Program Design Tools & C Programming

The Art of Programming through Algorithms, Flowcharts, and C Fundamentals

A comprehensive overview of program design methodology and the C programming language fundamentals including history, character sets, tokens, data types, variables, constants, and storage classes.

Computer Science Department

Academic Year 2025-2026

Table of Contents

Program Design Tools

C Programming

Learn the fundamentals of programming and C language

Introduction to Program Design Tools

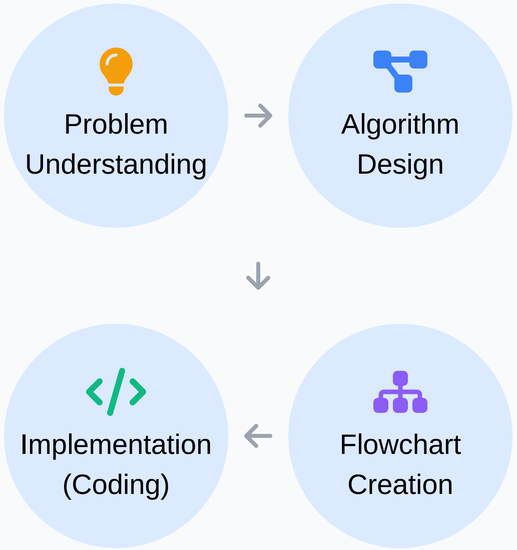
Program design tools help developers plan and visualize software solutions before coding begins. They're essential for:

- ✓ **Problem Analysis:** Breaking down complex problems into manageable steps
- ✓ **Logic Design:** Creating structured logic flow independent of programming language
- ✓ **Communication:** Facilitating clear understanding among team members
- ✓ **Documentation:** Creating reference materials for future maintenance

Main Program Design Tools:

- **Algorithms:** Step-by-step procedures for solving problems
- **Flowcharts:** Visual representations of algorithms using standardized symbols
- **Pseudocode:** Informal high-level descriptions of program logic

The Program Design Process



Problem Understanding → Algorithm Design

↓

Implementation (Coding) ← Flowchart Creation

Proper program design reduces development time and improves code quality

What is an Algorithm?

An algorithm is a finite sequence of well-defined, computer-implementable instructions to solve a specific problem or perform a computation.

- ✓ **Finite:** Algorithm must terminate after a finite number of steps
- ✓ **Definite:** Each step must be precisely defined and unambiguous
- ✓ **Input:** Algorithm takes zero or more inputs
- ✓ **Output:** Algorithm produces at least one output
- ✓ **Independent:** Works regardless of programming language

Common Examples:

- Searching algorithms (Binary search, Linear search)
- Sorting algorithms (Bubble sort, Merge sort, Quick sort)
- Mathematical calculations (Greatest Common Divisor)
- Graph algorithms (Shortest path, Minimum spanning tree)

Factorial Algorithm Example

```
1. Start
2. Input number n
3. Initialize fact = 1
4. For i = 1 to n
    fact = fact × i
5. Output fact
6. End
```

```
Example execution for n = 4:
fact = 1
i = 1: fact = 1 × 1 = 1
i = 2: fact = 1 × 2 = 2
i = 3: fact = 2 × 3 = 6
i = 4: fact = 6 × 4 = 24
Output: 24
```

Flowcharts: Purpose and Use

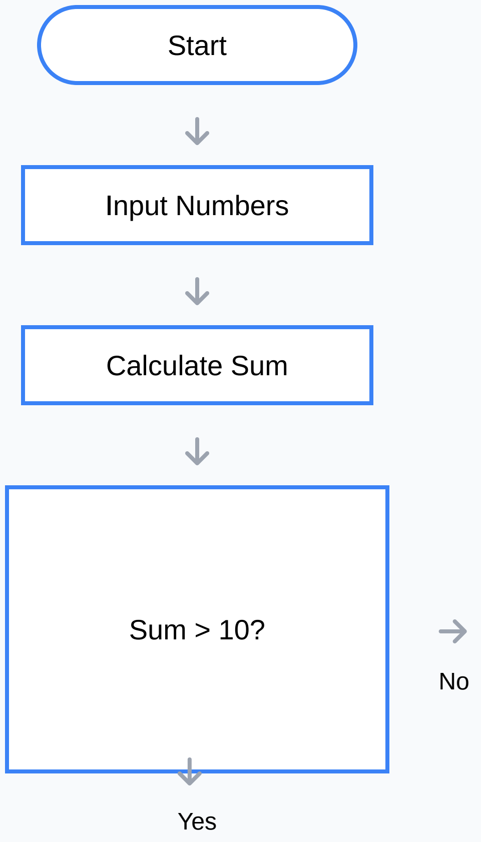
A flowchart is a diagrammatic representation of an algorithm or process using standardized symbols connected by arrows that show the sequence of steps.

- **Visualization:** Translates complex logic into an easy-to-understand visual format
- **Communication:** Provides a universal language for sharing ideas across teams
- **Problem Analysis:** Helps identify logical errors and inefficiencies before coding
- **Documentation:** Serves as valuable program documentation for maintenance and updates

Key Applications in Programming:

- **Algorithm Design:** Planning logical steps before writing code
- **Program Debugging:** Identifying and resolving logical errors
- **Process Optimization:** Improving efficiency of existing processes

Basic Flowchart Structure



"Flowcharts make complex processes simple to understand at a glance"

Key Flowchart Symbols

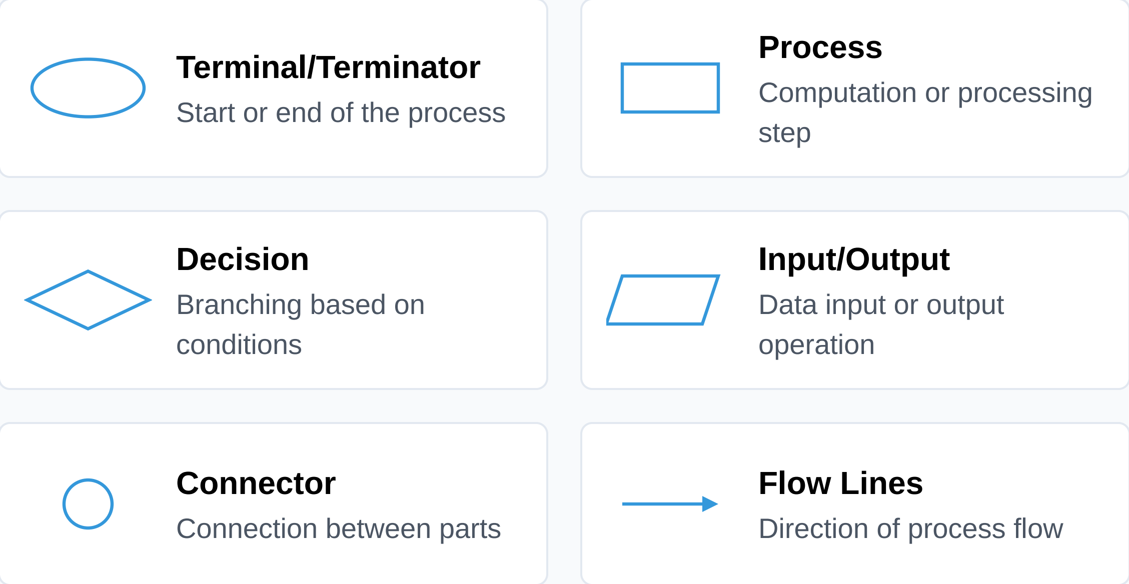
Flowcharts use standardized symbols to represent different steps and actions in a process. Understanding these symbols is essential for reading and creating flowcharts.

- **Universal Communication:** Standard symbols ensure consistent interpretation across different teams and disciplines
- **Clarity of Process:** Each symbol's distinct shape visually indicates its function in the workflow
- **Logical Structure:** Symbols guide the reader through the sequence of operations and decision points

Types of Flowcharts:

- Process Flowcharts (sequence of operations)
- Decision Flowcharts (branching logic)
- Data Flowcharts (data movement through systems)
- System Flowcharts (hardware and software interactions)

Common Flowchart Symbols



Terminal/Terminator
Start or end of the process

Process
Computation or processing step

Decision
Branching based on conditions

Input/Output
Data input or output operation

Connector
Connection between parts

Flow Lines
Direction of process flow

These standard symbols provide a universal language for algorithm representation

Example Program Flowcharts

Flowcharts translate algorithms into visual representations that make logic easier to follow and understand.

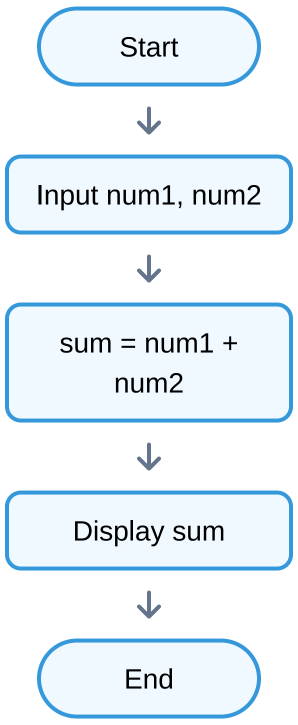
- **Purpose:** These examples demonstrate how common programming problems can be visually represented before coding
- **Clarity:** Flowcharts make logical flow and decision points immediately visible
- **Complexity Management:** Even complex algorithms become easier to understand when visualized

Benefits for Programming:

- Makes logical errors visible before coding begins
- Provides a language-independent blueprint for implementation
- Serves as documentation for the program's logic
- Facilitates communication with non-technical stakeholders

Common Flowchart Examples

1. Adding Two Numbers



Start

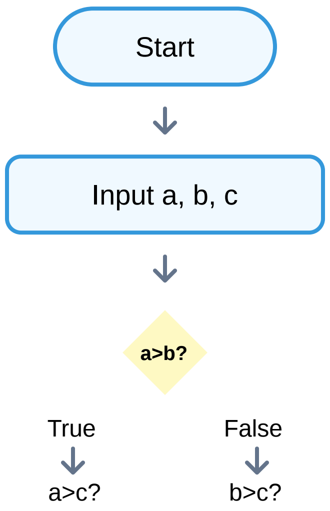
Input num1, num2

sum = num1 + num2

Display sum

End

2. Finding Largest Among Three Numbers



Start

Input a, b, c

a>b?

True

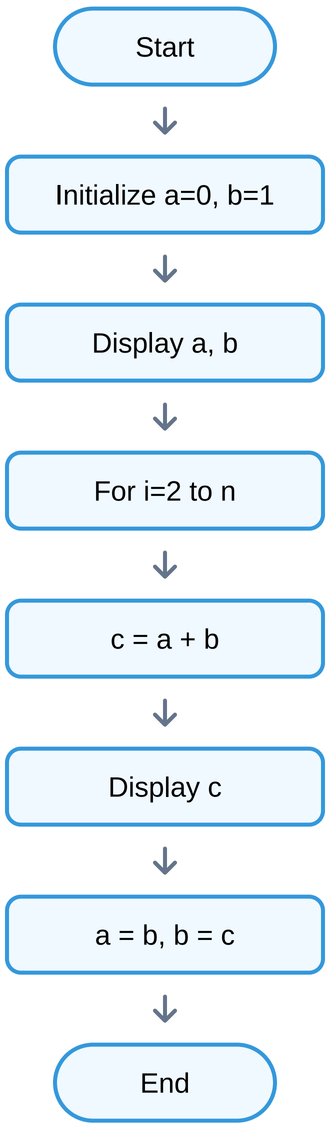
a>c?

False

b>c?

Simplified representation (full flowchart would include additional decision paths and output display)

3. Fibonacci Series Generation



Start

Initialize a=0, b=1

Display a, b

For i=2 to n

c = a + b

Display c

a = b, b = c

End

Advantages & Disadvantages of Flowcharts

Advantages

- **Improved Communication**
  Visually represents logic in a way that's easy for both technical and non-technical stakeholders to understand
- **Effective Analysis & Design**
  Helps identify logical errors, redundancies, and inefficiencies before coding begins
- **Documentation**
  Serves as excellent program documentation for future maintenance and updates
- **Debugging Aid**
  Makes it easier to trace program flow and identify bugs in the logic
- **Language Independence**
  Represents logic without being tied to any specific programming language

Disadvantages

- **Complex for Large Programs**
  Becomes unwieldy and difficult to manage for large, complex programs
- **Time-Consuming to Create**
  Creating and updating detailed flowcharts can be labor-intensive
- **Difficult to Modify**
  Changes in program logic often require complete redrawing of flowcharts
- **Lack of Standardization**
  No precise standard for the level of detail to include in a flowchart
- **Can Become Obsolete**
  May not be updated when code changes, resulting in outdated documentation

History of C Programming

C is one of the most influential programming languages, developed in the early 1970s at Bell Labs by Dennis Ritchie.

- **Origins:** Evolved from the B language (developed by Ken Thompson) and BCPL language
- **Purpose:** Initially developed to implement the Unix operating system
- **Standardization:** First formalized in "The C Programming Language" by Kernighan & Ritchie (K&R C, 1978)
- **Influence:** Has influenced many modern languages including C++, Java, JavaScript, C#, and Python
- **Legacy:** Still widely used for system programming, embedded systems, and applications requiring high performance

C Language Evolution Timeline

C

Dennis Ritchie (1941-2011)

1969-1973:
C development at Bell Labs by Dennis Ritchie

1978:
K&R C - First book on C published ("The C Programming Language")

1989/1990:
ANSI C / C89/C90 - First standardized version

1999:
C99 - Added new features including inline functions and variable-length arrays

2011/2018:
C11 and C17/C18 - Modern standards with multi-threading support

Importance & Features of C

Despite being over 50 years old, C programming language remains fundamentally important in modern computing due to:

- **Hardware Proximity:** Close relationship with hardware architecture, making it ideal for system programming
- **Performance:** Exceptional speed and memory efficiency for resource-intensive applications
- **Portability:** Programs can be compiled on different platforms with minimal changes
- **Foundation for Other Languages:** Influenced popular languages including C++, Java, JavaScript, and Python
- **OS Development:** Core language for developing operating systems (Unix, Linux, Windows kernels)

Key Features of C Language

Procedural Language

Follows top-down approach with functions as basic units of programming

Low-Level Memory Access

Direct manipulation of memory using pointers and addresses

Rich Set of Operators

Comprehensive collection of operators for various operations

Structured Programming

Supports block structures, functions, and structured control statements

Modularity

Programs can be divided into modules for easier maintenance

"C is a language that combines the elements of high-level languages with the functionalism of assembly language."

— Dennis Ritchie, creator of C

Character Set in C

A character set is a collection of valid characters that can be used in a C program. The C language supports the following character groups:

- **Alphabets:** Both uppercase (A-Z) and lowercase (a-z) letters are allowed in variable names, strings, and other identifiers
- **Digits:** Numerical characters (0-9) used for constants, variable names (except as first character), and array indices
- **Special Characters:** Punctuation marks, brackets, operators, and other symbols used for program structure and operations
- **White Space:** Spaces, newlines, tabs, and other invisible characters used to format code for readability

Note:
The ASCII character set is a subset of the C character set, with values from 0 to 127.

C Character Set Categories

Alphabets

A-Z   a-z

Used for identifiers, variable names, function names

Digits

0-9

Used for numeric constants and values

Special Characters

+ - * /   = < > !   ( ) { }   [ ] , ;   # $ & | ^ % ~ _   " ' \ ?

Used for operators, punctuation, and structure

White Space

Space   Tab   Newline   Carriage Return   Form Feed

Used for code formatting and readability

Example Usage in Code

```
int main() {
    // Alphabets & digits in identifiers
    int counter123 = 0;
    // Special chars for operations
    counter123 = (counter123 + 5) * 2;
    return 0;
}
```

Tokens in C

Tokens are the smallest individual units in a C program that are meaningful to the compiler. The C compiler breaks a program into the smallest possible units and proceeds to the various stages of compilation.

- **Definition:** Fundamental building blocks recognized by the compiler
- **Role:** Form meaningful expressions and statements in C programs
- **Syntax Rules:** Each token must follow C language syntax rules

Example C Program with Tokens:

```
int main() {
  int x = 10;
  return 0;
}
```

This simple program contains tokens like: keywords (`int` , `return`), identifiers (`main` , `x`), constants (`10` , `0`), and punctuators (`()` , `{}` , `;`).

Types of Tokens in C

Keywords

Reserved words with predefined meanings

Examples: `int` , `float` , `if` , `else` , `while`

Identifiers

Names given to variables, functions, arrays, etc.

Examples: `main` , `count` , `sum` , `temp`

Constants

Fixed values that cannot be modified

Examples: `10` , `3.14` , `'A'` , `0xFF`

Strings

Sequence of characters enclosed in double quotes

Example: `"Hello World"`

Operators

Symbols that perform operations on operands

Examples: `+` , `-` , `*` , `/` , `==` , `!=`

Punctuators

Special symbols with syntactic meaning

Examples: `{}` , `()` , `;` , `,` , `[]`

Keywords & Identifiers

Keywords

Definition

Reserved words with predefined meanings in C language that cannot be used as identifiers

Examples

`int`, `char`, `float`, `if`, `else`, `while`, `for`, `return`

Characteristics

All keywords must be written in lowercase

ANSI C has 32 keywords, C99 added more

Cannot be redefined in a program

Uses

Define data types (int, char, float)

Control program flow (if, else, switch)

Create loops (for, while, do)

Restrictions

Cannot be used as variable, function, or any identifier names

Fixed set defined by the C standard

Identifiers

Definition

Names given by programmers to variables, functions, arrays, and other user-defined items

Examples

`age`, `studentName`, `calculate_sum`, `_count`, `MAX_VALUE`

Naming Rules

Must start with a letter or underscore

Can contain letters, digits, and underscores

Case sensitive (count ≠ Count)

Best Practices

Use descriptive names that convey purpose

Follow consistent naming conventions

Avoid names that are too similar

Practical Limits

No official limit on length, but most compilers recognize only the first 31 characters

Cannot use C keywords as identifiers

Constants, Variables, and Data Types

C programming uses these fundamental elements for data storage and manipulation:

Variables: Named storage locations that can be modified during program execution

`int age = 25;` // Value can change

Constants: Fixed values that cannot be modified

`const float PI = 3.14159;`

`#define MAX_SIZE 100` // Preprocessor directive

Literals: Fixed values used directly in code

`int x = 10;` // 10 is an integer literal

`char ch = 'A';` // 'A' is a character literal

Key Concepts:

- Variables must be declared before use
- C is a strongly typed language
- Each variable has a specific type that cannot be changed
- Type determines the range of values a variable can hold

Data Types in C

Integer Types

int (4 bytes)
-2,147,483,648 to 2,147,483,647

short int (2 bytes)
-32,768 to 32,767

long int (4-8 bytes)
Platform-dependent range

unsigned int (4 bytes)
0 to 4,294,967,295

Floating-Point Types

float (4 bytes)
~1.2E-38 to 3.4E+38

double (8 bytes)
~1.7E-308 to 1.7E+308

Character Types

char (1 byte)
-128 to 127 or 0 to 255

Other Types

void
Represents absence of type

_Bool (C99)
0 (false) or 1 (true)

ℹ Type sizes may vary between platforms and compilers

Storage Classes in C

Storage classes in C define the scope, lifetime, and visibility of variables. They determine where variables are stored, how long they exist, and which parts of a program can access them.

- **auto:** Default storage class for local variables. Variables are automatically created and destroyed within their scope.
- **register:** Suggests to store variables in CPU registers for faster access. The compiler may ignore this suggestion based on available registers.
- **static:** Preserves variable values between function calls. Static variables are initialized only once and retain their value throughout program execution.
- **extern:** Declares variables that are defined in other files or elsewhere in the program. Used for global variables shared across multiple files.

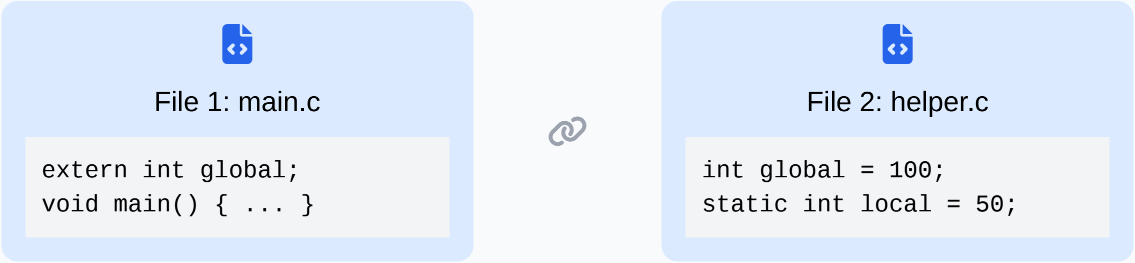
Code Example:

```
void demoFunction() {
    auto int a = 10;        // Local variable
    static int count = 0;   // Static - retains value
    register int fast = 5;  // Register - for faster access

    count++;
    printf("%d %d %d\n", a, count, fast);
}
```

Storage Class Properties

Storage Class	Scope	Lifetime	Default Value	Memory
auto	Local	Function/Block	Garbage	Stack
register	Local	Function/Block	Garbage	Register/Stack
static	Local	Program lifetime	Zero	Data segment
extern	Global	Program lifetime	Zero	Data segment



File 1: main.c

```
extern int global;
void main() { ... }
```

File 2: helper.c

```
int global = 100;
static int local = 50;
```

The **extern** keyword allows variables to be shared across multiple files

Summary & Key Takeaways

Program Design Tools

- **Algorithms**
  Step-by-step procedures for solving problems, independent of programming languages
- **Flowcharts**
  Visual representations of algorithms using standardized symbols
- **Flowchart Symbols**
  Process, decision, I/O, terminal, connector symbols communicate program flow
- **Advantages**
  Improved communication, effective analysis, better documentation, debugging aid
- **Limitations**
  Complex for large programs, time-consuming to create, difficult to modify

C Programming

- **History & Importance**
  Developed by Dennis Ritchie (1970s), foundation for UNIX OS, influenced many modern languages
- **Character Set & Tokens**
  Letters, digits, special symbols form tokens (keywords, identifiers, constants, operators)
- **Variables & Constants**
  Variables store changeable values; constants (const, #define) store fixed values
- **Data Types**
  int, float, char, double each with specific size and range to store different types of data
- **Storage Classes**
  auto, register, static, extern determine variable scope, lifetime, and visibility

Key Insight

Strong foundation in program design tools and C fundamentals is essential for developing efficient, well-structured software solutions