---

Pateel G P

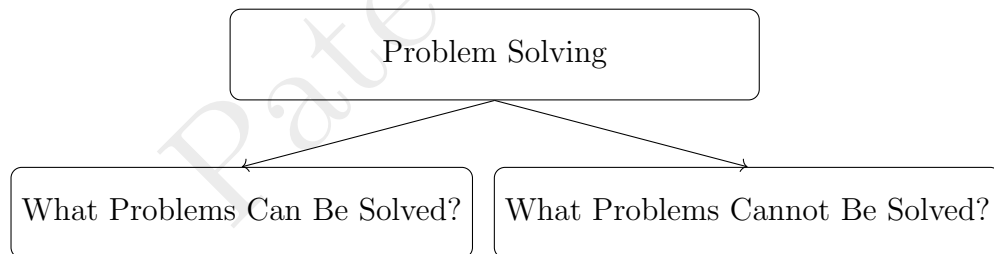
---

# Introduction to Computer and C language

---

## 1.1 Introduction

C programming is well-suited for solving structured, logical, and computation-based problems. It is widely used in system programming, embedded systems, and applications that require high performance and control.



### 1.1.1 What Problems Cannot Be Solved?

Problems that are unclear, subjective, or based on human emotions and creativity are difficult or impossible to solve using C programming. These typically lack a clear algorithmic structure.

### Examples

- Managing personal relationships (e.g., making friends, resolving conflicts)
- Creative tasks (e.g., writing novels, poetry, painting)
- Human-centric logistics (e.g., shifting house, planning travel routes)
- Medical diagnosis through microscopic image interpretation

### 1.1.2 What Problems Can Be Solved?

Problems that are well-defined, mathematical, logical, and can be broken down into a sequence of finite steps are generally solvable using C programming. These include mathematical and computational.

### Examples

- Solving mathematical equations (e.g., linear, quadratic, differential equations)
- Performing statistical computations (e.g., mean, standard deviation, regression)
- Data processing tasks (e.g., searching, sorting, writing to files)
- Control and automation systems (e.g., traffic lights, path finding)
- Optimization problems (e.g., resource allocation, memory management)

## 1.2 Introduction to Computer System

A computer system is an integrated setup of hardware and software designed to perform data processing tasks efficiently. It accepts input, processes it, stores information, and produces output. The [Figure 1.1](#) show the block diagram of computer system.

**Input Unit:** An input device is a hardware or peripheral device used to send data to a computer. An input device allows users to communicate and feed instructions and data to computers for processing, display, storage and/or transmission.

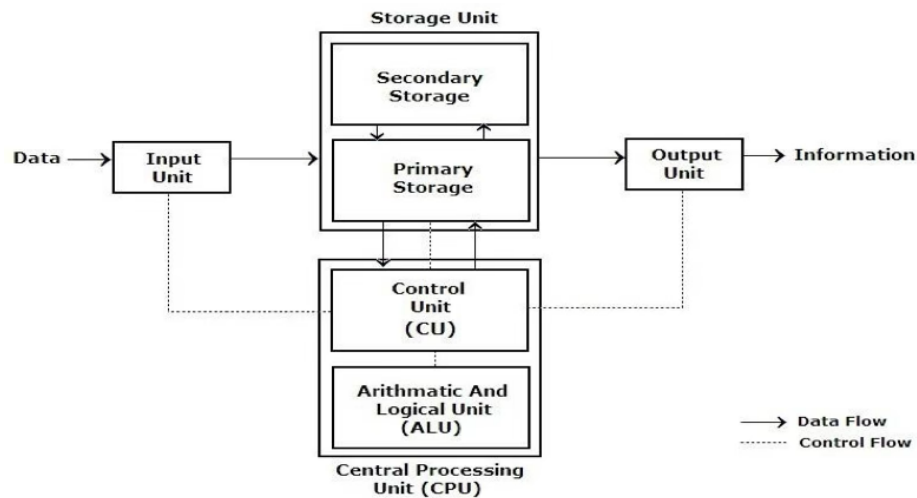


Figure 1.1: Block diagram of computer system

Some of the most popularly used input devices are: Mouse, Light Pen, Touch Screen, Keyboard, Scanner, OCR and MICR, Bar Code Reader, Joy Stick, etc.

**Output Unit:** The processed data is displayed in the form of result through the output device. Some of the most popularly used Output devices are: Visual Display Unit (Monitor), Printers (Dot Matrix, Line Printers, Ink-jet, Laser Printer), Plotters, etc.

**Central Processing Unit:** The Central Processing Unit (CPU) is known as the heart of the computer which takes control of the entire processing system of a computer. It performs the basic arithmetical, logical, and input/output operations of a computer system. It also transfers information to and from other components, such as a disk drive or the keyboard.

The CPU has three important sub units:

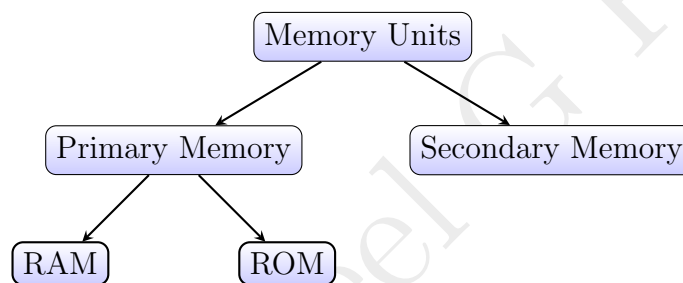
1. Arithmetic-Logic Unit
2. Control Unit
3. Memory Unit

**Arithmetic-Logic Unit (ALU):** The Arithmetic Logic Unit (ALU) is an essential component of a computer's central processing unit (CPU) responsible for performing arithmetic operations such as addition, subtraction, multiplication, and division, as well as logical operations like comparisons and Boolean logic. It processes

data received from input devices or main memory, executes the required calculations or logical evaluations, and then stores the results back in the main memory. By enabling decision-making based on comparisons, the ALU plays a crucial role in the execution of programs and overall computer functionality.

**Control Unit:** The Control Unit (CU) is a vital part of the CPU that manages and coordinates the operations of all other components in the system. It controls the flow of data and instructions between different units, ensuring that each part of the computer works in harmony. The CU fetches instructions from the main memory, decodes them to understand the required operation, and then directs the Arithmetic Logic Unit (ALU) and other components to carry out the specified tasks. In essence, it acts as the brain's "traffic controller" within the computer.

**Memory Unit:** Computer memory is divided into two types: Primary memory and Secondary memory.



**Primary Memory:** The Primary memory is also called Main memory, is used to store data during processing. Once the CPU has carried out an instruction, it needs the result to be stored. This storage space is provided by the computer's memory.

The storage capacity of the memory is generally measured in megabytes:

- 1 nibble = 4 bits
- 8 Bits = 1 Byte
- 1024 Bytes = 1 Kilobyte (KB)
- 1024 Kilobytes = 1 Megabyte (MB)
- 1024 Megabytes = 1 Gigabyte (GB)

Different kinds of primary memory are:

★ Random Access Memory (RAM)

★ Read Only Memory (ROM)

### **Random Access Memory**

- RAM is a volatile memory, which means that the stored information is lost when the power is switched off.
- Used to read and write data in RAM.
- It temporarily stores data and instructions that the CPU needs while performing tasks.
- Access time for RAM is much faster compared to secondary storage devices.
- RAM comes in different types such as DRAM (Dynamic RAM) and SRAM (Static RAM).
- The larger the RAM size, the more programs and data the system can handle simultaneously.

### **Read Only Memory**

- We can only read the data from ROM; we cannot write anything into it, and the data is permanent.
- ROM is a non-volatile memory, meaning it retains data even when the power is turned off.
- It is used to store software like BIOS, firmware, and other essential startup instructions.
- The contents of ROM are written during manufacturing and are not meant to be modified during normal operation.
- ROM types include PROM (Programmable ROM), EPROM (Erasable Programmable ROM), and EEPROM (Electrically Erasable Programmable ROM).

### **Secondary Memory**

- The data stored in it is permanent.
- Data can be deleted if necessary.

- It is cheaper than primary memory.
- It has high storage capacity.

There are different kinds of secondary storage devices available. Few of them are:

- Floppy Disk
- Fixed or Hard Disk
- Optical Disk like: CD (Compact Disk), DVD (Digital Versatile Disk)
- Magnetic Tape Drive

ROM (Read Only Memory)	RAM (Random Access Memory)
Non-volatile memory	Volatile memory
Cheaper than RAM	More expensive than ROM
Cannot be updated or corrected	Can be updated and corrected
Serves as permanent data storage	Serves as temporary data storage

Table 1.1: Comparison between ROM and RAM

### 1.3 Program Development Steps

Program development is a structured process used to design, implement, test, and maintain software applications. The main steps involved are:

1. **Program Analysis:** This step involves a detailed examination of the problem to identify the most suitable solution. It includes:
  - **Problem Definition:** Clearly define the problem by identifying the input, expected output, and constraints.
  - **Mathematical Modeling:** Represent the problem in a mathematical or logical form to better understand its structure.
  - **Exploring Multiple Solutions:** Consider different possible algorithms or approaches to solve the problem.
  - **Selecting the Best Solution:** Choose the most efficient and feasible solution based on factors such as time complexity, memory usage, and scalability.

2. **Algorithm Design:** Develop a step-by-step solution for the problem in the form of an algorithm or flowchart that outlines the logic clearly.
3. **Program Coding:** Convert the designed algorithm into source code using a programming language such as C.
4. **Compilation and Execution:** It is the process of converting the program written in text editor(Source code) to machine understandable code (Machine Code). The execution is the process of operation performed by the machine based on the program.
5. **Debugging and Testing:** Debugging is the process of identifying and correcting the errors. Test the program with different sets of input data, including boundary cases, to ensure its correctness and reliability.
6. **Documentation:**  
Prepare documentation that explains the program's functionality, structure, logic, and usage instructions.

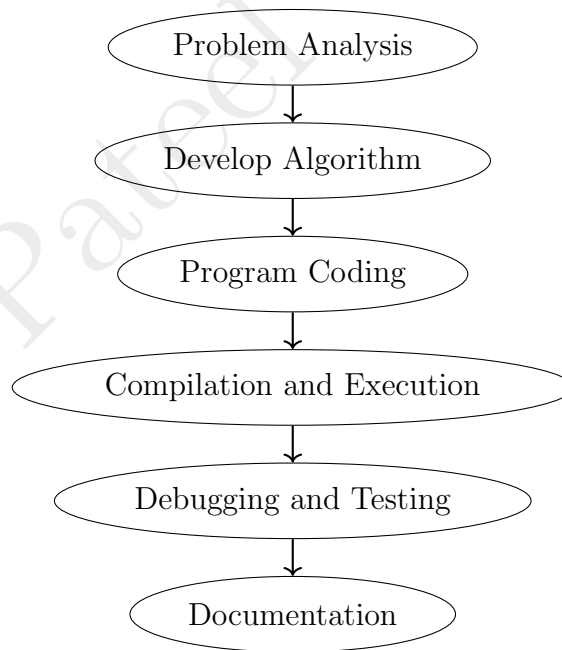


Figure 1.2: Program Development Steps in C Programming



**Problem Analysis:** Example to Calculate the Mean of  $n$  Numbers:

- **Problem Definition:**

- *Input:*  $n$  numbers (e.g., marks of students).
- *Expected Output:* The mean (average) value.
- *Constraints:*  $n > 0$ , all inputs must be valid numbers.

- **Mathematical Modeling:**

$$\text{Mean} = \frac{\text{Sum of all numbers}}{n}$$

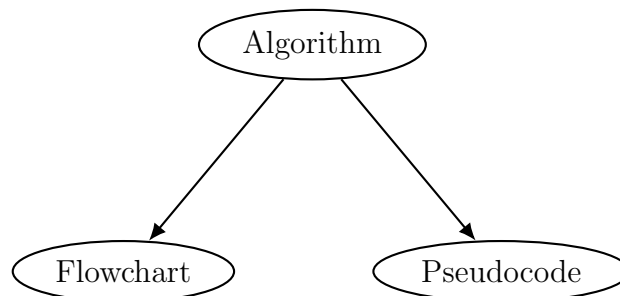
- **Exploring Multiple Solutions:**

1. Read numbers one by one from the user and calculate the sum.
2. Store numbers in an array, then use a loop to compute the sum.
3. Use a built-in function (if available) to directly compute the mean.

- **Selecting the Best Solution:** For basic implementation, reading numbers in a loop and computing the sum, then dividing by  $n$ , is efficient and straightforward.

## 1.4 Algorithm




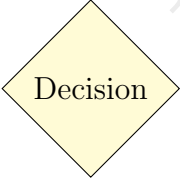

An algorithm is a finite, step-by-step set of instructions designed to solve a specific problem or perform a task. It must be unambiguous, effective, and terminate after a finite number of steps.



**Flowchart:** A flowchart is a visual representation of an algorithm using standardized symbols such as ovals, rectangles, diamonds, and arrows to illustrate the flow of control and operations in a program or process.

**Pseudocode:** Pseudocode is a high-level, informal description of an algorithm using english-like languages but without strict syntax rules. It is used to plan and explain logic before actual coding.

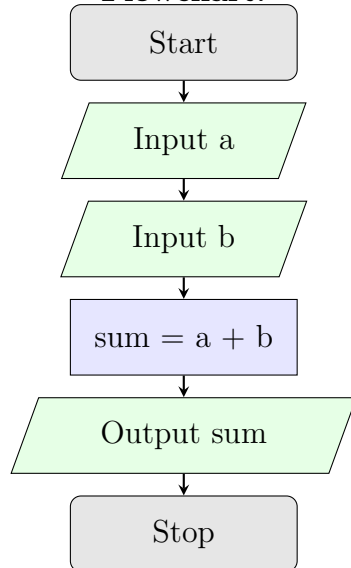
Table 1.2: Common Flowchart Symbols and Their Meanings

Symbol	Name	Description
	Terminal	Represents the start or end of a flowchart.
	Input/Output	Represents input to or output from a process.
	Process	Represents a process, operation, or action to be performed.
	Decision	Represents a decision-making step; used to show branching.
	Flowline	Shows the direction or flow of the process.

**Example 1:** Write the flowchart and algorithm/pseudocode to add two numbers.

### Addition of Two Numbers

**Flowchart:**



**Algorithm**

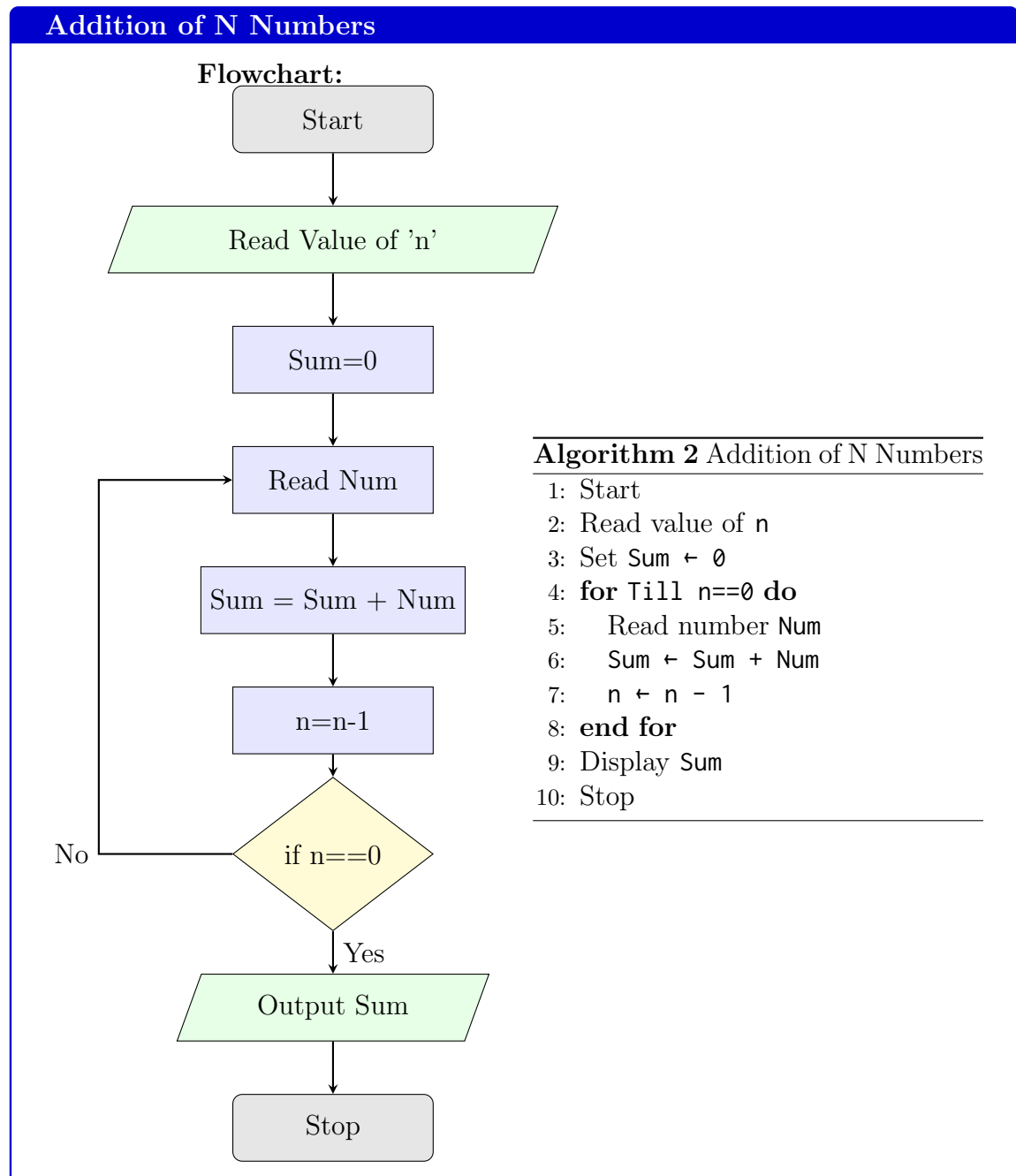
---

**Algorithm 1** Addition of Two Numbers

---

- 1: Start
  - 2: Read number a
  - 3: Read number b
  - 4: Compute  $\text{sum} \leftarrow a + b$
  - 5: Display sum
  - 6: Stop
-

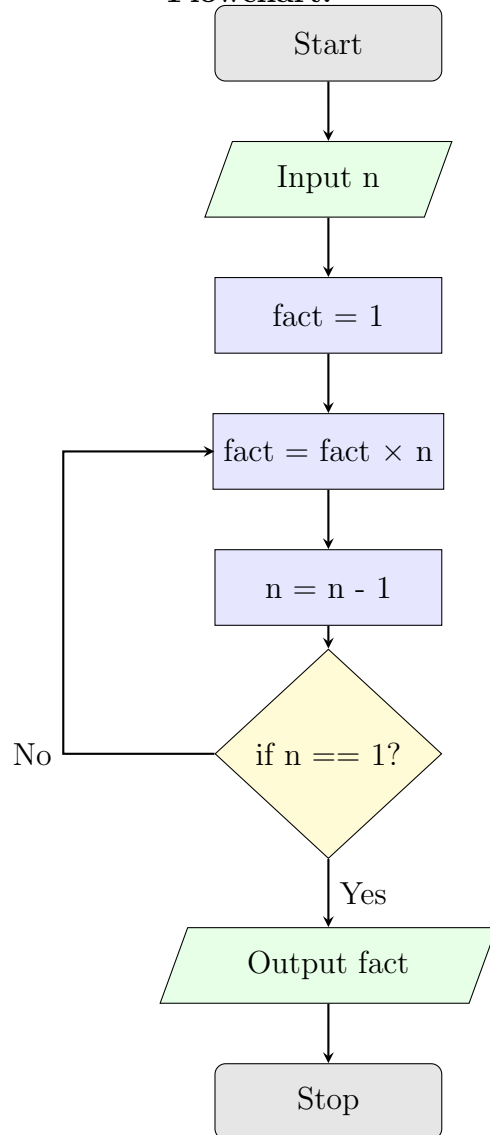
**Example 2:** Write the flowchart and algorithm to add n numbers.



**Example 3:** Flowchart and algorithm for factorial of a number.

### Factorial:

#### Flowchart:



#### Algorithm

---

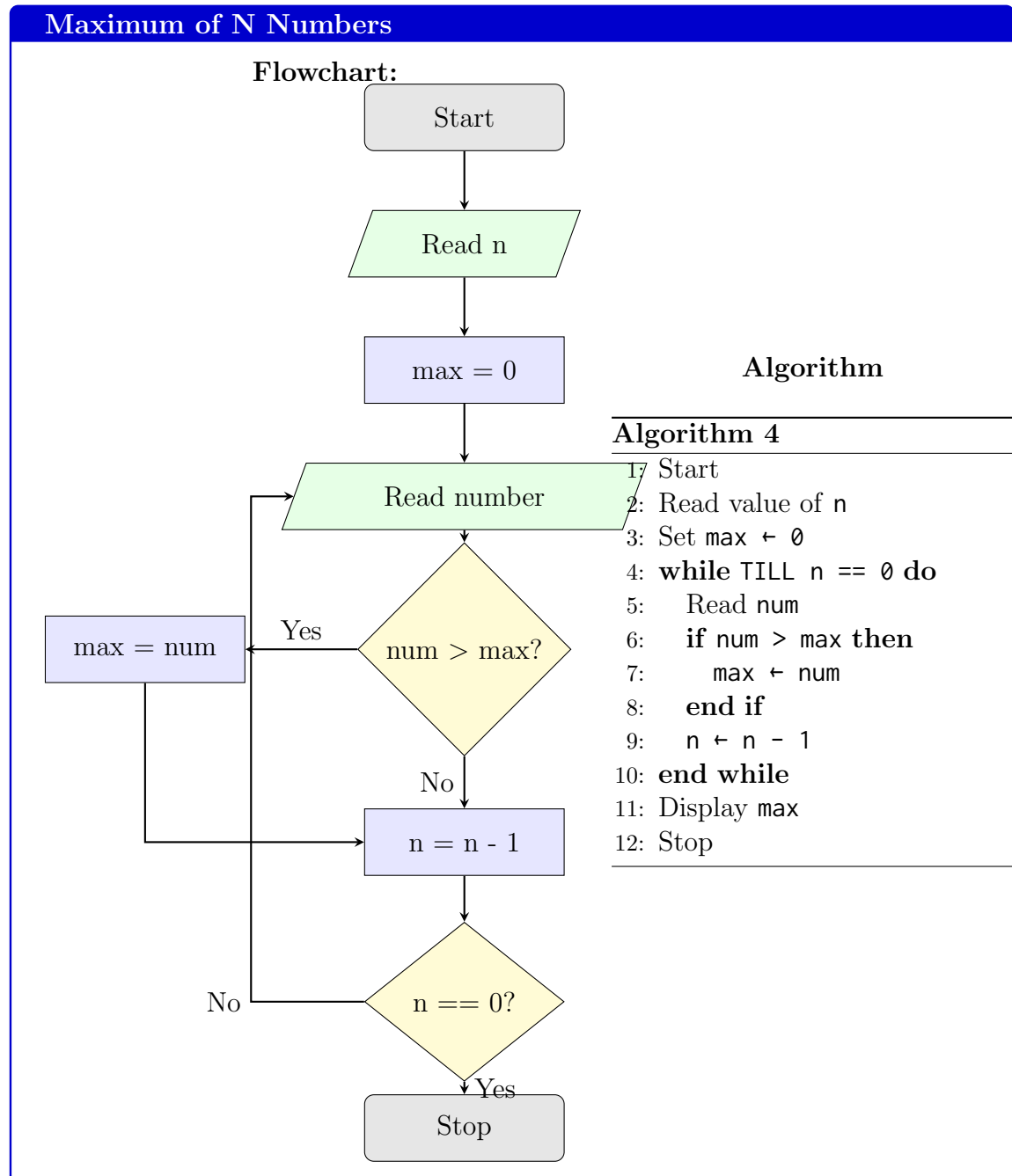
**Algorithm 3** Factorial of a Number

---

```
1: Start
2: Read the value of n
3: for Till n == 1 do
4:   fact ← fact × n
5:   n ← n - 1
6: end for
7: Display the fact
8: Stop
```

---

**Example 4:** Write the flowchart and algorithm to find the maximum of N numbers.



**Exercises****1. Check whether a number is positive, negative, or zero.**

Write an algorithm and draw a flowchart to check whether a given number is:

- Positive
- Negative
- Zero

**2. Check whether a number is divisible by both 5 and 11.**

Design a flowchart and algorithm to read an integer and determine whether it is divisible by both 5 and 11.

**3. Check whether a number is prime.**

Write an algorithm and draw a flowchart to check whether a given number is a prime number or not.

**4. Area of a circle.**

Write an algorithm and draw a flowchart

**5. Area of a rectangle.**

Write an algorithm and draw a flowchart

## 1.5 Compilation Model

Compilation is the process of translating the human-readable source code written in a programming language (like C) into machine readable instructions (binary code) that the computer can execute. It is done in multiple stages are pre-processing, compiling, assembling, and linking to produce the final executable file.

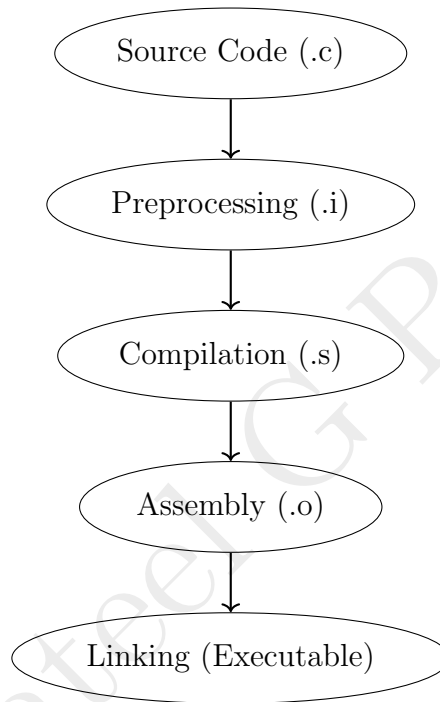


Figure 1.3: Compilation Process in C Programming

1. **Source Code (.c)** Written in C using any text editor. This is the human-readable form of the program.  
Example: `nano hello.c`
2. **Preprocessor (.i)** Handles macros, header file inclusion, and comment removal. Output is expanded source code in plain text.  
Command: `gcc -E hello.c -o hello.i`
3. **Compiler (.s)** Translates preprocessed code into assembly language. Assembly is CPU-specific but still human-readable.  
Command: `gcc -S hello.i -o hello.s`



4. **Assembler (.o)** Converts assembly code into machine code (object file). Output is binary and not human-readable.  
Command: `gcc -c hello.s -o hello.o`
5. **Linker (Executable)** Links object files and required libraries to make the final program. Produces a ready-to-run executable file.  
Command: `gcc hello.o -o hello`

## 1.6 History and Background of C

- C was developed at Bell Laboratories as a general-purpose systems programming language.
- It played a key role in the development of the UNIX<sup>™</sup> operating system and gained popularity with UNIX<sup>™</sup>.
- C is a third-generation high-level language that produces fast and efficient code.
- It is sometimes called a "low-level high-level language" because:
  - It has control structures (like `if`, `while`) and structured data types (arrays, records) found in high-level languages.
  - It also provides low-level features such as bit manipulation and register variables.
- Compared to assembly language, C is more portable and easier to maintain.
- However, its syntax can be less readable compared to many other high-level languages.

## 1.7 Structure of a C Program

A C program is typically divided into the following sections:

1. **Documentation Section** This section contains comments that describe the purpose of the program, author details, date, and other relevant notes.  
Example: `/* Program: Addition of Three Numbers */`

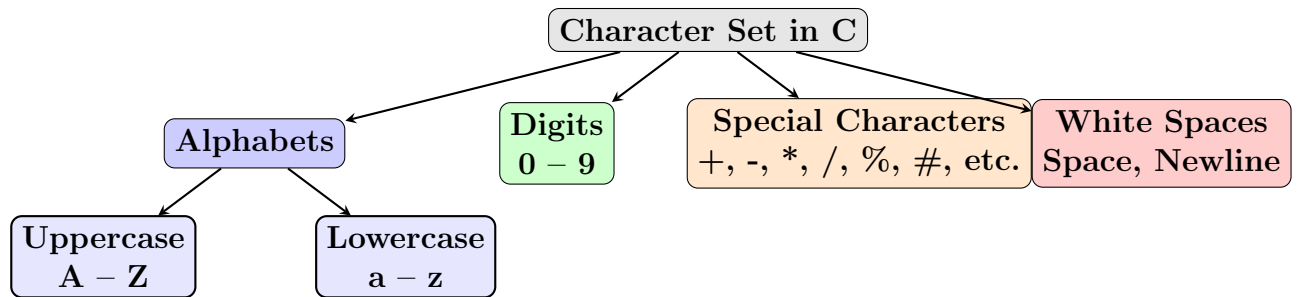
2. **Header Section** Includes standard library header files required for the program.  
Example: `#include <stdio.h>` is used for input and output functions.
3. **Definition Section** Contains symbolic constants and macros defined using `#define` or `const`.  
Example: `#define C 30` fixes a constant value for C.
4. **Global Declaration Section** Declares variables and function prototypes that are accessible throughout the program.  
Example: `int calcSum(int a, int b, int c);`
5. **Main Section** This is the entry point of the program where execution begins. It usually contains:
  - **Local Declaration Section** – Declares variables local to `main`.
  - **Executable Section** – Contains actual statements to perform the task.
6. **Subprogram Section** Contains user-defined functions that perform specific tasks and can be reused in the program.  
Example: `int calcSum(int a, int b, int c)` adds three numbers.

**Example:**

```
1  /* Documentation Section
2     Program: Addition of Three Numbers
3     Author  : Pateel G P
4     Date   : 11-08-2025
5     Purpose: Demonstrate C program structure with function and #define
6  */
7
8  // Header Section
9  #include <stdio.h>
10
11 // Definition Section
12 #define C 30    // constant number
13
14 // Global Declaration Section
15 int calcSum(int a, int b, int c);
16
17 // Main Section
18 int main() {
19
20     // Local Declaration Section
21     int a, b, sum;
22
23     // Executable Section
24     printf("Enter two numbers: ");
25     scanf("%d%d", &a, &b);
26
27     sum = calcSum(a, b, C);
28     printf("Sum of three numbers: %d\n", sum);
29
30     return 0;
31 }
32
33 // Subprogram Section
34 int calcSum(int a, int b, int c) {
35     return a + b + c;
36 }
```

## 1.8 Character Set in C

In C programming, a character set refers to all the characters that can be used to write a program. These are the basic building blocks for creating tokens, identifiers, and other program elements.



## 2. Digits

- Decimal digits: 0 to 9
- Used in numeric constants, array indices, etc.

## 3. White Spaces

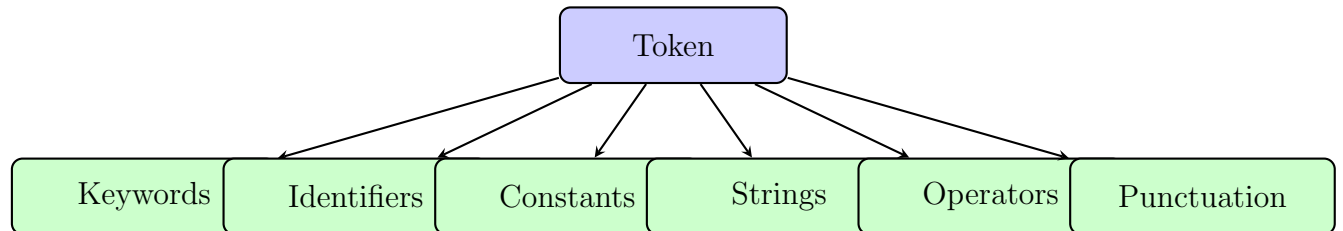
- Space, Tab (`\t`), Newline (`\n`)
- Used to separate tokens and improve readability
- `\n` : Newline
- `\t` : Horizontal Tab
- `\\` : Backslash
- `\'` : Single Quote
- `\"` : Double Quote

## 4. Special Characters

- `+ - * / % = < > &`
- `! ~ ^ ? : ; , . # _ ( ) { } [ ]`
- Used for operators, punctuation, and syntax

## 1.9 C Tokens

In C programming, tokens are the smallest units that the compiler recognizes. The types of tokens in C are:



- **Keywords**

Reserved words with special meaning in the language syntax. They cannot be used as variable or function names.

*Example:* `int`, `if`, `return`

- **Identifiers**

Names given by the programmer to represent variables, functions, arrays, etc. Must start with a letter or underscore, followed by letters, digits, or underscores.

*Example:* `sum`, `total_marks`

- **Constants**

Fixed values that remain unchanged during program execution. Includes integer, floating-point, character, and string constants.

*Example:* `100`, `3.14`, `'A'`, `"Hello"`

- **Strings**

A sequence of characters enclosed in double quotes, representing text.

*Example:* `"C Programming"`

- **Operators**

Symbols that perform operations on variables and values, such as arithmetic, logical, relational, and bitwise operations.

*Example:* `+`, `-`, `*`, `/`, `&&`, `==`

- **Punctuation / Special Symbols**

Characters used for grouping, separating, and structuring the program syntax.

*Example:* `{`, `}`, `;`, `,`

### 1.9.1 Keywords:

Keywords are reserved words that have a special meaning in the C programming language. They form the core syntax and structure of C programs. Since keywords are predefined by the language, programmers cannot use them as names for variables, functions, or identifiers. Each keyword performs a specific function or denotes a particular data type, control structure, or other language feature.

Keywords help the compiler recognize the instructions and the flow of the program. Using keywords incorrectly (e.g., as variable names) will result in compilation errors.

#### Complete List of Keywords in C (ANSI C Standard):

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			

### 1.9.2 Identifiers

Identifiers are names given by the programmer to represent variables, functions, arrays, or other user-defined elements in a C program. They are used to uniquely identify these entities in the source code.

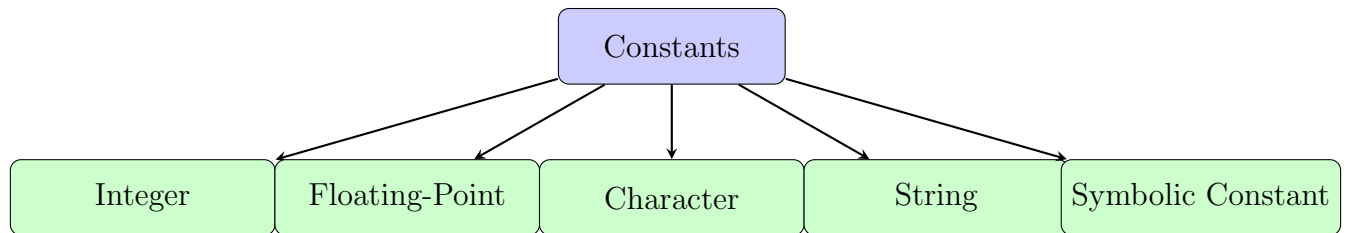
#### Rules for Identifiers:

- Must begin with a letter (A-Z or a-z) or an underscore (\_).
- Can contain letters, digits (0-9), and underscores.
- Cannot be the same as any C keyword.
- Case sensitive (e.g., `sum` and `Sum` are different).

**Examples:** `sum`, `total_marks`, `_index`, `counter1`, `calculateAverage`

### 1.9.3 Constants

Constants are fixed values that do not change during the execution of a program. They represent literal data in the source code. Different types of constants serve different purposes in C programming.



### 1. Integer Constants

- Integer constants represent whole numbers without any fractional or decimal parts.
- They can be positive or negative.
- Can be specified in decimal (base 10), octal (base 8, starting with 0), or hexadecimal (base 16, starting with 0x) formats.
- **Examples:** 100, -25, 0, 075 (octal), 0xFF (hexadecimal)

### 2. Floating-Point Constants

- Floating-point constants represent real numbers that have fractional parts.
- They are written with a decimal point or in exponential notation.
- Used to approximate real values in calculations.
- **Examples:** 3.14, -0.001, 2.0, 1.5e3 (scientific notation for 1500)

### 3. Character Constants

- Character constants represent single characters enclosed in single quotes.
- Internally stored as integer ASCII values.
- Special characters like newline and tab are represented with escape sequences.
- **Examples:** 'A', 'z', '\n' (newline), '\t' (tab)

#### 4. String Constants

- String constants (or string literals) are sequences of characters enclosed in double quotes.
- Stored as arrays of characters terminated by a null character `'\0'`.
- Used for text manipulation and display.
- **Examples:** "Hello", "C Programming", "12345"

#### 5. Symbolic Constants

Symbolic constants are programmer-defined names that represent fixed values throughout a program. They improve code readability and maintainability by replacing literal constants with meaningful identifiers.

In C programming, symbolic constants can be defined using:

- **The `#define` preprocessor directive:** This replaces occurrences of the name with the specified value during the preprocessing stage.

```
#define MAX_SIZE 100
```

Here, `MAX_SIZE` acts as a symbolic constant for the value `100`.

- **The `const` qualifier:** This declares a variable whose value cannot be modified after initialization.

```
const int MAX_SIZE = 100;
```

In this case, `MAX_SIZE` is a constant integer variable with value `100`.

Using symbolic constants allows programmers to avoid scattering literal values throughout the code, making updates easier and reducing errors.

**Example:**



```
1 #define PI 3.14159
2
3 const int MAX_STUDENTS = 50;
4
5 float circle_area(float radius)
6 {
7     return PI * radius * radius;
8 }
```

### 1.9.4 Strings

In C, a string is a sequence of characters stored as an array of type `char`. Strings are always terminated by a special null character `'\0'` which indicates the end of the string.

#### Characteristics:

- Strings are enclosed in double quotes, e.g., `"Hello"`.
- Internally stored as a contiguous sequence of characters followed by the null terminator.
- The length of the string is the number of characters before the null terminator.
- Strings can be manipulated using standard library functions such as `strcpy()`, `strlen()`, `strcat()`, etc.

#### Example:

```
char greeting[] = "Hello, World!";
```

**Note:** The null character `'\0'` is automatically added by the compiler at the end of string literals.

**Usage:** Strings are widely used to store and manipulate text in C programs.

### 1.9.5 Punctuation / Special Symbols

Punctuation and special symbols in C are characters that have syntactic significance and are used to organize the program's structure, separate statements, or perform specific operations. They are essential for the correct parsing and execution of C code.

#### Common Punctuation and Special Symbols:

- ; — Statement terminator (ends a statement)
- {, } — Curly braces used to define blocks of code (e.g., function bodies, loops)
- (), parentheses used for function calls, grouping expressions, and parameters
- [], square brackets used for array subscripting
- , — Comma used to separate expressions or variables
- # — Preprocessor directive symbol
- :: (in C++ but not C)
- . — Member access operator (accessing members of a structure or union)
- -> — Pointer member access operator (accessing members via pointer)

**Usage:** These symbols organize the structure of the program, separate instructions, and provide access to complex data types.

**Example:**

```
1 int main()  
2 {  
3     int arr[5];  
4     arr[0] = 10;  
5     return 0;  
6 }
```

### 1.9.6 Variables

Variables are the name given by the user to store data that can be modified during program execution. Each variable has a specific data type which determines the size and layout of its memory.

**Variable Declaration:** Declaring a variable specifies its type and name, allowing the compiler to allocate memory.

**Syntax:** data\_type variable\_name;

**Example:**

```
int age;  
float temperature;  
char grade;
```

**Variable Initialization:** Assigning an initial value to a variable at declaration or later helps avoid unpredictable behavior.

**Syntax:** data\_type variable\_name = value;

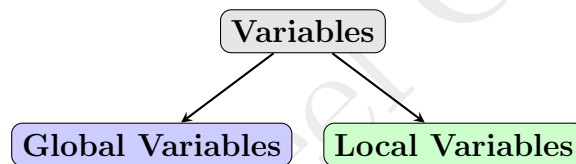
**Example:**

```
1 int age = 25;  
2 float temperature = 36.5;  
3 char grade = 'A';
```

Variables can also be assigned new values after declaration:

```
1 age = 30;  
2 grade = 'B';
```

#### 1.9.6.1 Local and Global Variables:



**Global Variables:**

- Declared outside all functions, usually at the top of the program.
- Accessible from any function within the program.
- Exist throughout the program's execution.
- Useful for sharing data between multiple functions.

**Example:**

```
1 int count = 0; // Global variable  
2  
3 void increment() {  
4     count++; // Accessing global variable  
5 }
```

**Local Variables:**

- Declared inside a function or block.
- Accessible only within that function or block.
- Created when the function is called and destroyed when it returns.
- Used for temporary storage during function execution.

**Example:**

```
1 void increment() {  
2     int temp = 0; // Local variable  
3     temp++;  
4 }
```

**Notes:**

- Variables must be declared before use.
- Initialization is recommended to prevent unpredictable values.

**Usage:** Variables are fundamental for storing and manipulating data dynamically during program execution.

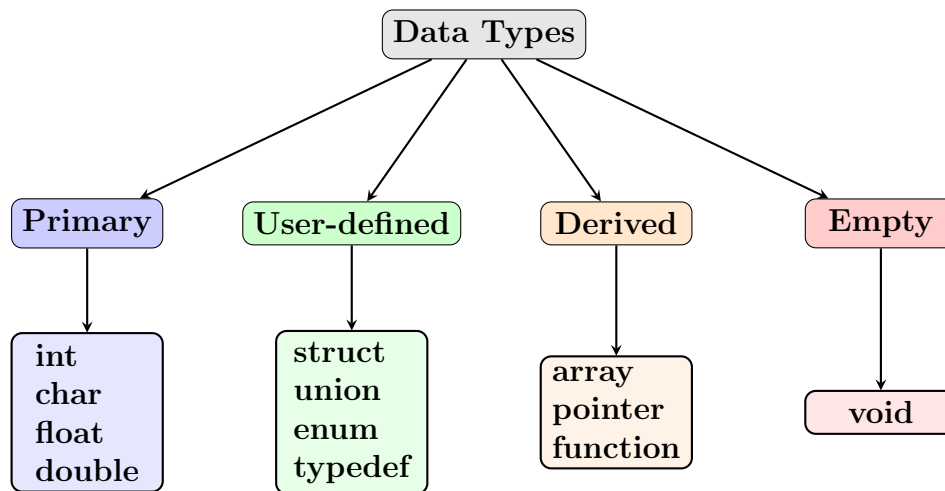
**Example:**

```
1 char letter;  
2 int overtime, day_of_month, UB40;  
3 signed short int salary;  
4 signed short salary;  
5 short int salary;  
6 short salary;  
7 unsigned long hours;  
8 float sigma_squared, X_times_2;
```

The 3rd, 4th, 5th, and 6th examples are equivalent since **signed** and **int** are assumed if not specified.

## 1.9.7 Data Types

Data types specify the type of data a variable can hold and determine the amount of memory allocated to it. C language provides several built-in data types to store different kinds of data.



#### 1.9.7.1 Primary Data Types

- **int:** Stores integer values (whole numbers) without decimals. Size is typically 2 or 4 bytes depending on the system.  
**Example:** `int age = 25;`
- **float:** Stores single precision floating-point numbers (decimal values). Size is typically 4 bytes.  
**Example:** `float temperature = 36.5;`
- **double:** Stores double precision floating-point numbers, with higher precision and range than `float`. Size is typically 8 bytes.  
**Example:** `double pi = 3.14159;`
- **char:** Stores a single character. Size is 1 byte. Characters are stored as ASCII codes internally.  
**Example:** `char grade = 'A';`

#### 1.9.7.2 Derived Data Types

- **Arrays:** Collection of elements of the same data type stored in contiguous memory locations.  
**Example:** `int marks[5];`
- **Pointers:** Variables that store the address of other variables.  
**Example:** `int *ptr;`

- **Structures:** User-defined data types that group variables of different data types under one name.

**Example:** `struct Student { int id; char name[20]; };`

- **Unions:** Similar to structures but store different data types in the same memory location.

**Example:** `union Data { int i; float f; };`

### 1.9.7.3 Qualifier Data Types

Qualifiers modify the meaning and size of the basic data types.

- **signed:** (default for `int`) Can store positive and negative values.
- **unsigned:** Stores only non-negative values, effectively doubling the positive range.
- **short:** Smaller size integer, typically 2 bytes.
- **long:** Larger size integer, typically 4 or 8 bytes.
- **long long:** Extended integer size, typically 8 bytes.

#### 1.9.7.4 Size and Range of Data Types

##### (a) 16-bit Architecture (Turbo C )

Data Type	Size (bytes)	Range
char	1	-128 to 127 (signed)
unsigned char	1	0 to 255
int	2	-32,768 to 32,767
unsigned int	2	0 to 65,535
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
float	4	$\pm 3.4e\pm 38$
double	8	$\pm 1.7e\pm 308$

##### (b) 32-bit Architecture (Modern GCC/Clang)

Data Type	Size (bytes)	Range
char	1	-128 to 127 (signed)
unsigned char	1	0 to 255
int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
long	4	Same as int
unsigned long	4	Same as unsigned int
long long	8	Very large range
float	4	$\pm 3.4e\pm 38$
double	8	$\pm 1.7e\pm 308$

**Note:** Sizes may vary depending on the compiler and system architecture.

#### Example Declarations

```

1 int count = 10;
2 unsigned int age = 25;
3 float pi = 3.14f;
4 double e = 2.718281828;
5 char letter = 'A';
6 long long bigNumber = 9223372036854775807;

```

## Sample C Program with Token and Variable Explanation

```
1 #include <stdio.h>
2
3 int main() {
4     int num1, num2, sum;
5
6     printf("Enter two numbers: ");
7     scanf("%d%d", &num1, &num2);
8
9     sum = num1 + num2;
10
11    printf("Sum = %d\n", sum);
12
13    return 0;
14 }
```

### Token Classification:

- **Keywords:** int, return
- **Identifiers:** main, num1, num2, sum, printf, scanf
- **Operators:** =, +, &
- **Punctuators:** (, ), {, }, ,, " "

### Data Types:

- **int** – integer data type used for variables num1, num2, and sum

### Variables:

- num1, num2, and sum are variables declared to store integer values.
- These variables are local to the main() function.
- Variables hold user input and the result of the addition operation.
- Variables are declared using the int data type indicating they store integer values.



## Sample C Program with Explanation of Tokens

```
1  /* Program to calculate area of a circle */
2
3  #include <stdio.h>
4
5  #define PI 3.14159
6
7  int main() {
8      float radius;
9      float area;
10
11     printf("Enter radius: ");
12     scanf("%f", &radius);
13
14     area = PI * radius * radius;
15
16     printf("Area = %.2f\n", area);
17     return 0;
18 }
```

### Explanation of tokens used:

- **Keywords:** int, return, float
- **Identifiers:** main, radius, area, printf, scanf
- **Constants:** PI (symbolic constant), numeric literals 3.14159, 0
- **Operators:** =, \*, &
- **Punctuators:** ();, {}, ,, " "

**Exercise****Identify Illegal Data Definition Statements:**

Which of the following data definition statements are illegal in C? Explain why.

```
int a, b = 4, x = 4.3, yz = 'D';  
char c = 257, d, 3rd_var;  
short Default, Default_Value, default, default2;  
long int, e = 123456789;  
float f = 123.456, g = 100 e-6;  
unsigned h = -1;  
const int i = 4, j, k = 0;  
unsigned float l = 1.234;
```

**Exercise**

Consider the following data definitions inside the `main` function:

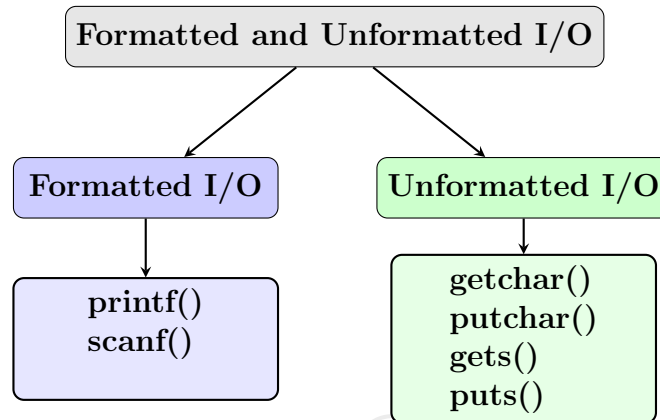
```
int a, b = 4, c;  
char letter = 'A', digit = '1';  
float x = 258.34, y;  
unsigned val;  
const char q = 'A';
```

Identify which of the following assignments are illegal and explain why. Also, for the legal assignments, state the new value of the assigned variable:

```
b = a;  
a = 0xAda;  
b = letter;  
a = digit;  
b = c;  
digit = 66;  
digit = '\102';  
digit = 0102;  
letter = '\9';  
letter = "A";  
letter = q;  
x = 12.345;  
x = 1234.5 e-2;  
y = x;  
val = 0177777U;  
q = 100;  
q = 'Q';
```

## 1.10 Formatted and Unformatted Input/Output

In C programming, input/output (I/O) operations are performed using library functions declared in the `stdio.h` header file. I/O functions can be classified into two categories:



- Formatted I/O Functions
- Unformatted I/O Functions

### 1. Formatted Input/Output

Formatted I/O functions allow the programmer to control the way data is read from or written to the standard input/output device. They use *format specifiers* (e.g., `%d`, `%f`, `%c`, `%s`) to specify the type and layout of data.

#### Common Functions:

- `printf()`: Displays output in a formatted way.
- `scanf()`: Reads input according to a specified format.

#### Examples of Formatted Input/Output in C

##### Example 1: Integer Output

```
1 #include <stdio.h>
2 int main() {
3     int num = 42;
4     printf("Number: %d\n", num);
```

```
5     return 0;
6 }
```

**Output:**

Number: 42

**Example 2: String Output**

```
1 #include <stdio.h>
2 int main() {
3     char name[] = "Alice";
4     printf("Hello, %s!\n", name);
5     return 0;
6 }
```

**Output:**

Hello, Alice!

**Example 3: Multiple Variables in One Statement**

```
1 #include <stdio.h>
2 int main() {
3     int age = 25;
4     float marks = 88.75;
5     printf("Age: %d, Marks: %.1f\n", age, marks);
6     return 0;
7 }
```

**Output:**

Age: 25, Marks: 88.8

**Example 4: Reading Input using scanf**

```
1 #include <stdio.h>
2 int main() {
3     int age;
4     char name[20];
5     printf("Enter your name: ");
6     scanf("%s", name);
7     printf("Enter your age: ");
8     scanf("%d", &age);
9     printf("Name: %s, Age: %d\n", name, age);
10    return 0;
11 }
```

**Sample Input/Output:**

```
1 Enter your name: John
2 Enter your age: 30
3 Name: John, Age: 30
```

## 2. Unformatted Input/Output

**Definition:** Unformatted I/O functions handle data without using format specifiers. They are generally used for single characters or strings. They are simpler but offer less control over formatting.

**Common Functions:**

- `getchar()` — Reads a single character from input.
- `putchar()` — Writes a single character to output.
- `gets()` — Reads a string until a newline (deprecated; use `fgets()` instead).
- `puts()` — Outputs a string followed by a newline.

**Example:**

```
1 #include <stdio.h>
2 int main() {
3     char ch;
4     char name[30];
5
6     printf("Enter a character: ");
7     ch = getchar(); // Reads a single character
8
9     printf("You entered: ");
10    putchar(ch); // Prints the character
11    printf("\n");
12
13    printf("Enter your name: ");
14    fgets(name, sizeof(name), stdin); // Reads a string safely
15
16    printf("Hello, ");
17    puts(name); // Prints string with newline
18
19    return 0;
20 }
```

**Sample Input/Output:**

```
1 Enter a character: P
2 You entered: P
3 Enter your name: Patil
4 Hello, Patil
```

**Key Points:**

- `getchar()` reads exactly one character at a time.
- `putchar()` displays exactly one character.
- `puts()` automatically moves to the next line after printing.
- `gets()` is unsafe and should not be used.

## Example of Format Specifiers in C

Specifier	Data Type / Use	Example Code	Output
%d	Signed int (decimal)	int x = -42; printf("%d", x);	-42
%u	Unsigned int	unsigned x = 42; printf("%u", x);	42
%c	Character	char c = 'A'; printf("%c", c);	A
%s	String	char s[] = "Hi"; printf("%s", s);	Hi
%f	Float / double (6 decimals default)	printf("%f", 3.1);	3.100000
%.2f	Float with 2 decimals	printf("%.2f", 3.1);	3.10
%8.2f	Float, width=8, 2 decimals	printf("%8.2f", 3.1);	3.10
%08.2f	Float, width=8, zero padded	printf("%08.2f", 3.1);	00003.10
%e	Float (scientific, lowercase)	printf("%e", 3.1);	3.100000e+00
%E	Float (scientific, uppercase)	printf("%E", 3.1);	3.100000E+00
%x	Unsigned int (hex lowercase)	int n=255; printf("%x", n);	ff
%X	Unsigned int (hex uppercase)	int n=255; printf("%X", n);	FF
%o	Unsigned int (octal)	int n=10; printf("%o", n);	12