usepackageunderscore

# Structured Programming with C

## Theories & Fundamentals

**Subject Code: DI05016061**

Milav Dabgar

June 18, 2026

# Contents

CHAPTER 1

# Introduction to C Language

## 1.1 Art of Programming with Flowcharts

### 1.1.1 Introduction to the Art of Programming

Programming is inherently a creative process—an art form where logic, mathematics, and structured thinking converge to solve complex problems. Before writing a single line of code in any programming language like C, a programmer must meticulously design a solution strategy. This step is often referred to as algorithm design or logic building.

**Definition**

Algorithm An algorithm is a finite, well-defined sequence of unambiguous instructions intended to solve a specific problem or perform a computation. It is the blueprint of a program, independent of any programming language syntax.

To visualize these algorithms, programmers utilize **flowcharts**. A flowchart is a graphical representation of an algorithm, workflow, or process. It uses standardized symbols connected by arrows to indicate the flow of execution, making complex logical sequences easier to understand, analyze, and communicate.

**Key Concept**

Why Flowcharts Matter Flowcharts bridge the gap between human reasoning and machine execution. They abstract away syntax-specific details, allowing programmers to focus purely on the logic, control flow, and data transformations required to achieve the desired outcome.

### 1.1.2 Basic Notations and Symbols

Flowcharts rely on standard geometric shapes, primarily defined by ANSI (American National Standards Institute) and ISO, to represent different types of actions or steps in a process.

| Symbol Name | Shape | Description / Function |
|---|---|---|
| Terminal (Start/End) | Start/End | Indicates the beginning or end of a program or sub-process. Every flowchart must have a defined start and end. |
| Input / Output | I / O | Represents an input operation (e.g., reading a variable) or an output operation (e.g., displaying a result). |
| Processing | Process | Represents a mathematical computation, variable assignment, or data manipulation step. |
| Decision | Condition | Indicates a branching point based on a logical condition (True/False or Yes/No). |
| Flow Lines | ⟶ | Arrows that connect symbols, indicating the exact sequence and direction of execution flow. |
| Connector | A | Used to connect different parts of a complex flowchart, often jumping across pages or long distances to avoid crossing lines. |

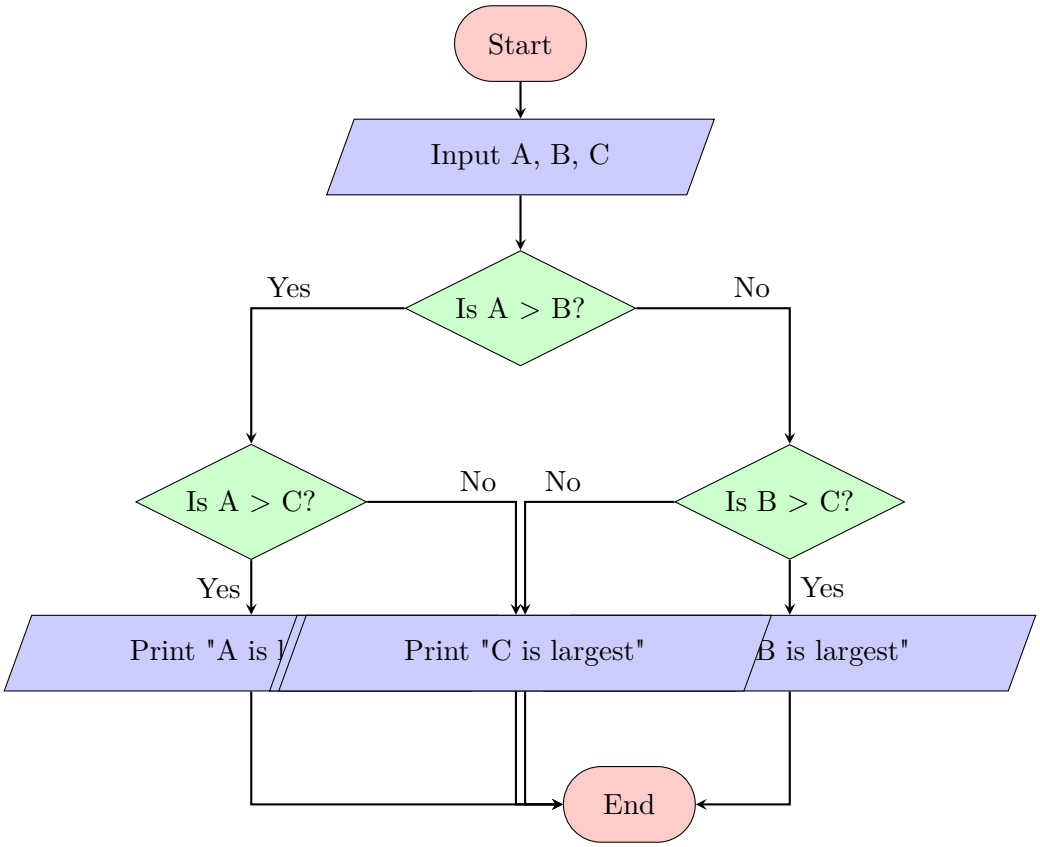
Table 1.1: Standard Flowchart Symbols

### 1.1.3 Flowchart Examples

Let us look at a few examples to understand how logic is visually constructed.

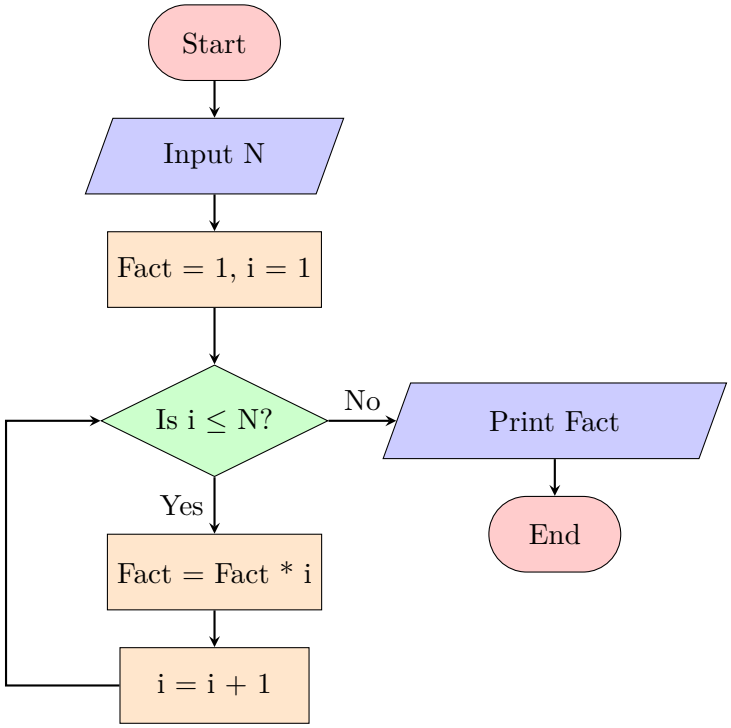
**Example 1: Finding the Largest of Three Numbers**

This flowchart demonstrates basic decision-making (branching) logic.




**Example 2: Calculating Factorial Using a Loop**

This flowchart illustrates iterative (looping) logic.




### 1.1.4 Advantages and Disadvantages of Flowcharts

**Summary Cheatsheet**

Pros & Cons of Flowcharts **Advantages:**

- **Visual Clarity:** Complex logic is easier to grasp graphically than via textual code.
- **Effective Communication:** A universal language bridging technical and non-technical stakeholders.
- **Effective Analysis:** Breaks down a problem into manageable, distinct steps.
- **Debugging Aid:** Helps locate logic errors prior to actual coding.
- **Documentation:** Serves as excellent blueprint documentation for future system maintenance.

**Disadvantages:**

- **Complex for Large Systems:** Can become overwhelmingly intricate, messy, and hard to follow for massive enterprise applications.
- **Time-Consuming:** Drawing shapes and arrows is labor-intensive compared to writing pseudocode.
- **Difficult to Modify:** Adding a single step in the middle of a process often requires redrawing a significant portion of the flowchart.
- **No Standardization for Nuance:** While basic shapes are standard, modeling advanced software design patterns (like Object-Oriented paradigms) using simple flowcharts is inadequate.

## 1.2 C Language Introduction

### 1.2.1 History of C

The C programming language is one of the most historically significant and widely used programming languages in the world. It was developed at **Bell Laboratories** in **1972** by **Dennis Ritchie**.

C was designed primarily as a system programming language to write an operating system. Specifically, it was created to rewrite the **UNIX** operating system, which was originally written in assembly language.

The lineage of C traces back to earlier languages:

- **ALGOL 60** (1960): Provided the basis for structured programming.
- **CPL** (Combined Programming Language, 1963): Brought ALGOL to system programming but was too complex.

- **BCPL** (Basic CPL, 1967): Simplified CPL, developed by Martin Richards.
- **B** (1970): Developed by Ken Thompson, it was a stripped-down version of BCPL used for early UNIX development.
- **C** (1972): Dennis Ritchie added data typing and other features to B, resulting in C.

**Key Concept**

The ANSI C Standard In 1989, the American National Standards Institute (ANSI) established a standard specification for C, known as ANSI C or C89. The International Organization for Standardization (ISO) adopted it in 1990 (C90). Subsequent updates include C99, C11, C18, and C2x, continually adding features while maintaining backward compatibility.

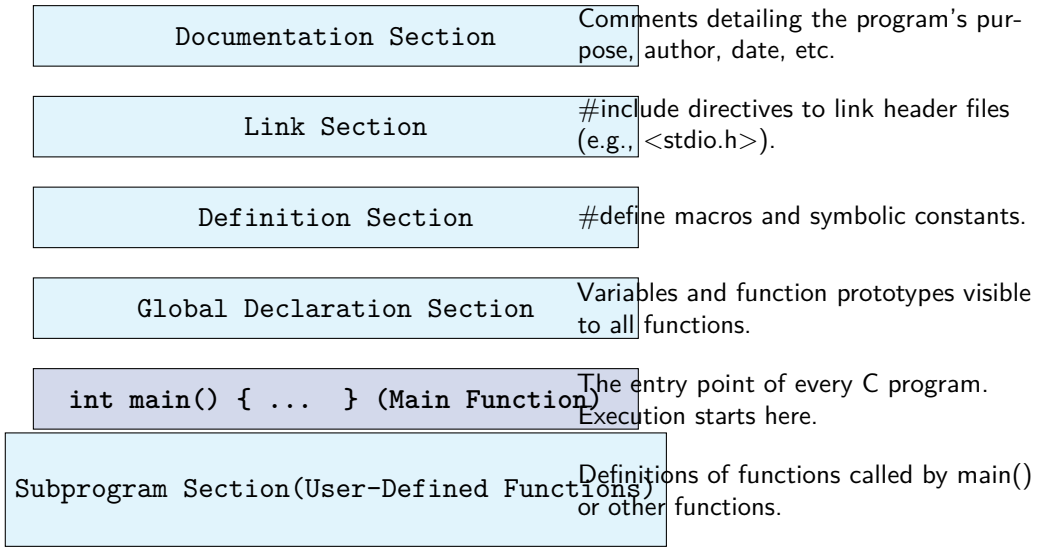
### 1.2.2 Importance of C Language

Despite the proliferation of high-level, object-oriented, and garbage-collected languages (like Java, Python, and C#), C remains indispensable. Its enduring importance is due to several critical factors:

- **System-Level Access:** C allows direct manipulation of hardware, memory addresses, and system resources through pointers. This makes it the language of choice for operating system kernels (Linux, Windows), device drivers, and embedded systems (microcontrollers, IoT devices).
- **High Performance and Efficiency:** C code compiles down directly to highly optimized native machine code. Without the overhead of a virtual machine or a heavy runtime garbage collector, C programs execute with blistering speed and minimal memory footprint.
- **Portability:** A C program written on one machine can often be compiled and run on completely different hardware architecture with little to no modification, provided a C compiler exists for the target architecture.
- **Foundation for Modern Languages:** C is the *lingua franca* of programming. Languages like C++, Java, C#, JavaScript, and Objective-C heavily borrow their syntax and core concepts from C. Mastering C provides a strong conceptual foundation for learning almost any other procedural or object-oriented language.
- **Rich Standard Library:** While the language itself is small, it comes with a robust standard library for input/output, string manipulation, memory allocation, and mathematical computations.

### 1.2.3 Basic Structure of a C Program

A C program is fundamentally a collection of functions and variables. The basic structure dictates how these components are organized within a source file.




**Example**

A Simple C Program Let us examine the structure using the classic "Hello, World!" program.

```
/* Documentation: My First C Program */
```

```c
#include <stdio.h>  // Link Section: Standard I/O library

#define PI 3.14159  // Definition Section: Macro definition

int global_var = 10; // Global Declaration

// Main function: Program entry point
int main() {
    // Local declarations
    int local_var = 5;

    // Executable statements
    printf("Hello, World!\n");
    printf("Global: %d, Local: %d\n", global_var, local_var);

    return 0; // Return statement to OS
}
```

**Important / Warning**

The `main()` Function is Mandatory Every executable C program must contain exactly one function named `main()`. Without it, the linker will throw an error, as it won't know where to begin executing the program.

## 1.3 Process of Compiling and Executing a C Program

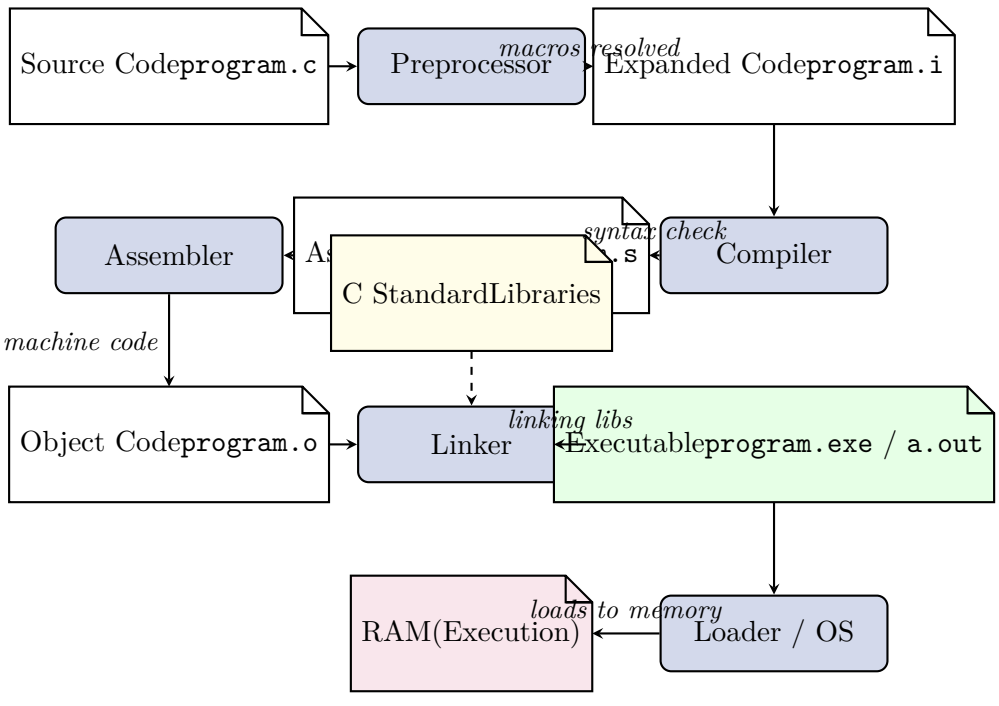
Writing code in a `.c` file is only the first step. The computer's CPU cannot directly understand C code; it only understands raw binary machine code (0s and 1s). The transformation from human-readable source code to an executable binary file involves a multi-stage pipeline known as the **compilation process**.

**Definition**

Compiler A compiler is a specialized software program that translates source code written in a high-level programming language (like C) into low-level machine code or object code that can be executed by the target computer architecture.

### 1.3.1 The Compilation Pipeline

The journey from source code to execution involves four distinct phases: Preprocessing, Compiling, Assembling, and Linking.



**1. Preprocessing**

The preprocessor is the first program invoked. It does not analyze C syntax. Instead, it performs text substitution based on preprocessor directives (commands starting with `#`).

- **File Inclusion:** Replaces `#include <stdio.h>` with the actual contents of the `stdio.h` file.
- **Macro Expansion:** Replaces macros defined by `#define` (e.g., `#define MAX 100`) with their literal values throughout the code.
- **Comment Removal:** Strips all `// line` and `/* block */` comments from the source file, as the compiler doesn't need them.

The output is an expanded *pure C* source file, often carrying a `.i` extension internally.

**2. Compilation**

The compiler takes the expanded `.i` file and translates it into **Assembly language** (`.s` file). During this phase, the compiler performs rigorous checks:

- **Lexical and Syntax Analysis:** Ensures the code follows the grammatical rules of C (e.g., checking for missing semicolons or mismatched braces).
- **Semantic Analysis:** Checks for type correctness (e.g., trying to assign a string to an integer variable).
- **Optimization:** Attempts to optimize the code for speed or size.

If errors are found (syntax errors), the process halts, and the compiler outputs error messages.

**3. Assembly**

The assembler translates the human-readable assembly instructions (`.s` file) into raw binary **Machine Code** (`.o` or `.obj` file). This object file contains actual CPU instructions but is not yet a complete executable. It contains unresolved references to external functions (like `printf`).

**4. Linking**

The object file lacks the actual binary code for standard library functions like `printf()` or `scanf()`. The **Linker** takes the object file (`.o`) and combines it with pre-compiled object files from the C Standard Library.

- It resolves external references, connecting function calls in your code to the actual function definitions in the libraries.
- The final output is a standalone **Executable File** (e.g., `a.out` on Linux/macOS, or `program.exe` on Windows).

**Key Concept**

The Loader Once the executable is generated, running the program invokes the **Loader**, an OS component. The loader reads the executable file from the hard drive, loads it into the computer's Random Access Memory (RAM), initializes registers, and instructs the CPU to begin executing the `main()` function.

## 1.4 Header Files and Library Functions

One of C's design philosophies is to keep the core language small and minimal. Features like input/output, string manipulation, math operations, and memory allocation are not built directly into the C language syntax. Instead, they are provided via the **C Standard Library**.

### 1.4.1 Library Functions

**Definition**

Library Function A library function is a pre-written, pre-compiled, and rigorously tested block of code (function) designed to perform a specific, commonly needed task. These functions are packaged into libraries and made available for programmers to use, promoting code reusability and saving development time.

For example, instead of writing complex low-level code to interface with the operating system and graphics hardware just to display text on the screen, a programmer simply calls the `printf()` library function.

Library functions are generally categorized by their purpose:

- **I/O Functions:** `printf()`, `scanf()`, `getchar()`, `putchar()`
- **Mathematical Functions:** `sin()`, `cos()`, `sqrt()`, `pow()`
- **String Handling:** `strlen()`, `strcpy()`, `strcmp()`, `strcat()`
- **Utility Functions:** `malloc()`, `free()`, `exit()`, `rand()`

### 1.4.2 Header Files

While the actual implementation (the compiled binary code) of library functions resides in library files (like `libc.a` or `libc.so`), the compiler needs to know the *interface* of these functions before it can compile your code. It needs to know the function's name, its return type, and the types of arguments it accepts. This interface definition is called a **function prototype**.

This is where header files come in.

**Definition**

Header File A header file is a text file with a `.h` extension that contains C declarations and macro definitions to be shared between several source files. They act as an index or catalog, providing the compiler with the necessary prototypes for the library functions you intend to use.

To use a library function, you must include its corresponding header file at the top of your program using the `#include` preprocessor directive.

**Example**

Commonly Used Header Files in C

- `<stdio.h>`: Standard Input/Output. Contains prototypes for `printf, scanf, fopen, fclose`, etc.
- `<stdlib.h>`: Standard Library. Contains utility functions like `malloc, free, atoi, rand, exit`.
- `<string.h>`: String manipulation functions like `strlen, strcpy, strcmp`.
- `<math.h>`: Mathematical functions like `sqrt, pow, sin, cos`.
- `<ctype.h>`: Character classification functions like `isalpha, isdigit, toupper, tolower`.
- `<stdbool.h>`: Defines the boolean data type (`bool`, `true`, `false`) introduced in C99.

#### Syntax of `#include`

There are two ways to include a header file, and the subtle difference dictates where the preprocessor searches for the file.

1. **Angle Brackets `< >`:** Used for standard system header files.

   ```
   #include <stdio.h>
   ```

   The preprocessor searches for `stdio.h` in the standard system directories where the compiler is installed (e.g., `/usr/include` on Linux).

2. **Double Quotes `" "`:** Used for user-defined header files.

   ```
   #include "my_math.h"
   ```

   The preprocessor first searches in the current directory containing the source code. If it doesn't find it there, it falls back to searching the standard system directories.

**Important / Warning**

Including Math Library in Linux When compiling a C program that uses `<math.h>` on a Linux system using GCC, simply including the header file is not enough. You must explicitly tell the linker to link the math library (`libm`) using the `-lm` flag:
`gcc program.c -o program -lm`

## 1.5 Control Statements in C

Normally, a C program executes sequentially; instructions are executed one after the other in the order they appear. However, real-world problems require programs to make decisions and repeat actions. **Control statements** alter the sequential flow of execution, enabling complex logic. They are broadly categorized into decision-making (branching) statements and iterative (looping) statements.

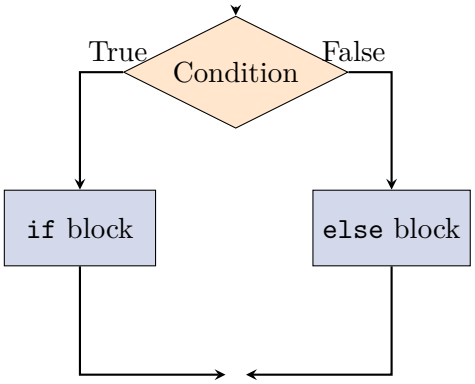
### 1.5.1 Decision Making Statements

Decision-making statements evaluate a boolean condition (True or False) and execute a specific block of code based on the result. In C, any non-zero value is evaluated as True, and zero is evaluated as False.

#### 1. The `if-else` Statement

The `if` statement evaluates a condition. If it is true, the block of code inside the `if` is executed. The optional `else` block is executed if the condition is false.

```c
if (condition) {
    // Executes if condition is True
} else {
    // Executes if condition is False
}
```

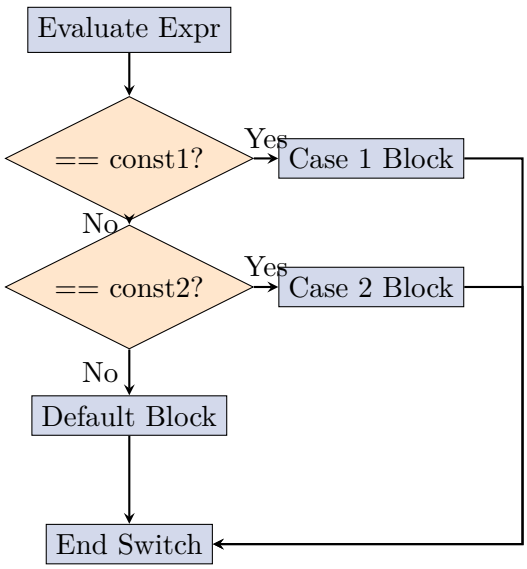



**Nested `if-else` and `else if` Ladder:** You can chain multiple conditions using an `else if` ladder, or nest `if` statements within each other for complex multi-way branching.

#### 2. The `switch` Statement

The `switch` statement is a cleaner alternative to a long `else if` ladder when testing a single variable against multiple constant integer or character values.

```c
switch(expression) {
    case constant1:
        // code
        break;
    case constant2:
        // code
        break;
    default:
        // fallback code
}
```



**Important / Warning**

The Importance of `break` In a `switch` statement, if a matching `case` is found, execution begins there and **falls through** to subsequent cases automatically unless a `break` statement is encountered. The `break` statement forces an immediate exit from the `switch` block.

### 1.5.2 Iterative Statements (Loops)

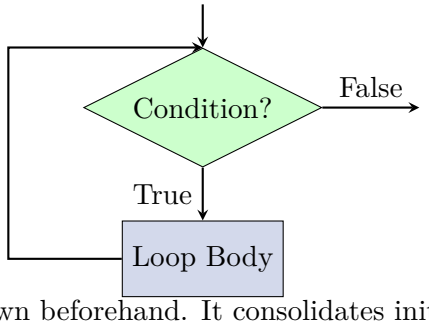
Loops execute a block of code repeatedly as long as a specified condition remains true. Loops are categorized into Entry-Controlled and Exit-Controlled loops.

#### Entry-Controlled Loops

In these loops, the condition is evaluated **before** entering the loop body. If the condition is initially false, the loop body might execute zero times.

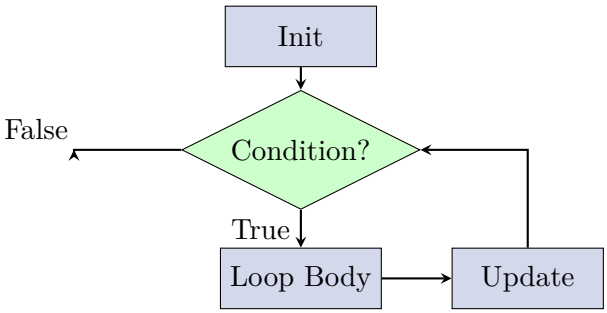
**1. The `while` Loop:** The simplest loop. It repeatedly executes while a condition is true.

```
while (condition) {
    // Loop body
    // update statement
}
```




**2. The `for` Loop:** Best suited when the number of iterations is known beforehand. It consolidates initialization, condition checking, and updating into a single line.

```
for (init; condition; update) {
    // Loop body
}
```

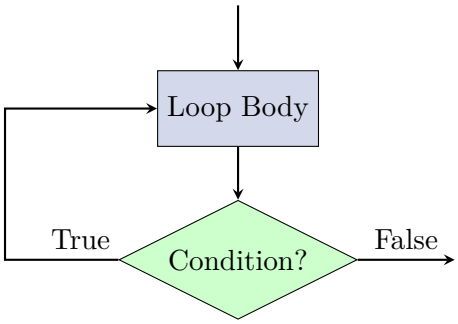



#### Exit-Controlled Loops

In these loops, the loop body is executed first, and **then** the condition is evaluated. This guarantees that the loop body executes at least once, even if the condition is initially false.

**The `do-while` Loop:**

```
do {
    // Loop body
    // update statement
} while (condition);
```




**Summary Cheatsheet**

Loop Selection Guide

- Use **`for`** loop when you know exactly how many times you want to iterate (e.g., iterating over an array).
- Use **`while`** loop when the number of iterations is unknown and depends on a dynamic condition (e.g., reading a file until EOF).

- Use **do-while** loop when you must execute the loop logic at least once before checking the condition (e.g., presenting a menu to a user).

## 1.6 Storage Classes in C

In C, declaring a variable simply using a data type (like `int x;`) only specifies what kind of data the variable can hold. However, every variable possesses another critical attribute defined by its **Storage Class**.

**Definition**

Storage Class A storage class dictates four fundamental properties of a variable:

1. **Storage Location:** Where the variable is stored in the hardware (RAM or CPU Register).
2. **Default Initial Value:** The value it holds if not explicitly initialized (garbage or zero).
3. **Scope:** The region of the program where the variable is visible and accessible.
4. **Lifetime:** The duration for which the variable retains its memory allocation during program execution.

C provides four storage classes: `auto`, `extern`, `static`, and `register`.

### 1.6.1 1. Automatic Storage Class (`auto`)

- **Keyword:** `auto` (rarely explicitly written, as it is the default).
- **Storage:** RAM (specifically, the Stack segment).
- **Default Value:** Garbage value (unpredictable).
- **Scope:** Local to the block `{}` or function in which it is defined.
- **Lifetime:** Exists only while the block/function is executing. Destroyed upon exit.

Every variable declared inside a function without a storage class specifier defaults to `auto`.

```
void myFunction() {
    int x;       // Inherently 'auto int x;'
    auto int y;  // Explicit auto declaration (redundant)
}
```

### 1.6.2 2. External Storage Class (`extern`)

- **Keyword:** `extern` (used to declare, not define).
- **Storage:** RAM (Data segment).
- **Default Value:** Zero (0).
- **Scope:** Global; visible throughout the entire program (even across multiple `.c` files).
- **Lifetime:** Persists for the entire duration of the program.

Global variables are external by default. The `extern` keyword is specifically used to tell the compiler that a variable is defined in *another* file or later in the same file, preventing a "variable not declared" error without allocating new memory.

### 1.6.3 3. Static Storage Class (`static`)

The `static` keyword behaves differently depending on where it is applied (local vs. global scope).

- **Storage:** RAM (Data segment, not the stack).
- **Default Value:** Zero (0).

**Case A: Local Static Variables** When declared inside a function, a static variable retains its value between function calls. It is initialized only once. Its scope remains local to the function, but its lifetime is the entire program duration.

**Example**

Static Variable Example

```c
#include <stdio.h>
void counter() {
    int a = 0;          // Re-initialized to 0 every call
    static int b = 0; // Initialized ONCE. Value persists.
    a++; b++;
    printf("a = %d, b = %d\n", a, b);
}
int main() {
    counter(); // Outputs: a = 1, b = 1
    counter(); // Outputs: a = 1, b = 2
    counter(); // Outputs: a = 1, b = 3
    return 0;
}
```

**Case B: Global Static Variables** When applied to a global variable or function, `static` restricts its scope exclusively to the file in which it is declared. It prevents other files from accessing it via `extern`, effectively providing encapsulation and hiding the variable from the linker.

### 1.6.4 4. Register Storage Class (`register`)

- **Keyword:** `register`.
- **Storage:** CPU Register (if available), otherwise falls back to RAM (like `auto`).
- **Default Value:** Garbage value.
- **Scope:** Local.
- **Lifetime:** Block duration.

Registers are tiny memory locations built directly into the CPU. Accessing them is orders of magnitude faster than accessing RAM. The `register` keyword is a hint to the compiler to store a heavily used variable (like a loop counter) in a CPU register for speed optimization.

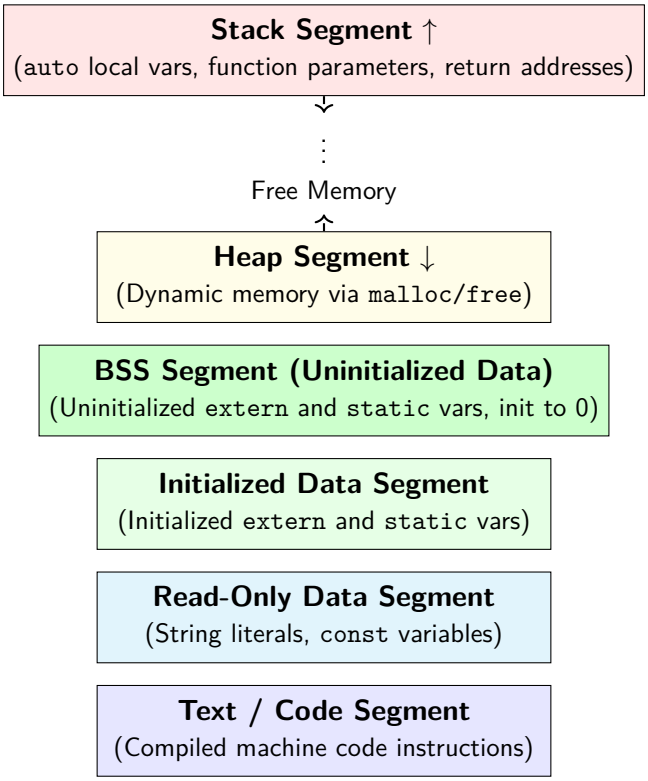
**Important / Warning**

Limitations of `register`

1. CPU registers are scarce. The compiler may ignore your request and treat it as `auto`.
2. You cannot use the address-of operator (`&`) on a register variable, because registers do not have standard memory addresses like RAM.

### 1.6.5 Summary: Memory Map Visualization

Understanding where variables live is crucial. The memory allocated to a running C program is divided into segments.




**Summary Cheatsheet**

Storage Classes Summary

| Storage Class | Storage | Default Init | Scope | Lifetime |
|---|---|---|---|---|
| `auto` | Stack (RAM) | Garbage | Local (Block) | Function execution |
| `extern` | Data Seg (RAM) | Zero | Global | Program duration |
| `static` (local) | Data Seg (RAM) | Zero | Local (Block) | Program duration |
| `register` | CPU Register | Garbage | Local (Block) | Function execution |

## 1.7 Type Qualifiers in C

Type qualifiers are keywords used in variable declarations that alter the default properties of a data type. While a data type (like `int`) specifies the size and representation of a variable, a type qualifier adds semantic constraints regarding how the variable can be accessed or modified by the program and the hardware.

**Definition**

Type Qualifier A type qualifier modifies the behavior of a data type, giving specific instructions to the compiler regarding optimization and memory access semantics for that variable. C supports three primary type qualifiers: `const`, `volatile`, and `restrict`.

### 1.7.1 1. The `const` Qualifier

The `const` (constant) qualifier specifies that the value of a variable **cannot be modified** after it has been initialized. It makes the variable read-only. Any attempt to modify a `const` variable will result in a compile-time error.

```
const int MAX_USERS = 100;
// MAX_USERS = 150; // ERROR: assignment of read-only variable
```

**Key Concept**

Why use `const`?

- **Safety:** Prevents accidental modification of critical values (like Pi or hardware pin definitions).
- **Intent:** Clearly communicates to other developers that a value is meant to be invariant.
- **Optimization:** Allows the compiler to optimize the code better, potentially storing the value in read-only

memory (ROM) on embedded systems.

**Pointers and `const`:** The placement of `const` with pointers creates different constraints:

- `const int *ptr;`: Pointer to a constant integer. You can change where the pointer points, but you cannot change the value it points to via the pointer.
- `int * const ptr;`: Constant pointer to an integer. You cannot change where the pointer points, but you can change the value it points to.

### 1.7.2 2. The `volatile` Qualifier

The `volatile` qualifier tells the compiler that a variable's value **may change at any time, outside the control of the executing program**. Therefore, the compiler must **not** optimize accesses to this variable.

Normally, if a compiler sees a variable being read continuously without any code modifying it, it will optimize the program by reading the variable once from RAM and caching it in a CPU register. However, in system-level programming, a variable might change due to external factors:

- **Hardware Registers:** A memory-mapped hardware status register might change when an external sensor triggers.
- **Interrupt Service Routines (ISRs):** An interrupt handler might modify a global flag that the main loop is checking.
- **Multi-threading:** Another thread in a concurrent program might modify shared data.

**Example**

The need for `volatile` Consider a hardware status flag:

```
int status_flag = 0; // Changed by a hardware interrupt

while (status_flag == 0) {
    // Wait for hardware to set flag to 1
}
```

Without `volatile`, an optimizing compiler might cache `status`$_flaginaregister, noticetheloopdoesn'tchangeit, andturnit$`Declaring it as volatile int status`$_flag = 0; forcesthecompilertofetchthefreshvaluefromRAMon$`every`$iterat$

### 1.7.3 3. The `restrict` Qualifier

Introduced in the C99 standard, `restrict` is a qualifier used exclusively with pointers. It is an optimization hint given to the compiler.

By using `restrict`, the programmer promises the compiler that for the lifetime of the pointer, only that specific pointer (or a value directly derived from it) will be used to access the object it points to. **There will be no "pointer aliasing"** (two different pointers pointing to the same memory block).

```
void update_arrays(int * restrict a, int * restrict b, int * val) {
    *a += *val;
    *b += *val;
}
```

**Important / Warning**

Danger of `restrict` `restrict` is a promise you make to the compiler to help it generate faster code (e.g., by keeping values in registers longer or reordering instructions). The compiler *does not check* if you keep this promise. If you pass overlapping memory arrays to a function with `restrict` pointers, the resulting behavior is completely undefined and will lead to subtle, hard-to-track bugs.

CHAPTER 2

# Arrays and Pointers

## 2.1 Arrays: Foundations of Linear Data Storage

**Definition**

Array An **array** is a fundamental data structure consisting of a collection of elements, typically of the same data type, stored in contiguous memory locations. Elements can be accessed efficiently using an integer index representing their offset from the start of the array.

### 2.1.1 One-Dimensional (1D) Arrays

In C, a one-dimensional array is the simplest form of an array. It represents a linear sequence of elements.

**Key Concept**

Array Declaration and Memory When an array is declared as `type name[size];`, the compiler allocates exactly `sizeof(type) * size` bytes of contiguous memory. The array name acts as a constant pointer to the first element's address.

#### Initialization and Access

Arrays can be initialized at the time of declaration. If the size is omitted during initialization, the compiler automatically infers it based on the number of elements provided.

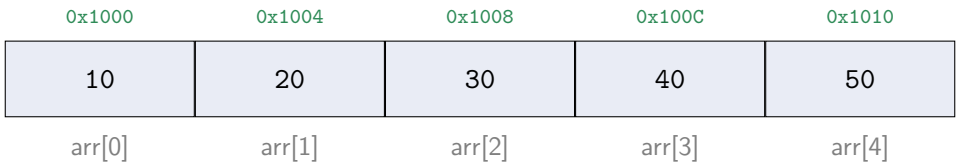
**Example**

1D Array Initialization

```c
#include <stdio.h>

int main() {
    // Explicit size and full initialization
    int scores[5] = {85, 90, 78, 92, 88};

    // Implicit size inference
    int temperatures[] = {32, 34, 31, 35};

    // Partial initialization (remaining elements are zero-initialized)
    int counters[10] = {1, 2, 3};

    printf("Third score: %d\n", scores[2]); // Accessing the 3rd element (0-indexed)
    return 0;
}
```

#### Memory Representation of 1D Arrays

Arrays are stored contiguously in memory. Here is a conceptual visualization of the memory map for an integer array `int arr[5] = {10, 20, 30, 40, 50};`, assuming a 32-bit (4-byte) integer starting at memory address `0x1000`.




**Important / Warning**

Array Bounds Checking C does **not** perform bounds checking on array accesses. Accessing an element out of bounds (e.g., `arr[5]` in an array of size 5) leads to Undefined Behavior (UB), potentially overwriting adjacent memory or causing a segmentation fault.

### 2.1.2 Multi-dimensional Arrays

C supports arrays of arrays, known as multi-dimensional arrays. The most common type is the two-dimensional (2D) array, often used to represent matrices, grids, or tables.

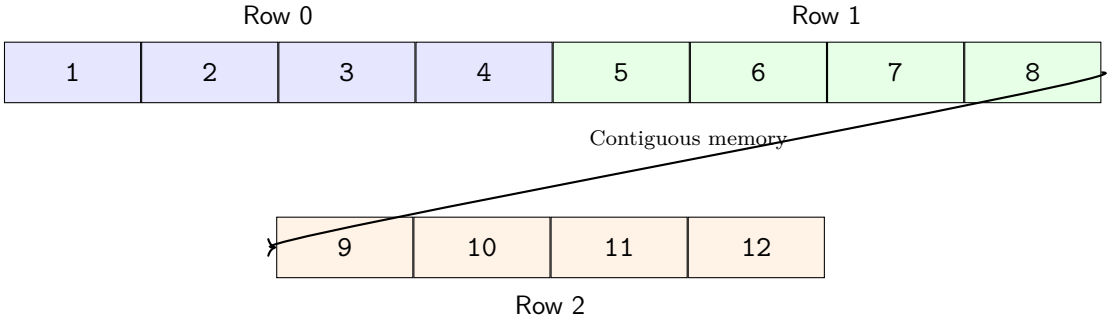
#### 2D Array Memory Layout (Row-Major Order)

Although logically visualized as a grid of rows and columns, memory is purely linear. C stores multi-dimensional arrays in **row-major order**, meaning all elements of the first row are stored contiguously, followed by all elements of the second row, and so forth.

**Example**

Declaring and Processing a 2D Array

```c
#include <stdio.h>

int main() {
    // A 3x4 matrix (3 rows, 4 columns)
    int matrix[3][4] = {
        {1, 2, 3, 4},      // Row 0
        {5, 6, 7, 8},      // Row 1
        {9, 10, 11, 12}    // Row 2
    };

    // Iterating through the 2D array
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%4d ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```



**Summary Cheatsheet**

Array Index Calculation For a 2D array `arr[M][N]`, the memory address of element `arr[i][j]` is calculated as:
**Address** = `Base_Address + (i * N + j) * sizeof(type)`

## 2.2 Variable Length Arrays (VLAs) and Flexible Array Members

Traditional C arrays require their size to be a compile-time constant expression. However, the C99 standard introduced two powerful features to handle dynamic sizing requirements without explicit dynamic memory allocation using `malloc`: Variable Length Arrays and Flexible Array Members.

### 2.2.1 Variable Length Arrays (VLAs)

**Definition**

Variable Length Array (VLA) A Variable Length Array is an array whose size is determined at *runtime* rather than at compile time. Despite the name, once a VLA is created, its size cannot change. The "variable" aspect simply refers to the fact that its size expression is evaluated during execution.

VLAs are typically allocated on the stack. They offer a simpler syntax compared to `malloc` and automatically deallocate their memory when the program execution leaves the block where they are defined.

**Example**

Using Variable Length Arrays

```c
#include <stdio.h>

void process_data(int size) {
    // VLA declaration - 'size' is evaluated at runtime
    int buffer[size];

    // Initialize and print VLA
    for (int i = 0; i < size; i++) {
        buffer[i] = i * 2;
        printf("%d ", buffer[i]);
    }
    printf("\n");
} // 'buffer' is automatically deallocated here

int main() {
    int n;
    printf("Enter array size: ");
    scanf("%d", &n);

    if (n > 0 && n < 1000) { // Always validate bounds!
        process_data(n);
    }
    return 0;
}
```

**Important / Warning**

VLA Stack Overflow Risk Because VLAs are allocated on the thread's stack (which has a limited size, often a few megabytes), creating a very large VLA can cause a stack overflow and crash the program. For large arrays, always prefer heap allocation via `malloc`. Note that VLAs were made optional in the C11 standard.

### 2.2.2 Flexible Array Members

Flexible Array Members (FAMs) solve the problem of structures that need to contain an array of a dynamic size as their final element.

**Key Concept**

Flexible Array Member Syntax A FAM must be the **last** member of a struct. It is declared as an array with an empty dimension `[]`. The struct must have at least one other named member before the FAM.

When you use a FAM, the `sizeof` operator does **not** include the flexible array in the size of the structure. To allocate memory for such a struct, you allocate enough space for the struct's normal members plus the desired size of the flexible array.

**Example**

Implementing Flexible Array Members

```c
#include <stdio.h>
#include <stdlib.h>

// Struct with a Flexible Array Member
struct Packet {
    int length;       // Metadata
    int id;
    char data[];      // Flexible Array Member (must be last)
};

int main() {
    int data_size = 256;

    // Allocate memory for the struct + the flexible array space
    struct Packet *p = malloc(sizeof(struct Packet) + (data_size * sizeof(char)));

    if (p == NULL) {
        perror("Memory allocation failed");
        return 1;
    }

    p->length = data_size;
    p->id = 101;

    // Use the FAM
    for (int i = 0; i < data_size; i++) {
        p->data[i] = 'A' + (i % 26);
    }

    printf("Packet ID: %d, Data[0]: %c\n", p->id, p->data[0]);

    free(p);
    return 0;
}
```

**Summary Cheatsheet**

VLAs vs. Flexible Array Members

- **VLA:** Allocated on the stack. Automatically freed. Cannot be part of a struct. Size defined at declaration.
- **FAM:** Allocated on the heap (via `malloc`). Must be manually freed. Must be the last member of a struct. Size defined at allocation.

## 2.3 Advanced Control Flow

While standard control structures (`if`, `for`, `while`, `switch`) are sufficient for most programming tasks, C provides several advanced control flow mechanisms for specific, niche scenarios, such as jumping out of deeply nested loops or handling errors gracefully.

### 2.3.1 The goto Statement

The goto statement transfers control unconditionally to a labeled statement within the same function.

**Definition**

goto Syntax

```
    goto label;
    // ...
label:
    statement;
```

Although widely discouraged because it can lead to unreadable "spaghetti code," goto is highly effective in C for a specific use case: **centralized error cleanup**. When a function acquires multiple resources (files, memory, locks) and an error occurs midway, goto can jump to a cleanup section, avoiding deeply nested if statements or redundant cleanup code.

**Example**

Using goto for Error Handling

```c
#include <stdio.h>
#include <stdlib.h>

int process_files() {
    FILE *f1 = fopen("file1.txt", "r");
    if (!f1) goto cleanup_none;

    FILE *f2 = fopen("file2.txt", "w");
    if (!f2) goto cleanup_f1;

    void *buffer = malloc(1024);
    if (!buffer) goto cleanup_f2;

    // Perform operations...
    printf("Success!\n");

    free(buffer);
    fclose(f2);
    fclose(f1);
    return 0;

// Cleanup labels in reverse order of allocation
cleanup_f2:
    fclose(f2);
cleanup_f1:
    fclose(f1);
cleanup_none:
    return -1; // Return error code
}
```

### 2.3.2 The NULL Statement

The NULL statement is an expression statement consisting of just a semicolon (;). It does nothing and is used where the language syntax requires a statement, but the program logic does not.

A common use case is in loops where all the work is done within the loop's condition or iteration expressions.

**Example**

The NULL Statement in Action

```c
#include <stdio.h>

int main() {
```

```c
    char str[] = "Hello World";
    char *ptr = str;

    // Find the end of the string.
    // All work is done in the condition and increment.
    // The loop body is a NULL statement.
    while (*ptr++ != '\0')
        ;

    printf("String length: %ld\n", ptr - str - 1);
    return 0;
}
```

### 2.3.3 The Comma Operator

The comma operator (`,`) evaluates its left operand, discards the result, and then evaluates its right operand. The type and value of the entire expression are the type and value of the right operand. It acts as a **sequence point**.

It is predominantly used in `for` loop headers to initialize or increment multiple variables simultaneously.

**Example**

Reversing an Array using the Comma Operator

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = 5;

    // i is initialized to 0, j to n-1
    // both are updated in the iteration step
    for (int i = 0, j = n - 1; i < j; i++, j--) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    return 0;
}
```

### 2.3.4 Non-Local Jumps: `setjmp` and `longjmp`

While `goto` only allows jumping within the same function, C provides a mechanism for **non-local jumps** across function boundaries using `setjmp` and `longjmp`, defined in `<setjmp.h>`. This is C's primitive equivalent of `try/catch` exception handling.

**Key Concept**

How setjmp and longjmp work

- `setjmp(jmp_buf env)`: Saves the current execution context (stack pointer, registers, program counter) into the `env` variable. It returns `0` when called directly.
- `longjmp(jmp_buf env, int val)`: Restores the context saved in `env`. Execution resumes as if `setjmp` just returned, but this time `setjmp` returns the value `val` instead of `0`.

**Example**

Simulating Exceptions with setjmp/longjmp

```c
#include <stdio.h>
#include <setjmp.h>
```

```c
jmp_buf exception_env;

void deep_function(int val) {
    if (val < 0) {
        printf("Error: Negative value detected. Throwing exception!\n");
        longjmp(exception_env, 1); // Jump back to setjmp, returning 1
    }
    printf("Processing value: %d\n", val);
}

int main() {
    int jump_val = setjmp(exception_env);

    if (jump_val == 0) {
        // "try" block
        printf("Calling deep function...\n");
        deep_function(10);
        deep_function(-5); // This will trigger the longjmp
        deep_function(20); // This line is never reached
    } else {
        // "catch" block
        printf("Exception caught! Error code: %d\n", jump_val);
    }

    return 0;
}
```

## 2.4 Pointers: Memory Addressing and Manipulation

Pointers are widely considered the most powerful and challenging feature of the C language. They provide direct access to memory, enabling dynamic memory allocation, efficient array manipulation, and the construction of complex data structures like linked lists and trees.

**Definition**

Pointer A **pointer** is a variable whose value is the memory address of another variable or function. Instead of holding a data value directly, it "points" to the location where the data resides in memory.

### 2.4.1 Core Concepts: Initialization, Address-of, and Dereference

To work with pointers, you must understand two fundamental unary operators:

- The **Address-of operator (&)**: Returns the memory address of its operand.
- The **Dereference operator (*)**: Accesses the value stored at the memory address the pointer holds.

**Example**

Pointer Basics

```c
#include <stdio.h>

int main() {
    int var = 42;           // A standard integer variable
    int *ptr = &var;        // ptr holds the address of var

    printf("Value of var: %d\n", var);
    printf("Address of var: %p\n", (void*)&var);
    printf("Value of ptr (Address): %p\n", (void*)ptr);

    // Dereferencing the pointer to modify the value
    *ptr = 100;
    printf("New value of var: %d\n", var); // Output: 100
```

```
    return 0;
}
```

**Visualizing Pointers in Memory**

Let's visualize the memory state of the variables in the previous example. Assume `var` is located at memory address `0x2000` and `ptr` itself is stored at `0x3000`.


### 2.4.2 Size of Pointers

The size of a pointer variable depends strictly on the architecture of the system (the size of its memory address bus), **not** on the data type it points to.

- On a **32-bit system**, pointers are 4 bytes.
- On a **64-bit system**, pointers are 8 bytes.

An `int *`, a `char *`, and a `double *` will all have the exact same size on a given machine, as they all merely store memory addresses.

### 2.4.3 Special Pointer Types

Understanding different states and types of pointers is crucial for writing robust and bug-free C code.

**Key Concept**

NULL Pointer A pointer that does not point to any valid memory location. It is defined as a macro `NULL` (typically `((void*)0)`). Dereferencing a NULL pointer leads to a program crash (Segmentation Fault).

```
int *ptr = NULL;
if (ptr != NULL) { /* Safe to dereference */ }
```

**Key Concept**

Void Pointer (`void *`) A generic pointer that has no associated data type. It can hold an address of any data type and can be typecast to any type. However, **it cannot be dereferenced** directly, nor can pointer arithmetic be performed on it without casting it to a specific type first.

```
int x = 10;
void *ptr = &x;
int *int_ptr = (int *)ptr; // Cast before use
```

**Important / Warning**

Wild Pointers A wild pointer is a pointer that has been declared but **not initialized**. It points to a random, unpredictable memory location. Dereferencing it is extremely dangerous and causes undefined behavior.

```
int *ptr;      // Wild pointer!
*ptr = 50;     // CRITICAL BUG: Writing to random memory
```

**Important / Warning**

Dangling Pointers A dangling pointer is a pointer that points to a memory location that has already been freed or has gone out of scope.

```
int *ptr = malloc(sizeof(int));
free(ptr);
// ptr is now a dangling pointer!
*ptr = 10; // Undefined Behavior
ptr = NULL; // Best practice: nullify after freeing
```

## 2.5 Advanced Pointers

Building upon the fundamentals of pointers, C allows for complex pointer constructs that provide high-level abstractions, such as modifying pointers within functions and dynamically executing functions.

### 2.5.1 Double Pointers (Pointers to Pointers)

A pointer holds the address of a variable. A **double pointer** (or pointer to a pointer) holds the address of another pointer.

**Definition**

Double Pointer Syntax A double pointer is declared with two asterisks: `type **name;`. To access the final value, you must dereference it twice: `**name`.

**Use Case: Modifying a Pointer inside a Function**

In C, all function arguments are passed by value. If you pass a pointer to a function and modify the pointer itself (change where it points), the change is not reflected in the calling function. To alter the original pointer, you must pass a pointer to the pointer.

**Example**

Allocating Memory via a Function

```
#include <stdio.h>
#include <stdlib.h>

// Pass a double pointer to modify the caller's pointer
void allocate_memory(int **p) {
    // Dereference once to access the original pointer
    *p = (int *)malloc(sizeof(int));
    if (*p != NULL) {
        **p = 100; // Dereference twice to assign value
    }
}

int main() {
    int *my_ptr = NULL;

    // Pass the address of the pointer
    allocate_memory(&my_ptr);

    if (my_ptr != NULL) {
        printf("Value: %d\n", *my_ptr); // Output: 100
        free(my_ptr);
    }
    return 0;
}
```

### Use Case: Dynamic 2D Arrays

Double pointers are the standard mechanism for allocating true dynamic multi-dimensional arrays, where each row can potentially have a different length (an array of arrays).

```
// Allocating a 3x4 dynamic 2D array
int **matrix = malloc(3 * sizeof(int *));
for (int i = 0; i < 3; i++) {
    matrix[i] = malloc(4 * sizeof(int));
}
// matrix[i][j] is now valid
```

## 2.5.2 Function Pointers

Memory doesn't just store data; it also stores executable machine code. A **function pointer** is a pointer that points to the memory address where a function's executable code resides.

**Key Concept**

Function Pointer Syntax Declaration: `return_type (*pointer_name)(parameter_types);`
The parentheses around `*pointer_name` are mandatory. Without them, the compiler parses it as a function returning a pointer.

Function pointers are incredibly powerful for creating callback mechanisms, implementing state machines, and writing generic sorting or searching functions (like `qsort` in the C standard library).

**Example**

Using Function Pointers for Callbacks

```
#include <stdio.h>

// Two standard functions matching the same signature
int add(int a, int b) { return a + b; }
int multiply(int a, int b) { return a * b; }

// A function that accepts a function pointer as an argument
void compute(int x, int y, int (*operation)(int, int)) {
    // Calling the function via the pointer
    int result = operation(x, y);
    printf("Result: %d\n", result);
}

int main() {
    // Passing 'add' function address
    compute(5, 3, add);      // Output: 8

    // Passing 'multiply' function address
    compute(5, 3, multiply); // Output: 15

    return 0;
}
```

**Summary Cheatsheet**

Arrays of Function Pointers You can create an array of function pointers to replace large `switch` statements, often called a **jump table**.

```
int (*math_ops[2])(int, int) = {add, multiply};
int result = math_ops[1](10, 5); // Calls multiply(10, 5)
```

## 2.6 Multilevel Pointers

While single (*) and double (**) pointers are common, C logically permits an arbitrary depth of pointer indirection. A pointer can point to a pointer, which points to another pointer, and so on. These are known as **multilevel pointers**.

### 2.6.1 Understanding Multiple Levels of Indirection

A triple pointer (***) holds the address of a double pointer. A quadruple pointer (****) holds the address of a triple pointer. Each asterisk represents one layer of indirection that must be resolved (dereferenced) to reach the underlying data.

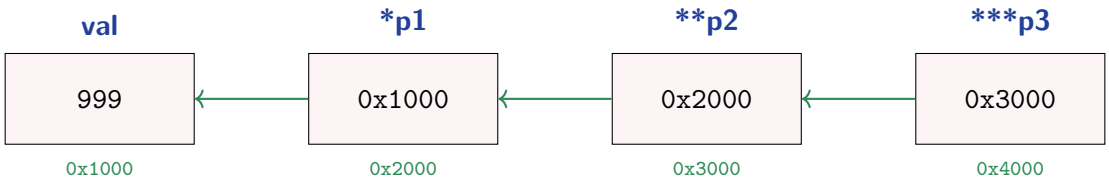
**Example**

Triple Indirection in Code

```c
#include <stdio.h>

int main() {
    int val = 999;

    int *p1 = &val;        // Level 1: Pointer to int
    int **p2 = &p1;        // Level 2: Pointer to pointer to int
    int ***p3 = &p2;       // Level 3: Pointer to double pointer

    // All of these print 999
    printf("val   = %d\n", val);
    printf("*p1   = %d\n", *p1);
    printf("**p2  = %d\n", **p2);
    printf("***p3 = %d\n", ***p3);

    return 0;
}
```

### 2.6.2 Memory Visualization of Multilevel Pointers

To truly grasp multilevel pointers, visualizing the chain of addresses is essential. Let's trace the memory layout of the example above.




### 2.6.3 Practical Limits of Indirection

Although C does not strictly impose a hard limit on pointer depth in the standard (implementations must support at least 12 levels of pointer derivation in C99/C11), practical software engineering restricts indirection depth.

**Important / Warning**

The Complexity Limit Anything beyond a double pointer (**) is generally considered a "code smell" in modern C programming. Triple pointers (***) and beyond drastically reduce code readability and increase the likelihood of segmentation faults due to dereferencing errors.

If you find yourself needing a `***p`, it is almost always better to encapsulate the data in a `struct` and pass a pointer to that struct instead. This replaces abstract memory indirection with meaningful semantic names.

## Summary Cheatsheet

Refactoring Multilevel Pointers **Bad Practice:** `void create_3d_array(int ****matrix, int x, int y, int z);`

**Good Practice:**

```
typedef struct {
    int ***data;
    int x, y, z;
} Tensor3D;

void create_tensor(Tensor3D *tensor, int x, int y, int z);
```

CHAPTER 3

# Functions

## 3.1 Need for Functions, Definition, and Elements

### 3.1.1 Introduction

As programs grow in size and complexity, writing all code inside a single `main()` function becomes increasingly difficult to manage, debug, and maintain. Structured programming addresses this issue through **modularization**—the process of breaking down a large, monolithic program into smaller, manageable, and independent blocks of code called **functions**.

**Definition**

Function A function is a self-contained block of statements that performs a specific, well-defined task. Every C program has at least one function, `main()`, which serves as the entry point of execution.

### 3.1.2 The Need for Functions

The use of functions in C programming is not just a syntactic convenience; it is a fundamental principle of software engineering. The primary reasons for using functions include:

- **Modularity (Divide and Conquer):** Complex problems can be decomposed into smaller, simpler sub-problems. Each sub-problem can be implemented as a separate function, making the overall logic easier to understand.
- **Code Reusability:** Once a function is written and tested, it can be invoked multiple times from different parts of the program, or even included in other programs, eliminating code duplication.
- **Maintainability and Debugging:** If an error occurs, it is easier to locate and fix it within a small function than to search through thousands of lines of monolithic code. Updates or optimizations can be applied to a specific function without affecting the rest of the system.
- **Abstraction:** Functions hide the complex implementation details from the caller. The user of a function only needs to know *what* it does, its inputs, and its outputs, without worrying about *how* it achieves the result.

**Key Concept**

Abstraction Principle Functions enforce abstraction by separating the *interface* (how a function is called) from the *implementation* (how a function works). This separation of concerns is critical for building large-scale software systems.

### 3.1.3 Elements of User-Defined Functions

While C provides numerous built-in functions via the Standard Library (e.g., `printf()`, `scanf()`, `sqrt()`), programmers frequently need to create their own custom functions, known as **User-Defined Functions (UDFs)**.

The creation and utilization of a user-defined function involve three essential elements:

1. **Function Declaration (Prototype)**
2. **Function Call**
3. **Function Definition**

### 1. Function Declaration (Prototype)

The function declaration informs the C compiler about the function's name, return type, and parameters before the function is actually defined or called. This allows the compiler to perform type-checking during compilation.

**Syntax:**

```
return_type function_name(type1 arg1, type2 arg2, ...);
```

### 2. Function Call

A function call transfers the control of execution from the calling function (e.g., `main()`) to the called function. Arguments (actual parameters) are passed to the function, and execution jumps to the function's block.

**Syntax:**

```
function_name(actual_arg1, actual_arg2, ...);
```

### 3. Function Definition

The function definition contains the actual body of the function—the block of C statements that execute when the function is called.

**Syntax:**

```
return_type function_name(type1 param1, type2 param2, ...) {
    // Local variable declarations
    // Executable statements
    // return statement (if return_type is not void)
}
```

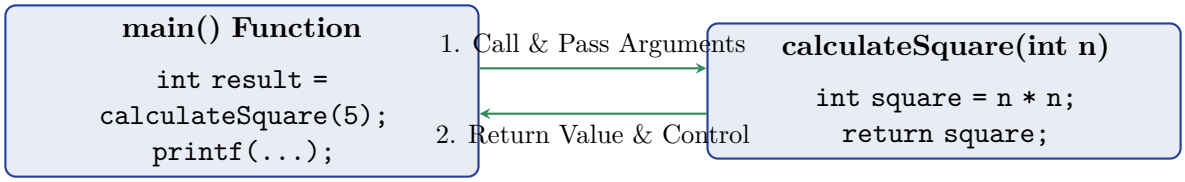
**Example**

Elements of a Function: A Complete Example The following program demonstrates the three elements of a user-defined function that calculates the square of a number.

```c
#include <stdio.h>

// 1. Function Declaration (Prototype)
int calculateSquare(int n);

int main() {
    int num = 5;
    int result;

    // 2. Function Call
    // 'num' is the actual parameter
    result = calculateSquare(num);

    printf("The square of %d is %d\n", num, result);
    return 0;
}

// 3. Function Definition
// 'n' is the formal parameter
int calculateSquare(int n) {
    int square = n * n;
    return square; // Return value to the caller
}
```

## 3.1.4 Visualizing Control Flow

When a function is called, execution pauses in the current function, jumps to the called function, executes its statements, and then returns to the exact point immediately following the call.




**Important / Warning**

Formal vs. Actual Parameters A common source of confusion is the distinction between parameters. **Actual Parameters** are the variables or values passed *during the function call* (e.g., `calculateSquare(num);`). **Formal Parameters** are the variables declared *in the function definition* to receive the values (e.g., `int calculateSquare(int n)`).

## 3.2 Function Declaration, Return Values, and Function Calls

### 3.2.1 Function Declaration (Prototyping) in Depth

A function declaration, commonly referred to as a **function prototype**, provides the compiler with essential metadata about a function before its actual implementation is encountered. It specifies the function's name, the type of value it returns, and the types of arguments it expects.

**Key Concept**

Why is Prototyping Necessary? In older versions of C (like C89), if a function was called before being declared, the compiler implicitly assumed it returned an `int` and accepted any number of arguments. This led to disastrous, hard-to-find bugs if the actual definition differed. Modern C standards (C99 and later) require strict prototyping, allowing the compiler to perform rigorous type-checking.

**Example of valid prototypes:**

```
float calculateArea(float radius); // Parameter name included (good practice)
float calculateArea(float);        // Parameter name omitted (perfectly valid)
void printMenu(void);              // Explicitly states no parameters
```

### 3.2.2 Return Values and their Types

Functions in C can communicate results back to the calling function using the `return` statement. The type of data returned must match the `return_type` specified in the function declaration and definition.

#### The `return` Statement

The `return` statement has two primary roles:

1. It immediately terminates the execution of the function.
2. It transfers the specified value back to the caller.

**Summary Cheatsheet**

Return Type Rules

- A function can only return **one single value** natively.
- The returned value can be of any primary data type (`int`, `float`, `char`), a pointer, or a `struct`.
- Functions cannot return arrays directly (though they can return pointers to arrays).
- If a function does not return any value, its return type must be declared as `void`. In a `void` function, a bare `return;` statement can be used to exit early.

**Example**

Returning Different Types

```c
#include <stdio.h>
#include <stdbool.h>

// Returns a boolean value
bool isEven(int number) {
    return (number % 2 == 0);
}

// Returns a character
char getGrade(float score) {
    if (score >= 90) return 'A';
    if (score >= 80) return 'B';
    return 'C'; // Multiple return statements are allowed
}
```

### 3.2.3 Function Calls and Argument Passing Mechanisms

A function call is an expression that passes control and arguments (if any) to a function. In C, arguments are passed using a strict mechanism known as **Call by Value**.

#### Call by Value

In C, all function arguments are passed by value. This means that a *copy* of the actual argument's value is made and assigned to the formal parameter in the function's local memory scope.

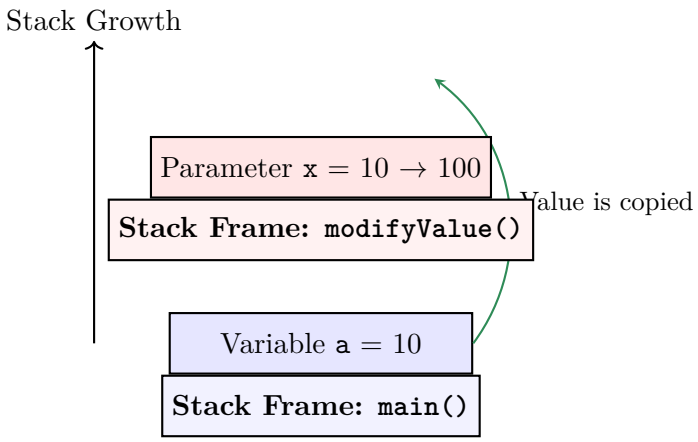
**Important / Warning**

Call by Value Limitation Because the called function operates on a *copy* of the data, any modifications made to the formal parameters inside the function will **not** affect the original actual parameters in the calling function.

```c
#include <stdio.h>

void modifyValue(int x) {
    x = 100; // Modifying the copy
    printf("Inside function: x = %d\n", x);
}

int main() {
    int a = 10;
    modifyValue(a);
    printf("Inside main: a = %d\n", a); // 'a' remains 10
    return 0;
}
```

#### Visualizing Call by Value in Memory

To understand function calls deeply, we must visualize memory. When a function is called, a new **Stack Frame** (or Activation Record) is created on the Call Stack. This frame holds the function's local variables, formal parameters, and the return address.




As shown above, `main()` and `modifyValue()` have distinct stack frames. Modifying `x` in the upper frame has zero impact on `a` in the lower frame. *(Note: To modify the original variable, C uses pointers to simulate Call by Reference, which will be covered in later units).*

## 3.3 Categories of User-Defined Functions

Depending on whether arguments are passed to a function and whether a value is returned from it, user-defined functions in C can be broadly classified into four distinct categories. Understanding these categories is essential for designing appropriate interfaces for your program modules.

| Category | Arguments Passed? | Value Returned? |
|---|---|---|
| **Category 1** | No | No |
| **Category 2** | Yes | No |
| **Category 3** | Yes | Yes |
| **Category 4** | No | Yes |

### 3.3.1 1. Function with No Arguments and No Return Value

These functions do not receive any data from the calling function, nor do they send any data back. They act as independent, isolated blocks of execution. They are typically used for tasks that rely entirely on hardcoded data, global variables, or input/output operations isolated within the function.

- **Declaration:** `void functionName(void);`
- **Use Case:** Printing a welcome banner, clearing the screen, or isolated interactive menus.

```c
void printHeader(void) {
    printf("================================\n");
    printf("    STUDENT DATABASE SYSTEM     \n");
    printf("================================\n");
}
```

### 3.3.2 2. Function with Arguments and No Return Value

These functions receive data (arguments) from the caller but do not return a result. The calling function supplies the necessary inputs, and the called function processes them, usually resulting in a side effect such as printing to the console or modifying a file.

- **Declaration:** `void functionName(type1 arg1, type2 arg2);`
- **Use Case:** Printing formatted output based on inputs, logging errors.

```c
void printSum(int a, int b) {
    int sum = a + b;
    // The result is printed directly, not returned
    printf("The sum of %d and %d is %d\n", a, b, sum);
}
```

### 3.3.3 3. Function with Arguments and Return Value

This is the most flexible, robust, and commonly used category. It closely mirrors mathematical functions: data flows in via arguments, gets processed, and a result flows out via the return value. This category minimizes side effects and dependency on global state, leading to highly modular code.

- **Declaration:** `return_type functionName(type1 arg1, type2 arg2);`
- **Use Case:** Mathematical calculations, data transformations, string manipulations.

```
// Takes two arguments, returns one integer
int calculateMax(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}

// In main: int max_val = calculateMax(10, 20);
```

**Key Concept**

Pure Functions A function that falls into Category 3 and has no side effects (e.g., no printing, no modifying global variables) is often called a **pure function**. Pure functions always produce the same output for the same input, making them incredibly easy to test and debug.

### 3.3.4 4. Function with No Arguments and Return Value

These functions do not require any input from the caller, but they generate and return a value. This often happens when the function acquires data from an external source (like a keyboard, file, or hardware sensor) and passes that data back to the rest of the program.

- **Declaration:** `return_type functionName(void);`
- **Use Case:** Reading a value from the user, getting the current system time, generating a random number.

```
#include <stdio.h>

float getTemperatureReading(void) {
    float temp;
    printf("Enter current temperature reading: ");
    scanf("%f", &temp);
    return temp; // Return the read value to caller
}
```

**Definition**

Summary of Data Flow The categories essentially define the **data flow** between functions:

- **Category 1:** Zero data flow.
- **Category 2:** One-way data flow (Caller → Function).
- **Category 3:** Two-way data flow (Caller → Function → Caller).
- **Category 4:** One-way data flow (Function → Caller).

## 3.4 Passing Arrays, Recursion, and Variable Scope

### 3.4.1 Passing Arrays to Functions

While basic data types in C are passed by value (copying the data), arrays are handled differently to optimize performance and memory usage.

### Array Decay

When an array is passed as an argument to a function, C does **not** copy the entire array. Copying an array with millions of elements would be prohibitively slow. Instead, the array "decays" into a pointer to its first element. The function receives the **base address** of the array.

**Important / Warning**

Passing Size is Mandatory Because the function only receives a pointer to the first element, it loses the information about the array's total size. Therefore, when passing an array to a function, you must almost always pass its length as a separate integer parameter.

```c
#include <stdio.h>

// Formal parameter arr[] implies a pointer to int
void printArray(int arr[], int size) {
    for(int i = 0; i < size; i++) {
        // Modifying arr[i] here WILL modify the original array!
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    // Passing base address 'numbers' and its size
    printArray(numbers, size);
    return 0;
}
```

## 3.4.2 Recursion

**Definition**

Recursion Recursion is a programming technique where a function calls **itself** directly or indirectly to solve a smaller instance of the same problem.

A properly designed recursive function must possess two fundamental components:

1. **Base Case (Terminating Condition):** A simple condition where the function can return a result immediately without calling itself. Without a base case, recursion leads to infinite loops and stack overflow.
2. **Recursive Case:** The part of the function where it calls itself with a modified parameter, incrementally moving closer to the base case.

### Example: Calculating Factorial Recursively

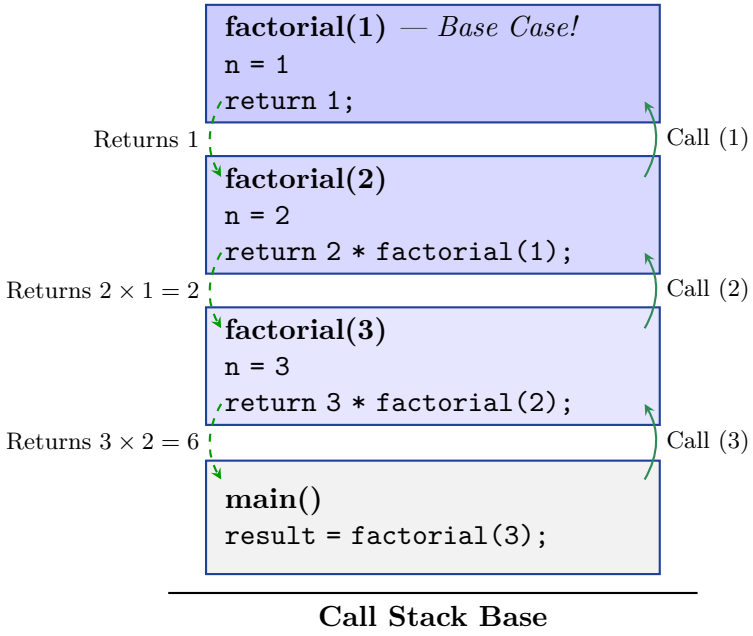
The mathematical definition of a factorial is inherently recursive: $N! = N \times (N-1)!$ with the base case $0! = 1$ and $1! = 1$.

```c
int factorial(int n) {
    if (n == 0 || n == 1) { // Base Case
        return 1;
    } else {                // Recursive Case
        return n * factorial(n - 1);
    }
}
```

### Visualizing the Recursive Call Stack

Understanding recursion requires visualizing the Call Stack. Each time `factorial()` calls itself, execution suspends, and a new stack frame is pushed onto the stack. Only when the base case is reached do the functions begin returning values and popping frames off the stack.

Below is the highly detailed call stack diagram for `factorial(3)`:




As the diagram illustrates, **Winding** occurs as frames pile up waiting for a result. **Unwinding** occurs once the base case is hit, cascading return values back down the stack until `main()` receives the final answer (6).

### 3.4.3 Scope, Visibility, and Lifetime of Variables

Understanding where a variable can be accessed (Scope/Visibility) and how long it remains in memory (Lifetime) is critical in structured C programming.

**Key Concept**

Scope vs. Lifetime

- **Scope (Visibility):** The region of the source code where a variable's name is recognized and can be accessed. (Spatial aspect).
- **Lifetime:** The duration of time during program execution that the variable occupies a valid memory location. (Temporal aspect).

**Storage Classes in C**

C defines four primary storage classes that dictate scope, default initial values, and lifetime:

1. **Automatic (`auto`):** The default for local variables declared inside a block or function.
   - **Scope:** Local (Block scope).
   - **Lifetime:** Exists only while the block/function is executing.
   - **Default Value:** Garbage value.
2. **Static (`static`):**
   - **Scope:** Local to the block in which it is defined.
   - **Lifetime:** Exists for the *entire duration* of the program. It retains its value between function calls.
   - **Default Value:** Zero (0).
3. **External (`extern`):** Used for global variables.
   - **Scope:** Global (File scope, accessible everywhere after declaration).
   - **Lifetime:** Entire duration of the program.
   - **Default Value:** Zero (0).

4. **Register (`register`):** Suggests the compiler store the variable in a CPU register instead of RAM for faster access.
    - **Scope:** Local.
    - **Lifetime:** Same as `auto`.

**Example**

Demonstrating `static` vs `auto`

```
#include <stdio.h>

void counterFunction() {
    int auto_var = 0;         // Re-initialized every call
    static int static_var = 0; // Initialized ONCE

    auto_var++;
    static_var++;

    printf("Auto: %d, Static: %d\n", auto_var, static_var);
}

int main() {
    counterFunction(); // Prints: Auto: 1, Static: 1
    counterFunction(); // Prints: Auto: 1, Static: 2
    counterFunction(); // Prints: Auto: 1, Static: 3
    return 0;
}
```

Notice how `static_var` preserves its state across multiple invocations of the function, living outside the standard stack frame lifecycle.

## 3.5 Advanced Function Concepts

As C evolved through various standards (C99, C11), several advanced features were introduced to give programmers finer control over function execution, performance optimization, and compiler diagnostics.

### 3.5.1 Variadic Functions

Usually, a function in C takes a fixed number of arguments. However, C provides a mechanism to create **variadic functions**—functions that can accept a variable (indefinite) number of arguments. The most famous example of a variadic function is `printf()`.

To create a variadic function, you must include the `<stdarg.h>` header file and use an ellipsis (`...`) as the last parameter in the function declaration.

#### Macros for Variadic Functions

The `<stdarg.h>` library provides specific macros to iterate through the unnamed arguments:

- `va_list`: A type used to declare a variable (often named `args`) that will hold the list of variable arguments.
- `va_start(args, last_fixed_arg)`: Initializes the `va_list` variable. It requires the name of the last known fixed argument.
- `va_arg(args, type)`: Retrieves the next argument in the list of the specified `type`.
- `va_end(args)`: Cleans up the memory assigned to the `va_list`.

**Example**

Example: Variadic Summation This function calculates the sum of an arbitrary number of integers.

```
#include <stdio.h>
#include <stdarg.h>

// First argument specifies how many numbers follow
```

```c
int sum_all(int count, ...) {
    int sum = 0;
    va_list args;

    va_start(args, count); // Initialize argument list

    for (int i = 0; i < count; i++) {
        // Fetch next argument as an integer
        sum += va_arg(args, int);
    }

    va_end(args); // Clean up
    return sum;
}

int main() {
    printf("Sum of 3 numbers: %d\n", sum_all(3, 10, 20, 30));
    printf("Sum of 5 numbers: %d\n", sum_all(5, 1, 2, 3, 4, 5));
    return 0;
}
```

#### The `va_copy` Macro

Introduced in C99, `va_copy(dest, src)` safely copies the state of one `va_list` to another. This is highly useful if you need to iterate over the variable arguments more than once (e.g., first pass to count lengths, second pass to allocate memory and concatenate strings). After using `va_copy`, you must call `va_end()` on the destination list as well.

### 3.5.2 Inline Functions

Function calls introduce overhead: pushing arguments to the stack, jumping to a new memory address, managing stack frames, and jumping back. For very small, frequently executed functions, this overhead can impact performance.

Introduced in C99, the `inline` keyword is a hint to the compiler to perform **inline expansion**. The compiler attempts to replace the function call directly with the function's body code, eliminating the call overhead.

```c
// Requesting the compiler to inline this function
inline int square(int x) {
    return x * x;
}

int main() {
    int result = square(5);
    // The compiler might replace the above line with:
    // int result = 5 * 5;
    return 0;
}
```

**Important / Warning**

Inlining is a Request, Not a Command The `inline` keyword is merely a suggestion to the compiler. If the function is complex, contains loops, or is recursive, the compiler will likely ignore the `inline` request and compile it as a standard function call to prevent massive "code bloat" (where the executable file size becomes unacceptably large).

### 3.5.3 `_Noreturn` Functions

Introduced in the C11 standard, the `_Noreturn` specifier informs the compiler that a function will absolutely never return control back to the caller.

This is not the same as a `void` function. A `void` function finishes its execution and returns control. A `_Noreturn` function physically halts the current thread of execution or terminates the entire program. Examples in the standard library include `exit()` and `abort()`.

**Key Concept**

Why use `_Noreturn`? Using `_Noreturn` serves two purposes: 1. It improves compiler optimization (the compiler doesn't need to generate code to handle a return from this function). 2. It prevents compiler warnings about "missing return statements" or "unreachable code" following the function call.

By including `<stdnoreturn.h>`, you can use the more readable alias `noreturn`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void critical_error(const char* msg) {
    fprintf(stderr, "FATAL ERROR: %s\n", msg);
    exit(EXIT_FAILURE); // Terminates program, never returns
}

int main() {
    int divisor = 0;
    if (divisor == 0) {
        critical_error("Division by zero attempted!");
    }
    // Execution will never reach here
    printf("This will not print.\n");
    return 0;
}
```

CHAPTER 4

# User-defined Datatypes

## 4.1 User-Defined Datatypes: Structures and Unions

**Definition**

User-Defined Datatypes In C, a user-defined datatype enables programmers to group multiple variables (of the same or different types) into a single composite entity. The most common user-defined types are **Structures** (`struct`) and **Unions** (`union`).

### 4.1.1 C Structures

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

**Key Concept**

Structure Declaration and Definition To define a structure, the `struct` keyword is used, followed by a tag name and a block of member declarations.

```
struct Student {
    int roll_no;
    char name[50];
    float marks;
}; // Semicolon is mandatory
```

### 4.1.2 Accessing and Initializing Structures

Structure members are accessed using the dot operator (`.`). For structure pointers, the arrow operator (`->`) is used.

**Example**

Initializing and Accessing

```
#include <stdio.h>
#include <string.h>

struct Point {
    int x;
    int y;
};

int main() {
    // Initialization at declaration
    struct Point p1 = {10, 20};

    // Member-wise initialization/assignment
    struct Point p2;
    p2.x = 30;
    p2.y = 40;
```

```c
    printf("p1: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

### 4.1.3 Copying Structures

C allows direct assignment of one structure variable to another of the same type using the assignment operator (=). This performs a shallow copy of all members.

```c
struct Point p3 = p1; // p3.x becomes 10, p3.y becomes 20
```

### 4.1.4 Unions

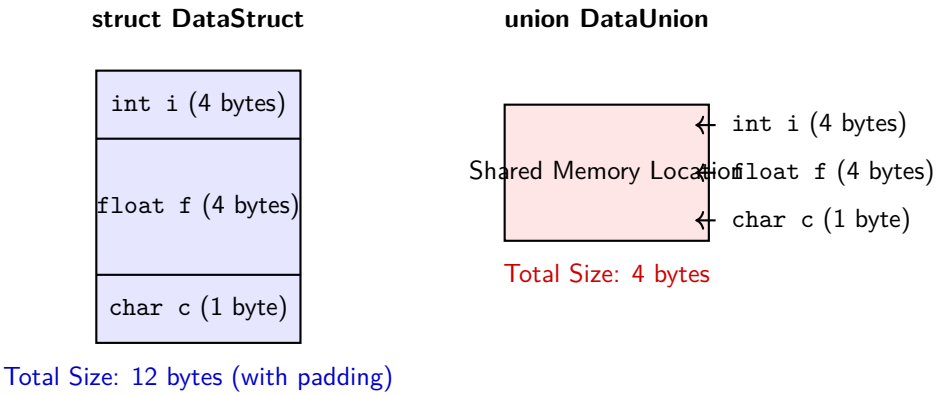
A union is similar to a structure, but all its members share the *same memory location*. Therefore, a union can store only one of its non-overlapping members at any given time.

**Definition**

Union A union allocates enough memory to hold its largest member. Modifying one member overwrites the value of the others.

```c
union Data {
    int i;
    float f;
    char str[20];
};
```

### 4.1.5 Structures vs Unions: Memory Allocation




**Summary Cheatsheet**

Comparison: Struct vs Union

| Feature | Structure | Union |
|---|---|---|
| **Keyword** | `struct` | `union` |
| **Memory** | Allocated for each member separately. | Shared memory for all mem |
| **Size** | $\geq$ Sum of the sizes of all members. | Equal to the size of the larg |
| **Access** | All members can be accessed simultaneously. | Only one member can be sa |

### 4.1.6 Passing Structures to Functions

Structures can be passed to functions in two ways:

1. **By Value:** A copy of the entire structure is created. Changes made inside the function do not affect the original structure. This can be inefficient for large structures.
2. **By Reference (Pointers):** The address of the structure is passed. It is memory efficient, and changes affect the original structure.

**Example**

Passing by Reference

```c
#include <stdio.h>

struct Rectangle {
    int length;
    int width;
};

// Passing by reference
void scaleRectangle(struct Rectangle *r, int factor) {
    r->length *= factor;  // using arrow operator
    r->width *= factor;
}

int main() {
    struct Rectangle rect = {10, 5};
    scaleRectangle(&rect, 2);
    printf("Scaled: %d x %d\n", rect.length, rect.width);
    return 0;
}
```

### 4.1.7 The typedef Keyword

`typedef` is used to create an alias for an existing datatype. It is heavily used with structures to avoid typing `struct` repeatedly.

```c
typedef struct {
    int id;
    float salary;
} Employee;

Employee emp1 = {101, 55000.0}; // No need for 'struct' keyword
```

### 4.1.8 Structure Pointers

A structure pointer holds the memory address of a structure variable. The `->` (arrow) operator is used to access structure members via a pointer, which is shorthand for `(*ptr).member`.

**Important / Warning**

Uninitialized Pointers Always ensure a structure pointer points to a valid memory location (either a declared variable or dynamically allocated memory using `malloc`) before accessing its members. Dereferencing an uninitialized pointer leads to segmentation faults.

## 4.2 The Preprocessor

**Definition**

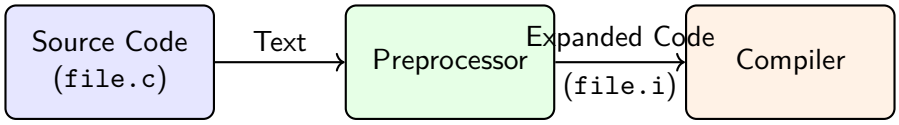
The C Preprocessor The C Preprocessor is a macro processor that transforms the C source code automatically before it is passed to the compiler. It is not part of the compiler itself but acts as a preliminary text-substitution tool.

### 4.2.1 Overview of the Preprocessor

All preprocessor directives begin with a hash symbol (`#`). They must be the first non-whitespace character on a line, and they do not end with a semicolon. The preprocessor performs several crucial tasks:

- **Macro Substitution:** Replacing identifiers with their defined values or code blocks.
- **File Inclusion:** Inserting the contents of other files (usually header files) into the source code.

- **Conditional Compilation:** Compiling specific parts of the code based on certain conditions.




### 4.2.2 Conditional Compilation

Conditional compilation allows you to control which parts of the source code are compiled and which are ignored by the compiler. This is particularly useful for debugging, cross-platform compatibility, and feature toggling.

**Key Concept**

Conditional Directives

- `#if`, `#elif`, `#else`, `#endif`: Used for evaluating constant integer expressions.
- `#ifdef`: True if a macro is defined.
- `#ifndef`: True if a macro is *not* defined.

**Example**

Platform-Specific Compilation

```c
#include <stdio.h>

#define WINDOWS 1
#define LINUX 2
#define OS WINDOWS

int main() {
#if OS == WINDOWS
    printf("Compiling for Windows...\n");
#elif OS == LINUX
    printf("Compiling for Linux...\n");
#else
    printf("Unknown OS...\n");
#endif
    return 0;
}
```

**Important / Warning**

Expression Evaluation The `#if` directive can only evaluate constant integer expressions. You cannot use variables, function calls, or `sizeof` operators in preprocessor conditions, as these are evaluated during compilation or runtime, not preprocessing.

### 4.2.3 The `#ifdef` and `#ifndef` Directives

These are specialized shortcuts for checking macro definitions.

- `#ifdef MACRO`: Equivalent to `#if defined(MACRO)`
- `#ifndef MACRO`: Equivalent to `#if !defined(MACRO)`

```c
#define DEBUG_MODE

int main() {
#ifdef DEBUG_MODE
    printf("Debug mode is active. Tracing enabled.\n");
#endif
    return 0;
}
```

## 4.3 Include Guards, #undef, #pragma, and #error

### 4.3.1 Include Guards

When writing larger C programs, header files (`.h`) are frequently included in multiple source files or even within other header files. This can lead to the **multiple inclusion problem**, where the compiler attempts to redefine structures, macros, or functions, resulting in compilation errors.

**Definition**

Include Guard An include guard (or macro guard) is a conditional compilation construct used to prevent a header file from being included multiple times in a single compilation unit.

**Example**

The `#ifndef` Pattern The standard way to implement an include guard uses `#ifndef`, `#define`, and `#endif`.

```
/* math_utils.h */
#ifndef MATH_UTILS_H   // Check if macro is NOT defined
#define MATH_UTILS_H   // Define the macro immediately

// Declarations go here
int add(int a, int b);
struct Vector2D { float x; float y; };

#endif /* MATH_UTILS_H */
```

### 4.3.2 The #undef Directive

The `#undef` directive removes a macro definition. Once a macro is undefined, it is as if it was never defined, and subsequent `#ifdef` checks will fail.

```
#define BUFFER_SIZE 1024
/* Use BUFFER_SIZE */

#undef BUFFER_SIZE
#define BUFFER_SIZE 2048 // Redefine without warning
```

### 4.3.3 The #pragma Directive

The `#pragma` directive provides a way to issue compiler-specific commands. Because it is compiler-specific, a `#pragma` that works in GCC might be ignored by MSVC.

**Key Concept**

Common Pragmas

- `#pragma once`: A modern, non-standard but widely supported alternative to include guards. It tells the compiler to include the header file only once. It is less error-prone and slightly faster than traditional `#ifndef` guards.
- `#pragma pack(n)`: Instructs the compiler to align structure members on an `n`-byte boundary, overriding the default padding. This is useful for exact memory mapping (e.g., network packets).

```
#pragma pack(1) // Force 1-byte alignment (no padding)
struct NetworkHeader {
    char flag;
    int id;
}; // Size will be 5 bytes instead of 8 bytes
#pragma pack()  // Reset to default alignment
```

### 4.3.4 The `#error` Directive

The `#error` directive forces the compiler to halt compilation and output a specific error message. It is heavily used in conditional compilation to ensure that necessary configurations or macros are defined before proceeding.

**Example**

Using `#error` for configuration checking

```
#ifndef API_VERSION
#error "API_VERSION must be defined before compiling!"
#endif

#if API_VERSION < 3
#error "This code requires API_VERSION 3 or higher."
#endif
```

## 4.4 Macros

**Definition**

Macros A macro is a fragment of code which has been given a name. Whenever the name is used in the code, it is replaced by the contents of the macro by the preprocessor before the actual compilation begins.

### 4.4.1 Overview of Macros

Macros are created using the `#define` directive. There are two primary types of macros:

1. **Object-like Macros:** These resemble data objects or constants.
2. **Function-like Macros:** These take arguments and resemble function calls.

**Example**

Object-like vs Function-like Macros

```
#include <stdio.h>

// Object-like Macro
#define PI 3.14159
#define MAX_USERS 100

// Function-like Macro
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    float area = PI * SQUARE(5);
    int highest = MAX(10, 20);
    printf("Area: %f, Highest: %d\n", area, highest);
    return 0;
}
```

### 4.4.2 Macro Pitfalls and Parentheses

When defining function-like macros, you *must* generously wrap arguments and the entire expression in parentheses. Because macros perform simple text substitution, operator precedence can cause severe logic errors.

**Important / Warning**

The Parenthesis Trap

```
// BAD MACRO
#define DOUBLE(x) x * 2
```

```c
int result = DOUBLE(3 + 4);
// Expands to: 3 + 4 * 2 = 11 (Expected 14!)

// GOOD MACRO
#define DOUBLE_SAFE(x) ((x) * 2)

int safe_result = DOUBLE_SAFE(3 + 4);
// Expands to: ((3 + 4) * 2) = 14
```

### 4.4.3 Macros vs. Functions

While macros can mimic functions, their underlying mechanism (text substitution vs. branching and stack frames) creates significant differences in performance, safety, and behavior.

**Summary Cheatsheet**

Comparison Table: Macros vs. Functions

| Feature | Macros | Functions |
|---|---|---|
| **Execution** | Preprocessed before compilation (Text substitution). | Executed during r branching). |
| **Speed** | Faster (no function call overhead). | Slower (overhead ping from stack). |
| **Code Size** | Increases code size (inline expansion at every usage). | Smaller code size called many times |
| **Type Checking** | No type checking (can lead to hidden bugs). | Strict type checkin |
| **Debugging** | Hard to debug (source code doesn't match compiled code). | Easy to debug (c tions). |
| **Side Effects** | Arguments evaluated multiple times (e.g., `MAX(x++, y)` causes double increment). | Arguments evalua before passing. |

## 4.5 Preprocessor Operators and Predefined Macros

### 4.5.1 Preprocessor Operators

C provides two special operators specifically for use within macros: the Stringizing operator (`#`) and the Token Pasting operator (`##`).

**Key Concept**

Stringizing Operator (`#`) The `#` operator converts a macro parameter into a string literal. It is highly useful for debugging and logging macros.

```c
#define PRINT_VAR_NAME_AND_VALUE(var) printf(#var " = %d\n", var)

int main() {
    int counter = 42;
    // Expands to: printf("counter" " = %d\n", counter)
    PRINT_VAR_NAME_AND_VALUE(counter); // Output: counter = 42
    return 0;
}
```

**Key Concept**

Token Pasting Operator (`##`) The `##` operator concatenates (pastes) two tokens together to form a single token. This is used to dynamically generate variable names or function names.

```c
#define DECLARE_STRUCT(name) struct name##_Data { int id; float val; }

// Expands to: struct Sensor_Data { int id; float val; }
DECLARE_STRUCT(Sensor);
```

### 4.5.2 Predefined Macros

The C standard defines several macros that are always available. They provide useful information about the compilation environment and source file.

**Summary Cheatsheet**

Standard Predefined Macros

- `__FILE__`: A string literal representing the current filename.
- `__LINE__`: An integer representing the current line number.
- `__DATE__`: A string literal of the compilation date ("Mmm dd yyyy").
- `__TIME__`: A string literal of the compilation time ("hh:mm:ss").
- `__STDC__`: Defined as 1 if the compiler complies with the ANSI C standard.

**Example**

Using Predefined Macros for Error Logging

```c
#include <stdio.h>
#include <stdlib.h>

#define ASSERT(condition) \
    if (!(condition)) { \
        fprintf(stderr, "Assertion failed in %s at line %d\n", __FILE__, __LINE__); \
        exit(EXIT_FAILURE); \
    }

int main() {
    int *ptr = NULL;
    ASSERT(ptr != NULL); // Will terminate program and log exact location
    return 0;
}
```

### 4.5.3 Creating Your Own Macros: Variadic Macros

C99 introduced variadic macros, which can accept a variable number of arguments, similar to `printf`. The arguments are represented by an ellipsis (`...`) and accessed using the `__VA_ARGS__` identifier.

```c
#define LOG_ERROR(format, ...) fprintf(stderr, "[ERROR] " format "\n", __VA_ARGS__)

int main() {
    int code = 404;
    LOG_ERROR("Failed to load file. Code: %d", code);
    return 0;
}
```

## 4.6 Advanced Data Types

### 4.6.1 `#define` Constants vs `const` Variables

There are two ways to define constants in C: using the preprocessor (`#define`) and using the `const` keyword.

**Summary Cheatsheet**

`#define` vs `const`

- **`#define MAX 100`:** Handled by the preprocessor. Does not occupy memory (it's simply substituted). Has no scope (global from the point of definition) and lacks type checking.
- **`const int MAX = 100;`:** Handled by the compiler. It is an actual variable stored in memory (often read-only memory). It respects block scope and provides strict type checking.

Generally, modern C programming prefers `const` over `#define` for data constants due to type safety and debugging visibility.

### 4.6.2 Advanced Uses of `typedef`

While `typedef` is commonly used for structures, its true power shines when simplifying complex types, such as multi-dimensional arrays and function pointers.

**Example**

Simplifying Function Pointers with typedef Without `typedef`, declaring a function pointer is notoriously difficult to read.

```
// Standard function pointer declaration
void (*func_ptr)(int, float);

// Using typedef
typedef void (*CallbackFunc)(int, float);

CallbackFunc my_callback; // Much cleaner!
```

### 4.6.3 Complex Number Types (`<complex.h>`)

Starting with C99, C supports complex number math natively. The header `<complex.h>` defines macros and functions for complex arithmetic.

```
#include <stdio.h>
#include <complex.h>

int main() {
    // Define a complex number: 3.0 + 4.0i
    double complex z1 = 3.0 + 4.0 * I;
    double complex z2 = 1.0 - 2.0 * I;

    double complex sum = z1 + z2;

    // creal() extracts real part, cimag() extracts imaginary part
    printf("Sum: %.1f + %.1fi\n", creal(sum), cimag(sum));
    return 0;
}
```

### 4.6.4 Designated Initializers (C99)

Designated initializers, introduced in C99, allow you to initialize specific array elements or structure members by name or index. This greatly improves readability and avoids errors when structure definitions change.

**Example**

Designated Initializers in Arrays and Structures

```
struct Configuration {
    int baud_rate;
    int data_bits;
    int parity;
```

```
};

int main() {
    // Structure designated initialization
    // Order doesn't matter, uninitialized members are zeroed
    struct Configuration cfg = {
        .data_bits = 8,
        .baud_rate = 9600
    };

    // Array designated initialization
    // Indices 0, 1, 2 are zeroed
    int numbers[10] = { [3] = 40, [8] = 90 };

    return 0;
}
```

CHAPTER 5

# File Handling in C

## 5.1 File Handling Basics

**Definition**

File and File Handling A **file** is a contiguous block of related data stored on a non-volatile secondary storage device (such as a hard drive, SSD, or flash drive) that retains data even when the computer is powered off. **File handling** in C refers to the process of creating, reading, writing, and managing these files through a C program using standard library functions.

### 5.1.1 The Necessity of File Handling

In basic C programs, input is typically provided via the keyboard (standard input) and output is displayed on the console (standard output). However, this approach has significant limitations:

- **Volatility:** Data stored in variables, arrays, or dynamic memory during execution is lost entirely once the program terminates.
- **Data Volume:** Manually entering large datasets (e.g., thousands of student records) every time the program runs is impractical and error-prone.
- **Portability and Sharing:** Console output cannot be easily moved to another system or processed by a different application.

**Key Concept**

The Concept of Streams In C, all input and output operations are performed with **streams**. A stream is a sequence of bytes flowing into or out of the program. When you open a file, C associates a stream with that file. The `FILE` structure (defined in `<stdio.h>`) holds all the information the system needs to manage the stream, including the file's location, current position, buffer information, and error status.

### 5.1.2 Naming a File

Before operating on a file, you must identify it by its name. A file name generally consists of two parts:

1. **Primary Name:** The main identifier for the file (e.g., `student_records`).
2. **Extension:** An optional suffix indicating the file type (e.g., `.txt`, `.csv`, `.dat`).

**Important / Warning**

File Path Considerations When opening a file, you can specify just the file name (e.g., `"data.txt"`), which assumes the file is in the current working directory of the program. Alternatively, you can provide an **absolute path** (e.g., `"C:\\data\\records.txt"` on Windows, or `"/home/user/records.txt"` on Linux). Note the use of double backslashes (\\) in Windows paths within C strings to escape the backslash character.

### 5.1.3 Creating and Opening a File

To perform any operation on a file, it must first be opened using the `fopen()` function. This function serves a dual purpose: if the file does not exist (and the mode permits), it **creates** the file; otherwise, it **opens** the existing file.

```
FILE *fopen(const char *filename, const char *mode);
```

The function returns a pointer to a `FILE` structure. If the file cannot be opened (e.g., due to permission issues or non-existent path), it returns `NULL`.

**Example**

Opening a File

```c
#include <stdio.h>

int main() {
    FILE *filePtr;

    // Open (or create) a file for writing
    filePtr = fopen("example.txt", "w");

    if (filePtr == NULL) {
        printf("Error: Could not open the file.\n");
        return 1; // Exit with error code
    }

    printf("File opened successfully!\n");
    // ... operations on the file ...

    fclose(filePtr);
    return 0;
}
```

### 5.1.4 Closing a File

Once all operations on a file are complete, you must explicitly close the file using the `fclose()` function.

```
int fclose(FILE *stream);
```

**Important / Warning**

Always Close Your Files! Closing a file is critical because:

- It flushes any unwritten data remaining in the memory buffers to the physical disk.
- It releases system resources (like file descriptors) locked by the operating system.
- Failing to close a file can lead to data loss or file corruption.

### 5.1.5 Writing to a File

C provides several functions for writing text to a file:

- `fputc(char c, FILE *fp)`: Writes a single character to the file.
- `fputs(const char *str, FILE *fp)`: Writes a string (without the null-terminator) to the file.
- `fprintf(FILE *fp, const char *format, ...)`: Writes formatted text, exactly like `printf`, but to a file stream.

**Example**

Writing Formatted Data

```c
#include <stdio.h>

int main() {
    FILE *fp = fopen("output.txt", "w");
    if (fp == NULL) return 1;

    int age = 20;
```

```c
    char name[] = "Alice";

    // Writing a string
    fputs("Student Record:\n", fp);
    // Writing formatted data
    fprintf(fp, "Name: %s, Age: %d\n", name, age);
    // Writing a single character
    fputc('.', fp);

    fclose(fp);
    return 0;
}
```

### 5.1.6 Reading from a File

Similarly, functions to read text include:

- `fgetc(FILE *fp)`: Reads a single character. Returns `EOF` when the end of the file is reached.
- `fgets(char *str, int n, FILE *fp)`: Reads a string of up to `n-1` characters or until a newline is found.
- `fscanf(FILE *fp, const char *format, ...)`: Reads formatted text from the file.

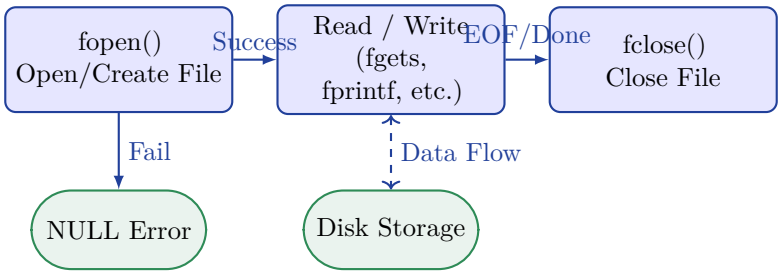



Figure 5.1: Standard File I/O Lifecycle in C

**Example**

Reading an Entire File To read an entire file character by character, we often use a `while` loop that checks for `EOF`.

```c
#include <stdio.h>

int main() {
    FILE *fp = fopen("output.txt", "r");
    if (fp == NULL) {
        printf("File not found!\n");
        return 1;
    }

    char ch;
    // Read character by character until End Of File (EOF)
    while ((ch = fgetc(fp)) != EOF) {
        putchar(ch); // Print character to console
    }

    fclose(fp);
    return 0;
}
```

## 5.2 File Opening Modes and Pointers

### 5.2.1 File Opening Modes

When using `fopen()`, the second argument is a string that specifies the **mode** in which the file should be opened. The mode dictates whether you can read, write, or append data, and what happens if the file already exists or doesn't exist.

**Summary Cheatsheet**

Standard Text File Opening Modes

- `"r"` (Read): Opens an existing text file for reading. The file must exist. Returns `NULL` if it doesn't.
- `"w"` (Write): Opens a text file for writing. If the file exists, its contents are **truncated** (erased completely). If it does not exist, a new file is created.
- `"a"` (Append): Opens a text file for writing in append mode. Data is added to the **end** of the file without erasing existing contents. Creates a new file if it doesn't exist.
- `"r+"` (Read/Update): Opens a text file for both reading and writing. The file must exist. Does not truncate.
- `"w+"` (Write/Update): Opens a text file for reading and writing. Truncates the file if it exists, creates it if it doesn't.
- `"a+"` (Append/Update): Opens a text file for reading and appending. All write operations occur at the end of the file. Creates a new file if it doesn't exist.

**Important / Warning**

Text vs. Binary Modes By default, the modes above open files as **text streams**. To open a file as a **binary stream**, you simply append the letter `b` to the mode string (e.g., `"rb"`, `"wb"`, `"ab+"`). Text streams may perform translations on certain characters (like converting newline `\n` to carriage return + newline `\r\n` on Windows), whereas binary streams process raw bytes with no translation.

### 5.2.2 Understanding File I/O Buffering

A crucial concept in file handling is **buffering**. When you write data to a file using `fprintf` or `fputc`, the data is not immediately written to the physical hard drive. Disk I/O operations are extremely slow compared to CPU speeds. To optimize performance, the C standard library writes data to a temporary block of RAM called a **buffer**. Once the buffer is full, or when the file is closed via `fclose()`, the entire buffer is "flushed" (written) to the disk in a single operation.

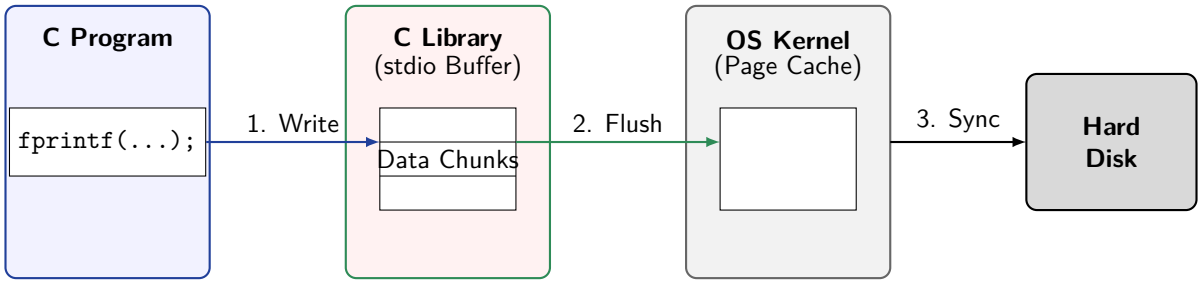



Figure 5.2: File I/O Buffering Architecture. Writing goes through multiple layers of buffers to minimize expensive hardware interactions.

**Key Concept**

The fflush() Function Sometimes you need data to be written to the disk immediately, without waiting for the buffer to fill or the file to close (e.g., writing real-time logs). You can forcefully push the contents of the buffer to the OS by using `fflush(FILE *stream);`.

### 5.2.3 Accessing and Manipulating the File Pointer

The `FILE` structure maintains an internal **file position indicator** (often just called the file pointer). This indicator points to the exact byte location within the file where the next read or write operation will occur.

When you open a file in `"r"` or `"w"` mode, the position indicator starts at byte 0 (the beginning of the file). In `"a"` mode, it starts at the end of the file. C provides functions to control this indicator randomly, enabling **Random Access** to files.

### 1. ftell()

The `ftell()` function returns the current position of the file pointer as a `long int`, representing the number of bytes from the beginning of the file.

```
long int ftell(FILE *stream);
```

### 2. rewind()

The `rewind()` function forcefully moves the file pointer back to the very beginning of the file (byte 0). It is equivalent to `fseek(stream, 0L, SEEK_SET)`.

```
void rewind(FILE *stream);
```

### 3. fseek()

The `fseek()` function is the most powerful tool for random access. It allows you to move the file pointer to any specific location.

```
int fseek(FILE *stream, long int offset, int whence);
```

- **`offset`:** The number of bytes to move. Positive moves forward, negative moves backward.
- **`whence`:** The reference point from which the offset is calculated. It takes one of three macros defined in `<stdio.h>`:
  - `SEEK_SET`: Beginning of the file.
  - `SEEK_CUR`: Current position of the file pointer.
  - `SEEK_END`: End of the file.

**Example**

Random Access using fseek and ftell

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("random.txt", "w+");
    fputs("ProgrammingInC", fp);

    // Pointer is currently at the end. Find the size.
    long size = ftell(fp);
    printf("File size: %ld bytes\n", size); // Prints 14

    // Move pointer to the 11th byte from the start (0-indexed)
    fseek(fp, 11, SEEK_SET);
    fputs("C++", fp); // Overwrites "nC" with "C++"

    // Move pointer back to start
    rewind(fp);

    char buffer[20];
    fscanf(fp, "%s", buffer);
    printf("Modified text: %s\n", buffer); // ProgrammingInC++

    fclose(fp);
    return 0;
}
```

## 5.3 Binary Files and Error Handling

### 5.3.1 Binary Files vs. Text Files

So far, we have largely discussed text files, which store human-readable characters. However, files can also be opened in **binary mode**. Binary files store data in the exact same raw memory format as it exists in the computer's RAM (zeros and ones).

| Text Files | Binary Files |
|---|---|
| Stores data as plain readable text (ASCII characters). | Stores data as raw bytes (binary representation). |
| Character translations may occur (e.g., `\n` to `\r\n` on Windows). | No character translations occur. Data is read/written verbatim. |
| Can be opened and read in basic text editors like Notepad. | Cannot be easily read in text editors; attempting to do so yields gibberish. |
| Generally slower for reading/writing complex structures. | Faster and more memory-efficient for complex structures, arrays, and images. |
| Uses functions like `fprintf`, `fscanf`, `fgets`. | Uses `fread` and `fwrite`. |

Table 5.1: Comparison of Text and Binary Files

### 5.3.2 Reading and Writing in a Binary File

When dealing with arrays, structs, or large blocks of data, binary file operations are superior. Instead of formatting data into strings, you dump a block of memory directly to the disk using `fwrite()` and retrieve it using `fread()`.

#### fwrite()

The `fwrite()` function writes a block of data from an array or a structure into a binary file.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- `ptr`: Pointer to the memory block containing the data to be written.
- `size`: The size, in bytes, of a single element (usually found using `sizeof`).
- `nmemb`: The number of elements to write.
- `stream`: The file pointer to write to.

#### fread()

The `fread()` function reads a block of data from a binary file directly into a memory address.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

**Example**

Storing and Loading C Structures via Binary Files Binary files are incredibly useful for saving database-like records (structs).

```c
#include <stdio.h>

struct Student {
    int roll;
    char name[30];
    float marks;
};

int main() {
    struct Student s1 = {101, "John Doe", 88.5};
    struct Student s2;

    // 1. Write structure to binary file
    FILE *fp = fopen("student.dat", "wb");
    if(fp != NULL) {
        fwrite(&s1, sizeof(struct Student), 1, fp);
        fclose(fp);
    }

    // 2. Read structure from binary file
    fp = fopen("student.dat", "rb");
```

```c
    if(fp != NULL) {
        // Read 1 block of size sizeof(struct Student) into s2
        fread(&s2, sizeof(struct Student), 1, fp);
        fclose(fp);

        printf("Record Loaded:\n");
        printf("Roll: %d, Name: %s, Marks: %.2f\n",
                s2.roll, s2.name, s2.marks);
    }

    return 0;
}
```

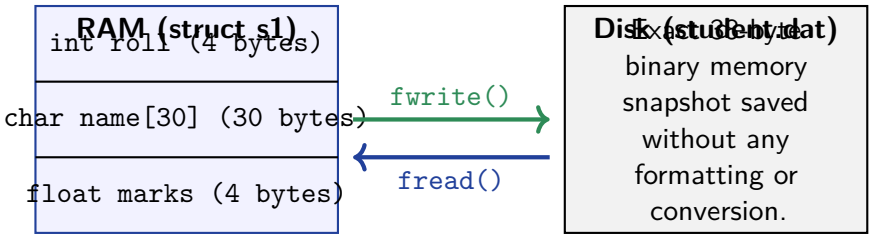



Figure 5.3: Memory mapping during binary file I/O.

### 5.3.3 File Error Handling

When working with files, numerous things can go wrong: the file might not exist, you might not have permission to write to it, or the disk might be full. Robust C programs handle these errors gracefully instead of crashing.

#### Checking EOF and Errors

When reading a file, you need to know if you've stopped reading because the file ended, or because an error occurred (like disk disconnection).

- `int feof(FILE *stream)`: Returns a non-zero value (true) if the End-Of-File indicator is set for the stream.
- `int ferror(FILE *stream)`: Returns a non-zero value (true) if the error indicator is set for the stream.
- `void clearerr(FILE *stream)`: Clears both the EOF and error indicators for the stream.

**Key Concept**

feof() vs EOF macro The `EOF` macro is returned by functions like `fgetc()` when a read fails. However, a read can fail either due to reaching the end of the file **OR** due to a read error. To distinguish between the two, you use `feof()` and `ferror()` after a read function returns `EOF`.

#### Using perror()

When a standard library function like `fopen()` fails, it sets a global integer variable called `errno` (defined in `<errno.h>`). The `perror()` function translates this error number into a human-readable string and prints it to the standard error stream.

```c
void perror(const char *str);
```

**Example**

Comprehensive Error Handling

```c
#include <stdio.h>
#include <errno.h>

int main() {
    // Attempting to read a file that doesn't exist
```

```c
    FILE *fp = fopen("ghost_file.txt", "r");

    if (fp == NULL) {
        // perror automatically appends the system error message
        perror("Failed to open file");
        // Output might be: "Failed to open file: No such file or directory"
        return 1;
    }

    // Example of checking errors during reading
    char buffer[100];
    if (fgets(buffer, sizeof(buffer), fp) == NULL) {
        if (feof(fp)) {
            printf("Reached end of file immediately.\n");
        } else if (ferror(fp)) {
            perror("Error reading from file");
            clearerr(fp); // Clear the error state
        }
    }

    fclose(fp);
    return 0;
}
```