Flowchart and Algorithm:

Flowchart :

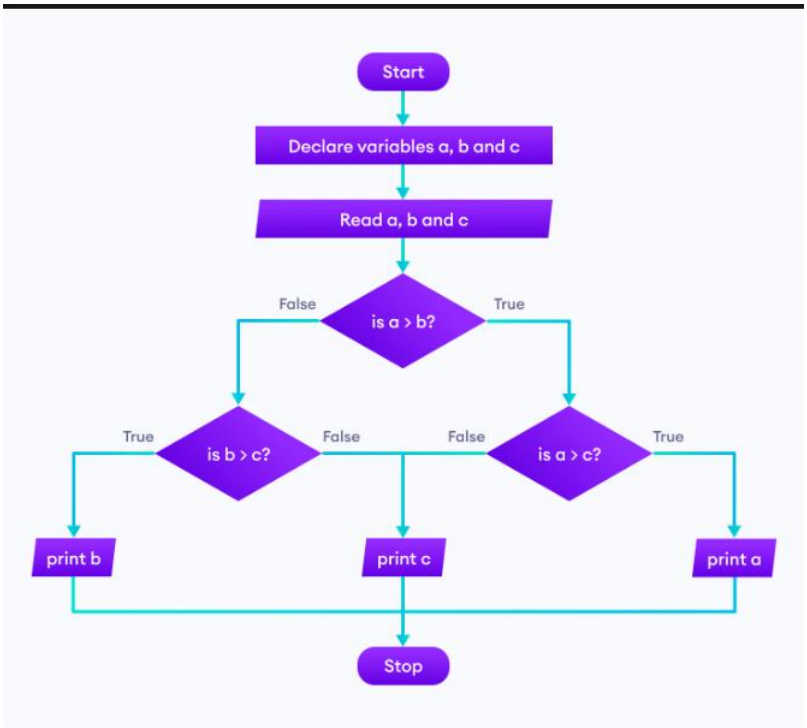
Introduction

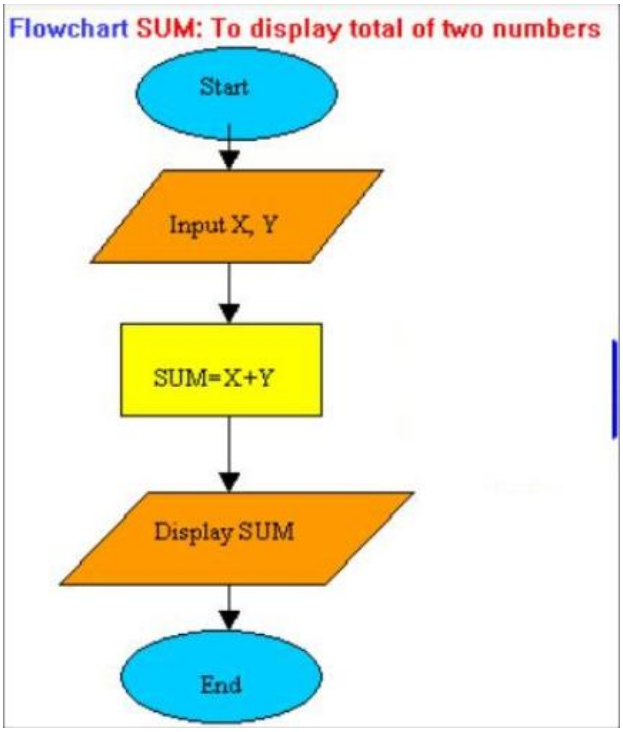
A flowchart is a graphical representation of an algorithm or a process. It visually illustrates the sequence of steps, decisions, and operations involved in solving a problem or executing a task. Flowcharts use standardized symbols to represent different types of actions or steps (e.g., ovals for start/end, rectangles for processes, diamonds for decisions) and connecting arrows to indicate the flow of control and information. Flowcharts serve as a visual aid for understanding, designing, documenting, and analyzing algorithms and processes, making complex logic more accessible and easier to follow, especially for beginners in programming. They are a powerful tool for planning and visualizing program logic before writing actual code.

- **Advantages of using a Flowchart**
    - Flowcharts are a better way of communicating the logic of the system.
    - Flowcharts act as a guide for blueprint during program designed.
    - Flowcharts help in debugging process.
    - With the help of flowcharts programs can be easily analyzed.
    - They provide better documentation.
    - Flowcharts serve as a good proper documentation.

- **Disadvantages of using a Flowchart**
    - It is difficult to draw flowcharts for large and complex programs.
    - There is no standard to determine the amount of detail.
    - It is very difficult to modify the Flowchart.
    - Making a flowchart is costly.
    - If changes are done in software, then the flowchart must be redrawn

Examples of flowchart





- **Algorithm:**

An algorithm is a finite set of well-defined, unambiguous instructions or a step-by-step procedure for solving a computational problem or achieving a specific task. Algorithms are fundamental to computer science and programming, providing the logical blueprint for how a program will process data and produce results. Key properties of algorithms include being finite (terminating after a finite number of steps), definite (each step is precisely defined), having inputs,

producing outputs related to those inputs, and being effective (each step can be performed in a finite amount of time).

- **Advantages of Algorithms:**
    - It is easy to understand.
    - An algorithm is a step-wise representation of a solution to a given problem.
    - In an Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

- **Disadvantages of Algorithms:**
    - Writing an algorithm takes a long time so it is time-consuming.
    - Understanding complex logic through algorithms can be very difficult.
    - Branching and Looping statements are difficult to show in Algorithms(imp).

- **Developing and writing algorithm using pseudo codes**

1. What is an Algorithm?

- An algorithm is a step-by-step procedure to solve a problem or perform a task.
- It's like a recipe in cooking: a set of instructions that leads to the desired result.

2. What is Pseudocode?

- Pseudocode is a way to describe an algorithm in plain, structured English (not actual programming code).
- It's not bound to any programming language.
- It helps you think logically before coding.

4. Steps in Developing an Algorithm

When you want to write an algorithm:

1. Understand the problem → What is the input? What is the expected output?
2. Plan the steps → Break down the solution into logical steps.

3. Write in pseudocode → Use simple keywords like START, IF, WHILE, PRINT.
4. Check correctness → Dry-run with examples to verify.
5. Convert to real code later (in C, Python, Java, Dart, etc.).

4. Example Problem & Pseudocode

Problem: Find the largest of two numbers.

Algorithm steps:

1. Start
2. Input two numbers
3. Compare the numbers
4. Print the larger number
5. Stop

Pseudocode:

```
START
  READ number1, number2
  IF number1 > number2 THEN
     PRINT number1
  ELSE
     PRINT number2
  ENDIF
STOP
```

Algorithm

1. What is an Algorithm?

An algorithm is a clear, step-by-step method for solving a problem or performing a task.

Think of it as a recipe: it defines the sequence of actions needed to get the final result.

---

2. What is Pseudocode?

Pseudocode is a simplified, language-agnostic way of describing an algorithm using plain English (with some programming-like structure).

It ignores syntax details of a programming language.

It helps you plan before coding.

---

3. Steps in Developing an Algorithm with Pseudocode

Understand the Problem – Identify inputs, outputs, and requirements.

Design the Solution – Break the problem into small logical steps.

Write Pseudocode – Express steps using simple keywords like READ, IF, WHILE, PRINT.

Validate – Dry-run with sample data to confirm correctness.

Translate into Code – Convert pseudocode into a real programming language.

---

4. Advantages of Using Pseudocode

Easy to read & understand – No coding knowledge required.
Focuses on logic – No distractions with syntax errors.
Can be written quickly – Faster than actual coding.
Helps in teamwork – Programmers and non-programmers can communicate.
Language-independent – Can later be converted to any programming language.

---

5. Disadvantages of Using Pseudocode

Not executable – You can't run pseudocode directly.
No strict standard – Different people may write pseudocode differently.
May be ambiguous – Lack of syntax rules can lead to misinterpretation.
Still needs real code – It's only a planning tool, not the final product.

---

6. Examples

Example 1: Find the Largest of Two Numbers

Algorithm:

Start

Read two numbers

Compare them

Print the larger number

Stop

Pseudocode:

```
START
  READ number1, number2
  IF number1 > number2 THEN
     PRINT number1
  ELSE
     PRINT number2
  ENDIF
STOP
```

- **Constants , variables and DataTypes:**

1. Character Set

Before understanding identifiers and tokens, you need to know what characters are valid in your C source.

Source character set: all the characters that can appear in the program's source file. This includes letters (A-Z, a-z), digits (0-9), punctuation, whitespace (space, tab, newline), special symbols, etc. The standard requires a *basic character set*, plus possibly *extended characters*.

Execution character set: the set of characters the compiled program uses when it runs, especially for character constants / string literals etc. After preprocessing, string/character constants from the source set get mapped to the execution set.

Hence:

Not every encoding is allowed, but implementations usually accept ASCII or UTF-8 etc.

When naming identifiers, using characters outside the basic source set may lead to implementation-defined behaviour.

---

2. Tokens, Keywords, Identifiers

Tokens

Tokens are the smallest meaningful building blocks in a C program. The compiler breaks your source into tokens. types of tokens include: punctuators, keywords, strings (literals), operators, identifiers, constants.

Examples of tokens:

int, if, return (keywords)

x, sum, count (identifiers)

123, 3.14, 'a', "hello" (constants / literals)

Symbols like (,), {, }, ;, , etc. (punctuators)

Operators: +, -, *, /, %, ==, <, &&, etc.

Keywords

Reserved words that have special meaning in C. Cannot be used as identifiers.

Examples: int, char, float, double, if, else, for, while, switch, return, struct, typedef, const, extern, static, etc.

The exact number of keywords depends on the C standard version (C90, C99, C11, C23 etc.).

Identifiers

Names you give to variables, functions, arrays, etc.

Rules:

Must start with a letter (A-Z or a-z) or underscore _.

After the first character, can have letters, digits (0-9), or underscore.

Cannot be a keyword.

Case-sensitive (MyVar is different from myvar).

Some older constraints: only first N characters considered (but modern compilers accept longer names).

---

3. Constants

Constants (or literals) are fixed values in the code. They don’t change during execution.

- **Types of constants / literals:**

Integer constants: e.g. 123, -45, 0, 07 (octal), 0x1A (hexadecimal).

Floating-point constants: e.g. 3.14, -0.001, 2.0e5 etc.

Character constants: a single character in single quotes, e.g. 'a', 'Z', '\n' etc. These are usually stored as integer types corresponding to the character encoding.

String constants (literals): sequence of characters in double quotes, e.g. "Hello, world!". These are arrays of char ending in a null character \0.

Named constants: using const keyword or preprocessor #define. More in section on symbolic constants.

---

4. Variables

Variables are named storage locations whose contents *can* change during execution.

Each variable has a type which determines what kind of data it can hold, and how much memory is needed.

---

5. Data Types

C is statically typed. That means you declare a variable's type at compile time, and once declared, its type cannot change.

Major built-in data types include:

Category	Examples	Description
Integer types	int, short, long, unsigned int, signed char etc.	Whole numbers (positive/negative/zero).
Character type	char	Single character, typically 1 byte.
Floating types	float, double, long double	Numbers with fractional part.
Void	void	Represents absence of type; used in functions returning no value, or pointers to unknown types etc.

Declaration of Variables & Assigning Values

Declaration

You tell the compiler: here is a variable of a certain type, with a certain name.

Syntax:

type variable_name;

or multiple:

type var1, var2, var3;

You can also declare and initialize together:

type variable_name = initial_value;

Example:

int age;

float salary;

char grade;

int x = 10;

char c = 'A';

- ## Managing Input and Output Operations:

Standard I/O library: functions for input/output are in stdio.h

Streams: stdin, stdout, stderr are standard input, output, error streams.

Formatted vs unformatted I/O:

Formatted I/O means using format specifiers—reading or writing values in particular formats.

Unformatted I/O involves simpler functions (often character or string based) without format specifiers.

Reading / Writing a Character

These are the basic character-level I/O functions.

Function	Description
getchar()	Reads one character from stdin. Blocks until input is available. Returns the character (as an int), or EOF.

Function	Description
putchar(int c)	Writes one character (the c) to stdout.

Formatted Input & Formatted Output

Formatted Output: printf() family

printf(const char *format, ...) prints to standard output using format specifiers

Format specifiers: %d / %i for integers, %f for floats, %lf for double, %c for character, %s for string, %u, %o, %x, etc.

Formatted Input: scanf() family

scanf(const char *format, ...) reads from stdin according to a format string. Variables must be passed by pointer (using &) so scanf can store into them.

Common specifiers are same as printf: %d, %f, %lf, %c, %s, etc.

Operators and Expressions in C Language

Introduction

Operators are special symbols that perform specific operations on one or more operands (values or variables). Expressions are combinations of operands and operators that evaluate to a single value. In C programming, operators are fundamental building blocks that allow you to manipulate data and control program flow.

An expression can be as simple as a single variable or constant, or as complex as a mathematical formula involving multiple operators and operands.

Arithmetic Operators

Arithmetic operators perform mathematical calculations on numeric operands.

Binary Arithmetic Operators:

+ (Addition): Adds two operands

- (Subtraction): Subtracts the second operand from the first

* (Multiplication): Multiplies two operands

/ (Division): Divides the first operand by the second

% (Modulo): Returns the remainder after division

Unary Arithmetic Operators:

+ (Unary plus): Indicates positive value

- (Unary minus): Negates the value

```
int a = 10, b = 3;
int sum = a + b;       // 13
int diff = a - b;      // 7
int product = a * b;  // 30
int quotient = a / b; // 3 (integer division)
int remainder = a % b; // 1
```

Relational Operators

Relational operators compare two operands and return either 1 (true) or 0 (false).

== (Equal to): Returns 1 if operands are equal

!= (Not equal to): Returns 1 if operands are not equal

> (Greater than): Returns 1 if left operand is greater

< (Less than): Returns 1 if left operand is smaller

>= (Greater than or equal to)

<= (Less than or equal to)

```
int x = 5, y = 8;
int result1 = (x == y);  // 0 (false)
int result2 = (x < y);   // 1 (true)
int result3 = (x >= 5);  // 1 (true)
```

Logical Operators

Logical operators perform logical operations on boolean expressions.

&& (Logical AND): Returns 1 if both operands are true

|| (Logical OR): Returns 1 if at least one operand is true

! (Logical NOT): Returns 1 if operand is false, 0 if operand is true

```
int a = 1, b = 0;
int result1 = a && b;  // 0 (false)
int result2 = a || b;  // 1 (true)
int result3 = !a;      // 0 (false)
```

Assignment Operators

Assignment operators assign values to variables.

Simple Assignment:

= (Assignment): Assigns right operand value to left operand

Compound Assignment Operators:

+= (Add and assign): a += b is equivalent to a = a + b

-= (Subtract and assign)

*= (Multiply and assign)

/= (Divide and assign)

%= (Modulo and assign)

```
int x = 10;
x += 5;  // x becomes 15
x -= 3;  // x becomes 12
x *= 2;  // x becomes 24
```

Increment and Decrement Operators

These operators increase or decrease the value of a variable by 1.

Increment Operator (++):

Pre-increment: ++variable (increment first, then use)

Post-increment: variable++ (use first, then increment)

Decrement Operator (--):

Pre-decrement: --variable (decrement first, then use)

Post-decrement: variable-- (use first, then decrement)

```
int a = 5;
int b = ++a;  // a becomes 6, b gets 6
int c = a--;  // c gets 6, a becomes 5
```

Conditional Operator (Ternary Operator)

The conditional operator ? : provides a shorthand way to write simple if-else statements.

Syntax: condition ? expression1 : expression2

```
int a = 10, b = 20;
int max = (a > b) ? a : b;  // max gets 20
```

Bitwise Operators

Bitwise operators perform operations on individual bits of operands.

& (Bitwise AND): Sets bit to 1 if both bits are 1

| (Bitwise OR): Sets bit to 1 if at least one bit is 1

^ (Bitwise XOR): Sets bit to 1 if bits are different

~ (Bitwise NOT): Inverts all bits

<< (Left shift): Shifts bits to the left

>> (Right shift): Shifts bits to the right

```
int a = 5;    // Binary: 0101
int b = 3;    // Binary: 0011
int result1 = a & b;  // 1 (Binary: 0001)
int result2 = a | b;  // 7 (Binary: 0111)
int result3 = a << 1; // 10 (Binary: 1010)
```

Arithmetic Expressions

Arithmetic expressions combine variables, constants, and arithmetic operators to produce numeric results. They can be simple or complex and follow mathematical rules.

```
int result = 2 * 3 + 4 * 5 - 6;  // Complex arithmetic expression
float average = (a + b + c) / 3.0;  // Using parentheses for clarity
```

Evaluation of Expressions

Expression evaluation follows specific rules:

Precedence: Operators with higher precedence are evaluated first

Associativity: When operators have the same precedence, associativity determines evaluation order

Parentheses: Override normal precedence and associativity rules

Short-circuit evaluation: In logical expressions, evaluation may stop early

Precedence of Arithmetic Operators

From highest to lowest precedence:

() (Parentheses)

+, - (Unary plus and minus)

*, /, % (Multiplication, division, modulo)

+, - (Addition and subtraction)

int result = 2 + 3 * 4; // Evaluates to 14, not 20

int result2 = (2 + 3) * 4; // Evaluates to 20

Type Conversions in Expressions

When operands of different types are used in expressions, C performs automatic type conversions following these rules:

Implicit Type Conversion (Type Promotion):

char and short are promoted to int

If one operand is double, the other is converted to double

If one operand is float, the other is converted to float

If one operand is unsigned long, the other is converted to unsigned long

int a = 5;

float b = 2.5;

float result = a + b; // 'a' is converted to float, result is 7.5

Explicit Type Conversion (Type Casting):

int a = 10, b = 3;

float result = (float)a / b; // Explicit casting to get 3.333...

Operator Precedence and Associativity

Complete precedence table (highest to lowest):

(), [], ->, . (Left to right)

!, ~, ++, --, +, -, *, &, sizeof (Right to left)

*, /, % (Left to right)

+, - (Left to right)

Evaluation of Expressions

The compiler evaluates expressions based on:

Order of Evaluation:

Precedence determines which operations are performed first

Associativity determines the order when precedence is equal

Parentheses can override natural precedence.