The Little Book of C

Version 0.3.0

Duc-Tam Nguyen

2025-10-02

Table of contents

Content

A 100-step journey to learn C from first principles

24. The `switch` Statement
25. The `while` Loop
26. The `for` Loop
27. The `do-while` Loop
28. Breaking and Continuing Loops
29. Using `goto` Safely (and Why to Avoid It)
30. Patterns of Control Flow

Chapter 4. Functions and Scope

31. Defining and Calling Functions
32. Function Parameters and Return Values
33. Local and Global Variables
34. Scope and Lifetime
35. Header Declarations (`.h` files)
36. Pass by Value Explained
37. Recursion and Base Cases
38. Function Prototypes and Order
39. Inline Functions
40. Organizing Code with Functions

Chapter 5. Arrays and Strings

41. Declaring Arrays
42. Indexing and Bounds
43. Multidimensional Arrays
44. Iterating over Arrays
45. Strings as Character Arrays
46. String Literals and Null Terminators
47. Common String Functions (`strlen`, `strcpy`, `strcmp`)
48. Inputting Strings
49. Arrays vs. Pointers (A Gentle Intro)
50. Common Array Pitfalls

Chapter 6. Pointers and Memory

51. What Is a Pointer
52. The Address-of (`&`) and Dereference (`*`) Operators
53. Pointer Arithmetic
54. Arrays and Pointers Revisited
55. Function Parameters with Pointers

56. Dynamic Memory Allocation with `malloc`
57. Using `free` Safely
58. Pointer to Pointer
59. `NULL` and Dangling Pointers
60. Debugging Memory Errors

Chapter 7. Structures and Modular Design

61. Defining `struct` Types
62. Accessing Structure Members
63. Structures and Functions
64. Nested Structures
65. Arrays of Structures
66. Typedefs for Simpler Names
67. Enums and Symbolic Constants
68. Unions and Shared Memory
69. Organizing Code into Modules
70. Splitting Code into `.c` and `.h` Files

Chapter 8. The Power of the Preprocessor

71. What Is the Preprocessor
72. `#include` and Header Guards
73. Defining Macros with `#define`
74. Working with Paths and Filenames
75. Conditional Compilation (`#if`, `#ifdef`)
76. Function-like Macros
77. Debugging with `#error` and `#warning`
78. Built-in Macros (`__FILE__`, `__LINE__`)
79. The Compilation Pipeline (Preprocess → Compile → Link)
80. Balancing Macros and Functions

Chapter 9. Files, Tools, and Concurrency

81. File I/O Basics: `fopen`, `fclose`
82. Reading and Writing Files
83. Working with Binary Files
84. Error Handling in File Operations
85. Command-Line Arguments
86. Using `make` and Makefiles
87. Debugging with `gdb`

88. Understanding Linking and Libraries
89. Simple Threads with `<threads.h>`
90. Synchronization and Data Safety

Chapter 10. Putting It All Together

91. Mini Project 1: Text Analyzer
92. Mini Project 2: Guessing Game
93. Mini Project 3: Calculator
94. Mini Project 4: File Copy Utility
95. Mini Project 5: Simple Logger
96. Mini Project 6: Contact Book
97. Mini Project 7: Matrix Operations
98. Mini Project 8: JSON-like Parser
99. Mini Project 9: Mini Shell
100. Mini Project 10: Tiny HTTP Server

The Book

Chapter 1. Getting started with C

1. What Is C and Why Learn It

C is a language that shaped the modern world of computing.
It is small, fast, and close to the machine, yet expressive enough to build entire operating systems, compilers, databases, and games.
Learning C means learning how computers actually work, memory, data, and control flow, all laid bare, without layers of abstraction in the way.

C has been around for decades because it is both powerful and simple.
It gives you a direct line to the processor and memory, and rewards careful, thoughtful programming.
Almost every modern language borrows ideas from C, so mastering it gives you a foundation for understanding them all.

When you learn C, you learn to think like a systems programmer:

- How data is stored in memory
- How instructions are executed step by step
- How to reason about performance, correctness, and clarity

C teaches discipline, precision, and clarity. It is not only a tool, but a teacher.

Tiny Code

Let's see the smallest C program possible:

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Save it as `hello.c`, then compile and run:

```
gcc hello.c -o hello
./hello
```

You'll see:

```
Hello, world!
```

This is your first conversation with the machine.

Why It Matters

Every journey in programming begins with understanding how to talk to the computer. C helps you do that at the deepest level. It shows how your code becomes instructions and how those instructions shape the behavior of your program.

Even if you move on to higher-level languages, C gives you a foundation to understand what happens beneath the surface, why memory matters, why types matter, and how to write efficient, predictable code.

Try It Yourself

1. Modify the program to print your name instead of "Hello, world!"
2. Remove the `return 0;` line. What happens when you compile and run?
3. Add another `printf` line. Can you print two messages?
4. Change `int main(void)` to `int main()`. Does it still compile?
5. Try using `puts("Hello");` instead of `printf`. What's the difference?

Learning C begins with curiosity and courage. You'll write small programs at first, but each one will bring you closer to understanding the heart of computing.

2. Installing a C Compiler

Before you can run any C program, you need a compiler, a tool that translates human-readable code into machine instructions your computer can execute. C doesn't run through an interpreter like Python or JavaScript. Every program must be compiled into an executable file.

Once your compiler is ready, you can transform code like `hello.c` into a working program with a single command.

2.1 What a Compiler Does

A C compiler performs several steps:

1. Preprocessing, expands macros and includes headers
2. Compilation, translates C code into assembly
3. Assembly, converts assembly into object code
4. Linking, combines object code with libraries into an executable

You don't need to do these steps manually; the compiler does it all when you run `gcc hello.c -o hello`. Still, understanding them helps you appreciate how C turns text into software.

2.2 Choosing a Compiler

There are several standard C compilers:

- GCC (GNU Compiler Collection), available on Linux, macOS, and Windows (via MinGW or WSL)
- Clang, fast, modern, and available on macOS and Linux
- MSVC (Microsoft Visual C++), part of Visual Studio on Windows

You can use any of these; they all follow the same standard and will work for this book.

2.3 Installing on Your System

On Linux Most distributions include GCC. Try:

```
gcc --version
```

If it's missing, install it:

```
sudo apt install build-essential   # Debian/Ubuntu
sudo dnf install gcc               # Fedora
```

On macOS Install the Xcode Command Line Tools:

```
xcode-select --install
```

This includes Clang, which works exactly like GCC for our purposes.

On Windows You have a few options:

- MinGW, lightweight and easy to install: mingw-w64.org
- WSL, install a Linux environment inside Windows and use GCC
- Visual Studio, install the C++ development workload

For simplicity, MinGW or WSL are recommended if you follow examples in this book.

Tiny Code

After installation, verify your compiler works:

```
gcc --version
```

Then compile your first program:

```
gcc hello.c -o hello
./hello
```

If you see `Hello, world!`, your toolchain is ready.

Why It Matters

Your compiler is the bridge between your ideas and the machine. It doesn't just check your syntax, it builds your program step by step, optimizing and linking it with system libraries. Learning to use the compiler early helps you troubleshoot, experiment, and take control of how your programs are built.

You'll use the same compiler to handle every program in this book, from tiny scripts to full projects.

Try It Yourself

1. Run `gcc --version` or `clang --version` to check your setup.
2. Compile `hello.c` with different output names using `-o`, for example `-o greet`.
3. Add a missing semicolon in your code. What error message do you get?
4. Try compiling with `clang hello.c -o hello` if you have Clang installed.
5. Run your program from a different directory to understand relative paths.

Once your compiler is working, you're ready to dive into real programming, learning how to express ideas in code, line by line.

3. Writing Your First C Program

Now that your compiler is ready, it's time to write a complete C program from scratch. A C program is just plain text that follows specific rules of structure and syntax. Every program begins the same way, with a main function, the entry point where execution starts.

This section will help you build your first working program step by step.

3.1 The Structure of a C Program

Every C program follows a basic shape:

```c
#include <stdio.h>

int main(void) {
    // your code here
    return 0;
}
```

- `#include <stdio.h>` tells the compiler you want to use the Standard Input/Output library.
- `int main(void)` defines the main function, where your program begins.
- `{ ... }` marks the start and end of the function body.
- `return 0;` signals that the program finished successfully.

3.2 Writing and Running the Program

Open a text editor and type this code:

```c
#include <stdio.h>

int main(void) {
    printf("Welcome to C!\n");
    return 0;
}
```

Save the file as `welcome.c`. Now compile and run it:

```
gcc welcome.c -o welcome
./welcome
```

You should see:

```
Welcome to C!
```

3.3 Understanding `printf`

The function `printf` is used to print messages to the screen. It takes a format string and optional arguments.

For now, you can think of it as a simple way to display text. The special symbol `\n` adds a newline, moving the cursor to the next line.

Example:

```c
printf("Hello\nWorld\n");
```

Output:

```
Hello
World
```

3.4 Comments and Readability

Comments help others (and your future self) understand your code.

```c
// This is a single-line comment

/*
 This is a multi-line comment.
 You can explain logic here.
*/
```

Comments are ignored by the compiler but valued by programmers.

Tiny Code

Try this short example:

```c
#include <stdio.h>

int main(void) {
    // Print two lines
    printf("C is powerful.\n");
    printf("Let's start learning!\n");
    return 0;
}
```

Compile and run:

```
gcc start.c -o start
./start
```

You'll see two lines of output.

Why It Matters

Your first C program may seem simple, but it contains every essential element of real software:

- A clear entry point (`main`)
- Standard library usage (`stdio.h`)
- Output to the user (`printf`)
- A clean exit (`return 0`)

This structure forms the backbone of all C programs, no matter how large they grow.

Try It Yourself

1. Change the text in `printf` to your favorite quote.
2. Add a second `printf` call with a different message.
3. Remove the `\n` and see how the output changes.
4. Add comments above each line explaining what it does.
5. Delete `return 0;` and recompile. Does it still work? Why?

Every big program begins as a small one. Once you can write and run code, you can build anything, one line at a time.

4. Understanding `main` and Return Values

Every C program starts its journey in one special place, the `main` function. When your program runs, the operating system calls `main` first. What happens inside determines everything that follows: what the program does, what it prints, and what it returns when finished.

Let's explore how `main` works and why its return value matters.

4.1 The Role of `main`

`main` is the entry point. Without it, the compiler cannot link your program into a valid executable.

Basic form:

```
int main(void) {
    // your code here
    return 0;
}
```

- `int` means `main` returns an integer value to the operating system.
- `void` means `main` takes no arguments.
- The curly braces `{}` contain the code that runs when the program starts.

4.2 Returning Values

The value returned by `main` tells the operating system whether the program succeeded or failed.

By convention:

- `return 0;` means success
- Nonzero (like `return 1;`) means error or failure

You can check this in a terminal:

```
./program
echo $?
```

The `echo $?` command prints the exit code of the last program.

4.3 Variants of `main`

There are two valid forms:

```
int main(void) { ... }   // No arguments
int main(int argc, char *argv[]) { ... }   // With command-line arguments
```

We'll use the simpler `void` form for now. You'll learn the version with `argc` and `argv` later when working with command-line tools.

4.4 Flow of Execution

When you run your program, this happens:

1. The operating system loads the program into memory
2. It calls `main`
3. Your code runs line by line
4. When `return` is reached (or the end of the function), control returns to the OS

This is the simplest model of program execution, clear, linear, and predictable.

Tiny Code

Try this small experiment:

```
#include <stdio.h>

int main(void) {
    printf("Program is running...\n");
    return 0;
}
```

Compile and run:

```
gcc run.c -o run
./run
echo $?
```

You'll see `Program is running...` and then `0`, confirming success.

Now change the code:

```c
return 1;
```

Run again. `echo $?` will print `1`.

Why It Matters

Every program must begin and end cleanly. Returning the correct value helps you write software that interacts well with other programs and scripts. Professional developers often check return codes in automation, testing, and system tools.

Understanding `main` teaches you that programs are conversations with the operating system, you say what you did through your exit code.

Try It Yourself

1. Write a program that returns `0` and prints "All good."
2. Change it to return `1` and print "Something went wrong."
3. Run both and compare `echo $?` outputs.
4. Remove the `return` line entirely. What does your compiler do?
5. Replace `int main(void)` with `void main(void)`. Does it compile? Why is `int main` preferred?

When you understand `main`, you understand how your program begins, how it ends, and how it communicates success or failure, the three pillars of every executable.

5. Printing with `printf`

One of the first things every C program learns to do is print text to the screen. The function `printf` (print formatted) is your main tool for displaying messages, numbers, and results. It's part of the Standard I/O Library (`stdio.h`) and one of the most useful functions in all of C.

Let's explore how `printf` works and how to control its output.

5.1 The Basics

A simple `printf` call looks like this:

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

The text inside quotes is a string literal. The `\n` at the end is a newline character, it moves the cursor to the next line.

Output:

```
Hello, world!
```

5.2 Strings and Escape Sequences

Inside a string, you can include special sequences beginning with a backslash:

Sequence	Meaning
`\n`	New line
`\t`	Tab
`\\`	Backslash
`\"`	Double quote
`\a`	Alert (beep)

Example:

```c
printf("Name:\tAlice\nAge:\t20\n");
```

Output:

```
Name:   Alice
Age:    20
```

5.3 Printing Numbers

`printf` can also print numbers. You use format specifiers to tell it what kind of value to print.

Specifier	Type	Example
`%d`	int	`printf("%d", 42);`
`%f`	double	`printf("%f", 3.14);`
`%c`	char	`printf("%c", 'A');`
`%s`	string	`printf("%s", "Hi");`

Example:

```
int age = 30;
printf("I am %d years old.\n", age);
```

Output:

```
I am 30 years old.
```

5.4 Combining Text and Values

You can mix text and placeholders freely:

```
int x = 5, y = 7;
printf("Sum of %d and %d is %d\n", x, y, x + y);
```

Each `%` symbol corresponds to an argument after the string. They are matched in order.

Output:

```
Sum of 5 and 7 is 12
```

5.5 Formatting Numbers

You can control width and precision:

```c
double pi = 3.1415926535;
printf("pi = %.2f\n", pi);
```

Output:

```
pi = 3.14
```

Here `%.2f` means "print 2 digits after the decimal."

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int apples = 5;
    double price = 1.25;

    printf("I bought %d apples at $%.2f each.\n", apples, price);
    printf("Total cost: $%.2f\n", apples * price);

    return 0;
}
```

Compile and run to see formatted values printed neatly.

Why It Matters

Output is how programs communicate. With `printf`, you can inspect variables, debug logic, and make programs interactive. It's not just about text, it's about making invisible computation visible.

As you learn more, you'll use `printf` constantly to understand what your code is doing.

Try It Yourself

1. Print your name, age, and favorite number using `%s` and `%d`.
2. Experiment with `%f` and `%.3f` to show different decimal places.
3. Print `\t` and `\n` to see how tabs and newlines affect layout.
4. Mix characters and numbers: `printf("Char: %c, Code: %d", 'A', 'A');`
5. Write a sentence combining three variables of different types.

Every program starts by talking to you through text. Mastering `printf` teaches you to control that voice, clear, precise, and expressive.

6. Comments and Code Readability

Code is written for humans first, computers second. The compiler doesn't need explanations, but future readers (including you) will. That's where comments come in, notes you leave inside your code to describe what's happening and why.

Good comments make your code easier to read, maintain, and extend. They turn lines of logic into a story that others can follow.

6.1 Why Comment Your Code

Comments help you:

- Explain *why* something is written a certain way
- Mark complex sections for future reference
- Remind yourself of unfinished work
- Communicate intent to teammates

A program without comments might still work, but it's harder to understand. A well-commented program tells both the *what* and the *why*.

6.2 Types of Comments in C

C supports two styles of comments:

1. Single-line comment

```
// This is a single-line comment
```

2. Multi-line comment

```
/* This is a
   multi-line comment */
```

Single-line comments are best for short notes. Multi-line comments are useful for describing longer ideas, algorithms, or sections.

6.3 Where to Use Comments

Use comments:

- Above a function or block to describe its purpose
- Beside tricky or non-obvious code
- At the top of a file to outline the program
- To mark TODOs for future changes

Example:

```
#include <stdio.h>

int main(void) {
    // Print a friendly greeting
    printf("Hello, world!\n");

    /* TODO:
       Add user input here later
    */

    return 0;
}
```

6.4 What Not to Do

Avoid obvious or redundant comments:

```
int x = 10; // set x to 10  ← unnecessary
```

Instead, focus on intent:

```
int retries = 10; // maximum number of connection attempts
```

Comments should add meaning, not repeat the code.

Tiny Code

Try this short example:

```c
#include <stdio.h>

int main(void) {
    // Program to calculate total price

    int items = 3;       // number of products
    double price = 2.5;  // price per item
    double total = items * price; // compute total

    printf("Total: $%.2f\n", total);

    return 0; // program ended successfully
}
```

Compile and run, then read it as if you were a new developer. Every line tells a story.

Why It Matters

Code is read more often than it's written. A comment today can save hours of confusion tomorrow. By documenting your reasoning, you make your programs friendlier, clearer, and easier to improve.

Comments are part of your craft, invisible to the machine, invaluable to the human.

Try It Yourself

1. Add single-line comments to describe each step of a simple math program.
2. Use a multi-line comment at the top of your file to describe its purpose.
3. Write a `TODO` comment for a feature you plan to add later.
4. Add a confusing variable name, then explain it with a comment.
5. Remove redundant comments that don't add information.

Readable code is thoughtful code. Every comment is a note to your future self, write them with care.

7. Variables and Basic Types

Programs need a way to store information, numbers, characters, and other data that change as your program runs. In C, you do this with variables. A variable is a named piece of memory that can hold a value.

Before you can use a variable, you must declare it and tell C what type of data it will store.

7.1 What Is a Variable

A variable has three parts:

1. Name, what you call it
2. Type, what kind of data it holds
3. Value, what's stored inside

For example:

```
int age = 21;
```

- `int` is the type (integer)
- `age` is the name
- `21` is the value

This line creates a space in memory named `age` to hold an integer.

7.2 Declaring and Initializing

You declare a variable by writing its type followed by its name:

```
int count;
```

You can also give it a starting value (initialization):

```
int count = 10;
```

You may declare multiple variables of the same type in one line:

```
int a = 1, b = 2, c = 3;
```

But clarity is often better than compactness.

7.3 Basic Data Types

C has several fundamental types:

Type	Description	Example
`int`	Whole numbers	`int x = 5;`
`float`	Decimal numbers (less precise)	`float g = 9.8;`
`double`	Decimal numbers (more precise)	`double pi = 3.1416;`
`char`	Single character	`char grade = 'A';`
`_Bool`	Boolean value (0 or 1)	`_Bool ok = 1;`

7.4 Naming Variables

Names should be:

- Descriptive: `score`, `height`, `count`
- Lowercase with underscores: `total_sum`
- Not keywords like `int`, `return`, or `if`

Valid examples:

```
int total;
double average_height;
char first_letter;
```

Invalid examples:

```
int 3number;   // cannot start with a digit
int return;    // keyword not allowed
```

7.5 Changing Values

Once declared, you can assign new values:

```
int score = 10;
score = 15; // overwrite with new value
```

The latest assignment replaces the old one.

Tiny Code

Here's a short example:

```c
#include <stdio.h>

int main(void) {
    int apples = 5;
    double price = 2.5;
    char currency = '$';

    double total = apples * price;

    printf("Total cost: %c%.2f\n", currency, total);

    return 0;
}
```

Output:

```
Total cost: $12.50
```

Why It Matters

Variables are the building blocks of all programs. They hold the data your code will process, transform, and display. By understanding types, you gain control over how much memory your data uses and how the compiler interprets it.

Knowing what each variable represents keeps your programs precise and readable.

Try It Yourself

1. Declare an `int` called `age` and print it.
2. Add a `double` called `height` and print it with `%.2f`.
3. Create a `char` called `initial` and print it using `%c`.
4. Change a variable's value and print before and after.
5. Try naming a variable with an invalid character and read the error message.

Variables give life to your programs. Once you can store and name data, you can begin to calculate, compare, and create.

8. Declaring and Initializing Variables

Declaring a variable tells the compiler what kind of data you want to store and what name you'll use to refer to it. Initializing a variable gives it a starting value. Both are essential: declaration defines the shape, initialization gives it life.

In C, uninitialized variables contain garbage values, random data left in memory, so it's always a good habit to initialize them before use.

8.1 Declaration

To declare a variable, specify its type followed by a name:

```
int count;
double price;
char letter;
```

At this stage, memory is reserved for each variable, but their contents are undefined.

If you print `count` before giving it a value, you'll get unpredictable results:

```
printf("%d\n", count); // undefined behavior
```

Always assign a value before using a variable.

8.2 Initialization

You can give a variable a value as soon as you declare it:

```
int count = 10;
double pi = 3.14159;
char grade = 'A';
```

This is called initialization, setting the initial value at creation. It's the safest, clearest way to define variables.

8.3 Combined Declarations

You can declare multiple variables of the same type in one line:

```c
int a = 1, b = 2, c = 3;
```

This is valid, but avoid combining different types:

```c
int x = 1, y = 2;  // ok
int a = 1, double b = 2.0;  // invalid
```

Each type must be declared separately.

8.4 Initialization Later

Sometimes you know a variable's type, but not its value yet:

```c
int score;
score = 100; // assign later
```

That's fine, as long as you assign before use.

8.5 Constants vs. Variables

If a value should never change, mark it as constant:

```c
const double TAX_RATE = 0.08;
```

Trying to modify it later causes a compile error. Constants make your code safer and clearer.

Tiny Code

Try this short program:

```c
#include <stdio.h>

int main(void) {
    int age = 25;           // initialized
    double height;          // declared
    height = 1.75;          // assigned later
    const double PI = 3.14; // constant

    printf("Age: %d\n", age);
```

```c
    printf("Height: %.2f m\n", height);
    printf("PI: %.2f\n", PI);

    return 0;
}
```

Output:

```
Age: 25
Height: 1.75 m
PI: 3.14
```

Why It Matters

Uninitialized variables are one of the most common beginner mistakes. They lead to unpredictable behavior and subtle bugs. By initializing variables early, you ensure your program behaves consistently and is easier to read.

Constants also help communicate intent, if something should never change, declare it so.

Try It Yourself

1. Declare and initialize three variables: `int`, `double`, and `char`.
2. Print them all with correct format specifiers.
3. Create a `const int` called `MAX_SCORE` and try changing it. What happens?
4. Declare a variable without initializing it, print it, and observe the output.
5. Fix your program by initializing every variable properly.

Initialization turns empty memory into meaningful data. By giving every variable a clear starting point, you build reliable and predictable programs.

9. Compiling and Running Programs

Writing code is only the first half of programming. To see your program in action, you must compile it, transform human-readable C code into machine code the computer can execute. This process is handled by a compiler, and understanding how it works helps you fix errors and control your builds.

9.1 From Source to Executable

A C file like `hello.c` is source code, plain text. To run it, you must translate it into a binary executable.

When you use a command like:

```
gcc hello.c -o hello
```

the compiler does several steps internally:

1. Preprocessing, handles `#include` and `#define`
2. Compilation, translates code into assembly
3. Assembly, converts assembly into object code (`.o` file)
4. Linking, combines your code with libraries (like `stdio`)

The final result is an executable file (`hello`) that your system can run.

9.2 Using GCC or Clang

Most systems use GCC or Clang to compile C programs. They share the same basic commands:

```
gcc file.c -o program
clang file.c -o program
```

Run the program with:

```
./program
```

If you omit `-o`, the compiler outputs a default file called `a.out`.

```
gcc hello.c
./a.out
```

9.3 Handling Errors

If there's a mistake, the compiler prints an error:

```
gcc bad.c -o bad
bad.c: In function 'main':
bad.c:3:5: error: expected ';' before 'return'
```

Read error messages carefully, they show file, line number, and reason.

Example:

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!") // missing semicolon
    return 0;
}
```

Fix: add `;` at the end of the `printf` line.

9.4 Warnings

Even if your code compiles, the compiler may issue warnings:

```
gcc test.c -o test
test.c:4:5: warning: unused variable 'x'
```

Warnings aren't fatal but often signal bugs or bad habits. You can enable stricter checks with:

```
gcc -Wall -Wextra -pedantic file.c -o program
```

Always treat warnings seriously.

9.5 Running the Program

Once compiled, run your program from the terminal:

```
./program
```

The computer now executes your compiled code line by line.

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int x = 5;
    int y = 10;
    printf("Sum: %d\n", x + y);
    return 0;
}
```

Compile and run:

```
gcc sum.c -o sum
./sum
```

Output:

```
Sum: 15
```

Why It Matters

Understanding the build process helps you troubleshoot and improve your workflow. You'll learn what errors mean, how linking works, and how to organize large projects. Compiling isn't a black box, it's a step-by-step transformation you control.

Each time you compile, you move from *idea* to *executable reality*.

Try It Yourself

1. Write a simple program that prints a message. Compile and run it.
2. Remove a semicolon and observe the compiler error.
3. Add `-Wall` and fix all warnings.
4. Compile two programs: one with `-o hello` and one without. Compare results.
5. Introduce a typo in `printf` and read the error carefully.

Compilation is the bridge from writing to running. Once you master it, your ideas can come alive on any computer.

10. Common Beginner Mistakes

Every new C programmer stumbles on the same kinds of problems. That's perfectly normal, these mistakes are how you learn what the compiler expects and how C actually works. Recognizing them early saves hours of confusion and helps you debug with confidence.

Let's look at the most common ones, why they happen, and how to fix them.

10.1 Missing Semicolons

Every C statement must end with a semicolon:

```
printf("Hello, world!")   //   missing semicolon
```

Fix it:

```
printf("Hello, world!");  //   correct
```

The compiler will stop with an error like "expected ';' before 'return' ". Whenever you see a syntax error, first check for a missing `;`.

10.2 Forgetting `#include`

If you use `printf` or `scanf` without including `<stdio.h>`, you'll see errors like:

```
undefined reference to printf
```

Always include necessary headers:

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

This tells the compiler what functions you're using and their correct signatures.

10.3 Using Uninitialized Variables

C doesn't automatically set variables to zero. If you forget to give them a value, they contain random memory data:

```c
int x;
printf("%d\n", x); //  unpredictable result
```

Fix it by initializing:

```c
int x = 0;
```

Always initialize before use.

10.4 Wrong Format Specifiers

Each format specifier in `printf` must match the type:

```c
int x = 10;
printf("%f\n", x); //  wrong: %f expects double
```

Fix:

```c
printf("%d\n", x); //  matches int
```

Mismatched specifiers lead to nonsense output or crashes.

10.5 Forgetting Return Type in `main`

Always declare `main` with a return type:

```c
main() { ... } //  old-style declaration
```

Fix:

```c
int main(void) { ... } //  standard form
```

Without `int`, the compiler may issue warnings or misinterpret the function.

10.6 Mismatched Braces or Parentheses

Each `{` must have a matching `}`. If you see errors like *"expected '}' at end of input"*, check your brackets.

Good editors highlight matching pairs, use them to spot missing ones quickly.

10.7 Using = Instead of ==

= assigns, == compares.

```
if (x = 5) { ... } //  assigns 5 to x
if (x == 5) { ... } //  checks equality
```

Assignment inside conditions can cause subtle bugs. Many compilers warn about this if you enable `-Wall`.

10.8 Ignoring Warnings

Warnings are hints from the compiler. Even if your program runs, warnings often mean hidden problems.

Always compile with:

```
gcc -Wall -Wextra -pedantic file.c -o file
```

and read every message carefully.

Tiny Code

Here's a buggy example, can you spot the mistakes?

```c
#include <stdio.h>

int main(void) {
    int age
    printf("Age is: %d\n", age);
    return 0;
}
```

Problems:

1. Missing semicolon after `int age`
2. `age` not initialized before printing

Fixed version:

```c
#include <stdio.h>

int main(void) {
    int age = 25;
    printf("Age is: %d\n", age);
    return 0;
}
```

Why It Matters

Mistakes are not failures, they're feedback. Every error message teaches you something about C's rules and precision. By studying common errors, you develop instincts that prevent them before they happen.

Learning to fix bugs is as valuable as writing code itself.

Try It Yourself

1. Write a small program and intentionally forget a semicolon. Read the error.
2. Use the wrong format specifier, then correct it.
3. Declare a variable but don't initialize it, print it, then fix it.
4. Remove a brace and see what error the compiler gives.
5. Turn on `-Wall` and clean up all warnings.

Every expert started by breaking their code. The difference is that they kept fixing it.

Chapter 2. Working with Data

11. Integers, Floats, and Characters

Every program works with data, and in C, each piece of data has a type. The three most common types you'll use from the start are integers, floating-point numbers, and characters. They represent whole numbers, real numbers, and single symbols, the building blocks of computation.

Let's explore each one and how to use them correctly.

11.1 Integers

An integer stores whole numbers, positive or negative, without decimals.

```
int apples = 5;
int temperature = -10;
```

`int` typically uses 4 bytes of memory, storing values roughly between -2 billion and +2 billion (platform dependent).

Other integer types:

- `short`, smaller range, 2 bytes
- `long`, larger range, often 8 bytes
- `unsigned int`, only nonnegative numbers, doubles the positive range

Examples:

```
short s = 100;
long big = 1000000L;
unsigned int u = 42U;
```

Suffixes like `L` or `U` make intent clear.

11.2 Floating-Point Numbers

A float or double stores decimal numbers.

```
float pi = 3.14f;
double radius = 2.5;
```

- `float` is single-precision (around 6–7 digits)
- `double` is double-precision (around 15–16 digits)

Always use `double` when you need accuracy. Use the suffix `f` for floats (`3.14f`), or leave it off for doubles.

Operations:

```
double area = 3.14 * radius * radius;
```

11.3 Characters

A character (`char`) stores a single symbol, like `'A'` or `'3'`. Use single quotes `' '`, not double quotes `" "` which are for strings.

```
char grade = 'A';
char symbol = '#';
```

Under the hood, a `char` is actually a small integer representing an ASCII code:

```
char letter = 'A';
printf("%c %d\n", letter, letter);
```

Output:

```
A 65
```

You can treat characters as numbers and perform arithmetic on them:

```
char next = letter + 1; // 'B'
```

11.4 Mixed Operations

When you combine types, C automatically promotes smaller types to larger ones.

Example:

```
int x = 5;
double y = 2.0;
double result = x + y; // x converted to double
```

Be careful when dividing integers:

```
int a = 5, b = 2;
printf("%d\n", a / b); // prints 2 (integer division)
```

Use a float or double for precise results:

```c
printf("%.2f\n", (double)a / b); // prints 2.50
```

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int age = 25;
    double height = 1.75;
    char initial = 'J';

    printf("Age: %d\n", age);
    printf("Height: %.2f m\n", height);
    printf("Initial: %c\n", initial);

    return 0;
}
```

Output:

```
Age: 25
Height: 1.75 m
Initial: J
```

Why It Matters

Choosing the right type helps your program store values accurately and efficiently. Integers count, floats measure, and chars label. As your programs grow, you'll combine these types in arrays, structures, and functions, everything starts here.

C gives you control over precision, size, and interpretation, essential for reliable software.

Try It Yourself

1. Declare three variables: `int score`, `double temperature`, and `char grade`. Print them.
2. Perform integer division (`7 / 2`) and floating-point division (`7.0 / 2`). Compare results.
3. Print both `%c` and `%d` of `'A'` to see its ASCII value.
4. Create an `unsigned int` and assign `-1`. What happens?

5. Add two `char` letters (`'A' + 1`) and print the result.

Numbers count the world. Characters name it. Together, they give your programs a voice.

12. Type Conversions and Casting

C is a strongly typed language, which means every value has a specific type, and operations depend on it. However, C also allows type conversion, turning one type into another, either automatically or manually. Understanding when and how conversions happen keeps your programs correct and precise.

12.1 Implicit Conversions

An implicit conversion happens automatically when C promotes one type to another during an expression.

Example:

```
int x = 5;
double y = 2.0;
double result = x + y; // x is promoted to double
```

The integer `x` is automatically converted to `double` before addition. This rule is called type promotion.

When different types meet, C usually promotes smaller or narrower types to larger ones to preserve precision.

Promotion hierarchy (small → large):

```
char → int → float → double
```

12.2 Integer Division and Promotion

If both operands are integers, the result is an integer, even if you store it in a float:

```
int a = 5, b = 2;
float result = a / b; // result = 2.0, not 2.5
```

Why? Because `a / b` is done as integer division first.

To fix it, convert one operand:

```
float result = (float)a / b; // now result = 2.5
```

Promotion must happen before the operation.

12.3 Explicit Casting

You can force a conversion using a cast:

```
(type)expression
```

Example:

```
int total = 7, count = 2;
double average = (double)total / count;
```

`(double)total` tells the compiler to treat `total` as a double before division.

Without casting, `7 / 2` would give `3`, not `3.5`.

12.4 Narrowing Conversions

Converting from a larger type to a smaller one can lose information:

```
double pi = 3.14159;
int truncated = (int)pi; // truncated = 3
```

The fractional part is dropped.

Similarly:

```
int big = 1000;
char small = (char)big; // may overflow
```

If `char` holds only -128 to 127, `1000` wraps around to another value. Use narrowing conversions carefully.

12.5 Mixed-Type Expressions

In mixed operations, C promotes operands to a common type before computing:

```
int a = 3;
float b = 4.5;
double c = 2.0;
double result = a + b * c; // a→float→double
```

The compiler ensures both sides of an operator are of compatible types.

Tiny Code

Try this short program:

```
#include <stdio.h>

int main(void) {
    int a = 5, b = 2;

    printf("Integer division: %d / %d = %d\n", a, b, a / b);
    printf("Float division (cast): %d / %d = %.2f\n", a, b, (double)a / b);

    double pi = 3.14159;
    int truncated = (int)pi;
    printf("Truncated value of pi: %d\n", truncated);

    return 0;
}
```

Output:

```
Integer division: 5 / 2 = 2
Float division (cast): 5 / 2 = 2.50
Truncated value of pi: 3
```

Why It Matters

Type conversion is subtle but powerful. It decides whether your results are precise or rounded, correct or wrong. Knowing when to cast ensures your calculations match your intentions. Professional C programmers cast deliberately, never accidentally.

Try It Yourself

1. Compute `7 / 3` as both integer and floating-point division.
2. Cast a `double` to `int` and observe truncation.
3. Print `(char)65` and see what character it produces.
4. Store `300` in a `char` and print it, what happens?
5. Experiment with `(float)(a / b)` vs `(float)a / b`. What's the difference?

Casting gives you control over precision and intent. In C, a single cast can turn confusion into clarity.

13. Constants and Literals

In programming, some values never change. They might represent physical constants, configuration values, or fixed parameters your program depends on. In C, these unchanging values are called constants, and the values you write directly in code, like `42` or `'A'`, are called literals.

Learning to define and use constants properly makes your code clearer, safer, and easier to maintain.

13.1 What Are Constants

A constant is a named value that cannot be modified once set. You declare one with the keyword `const`:

```
const int DAYS_IN_WEEK = 7;
const double PI = 3.14159;
```

Trying to change it later will cause a compiler error:

```
PI = 3.14;  //  error: assignment of read-only variable
```

Constants behave like regular variables in every way except that their values cannot change.

13.2 Why Use Constants

Constants improve your code in three ways:

1. Clarity – names describe what values mean
2. Safety – prevents accidental changes
3. Maintainability – change once, apply everywhere

Instead of repeating a literal value:

```
area = 3.14159 * r * r;
```

Use a constant:

```
const double PI = 3.14159;
area = PI * r * r;
```

Now, if you ever adjust the value, you only change it in one place.

13.3 Literals

A literal is a value written directly in the code. Each literal has a type determined by its form:

Type	Example	Description
Integer	`42`, `-10`, `0xFF`	decimal, negative, hexadecimal
Floating	`3.14`, `2e3`	decimal or scientific notation
Character	`'A'`, `'9'`, `'\n'`	single character
String	`"Hello"`	sequence of characters

You can also use suffixes:

- `L` for `long`: `100L`
- `U` for `unsigned`: `42U`
- `F` for `float`: `3.14F`

Example:

```
float f = 1.0F;
unsigned int u = 42U;
long n = 1000000L;
```

13.4 Constant Expressions

You can build constants from other constants:

```
const double RADIUS = 2.0;
const double AREA = 3.14159 * RADIUS * RADIUS;
```

As long as all operands are constants, the result is a constant too.

13.5 The `#define` Directive

Before `const`, C programmers used macros to define constants:

```
#define PI 3.14159
#define DAYS 7
```

Macros don't reserve memory, they're replaced by the preprocessor before compilation. However, `const` is safer and preferred in modern C because it respects type checking.

Use `#define` for compile-time flags, not numeric constants, unless necessary.

Tiny Code

Try this:

```
#include <stdio.h>

#define TAX_RATE 0.08  // macro constant

int main(void) {
    const double PI = 3.14159;  // const variable

    double radius = 2.0;
    double area = PI * radius * radius;

    printf("Area: %.2f\n", area);
    printf("Tax rate: %.2f\n", TAX_RATE);

    return 0;
}
```

Output:

```
Area: 12.57
Tax rate: 0.08
```

Why It Matters

Constants make your code self-documenting. A literal like `7` means little by itself, but `DAYS_IN_WEEK` tells a clear story. They protect important values and make updates easy, one edit, everywhere fixed.

Good C programs are full of meaningful names, not magic numbers.

Try It Yourself

1. Define constants for `PI`, `E`, and `G` (gravity). Print them.
2. Replace a repeated number (like `60`) with a named constant.
3. Create a macro `#define MAX_SCORE 100` and print it.
4. Try changing a `const` variable after declaration, read the compiler error.
5. Mix literals: assign `3.14F`, `42U`, and `100L` to variables and print them.

Constants express intention and stability. They remind both you and the compiler that some values are meant to stay the same.

14. Operators and Expressions

An expression is a combination of values, variables, and operators that produces a result. Operators are the symbols that tell C what to do, add, subtract, compare, assign, and more. Together, they form the grammar of computation in your programs.

Understanding operators is key to writing clear, correct, and efficient code.

14.1 What Is an Expression

An expression can be as simple as a single value:

```
5
```

Or as complex as:

```
(a + b) * (c - d / e)
```

Every expression has a type and a value. When the compiler evaluates it, it computes and returns that value.

14.2 Basic Categories of Operators

C groups operators by purpose:

Category	Examples	Description
Arithmetic	`+ - * / %`	math operations
Assignment	`= += -= *= /=`	store results
Comparison	`== != > < >= <=`	compare values
Logical	`&&` `|` `| !`	combine conditions
Bitwise	`&` `| ^ ~ << >>`	manipulate bits
Increment / Decrement	`++ --`	increase or decrease
Miscellaneous	`sizeof`, `,`, `?:`	special operators

You'll use arithmetic and comparison most often early on.

14.3 Arithmetic Operators

Operator	Meaning	Example	Result
`+`	addition	`5 + 2`	`7`
`-`	subtraction	`5 - 2`	`3`
`*`	multiplication	`5 * 2`	`10`
`/`	division	`5 / 2`	`2` (integer division)
`%`	remainder	`5 % 2`	`1`

If both operands are integers, the result is integer. For real division, cast one operand to `double`:

```
double r = (double)5 / 2; // 2.5
```

14.4 Assignment Operators

The `=` operator stores a value in a variable:

```c
int x = 5;
```

You can combine operations with assignment:

Operator	Example	Meaning
`+=`	`x += 2`	`x = x + 2`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 2`	`x = x * 2`
`/=`	`x /= 5`	`x = x / 5`
`%=`	`x %= 3`	`x = x % 3`

These make updates concise and expressive.

14.5 Comparison Operators

Used to compare values in conditions. Each returns 1 (true) or 0 (false):

Operator	Meaning	Example
`==`	equal to	`x == y`
`!=`	not equal	`x != y`
`>`	greater than	`x > y`
`<`	less than	`x < y`
`>=`	greater or equal	`x >= y`
`<=`	less or equal	`x <= y`

Example:

```c
if (age >= 18) {
    printf("Adult\n");
}
```

14.6 Logical Operators

Combine or invert boolean results:

Operator	Meaning	Example
`&&`	AND	`(x > 0 && y > 0)`

Operator	Meaning	Example
`||`	OR	`(x == 0 || y == 0)`
`!`	NOT	`!(x > 5)`

Used in `if`, `while`, and other conditional statements.

14.7 Increment and Decrement

Quickly increase or decrease a variable by 1:

```
x++; // same as x = x + 1
x--; // same as x = x - 1
```

Two forms:

- Postfix (`x++`), use value, then increment
- Prefix (`++x`), increment, then use value

Example:

```
int x = 5;
printf("%d\n", x++); // prints 5, then x becomes 6
printf("%d\n", ++x); // increments first, prints 7
```

14.8 Combining Operators

You can mix operators in expressions:

```
int result = (a + b) * c - d / e;
```

Use parentheses to control order of evaluation, they make your intent explicit.

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int a = 5, b = 2;

    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a %% b = %d\n", a % b);

    a += 3;
    printf("a after += 3: %d\n", a);

    printf("Is a > b? %d\n", a > b);

    return 0;
}
```

Output:

```
a + b = 7
a - b = 3
a * b = 10
a / b = 2
a % b = 1
a after += 3: 8
Is a > b? 1
```

Why It Matters

Operators are the verbs of programming. They let you compute, compare, and combine values. By mastering them, you gain expressive power to build logic, perform math, and make decisions.

Every algorithm you write is built from these small, precise actions.

Try It Yourself

1. Create two integers and test all arithmetic operators.
2. Use compound assignments (`+=`, `*=`) to update values.

3. Compare two numbers and print the results of all six comparisons.
4. Combine conditions using `&&` and `||`.
5. Experiment with `x++` vs `++x` in `printf` to see the difference.

Operators are the language of thought in C. Once fluent, you can translate any idea into computation.

15. Arithmetic Operators

Arithmetic operators are the foundation of numerical computation in C. They let you perform mathematical operations on numbers, adding, subtracting, multiplying, dividing, and finding remainders. Even complex algorithms start with these simple actions.

In this section, you'll see how each operator works, how C handles division, and where small details matter.

15.1 The Basic Operators

Operator	Meaning	Example	Result
`+`	Addition	`5 + 2`	`7`
`-`	Subtraction	`5 - 2`	`3`
`*`	Multiplication	`5 * 2`	`10`
`/`	Division	`5 / 2`	`2` (integer division)
`%`	Modulus (remainder)	`5 % 2`	`1`

These five operators work with both constants and variables:

```c
int a = 10, b = 3;
int sum = a + b;       // 13
int diff = a - b;      // 7
int prod = a * b;      // 30
int quot = a / b;      // 3
int rem = a % b;       // 1
```

15.2 Integer Division

When both operands are integers, division truncates toward zero:

```c
printf("%d\n", 5 / 2);  // prints 2
printf("%d\n", 7 / 3);  // prints 2
printf("%d\n", -7 / 3); // prints -2
```

If you want fractional results, convert at least one operand to `double`:

```c
printf("%.2f\n", (double)5 / 2); // 2.50
```

15.3 The Modulus Operator

`%` gives the remainder after integer division:

```c
printf("%d\n", 10 % 3); // 1
printf("%d\n", 9 % 3);  // 0
```

It's often used in:

- Checking even/odd numbers (`x % 2 == 0`)
- Wrapping counters (`i = (i + 1) % 10`)
- Detecting multiples (`n % 5 == 0`)

It works only with integers, not floats.

15.4 Order of Operations

C follows standard operator precedence:

1. Parentheses `( )`
2. Multiplication, Division, Modulus (`* / %`)
3. Addition, Subtraction (`+ -`)

Example:

```c
int r = 2 + 3 * 4; // 2 + 12 = 14
```

Use parentheses to make your intent clear:

```c
int r = (2 + 3) * 4; // 20
```

Parentheses improve both correctness and readability.

15.5 Unary Plus and Minus

You can use + or - before a number or variable:

```
int a = 5;
int b = -a;  // -5
int c = +a;  // 5
```

This doesn't add or subtract, it simply indicates the sign.

15.6 Overflow and Underflow

C doesn't check for arithmetic overflow. If a result is too large for the type, it wraps around:

```
int big = 2147483647; // max 32-bit int
printf("%d\n", big + 1); // overflow -> -2147483648
```

Use larger types (`long long`) or libraries for critical arithmetic.

15.7 Mixing Types

When operands differ, C promotes smaller types automatically:

```
int x = 5;
double y = 2.0;
double z = x / y; // x promoted to double
```

Be aware of implicit conversions to avoid surprises.

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int a = 7, b = 3;

    printf("a + b = %d\n", a + b);
```

```c
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a %% b = %d\n", a % b);

    double precise = (double)a / b;
    printf("Precise division: %.2f\n", precise);

    return 0;
}
```

Output:

```
a + b = 10
a - b = 4
a * b = 21
a / b = 2
a % b = 1
Precise division: 2.33
```

Why It Matters

Arithmetic is the heartbeat of every program. From counting iterations to computing results, these five operators appear everywhere. Understanding how C performs math, especially integer division and precedence, prevents subtle bugs.

In C, precision is power.

Try It Yourself

1. Compute the sum, difference, product, and remainder of any two numbers.
2. Use integer division and floating-point division side by side.
3. Write an expression using parentheses to change order of evaluation.
4. Test `x % 2` for even or odd numbers.
5. Add 1 to `2147483647` (max `int`) and see what happens.

Arithmetic operators are small symbols with big consequences. Once you master them, you can build logic that measures, counts, and calculates everything your programs need.

16. Comparison and Logical Operators

Programs often need to make decisions, to choose one path if something is true and another if it isn't. Comparison and logical operators let you express those decisions clearly. They evaluate conditions and return either true (1) or false (0).

Together, they're the foundation of all conditional logic in C.

16.1 Comparison Operators

Comparison (or relational) operators compare two values. They don't change the values, they just check relationships.

Operator	Meaning	Example	Result
`==`	equal to	`5 == 5`	`1`
`!=`	not equal to	`5 != 3`	`1`
`>`	greater than	`5 > 3`	`1`
`<`	less than	`3 < 5`	`1`
`>=`	greater or equal	`5 >= 5`	`1`
`<=`	less or equal	`3 <= 4`	`1`

Example:

```c
int age = 18;
printf("%d\n", age >= 18); // prints 1 (true)
```

These expressions return integers, `1` for true, `0` for false.

16.2 Common Pitfall: = vs ==

A single `=` assigns a value, while `==` compares values.

```c
if (x = 5) { ... } //  assigns 5 to x, always true
if (x == 5) { ... } //  compares x to 5
```

Always double-check equality conditions.

16.3 Logical Operators

Logical operators combine multiple conditions:

Operator	Meaning	Example	True When
`&&`	AND	`x > 0 && y > 0`	both true
`||`	OR	`x > 0 || y > 0`	either true
`!`	NOT	`!(x > 0)`	condition false

Examples:

```
if (x > 0 && y > 0) {
    printf("Both positive\n");
}

if (x == 0 || y == 0) {
    printf("At least one zero\n");
}

if (!(x > 10)) {
    printf("x is not greater than 10\n");
}
```

16.4 Short-Circuit Evaluation

C uses short-circuit logic:

- In `&&`, if the first condition is false, the second is not checked.
- In `||`, if the first condition is true, the second is not checked.

This saves time and prevents unnecessary work.

Example:

```
if (ptr != NULL && *ptr == 10) { ... }
```

Here, `*ptr` is only evaluated if `ptr` is not null, avoiding crashes.

16.5 Combining Conditions

You can group conditions with parentheses:

```
if ((x > 0 && y > 0) || z > 0) {
    printf("At least one positive value\n");
}
```

Parentheses make intent clear and control evaluation order.

16.6 Boolean Results in C

C doesn't have a separate boolean type before C99. Conditions return integers:

- 1 means true
- 0 means false

You can include `<stdbool.h>` for `bool`, `true`, and `false`:

```
#include <stdbool.h>
bool valid = true;
if (valid) { printf("OK\n"); }
```

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int x = 5, y = 10;

    printf("x == y: %d\n", x == y);
    printf("x != y: %d\n", x != y);
    printf("x < y: %d\n", x < y);
    printf("x > y: %d\n", x > y);
    printf("(x < y) && (y < 20): %d\n", (x < y) && (y < 20));
    printf("(x > 0) || (y < 0): %d\n", (x > 0) || (y < 0));
    printf("!(x == 5): %d\n", !(x == 5));

    return 0;
}
```

Output:

```
x == y: 0
x != y: 1
x < y: 1
x > y: 0
(x < y) && (y < 20): 1
(x > 0) || (y < 0): 1
!(x == 5): 0
```

Why It Matters

Comparison and logical operators turn data into decisions. They're how programs react, testing conditions, guiding flow, and enabling reasoning. From simple checks to complex algorithms, these operators power every `if`, `while`, and `for`.

Without them, programs couldn't think.

Try It Yourself

1. Write conditions to test if a number is positive, negative, or zero.
2. Combine conditions: check if a value is between 10 and 20.
3. Practice `&&`, `||`, and `!` in one expression.
4. Use short-circuit logic with a null pointer check.
5. Print results of comparisons directly with `%d`.

Logic gives your programs intelligence. With comparisons and conditions, C starts to *decide* instead of just *compute*.

17. Operator Precedence

When an expression contains multiple operators, C must decide which operation to perform first. This order is called operator precedence. It's like arithmetic: multiplication happens before addition unless you use parentheses.

Understanding precedence and associativity ensures your code does what you mean, not just what you write.

17.1 Why Precedence Matters

Consider:

```
int x = 2 + 3 * 4;
```

Is `x` equal to 20 or 14? C follows mathematical rules, multiplication happens first, so `x = 14`.

If you want addition first:

```
int x = (2 + 3) * 4; // x = 20
```

Parentheses always override precedence. Use them to make your intent clear.

17.2 Precedence Table

Here's a simplified table of operator precedence (from highest to lowest):

Precedence	Operator	Description	Associativity
1	`()`	Parentheses	Left to right
2	`++ --`	Increment, Decrement	Right to left
3	`* / %`	Multiplication, Division, Modulus	Left to right
4	`+ -`	Addition, Subtraction	Left to right
5	`< <= > >=`	Comparisons	Left to right
6	`== !=`	Equality	Left to right
7	`&&`	Logical AND	Left to right
8	`||`	Logical OR	Left to right
9	`= += -= *= /=`	Assignment	Right to left

Operators higher in the table are evaluated first.

17.3 Associativity

When two operators have the same precedence, associativity decides the direction of evaluation.

Example (left to right):

```
int result = 10 / 2 * 3; // (10 / 2) * 3 = 15
```

Example (right to left):

```
int x = y = z = 1; // same as x = (y = (z = 1))
```

17.4 Combining Operators

Without parentheses, mixed operators can surprise you:

```
int a = 2, b = 3, c = 4;
int result = a + b * c; // 2 + (3 * 4) = 14
```

Parentheses clarify meaning:

```
int result = (a + b) * c; // (2 + 3) * 4 = 20
```

Always use parentheses for clarity, even if you know the precedence.

17.5 Unary vs Binary Operators

Unary operators (`-x`, `++x`, `--x`) have higher precedence than binary ones:

```
int x = 5;
int y = -x * 2; // same as (-5) * 2 = -10
```

Postfix `x++` is evaluated after the value is used, while prefix `++x` increments before use.

Example:

```
int a = 5;
printf("%d\n", a++ * 2); // prints 10, then a = 6
```

17.6 Logical Expressions

When combining conditions, remember:

- `&&` has higher precedence than `||`
- Always use parentheses for clarity

Example:

```
if (a > 0 && b > 0 || c > 0) { ... }
// same as: if ((a > 0 && b > 0) || c > 0)
```

If you wanted `a > 0 && (b > 0 || c > 0)`, add parentheses.

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int a = 2, b = 3, c = 4;

    printf("a + b * c = %d\n", a + b * c);
    printf("(a + b) * c = %d\n", (a + b) * c);
    printf("a + b > c = %d\n", a + b > c);
    printf("a + (b > c) = %d\n", a + (b > c));

    return 0;
}
```

Output:

```
a + b * c = 14
(a + b) * c = 20
a + b > c = 1
a + (b > c) = 2
```

Why It Matters

Operator precedence is one of C's most common sources of subtle bugs. The compiler always follows the rules, even when you didn't intend them. Parentheses make code predictable, readable, and error-free.

When in doubt, add them.

Try It Yourself

1. Write `2 + 3 * 4` and `(2 + 3) * 4`. Compare results.
2. Evaluate `10 / 2 * 3`, why isn't it `10 / (2 * 3)`?
3. Experiment with `++a * 2` and `a++ * 2`.
4. Combine `&&` and `||` in one `if` statement and observe.
5. Rewrite a complex expression using parentheses for clarity.

Precedence defines *how* C thinks. Parentheses define *what you mean*. Always make your intent explicit.

18. Reading Input with `scanf`

So far, your programs have only printed information. Now it's time to make them interactive, to let users type values that your program can read and use. In C, the standard way to read input is with the function `scanf`, part of `<stdio.h>`.

Understanding how `scanf` works gives your programs a voice that listens.

18.1 What Is `scanf`

`scanf` reads formatted input from standard input (usually the keyboard). Its name comes from *scan formatted*.

Basic usage:

```c
scanf("format", &variable);
```

- The first argument is a format string, like `"%d"` or `"%f"`.
- The second is the address of the variable where the result will be stored.

The ampersand (`&`) gives `scanf` a pointer to the variable.

18.2 Reading Different Types

Use format specifiers matching the type of data:

Type	Format	Example
`int`	`%d`	`scanf("%d", &x);`
`double`	`%lf`	`scanf("%lf", &d);`
`float`	`%f`	`scanf("%f", &f);`
`char`	`%c`	`scanf("%c", &ch);`
`string` (char array)	`%s`	`scanf("%s", name);` (no `&`)

Example:

```
int age;
double height;
scanf("%d %lf", &age, &height);
```

This reads two values separated by space.

18.3 The `&` Operator

`scanf` needs the address of each variable so it can store input there.

```
int x;
scanf("%d", &x);
```

Without `&`, `scanf` wouldn't know where to put the data, the program may crash.

Exception: strings already represent addresses, so no `&` is used:

```
char name[20];
scanf("%s", name);
```

18.4 Handling Multiple Inputs

You can read several values at once:

```c
int a, b;
printf("Enter two numbers: ");
scanf("%d %d", &a, &b);
printf("Sum = %d\n", a + b);
```

When the user types `3 5` and presses Enter, both values are read.

18.5 Dealing with Whitespace

`scanf` skips whitespace (spaces, tabs, newlines) before numbers and strings, but not before `%c`. To skip whitespace when reading a character, add a space before `%c`:

```c
scanf(" %c", &ch);
```

That leading space tells `scanf` to ignore any leftover newlines.

18.6 Input Validation

Always check if `scanf` succeeded:

```c
int x;
if (scanf("%d", &x) == 1) {
    printf("You entered %d\n", x);
} else {
    printf("Invalid input!\n");
}
```

`scanf` returns the number of values successfully read.

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int age;
    double height;
    char initial;
```

```
    printf("Enter your age, height, and initial: ");
    scanf("%d %lf %c", &age, &height, &initial);

    printf("Age: %d\n", age);
    printf("Height: %.2f\n", height);
    printf("Initial: %c\n", initial);

    return 0;
}
```

Input:

```
25 1.75 J
```

Output:

```
Age: 25
Height: 1.75
Initial: J
```

Why It Matters

Input is what turns static programs into interactive tools. With `scanf`, you can read numbers, words, and characters, building programs that respond to users. Every calculator, menu, or configuration system starts with reading data.

Learning to use `scanf` safely is a rite of passage in C.

Try It Yourself

1. Write a program that asks for two integers and prints their sum.
2. Read a `char` after reading a number, test the `" %c"` fix.
3. Ask for name and age, then print "Hello, NAME, you are AGE."
4. Try entering the wrong type (e.g., letters for `%d`) and see what happens.
5. Check `scanf`'s return value to handle invalid input gracefully.

When programs can read, they can think. `scanf` is your first step toward true interaction.

19. The `sizeof` Operator

In C, every variable and type occupies a specific amount of memory. The `sizeof` operator lets you ask exactly how much, in bytes. Knowing the size of data is crucial for memory management, portability, and understanding how your program interacts with hardware.

With `sizeof`, you can measure the building blocks of your program.

19.1 What Is `sizeof`

`sizeof` is a compile-time operator that returns the size (in bytes) of a type or variable. You can use it in two forms:

```
sizeof(type)
sizeof expression
```

Examples:

```
sizeof(int)        // size of the int type
sizeof x           // size of the variable x
```

It returns a value of type `size_t` (an unsigned integer).

19.2 Sizes of Basic Types

Sizes can vary by platform, but typical 64-bit systems follow:

Type	Example	Typical Size (bytes)
`char`	`'A'`	1
`int`	`42`	4
`short`	`10`	2
`long`	`1000L`	8
`float`	`3.14f`	4
`double`	`3.14`	8
`long double`	,	16

Check your system by printing them yourself.

19.3 Using `sizeof` with Variables

You can pass a variable instead of a type:

```
int x = 10;
printf("x is %zu bytes\n", sizeof x);
```

The `%zu` format specifier is for `size_t`.

This works even if you change the type of `x`.

19.4 Using `sizeof` with Arrays

For arrays, `sizeof` returns the total size in bytes, not the number of elements.

Example:

```
int arr[5];
printf("%zu\n", sizeof arr); // 5 * sizeof(int)
```

To find the element count:

```
int count = sizeof arr / sizeof arr[0];
printf("Number of elements: %d\n", count);
```

This trick is common in C programming.

19.5 Parentheses Rules

For types, you must use parentheses:

```
sizeof(int)  //
```

For variables, parentheses are optional:

```
sizeof x     //
sizeof(x)    //
```

But many developers always include them for consistency.

19.6 Portable Programming

Different machines may have different type sizes. Instead of assuming, measure:

```
printf("int: %zu bytes\n", sizeof(int));
printf("double: %zu bytes\n", sizeof(double));
```

`sizeof` helps you write portable programs that adapt to any system.

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int i;
    double d;
    char c;
    int arr[10];

    printf("Size of int: %zu bytes\n", sizeof(int));
    printf("Size of double: %zu bytes\n", sizeof d);
    printf("Size of char: %zu bytes\n", sizeof c);
    printf("Size of arr: %zu bytes\n", sizeof arr);
    printf("Number of elements in arr: %zu\n", sizeof arr / sizeof arr[0]);

    return 0;
}
```

Sample Output:

```
Size of int: 4 bytes
Size of double: 8 bytes
Size of char: 1 bytes
Size of arr: 40 bytes
Number of elements in arr: 10
```

Why It Matters

`sizeof` connects you to the physical reality of your program, how much memory it uses. It's essential when allocating memory dynamically, designing data structures, or writing portable code. Knowing your sizes makes you a precise and careful programmer.

C gives you control; `sizeof` tells you what you're controlling.

Try It Yourself

1. Print the size of every basic type on your machine.
2. Create an array of 20 doubles and calculate its element count.
3. Compare `sizeof(5)` and `sizeof(5.0)`. What's different?
4. Create a `struct` with 3 fields and measure its size.
5. Use `sizeof` with a pointer, what does it return?

Every byte matters in C. With `sizeof`, you can see exactly how your code shapes memory.

20. Debugging Type Errors

Every programmer makes mistakes, especially when learning C's strict type system. Type errors happen when you use a value in a way that doesn't match its declared type. Fortunately, the compiler catches most of them for you. Learning to read and fix type errors is a vital skill for writing correct programs.

20.1 What Is a Type Error

A type error occurs when an operation or function receives the wrong kind of data.

Examples:

```
int x = 10;
double y = 2.5;
printf("%d\n", y); //   using %d for a double
```

Here, `%d` expects an `int`, but `y` is a `double`. The output will be incorrect, and the compiler should warn you.

Type errors don't always stop your program, sometimes they cause undefined behavior, where results are unpredictable.

20.2 Common Type Mistakes

Here are typical beginner errors and how to fix them:

1. Mismatched format specifier

```c
double pi = 3.14;
printf("%d\n", pi); //  wrong
printf("%f\n", pi); //  correct
```

2. Using incompatible operands

```c
int a = 5;
char b = 'A';
int sum = a + b; //  allowed (char promoted to int)
int wrong = "Hello" + a; //  invalid types
```

3. Assigning one type to another without casting

```c
int n = 5;
double x = n / 2; //  integer division, result is 2.0
double y = (double)n / 2; //  cast for precision
```

4. Pointer type mismatch

```c
int *p;
double d;
p = &d; //  pointer types incompatible
```

5. Calling functions with wrong argument types

```c
printf("%d\n", 3.14); //  expects int
```

20.3 Reading Compiler Errors

Compilers are your friends. When you see a message like:

```
warning: format ‘%d' expects argument of type ‘int', but argument 2 has type ‘double'
```

Break it down:

- `%d` means you told the compiler to print an `int`
- The second argument was a `double`
- You need `%f` instead

Always read errors carefully, they tell you exactly what went wrong.

20.4 Using Compiler Warnings

Turn on all warnings:

```
gcc -Wall -Wextra -pedantic program.c -o program
```

Warnings aren't fatal, but they highlight risky code. Treat them as errors to fix, not messages to ignore.

20.5 Implicit Type Conversions

Sometimes C converts types for you, but not always safely:

```
int x = 5;
double y = 2.5;
printf("%f\n", x + y); //  x promoted to double
```

But mixing types can cause precision loss:

```
double result = 5 / 2; //  both int → result = 2.0
```

Fix with explicit casting:

```
double result = (double)5 / 2; //  result = 2.5
```

20.6 Warnings vs Errors

- Warnings: the program can compile, but something looks suspicious
- Errors: the compiler cannot produce an executable

Even one warning is a sign of possible bugs, clean them all.

Tiny Code

Try this buggy code:

```c
#include <stdio.h>

int main(void) {
    int a = 5;
    double b = 2.5;

    printf("%d + %f = %f\n", a, b, a + b); //   wrong format specifiers
    return 0;
}
```

Compile with warnings:

```
gcc -Wall bug.c -o bug
```

Fix it:

```
printf("%d + %.1f = %.1f\n", a, b, a + b); //
```

20.7 Strategies for Debugging Type Issues

1. Check format specifiers for `printf` and `scanf`
2. Verify variable types, does the type match its usage?
3. Add casts for mixed-type arithmetic
4. Enable warnings and fix all of them
5. Simplify expressions when errors get confusing

A quick `printf` test can help you inspect variable types and values.

Why It Matters

C gives you power and precision, but it expects discipline. Type mismatches can lead to silent errors or crashes. By paying attention to types, you make your programs robust and portable.

Good C programmers write code the compiler never complains about.

Try It Yourself

1. Print a `double` with `%d` and fix the warning.
2. Assign `5 / 2` to a double, then fix it with casting.
3. Read input into a `char` using `%d`, observe the warning.
4. Mix an `int` and a string (`"Hello" + 2`), read the error message.
5. Turn on `-Wall` and clean all warnings from your last program.

C's type system isn't an obstacle, it's a guide. Follow its rules, and your programs will be safer, faster, and more reliable.

Chapter 3. Control Flow

21. The `if` Statement

Every useful program needs to make choices. Sometimes you want to do something only if a condition is true, like printing a message, checking a score, or skipping an action. That's exactly what the `if` statement does. It's how your program starts to *think* for itself.

21.1 What Is an `if` Statement

The `if` statement lets you run a block of code only when a condition is true.

Basic form:

```
if (condition) {
    // code runs only if condition is true
}
```

If the condition is true (non-zero), the code inside the braces `{}` executes. If it's false (zero), the code is skipped.

21.2 A Simple Example

```
#include <stdio.h>

int main(void) {
    int score = 85;
```

```
    if (score >= 60) {
        printf("You passed!\n");
    }

    printf("Program finished.\n");
    return 0;
}
```

Output:

```
You passed!
Program finished.
```

If you change `score` to `40`, only `Program finished.` prints, the condition failed, so the `if` block is skipped.

21.3 Conditions Are Just Expressions

Any expression can be used as a condition. C treats:

- `0` as false
- any non-zero value as true

```
if (1) { printf("Always runs!\n"); }
if (0) { printf("Never runs!\n"); }
```

You can also use variables:

```
int flag = 5;
if (flag) { printf("True!\n"); } // nonzero → true
```

21.4 Without Braces

If your `if` controls only one statement, braces are optional:

```
if (score >= 60)
    printf("Passed!\n");
```

But it's safer to always use braces, especially as programs grow:

```c
if (score >= 60) {
    printf("Passed!\n");
}
```

This avoids mistakes when adding new lines later.

21.5 Common Pitfall: = vs ==

Remember:

- = assigns a value
- == compares values

So this is wrong:

```c
if (x = 5) { ... } // assigns 5 to x, always true
```

It should be:

```c
if (x == 5) { ... } // compares x to 5
```

A common beginner bug, always double-check your equality signs.

21.6 Nesting `if` Statements

You can place one `if` inside another:

```c
if (score >= 60) {
    if (score >= 90) {
        printf("Excellent!\n");
    }
    printf("Passed!\n");
}
```

Here, a score of 95 prints both messages, while 75 prints only "Passed!".

Tiny Code

Try this program:

```c
#include <stdio.h>

int main(void) {
    int age;

    printf("Enter your age: ");
    scanf("%d", &age);

    if (age >= 18) {
        printf("You can vote.\n");
    }

    if (age >= 60) {
        printf("You get a senior discount!\n");
    }

    printf("Goodbye!\n");
    return 0;
}
```

If you input `65`, you'll see both messages. If you input `17`, you'll see only "Goodbye!".

Why It Matters

The `if` statement is your program's first decision-maker. It lets you control flow, doing different things based on data. Without it, your program would be a straight line. With it, your program starts responding, branching, and reasoning.

Try It Yourself

1. Write an `if` that checks if a number is even (`x % 2 == 0`).
2. Ask the user for a grade, and print "Passed" if it's at least 50.
3. Check if a value is negative, zero, or positive using nested `if`s.
4. Use a variable as a condition (`if (flag)`) and test different values.
5. Experiment with `if (x = 5)`, see why it always runs.

With `if`, your code stops being a script and starts becoming interactive logic, reacting to data and making choices just like you do.

22. The `else` and `else if` Clauses

Life isn't just *yes* or *no*, sometimes there's a second path to take when a condition fails. That's what `else` is for. It lets your program say, "If not this, then do that." And when you have multiple possibilities, `else if` helps you chain them together.

With these, your program can handle all outcomes, not just one.

22.1 Adding an `else`

An `if` checks a condition, an `else` catches what happens when that condition isn't true.

```
if (condition) {
    // do this if true
} else {
    // do this if false
}
```

Example:

```
int score = 45;

if (score >= 60) {
    printf("You passed!\n");
} else {
    printf("You failed.\n");
}
```

If `score` is 75, it prints "You passed!". If `score` is 45, it prints "You failed."

One condition, two paths, true or false.

22.2 Using `else if` for Multiple Choices

What if you have more than two outcomes? That's where `else if` comes in:

```
if (score >= 90) {
    printf("Grade: A\n");
} else if (score >= 80) {
    printf("Grade: B\n");
} else if (score >= 70) {
```

```
    printf("Grade: C\n");
} else {
    printf("Grade: F\n");
}
```

C checks these in order, top to bottom:

- If one condition is true, that block runs and the rest are skipped.
- If none match, the final `else` runs.

This creates a decision ladder.

22.3 How It Works

Let's say `score = 85`:

1. `score >= 90`? No. Skip.
2. `score >= 80`? Yes. Run "Grade: B."
3. Stop, no need to check further.

Only one branch runs per chain.

22.4 Nesting vs Chaining

You could write nested `if`s:

```
if (score >= 60) {
    if (score >= 90) {
        printf("Excellent!\n");
    } else {
        printf("Good job!\n");
    }
} else {
    printf("Try again.\n");
}
```

But `else if` chains are easier to read and maintain. Use them when conditions are mutually exclusive.

22.5 Common Mistakes

1. Dangling `else`, every `else` matches the nearest unmatched `if`. Always use braces `{}` to avoid confusion:

```
if (x > 0)
    if (x < 10)
        printf("Small\n");
    else
        printf("Big\n"); // attaches to inner if
```

2. Missing braces, causes unexpected grouping. Always write:

```
if (x > 0) {
    printf("Positive\n");
} else {
    printf("Non-positive\n");
}
```

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int temp;

    printf("Enter temperature: ");
    scanf("%d", &temp);

    if (temp >= 30) {
        printf("It's hot!\n");
    } else if (temp >= 20) {
        printf("It's warm.\n");
    } else if (temp >= 10) {
        printf("It's cool.\n");
    } else {
        printf("It's cold!\n");
    }

    return 0;
}
```

Input → 25 Output → `It's warm.`

22.6 The Order Matters

Always start with the most specific or highest condition first. Otherwise, broader conditions may trigger early and skip later checks.

Example:

```
if (score >= 60)      // triggers first
else if (score >= 90) // never reached
```

Reverse the order to handle top cases first.

Why It Matters

`if`, `else if`, and `else` form the decision tree of your program. They allow branching logic, responding differently to every situation. With them, you can model real-world thinking: "If this is true, do that; otherwise, try this; if all else fails, do something else."

Try It Yourself

1. Ask for a number and print if it's positive, negative, or zero.
2. Ask for an age and print if someone is a child, teen, adult, or senior.
3. Grade a test score from 0–100 using `if-else if-else`.
4. Change the order of checks, see how it changes results.
5. Add an `else` to catch invalid input (like negative scores).

With `else` and `else if`, your programs start thinking in complete sentences — handling not just "yes" but "otherwise" and "maybe."

23. Nested Conditionals

Sometimes, one decision isn't enough. You might want to make a choice inside another choice, for example, first check if a user passed, then check *how well* they passed. That's where nested conditionals come in.

A nested `if` is simply an `if` statement inside another `if` block. It's how you build step-by-step decisions, just like real thinking.

23.1 What Are Nested `if`s

You can place any `if` or `else` inside another `if`:

```
if (condition1) {
    if (condition2) {
        // runs only if both condition1 and condition2 are true
    }
}
```

Each layer adds one more level of decision-making.

23.2 A Simple Example

```
int score = 95;

if (score >= 60) {
    printf("You passed!\n");

    if (score >= 90) {
        printf("Excellent!\n");
    }
}
```

- The first `if` checks if you passed.
- The second `if` checks if your score is also excellent.

Output:

```
You passed!
Excellent!
```

If `score = 70`, you only get `You passed!`. If `score = 40`, neither condition runs.

23.3 Real-World Example

Imagine a login system:

```c
int logged_in = 1;
int admin = 0;

if (logged_in) {
    printf("Welcome!\n");

    if (admin) {
        printf("Admin panel unlocked.\n");
    } else {
        printf("Standard user access.\n");
    }
} else {
    printf("Please log in first.\n");
}
```

Depending on your flags, the program prints different paths. This is decision layering, logic inside logic.

23.4 Nested vs Combined Conditions

Nested conditions can sometimes be simplified using logical operators:

```c
if (score >= 60) {
    if (score < 90) {
        printf("Good job!\n");
    }
}
```

This can become:

```c
if (score >= 60 && score < 90) {
    printf("Good job!\n");
}
```

Both do the same thing. When possible, combine conditions for clarity. When logic grows complex, keep nesting, it's okay!

23.5 Indentation Matters

Indentation helps you see structure clearly:

```
if (a > 0) {
    if (b > 0) {
        printf("Both positive\n");
    }
}
```

Without indentation, nested logic becomes confusing. Always align braces and indent inner blocks so they're easy to read.

23.6 Using `else` in Nested Structures

Each `if` can have its own `else`:

```
if (x > 0) {
    if (x % 2 == 0) {
        printf("Positive even number\n");
    } else {
        printf("Positive odd number\n");
    }
} else {
    printf("Not positive\n");
}
```

Input → `4` → `Positive even number` Input → `3` → `Positive odd number` Input → `-5` → `Not positive`

Every branch is now covered.

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int age;
```

```
    printf("Enter your age: ");
    scanf("%d", &age);

    if (age >= 18) {
        printf("You are an adult.\n");

        if (age >= 65) {
            printf("You are also eligible for senior benefits.\n");
        }
    } else {
        printf("You are a minor.\n");
    }

    return 0;
}
```

Output for `70`:

```
You are an adult.
You are also eligible for senior benefits.
```

Why It Matters

Nested conditionals let your programs handle multi-step logic, one question leading to another. They're perfect for menus, game rules, and complex checks. When your program needs to say, "If this, then check that," nesting is the way.

Try It Yourself

1. Write a program that checks if a number is positive. Inside, check if it's even or odd.
2. Ask for a score. If it's passing, check if it's an A, B, or C.
3. Simulate a login: if logged in, greet; if admin, show a special message.
4. Combine two nested conditions into one using `&&`.
5. Add indentation and braces to a messy nested `if`, see how much clearer it looks.

Nested conditionals let your programs think in layers, just like you. Each level adds more nuance, helping your code make smarter decisions.

24. The `switch` Statement

Sometimes you want your program to choose between many options, not just two or three. You could write a long chain of `if`, `else if`, `else`, but that quickly becomes messy. C gives you a cleaner tool for this job: the `switch` statement.

Think of `switch` like a menu. You pick a value, and C jumps straight to the matching option.

24.1 What Is a `switch`

A `switch` compares a single value against a list of constant cases. When it finds a match, it runs the code for that case.

Basic form:

```
switch (expression) {
    case value1:
        // code for value1
        break;
    case value2:
        // code for value2
        break;
    default:
        // code if no case matches
        break;
}
```

- The `expression` is usually an integer or character.
- Each `case` is like a label, if it matches, that block runs.
- The optional `default` runs when nothing else matches.

24.2 A Simple Example

```
#include <stdio.h>

int main(void) {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
```

```
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        default:
            printf("Invalid day\n");
    }

    return 0;
}
```

Output:

```
Wednesday
```

The program jumps straight to `case 3`.

24.3 The Role of `break`

After each `case`, you usually write `break;` to exit the `switch`.

If you forget it, C will fall through, it keeps running the next cases too:

```
int x = 2;

switch (x) {
    case 1:
        printf("One\n");
    case 2:
        printf("Two\n");
    case 3:
        printf("Three\n");
}
```

Output:

```
Two
Three
```

C didn't stop after `case 2`. To prevent this, always include `break;` unless you want fall-through intentionally.

24.4 Using `default`

`default` catches anything that doesn't match:

```
switch (grade) {
    case 'A':
        printf("Excellent\n");
        break;
    case 'B':
        printf("Good\n");
        break;
    default:
        printf("Invalid grade\n");
}
```

It's like the `else` in an `if` chain, always optional but useful.

24.5 Grouping Cases

You can group multiple cases that share the same code:

```
switch (ch) {
    case 'a':
    case 'A':
        printf("Vowel A\n");
        break;
    case 'e':
    case 'E':
        printf("Vowel E\n");
        break;
    default:
        printf("Not A or E\n");
}
```

Grouped cases help handle upper/lowercase or related values easily.

24.6 Switch vs If

Use `switch` when:

- You're checking one variable against fixed values
- You want clear, organized options

Use `if` when:

- You need ranges, conditions, or complex logic

Example:

```
if (score >= 90) ...  // can't do this in switch
```

`switch` is simpler when your values are exact (like menu numbers or keys).

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int option;

    printf("Choose an option (1-3): ");
    scanf("%d", &option);

    switch (option) {
        case 1:
            printf("Start game\n");
            break;
        case 2:
            printf("Load game\n");
            break;
        case 3:
            printf("Quit\n");
            break;
        default:
            printf("Invalid choice\n");
    }
```

```
    return 0;
}
```

Input → 2 Output → `Load game`

24.7 Nested Switches (Optional)

You can nest `switch` statements, but keep them readable:

```
switch (userType) {
    case 1:
        switch (permission) {
            case 0: printf("Guest\n"); break;
            case 1: printf("Member\n"); break;
        }
        break;
}
```

Use indentation and braces to stay organized.

Why It Matters

The `switch` statement helps you handle many fixed options cleanly. Instead of long `if` chains, you get a simple structure, easy to read, easy to extend. It's perfect for menus, commands, and settings.

Think of it as your program's choice board.

Try It Yourself

1. Write a program that prints the day of the week (1–7).
2. Create a grade system using `'A'`, `'B'`, `'C'`, `'D'`, `'F'`.
3. Add a menu with 3 options and a `default` for invalid ones.
4. Test what happens when you remove a `break`.
5. Group two cases together (like `'a'` and `'A'`).

With `switch`, your programs start feeling menu-driven — they don't just think, they *offer choices*.

25. The `while` Loop

In many programs, you'll need to repeat something over and over, like printing numbers, reading input, or waiting for a condition. Instead of copying the same code many times, you can use a loop. The `while` loop is the simplest kind: it repeats a block of code as long as a condition is true.

Think of it like saying, "While it's still raining, keep the umbrella open."

25.1 What Is a `while` Loop

A `while` loop runs again and again until its condition becomes false.

```
while (condition) {
    // code runs while condition is true
}
```

Each time it reaches the end of the block, it checks the condition again. If the condition is still true, it repeats. If false, it exits and moves on.

25.2 A Simple Example

```
#include <stdio.h>

int main(void) {
    int count = 1;

    while (count <= 5) {
        printf("Count: %d\n", count);
        count++; // don't forget to update!
    }

    printf("Done!\n");
    return 0;
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Done!
```

This loop runs 5 times because the condition `count <= 5` starts true, then eventually becomes false.

25.3 The Condition Must Change

If you forget to update the variable inside the loop, it'll never end:

```c
int x = 1;
while (x <= 5) {
    printf("x = %d\n", x);
    // missing x++
}
```

This creates an infinite loop, the condition is always true. Always make sure something inside the loop moves it toward stopping.

25.4 Counting Down

You can loop backward too:

```c
int n = 5;
while (n > 0) {
    printf("%d...\n", n);
    n--;
}
printf("Blast off!\n");
```

Output:

```
5...
4...
3...
2...
1...
Blast off!
```

25.5 Using `while` for User Input

You can loop until a certain value appears:

```c
#include <stdio.h>

int main(void) {
    int number = 0;

    while (number != 42) {
        printf("Enter a number (42 to quit): ");
        scanf("%d", &number);
    }

    printf("You found the answer!\n");
    return 0;
}
```

The loop continues until you enter `42`.

25.6 Infinite Loops (Intentional)

Sometimes infinite loops are useful, especially in programs that should run forever (like servers or games). You can write one with `while (1)`:

```c
while (1) {
    // do something forever
}
```

You'll need a `break` or exit condition inside to stop it safely.

25.7 Nested Loops

You can put one `while` loop inside another:

```c
int i = 1;
while (i <= 3) {
    int j = 1;
    while (j <= 2) {
        printf("i=%d, j=%d\n", i, j);
```

```
        j++;
    }
    i++;
}
```

Output:

```
i=1, j=1
i=1, j=2
i=2, j=1
i=2, j=2
i=3, j=1
i=3, j=2
```

Tiny Code

Try this countdown:

```
#include <stdio.h>

int main(void) {
    int i = 10;

    while (i > 0) {
        printf("%d ", i);
        i--;
    }

    printf("\nLiftoff!\n");
    return 0;
}
```

Output:

```
10 9 8 7 6 5 4 3 2 1
Liftoff!
```

Why It Matters

The `while` loop gives your program repetition with control. You decide the rule, and C keeps repeating until that rule breaks. Loops turn one action into many, saving time, space, and effort.

Once you understand loops, you'll see them everywhere — from reading files to running games.

Try It Yourself

1. Print numbers from 1 to 10 using a `while` loop.
2. Write a program that sums numbers from 1 to 100.
3. Loop until the user enters a negative number.
4. Create a countdown from 5 to 1, then print "Go!".
5. Make a guessing loop that stops when the user guesses `7`.

The `while` loop is your first taste of repetition and automation. Tell it the rule, and it'll handle the rest, patiently and precisely.

26. The `for` Loop

The `for` loop is one of the most common loops in C. It's perfect when you know exactly how many times you want something to repeat, like printing numbers from 1 to 10 or iterating over an array.

You can think of it like a built-in counter that says: "Start here, repeat until this condition, and change this each time."

26.1 What Is a `for` Loop

A `for` loop has three parts, start, condition, and update, all in one line:

```
for (start; condition; update) {
    // code runs while condition is true
}
```

Each time the loop runs:

1. The start sets up your variable (usually a counter).
2. The condition is checked, if true, run the body.
3. After the body, the update runs (like `i++`).

Then it loops again.

26.2 A Simple Example

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 5; i++) {
        printf("i = %d\n", i);
    }
    return 0;
}
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

Let's break it down:

- Start → `int i = 1`
- Condition → `i <= 5`
- Update → `i++` (adds 1 each loop)

When `i` reaches 6, the condition fails, and the loop stops.

26.3 Counting Down

You can count backward too:

```c
for (int n = 5; n > 0; n--) {
    printf("%d...\n", n);
}
printf("Blast off!\n");
```

Output:

```
5...
4...
3...
2...
1...
Blast off!
```

26.4 Skipping Steps

You can increase by more than one:

```c
for (int i = 0; i <= 10; i += 2) {
    printf("%d ", i);
}
```

Output: `0 2 4 6 8 10`

The `update` can do anything, add, subtract, multiply, or even call a function.

26.5 Omitting Parts

Each part of the `for` loop is optional:

```c
int i = 0;
for (; i < 3; ) {
    printf("%d\n", i);
    i++;
}
```

Even this works (infinite loop):

```c
for (;;) {
    printf("Forever!\n");
}
```

Though usually you'll use a `while` for that.

26.6 Nested `for` Loops

You can nest loops to repeat patterns in two dimensions:

```c
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        printf("i=%d, j=%d\n", i, j);
    }
}
```

Output:

```
i=1, j=1
i=1, j=2
i=2, j=1
i=2, j=2
i=3, j=1
i=3, j=2
```

Great for grids, tables, and matrices.

26.7 When to Use `for`

Use `for` when:

- You know how many times to loop
- You're counting or iterating over a sequence
- The variable naturally updates each time

If the number of repetitions is unknown, use a `while` instead.

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int sum = 0;

    for (int i = 1; i <= 10; i++) {
```

```
        sum += i;
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

Output:

```
Sum = 55
```

The loop runs 10 times and keeps adding `i` to `sum`.

Why It Matters

The `for` loop is your structured repeater, short, clean, and powerful. It keeps counting for you, so you can focus on what to do each step. Once you get used to it, you'll use `for` loops all the time, from simple counters to array processing.

Try It Yourself

1. Print numbers from 1 to 10 using a `for` loop.
2. Print only even numbers from 0 to 20.
3. Count down from 10 to 1 and print "Go!".
4. Calculate the sum of numbers from 1 to 100.
5. Create a multiplication table (nested loop).

With `for`, repetition becomes neat and predictable. It's your programmable timer, set it, and let it run.

27. The `do-while` Loop

So far, you've seen loops that check first, then run. The `do-while` loop flips that order, it runs first, then checks. That means it always runs at least once, no matter what.

Think of it like saying, "Do this thing once, and if it's still okay, keep doing it."

27.1 What Is a `do-while` Loop

The `do-while` loop looks like this:

```
do {
    // code to repeat
} while (condition);
```

Notice the semicolon at the end, that's important! The body runs first, then the condition is tested. If the condition is true, the loop repeats.

27.2 A Simple Example

```
#include <stdio.h>

int main(void) {
    int count = 1;

    do {
        printf("Count: %d\n", count);
        count++;
    } while (count <= 5);

    printf("Done!\n");
    return 0;
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Done!
```

Even if `count` started larger than 5, the body would still run once.

27.3 Always Runs at Least Once

Compare this to a regular `while` loop:

```
int n = 10;

while (n < 5) {
    printf("This never prints!\n");
}
```

But with `do-while`:

```
int n = 10;

do {
    printf("This runs once!\n");
} while (n < 5);
```

Even though `n < 5` is false, you'll still see one print. That's the main difference, check later, not before.

27.4 A Common Use: Input Validation

`do-while` is perfect when you want the user to do something at least once, like entering a number until it's valid.

```
#include <stdio.h>

int main(void) {
    int number;

    do {
        printf("Enter a positive number: ");
        scanf("%d", &number);
    } while (number <= 0);

    printf("You entered: %d\n", number);
    return 0;
}
```

The prompt appears at least once, even if the first input is wrong.

27.5 The Condition at the End

Unlike `for` and `while`, the `do-while` condition goes after the block. So don't forget the semicolon:

```
do {
    // body
} while (condition); // ← required
```

Leaving it out causes a compile error.

27.6 Infinite `do-while`

You can make an intentional infinite loop:

```
do {
    // repeat forever
} while (1);
```

Useful when you always want one full run before checking a break.

Tiny Code

Try this:

```
#include <stdio.h>

int main(void) {
    int i = 3;

    do {
        printf("i = %d\n", i);
        i--;
    } while (i > 0);

    printf("Loop finished.\n");
    return 0;
}
```

Output:

```
i = 3
i = 2
i = 1
Loop finished.
```

Why It Matters

The `do-while` loop gives you at least one guaranteed run, which makes it great for tasks like user input, menus, and retries. Whenever you want to "try first, check later," this is your go-to loop.

It's another tool in your looping toolbox, one that always *gets things started*.

Try It Yourself

1. Print "Hello!" three times with a `do-while` loop.
2. Ask for a password until the user types `1234`.
3. Count down from 5 using `do-while`.
4. Make a menu that repeats until the user enters `0`.
5. Compare a `while` and a `do-while` that both check `x < 0`. What happens?

The `do-while` is your one-time starter, it makes sure your code runs at least once before asking, "Should I keep going?"

28. Breaking and Continuing Loops

Sometimes you don't want a loop to run all the way to the end. Maybe you want to stop early when something happens, or skip one step and move to the next.

That's exactly what `break` and `continue` do — they give you extra control inside any loop.

Think of them as "emergency exits" and "shortcuts" for your loop.

28.1 The `break` Statement

`break` stops the loop completely and jumps out right away. The program continues with the first line after the loop.

Basic form:

```c
while (condition) {
    if (something_happened) {
        break; // exit loop
    }
}
```

28.2 Example: Stop When Found

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            printf("Found 5! Stopping.\n");
            break;
        }
        printf("%d ", i);
    }

    printf("\nLoop ended.\n");
    return 0;
}
```

Output:

```
1 2 3 4 Found 5! Stopping.
Loop ended.
```

When `i == 5`, the `break` runs, loop ends instantly. Numbers 6 to 10 are skipped.

28.3 The `continue` Statement

`continue` skips the rest of the loop body and jumps back to the next iteration. The loop doesn't stop, it just moves on early.

Basic form:

```c
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    printf("%d ", i);
}
```

Output:

```
1 2 4 5
```

When `i == 3`, `continue` skips the `printf`. The loop jumps right back to the top.

28.4 Using `break` and `continue` in `while` Loops

They work in any kind of loop, `for`, `while`, or `do-while`.

Example:

```c
int n = 0;
while (n < 10) {
    n++;

    if (n == 3) continue;  // skip 3
    if (n == 8) break;     // stop at 8

    printf("%d ", n);
}
```

Output:

```
1 2 4 5 6 7
```

28.5 Nested Loops and `break`

If you have loops inside loops, `break` only exits the current one.

```c
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) break;
        printf("i=%d j=%d\n", i, j);
    }
}
```

Output:

```
i=1 j=1
i=2 j=1
i=3 j=1
```

Each inner loop stops at `j == 2`, but the outer one keeps going.

28.6 Why Use Them

- `break` is great when you find what you're looking for early.
- `continue` is great when you want to skip certain cases but keep looping.

They make loops more flexible, you don't always have to go all the way.

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int num;

    while (1) {
        printf("Enter a number (0 to quit): ");
        scanf("%d", &num);

        if (num == 0) {
            printf("Goodbye!\n");
            break; // exit the loop
        }

        if (num < 0) {
            printf("Negative skipped.\n");
            continue; // skip to next input
        }

        printf("You entered %d\n", num);
    }
```

```
    return 0;
}
```

Try entering positive, negative, and zero values. You'll see how `break` and `continue` shape the loop's path.

Why It Matters

Loops aren't always simple start-to-finish runs. Real logic often needs early exits or skipped steps. With `break` and `continue`, you get fine-grained control — you decide when to stop or move on.

Try It Yourself

1. Print numbers from 1 to 10, but stop when you reach 7.
2. Print numbers 1 to 10, skipping all even numbers.
3. Read numbers until you get `0`, but skip negatives.
4. In a `for` loop, break early when the sum exceeds 50.
5. Create a nested loop and use `break` in the inner one.

`break` is your exit key, `continue` is your skip button — together they make your loops smarter and more dynamic.

29. Using `goto` Safely (and Why to Avoid It)

There's a little keyword in C called `goto`. It lets you jump directly to another part of your program, like teleporting to a label. It can be powerful, but also confusing if used too much. That's why experienced programmers say:

> "Use it only when you really need to."

Let's see how it works, and when it's better to use something else.

29.1 What Is `goto`

The `goto` statement jumps straight to a label, a line in your program marked with a name and a colon.

Basic form:

```
goto label;
// ...
label:
    // code here
```

When `goto` runs, the program skips everything between the jump and the label. Execution continues at the label.

29.2 A Simple Example

```
#include <stdio.h>

int main(void) {
    int x = 1;

    if (x == 1)
        goto skip;

    printf("This line is skipped!\n");

skip:
    printf("Jumped to label.\n");
    return 0;
}
```

Output:

```
Jumped to label.
```

Because `x == 1`, `goto skip;` jumps straight to the label. The line before it is never executed.

29.3 Common Use: Breaking Out of Nested Loops

Sometimes you want to escape multiple loops at once. A single `break` only exits the current loop, but a `goto` can jump out of all of them.

Example:

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 3; i++) {
        for (int j = 1; j <= 3; j++) {
            if (i * j == 4)
                goto found;
        }
    }

found:
    printf("Stopped when i * j = 4\n");
    return 0;
}
```

Output:

```
Stopped when i * j = 4
```

Once the condition is met, `goto` jumps straight out.

29.4 Why Many Programmers Avoid It

`goto` can make code hard to read and debug. If used often, your program's flow jumps all over the place — it's easy to lose track of what happens next.

Compare:

```c
goto step2;
// ...
step2:
goto step3;
// ...
step3:
printf("Done!\n");
```

vs. a clean loop or function, much easier to follow!

That's why most of the time, loops, functions, or `break` statements are better choices.

29.5 When It's Useful

`goto` is okay for:

- Exiting deeply nested loops
- Handling errors (jump to cleanup)
- Early exits when code is too repetitive

Here's an example from system programming:

```
FILE *file = fopen("data.txt", "r");
if (!file) goto error;

char *buffer = malloc(100);
if (!buffer) goto cleanup;

printf("File opened and buffer allocated.\n");

cleanup:
    if (file) fclose(file);
    if (buffer) free(buffer);
error:
    return 0;
```

Here, `goto` jumps to cleanup code safely, instead of repeating `fclose` and `free` in many places.

29.6 Best Practices

If you use `goto`:

- Only jump forward, not backward (avoid loops with `goto`)
- Keep labels close to the `goto`
- Use clear label names (like `cleanup:` or `exit:`)
- Prefer loops or functions for normal flow

Tiny Code

Try this:

```c
#include <stdio.h>

int main(void) {
    int n;

    while (1) {
        printf("Enter a number (0 to quit): ");
        scanf("%d", &n);

        if (n == 0)
            goto end;

        printf("You entered %d\n", n);
    }

end:
    printf("Goodbye!\n");
    return 0;
}
```

Output:

```
Enter a number (0 to quit): 5
You entered 5
Enter a number (0 to quit): 0
Goodbye!
```

The `goto` jumps straight to `end:` when the user enters `0`.

Why It Matters

`goto` is part of C's toolbox, but it's a sharp tool. You can use it to escape tricky situations, but if you rely on it too much, your program becomes a maze.

Learn it, understand it, but reach for loops, breaks, and functions first.

Try It Yourself

1. Write a loop that jumps to a label when the user types `-1`.
2. Create nested loops and use `goto` to break out of both.

3. Try replacing a `goto` with a `break`, see which looks clearer.
4. Write a small program with a `cleanup:` label to close files.
5. Experiment with labels placed before and after the `goto`.

`goto` is like an emergency exit — nice to know where it is, but you hope you never have to use it!

30. Patterns of Control Flow

Now that you've learned about if statements, loops, break, continue, and even goto, you've got all the tools to control how your program flows, when it chooses, repeats, or stops.

This section is about putting it all together. You'll see the common patterns of control flow that appear in almost every C program. Think of these as the building blocks of logic.

30.1 What Is Control Flow?

Control flow means *the order in which your program's statements run.* By default, C executes from top to bottom, one line at a time.

But with conditionals and loops, you can:

- Branch, choose one path or another (`if`, `else`)
- Repeat, run something again and again (`while`, `for`, `do-while`)
- Jump, move somewhere else (`break`, `continue`, `goto`)

Good programs combine these patterns clearly and simply.

30.2 The Sequence Pattern

The simplest pattern: do things one after another.

```
printf("Start\n");
printf("Step 1\n");
printf("Step 2\n");
printf("Done\n");
```

This is straight-line control flow, no conditions, no loops. Great for simple scripts or steps that always happen in order.

30.3 The Selection Pattern

Use `if`, `else if`, `else`, or `switch` when you need to choose between actions.

Example (using if):

```
if (score >= 90) printf("A\n");
else if (score >= 80) printf("B\n");
else printf("C or lower\n");
```

Example (using switch):

```
switch (menu) {
    case 1: printf("Play\n"); break;
    case 2: printf("Settings\n"); break;
    case 3: printf("Exit\n"); break;
    default: printf("Invalid\n");
}
```

This is branching control flow, only one path runs.

30.4 The Repetition Pattern

Loops let you repeat actions while conditions hold true. You've met three kinds:

- `while`, repeat *while true*
- `for`, repeat a fixed number of times
- `do-while`, run once, then check again

Example:

```
for (int i = 1; i <= 5; i++) {
    printf("%d ", i);
}
```

Output: `1 2 3 4 5`

This is iterative control flow, doing something many times.

30.5 The Nested Pattern

You can combine decisions and loops, one inside another.

```
for (int i = 1; i <= 3; i++) {
    if (i % 2 == 0) {
        printf("%d is even\n", i);
    } else {
        printf("%d is odd\n", i);
    }
}
```

Each loop iteration includes its own decision. This is a nested flow, logic inside logic.

30.6 The Early Exit Pattern

Sometimes you want to leave early when a condition is met. That's where `break`, `continue`, and return come in.

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) break; // stop at 5
    printf("%d ", i);
}
```

Or skip certain cases:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue; // skip 3
    printf("%d ", i);
}
```

These are shortcut flows, stop early, skip ahead, or exit cleanly.

30.7 The Guard Pattern

A guard checks that a condition is safe before running a block. It's like a door that opens only if the key fits.

Example:

```c
if (ptr != NULL) {
    printf("Pointer is valid.\n");
}
```

Or inside a loop:

```c
while (input > 0) {
    // do work
}
```

Guards prevent bad states, they're everywhere in robust programs.

30.8 The Menu Pattern

You'll see this often in interactive programs:

```c
int option;
do {
    printf("1. Add\n2. Subtract\n3. Quit\n");
    scanf("%d", &option);

    switch (option) {
        case 1: printf("Adding...\n"); break;
        case 2: printf("Subtracting...\n"); break;
        case 3: printf("Goodbye!\n"); break;
        default: printf("Invalid choice.\n");
    }
} while (option != 3);
```

This combines loops, switch, and input, a classic pattern!

30.9 Combining Patterns

Most real programs use multiple control flow patterns together:

- A loop that reads data
- An `if` that checks for errors
- A `break` to exit early
- Another loop for retrying

Your job is to combine them clearly, each part should make sense on its own.

Tiny Code

Try this full example:

```c
#include <stdio.h>

int main(void) {
    int n;

    while (1) {
        printf("Enter a number (0 to quit): ");
        scanf("%d", &n);

        if (n == 0) break;

        if (n % 2 == 0)
            printf("%d is even\n", n);
        else
            printf("%d is odd\n", n);
    }

    printf("Goodbye!\n");
    return 0;
}
```

This program:

- Loops until `0`
- Branches with `if`
- Breaks early A perfect example of combined flow.

Why It Matters

Control flow is the heart of programming. It's how you make your program *do the right thing at the right time*. By mixing these simple patterns, you can express almost any logic — decisions, repetitions, early exits, or menus.

You're not just writing code, you're designing behavior.

Try It Yourself

1. Combine `for` and `if` to print all even numbers between 1–20.
2. Write a guessing game loop with a `break` when the user is correct.
3. Make a menu that loops until the user chooses to exit.
4. Add guards (safety checks) before using variables.
5. Mix `if`, `for`, and `continue` in one program.

Now you've seen every major flow pattern in C. They're your logic toolkit, combine them, and your programs can handle anything!

Chapter 4. Functions and Scope

31. Defining and Calling Functions

As your programs grow, you'll notice some code starts repeating, printing menus, adding numbers, checking inputs. Instead of copying and pasting, you can wrap those steps into a function.

A function is like a mini-program inside your program. It takes input, does some work, and can return an answer. Functions make your code clearer, shorter, and easier to fix.

31.1 What Is a Function

A function is a named block of code that performs one task.

It usually has:

- A return type, what it gives back
- A name, how you call it
- Parameters, inputs it uses
- A body, the work it does

Example:

```c
int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Here:

- `int` → the return type (an integer)
- `add` → the name of the function
- `(int a, int b)` → parameters
- `{ int sum = a + b; return sum; }` → the function body

31.2 Calling a Function

Once you define a function, you can call it by name anywhere:

```
int result = add(3, 5);
printf("%d\n", result);  // prints 8
```

The program jumps to the function, runs it, returns a value, then continues.

Think of it as sending a message:

> "Hey `add`, can you sum 3 and 5 for me?" "Sure, here's 8!"

31.3 Anatomy of a Function

A function definition has this form:

```
return_type name(parameters) {
    // body (what to do)
}
```

Examples:

```
void greet() {
    printf("Hello!\n");
}
```

Here, `void` means no return value.

31.4 Functions with No Parameters

Some tasks don't need input. You can leave the parentheses empty:

```c
void say_hi() {
    printf("Hi there!\n");
}

int main(void) {
    say_hi();
    return 0;
}
```

Output:

```
Hi there!
```

31.5 Functions That Return Nothing

If your function just does something (like printing) and doesn't need to give back a result, use `void`:

```c
void print_line() {
    printf("-----------------\n");
}
```

You call it the same way:

```c
print_line();
```

31.6 Return Values

If you want to send a value back, use `return`. It ends the function and hands back the result:

```c
int square(int n) {
    return n * n;
}
```

```c
int main(void) {
    int x = 4;
    printf("%d\n", square(x));  // prints 16
    return 0;
}
```

31.7 Where to Define Functions

You can define functions:

- Before `main()` → so they're known early
- After `main()` → but then you need a prototype (we'll cover soon)

For now, place your functions above `main()` for simplicity.

31.8 Naming Functions

Choose clear names, what they *do*:

- `add_numbers`
- `print_menu`
- `find_max`

Avoid vague ones like `doit()` or `thing()`. Good names make code read like a story.

31.9 Multiple Functions

You can have as many functions as you want. Each one should do one thing well:

```c
void greet_user() {
    printf("Welcome!\n");
}

int sum(int a, int b) {
    return a + b;
}

int main(void) {
    greet_user();
    printf("Sum: %d\n", sum(2, 3));
    return 0;
}
```

Output:

```
Welcome!
Sum: 5
```

Tiny Code

Try this:

```c
#include <stdio.h>

int double_number(int n) {
    return n * 2;
}

int main(void) {
    int value;
    printf("Enter a number: ");
    scanf("%d", &value);

    int result = double_number(value);
    printf("Twice that is %d\n", result);
    return 0;
}
```

Why It Matters

Functions are your first step toward modular programming. They help break big problems into small, reusable parts. When each function does one clear job, your code becomes easier to read, test, and reuse.

Every great program, even huge ones, is built from lots of small, clear functions.

Try It Yourself

1. Write a function that prints "Hello, world!".
2. Write a function `square()` that returns `n * n`.
3. Write a function `sum()` that adds two numbers.
4. Write a function `average()` that takes two floats and returns their mean.
5. Write a function `greet(name)` that prints "Hello, name!".

Once you start thinking in functions, programming feels natural — each one is a small helper, ready when you call.

32. Function Parameters and Return Values

Functions become really useful when they can take input and send back output. That's what parameters and return values are for. They let your functions act like mini machines: you give them data, they do some work, and they hand you back a result.

Think of it like a vending machine: You put in coins (parameters) and get a snack (return value).

32.1 Parameters: Giving Input to a Function

A parameter is a variable that lives inside a function. It receives a value when the function is called.

Example:

```
void greet(char name[]) {
    printf("Hello, %s!\n", name);
}
```

Here, `name` is a parameter. When you call `greet("Alice")`, it prints:

```
Hello, Alice!
```

Each call can give a different input, and the function works with it.

32.2 Multiple Parameters

You can pass as many parameters as you want, separated by commas:

```
int add(int a, int b) {
    return a + b;
}

int main(void) {
    printf("Sum: %d\n", add(3, 5));
    return 0;
}
```

Here, `a` and `b` are parameters. You pass arguments (actual values) when calling it: `add(3, 5)`.

32.3 Parameter Types Matter

Each parameter must have a type. If you pass the wrong type, the compiler warns you (or errors out).

```
float multiply(float x, float y) {
    return x * y;
}
```

C is strongly typed, so your arguments should match.

32.4 The Return Value

A return value is what a function gives back. Use `return` followed by a value:

```
int square(int n) {
    return n * n;
}
```

In `main()`:

```
int result = square(4);
printf("Result: %d\n", result);
```

Output:

```
Result: 16
```

Once `return` runs, the function ends immediately.

32.5 Functions That Return Nothing

If your function just performs an action (like printing), and doesn't need to return anything, use `void`:

```
void say_hi(void) {
    printf("Hi!\n");
}
```

32.6 Functions That Take No Parameters

You can also define functions that don't need any input:

```
int get_magic_number(void) {
    return 42;
}
```

`void` inside the parentheses means "no parameters."

32.7 Input and Output Together

Many functions both take input and return output:

```
int triple(int n) {
    return n * 3;
}

int main(void) {
    int value = 7;
    int result = triple(value);
    printf("Triple: %d\n", result);
    return 0;
}
```

Output:

```
Triple: 21
```

32.8 Expressions with Return Values

Since functions can return a value, you can use them inside expressions:

```
int double_it(int x) { return x * 2; }

int main(void) {
    int sum = double_it(3) + double_it(4);
    printf("Sum: %d\n", sum);
}
```

Output:

```
Sum: 14
```

32.9 Returning Early

You can have multiple return statements inside one function — useful for checking conditions:

```c
int sign(int n) {
    if (n > 0) return 1;
    if (n < 0) return -1;
    return 0;
}
```

The function ends as soon as one `return` runs.

32.10 Matching the Return Type

If a function says it returns `int`, you must return an `int`, or you'll get a warning.

```c
int add(int a, int b) {
    return a + b;  // correct
}
```

If it says `void`, you can't return a value.

Tiny Code

Try this:

```c
#include <stdio.h>

float area_of_circle(float radius) {
    const float pi = 3.14159f;
    return pi * radius * radius;
}

int main(void) {
    float r;
    printf("Enter radius: ");
    scanf("%f", &r);
    printf("Area = %.2f\n", area_of_circle(r));
```

```
    return 0;
}
```

Input → 2 Output → `Area = 12.57`

Why It Matters

Functions are communication points in your program. Parameters let you send data in, return values send results back out.

Once you master this, you can start building libraries of reusable helpers that handle tasks all over your programs.

Try It Yourself

1. Write a function `square(int n)` that returns the square.
2. Write `max(int a, int b)` that returns the larger number.
3. Write `sum3(int a, int b, int c)` that adds three numbers.
4. Write `convert_to_celsius(float f)` that converts Fahrenheit to Celsius.
5. Write a function `is_even(int n)` that returns 1 if even, 0 if odd.

Every function is like a little conversation — you give it something to work with, and it answers back.

33. Local and Global Variables

When you write a program, variables live in different places. Some exist only inside a function, others can be seen everywhere. These two kinds are called local and global variables.

Understanding their scope (where they can be used) helps you avoid bugs, name clashes, and confusion.

33.1 What Are Local Variables

A local variable is declared inside a function. It's created when the function starts, and destroyed when the function ends.

You can only use it inside that function.

Example:

```c
void greet() {
    int count = 1;  // local variable
    printf("Hello %d time!\n", count);
}

int main(void) {
    greet();
    // printf("%d", count); //  Error: count not visible here
    return 0;
}
```

Here, `count` lives only inside `greet()`. Outside, it doesn't exist.

33.2 Why Use Local Variables

Local variables are private to their function. They prevent name conflicts and keep your logic clean.

You can use the same name in different functions:

```c
void f1() { int x = 5; printf("x in f1 = %d\n", x); }
void f2() { int x = 10; printf("x in f2 = %d\n", x); }
```

Each `x` is separate. This is safe and clear, each function manages its own data.

33.3 What Are Global Variables

A global variable is declared outside all functions. It can be used by any function in the file (or even others with `extern`).

```c
#include <stdio.h>

int total = 0;  // global variable

void add_one() {
    total++;
}

int main(void) {
    add_one();
```

```
    add_one();
    printf("Total = %d\n", total);
    return 0;
}
```

Output:

```
Total = 2
```

Here, `total` is shared by both `add_one()` and `main()`.

33.4 Lifetime and Storage

- Local variables: created each time the function runs, destroyed when it ends
- Global variables: created once, stay alive until the program finishes

Globals "remember" their values across function calls.

33.5 When Globals Help

Use globals when:

- You need to share a single value across many functions
- It's something central (like a global config or score)

Example:

```
int score = 0;

void increase() { score++; }
void reset() { score = 0; }
```

But be careful, too many globals make code hard to track.

33.6 When Globals Hurt

Globals are visible everywhere, which can cause trouble:

- Harder to see *who* changes them
- Possible naming conflicts
- Difficult to test functions independently

If a function can use local data, keep it local.

33.7 Local vs Global: Summary

Feature	Local Variable	Global Variable
Declared	Inside a function	Outside all functions
Visible in	That function only	All functions (same file)
Lifetime	Created when function starts	Lives for entire program
Storage	Stack	Static memory
Best for	Temporary data	Shared state

33.8 Shadowing (Careful!)

If a local variable has the same name as a global one, the local one hides the global inside that function.

```c
int x = 10;

void demo() {
    int x = 5; // shadows the global x
    printf("%d\n", x); // prints 5
}
```

Inside `demo()`, the local `x` wins. Outside, the global `x` is still 10.

33.9 Static Local Variables

You can give a local variable a memory across calls using `static`:

```c
void counter() {
    static int count = 0;
    count++;
    printf("Count: %d\n", count);
}
```

Each call remembers the previous value. This is useful for counters, caches, and state.

Tiny Code

Try this:

```c
#include <stdio.h>

int total = 0; // global

void add_points(int p) {
    total += p;
}

void show_total() {
    printf("Total points: %d\n", total);
}

int main(void) {
    int bonus = 5; // local
    add_points(10);
    add_points(bonus);
    show_total();
    return 0;
}
```

Output:

```
Total points: 15
```

Why It Matters

Variables live in scopes, like rooms in a house. Local ones stay private; globals live in the open. When you organize them wisely, your programs become clean, safe, and predictable.

Try It Yourself

1. Write a function that has a local counter and prints it.
2. Add a global score variable and update it from two functions.
3. Try shadowing a global variable, see what happens.
4. Create a `static` local variable that remembers how many times it's called.
5. Compare what happens when you print locals vs globals from different functions.

Good programs are like good stories, each part knows its role. Keep most variables local, use globals sparingly, and your code will stay easy to read and reason about.

34. Scope and Lifetime

Every variable in C lives in a certain place (its *scope*) and for a certain time (its *lifetime*). These two ideas go hand in hand, they tell you where you can use a variable, and how long it exists in memory.

Understanding scope and lifetime helps you avoid surprises like "Why can't I see that variable?" or "Why did it disappear?"

Let's explore these ideas step by step.

34.1 What Is Scope

Scope means *where* a variable can be used. It's the part of your program where the variable's name is visible and valid.

C has several kinds of scope:

- Block scope (inside `{ ... }`)
- Function scope
- File scope (global variables)
- Prototype scope (temporary, in declarations)

Most of the time, you'll work with block scope and file scope.

34.2 Block Scope (Local Variables)

A block is anything inside `{ ... }`, like the body of a function, loop, or `if` statement. Variables declared inside are only visible inside that block.

Example:

```c
#include <stdio.h>

int main(void) {
    int x = 10;

    if (x > 5) {
        int y = 20; // visible only inside if-block
        printf("x = %d, y = %d\n", x, y);
    }

    // printf("%d", y); //  Error: y not visible here
    return 0;
}
```

Here, `y` lives only inside the `if` block. Once the block ends, `y` is gone.

34.3 File Scope (Global Variables)

If you declare a variable outside all functions, it's visible from anywhere in the same file, this is called file scope.

```c
#include <stdio.h>

int total = 0; // file scope

void add_one() { total++; }

int main(void) {
    add_one();
    printf("%d\n", total); // OK: total is visible
    return 0;
}
```

Global variables like `total` are always in memory, they never vanish.

34.4 Function Scope (Labels and `goto`)

This one's rare. Labels used with `goto` are visible throughout a function, even before they appear in the code.

You don't need to worry much, just remember labels belong to the whole function.

34.5 Lifetime: How Long a Variable Exists

A variable's lifetime is how long it stays alive in memory.

- Local variables: created when the function starts, destroyed when it ends
- Static variables: created once, live until the program ends
- Global variables: created when the program starts, destroyed when it ends

Example:

```
void demo() {
    int x = 1;         // new x each time
    static int y = 1; // one y forever

    x++;
    y++;

    printf("x=%d, y=%d\n", x, y);
}
```

If you call `demo()` three times, output is:

```
x=2, y=2
x=2, y=3
x=2, y=4
```

`x` resets each call; `y` remembers across calls.

34.6 Scope Inside Loops

Each loop is its own little block. Variables declared inside exist only inside that loop.

```c
for (int i = 0; i < 3; i++) {
    printf("%d\n", i);
}
// printf("%d", i); //  Error: i not visible here
```

Once the loop ends, `i` is gone.

34.7 Shadowing

If you declare a new variable with the same name in an inner scope, it hides the outer one:

```c
int x = 10;

int main(void) {
    int x = 5; // shadows global x
    printf("%d\n", x); // prints 5
    return 0;
}
```

Be careful, shadowing can confuse you if used too much.

34.8 Scope Rules in Functions

Each function is like its own world. Local variables in one function aren't visible in another:

```c
void f1() { int a = 5; }
void f2() { /* a not visible here */ }
```

If you need data across functions, pass it as a parameter or use a global.

34.9 Best Practices

- Keep scopes small, declare variables where you need them
- Use local variables by default
- Avoid reusing the same name in nested scopes
- Use static only when you need persistence
- Use globals sparingly, only for truly shared data

Tiny Code

Try this:

```c
#include <stdio.h>

void greet(void) {
    int count = 1;          // block scope
    static int called = 0; // remembers across calls

    called++;
    printf("Greet #%d (called %d times)\n", count, called);
}

int main(void) {
    greet();
    greet();
    greet();
    return 0;
}
```

Output:

```
Greet #1 (called 1 times)
Greet #1 (called 2 times)
Greet #1 (called 3 times)
```

`count` is recreated every time, but `called` stays alive between calls.

Why It Matters

Scope and lifetime make your code predictable. They tell you exactly where a variable lives and when it goes away. Once you master them, you'll never wonder "Why can't I see that variable?" again.

Try It Yourself

1. Declare a variable inside an `if`, try printing it outside.
2. Make a loop with a variable inside, try printing it after.
3. Write a function with a `static` counter.

4. Shadow a global with a local variable, see which one wins.
5. Combine a local, static, and global variable in one program.

Every variable has its own *life story* — where it's born, where it lives, and when it disappears. Once you know its journey, your programs will feel much more under control.

35. Header Declarations (`.h` files)

As your programs grow, you'll start splitting them into multiple files. Maybe one file for math functions, another for printing, another for your main logic. To make them work together, you'll need header files.

Header files (`.h` files) are like blueprints, they tell the compiler *what exists*, so you can use it before it's actually defined.

Think of them as introductions between parts of your code:

> "Hey, there's a function called `add()`, you'll meet it later!"

35.1 Why We Need Headers

In small programs, you can put everything in one file. But as you build bigger projects, that gets messy. You'll want to split your code into modules:

- `math.c` → math functions
- `main.c` → main program
- `math.h` → declarations (the header)

Headers help C know about things before they're used. They act as contracts, describing the functions and types each file provides.

35.2 What Goes in a Header File

A header file usually contains:

1. Function declarations (prototypes)
2. Constant definitions (`#define`, `const`)
3. Type definitions (`typedef`, `struct`)
4. Includes for dependencies

No actual function *bodies*, just declarations.

Example: `math.h`

```c
#ifndef MATH_H
#define MATH_H

int add(int a, int b);
int subtract(int a, int b);

#endif
```

35.3 Including a Header

To use a header, include it with `#include "file.h"`:

```c
#include <stdio.h>
#include "math.h"

int main(void) {
    printf("%d\n", add(2, 3));
    return 0;
}
```

The compiler now knows `add()` exists, even if it's defined elsewhere.

35.4 Where to Put Function Definitions

The definitions (the actual code) live in `.c` files.

Example: `math.c`

```c
#include "math.h"

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}
```

Your program might look like this:

```
project/
   main.c
   math.c
   math.h
```

Compile together:

```
gcc main.c math.c -o program
```

35.5 Header Guards

Headers can be included by many files, but you only want them once. Otherwise, you'll get "redefinition" errors.

So, every header should have header guards:

```c
#ifndef FILE_NAME_H
#define FILE_NAME_H

// your declarations

#endif
```

They tell the compiler:

> "If this file is already included, skip it."

35.6 Example Project

Let's build a small program step by step.

math.h

```c
#ifndef MATH_H
#define MATH_H

int add(int a, int b);
int multiply(int a, int b);

#endif
```

math.c

```c
#include "math.h"

int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}
```

main.c

```c
#include <stdio.h>
#include "math.h"

int main(void) {
    printf("2 + 3 = %d\n", add(2, 3));
    printf("2 * 3 = %d\n", multiply(2, 3));
    return 0;
}
```

Compile:

```
gcc main.c math.c -o mathprog
```

Run:

```
2 + 3 = 5
2 * 3 = 6
```

35.7 The Difference Between `.h` and `.c`

File Type	Purpose
`.h` (header)	Declarations, tells what exists
`.c` (source)	Definitions, actual code

Your `.h` says *"this is what you can call"*, your `.c` says *"this is how it works"*.

35.8 Including System Headers

Use angle brackets for system headers (like `stdio.h`):

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

Use quotes for your own headers:

```
#include "math.h"
```

35.9 When to Make a Header

Create a header when:

- You split code into multiple files
- You want to share functions or types
- You're building a library

If your program fits in one file, you don't need one, yet.

Tiny Code

Try this:

greet.h

```
#ifndef GREET_H
#define GREET_H

void say_hello(void);

#endif
```

greet.c

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

void say_hello(void) {
    printf("Hello, C world!\n");
}
```

main.c

```c
#include "greet.h"

int main(void) {
    say_hello();
    return 0;
}
```

Compile:

```
gcc main.c greet.c -o greet
```

Run:

```
Hello, C world!
```

Why It Matters

Headers make your code organized and reusable. They let you build libraries, modules, and projects with many files. Without them, large programs would quickly turn into spaghetti!

Try It Yourself

1. Create a header and source for simple math functions.
2. Add header guards and test including it twice.
3. Split a program into `main.c` and `helper.c` + `helper.h`.
4. Try including your header from two different `.c` files.
5. Add a `typedef struct` to a header, use it in `main.c`.

Headers are your program's road signs — they tell every file what's available and how to find it.

36. Pass by Value Explained

When you call a function in C, your data doesn't *travel* there — instead, a copy is made and sent in. This idea is called pass by value.

It's one of the most important things to understand in C, because it affects how changes inside functions behave. Let's break it down step by step.

36.1 What Does "Pass by Value" Mean

When you call a function with arguments, C copies the values into its parameters. That means the function gets its own independent copies.

Any change made to those copies does not affect the original variables.

Example:

```c
#include <stdio.h>

void double_it(int x) {
    x = x * 2;
    printf("Inside: x = %d\n", x);
}

int main(void) {
    int n = 5;
    double_it(n);
    printf("Outside: n = %d\n", n);
    return 0;
}
```

Output:

```
Inside: x = 10
Outside: n = 5
```

See what happened? `x` changed inside the function, but `n` stayed the same. That's because C made a copy of `n` and passed it in.

36.2 Memory View

Imagine variables as boxes in memory. When you pass one into a function, C makes a new box with the same value.

The two boxes are separate, changing one doesn't touch the other.

```
main:   n = 5
double_it: x = 5 → changed to 10
```

After `double_it` finishes, its `x` is destroyed. `n` is still 5 in `main()`.

36.3 Why This Matters

If you expect your function to modify a variable (like updating a score), pass by value won't work, the changes won't be seen outside.

To actually change the original, you need to pass its address, that's called pass by reference, and you'll learn that soon when we study pointers.

36.4 Multiple Parameters

Each parameter is copied separately:

```c
void add(int a, int b) {
    a = a + b;
    printf("Inside: a = %d, b = %d\n", a, b);
}

int main(void) {
    int x = 2, y = 3;
    add(x, y);
    printf("Outside: x = %d, y = %d\n", x, y);
}
```

Output:

```
Inside: a = 5, b = 3
Outside: x = 2, y = 3
```

Again, nothing changed in `main`.

36.5 Safe and Predictable

Pass by value is safe, functions can't accidentally overwrite your data. They work only with copies, so each call is isolated.

This makes debugging easier, you always know where changes happen.

36.6 Returning New Values

If you want a function to give back a new value, you can return it instead of modifying the input.

```c
int double_it(int x) {
    return x * 2;
}

int main(void) {
    int n = 5;
    n = double_it(n); // save returned value
    printf("n = %d\n", n);
}
```

Output:

```
n = 10
```

You're not modifying `n` inside the function — you're taking the result and storing it outside.

36.7 Pass by Value with Arrays (Sneak Peek)

Arrays behave a little differently, when you pass an array, it *acts like* passing a pointer (more on that later).

But for basic types (int, float, char), C always passes by value.

36.8 Common Mistake

Beginners often expect this to work:

```c
void add_one(int n) {
    n++;
}

int main(void) {
    int x = 5;
    add_one(x);
    printf("%d\n", x); //  still 5, not 6
}
```

`x` doesn't change because you only modified the copy. To fix it, you'll learn how to pass by reference using pointers soon.

Tiny Code

Try this:

```c
#include <stdio.h>

void reset(int n) {
    n = 0;
    printf("Inside reset: n = %d\n", n);
}

int main(void) {
    int value = 10;
    reset(value);
    printf("Outside reset: value = %d\n", value);
    return 0;
}
```

Output:

```
Inside reset: n = 0
Outside reset: value = 10
```

Your variable outside is safe and untouched.

Why It Matters

Pass by value is how C keeps your functions independent and predictable. You can trust that variables won't be changed accidentally. When you *do* want to modify something, you'll use pointers to pass its address (coming soon!).

Try It Yourself

1. Write a function that tries to change an integer, check if it sticks.
2. Return a new value instead of modifying the input.
3. Try passing two variables and print both inside and outside.
4. Predict the output before running, then check!

5. Rewrite a pass-by-value function to return a new result.

In C, functions get their own copies — they can *read* your data, but not *touch* it, unless you explicitly give them the key (a pointer).

37. Recursion and Base Cases

You've already learned that loops let you repeat actions. But C has another elegant way to repeat: recursion.

Recursion means a function calls itself. It's like saying, "I'll solve this big problem by solving a smaller one of the same kind."

Recursion sounds fancy, but once you get it, it's a beautiful tool for breaking problems into smaller steps.

37.1 What Is Recursion

A recursive function is one that calls itself, usually with simpler input.

It must always have two parts:

1. Base case, when to stop
2. Recursive case, when to call itself again

Without a base case, it would run forever (and crash your program).

37.2 A Simple Example

Let's print numbers from 1 to 5 using recursion:

```c
#include <stdio.h>

void count_up(int n) {
    if (n == 0) return; // base case

    count_up(n - 1);    // recursive call
    printf("%d ", n);
}

int main(void) {
    count_up(5);
```

```
    return 0;
}
```

Output:

```
1 2 3 4 5
```

Here's what happens:

- `count_up(5)` calls `count_up(4)`
- `count_up(4)` calls `count_up(3)` … and so on, until `n == 0` Then each call prints its number on the way back up.

37.3 The Base Case

The base case is what stops recursion. It's like saying, "If we're done, don't call again."

Without it, you'll get an infinite recursion → your program will crash with a "stack overflow."

Example of missing base case ( ):

```
void recurse() {
    printf("Hi\n");
    recurse(); // never stops
}
```

So always include a clear base case!

37.4 Factorial Example

Let's compute `n!` (factorial):

- `0! = 1`
- `n! = n * (n-1)!`

```
int factorial(int n) {
    if (n == 0) return 1;       // base case
    return n * factorial(n - 1); // recursive case
}
```

In `main()`:

```c
int main(void) {
    printf("%d\n", factorial(5));
    return 0;
}
```

Output:

```
120
```

Flow:

```
factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120
```

Each call *waits* for the next one's answer.

37.5 Recursion vs Loops

Recursion and loops both repeat, but they think differently:

Feature	Loop	Recursion
Uses	`for`, `while`	Function calls itself
State	Controlled by variables	Controlled by function calls
Needs base condition?	Yes	Yes (base case)
Easier for	Counting, iteration	Trees, divide-and-conquer

Use recursion when a problem can be broken down into smaller versions of itself.

37.6 Another Example: Sum of Numbers

We can write a function to sum numbers from 1 to `n`:

```c
int sum_to_n(int n) {
    if (n == 0) return 0;            // base case
    return n + sum_to_n(n - 1);      // recursive case
}
```

Example:

```c
int main(void) {
    printf("Sum = %d\n", sum_to_n(5));
    return 0;
}
```

Output:

```
Sum = 15
```

37.7 Be Careful with Large Input

Each recursive call uses memory on the call stack. Too many calls can cause stack overflow. For very large inputs, prefer loops.

37.8 Combining Base and Recursive Steps

The pattern always looks like this:

```
type function(args) {
    if (base_condition)
        return base_value;
    else
        return smaller_problem;
}
```

Once you spot this pattern, recursion feels natural.

37.9 Real-World Uses

Recursion is powerful for:

- Searching through files or trees
- Traversing linked lists
- Solving divide-and-conquer problems (like quicksort)
- Generating combinations, permutations, etc.

Even though you might not use it every day, understanding it helps you think like a programmer.

Tiny Code

Try this recursive countdown:

```c
#include <stdio.h>

void countdown(int n) {
    if (n == 0) {
        printf("Go!\n");
        return;
    }
    printf("%d...\n", n);
    countdown(n - 1);
}

int main(void) {
    countdown(5);
    return 0;
}
```

Output:

```
5...
4...
3...
2...
1...
Go!
```

Why It Matters

Recursion is a new way to think about problems — breaking big tasks into smaller ones until they're easy to solve. Once you understand base cases, recursion becomes a simple and elegant tool.

Try It Yourself

1. Write a recursive function to count down from 10 to 1.
2. Write a function to print all numbers from 1 to `n`.
3. Compute `factorial(n)` recursively.
4. Write `sum_to_n(n)` that adds all numbers from 1 to `n`.
5. Write a recursive function `power(a, b)` to compute `a^b`.

Recursion is like a mirror, each call reflects the same task, but a little smaller, until finally it ends.

38. Function Prototypes and Order

In C, your program is read top to bottom. So when you call a function, the compiler must already know it exists — what it's called, what it returns, and what parameters it takes.

But sometimes, you want to call a function that's defined later in the file. That's where function prototypes come in.

Think of a prototype like a forward declaration — a promise to the compiler:

> "This function exists, you'll see its full code soon."

38.1 The Problem Without a Prototype

Let's look at what happens when you don't declare a function before calling it:

```
#include <stdio.h>

int main(void) {
    greet(); //  Compiler doesn't know what greet is
    return 0;
}

void greet() {
```

```
    printf("Hello!\n");
}
```

The compiler reads from top to bottom. When it reaches `greet();` in `main`, it hasn't seen `greet` yet. So it doesn't know what kind of function it is, that's an error.

38.2 The Fix: Add a Prototype

You can fix this by telling C ahead of time what `greet()` looks like.

```
#include <stdio.h>

// Function prototype
void greet(void);

int main(void) {
    greet(); //  Compiler knows greet exists
    return 0;
}

// Function definition
void greet(void) {
    printf("Hello!\n");
}
```

Now C knows:

- The function's name (`greet`)
- Its return type (`void`)
- Its parameter list (`void`, meaning none)

38.3 What Is a Function Prototype

A function prototype is just the function header followed by a semicolon:

```
return_type name(parameter_list);
```

Examples:

```c
int add(int a, int b);
void print_hello(void);
float square(float x);
```

No body, just the declaration.

38.4 Why Prototypes Matter

Prototypes let you:

1. Call functions before they're defined
2. Catch type errors early
3. Organize code cleanly
4. Split code into multiple files

Without them, the compiler can't check if your arguments or return types match.

38.5 Example with Parameters

```c
#include <stdio.h>

int add(int a, int b); // prototype

int main(void) {
    int sum = add(2, 3);
    printf("Sum = %d\n", sum);
    return 0;
}

int add(int a, int b) {
    return a + b;
}
```

Output:

```
Sum = 5
```

Because the prototype comes first, the compiler knows `add()` takes two `int`s and returns an `int`.

38.6 Matching Prototype and Definition

The prototype and definition must match exactly:

- Same return type
- Same parameter types and order

If they don't, the compiler may throw a warning or error.

Example (wrong prototype):

```
int add(int a, int b);    // says it returns int
void add(int a, int b) {} // actually returns void, mismatch
```

Always copy the function header exactly and just add a semicolon.

38.7 Where to Put Prototypes

Usually, prototypes go:

- At the top of your file, before `main()`, or
- In a header file (`.h`), if the function lives in another `.c` file

This way, every file that uses the function can `#include` the header to learn about it.

Example:

math.h

```
int add(int a, int b);
```

math.c

```
#include "math.h"

int add(int a, int b) {
    return a + b;
}
```

main.c

```c
#include <stdio.h>
#include "math.h"

int main(void) {
    printf("%d\n", add(2, 3));
}
```

38.8 Prototypes Help Catch Mistakes

If you call a function with the wrong arguments, the compiler can catch it early, thanks to the prototype.

Example:

```c
int add(int a, int b);

int main(void) {
    add(2); //  compiler error: missing argument
}
```

Without a prototype, C wouldn't warn you — and you'd get undefined behavior at runtime. So prototypes make your programs safer.

38.9 When You Don't Need Them

If your function is defined before it's used, you don't need a prototype:

```c
void greet(void) {
    printf("Hi!\n");
}

int main(void) {
    greet(); // OK: defined earlier
}
```

But adding prototypes is still a good habit — especially for bigger projects.

Tiny Code

Try this:

```c
#include <stdio.h>

// Prototype
float multiply(float a, float b);

int main(void) {
    float x = 2.5, y = 4.0;
    printf("%.2f x %.2f = %.2f\n", x, y, multiply(x, y));
    return 0;
}

// Definition
float multiply(float a, float b) {
    return a * b;
}
```

Output:

```
2.50 x 4.00 = 10.00
```

Why It Matters

Function prototypes are promises you make to the compiler. They let you build programs top-to-bottom without worrying about order. You can organize code neatly and catch errors early — a habit every good C programmer develops.

Try It Yourself

1. Write a program where `main()` calls a function defined *below*. Add a prototype at the top.
2. Try removing the prototype, see the error.
3. Create a prototype that takes two ints and returns a float.
4. Put a prototype in a header file and include it in your `.c` file.
5. Try mismatching the prototype and definition, watch what happens.

With prototypes, your compiler becomes a helpful teammate — it checks your work and keeps your program organized.

39. Inline Functions

You've seen how functions make code cleaner and reusable. But sometimes, calling a function can be a bit slower than running the code directly — especially when the function is tiny and called many times.

That's where inline functions come in. They're a way of telling the compiler,

> "Instead of jumping to this function, just copy its code right here."

It's like giving the compiler a shortcut: no extra call, no return, just straight execution.

39.1 What Is an Inline Function

Normally, when you call a function:

1. The program jumps to that function's code.
2. Runs it.
3. Returns back to where it left off.

For large functions, that's fine. But for tiny ones, the overhead can be noticeable.

An inline function suggests to the compiler:

> "Skip the jump, insert the body right at the call site."

39.2 Syntax

Use the `inline` keyword before the return type:

```
inline int square(int x) {
    return x * x;
}
```

Now, when you call `square(5)`, the compiler *may* replace it with `5 * 5` directly.

39.3 Example

```c
#include <stdio.h>

inline int add_one(int n) {
    return n + 1;
}

int main(void) {
    int x = 5;
    printf("%d\n", add_one(x)); // may become printf("%d\n", x + 1);
    return 0;
}
```

Output:

```
6
```

The behavior is the same — the difference is how the compiler executes it.

39.4 Inline Is a Hint

Important: `inline` is just a suggestion. The compiler can ignore it if it decides inlining isn't worthwhile.

You can think of it like saying,

> "Hey compiler, this function is small, maybe inline it?"

You'll still get the same result, whether it inlines or not.

39.5 When to Use Inline

Good candidates for `inline` are:

- Very short functions (1–3 lines)
- Functions called many times
- Functions that don't do I/O or complex logic

Example:

```c
inline int min(int a, int b) {
    return (a < b) ? a : b;
}
```

Avoid inlining large or complicated functions — that can make your compiled program bigger and even slower.

39.6 Inline vs Macros

Before `inline`, programmers used macros for small helpers:

```
#define SQUARE(x) ((x)*(x))
```

But macros don't respect types or syntax rules — they're just text replacements.

Inline functions are safer:

- Type-checked by the compiler
- Easier to debug
- No weird surprises with parentheses

So, prefer inline functions over macros whenever possible.

39.7 Inline with Header Files

Inline functions are often defined in header files. That way, each file that includes the header can see the full definition.

Example: `math_utils.h`

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

inline int square(int x) {
    return x * x;
}

#endif
```

Then include it in your `.c` files:

```
#include "math_utils.h"
```

Each file gets its own inline copy.

39.8 Inline with `static`

If you mark a function as both `static` and `inline`, it becomes private to the file — each `.c` file has its own local version.

```
static inline int cube(int n) {
    return n * n * n;
}
```

Useful for small helpers that shouldn't be visible outside.

39.9 When Not to Use Inline

Avoid using `inline` on:

- Large functions
- Recursive functions (can't inline themselves)
- Functions that you rarely call

Inlining huge functions can increase code size and reduce performance.

Tiny Code

Try this:

```
#include <stdio.h>

inline int triple(int x) {
    return x * 3;
}

int main(void) {
    for (int i = 1; i <= 5; i++) {
        printf("Triple of %d is %d\n", i, triple(i));
    }
    return 0;
}
```

Output:

```
Triple of 1 is 3
Triple of 2 is 6
Triple of 3 is 9
Triple of 4 is 12
Triple of 5 is 15
```

Simple, fast, and clean.

Why It Matters

Inline functions combine the clarity of normal functions with the speed of direct code. They help you write clean helpers without worrying about overhead.

Remember: `inline` is not magic, just a helpful suggestion for small, frequently used functions.

Try It Yourself

1. Write an `inline` function `square()` and call it in a loop.
2. Compare performance between `SQUARE(x)` macro and `inline square(x)`.
3. Try inlining a large function, see if your compiler warns you.
4. Put an inline helper in a header file and include it.
5. Add `static` to make it file-local.

`inline` is like saying to the compiler, "Hey, this one's tiny, just drop it in right here!" A small trick for big clarity.

40. Organizing Code with Functions

By now, you've learned how to write, call, and declare functions, and even how to use prototypes and inline helpers. Now it's time to see the big picture: how functions help you organize your program.

Functions aren't just about saving keystrokes, they're about structuring your code like a well-arranged toolbox, where every tool has a clear name and purpose.

Let's learn how to think in functions, building your programs piece by piece.

40.1 Why Organize with Functions

Imagine writing everything in `main()`: you'd have 100 lines for setup, 50 lines for logic, 30 for output, total chaos.

Functions let you break that chaos into logical sections:

- Each part does one thing well
- You can test parts separately
- You can reuse them later

It's the difference between a messy desk and a neat one with labeled drawers.

40.2 Think "One Task = One Function"

Every function should do one clear job. If you can describe it with a short verb phrase — like `print_menu`, `get_input`, `compute_total`, you're on the right track.

Example:

```
void print_menu(void) {
    printf("1. Add\n2. Subtract\n3. Quit\n");
}
```

This makes your code self-documenting, easy to read and understand.

40.3 Breaking a Program into Parts

Let's say we're writing a simple calculator. Instead of stuffing everything into `main()`, we can organize it like this:

```
#include <stdio.h>

void show_menu(void);
int add(int a, int b);
int subtract(int a, int b);

int main(void) {
    int choice, x, y;

    show_menu();
    printf("Choose: ");
```

```
    scanf("%d", &choice);

    printf("Enter two numbers: ");
    scanf("%d %d", &x, &y);

    if (choice == 1)
        printf("Result: %d\n", add(x, y));
    else if (choice == 2)
        printf("Result: %d\n", subtract(x, y));
    else
        printf("Goodbye!\n");

    return 0;
}

void show_menu(void) {
    printf("1. Add\n2. Subtract\n3. Quit\n");
}

int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
```

Each function has one tiny, clear job. If something breaks, you know exactly where to look.

40.4 Avoid "God Functions"

A God function does *everything*. It's long, hard to read, and easy to break.

Bad:

```
void program() {
    printf("Welcome\n");
    int a, b;
    scanf("%d %d", &a, &b);
    int c = a + b;
    printf("%d", c);
    // many more lines...
}
```

Good:

```c
void greet(void);
void get_numbers(int *a, int *b);
int add(int a, int b);
void show_result(int sum);
```

Small, simple pieces fit together like building blocks.

40.5 Group by Purpose

Keep related functions together:

- All math helpers in one place
- All printing functions together
- All input functions together

Later, you'll learn to split these into separate `.c` and `.h` files — but even within one file, grouping helps clarity.

Example structure:

```c
// Math
int add(...);
int subtract(...);

// Input
void get_numbers(...);

// Output
void show_result(...);
```

40.6 Reuse Functions

Once you've written a clean, general function, you can use it everywhere. No need to rewrite the same logic.

Example:

```c
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

Now you can call `max()` in any program that needs a quick comparison.

40.7 Top-Down Design

Start by sketching the big picture, the main steps, then fill in details.

Example outline:

```
int main(void) {
    greet_user();
    int choice = menu();
    handle_choice(choice);
    return 0;
}
```

Later, write those functions one by one. This keeps `main()` short, more like a storyboard than a script.

40.8 Testing Each Function

Because functions are independent, you can test them easily.

Example:

```
printf("Add test: %d\n", add(2, 3)); // should print 5
```

You can build confidence piece by piece instead of debugging a huge file all at once.

40.9 Refactoring into Functions

If you notice the same code appearing twice, extract it into a new function.

Before:

```
printf("Enter number: ");
scanf("%d", &n);
```

After:

```
int get_number(void) {
    int n;
    printf("Enter number: ");
    scanf("%d", &n);
    return n;
}
```

Now you can just call `get_number()` whenever you need it.

Tiny Code

Try this mini-program:

```c
#include <stdio.h>

void greet(void) {
    printf("Welcome!\n");
}

int get_input(void) {
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);
    return x;
}

void show_double(int n) {
    printf("Twice that is %d\n", n * 2);
}

int main(void) {
    greet();
    int num = get_input();
    show_double(num);
    return 0;
}
```

Clean, simple, and easy to follow — each step is its own function.

Why It Matters

Functions turn messy code into organized chapters. They make your programs easier to read, debug, and expand. When each part has one purpose, you can grow your code without fear.

Try It Yourself

1. Take one of your earlier programs and break it into functions.

2. Keep `main()` short, just 5-10 lines describing the flow.
3. Give each function a clear, action-based name.
4. Group related functions together.
5. Test each function on its own before combining them.

Functions are the building blocks of every C program. Use them like paragraphs in an essay — each one should say one clear thing.

Chapter 5. Arrays and Strings

41. Declaring Arrays

So far, you've learned how to store single values, one `int`, one `float`, one `char`. But what if you want to store many values of the same type, like a list of numbers, or a word made of letters?

That's where arrays come in. An array is like a row of boxes, each holding one value of the same type. You can access each box by its position, called an index.

41.1 What Is an Array

An array is a collection of elements of the same type, stored side by side in memory.

For example, an array of 5 integers looks like this:

```
+----+----+----+----+----+
|  0 |  1 |  2 |  3 |  4 |
+----+----+----+----+----+
```

Each slot has an index, starting from 0. So the first element is at index `0`, not `1`.

41.2 Declaring an Array

To declare an array, you write the type, the name, and the size in square brackets:

```c
int numbers[5];
```

This creates space for 5 integers, all initialized with garbage values (whatever happens to be in memory).

You can also declare arrays of other types:

```c
float prices[10];
char letters[26];
```

41.3 Initializing Arrays

You can give an array initial values at the same time:

```c
int numbers[5] = {1, 2, 3, 4, 5};
```

If you leave out the size, C will count for you:

```c
int numbers[] = {1, 2, 3, 4, 5};
```

You can also partially fill an array, C fills the rest with zeros:

```c
int scores[5] = {10, 20}; // {10, 20, 0, 0, 0}
```

41.4 Accessing Elements

You can access (read or write) each element using its index:

```c
numbers[0] = 42;        // set first element
printf("%d\n", numbers[0]); // read first element
```

Indexes go from `0` to `size - 1`. If your array has 5 elements, valid indexes are `0, 1, 2, 3, 4`.

41.5 Arrays and Loops

Arrays and loops go hand in hand. You can use a `for` loop to set or print all elements:

```c
for (int i = 0; i < 5; i++) {
    numbers[i] = i * 2;
}
```

Now `numbers` contains `{0, 2, 4, 6, 8}`.

41.6 Memory Layout

All elements in an array are stored contiguously, one after another. This makes arrays fast and efficient.

You can imagine them as one long shelf of identical boxes.

41.7 Fixed Size

In C, array sizes are fixed, once you choose a size, you can't change it later. If you need a flexible list, you'll use pointers and dynamic memory (coming soon in Chapter 6).

Tiny Code

Here's a simple example:

```c
#include <stdio.h>

int main(void) {
    int scores[5] = {90, 85, 88, 92, 95};

    printf("Scores:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", scores[i]);
    }

    printf("\n");
    return 0;
}
```

Output:

```
Scores:
90 85 88 92 95
```

You just created a list of scores, neat and tidy in memory.

Why It Matters

Arrays are the foundation of data handling in C. They let you store and work with groups of values efficiently, whether it's numbers, characters, or more complex data later. Once you master arrays, you'll unlock strings, matrices, and even dynamic memory.

Try It Yourself

1. Declare an array of 5 integers and fill it with your favorite numbers.
2. Write a loop to print all elements.
3. Try changing one element and print again.
4. Create an array of `char` with the first 5 letters of the alphabet.
5. Leave out the size, let C count for you.

Arrays are your first step from single values to collections — they let your programs remember not just one thing, but many, all in order.

42. Indexing and Bounds

Arrays give you many boxes, but you need to know exactly which box you're working with. That's what indexing is for, it lets you access each element in an array by its position.

In C, array indexes always start at 0, not 1. That means the first element is `array[0]`, and the last element is `array[size - 1]`.

42.1 Index Basics

To access an element, write the array name followed by the index in square brackets:

```c
int numbers[3] = {10, 20, 30};

printf("%d\n", numbers[0]); // prints 10
printf("%d\n", numbers[1]); // prints 20
printf("%d\n", numbers[2]); // prints 30
```

Each index gives you direct access to one slot in memory.

You can also assign values the same way:

```c
numbers[1] = 99; // change second element
```

Now `numbers` becomes `{10, 99, 30}`.

42.2 Zero-Based Indexing

Since counting starts at 0, an array with `n` elements has indexes from 0 to n-1. If you create an array of 5 elements, the valid indexes are `0, 1, 2, 3, 4`.

```
int data[5];
```

Valid indexes: `data[0]` … `data[4]`

Trying to access `data[5]` is a mistake, that index doesn't exist.

42.3 Out-of-Bounds Access

C will not stop you from going outside an array's range. It won't warn you, it won't crash immediately, but it will lead to undefined behavior.

Example of a bug:

```
int values[3] = {1, 2, 3};
values[3] = 100; //   out of bounds
```

This writes to memory that doesn't belong to the array. Sometimes nothing seems wrong, sometimes the program crashes, it's unpredictable. Always make sure your indexes are within range.

42.4 Using Loops Safely

When looping through arrays, use the correct condition to stay in bounds:

```
int numbers[5] = {2, 4, 6, 8, 10};

for (int i = 0; i < 5; i++) {
    printf("%d ", numbers[i]);
}
```

The loop runs from `i = 0` to `i = 4`. If you write `i <= 5`, you'll go one step too far.

42.5 Calculating the Last Index

If you know the size, the last index is always `size - 1`.

You can also compute the number of elements using `sizeof`:

```
int numbers[5];
int length = sizeof(numbers) / sizeof(numbers[0]);
```

Now `length` holds `5`, so you can loop safely:

```
for (int i = 0; i < length; i++) {
    // safe access
}
```

42.6 Reading and Writing Elements

You can use indexes to both read and write values:

```
int scores[3];
scores[0] = 70;
scores[1] = 80;
scores[2] = 90;

printf("First score: %d\n", scores[0]);
```

Each element acts like its own variable.

42.7 Indexes as Variables

You don't have to use numbers directly; variables work too:

```
int i = 2;
printf("%d\n", numbers[i]); // prints third element
```

This is handy in loops and calculations.

42.8 Accessing in Reverse

You can loop backward from the last element to the first:

```
for (int i = 4; i >= 0; i--) {
    printf("%d ", numbers[i]);
}
```

This prints elements in reverse order.

Tiny Code

```
#include <stdio.h>

int main(void) {
    int data[4] = {5, 10, 15, 20};

    printf("Forward: ");
    for (int i = 0; i < 4; i++) {
        printf("%d ", data[i]);
    }

    printf("\nReverse: ");
    for (int i = 3; i >= 0; i--) {
        printf("%d ", data[i]);
    }

    printf("\n");
    return 0;
}
```

Output:

```
Forward: 5 10 15 20
Reverse: 20 15 10 5
```

Why It Matters

Indexing is how you reach into arrays to use or change their contents. Learning to stay within bounds keeps your programs safe and bug-free. Every loop over an array depends on correct indexing.

Try It Yourself

1. Create an array of 5 numbers and print each with its index.
2. Change the middle element and print the array again.
3. Write a loop that prints the array in reverse.
4. Try using a variable index to access an element.
5. Experiment with an out-of-bounds index and observe what happens (but don't rely on it).

Arrays are precise, know your indexes, and they'll always work exactly as expected.

43. Multidimensional Arrays

So far, arrays have been a simple line of boxes, a single row. But what if you want a grid of numbers, like a table, or a matrix with rows and columns?

That's when you use multidimensional arrays. They let you store data in rows and columns, or even more dimensions if you need them.

43.1 What Is a Multidimensional Array

A multidimensional array is just an array of arrays. Each row is its own array, and all rows are grouped together.

For example, a 2D array of 3 rows and 4 columns looks like this:

```
+----------------------+
|  row 0  | [0][0] [0][1] [0][2] [0][3]
|  row 1  | [1][0] [1][1] [1][2] [1][3]
|  row 2  | [2][0] [2][1] [2][2] [2][3]
+----------------------+
```

You can imagine it like a spreadsheet, each element has a row and a column index.

43.2 Declaring a 2D Array

To declare a 2D array, write both sizes in square brackets:

```
int matrix[3][4];
```

This creates 3 rows and 4 columns, for a total of `3 * 4 = 12` integers.

You can also declare higher dimensions, but 2D is the most common.

43.3 Initializing a 2D Array

You can fill a 2D array at once using nested braces:

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

This creates:

```
1 2 3
4 5 6
```

You can also leave out the first size if you give all rows:

```
int grid[][2] = {
    {10, 20},
    {30, 40},
    {50, 60}
};
```

C will count the rows for you.

43.4 Accessing Elements

Each element needs two indexes, one for the row, one for the column:

```
printf("%d\n", matrix[1][2]); // row 1, column 2
```

In the earlier example, `matrix[1][2]` is `6`.

You can assign the same way:

```
matrix[0][0] = 99;
```

Now the top-left element is `99`.

43.5 Using Nested Loops

You can loop through a 2D array with nested loops, one for rows, one for columns:

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

This prints all elements row by row.

43.6 Changing Elements

You can update individual elements just like in 1D arrays:

```
matrix[0][1] = 42;
matrix[1][0] = 99;
```

After these changes, the array becomes:

```
1 42 3
99 5 6
```

43.7 Arrays with More Dimensions

You can make 3D arrays too, arrays of arrays of arrays:

```
int cube[2][3][4];
```

But start simple. Most problems can be solved with 1D or 2D arrays.

43.8 Memory Layout

All elements are stored contiguously in memory, just like 1D arrays. C stores them row by row, the first row, then the second, and so on.

This is called row-major order.

43.9 Real Examples

You might use 2D arrays for:

- A tic-tac-toe board (`3x3`)
- A chess board (`8x8`)
- A matrix in math or graphics
- Tables of scores or coordinates

Whenever you need rows and columns, 2D arrays are the natural fit.

Tiny Code

```c
#include <stdio.h>

int main(void) {
    int table[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    printf("Table:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", table[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Table:
1 2 3
4 5 6
```

Why It Matters

Multidimensional arrays let you model structured data, grids, tables, maps, matrices, with ease. They're a key building block for games, graphics, and numerical computing.

Try It Yourself

1. Create a `3x3` matrix of numbers and print it.
2. Use nested loops to fill a table with `i + j`.
3. Change one element and print again.
4. Make a 2D array of grades (rows = students, columns = tests).
5. Practice accessing elements at different positions.

Once you understand rows and columns, arrays start to feel like little worlds of data you can navigate freely.

44. Iterating over Arrays

Arrays are powerful because they hold many values, but to use those values, you need a way to go through them one by one. That process is called iteration.

When you *iterate* over an array, you visit each element in order, often using a loop. It's like checking each box in a row and reading what's inside.

44.1 Why Iterate

If you want to:

- Print all elements
- Compute a total or average
- Modify each value
- Search for something

You'll need to loop over the array.

Doing this manually for every element would be slow and error-prone:

```
printf("%d", arr[0]);
printf("%d", arr[1]);
printf("%d", arr[2]);
```

Instead, you use a loop to handle it automatically.

44.2 Using a for Loop

The `for` loop is perfect for arrays because it gives you a counter variable (an index):

```
int numbers[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; i++) {
    printf("%d\n", numbers[i]);
}
```

This starts from index `0`, goes up to `4`, and prints every element.

Each time through the loop:

- `i` moves to the next index
- `numbers[i]` accesses that element

44.3 Counting Automatically

You can use `sizeof` to compute the array length, so your loop always fits:

```
int length = sizeof(numbers) / sizeof(numbers[0]);
for (int i = 0; i < length; i++) {
    printf("%d ", numbers[i]);
}
```

Now if you change the array size later, the loop still works correctly.

44.4 Using while Loops

You can also use a `while` loop, though `for` is more common:

```
int i = 0;
while (i < 5) {
    printf("%d ", numbers[i]);
    i++;
}
```

Same result, just a different style.

44.5 Iterating in Reverse

You don't always need to go forward. You can loop backward too:

```
for (int i = 4; i >= 0; i--) {
    printf("%d ", numbers[i]);
}
```

This prints the array in reverse order.

44.6 Doing Work Inside the Loop

You can do more than just print, any logic works inside:

```
int sum = 0;
for (int i = 0; i < 5; i++) {
    sum += numbers[i];
}
printf("Sum = %d\n", sum);
```

You can also update each element:

```
for (int i = 0; i < 5; i++) {
    numbers[i] *= 2;
}
```

Now every value is doubled.

44.7 Nested Loops for 2D Arrays

If you're working with a 2D array, use one loop for rows and one for columns:

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};

for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]);
```

```
    }
    printf("\n");
}
```

Each `i` goes through a row, and each `j` walks across the columns.

44.8 Be Careful with Bounds

Never let your loop index go outside the array's range. If your array has 5 elements, stop at `i < 5`. Going to `i <= 5` causes undefined behavior, it might crash or print garbage.

44.9 Useful Patterns

Here are some common array patterns:

Sum of elements:

```
int sum = 0;
for (int i = 0; i < n; i++) sum += arr[i];
```

Find maximum:

```
int max = arr[0];
for (int i = 1; i < n; i++)
    if (arr[i] > max) max = arr[i];
```

Count positives:

```
int count = 0;
for (int i = 0; i < n; i++)
    if (arr[i] > 0) count++;
```

Tiny Code

```c
#include <stdio.h>

int main(void) {
    int data[5] = {3, 7, 2, 8, 5};
    int sum = 0;

    printf("Data: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", data[i]);
        sum += data[i];
    }

    printf("\nSum = %d\n", sum);
    return 0;
}
```

Output:

```
Data: 3 7 2 8 5
Sum = 25
```

Why It Matters

Iteration is how you work with collections. Instead of handling elements one by one, you write one clean loop that handles them all. It's efficient, readable, and a key part of almost every C program.

Try It Yourself

1. Create an array of 10 integers and print each one.
2. Write a loop to calculate the sum and average.
3. Loop backward and print in reverse order.
4. Double every element inside the loop.
5. Write a loop to find the largest number in the array.

Once you can iterate confidently, you can explore, transform, and analyze any data stored in arrays.

45. Strings as Character Arrays

In C, a string isn't a special type, it's just an array of characters. Each letter is stored in one slot, side by side, ending with a special symbol that marks the end of the string.

That's why learning strings is really about understanding character arrays.

45.1 What Is a String

A string is a sequence of characters terminated by a null character, written as `'\0'`. This special character tells C,

> "Stop reading, this is the end of the string."

Example:

```c
char word[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

You can picture it like this:

```
+---+---+---+---+---+----+
| H | e | l | l | o | \0 |
+---+---+---+---+---+----+
```

Every string in C must end with `'\0'`.

45.2 Using String Literals

Typing all the characters by hand can be tedious, so C lets you create strings more easily with string literals:

```c
char word[] = "Hello";
```

C automatically adds the `'\0'` at the end for you.

That means these two lines are the same:

```c
char a[] = {'H', 'i', '\0'};
char b[] = "Hi";
```

Both store `"Hi"` in memory.

45.3 Declaring Strings

You can declare a string with a fixed size:

```
char name[10];
```

This creates space for up to 9 visible characters, plus one `'\0'` at the end.

Always leave room for the terminator — if your string has 9 letters, the array must be at least size 10.

45.4 Accessing Characters

Strings are just arrays, so you can use indexes:

```
char word[] = "Cat";
printf("%c\n", word[0]); // prints C
printf("%c\n", word[1]); // prints a
printf("%c\n", word[2]); // prints t
```

You can also change individual characters:

```
word[0] = 'B'; // now "Bat"
```

45.5 Printing Strings

Use `%s` with `printf` to print an entire string:

```
char greeting[] = "Hello, world!";
printf("%s\n", greeting);
```

C will print characters one by one until it hits `'\0'`.

45.6 Reading Strings

Later, you'll learn how to read strings using `scanf` and safer alternatives. For now, you can assign string literals directly or use them in output.

45.7 Strings vs Characters

A character is a single letter, like `'A'`. A string is an array of letters, like `"A"`.

Notice the difference:

- `'A'` uses single quotes, a `char`
- `"A"` uses double quotes, a string (2 chars: `'A'` and `'\0'`)

45.8 Strings in Memory

C stores all string characters in contiguous memory, just like arrays. The `'\0'` marks where the string ends, not necessarily the end of the array. That's why you should never forget it, otherwise C will keep reading random memory!

45.9 Changing a String

You can modify elements of a string you've declared as an array:

```
char name[] = "Tom";
name[1] = 'i'; // now "Tim"
```

But if you declare it as a pointer to a literal:

```
char *name = "Tom";
```

You can't change it safely, string literals are read-only. Always use `char name[] = "..."` if you plan to modify it.

Tiny Code

```
#include <stdio.h>

int main(void) {
    char word[] = "Hello";

    printf("Word: %s\n", word);
    printf("First letter: %c\n", word[0]);

    word[0] = 'Y';
```

```
    printf("New word: %s\n", word);

    return 0;
}
```

Output:

```
Word: Hello
First letter: H
New word: Yello
```

Why It Matters

Strings are how you handle text in C, names, messages, words, anything made of letters. They're built on top of character arrays, so understanding them helps you work with text safely and clearly.

Try It Yourself

1. Declare a string `"Hello"` and print it.
2. Print each character using a loop.
3. Change one character and print again.
4. Create an array of size 6 and fill it with `"World"`.
5. Try forgetting `'\0'`, see what happens (it may print garbage!).

Strings in C are simple but powerful, once you see them as arrays with a clear end, they make perfect sense.

46. String Literals and Null Terminators

Every string in C ends with a special invisible character: `'\0'`, called the null terminator. This single symbol tells C where the string stops. Without it, your program would keep reading into memory, printing garbage or crashing.

String literals, words in double quotes like `"Hello"`, automatically include this terminator. So when you write `"Hi"`, C really stores three characters: `'H'`, `'i'`, and `'\0'`.

46.1 What Are String Literals

A string literal is text inside double quotes, like `"C programming"`. It's stored in memory as an array of characters ending with `'\0'`.

Example:

```
char greeting[] = "Hi";
```

Memory looks like:

```
+---+---+----+
| H | i | \0 |
+---+---+----+
```

C automatically adds `'\0'` at the end, you don't need to type it.

46.2 Declaring with String Literals

The easiest way to make a string is with a literal:

```
char word[] = "Hello";
```

This array has 6 elements: 5 letters + 1 terminator. If you print it:

```
printf("%s\n", word);
```

You'll see `Hello`, `'\0'` isn't shown because it's not a visible character.

46.3 Null Terminator: Why It Matters

C does not store the string's length anywhere. Instead, it depends on the terminator to know where to stop reading.

So `printf("%s", word);` works like this:

- Start at the first character.
- Keep printing until you find `'\0'`.
- Stop right there.

If `'\0'` is missing, it just keeps going into memory, printing whatever it finds, total chaos!

46.4 Adding Your Own Terminator

If you build a string manually, don't forget to add `'\0'`:

```
char msg[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Without it:

```
char bad[5] = {'H', 'e', 'l', 'l', 'o'}; //   no terminator
```

`printf("%s", bad);` may print random characters after "Hello".

46.5 Changing Strings Safely

You can modify a string declared as an array:

```
char name[] = "Tom";
name[0] = 'J'; // now "Jom"
```

But you cannot modify a string literal directly:

```
char *name = "Tom"; // stored in read-only memory
name[0] = 'J';      //   undefined behavior
```

If you want to change it, always declare with square brackets.

46.6 Measuring Length

To measure a string's visible length (not counting `'\0'`), use the function `strlen()` from `<string.h>`:

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

int main(void) {
    char text[] = "Hi";
    printf("Length = %zu\n", strlen(text));
    return 0;
}
```

Output:

```
Length = 2
```

`strlen` counts characters until it finds `'\0'`.

46.7 Adding the Terminator Manually

Sometimes you might build a string in pieces. Remember to add `'\0'` at the end yourself:

```
char word[6];
word[0] = 'H';
word[1] = 'e';
word[2] = 'l';
word[3] = 'l';
word[4] = 'o';
word[5] = '\0';
```

Now it's a valid string.

46.8 Strings Are Arrays with a Rule

Every string is an array of `char` that must end with `'\0'`. That's the only rule, but it's a very important one. Follow it, and your strings will behave perfectly.

Tiny Code

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

int main(void) {
    char greeting[] = "Hello";

    printf("String: %s\n", greeting);
    printf("Length: %zu\n", strlen(greeting));

    greeting[0] = 'Y';
    printf("New String: %s\n", greeting);

    return 0;
}
```

Output:

```
String: Hello
Length: 5
New String: Yello
```

Why It Matters

The null terminator is the heartbeat of every string in C. It marks the end, keeps your program safe, and makes functions like `printf` and `strlen` work correctly.

Once you understand it, you'll never be surprised by stray characters again.

Try It Yourself

1. Declare a string `"World"` and print it.
2. Count its length using `strlen()`.
3. Change one character and print again.
4. Try removing the terminator (leave it out) and see what happens.
5. Build a string manually, adding `'\0'` yourself.

Strings in C are friendly and simple, as long as you never forget to close them with `'\0'`.

47. Common String Functions (strlen, strcpy, strcmp)

C gives you a collection of handy tools for working with strings. They live in the `<string.h>` library, and they help you do things like measure, copy, and compare strings, safely and quickly.

Let's explore the three most common ones:

- `strlen` – find the length
- `strcpy` – copy one string into another
- `strcmp` – compare two strings

Once you know these, you'll be able to handle most everyday string tasks with confidence.

47.1 Measuring Length with `strlen`

`strlen()` tells you how many visible characters are in a string, not counting the null terminator.

```
#include <string.h>

size_t strlen(const char *str);
```

Example:

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

int main(void) {
    char word[] = "Hello";
    printf("Length: %zu\n", strlen(word));
    return 0;
}
```

Output:

```
Length: 5
```

`strlen` counts letters until it reaches `'\0'`. If your string is `"Hi"`, it returns `2`. If it's `"Hello"`, it returns `5`.

47.2 Copying Strings with `strcpy`

`strcpy()` copies the content of one string into another.

```
#include <string.h>

char *strcpy(char *dest, const char *src);
```

You need to make sure the destination has enough space for the source string plus `'\0'`.

Example:

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

int main(void) {
    char src[] = "Apple";
    char dest[10];

    strcpy(dest, src);
    printf("Copied string: %s\n", dest);
    return 0;
}
```

Output:

```
Copied string: Apple
```

Be careful: if the destination array is too small, `strcpy` will overflow, so always size it large enough!

A safer alternative is `strncpy(dest, src, size)`, which limits the number of characters copied.

47.3 Comparing Strings with `strcmp`

You can't compare strings with `==`, that would only check if they're in the same memory location, not if they have the same content. Use `strcmp()` instead.

```c
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

It returns:

- `0` if the strings are equal
- A negative number if `s1` comes before `s2`
- A positive number if `s1` comes after `s2`

Example:

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

int main(void) {
    char a[] = "Cat";
    char b[] = "Dog";

    if (strcmp(a, b) == 0)
        printf("Same!\n");
    else
        printf("Different!\n");

    return 0;
}
```

Output:

```
Different!
```

If you compare `"Apple"` and `"Banana"`, `strcmp("Apple", "Banana")` is negative, because `"Apple"` would come first alphabetically.

47.4 Summary of Key String Functions

Function	Purpose	Example
`strlen(s)`	Length of string (no `'\0'`)	`strlen("Hi")` → 2
`strcpy(d, s)`	Copy string `s` into `d`	`strcpy(dest, "Hello")`
`strcmp(a, b)`	Compare two strings	`strcmp("Cat", "Cat")` → 0

Remember: all these functions rely on `'\0'` to know where the string ends. If a string isn't terminated, the results are unpredictable.

47.5 Practical Example

Let's combine them:

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

int main(void) {
    char first[20];
    char second[20];

    strcpy(first, "Hello");
    strcpy(second, "World");

    printf("First: %s (%zu letters)\n", first, strlen(first));
    printf("Second: %s (%zu letters)\n", second, strlen(second));

    if (strcmp(first, second) == 0)
        printf("Strings are the same!\n");
    else
        printf("Strings are different!\n");

    return 0;
}
```

Output:

```
First: Hello (5 letters)
Second: World (5 letters)
Strings are different!
```

47.6 Common Mistakes

- Forgetting to include `<string.h>`
- Copying into an array that's too small
- Comparing strings with `==` instead of `strcmp()`
- Using `strlen` on uninitialized data

If you always initialize and terminate your strings properly, these functions will behave perfectly.

Tiny Code

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

int main(void) {
    char src[] = "C language";
    char dest[20];

    strcpy(dest, src);
    printf("Copied: %s\n", dest);
    printf("Length: %zu\n", strlen(dest));

    if (strcmp(src, dest) == 0)
        printf("They match!\n");

    return 0;
}
```

Output:

```
Copied: C language
Length: 10
They match!
```

Why It Matters

String functions save you from reinventing the wheel. Instead of writing loops to count, copy, or compare, you can use these tested, reliable tools. They're part of the core skillset for every C programmer.

Try It Yourself

1. Use `strlen` to find the length of `"Programming"`.
2. Copy `"Hello"` into another string using `strcpy`.
3. Compare `"Dog"` and `"Cat"` with `strcmp`.
4. Try copying a long string into a small array and see what happens.
5. Rewrite your own version of `strlen()` using a loop for practice.

Once you master these, strings will start to feel easy and familiar, just arrays with some friendly helper tools.

48. Inputting Strings

You've learned how to declare, print, and modify strings, now it's time to learn how to read them from the user. Getting text input is a big part of interactive programs, and in C, you have a few simple ways to do it.

Let's go step by step and learn how to safely take string input from the keyboard.

48.1 Using `scanf`

The simplest way to read a string is with `scanf()` and the `%s` format specifier:

```c
#include <stdio.h>

int main(void) {
    char name[20];
    printf("Enter your name: ");
    scanf("%s", name);
    printf("Hello, %s!\n", name);
    return 0;
}
```

If you type `Alice`, it prints:

```
Hello, Alice!
```

When you use `%s`, `scanf` reads characters until the first space. So if you type `Alice Smith`, it only reads `"Alice"`.

48.2 Always Leave Room

C doesn't know how big your array is. You must make sure your buffer (array) is big enough to hold all characters plus the `'\0'` terminator.

If your array is `char name[20];`, you can safely read up to 19 visible characters.

If you want to be extra careful, you can tell `scanf` to limit the input length:

```c
scanf("%19s", name);
```

This tells C: "read at most 19 characters."

48.3 Reading Full Lines with `fgets`

If you want to read spaces, use `fgets()` instead. It reads an entire line (including spaces) up to a maximum number of characters or until Enter is pressed.

```c
#include <stdio.h>

int main(void) {
    char sentence[50];
    printf("Enter a sentence: ");
    fgets(sentence, sizeof(sentence), stdin);
    printf("You wrote: %s", sentence);
    return 0;
}
```

If you type:

```
Hello world from C!
```

It prints:

```
You wrote: Hello world from C!
```

`fgets` includes the newline `\n` at the end if there's space. You can remove it if needed (we'll learn how later).

48.4 Comparing `scanf` and `fgets`

Function	Reads Spaces?	Needs Size Limit?	Stops At
`scanf("%s")`	No	Recommended	First space
`fgets()`	Yes	Required	Newline or end of buffer

If you're reading names without spaces, `scanf` is fine. If you're reading sentences or phrases, use `fgets`.

48.5 Avoiding Buffer Overflows

A buffer overflow happens when you read more characters than your array can hold. Always provide a limit with `scanf` or use `fgets`, which is safer.

Bad:

```
char word[5];
scanf("%s", word); //  dangerous
```

Good:

```
char word[5];
scanf("%4s", word); //  room for 4 chars + '\0'
```

48.6 Example: Asking for Two Words

You can read multiple strings by calling `scanf` multiple times:

```
char first[20], last[20];
printf("Enter first and last name: ");
scanf("%s %s", first, last);
printf("Hello, %s %s!\n", first, last);
```

If you input:

```
John Smith
```

You get:

```
Hello, John Smith!
```

48.7 Mixing `scanf` and `fgets`

If you mix `scanf` (which leaves a newline in the buffer) with `fgets`, you might need to clear the input buffer first. To keep things simple, try sticking with one method per program until you're comfortable.

48.8 Strings and Safety

Reading strings is one of the most common sources of errors in C. Follow these safety rules:

1. Always declare arrays large enough.
2. Always limit input size.
3. Use `fgets()` when reading full lines.
4. Remember strings must end with `'\0'`.

Follow those, and your input will always behave nicely.

Tiny Code

```c
#include <stdio.h>

int main(void) {
    char name[30];
    char quote[50];

    printf("What is your name? ");
    scanf("%29s", name);

    printf("Hello, %s! What's your favorite quote?\n", name);
    getchar(); // clear newline left by scanf
    fgets(quote, sizeof(quote), stdin);

    printf("Nice quote, %s!\n", name);
    printf("You said: %s", quote);

    return 0;
}
```

Example run:

```
What is your name? Alex
Hello, Alex! What's your favorite quote?
Practice makes perfect.
Nice quote, Alex!
You said: Practice makes perfect.
```

Why It Matters

Inputting strings lets your programs talk to people. It's how you turn your program from something fixed into something interactive and personal. With `scanf` and `fgets`, you can build forms, quizzes, chatbots, and more.

Try It Yourself

1. Ask the user for their first name and greet them.
2. Use `fgets()` to read a full sentence and print it.
3. Try `scanf("%10s", name)` to see how length limits work.
4. Combine `scanf` and `fgets` in one program, handle the newline correctly.
5. Write a program that reads two words and prints them reversed.

Once you master reading strings, you'll be ready to create friendly, interactive programs that respond to the user's words.

49. Arrays vs. Pointers (A Gentle Intro)

Arrays and pointers look a lot alike in C, and that can be confusing at first. They both deal with memory, they both use indexes, and sometimes you can even use them interchangeably.

But they're not the same thing. An array is a block of memory, while a pointer is a variable that holds an address.

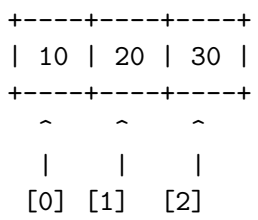
Let's walk through the connection step by step.

49.1 Arrays Are Blocks of Memory

When you write:

```c
int numbers[3] = {10, 20, 30};
```

C creates 3 slots in memory, side by side:



Each element lives at a specific address. The array's name (`numbers`) points to the address of the first element.

So, `numbers` is like saying "the memory starting at numbers[0]".

49.2 The Array Name as a Pointer

You can use the array name as a pointer to its first element:

```
printf("%p\n", numbers);      // address of first element
printf("%p\n", &numbers[0]);  // same address
```

They both print the same location in memory.

That's why this works:

```
*numbers = 42; // change first element
```

Here, `*numbers` means "the value at the address of numbers[0]".

Now `numbers[0]` becomes `42`.

49.3 Using Pointer Arithmetic

C lets you move through memory with pointer arithmetic. When you add `1` to a pointer, it moves to the next element, not just the next byte.

```
int *ptr = numbers;  // pointer to first element

printf("%d\n", *ptr);     // 10
printf("%d\n", *(ptr+1)); // 20
printf("%d\n", *(ptr+2)); // 30
```

Notice the pattern:

- `*(ptr + 0)` is the first element
- `*(ptr + 1)` is the second
- `*(ptr + 2)` is the third

That's exactly what `numbers[0]`, `numbers[1]`, `numbers[2]` do.

So in C, these are equivalent:

```
numbers[i] == *(numbers + i)
```

49.4 But Arrays Are Not Pointers

Even though `numbers` can act like a pointer, it's not truly one. A pointer is a variable that stores an address, you can reassign it. An array name is fixed, it's a label for a memory block.

```
int numbers[3] = {1, 2, 3};
int *ptr = numbers;

ptr = ptr + 1;    // OK
numbers = numbers + 1; //   Error: array name is not assignable
```

Arrays and pointers are *related*, but not the same kind of variable.

49.5 Arrays Decay to Pointers

When you pass an array to a function, it decays into a pointer — only the address of the first element is passed.

```
void print_first(int arr[]) {
    printf("%d\n", arr[0]);
}
```

Here, `arr` is really a pointer, not a full array copy. That's why functions can see changes to arrays, they both point to the same data.

49.6 Visual Summary

Concept	Description	Example
Array	Fixed block of memory	`int a[5];`
Pointer	Variable holding an address	`int *p;`
Array name	Acts like pointer to first element	`a == &a[0]`
Difference	Arrays can't be reassigned	`p = a;  a = p;`

49.7 Arrays and Strings

This connection explains why strings (arrays of `char`) work with pointers too:

```
char word[] = "Hi";
char *ptr = word;

printf("%c\n", *ptr);       // 'H'
printf("%c\n", *(ptr + 1)); // 'i'
```

You'll use this idea often when working with string functions, most take `char *`.

Tiny Code

```
#include <stdio.h>

int main(void) {
    int numbers[3] = {10, 20, 30};
    int *p = numbers; // points to first element

    printf("First: %d\n", *p);
    printf("Second: %d\n", *(p + 1));
    printf("Third: %d\n", *(p + 2));

    *(p + 1) = 99; // change second element
    printf("Updated second: %d\n", numbers[1]);

    return 0;
}
```

Output:

```
First: 10
Second: 20
Third: 30
Updated second: 99
```

Why It Matters

Understanding how arrays and pointers connect is one of the big leaps in C. It helps you write functions that handle arrays, work with strings, and explore memory safely.

Once you see arrays as "blocks" and pointers as "addresses," C starts to feel clear and logical.

Try It Yourself

1. Create an array and print the address of each element with `&array[i]`.
2. Use a pointer to walk through the array with `*(ptr + i)`.
3. Try changing an element through the pointer.
4. Pass the array to a function and print its elements.
5. Experiment with strings, print each character using pointer arithmetic.

Arrays are like houses; pointers are like maps. Learn how they line up, and you'll never get lost in memory again.

50. Common Array Pitfalls

Arrays are simple once you understand them, but they also come with a few easy-to-miss traps. Because C gives you a lot of freedom, it also expects you to be careful. If you know the most common mistakes, you can avoid hours of debugging and strange behavior.

Let's walk through the biggest pitfalls one by one.

50.1 Off-by-One Errors

The most common mistake is going one step too far. Remember, array indexes start at `0` and end at `size - 1`.

If your array has 5 elements, the valid indexes are `0, 1, 2, 3, 4`. Accessing `array[5]` is out of bounds.

Example:

```
int nums[5] = {1, 2, 3, 4, 5};
for (int i = 0; i <= 5; i++) { //  runs one too far
    printf("%d\n", nums[i]);
}
```

This tries to print `nums[5]`, which doesn't exist, and C won't warn you. It may print garbage or crash your program.

Fix: use `< size`, not `<= size - 1`

```
for (int i = 0; i < 5; i++) { ... }
```

50.2 Forgetting Array Bounds

C does not check array boundaries. If you write outside the valid range, you're overwriting memory that belongs to something else.

This is called undefined behavior. Your program might seem fine, then suddenly fail later for no clear reason.

Always use indexes you can trust, often from a loop counter or `sizeof` calculation.

```
int length = sizeof(nums) / sizeof(nums[0]);
```

50.3 Forgetting the Null Terminator

When working with strings, every array must end with `'\0'`. If you forget it, C doesn't know where the string stops.

Bad:

```
char name[3] = {'C', 'a', 't'}; //  no '\0'
printf("%s", name); // unpredictable output
```

Good:

```
char name[4] = {'C', 'a', 't', '\0'};
```

If you use a string literal (`"Cat"`), C adds `'\0'` automatically.

50.4 Mixing Up Size and Count

An array's size is the number of slots, but you may only use part of it.

Example:

```c
int scores[10];
int count = 3;
```

Only `scores[0]`, `scores[1]`, and `scores[2]` contain real data. You can't safely loop through all 10 unless you know the rest are valid.

Fix: always track how many elements are actually used.

50.5 Forgetting Arrays Are Fixed Size

You can't resize a normal array after it's created. This fails:

```c
int nums[3] = {1, 2, 3};
nums[3] = 4; //  can't extend
```

If you need more space, use dynamic memory (`malloc`), you'll learn that soon.

50.6 Passing Wrong Size to Functions

When you pass an array to a function, it becomes a pointer. That means the function can't know the size automatically.

Example:

```c
void print_array(int arr[]) {
    for (int i = 0; i < ???; i++) //  no size info
        printf("%d ", arr[i]);
}
```

Fix: always pass the size along:

```c
void print_array(int arr[], int size);
```

Then call:

```c
print_array(nums, 5);
```

50.7 Using Uninitialized Elements

When you create an array without setting values, the elements contain garbage, random memory leftovers.

```
int data[5];
printf("%d\n", data[0]); //  unpredictable
```

Fix: initialize your arrays:

```
int data[5] = {0}; // fills all with 0
```

50.8 Confusing Arrays with Pointers

Arrays and pointers are related but not identical. You can use pointer arithmetic, but you can't reassign an array name.

```
int nums[3] = {1, 2, 3};
int *p = nums;
p = p + 1;      //  moves pointer
nums = nums + 1; //  not allowed
```

Keep in mind: arrays are fixed blocks, not movable variables.

50.9 Wrong `sizeof` Usage

`sizeof(array)` gives total bytes of the whole array, but `sizeof(pointer)` gives only the pointer size (usually 8).

So inside functions, `sizeof(arr)` no longer works, you'll just get pointer size. Use it only in the same scope where the array is defined.

Outside functions:

```
int len = sizeof(arr) / sizeof(arr[0]);
```

Inside functions:

```
// not reliable, arr is now a pointer
```

Tiny Code

```c
#include <stdio.h>

void print_array(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main(void) {
    int data[5] = {10, 20, 30, 40, 50};

    print_array(data, 5);

    // data[5] = 60; //  out of bounds, don't do this

    return 0;
}
```

Output:

```
10 20 30 40 50
```

Why It Matters

Arrays are fast and powerful, but they trust you completely. C won't stop you from stepping out of bounds or using bad data. Once you understand the common pitfalls, you'll write safer, more reliable code.

Try It Yourself

1. Write a loop that accidentally goes one index too far, see what happens.
2. Create a string manually, try forgetting the null terminator.
3. Initialize an array partially and print all elements.
4. Pass an array to a function along with its size.
5. Compare `sizeof(arr)` in main vs inside a function.

Learning these lessons early means you'll spend more time coding with confidence and less time chasing mysterious bugs.

Chapter 6. Pointers and Memory

51. What Is a Pointer

Up until now, every variable you've used has stored a value, a number, a letter, or a piece of text. A pointer is different. Instead of holding a value, it holds the address of a value, where that value lives in memory.

You can think of it like this:

- A normal variable is the house (it contains the value).
- A pointer is the address written on a piece of paper (it tells you where the house is).

Once you understand pointers, you'll unlock some of C's most powerful abilities: working with arrays, strings, and dynamic memory.

51.1 Memory and Addresses

Every variable in your program lives somewhere in memory. That location is called its address, it's like a street number that identifies the exact spot.

You can see a variable's address using the address-of operator `&`:

```
int age = 21;
printf("%p\n", &age);
```

`%p` prints a memory address. You might see something like:

```
0x7ffee3b3a6c4
```

That's the address where `age` is stored.

51.2 Declaring a Pointer

A pointer is a variable designed to store an address. You declare it with a `*` after the type:

```
int *p;
```

This means: "`p` is a pointer to an int."

You can then assign it the address of an integer:

```c
int age = 21;
int *p = &age;  // p now stores the address of age
```

So `p` doesn't hold `21`, it holds something like `0x7ffee3b3a6c4`.

51.3 Dereferencing a Pointer

If `p` holds an address, how do you get the value stored there? You use the dereference operator `*`:

```c
printf("%d\n", *p);
```

`*p` means "go to the address stored in `p` and read the value there." So this prints `21`.

Now you have two ways to reach the same value:

- `age` → directly
- `*p` → indirectly through the pointer

51.4 Visualizing It

Let's see it in memory:

```
age = 21
p   = &age
*p  = 21
```

Or in a picture:

```
   +------+      +----+
age|  21  |      |  p |----> [address of age]
   +------+      +----+
```

The arrow shows that `p` "points" to `age`.

51.5 Using Pointers to Change Values

Because `*p` is another way of accessing the same memory, you can change the original value through the pointer:

```
*p = 30;
printf("%d\n", age); // prints 30
```

You didn't touch `age` directly, you modified it through its pointer.

This is why pointers are powerful: they let you work with memory itself, not just copies of values.

51.6 Pointer Types Must Match

A pointer's type must match the value it points to. You can't store the address of an `int` in a `char *` pointer.

Correct:

```
int *p;
int n = 5;
p = &n;
```

Wrong:

```
char *p;
int n = 5;
p = &n; // incompatible types
```

The type tells C how to interpret the memory.

51.7 Null Pointers

A pointer that doesn't point anywhere should be set to NULL. This is a safe "empty" value that says "I'm not pointing at anything."

```
int *p = NULL;
```

You can check before using it:

```
if (p != NULL) {
    printf("%d\n", *p);
}
```

Never try to use `*p` if `p` is NULL, it will crash your program.

51.8 Summary

Concept	Description	Example
`&`	Address-of operator	`p = &age;`
`*`	Dereference operator	`value = *p;`
Pointer	Stores address of a value	`int *p;`
`NULL`	Pointer to nowhere	`int *p = NULL;`

Tiny Code

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    int *ptr = &x;

    printf("x = %d\n", x);
    printf("Address of x = %p\n", &x);
    printf("ptr points to = %p\n", ptr);
    printf("Value at ptr = %d\n", *ptr);

    *ptr = 20;
    printf("x after change = %d\n", x);

    return 0;
}
```

Output:

```
x = 10
Address of x = 0x7ffee3b3a6c4
ptr points to = 0x7ffee3b3a6c4
Value at ptr = 10
x after change = 20
```

Why It Matters

Pointers let you share memory between parts of your program. They're essential for arrays, strings, functions, and dynamic memory. Once you see pointers as "addresses to values," they stop being scary and start feeling useful.

Try It Yourself

1. Create an `int` variable and a pointer to it.
2. Print both the value and address.
3. Change the value using the pointer.
4. Set a pointer to `NULL` and check before dereferencing.
5. Add another pointer pointing to the same variable and print through both.

Pointers are your key to understanding how C truly works under the hood. You're now ready to explore how they interact with operators in the next section!

52. The Address-of (&) and Dereference (*) Operators

Pointers come alive through two special symbols:

- `&` gets the address of a variable (where it lives in memory)
- `*` gets the value stored at that address

Together, they let you move easily between values and their memory locations, like having a GPS to find a house and a key to open its door.

Let's walk through them step by step.

52.1 The Address-of Operator `&`

The `&` symbol means "give me the address of this variable."

```
int age = 25;
printf("%p\n", &age);
```

Output:

```
0x7ffeef3c6b24
```

That number is where `age` is stored in memory.

You can store that address inside a pointer:

```
int *p = &age;
```

Now `p` points to `age`.

52.2 The Dereference Operator *

If `p` holds an address, then `*p` means "go to that address and get the value there."

```
int age = 25;
int *p = &age;

printf("%d\n", *p); // prints 25
```

The `*` in front of a pointer means "use the thing it points to."

You can also change the value stored there:

```
*p = 30; // updates age
printf("%d\n", age); // prints 30
```

You didn't change `age` directly, you updated it through its pointer.

52.3 Two Meanings of *

The `*` symbol appears in two contexts:

1. In a declaration, it defines a pointer:

   ```
   int *p;
   ```

   "`p` is a pointer to int."

2. In an expression, it accesses the pointed value:

   ```
   *p = 5;
   ```

   "Set the value stored at the address inside `p`."

You'll learn to tell them apart from context.

52.4 Combining `&` and `*`

`&` and `*` are opposites, one finds an address, the other follows it.

```
int x = 42;
int *p = &x;

printf("%d\n", *(&x)); // same as x
printf("%p\n", &(*p)); // same as p
```

Think of them like this:

- `&` → "take the address of"
- `*` → "follow the address"

Used together, they cancel out.

52.5 Example in Action

```
#include <stdio.h>

int main(void) {
    int value = 10;
    int *ptr = &value; // store address

    printf("Value: %d\n", value);
    printf("Address of value: %p\n", &value);
    printf("Pointer holds: %p\n", ptr);
    printf("Value at pointer: %d\n", *ptr);

    *ptr = 50;
    printf("New value: %d\n", value);

    return 0;
}
```

Output:

```
Value: 10
Address of value: 0x7ffee3b3a6c4
```

```
Pointer holds: 0x7ffee3b3a6c4
Value at pointer: 10
New value: 50
```

52.6 Visual Picture

```
+------------+       +------------+
|  variable  |       |   pointer  |
|   value=10 |<------| address=&variable |
+------------+       +------------+
```

- `&variable` gives you the arrow
- `*pointer` lets you follow it back

52.7 Common Mistakes

1. Forgetting `&` when assigning to a pointer

```c
int *p;
int n = 5;
p = n;     //  Wrong: n is not an address
p = &n;    //  Correct: &n is the address of n
```

2. Forgetting `*` when printing the value

```c
int n = 5;
int *p = &n;

printf("%d", p);   //  Wrong: prints address
printf("%d", *p);  //  Correct: prints value
```

3. Dereferencing an uninitialized pointer

```c
int *p;
*p = 5;   //  Wrong: p doesn't point anywhere yet
```

Always assign a valid address first or set it to `NULL`:

```c
int *p = NULL;
```

52.8 Step-by-Step Thought Process

When you see a pointer, ask yourself:

1. What type of value does it point to?
2. Does it currently hold a valid address?
3. Am I trying to access the address (`&`) or the value (`*`)?

These questions make pointers clear and safe to use.

Tiny Code

```c
#include <stdio.h>

int main(void) {
    int num = 100;
    int *p = &num;

    printf("num = %d\n", num);
    printf("&num = %p\n", &num);
    printf("p = %p\n", p);
    printf("*p = %d\n", *p);

    return 0;
}
```

Output:

```
num = 100
&num = 0x7ffee3b3a6c4
p = 0x7ffee3b3a6c4
*p = 100
```

Why It Matters

These two operators are the gateway to working with memory in C. They allow you to share data between functions, build complex data structures, and control memory directly. Understanding them now makes arrays, strings, and dynamic allocation much easier later.

Try It Yourself

1. Declare a variable and a pointer to it. Print both address and value.
2. Change the variable through the pointer.
3. Use `*(&variable)`, see it's the same as `variable`.
4. Print a pointer with and without `*`, note the difference.
5. Set a pointer to `NULL` and check before dereferencing.

Master `&` and `*`, and you'll have full control over how your data lives and moves inside memory.

53. Pointer Arithmetic

Pointers can do more than just store addresses, you can also move them. Because pointers represent memory locations, adding or subtracting values lets you step through memory. This feature is called pointer arithmetic, and it's especially useful when working with arrays.

53.1 Moving Through Memory

When you add `1` to a pointer, it moves forward by the size of the type it points to. It doesn't move one byte, it moves one element.

For example:

```
int *p;
p + 1; // moves by sizeof(int)
```

If `sizeof(int)` is 4 bytes, then `p + 1` advances 4 bytes in memory. So:

- `p + 1` moves to the next element
- `p + 2` moves two elements ahead
- `p - 1` moves one element back

53.2 Example with an Array

```
#include <stdio.h>

int main(void) {
    int numbers[3] = {10, 20, 30};
    int *p = numbers; // points to first element
```

```
    printf("First: %d\n", *p);
    printf("Second: %d\n", *(p + 1));
    printf("Third: %d\n", *(p + 2));

    return 0;
}
```

Each step moves to the next element in the array because arrays occupy contiguous memory.

53.3 Using Pointers Like Indexes

These two expressions are equivalent:

```
numbers[i]
*(numbers + i)
```

That means you can loop with either an index or a pointer:

```
for (int i = 0; i < 3; i++) {
    printf("%d ", *(p + i));
}
```

Or move the pointer directly:

```
for (int *ptr = numbers; ptr < numbers + 3; ptr++) {
    printf("%d ", *ptr);
}
```

Both print:

```
10 20 30
```

53.4 Subtracting Pointers

You can find the distance between two pointers that point into the same array:

```
int *start = &numbers[0];
int *end = &numbers[2];
printf("%ld\n", end - start); // prints 2
```

The result is the number of elements between them, not the number of bytes.

53.5 Staying Within Bounds

You can move a pointer inside an array, or one past the last element, but never before the first or beyond the end.

Incorrect:

```
int arr[3] = {1, 2, 3};
int *p = arr + 4; // out of range
```

Only pointers that refer to positions within the same array are valid for arithmetic.

53.6 Different Step Sizes for Different Types

The amount a pointer moves depends on its type:

```
int *pi;
char *pc;
double *pd;
```

If you add 1:

- `pi + 1` moves 4 bytes (size of `int`)
- `pc + 1` moves 1 byte (size of `char`)
- `pd + 1` moves 8 bytes (size of `double`)

C adjusts the step automatically based on the type.

53.7 Increment and Decrement

You can also use `++` and `--` on pointers:

```
int data[3] = {5, 10, 15};
int *p = data;

printf("%d\n", *p); // 5
p++;
printf("%d\n", *p); // 10
p++;
printf("%d\n", *p); // 15
```

Each increment moves one element forward.

53.8 Using Pointers in Loops

You can iterate through an array using pointer comparisons:

```
int *p = data;
int *end = data + 3;

while (p < end) {
    printf("%d ", *p);
    p++;
}
```

This pattern works well when you know where the array starts and ends.

53.9 Tiny Code Example

```
#include <stdio.h>

int main(void) {
    int arr[4] = {2, 4, 6, 8};
    int *p = arr;

    printf("Using pointer arithmetic:\n");
    for (int i = 0; i < 4; i++) {
        printf("%d ", *(p + i));
    }

    printf("\nUsing pointer increment:\n");
    for (int *q = arr; q < arr + 4; q++) {
        printf("%d ", *q);
    }

    printf("\n");
    return 0;
}
```

Output:

```
Using pointer arithmetic:
2 4 6 8
Using pointer increment:
2 4 6 8
```

53.10 Common Mistakes

1. Forgetting pointer step size

```
int *p = arr;
p = p + 1; // moves one int ahead (not one byte)
```

2. Going out of bounds

```
int *p = arr + 5; // invalid memory access
```

3. Mixing pointer types

```
char *p = (char *)arr; // reinterprets memory, risky
```

Pointer arithmetic should only be used on pointers of the correct type and within valid bounds.

Why It Matters

Pointer arithmetic gives you fine-grained control over how you move through arrays. It's what makes C powerful for working close to the hardware, you can navigate memory directly, one element at a time.

Try It Yourself

1. Create an array and print elements using `*(p + i)`.
2. Use `p++` in a loop to step through the array.
3. Compute the difference between two pointers.
4. Test pointer arithmetic with `char`, `int`, and `double` arrays.
5. Practice writing loops that use pointers instead of indexes.

Pointer arithmetic takes practice, but once it clicks, you'll see how naturally it fits with arrays and memory in C.

54. Arrays and Pointers Revisited

You've already seen that arrays and pointers are closely related, now it's time to connect the dots clearly. This section ties together what you've learned about both topics so far, showing when they behave the same, when they don't, and how to use them safely and confidently.

54.1 The Array Name Is (Almost) a Pointer

When you declare an array like this:

```
int numbers[3] = {10, 20, 30};
```

the name `numbers` represents the address of its first element. That means these two expressions are equivalent:

```
numbers == &numbers[0]
```

You can even assign that address to a pointer:

```
int *p = numbers; // same as &numbers[0]
```

Now `p` points to the same place as `numbers`.

54.2 Accessing Elements

You can use either array notation or pointer arithmetic, both work the same:

```
printf("%d\n", numbers[0]); // array style
printf("%d\n", *numbers);   // pointer style
```

Similarly:

```
numbers[2] == *(numbers + 2)
```

So in many cases, arrays and pointers are interchangeable in expressions.

54.3 Arrays in Memory

Arrays are stored in contiguous memory, one element after another.

Visualize `int numbers[3] = {10, 20, 30};` like this:

```
Address:  0x100  0x104  0x108
Values:    10     20     30
Indexes:    0      1      2
```

If you start with `numbers`,

- `numbers + 1` is the address of `numbers[1]`,
- `numbers + 2` is the address of `numbers[2]`.

C automatically adjusts by `sizeof(int)` for each step.

54.4 Pointers Are More Flexible

A pointer can move. You can make it point anywhere:

```
int *p = numbers;
p = p + 1; // now points to numbers[1]
```

But an array name is fixed, you cannot change it:

```
int *p = numbers;
p = p + 1;     // allowed
numbers = numbers + 1; // error: array name is not assignable
```

So think of the array name as a constant pointer.

54.5 Passing Arrays to Functions

When you pass an array to a function, it decays into a pointer. That means only the address of the first element is passed, not a full copy.

Example:

```
void print_array(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main(void) {
    int data[3] = {1, 2, 3};
    print_array(data, 3);
    return 0;
}
```

Inside `print_array`, `arr` is actually a pointer to the first element of `data`.

That's why you must also pass the size, C has no way to tell how big the array is from just the pointer.

54.6 Modifying Arrays Through Pointers

Because arrays and pointers refer to the same memory, you can modify an array's elements through a pointer:

```
int numbers[3] = {1, 2, 3};
int *p = numbers;

*p = 10;          // changes numbers[0]
*(p + 1) = 20;    // changes numbers[1]
*(p + 2) = 30;    // changes numbers[2]
```

Afterward, `numbers` becomes `{10, 20, 30}`.

54.7 Arrays of Different Types

Pointers must match the type of the array they point to:

```
int nums[3];
int *p = nums;   // correct
char *c = nums;  // wrong: type mismatch
```

Each pointer type understands its element size. A `char *` moves one byte at a time, an `int *` moves `sizeof(int)` bytes at a time.

54.8 Summary of the Relationship

Concept	Array	Pointer
Memory	Fixed block of elements	Can point anywhere
Name	Constant (cannot be reassigned)	Variable (can move)
Access	`arr[i]`	`*(p + i)`
Pass to function	Decays to pointer	Passed by value
Size info	Known at compile time	Must be tracked manually

Remember: arrays and pointers often look similar, but arrays are blocks, while pointers are labels that can move.

54.9 Tiny Code Example

```c
#include <stdio.h>

void print_elements(int *p, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", *(p + i));
    }
    printf("\n");
}

int main(void) {
    int numbers[3] = {10, 20, 30};
    int *p = numbers;

    printf("Using array indexing: ");
    for (int i = 0; i < 3; i++) {
        printf("%d ", numbers[i]);
    }

    printf("\nUsing pointer arithmetic: ");
    print_elements(p, 3);

    return 0;
}
```

Output:

```
Using array indexing: 10 20 30
Using pointer arithmetic: 10 20 30
```

Why It Matters

Understanding how arrays and pointers connect is essential in C. It explains how functions receive data, how strings work, and how memory is laid out. Once you see that `array[i]` and `*(array + i)` are two views of the same thing, you'll move smoothly between the two worlds.

Try It Yourself

1. Create an array and a pointer to it. Print each element using both syntax styles.
2. Move the pointer using `p++` and print the values.
3. Pass the array to a function and print all elements inside.
4. Try to reassign the array name (you'll see a compiler error).
5. Compare `array[i]` and `*(array + i)` and verify they're identical.

Once this connection clicks, C starts to feel much simpler, you'll see arrays and pointers as two faces of the same idea.

55. Function Parameters with Pointers

When you pass a variable to a function in C, the function usually receives a copy. That means any changes inside the function don't affect the original.

But what if you *want* the function to change something directly, like updating a value, filling an array, or swapping two numbers? That's where pointers come in.

By passing a pointer instead of a copy, you let the function work with the original data.

55.1 Passing by Value (the Default)

Normally, C passes by value. The function gets its own copy of the data.

```c
#include <stdio.h>

void add_one(int n) {
    n = n + 1;
}

int main(void) {
    int x = 5;
    add_one(x);
    printf("%d\n", x); // still 5
    return 0;
}
```

Even though `n` becomes `6` inside `add_one`, `x` outside stays `5` because `n` is a separate copy.

55.2 Passing by Pointer

If you pass a pointer, the function can follow it back to the original variable.

```c
#include <stdio.h>

void add_one(int *p) {
    *p = *p + 1;
}

int main(void) {
    int x = 5;
    add_one(&x);
    printf("%d\n", x); // now 6
    return 0;
}
```

Here's what's happening:

- `&x` gives the address of `x`
- `p` receives that address
- `*p` means "go to the original x and change it"

Now the function can update the real variable, not a copy.

55.3 Visualizing the Difference

Pass by value:

```
main: x = 5
add_one: n = 5 (copy)
```

Pass by pointer:

```
main: x = 5
add_one: p -> x (same variable)
```

When you use a pointer, you're handing the function a reference to the same memory location.

55.4 Example: Swapping Two Values

Let's write a simple `swap` function.

Incorrect version (by value):

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Call it like this:

```
int x = 10, y = 20;
swap(x, y);
```

After calling, `x` is still `10` and `y` is still `20`, the swap happened only inside the function.

Correct version (by pointer):

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Call it with addresses:

```
int x = 10, y = 20;
swap(&x, &y);
```

Now the original `x` and `y` are swapped.

55.5 Using Pointers with Arrays

When you pass an array to a function, it decays to a pointer automatically. That means the function already has access to the original memory.

Example:

```c
void fill_with_zeros(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = 0; // modifies original array
    }
}
```

Call it like this:

```c
int data[3] = {1, 2, 3};
fill_with_zeros(data, 3);
// now data = {0, 0, 0}
```

No need to use `&` when passing arrays, they're already pointers.

55.6 Pointers Make Functions More Powerful

Using pointers, functions can:

- Modify variables in the caller
- Fill or update arrays
- Return multiple results (by updating several variables)

This is how standard library functions like `scanf` work:

```c
int x;
scanf("%d", &x); // gives scanf the address of x
```

`scanf` follows the pointer and stores the input directly into `x`.

55.7 Checking for NULL

Sometimes a function might receive a pointer that doesn't point anywhere. It's a good habit to check for `NULL` before using it:

```c
void safe_add_one(int *p) {
    if (p != NULL) {
        *p = *p + 1;
    }
}
```

Never dereference (`*p`) a pointer unless you're sure it's valid.

55.8 Summary

Concept	Pass by Value	Pass by Pointer
What's passed	Copy of value	Address of value
Can modify original?	No	Yes
Used with arrays?	Yes (decays to pointer)	Yes
Needs `*` and `&`?	No	Yes

55.9 Tiny Code Example

```c
#include <stdio.h>

void double_value(int *p) {
    *p = *p * 2;
}

int main(void) {
    int num = 7;
    double_value(&num);
    printf("Doubled: %d\n", num);
    return 0;
}
```

Output:

```
Doubled: 14
```

Why It Matters

Passing by pointer is one of the most important skills in C. It lets you share data between functions without returning it. Once you get comfortable with `*` and `&`, you'll be able to write more flexible and efficient programs.

Try It Yourself

1. Write a function `reset(int *p)` that sets a variable to zero.
2. Write a function `square(int *p)` that replaces a number with its square.
3. Create a function `fill_array(int *arr, int size, int value)` that fills an array.

4. Modify a string inside a function using `char *`.
5. Try removing `&` or `*` in your calls and see what happens, watch how the behavior changes.

Once you understand how to pass by pointer, you'll have a solid foundation for building real programs that modify data directly.

56. Dynamic Memory Allocation with `malloc`

So far, every variable and array you've used had a fixed size, decided at compile time. But what if you don't know how much data you'll need until the program is running? For example, maybe you're reading user input, loading a file, or building a list that grows.

In those cases, you can use dynamic memory allocation, asking the computer for memory *while your program runs*. In C, this is done with the function `malloc`.

56.1 Static vs Dynamic Memory

Let's compare the two:

Type	When Decided	Example	Lifespan
Static	At compile time	`int arr[10];`	Freed automatically
Dynamic	At runtime	`int *p = malloc(...);`	You must free it

With `malloc`, you decide how much memory to get while your program runs. You can make arrays of any size, based on user input or other conditions.

56.2 Including `<stdlib.h>`

All memory management functions live in `<stdlib.h>`, so always include it at the top:

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

56.3 Using `malloc`

`malloc` stands for memory allocation. It takes the number of bytes you want and returns a pointer to the new memory block.

```
void *malloc(size_t size);
```

Example:

```
int *p = malloc(5 * sizeof(int));
```

This requests enough space for 5 integers. `malloc` returns the address of the first byte, we store it in an `int *` because that's the type of data we'll store.

56.4 Checking the Return Value

If `malloc` fails (for example, not enough memory), it returns `NULL`. Always check before using the pointer:

```
int *p = malloc(5 * sizeof(int));
if (p == NULL) {
    printf("Memory allocation failed\n");
    return 1;
}
```

If it's not `NULL`, the memory is ready to use.

56.5 Using the Allocated Memory

Once you have memory, use it like an array:

```
for (int i = 0; i < 5; i++) {
    p[i] = i * 10;
}
```

Or with pointer arithmetic:

```
*(p + 2) = 25;
```

56.6 Example: Create and Print an Array

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

int main(void) {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int *arr = malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    printf("You created an array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr); // free memory when done
    return 0;
}
```

Sample run:

```
Enter number of elements: 5
You created an array:
1 2 3 4 5
```

56.7 Why Use `sizeof`

Never assume the size of a type, it can vary between systems. Always use `sizeof` when allocating:

```c
int *p = malloc(n * sizeof(int));
double *d = malloc(n * sizeof(double));
```

This makes your code portable and safe.

56.8 The Memory You Get

`malloc` doesn't set the contents to zero, it gives you uninitialized memory. The values inside are unpredictable until you assign them.

If you want memory filled with zeros, use `calloc` instead (you'll learn this soon).

56.9 Don't Forget to Free Memory

Any memory you get from `malloc` must be released using `free()` once you're done. Otherwise, your program will leak memory (use it but never return it).

```
free(p);
```

After freeing, you can set the pointer to `NULL` to avoid accidental reuse:

```
free(p);
p = NULL;
```

56.10 Common Mistakes

1. Forgetting to free: Allocating repeatedly without freeing causes memory leaks.
2. Using memory after freeing:

```
free(p);
*p = 10; // invalid - memory no longer yours
```

3. Forgetting `sizeof`:

```
int *p = malloc(10); // allocates only 10 bytes, not 10 ints
```

Always multiply by `sizeof(type)`.

Tiny Code Example

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

int main(void) {
    int *data = malloc(3 * sizeof(int));
    if (data == NULL) {
        printf("Allocation failed\n");
        return 1;
    }

    data[0] = 10;
    data[1] = 20;
    data[2] = 30;

    for (int i = 0; i < 3; i++) {
        printf("%d ", data[i]);
    }
    printf("\n");

    free(data);
    data = NULL;
    return 0;
}
```

Output:

```
10 20 30
```

Why It Matters

Dynamic memory lets your programs adapt to real-world data. You're no longer limited to fixed-size arrays, you can create exactly as much space as you need, when you need it.

It's one of the most powerful features of C, but also one that demands responsibility, you manage the memory yourself.

Try It Yourself

1. Ask the user for a number, allocate an array of that size, fill it, and print it.
2. Create an array of `double` using `malloc`.

3. Practice checking for `NULL` before using the memory.
4. Try forgetting `free()` and run your program multiple times, see how memory usage changes.
5. Combine `malloc` and `free` in a loop to allocate and release memory safely.

Once you master `malloc`, your programs become flexible, ready to handle data of any size.

57. Using `free` Safely

Whenever you use `malloc` (or any function that allocates memory), you're borrowing space from the computer's memory. But borrowed memory must be returned. If you don't, your program will keep holding on to memory it no longer needs, this is called a memory leak.

The tool for returning memory in C is the function `free()`.

57.1 What `free` Does

The `free()` function releases a block of memory that you previously allocated with `malloc`, `calloc`, or `realloc`.

```
#include <stdlib.h>

free(pointer);
```

After calling `free`, the memory is returned to the system for reuse. You can't access or use that memory anymore, it no longer belongs to your program.

57.2 Basic Example

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

int main(void) {
    int *p = malloc(3 * sizeof(int));
    if (p == NULL) {
        printf("Allocation failed\n");
        return 1;
    }

    p[0] = 10;
```

```
    p[1] = 20;
    p[2] = 30;

    for (int i = 0; i < 3; i++) {
        printf("%d ", p[i]);
    }
    printf("\n");

    free(p); // release the memory
    return 0;
}
```

Output:

```
10 20 30
```

After `free(p);`, the memory used by `p` is no longer yours. Accessing it again would be undefined behavior.

57.3 Why You Must Free Memory

When your program runs, it asks the operating system for memory. If you keep allocating but never freeing, memory usage keeps growing, and eventually, your program might slow down or crash.

This problem is called a memory leak.

Example of a leak:

```
int *p = malloc(100 * sizeof(int));
// forgot to call free(p)
```

Each time you run this, your program uses a little more memory that never gets released.

57.4 Safe Freeing Practices

Here are good habits to make `free` safe and easy:

1. Always pair every malloc with a free. If you allocate memory, you should free it once you're done.

2. Free only once. Calling `free()` twice on the same pointer is undefined behavior. It may crash or corrupt memory.
3. Never use freed memory. Once freed, don't read or write through that pointer.
4. Set the pointer to NULL after freeing. That way, you can safely check before using it again.

Example:

```
free(p);
p = NULL;
```

Now, if you accidentally try to use it:

```
if (p != NULL) {
    *p = 42; // this won't run, because p is NULL
}
```

57.5 Common Mistakes

1. Forgetting to free:

```
int *data = malloc(10 * sizeof(int));
// No free, memory leak
```

2. Freeing twice:

```
free(data);
free(data); // invalid, already freed
```

3. Using after free:

```
free(data);
data[0] = 5; // invalid, memory no longer valid
```

4. Freeing unallocated memory:

```
int a;
free(&a); // invalid, 'a' wasn't allocated with malloc
```

These mistakes can cause subtle, hard-to-find bugs. Good pointer hygiene, clear naming, proper freeing, and setting to `NULL`, prevents them.

57.6 Freeing Arrays and Multiple Pointers

If you allocate multiple pointers, each must be freed separately:

```
int *a = malloc(5 * sizeof(int));
double *b = malloc(3 * sizeof(double));

free(a);
free(b);
```

You only need to call `free()` once per allocation.

If you allocate an array of pointers (like a list of strings), you must free each element and then the array itself, you'll learn that pattern later.

57.7 Visualizing

Before freeing:

```
p   > [10][20][30]
```

After freeing:

```
p   > (invalid memory)
```

Setting to `NULL` helps:

```
p = NULL
```

Now you can easily check:

```
if (p == NULL) {
    printf("Pointer is safe.\n");
}
```

Tiny Code Example

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

int main(void) {
    int *nums = malloc(4 * sizeof(int));
    if (nums == NULL) return 1;

    for (int i = 0; i < 4; i++) nums[i] = i + 1;

    printf("Array: ");
    for (int i = 0; i < 4; i++) printf("%d ", nums[i]);
    printf("\n");

    free(nums);
    nums = NULL; // reset pointer

    if (nums == NULL) {
        printf("Memory freed safely.\n");
    }

    return 0;
}
```

Output:

```
Array: 1 2 3 4
Memory freed safely.
```

Why It Matters

Memory in C is manual, you're in charge of asking for it *and* giving it back. By using `free()` carefully, your programs stay efficient, stable, and leak-free.

It's like borrowing library books, always return what you take.

Try It Yourself

1. Allocate an array of integers with `malloc`, fill it, print it, then free it.
2. Forget to call `free`, run the program multiple times and watch memory usage.
3. Free a pointer and then set it to `NULL`. Try accessing it again and check.

4. Try freeing twice, see how your program behaves (expect a crash or warning).
5. Make freeing part of your habit, always pair allocation and cleanup.

Once you build the habit of freeing memory properly, you'll write reliable programs that can run for hours without leaks or crashes.

58. Pointer to Pointer

By now, you've learned how a pointer stores the address of a variable. But did you know you can also have a pointer that stores the address of another pointer? That's called a pointer to pointer, and it might sound confusing at first, but it's just one more level of indirection.

Let's break it down step by step.

58.1 One Step Back: A Simple Pointer

When you write this:

```
int x = 10;
int *p = &x;
```

- `x` stores the value `10`
- `p` stores the address of `x`

We can draw it like this:

```
x: 10
p: &x
```

So:

- `x` → value (10)
- `p` → points to x
- `*p` → value at x (10)

58.2 Adding Another Layer

Now, let's add a pointer to that pointer:

```
int x = 10;
int *p = &x;
int pp = &p;
```

Here's what's happening:

- `pp` stores the address of `p`
- `*pp` is the value stored in `p` (which is `&x`)
- `pp` is the value stored in `x` (which is `10`)

So:

```
x = 10
p = &x
pp = &p
```

This chain means `pp` = 10.

58.3 Visualizing It

```
pp  > p  > x
            ↑
            10
```

Think of it as a chain of arrows:

- `pp` points to `p`
- `p` points to `x`
- `x` holds 10

Each `*` you use follows one arrow.

58.4 Accessing Values

Using the example above:

```c
printf("%d\n", x);     // prints 10
printf("%d\n", *p);    // prints 10
printf("%d\n", pp);  // prints 10
```

They all get the same value, you're just reaching it through different levels of pointers.

58.5 Why Use a Pointer to Pointer?

Pointer-to-pointer variables show up often in C, especially when:

1. You need to modify a pointer inside a function
2. You work with arrays of strings (`char argv`)
3. You manage dynamic memory, like a list of lists or a 2D array

Example 1: modifying a pointer inside a function

```c
void allocate_memory(int ptr) {
    *ptr = malloc(sizeof(int));
    ptr = 42;
}
```

Call it like this:

```c
int *p = NULL;
allocate_memory(&p);
printf("%d\n", *p); // prints 42
free(p);
```

The function receives the address of the pointer, so it can update it directly.

58.6 Example: Double Pointer in Action

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

void set_value(int pp) {
    *pp = malloc(sizeof(int));
    if (*pp != NULL) {
        pp = 99;
```

```
    }
}

int main(void) {
    int *p = NULL;
    set_value(&p);

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

Output:

```
Value: 99
```

Here:

- `p` is a pointer to int
- `&p` is a pointer to pointer
- The function allocates memory and sets the value safely

58.7 Common Mistakes

1. Confusing `*` count: Each `*` represents one level of indirection. If you have `int pp`, then:

- `pp` points to a pointer
- `*pp` is a pointer to int
- `pp` is the actual int

2. Forgetting to initialize: Never use a pointer-to-pointer without setting its target.

```
int pp;
pp = 5; // invalid, pp is uninitialized
```

Always initialize step by step.

3. Forgetting to free memory: If your pointer-to-pointer was used to allocate memory, remember to free it at the end.

58.8 Arrays of Strings Example

You'll often see `char argv` in `main()`:

```
int main(int argc, char argv) {
    printf("Program name: %s\n", argv[0]);
}
```

Here, `argv` is a pointer to a list of strings:

- `argv` points to the first element (a pointer to a string)
- `argv[i]` is a `char *`
- `*argv[i]` is a character

Tiny Code Example

```
#include <stdio.h>

int main(void) {
    int value = 10;
    int *p = &value;
    int pp = &p;

    printf("Value: %d\n", value);
    printf("*p: %d\n", *p);
    printf("pp: %d\n", pp);

    pp = 20;
    printf("Updated Value: %d\n", value);

    return 0;
}
```

Output:

```
Value: 10
*p: 10
pp: 10
Updated Value: 20
```

Why It Matters

A pointer to pointer is just one more layer of address following. It may seem abstract now, but it's essential for working with:

- Functions that allocate memory
- Command-line arguments
- Dynamic arrays and matrices

Understanding this concept opens the door to more advanced data structures in C.

Try It Yourself

1. Declare an integer, a pointer, and a pointer to pointer. Print their values.
2. Change the original integer using `pp` and confirm it updates.
3. Write a function that takes `int` and allocates memory for an int.
4. Print all addresses: `&x`, `p`, and `pp`, see how they relate.
5. Practice counting stars (`*`), each one moves you one level deeper.

Once you get comfortable, pointer-to-pointer code starts feeling like following a clear path, one `*` at a time.

59. NULL and Dangling Pointers

When working with pointers, two very common and important ideas are NULL pointers and dangling pointers. Both help you manage memory safely, and avoiding mistakes with them will save you hours of debugging later.

Let's walk through what they mean and how to handle them step by step.

59.1 What Is a NULL Pointer

A NULL pointer is a pointer that points to nothing. It doesn't hold the address of any valid variable or memory block.

In code:

```
int *p = NULL;
```

Here, `p` is a pointer, but it doesn't point anywhere, it's safely "empty".

Think of it as a mailbox with no address. You can check if it's empty, but you can't deliver mail to it.

59.2 Why NULL Is Useful

NULL pointers are useful for three reasons:

1. Initialization, Start all pointers with a known value (`NULL` instead of random garbage).
2. Checking before use, You can safely test if a pointer is valid:

```
if (p != NULL) {
    *p = 10;
}
```

3. Resetting after free, Once memory is released, setting the pointer to `NULL` avoids accidental reuse.

Using `NULL` is like putting up a clear sign: "This pointer isn't pointing to anything right now."

59.3 What Happens If You Dereference NULL

Dereferencing (`*p`) a NULL pointer is a serious error. It leads to undefined behavior, often a crash called a segmentation fault.

Example (don't do this):

```
int *p = NULL;
*p = 10; // invalid - p doesn't point to valid memory
```

Always check before dereferencing:

```
if (p != NULL) {
    *p = 10;
}
```

59.4 What Is a Dangling Pointer

A dangling pointer is a pointer that used to point to valid memory, but that memory has since been freed or gone out of scope.

Example:

```
int *p = malloc(sizeof(int));
*p = 42;
free(p); // memory released
*p = 10; // invalid - dangling pointer
```

Now p still has the old address, but that address is no longer valid.

Another example:

```
int *q;
{
    int x = 5;
    q = &x;
}
// x goes out of scope here
*q = 10; // invalid - q now dangles
```

Dangling pointers point to memory that no longer exists.

59.5 How to Avoid Dangling Pointers

Here are some good habits:

1. Set pointers to NULL after freeing

   ```
   free(p);
   p = NULL;
   ```

2. Never return the address of a local variable

   ```
   int *bad_pointer() {
       int x = 10;
       return &x; // wrong: x will disappear after function ends
   }
   ```

3. Be careful with pointer copies If two pointers point to the same memory, freeing one leaves the other dangling. You'll need to manage ownership carefully.

59.6 Visual Example

Before freeing:

```
p  > [ 42 ]
```

After freeing:

```
p  > ??? (invalid)
```

Setting to NULL helps:

```
p  > NULL
```

Now you can check:

```
if (p == NULL) {
    printf("Pointer is safe.\n");
}
```

59.7 Combining Checks

You can write safe, clear code by checking before use:

```
if (p != NULL) {
    printf("%d\n", *p);
} else {
    printf("Pointer not set.\n");
}
```

It's better to skip a line of code than crash your program.

59.8 Tiny Code Example

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

int main(void) {
    int *p = NULL;

    p = malloc(sizeof(int));
    if (p == NULL) {
        printf("Allocation failed\n");
        return 1;
    }

    *p = 100;
    printf("Value: %d\n", *p);
```

```
    free(p);
    p = NULL; // prevent dangling

    if (p == NULL) {
        printf("Pointer is now safe.\n");
    }

    return 0;
}
```

Output:

```
Value: 100
Pointer is now safe.
```

59.9 Common Mistakes

1. Dereferencing NULL:

```
int *p = NULL;
*p = 5; // invalid - crash
```

2. Using after free:

```
free(p);
printf("%d\n", *p); // invalid - memory not yours anymore
```

3. Returning address of local variable:

```
int *make_pointer() {
    int x = 10;
    return &x; // invalid - x disappears after return
}
```

Each of these leads to unpredictable behavior, sometimes immediate crashes, sometimes silent corruption.

Why It Matters

Pointers are powerful, but they must always point to something valid. NULL gives you a safe, predictable starting point. By setting freed pointers to NULL, you keep your code clean and easy to debug.

A good rule:

> Every pointer is either valid and ready to use, or safely set to NULL.

Try It Yourself

1. Declare a pointer, set it to NULL, and check before using it.
2. Allocate memory, free it, and try using it before setting to NULL, see what happens.
3. Write a function that returns a pointer. Try returning the address of a local variable and watch your compiler warn you.
4. Practice setting every pointer to NULL after freeing.
5. Use `if (p != NULL)` before every dereference, make it a habit.

Once you get comfortable managing NULL and avoiding dangling pointers, you'll write programs that are both safe and stable, even when working directly with memory.

60. Debugging Memory Errors

By now, you've seen how to use pointers, `malloc`, and `free`. But even careful programmers sometimes make mistakes with memory, forgetting to free, freeing twice, using NULL by accident, or stepping past array bounds.

These are called memory errors, and learning how to spot and fix them is a big part of becoming confident in C. This section will help you develop good debugging habits.

60.1 What Are Memory Errors

A memory error happens when your program misuses memory. Common examples include:

1. Memory leaks – not freeing memory you've allocated
2. Use after free – trying to use memory that's already been freed
3. Invalid reads/writes – accessing memory outside valid bounds
4. NULL dereference – using a pointer that points to nothing
5. Double free – freeing the same pointer twice

Each one can cause your program to behave unpredictably, sometimes it crashes, sometimes it silently corrupts data.

60.2 Why They're Hard to Catch

Unlike syntax errors, memory errors often don't show up immediately. Your program might compile fine, even run fine for a while, and then crash randomly later.

That's because C doesn't protect you from invalid memory access, it trusts you completely. So your best defense is awareness and good tools.

60.3 Build with Debug Information

Always compile with debugging symbols so you can trace problems:

```
gcc -g program.c -o program
```

The `-g` flag stores extra info for tools like `gdb` (debugger) and `valgrind` (memory checker).

60.4 Using `valgrind`

`valgrind` is a tool that watches every memory operation. It can tell you if you forgot to free memory, wrote past array bounds, or freed something twice.

Run your program like this:

```
valgrind ./program
```

You'll get a report showing:

- How much memory you allocated and freed
- Where leaks occurred
- What line caused an invalid access

Example output:

```
==12345== Invalid read of size 4
==12345== at 0x40055A: main (program.c:15)
==12345== Address 0x520304c is 0 bytes after a block of size 16
```

This tells you exactly where something went wrong.

60.5 Common Memory Mistakes and Fixes

1. Forgetting to free

```
int *p = malloc(10 * sizeof(int));
// missing free(p)
```

Fix:

```
free(p);
p = NULL;
```

2. Using after free

```
int *p = malloc(sizeof(int));
free(p);
*p = 10; // invalid - p is freed
```

Fix:

```
free(p);
p = NULL; // prevents accidental reuse
```

3. Writing past the end of an array

```
int arr[3] = {1, 2, 3};
arr[3] = 10; // invalid index (0,1,2 valid)
```

Fix:

```
for (int i = 0; i < 3; i++) {
    arr[i] = i + 1;
}
```

4. Dereferencing NULL

```
int *p = NULL;
*p = 5; // crash
```

Fix:

```c
if (p != NULL) {
    *p = 5;
}
```

5. Freeing twice

```c
int *p = malloc(sizeof(int));
free(p);
free(p); // invalid
```

Fix:

```c
free(p);
p = NULL;
```

60.6 Step-by-Step Debugging with `gdb`

You can also use the GNU Debugger (`gdb`) to trace your program line by line.

Start it:

```
gdb ./program
```

Then run:

```
(gdb) run
```

If your program crashes, type:

```
(gdb) backtrace
```

It will show you which function and line caused the crash.

60.7 Using Assertions

Assertions are sanity checks that help catch bugs early:

```c
#include <assert.h>

int *p = malloc(sizeof(int));
assert(p != NULL); // stop if allocation failed
```

If the condition fails, the program stops immediately, helping you catch issues before they spread.

60.8 Good Debugging Habits

1. Initialize all pointers to NULL
2. Check before using a pointer
3. Set pointers to NULL after freeing
4. Track every malloc with a matching free
5. Test small parts of code often

If something behaves strangely, suspect memory first.

60.9 Tiny Code Example

Here's a small program with a hidden memory bug:

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

int main(void) {
    int *arr = malloc(3 * sizeof(int));
    for (int i = 0; i <= 3; i++) { // mistake: should be i < 3
        arr[i] = i + 1;
    }
    for (int i = 0; i < 3; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    free(arr);
    return 0;
}
```

It compiles and runs, but it writes past the end of the array. Run it with `valgrind` and you'll see the warning.

Fix:

```c
for (int i = 0; i < 3; i++) {
    arr[i] = i + 1;
}
```

Why It Matters

Debugging memory errors is a skill that separates careful programmers from frustrated ones. Once you know how to use tools like `valgrind`, and adopt habits like checking `NULL` and pairing every `malloc` with `free`, you'll spend less time hunting bugs and more time building cool programs.

Try It Yourself

1. Write a small program with a memory leak, then find it using `valgrind`.
2. Intentionally go out of bounds on an array and watch the tool's output.
3. Practice pairing each allocation with a `free`.
4. Add `assert` statements to check that pointers are valid.
5. Run your programs with `gcc -g` and step through with `gdb` when you crash.

Over time, debugging will feel less like guesswork and more like detective work, clear, logical, and satisfying.

Chapter 7. Structures and modular design

61. Defining `struct` Types

So far, you've worked with basic data types, integers, floats, and characters. But what if you want to group related pieces of data together? For example, a student has a name, an ID, and a grade, three different types, but all part of one concept.

In C, you can build your own custom data types using `struct`, short for structure.

61.1 What Is a `struct`

A `struct` lets you combine variables of different types into one single unit. It's like a container that holds fields, each with its own name and type.

Here's the basic pattern:

```c
struct Student {
    int id;
    char name[50];
    float grade;
};
```

This defines a new type, `struct Student`, which has three members:

- `id`, an integer
- `name`, a string
- `grade`, a floating-point number

Think of it like a mini record, one box with multiple labeled slots.

61.2 Declaring and Using a `struct`

Once you've defined a structure, you can create variables of that type:

```c
struct Student alice;
```

You now have a student record named `alice` with its own `id`, `name`, and `grade`.

You can access each field using the dot operator (`.`):

```c
alice.id = 1;
strcpy(alice.name, "Alice");
alice.grade = 95.5;
```

Then print them:

```c
printf("ID: %d, Name: %s, Grade: %.1f\n", alice.id, alice.name, alice.grade);
```

61.3 Initializing a `struct`

You can set values when you create it, just like arrays:

```c
struct Student bob = {2, "Bob", 88.0};
```

Each value is assigned in order, first `id`, then `name`, then `grade`.

You can also use designated initializers (a nice, modern feature):

```c
struct Student carol = {.name = "Carol", .id = 3, .grade = 91.2};
```

This makes your code clearer, especially for large structures.

61.4 Multiple Variables

You can declare multiple variables at once:

```c
struct Student s1, s2, s3;
```

Each one is an independent copy with its own fields.

61.5 Array of Structures

You can store many `struct` values in an array, great for managing records:

```c
struct Student students[3] = {
    {1, "Alice", 95.5},
    {2, "Bob", 88.0},
    {3, "Carol", 91.2}
};
```

Loop through them easily:

```c
for (int i = 0; i < 3; i++) {
    printf("%s: %.1f\n", students[i].name, students[i].grade);
}
```

61.6 Copying Structures

You can assign one structure to another directly, C copies all fields:

```c
struct Student temp = alice;
```

Now `temp` is an exact copy of `alice`. (But be careful later, with pointers inside structs, this behaves differently.)

61.7 Anonymous `struct`

Sometimes you don't need a type name, you can define and use a `struct` right away:

```
struct {
    int x;
    int y;
} point;

point.x = 10;
point.y = 20;
```

This is handy for one-off uses where you don't plan to reuse the type elsewhere.

61.8 Combining Types

You can even have structures inside structures:

```
struct Date {
    int day, month, year;
};

struct Student {
    int id;
    char name[50];
    float grade;
    struct Date birthdate;
};
```

Now each student has their own birthday, a structure within a structure!

61.9 Tiny Code Example

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

struct Student {
    int id;
    char name[50];
```

```
    float grade;
};

int main(void) {
    struct Student alice;

    alice.id = 1;
    strcpy(alice.name, "Alice");
    alice.grade = 95.5;

    printf("Student Info:\n");
    printf("ID: %d\n", alice.id);
    printf("Name: %s\n", alice.name);
    printf("Grade: %.1f\n", alice.grade);

    return 0;
}
```

Output:

```
Student Info:
ID: 1
Name: Alice
Grade: 95.5
```

Why It Matters

`struct` is your first step toward structured programming. It lets you create your own data types that model real-world things, students, books, cars, customers, clearly and naturally.

Once you learn how to define and organize structures, you'll start building programs that deal with complex data, not just numbers and strings.

Try It Yourself

1. Create a `struct Book` with fields for title, author, and price.
2. Make two book variables, fill them, and print them.
3. Store several books in an array of `struct Book`.
4. Copy one structure to another and confirm all fields match.
5. Add a nested `struct Date` for a publish date and print it.

With `struct`, you're no longer just storing data, you're designing it.

62. Accessing Structure Members

Now that you know how to define a `struct`, let's learn how to work with its members, the individual fields inside it. You'll use these fields all the time, whether to store values, read them, or pass them around in your program.

Once you see the patterns, it becomes second nature.

62.1 The Dot Operator (`.`)

If you have a structure variable, you access its members using the dot (`.`) operator. It's short for "go inside this structure and get this field."

Example:

```
struct Point {
    int x;
    int y;
};

struct Point p;
p.x = 10;
p.y = 20;

printf("x = %d, y = %d\n", p.x, p.y);
```

Here:

- `p.x` means "the `x` field of structure `p`"
- `p.y` means "the `y` field of structure `p`"

The dot is your "member access" tool.

62.2 Initializing and Printing Members

You can set each field manually:

```
p.x = 3;
p.y = 7;
```

Or initialize when you create it:

```c
struct Point p = {3, 7};
```

Printing fields is just like printing any variable:

```c
printf("Point: (%d, %d)\n", p.x, p.y);
```

62.3 Assigning One Structure to Another

C allows full structure assignment, it copies every field.

```c
struct Point a = {1, 2};
struct Point b = a; // copies x and y
```

Now `b.x == 1` and `b.y == 2`.

Each field is duplicated automatically, no need to copy one by one.

62.4 Nested Access

If a structure contains another structure, use multiple dots to reach deep inside.

Example:

```c
struct Date {
    int day, month, year;
};

struct Student {
    int id;
    char name[50];
    struct Date birthday;
};

struct Student s = {1, "Alice", {15, 5, 2000}};
printf("%s was born on %d/%d/%d\n",
       s.name, s.birthday.day, s.birthday.month, s.birthday.year);
```

You can chain dots as far as needed, each one steps one level deeper.

62.5 Accessing Through Pointers

If you have a pointer to a structure, use the arrow operator (`->`) instead of the dot. The arrow means "follow the pointer, then access the field."

```
struct Point p = {5, 10};
struct Point *ptr = &p;

printf("%d\n", (*ptr).x);  // long form
printf("%d\n", ptr->x);    // short form, easier to read
```

Both lines mean the same thing. `ptr->x` is just shorthand for `(*ptr).x`.

62.6 Visual Picture

Think of the dot and arrow like this:

- `p.x`, "take p, then go inside to x"
- `ptr->x`, "follow pointer ptr, then go inside to x"

The arrow combines dereference (`*`) and access (`.`) into one step.

62.7 Example: Using -> with Dynamic Memory

Let's allocate a structure with `malloc` and use `->`:

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

struct Point {
    int x, y;
};

int main(void) {
    struct Point *p = malloc(sizeof(struct Point));
    if (p == NULL) return 1;

    p->x = 7;
    p->y = 9;

    printf("(%d, %d)\n", p->x, p->y);
```

```
    free(p);
    p = NULL;

    return 0;
}
```

Output:

```
(7, 9)
```

Notice we never used `(*p).x`, the arrow is simpler and safer.

62.8 Modifying Members

You can change fields directly:

```
p.x = p.x + 5;
p.y = 0;
```

Or through a pointer:

```
ptr->y = 25;
```

These updates affect only the field you change, the rest stay the same.

62.9 Passing Structures to Functions

You can pass a structure to a function by value or by pointer.

By value (copies all fields):

```
void print_point(struct Point p) {
    printf("(%d, %d)\n", p.x, p.y);
}
```

By pointer (modifies the original):

```c
void move_point(struct Point *p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}
```

This choice depends on whether you want to modify or just view the data.

62.10 Tiny Code Example

```c
#include <stdio.h>

struct Point {
    int x, y;
};

void move(struct Point *p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}

int main(void) {
    struct Point p = {10, 20};
    printf("Before: (%d, %d)\n", p.x, p.y);

    move(&p, 5, -10);
    printf("After: (%d, %d)\n", p.x, p.y);

    return 0;
}
```

Output:

```
Before: (10, 20)
After: (15, 10)
```

Why It Matters

Being able to access and modify structure members is what makes `struct` so powerful. You can organize your data, read and write specific fields, and pass whole records to functions easily.

Whether you're working with coordinates, students, books, or bank accounts, you'll use the dot and arrow operators constantly, they're the keys to navigating your data.

Try It Yourself

1. Create a `struct Rectangle` with `width` and `height` and print its area.
2. Write a function that doubles both fields of a `struct Point` using a pointer.
3. Create a nested structure and practice chaining dots (`.`).
4. Allocate a structure with `malloc` and use the arrow (`->`) to fill its fields.
5. Copy one structure to another and confirm both have the same values.

Once you're comfortable with `.` and `->`, structures become your best tool for building clear, organized programs.

63. Structures and Functions

You've learned how to define structures and how to access their members, now let's make them work together with functions. This is a key skill because real programs often use functions to create, modify, and display structured data.

You'll learn how to pass structures by value (copying them) or by pointer (sharing them), and when to use each.

63.1 Why Use Structures with Functions

Structures group related data. Functions group related behavior. Combining the two gives you clean, readable programs that handle data in a clear, modular way.

For example, if you have a `struct Point`, you might want functions to:

- create a new point
- print a point
- move a point

Each function focuses on one task, keeping your code simple and reusable.

63.2 Passing a Structure by Value

When you pass a structure by value, C makes a copy of the structure. The function can read or modify the copy, but the original remains unchanged.

Example:

```c
#include <stdio.h>

struct Point {
    int x, y;
};

void print_point(struct Point p) {
    printf("Point: (%d, %d)\n", p.x, p.y);
}

int main(void) {
    struct Point p = {5, 10};
    print_point(p); // sends a copy
    return 0;
}
```

Here, `p` is copied when passed. Changes inside `print_point` won't affect the original `p`.

63.3 Passing a Structure by Pointer

When you pass a pointer to a structure, the function works directly on the original. This is more efficient (no copying large data) and allows the function to modify the actual structure.

Example:

```c
void move_point(struct Point *p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}
```

Usage:

```c
struct Point p = {5, 10};
move_point(&p, 3, -2);
printf("Moved to: (%d, %d)\n", p.x, p.y);
```

Output:

```
Moved to: (8, 8)
```

The `&` operator gives the function the address of `p`, and the `->` operator accesses its members inside the function.

63.4 When to Use Value vs Pointer

Approach	What Happens	Use When
By Value	Structure is copied	You only need to read or print data
By Pointer	Function works on the original	You want to modify the structure

For small structures, passing by value is fine. For large ones (many fields), pointers are faster and save memory.

63.5 Returning a Structure

Functions can also return structures. This is useful when you want to create and send back a filled record.

Example:

```c
struct Point create_point(int x, int y) {
    struct Point p = {x, y};
    return p;
}

int main(void) {
    struct Point p = create_point(3, 4);
    printf("Created: (%d, %d)\n", p.x, p.y);
    return 0;
}
```

C copies the structure on return, just like returning an integer.

63.6 Returning a Pointer

You can also return a pointer, but be careful, never return the address of a local variable (it disappears when the function ends).

Wrong:

```
struct Point* bad_point() {
    struct Point p = {1, 2};
    return &p; // invalid: p is local
}
```

Right:

```
#include <stdlib.h>

struct Point* make_point(int x, int y) {
    struct Point *p = malloc(sizeof(struct Point));
    if (p != NULL) {
        p->x = x;
        p->y = y;
    }
    return p;
}
```

Just remember to `free()` the memory later.

63.7 Example: Functions for a Rectangle

Let's see structures and functions in action together.

```
#include <stdio.h>

struct Rectangle {
    int width;
    int height;
};

int area(struct Rectangle r) {
    return r.width * r.height;
}
```

```c
void double_size(struct Rectangle *r) {
    r->width *= 2;
    r->height *= 2;
}

int main(void) {
    struct Rectangle box = {5, 3};

    printf("Area: %d\n", area(box));

    double_size(&box);
    printf("New area: %d\n", area(box));

    return 0;
}
```

Output:

```
Area: 15
New area: 60
```

You pass by value when you only need to read data, and by pointer when you want to change it.

63.8 Combining Functions and Arrays of Structures

You can pass an array of structures to a function using a pointer.

```c
void print_all(struct Point *points, int count) {
    for (int i = 0; i < count; i++) {
        printf("(%d, %d)\n", points[i].x, points[i].y);
    }
}
```

This works because arrays decay into pointers when passed to functions.

63.9 Tiny Code Example

```c
#include <stdio.h>

struct Point {
    int x, y;
};

void shift(struct Point *p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}

void print(struct Point p) {
    printf("Point: (%d, %d)\n", p.x, p.y);
}

int main(void) {
    struct Point p = {2, 3};
    print(p);

    shift(&p, 4, 5);
    print(p);

    return 0;
}
```

Output:

```
Point: (2, 3)
Point: (6, 8)
```

Why It Matters

Using structures with functions helps you organize your code around real-world data and actions. Instead of passing multiple variables, you pass a single meaningful package. This keeps your functions focused, your code neat, and your logic easier to follow.

Try It Yourself

1. Create a `struct Circle` with a radius. Write a function to compute its area.

2. Write a function `void scale(struct Circle *c, double factor)` that multiplies the radius.
3. Write a function that creates and returns a new `struct Point`.
4. Try passing a structure by value and by pointer, see the difference.
5. Build a small program managing 3 students, functions to create, print, and update their grades.

Once you master structures and functions, your programs start feeling like small systems, each part clear, modular, and easy to understand.

64. Nested Structures

Sometimes, one structure alone isn't enough to describe something fully. A student has a name and grade, sure, but also a birth date, maybe an address, or a course record. Instead of stuffing all this data into one flat list of fields, you can use nested structures, structures *inside* other structures.

This helps you model real-world data clearly and keeps your code organized.

64.1 Why Nest Structures

Think of nested structures as "building blocks." Each block models one part of a bigger idea, and you can combine them to form complete objects.

Example: You might want to describe a `Student` like this:

- ID
- Name
- Birthday (which has day, month, year)

So you create a separate `struct Date`, and use it inside `struct Student`.

64.2 Defining a Nested Structure

Start by defining the smaller structure first:

```
struct Date {
    int day;
    int month;
    int year;
};
```

Then use it as a field in another structure:

```
struct Student {
    int id;
    char name[50];
    float grade;
    struct Date birthday;
};
```

Now every `Student` automatically includes a `Date`.

64.3 Initializing a Nested Structure

You can set nested values directly using chained dots:

```
struct Student s;
s.id = 1;
strcpy(s.name, "Alice");
s.grade = 95.0;

s.birthday.day = 15;
s.birthday.month = 5;
s.birthday.year = 2000;
```

Or use a single initializer:

```
struct Student s = {1, "Alice", 95.0, {15, 5, 2000}};
```

This sets the outer structure, then the inner one in order.

64.4 Accessing Nested Members

Use the dot (`.`) to step deeper into the structure:

```
printf("%s was born on %d/%d/%d\n",
       s.name, s.birthday.day, s.birthday.month, s.birthday.year);
```

You can chain as many dots as needed:

```
outer.inner.field
```

It's just like reading a path, "from the student, go to the birthday, then to the day."

64.5 With Pointers

If you have a pointer to a structure that contains another structure, use `->` for the outer access, then `.` for the inner.

```
struct Student *p = &s;
printf("%d\n", p->birthday.year);
```

Or combine both with parentheses:

```
printf("%d\n", (*p).birthday.year);
```

But `p->birthday.year` is easier to read, use it whenever possible.

64.6 Example: Student Record

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

struct Date {
    int day, month, year;
};

struct Student {
    int id;
    char name[50];
    float grade;
    struct Date birthday;
};

int main(void) {
    struct Student alice = {1, "Alice", 95.5, {15, 5, 2000}};

    printf("ID: %d\n", alice.id);
    printf("Name: %s\n", alice.name);
    printf("Grade: %.1f\n", alice.grade);
    printf("Birthday: %02d/%02d/%d\n",
           alice.birthday.day,
           alice.birthday.month,
           alice.birthday.year);
```

```
    return 0;
}
```

Output:

```
ID: 1
Name: Alice
Grade: 95.5
Birthday: 15/05/2000
```

64.7 Multiple Levels

You can nest more than once, structures can contain structures that contain others.

```
struct Address {
    char city[50];
    char country[50];
};

struct Date {
    int day, month, year;
};

struct Student {
    int id;
    char name[50];
    struct Date birthday;
    struct Address address;
};
```

Now you can do:

```
strcpy(s.address.city, "Paris");
printf("%s lives in %s\n", s.name, s.address.city);
```

Each layer adds more detail in a clear, structured way.

64.8 Passing Nested Structures to Functions

You can pass them the same way as normal structs, by value or pointer.

By value:

```
void print_student(struct Student s) {
    printf("%s: %d/%d/%d\n",
           s.name, s.birthday.day, s.birthday.month, s.birthday.year);
}
```

By pointer:

```
void print_student_ptr(struct Student *s) {
    printf("%s: %d/%d/%d\n",
           s->name, s->birthday.day, s->birthday.month, s->birthday.year);
}
```

Pointers are more efficient, especially if your structure is large.

64.9 Tiny Code Example

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

struct Date {
    int day, month, year;
};

struct Student {
    char name[50];
    struct Date birthday;
};

void print_student(struct Student s) {
    printf("%s was born on %02d/%02d/%d\n",
           s.name, s.birthday.day, s.birthday.month, s.birthday.year);
}

int main(void) {
    struct Student bob = {"Bob", {23, 8, 1999}};
```

```
    print_student(bob);
    return 0;
}
```

Output:

```
Bob was born on 23/08/1999
```

Why It Matters

Nested structures help you design data the way you think about it. Instead of juggling multiple separate variables, you describe real objects clearly: students with birthdays, cars with engines, orders with dates and totals.

They make your programs more readable, more logical, and easier to extend later.

Try It Yourself

1. Create a `struct Date` and `struct Book`, each book has a title, author, and publish date.
2. Fill in a few books and print their info with dates.
3. Add another level: each book has a `struct Price` with `currency` and `amount`.
4. Write a function that prints all fields cleanly.
5. Use pointers (`->`) to modify inner structures.

Once you get the hang of nesting, you'll see structures as building blocks, small, reusable pieces that describe anything you want.

65. Arrays of Structures

You've seen how to make a single `struct` that groups related data, now imagine you want to store many of them. For example, a school doesn't have just one student, it has hundreds. A library doesn't have just one book, it has shelves full.

That's where arrays of structures come in. They let you keep a collection of structured records, all of the same type, in one neat package.

65.1 Why Use Arrays of Structures

An array stores multiple items of the same type. If each item is a `struct`, you can manage lots of related objects together.

Example: a list of students, each with an ID, name, and grade.

```
struct Student {
    int id;
    char name[50];
    float grade;
};

struct Student students[3];
```

Now you can store 3 `Student` records in one array, `students[0]`, `students[1]`, `students[2]`.

65.2 Initializing the Array

You can fill each element one by one:

```
students[0].id = 1;
strcpy(students[0].name, "Alice");
students[0].grade = 95.0;

students[1].id = 2;
strcpy(students[1].name, "Bob");
students[1].grade = 88.5;
```

Or, use a single initializer list for all of them:

```
struct Student students[3] = {
    {1, "Alice", 95.0},
    {2, "Bob", 88.5},
    {3, "Carol", 91.2}
};
```

This sets up the whole table at once, clean and clear.

65.3 Accessing Fields

Accessing a structure inside an array is easy, use the index, then the field:

```
printf("%s got %.1f\n", students[0].name, students[0].grade);
```

In general:

```
array[index].field
```

The `.` operator always comes after the index.

65.4 Looping Over an Array of Structures

Because it's an array, you can loop through it with a simple `for` loop:

```
for (int i = 0; i < 3; i++) {
    printf("%d: %s (%.1f)\n",
           students[i].id,
           students[i].name,
           students[i].grade);
}
```

This prints every record in one go.

Output:

```
1: Alice (95.0)
2: Bob (88.5)
3: Carol (91.2)
```

65.5 Using Arrays with Functions

You can pass the array to a function just like any other array, it decays to a pointer. The function can then work with all the records.

Example:

```c
void print_students(struct Student list[], int count) {
    for (int i = 0; i < count; i++) {
        printf("%d: %s (%.1f)\n",
               list[i].id,
               list[i].name,
               list[i].grade);
    }
}
```

Call it like this:

```c
print_students(students, 3);
```

This keeps your main code simple and your logic reusable.

65.6 Adding New Records

You can modify elements in place:

```c
students[2].id = 4;
strcpy(students[2].name, "Dave");
students[2].grade = 89.0;
```

Arrays let you easily update, add, or replace records by index.

65.7 Searching for a Record

You can search an array of structures by looping through its elements:

```c
for (int i = 0; i < 3; i++) {
    if (strcmp(students[i].name, "Bob") == 0) {
        printf("Found Bob: grade = %.1f\n", students[i].grade);
    }
}
```

This is a simple way to find data until you learn more advanced data structures later.

65.8 Arrays of Nested Structures

If your structure contains another structure, you can still store many of them in an array.

```
struct Date { int day, month, year; };

struct Student {
    int id;
    char name[50];
    struct Date birthday;
};

struct Student class[2] = {
    {1, "Alice", {15, 5, 2000}},
    {2, "Bob", {20, 7, 1999}}
};

printf("%s was born in %d\n", class[1].name, class[1].birthday.year);
```

You can mix arrays and nested structures freely, C handles it all.

65.9 Tiny Code Example

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

struct Student {
    int id;
    char name[50];
    float grade;
};

int main(void) {
    struct Student students[3] = {
        {1, "Alice", 95.0},
        {2, "Bob", 88.5},
        {3, "Carol", 91.2}
    };

    printf("Student List:\n");
```

```
    for (int i = 0; i < 3; i++) {
        printf("%d. %s - Grade: %.1f\n",
               students[i].id,
               students[i].name,
               students[i].grade);
    }

    return 0;
}
```

Output:

```
Student List:
1. Alice - Grade: 95.0
2. Bob - Grade: 88.5
3. Carol - Grade: 91.2
```

Why It Matters

Arrays of structures let you manage collections of complex data easily. Whether it's students, books, employees, or inventory, you'll use this pattern again and again to build real-world programs.

It's the first step toward databases and records, storing, retrieving, and managing lots of information clearly.

Try It Yourself

1. Create a `struct Book` with title, author, and price.
2. Make an array of 5 books and print all their titles.
3. Write a function that searches for a book by name.
4. Add a field `in_stock` and update some records in a loop.
5. Print only the books that cost more than 20.

Arrays of structures are your first taste of structured data management, a skill you'll use in nearly every program you write from now on.

66. `typedefs` for Simpler Names

As your programs grow, your structure names can get pretty long, like `struct StudentRecord` or `struct NetworkConfiguration`. Typing `struct` every time can quickly become tiresome.

That's where `typedef` comes in. It lets you create shorter, simpler names for your types, making your code easier to read and write.

66.1 What `typedef` Does

The `typedef` keyword creates a type alias, a new name for an existing type. It doesn't create a new kind of data, it just gives an existing one a nickname.

For example:

```
typedef int Length;
```

Now `Length` means exactly the same as `int`. So you can write:

```
Length x = 10;
```

This is useful when you want your code to be self-documenting. `Length` clearly means "this value represents a length," not just any integer.

66.2 `typedef` with `struct`

You can use `typedef` to remove the need to write `struct` all the time.

Without `typedef`:

```
struct Point {
    int x;
    int y;
};

struct Point p;
```

With `typedef`:

```c
typedef struct {
    int x;
    int y;
} Point;

Point p;
```

Now you can use `Point` like a built-in type, no `struct` keyword needed.

66.3 Two Common Styles

There are two popular ways to use `typedef` with structures:

1. Anonymous struct + typedef

```c
typedef struct {
    int x;
    int y;
} Point;
```

Here, you don't name the struct separately, you only use `Point`.

2. Named struct + typedef

```c
typedef struct Point {
    int x;
    int y;
} Point;
```

This gives both the struct *and* the type a name. You can still refer to it as `struct Point` if needed.

Both styles are valid, choose whichever feels clearer to you.

66.4 Example Without and With `typedef`

Without `typedef`:

```c
struct Student {
    int id;
    char name[50];
};

struct Student s;
```

With `typedef`:

```c
typedef struct {
    int id;
    char name[50];
} Student;

Student s;
```

Cleaner, right? You get the same structure, but less typing and clutter.

66.5 `typedef` for Pointers

You can also create aliases for pointer types:

```c
typedef int* IntPtr;

IntPtr p1, p2;
```

Be careful though, `IntPtr p1, p2;` makes both `p1` and `p2` pointers, while `int *p1, p2;` makes only `p1` a pointer. So typedefs can make pointer declarations clearer.

66.6 Using `typedef` for Readability

Using descriptive typedefs makes code easier to understand:

```c
typedef float Temperature;
typedef unsigned long ID;

Temperature room = 24.5;
ID user = 12345;
```

You can instantly tell what each value represents, even though they're just basic types.

66.7 Combining with Arrays and Functions

Once you define a typedef, you can use it everywhere, in arrays, functions, or pointers.

```
typedef struct {
    int x, y;
} Point;

void print_point(Point p) {
    printf("(%d, %d)\n", p.x, p.y);
}

int main(void) {
    Point points[3] = {{1, 2}, {3, 4}, {5, 6}};
    for (int i = 0; i < 3; i++) {
        print_point(points[i]);
    }
    return 0;
}
```

No `struct` keyword needed anywhere.

66.8 Another Example: Complex Types

Let's define a structure with nested fields:

```
typedef struct {
    char name[50];
    float price;
} Product;

typedef struct {
    Product item;
    int quantity;
} Order;
```

Now `Order` is just a clean, readable type:

```
Order o = {{"Notebook", 2.5}, 10};
printf("%s x %d\n", o.item.name, o.quantity);
```

66.9 Tiny Code Example

```c
#include <stdio.h>

typedef struct {
    int x;
    int y;
} Point;

void move(Point *p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}

int main(void) {
    Point p = {3, 4};
    move(&p, 2, 1);
    printf("New position: (%d, %d)\n", p.x, p.y);
    return 0;
}
```

Output:

```
New position: (5, 5)
```

Notice how clean it looks, no `struct` in sight!

Why It Matters

`typedef` makes your code friendlier and more expressive. It's like giving your data types human-readable names. Instead of thinking in terms of "integers and structs," you can think in terms of Points, Students, Orders, and Temperatures, just like in real life.

When your programs get bigger, small readability improvements make a big difference.

Try It Yourself

1. Create a `typedef struct` named `Book` with `title` and `price`.
2. Make an array `Book library[3]` and fill it with data.
3. Write a function `void print_book(Book b)` to display one record.

4. Add another `typedef` for `Price` as `float` and use it in your `Book`.
5. Compare the code before and after `typedef`, which looks easier to read?

Once you start using `typedef`, you'll wonder how you ever lived without it, it's one of those small tools that makes C feel smooth and elegant.

67. Enums and Symbolic Constants

So far, you've used numbers to represent values, `0`, `1`, `2`, and so on. But what if your program needs to deal with categories, like days of the week, colors, or menu options?

You *could* use numbers for them, but then your code ends up full of mysterious values:

```
if (mode == 2) { /* ??? */ }
```

What does `2` mean here? It's not obvious.

To make code clearer, C gives you enums, a way to give names to sets of related integer constants.

67.1 What Is an `enum`

An enumeration (or `enum`) is a type that lets you list named constants.

For example:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
```

Here, `MONDAY` is `0`, `TUESDAY` is `1`, and so on, unless you say otherwise.

Now you can write:

```
enum Day today = WEDNESDAY;
```

Much clearer than:

```
int today = 2; // what does 2 mean?
```

67.2 Using `enum` Values

You can compare or switch on them like normal integers:

```c
if (today == FRIDAY) {
    printf("Weekend is near!\n");
}
```

Or use them in a `switch`:

```c
switch (today) {
    case MONDAY: printf("Start of week!\n"); break;
    case FRIDAY: printf("Almost weekend!\n"); break;
    default: printf("Another day...\n"); break;
}
```

Each name stands for a specific integer, but you don't have to remember which one.

67.3 Assigning Custom Values

By default, counting starts at `0`, but you can set your own starting values:

```c
enum ErrorCode { SUCCESS = 0, WARNING = 1, ERROR = 2 };
```

Or assign any number:

```c
enum Month { JAN = 1, FEB, MAR, APR }; // continues as 1, 2, 3, 4
```

C automatically counts up from the last value.

67.4 `typedef` with `enum`

You can pair `typedef` with `enum` to make it easier to use:

```c
typedef enum {
    RED,
    GREEN,
    BLUE
} Color;

Color background = BLUE;
```

Now you don't need to write `enum Color` every time, just `Color`.

67.5 Example: Menu Options

```c
#include <stdio.h>

typedef enum {
    MENU_START,
    MENU_OPTIONS,
    MENU_EXIT
} Menu;

int main(void) {
    Menu choice = MENU_START;

    switch (choice) {
        case MENU_START: printf("Game started!\n"); break;
        case MENU_OPTIONS: printf("Options menu.\n"); break;
        case MENU_EXIT: printf("Goodbye!\n"); break;
    }
    return 0;
}
```

Output:

```
Game started!
```

Each option has a name, so your program reads like a story.

67.6 Mixing `enum` and `switch`

Enums and switches work beautifully together. They make your control flow expressive and easy to change later, just add new cases!

If you later add `MENU_CREDITS`, you simply extend both the `enum` and the `switch`.

67.7 Enums Inside Structs

You can even put enums inside structures:

```c
typedef enum {
    TASK_TODO,
    TASK_IN_PROGRESS,
    TASK_DONE
} Status;

typedef struct {
    char title[50];
    Status state;
} Task;
```

Now you can write:

```c
Task t = {"Write Chapter", TASK_IN_PROGRESS};
```

Enums keep your data meaningful and tidy.

67.8 Symbolic Constants with `#define`

Before enums, many C programs used macros for named constants:

```c
#define MONDAY 0
#define TUESDAY 1
```

This works, but `enum` is safer, it creates a real type, not just text substitution.

You can still use macros for single constants:

```c
#define MAX_SIZE 100
#define PI 3.14159
```

These are handy for fixed values, while enums are best for sets of related values.

67.9 Tiny Code Example

```c
#include <stdio.h>

typedef enum {
    NORTH,
    EAST,
    SOUTH,
    WEST
} Direction;

int main(void) {
    Direction move = EAST;

    if (move == EAST) {
        printf("Going east!\n");
    }

    return 0;
}
```

Output:

```
Going east!
```

Now, anyone reading your code knows what `EAST` means, no guesswork needed.

Why It Matters

Enums turn magic numbers into meaningful names. They make your code self-explanatory and reduce errors, you'll never mix up `0` and `2` again when they're called `MONDAY` and `WEDNESDAY`.

And with `typedef`, they become as easy to use as `int`.

Try It Yourself

1. Create an enum `TrafficLight` with `RED`, `YELLOW`, `GREEN`.
2. Write a `switch` that prints what to do for each light.
3. Create an enum `Difficulty` for a game with `EASY`, `MEDIUM`, `HARD`.
4. Use it in a struct `GameSettings` with a field for difficulty.
5. Mix `#define` for constants like `MAX_SCORE` with enums for categories.

Enums help your code tell a story, instead of "if value == 2," you'll be writing "if state == GAME_OVER."

68. Unions and Shared Memory

So far, every `struct` you've written keeps all of its fields in memory at once. But sometimes, you only need one of several possible fields at a time.

For example, imagine a program that stores either an integer, a float, or a string, but never more than one at once. You could use a structure with all three fields, but that would waste space.

To save memory and represent "one-of-many" data, C gives you a tool called a union.

68.1 What Is a `union`

A `union` looks like a `struct`, but instead of separate storage for each field, all fields share the same memory space.

Only one member can hold a value at a time.

Example:

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

Here, `i`, `f`, and `str` all start at the same memory address. If you write to one, you overwrite the others.

68.2 Declaring and Using a `union`

You can define and use a union just like a struct:

```
union Data d;

d.i = 42;
printf("i = %d\n", d.i);

d.f = 3.14;
printf("f = %.2f\n", d.f);
```

But remember: when you set `d.f`, it overwrites the value of `d.i`. They live in the same space.

68.3 How Big Is a Union?

The size of a union is the size of its largest member. In our example, `str[20]` is the largest (20 bytes), so `sizeof(union Data)` is 20.

C must make sure there's enough space for the biggest member.

68.4 Why Use a Union?

Unions are useful when you want one variable that can store multiple types (but not at the same time).

They're common in:

- Data parsers (like reading binary files)
- Interpreters or expression trees
- Embedded systems, where memory is limited

They let you design memory-efficient records.

68.5 Using `typedef` with `union`

You can make them easier to use with `typedef`:

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

Data value;
```

Now you can just say `Data` instead of `union Data`.

68.6 Example: Tagged Union

Because unions don't remember what type they currently hold, you often combine them with an `enum` to track the active field.

```c
typedef enum { TYPE_INT, TYPE_FLOAT, TYPE_STRING } DataType;

typedef union {
    int i;
    float f;
    char str[20];
} DataValue;

typedef struct {
    DataType type;
    DataValue value;
} Variant;
```

Now you can store any of the three types safely:

```c
Variant v;
v.type = TYPE_INT;
v.value.i = 42;

if (v.type == TYPE_INT)
    printf("Integer: %d\n", v.value.i);
```

This pattern is called a tagged union, it's a safe, structured way to handle mixed data.

68.7 Union vs Struct

Feature	`struct`	`union`
Memory	Each member gets its own space	All members share one space
Size	Sum of all members	Size of largest member
Use case	Many values at once	One value at a time

68.8 Real Example: Network Packet

Imagine reading data from a network that can be either a number or text:

```c
typedef enum { PACKET_NUMBER, PACKET_TEXT } PacketType;

typedef union {
    int number;
```

```
    char text[64];
} PacketData;

typedef struct {
    PacketType type;
    PacketData data;
} Packet;
```

When you read a packet:

```
Packet p;
p.type = PACKET_NUMBER;
p.data.number = 1001;
```

Later, you check `p.type` to know how to interpret `p.data`.

68.9 Tiny Code Example

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

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

int main(void) {
    Data d;

    d.i = 42;
    printf("i: %d\n", d.i);

    d.f = 3.14;
    printf("f: %.2f\n", d.f);

    strcpy(d.str, "Hello");
    printf("str: %s\n", d.str);

    return 0;
}
```

Output:

```
i: 42
f: 3.14
str: Hello
```

Note: after writing to `str`, the values of `i` and `f` are no longer valid, they've been overwritten.

Why It Matters

Unions teach you how memory works behind the scenes. They give you fine-grained control, and save space when you need flexibility without duplication.

Once you understand unions, you'll have a deeper appreciation for how C represents data and how systems handle different types efficiently.

Try It Yourself

1. Create a `union Number` with `int` and `float`.
2. Assign each type in turn and print the value.
3. Build a `typedef` for a `union Value` with `int`, `float`, and `char text[20]`.
4. Combine it with an `enum` tag in a `struct Variant` to track which one is active.
5. Print different outputs based on the tag, just like a small variant type.

Unions may seem simple, but they unlock a powerful way of thinking about shared memory, one space, many possibilities.

69. Organizing Code into Modules

As your C programs grow, putting everything into a single file becomes messy fast. You'll find yourself scrolling through hundreds of lines, variables, functions, and structures all tangled together.

That's why professional C programmers split their code into modules. Each module focuses on one purpose, and C gives you the tools to organize them cleanly.

69.1 What Is a Module

A module is simply a pair of files that belong together:

- A header file (`.h`), contains declarations (what exists)
- A source file (`.c`), contains definitions (how it works)

Together, they define a single logical unit of your program, maybe a math library, a logger, or a student manager.

Example:

```
math_utils.h
math_utils.c
```

This is the foundation of modular C design.

69.2 Why Use Modules

Modules make your program:

- Easier to read, one file per topic
- Easier to maintain, fix or extend one piece at a time
- Reusable, include the same header in many programs
- Collaborative, different people can work on different parts

You don't need to scroll through unrelated code anymore.

69.3 The Header File (`.h`)

A header file declares what the module provides — the function prototypes, structures, and constants that others can use.

Example (`math_utils.h`):

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int subtract(int a, int b);

#endif
```

Here:

- `#ifndef`, `#define`, `#endif` prevent double inclusion (called a *header guard*)
- You only declare functions, not define them

69.4 The Source File (`.c`)

The source file implements how those functions actually work.

Example (`math_utils.c`):

```c
#include "math_utils.h"

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}
```

Notice you `#include` your own header, that way, your function signatures stay in sync.

69.5 Using the Module

Now you can use this module in your main program:

`main.c`

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    printf("3 + 4 = %d\n", add(3, 4));
    printf("10 - 2 = %d\n", subtract(10, 2));
    return 0;
}
```

Then compile both files together:

```
gcc main.c math_utils.c -o app
```

Your program now has cleanly separated parts, each focused and easy to understand.

69.6 Splitting Structures into Modules

You can also keep structure definitions in headers, so other files can use them:

`student.h`

```
#ifndef STUDENT_H
#define STUDENT_H

typedef struct {
    int id;
    char name[50];
    float grade;
} Student;

void print_student(Student s);

#endif
```

`student.c`

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

void print_student(Student s) {
    printf("%d - %s: %.1f\n", s.id, s.name, s.grade);
}
```

`main.c`

```
#include "student.h"
#include <string.h>

int main(void) {
    Student s = {1, "Alice", 95.0};
    print_student(s);
    return 0;
}
```

69.7 Header Guards

Always wrap header files in include guards:

```c
#ifndef FILE_NAME_H
#define FILE_NAME_H
// content
#endif
```

They prevent errors if the header is included more than once.

69.8 Private vs Public Functions

You can hide helper functions by declaring them static inside `.c` files.

```c
static void helper(void) {
    // only visible inside this file
}
```

This keeps your module's internal details private, only the header's functions are public.

69.9 Project Structure Example

Here's how a small project might look:

```
project/
  main.c
  math_utils.h
  math_utils.c
  student.h
  student.c
```

Each `.c` file has its own `.h` file. You include only what you need, keeping files focused and clean.

Tiny Code Example

math_utils.h

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int square(int n);

#endif
```

math_utils.c

```c
#include "math_utils.h"

int square(int n) {
    return n * n;
}
```

main.c

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    printf("Square of 5: %d\n", square(5));
    return 0;
}
```

Compile:

```
gcc main.c math_utils.c -o app
```

Output:

```
Square of 5: 25
```

Why It Matters

Modules turn your code into small, understandable pieces. Each file does one job well. You don't need to scroll endlessly, just open the file you need.

This is how real-world C projects are built, one clean module at a time.

Try It Yourself

1. Create a `math_utils` module with `add`, `subtract`, and `multiply` functions.
2. Create a `student` module with a `Student` struct and a `print_student` function.
3. Write a `main.c` file that uses both modules.
4. Add include guards to your headers.
5. Compile everything together, see how clean your program feels.

Once you learn modular design, your C programs go from "long scripts" to organized systems, easy to grow, debug, and share.

70. Splitting Code into `.c` and `.h` Files

Now that you understand what modules are, let's look more closely at how to split your code properly into `.c` and `.h` files. This is one of the most important skills in C programming, it keeps your code organized, readable, and scalable.

You've already seen examples with small modules. Here, you'll learn a clear step-by-step process you can follow every time you make a new module.

70.1 Why Split Files

In small programs, you can write everything in one file. But as soon as your code grows, it becomes confusing, functions get mixed together, and managing them becomes a headache.

By splitting code:

- You separate logic from declarations
- You avoid repetition (declare once, use everywhere)
- You make your program easier to maintain

In short, `.h` tells the world *what exists*, `.c` shows *how it works*.

70.2 What Goes in a `.h` File

A header file (`.h`) contains declarations only. These are like *promises* to the compiler: "I'll define this somewhere."

Put in your `.h` file:

- Function declarations (prototypes)
- `typedef`s and `struct`s
- `#define` constants

- `enum` definitions
- External variable declarations (if needed)

Never put function definitions or global variables here.

Example (`math_utils.h`):

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int square(int n);

#endif
```

The `#ifndef` / `#define` / `#endif` trio is called a header guard. It prevents errors if this file is included multiple times.

70.3 What Goes in a `.c` File

A `.c` file contains the definitions, the actual code that does the work.

Example (`math_utils.c`):

```
#include "math_utils.h"

int add(int a, int b) {
    return a + b;
}

int square(int n) {
    return n * n;
}
```

Notice how the `.c` file includes its own header. That way, if you change a function's signature in the `.h` file, the compiler will catch mismatches immediately.

70.4 Using the Module in `main.c`

Your main program includes the header and calls the functions.

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    int x = 3, y = 4;
    printf("%d + %d = %d\n", x, y, add(x, y));
    printf("%d squared = %d\n", x, square(x));
    return 0;
}
```

Now all three files work together.

70.5 How to Compile Multiple Files

You can compile everything in one step:

```
gcc main.c math_utils.c -o app
```

Or separately:

```
gcc -c math_utils.c   // produces math_utils.o
gcc -c main.c         // produces main.o
gcc main.o math_utils.o -o app
```

The `.o` files are object files, intermediate steps before linking.

This approach is helpful for bigger projects, where only changed files need recompilation.

70.6 Sharing Structures Between Files

If multiple files need the same structure, define it in a header so everyone includes it.

`student.h`

```c
#ifndef STUDENT_H
#define STUDENT_H

typedef struct {
    int id;
    char name[50];
```

```
    float grade;
} Student;

void print_student(Student s);

#endif
```

student.c

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

void print_student(Student s) {
    printf("%d - %s (%.1f)\n", s.id, s.name, s.grade);
}
```

main.c

```
#include "student.h"
#include <string.h>

int main(void) {
    Student s = {1, "Alice", 95.5};
    print_student(s);
    return 0;
}
```

70.7 Keeping Functions Private

Not every function needs to be public. If it's only used inside one file, make it static:

```
static void log_message(void) {
    printf("Debug info\n");
}
```

Static functions are hidden from other files. This keeps your module's interface small and clear.

70.8 Including Headers in the Right Order

A good habit is to include headers in this order:

1. The module's own header (`"file.h"`)
2. Other project headers
3. Standard library headers (`<stdio.h>`, `<stdlib.h>`)

Example:

```c
#include "math_utils.h"
#include "student.h"
#include <stdio.h>
```

This helps the compiler catch missing dependencies early.

70.9 Folder Organization

For small projects, everything can live in one folder. For larger ones, separate by purpose:

```
/src
    main.c
    math_utils.c
    student.c
/include
    math_utils.h
    student.h
```

Then compile with include paths:

```
gcc src/*.c -I include -o app
```

The `-I` flag tells the compiler where to find header files.

70.10 Tiny Code Example

math_utils.h

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int cube(int n);

#endif
```

math_utils.c

```c
#include "math_utils.h"

int cube(int n) {
    return n * n * n;
}
```

main.c

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    int n = 3;
    printf("%d cubed = %d\n", n, cube(n));
    return 0;
}
```

Compile:

```
gcc main.c math_utils.c -o app
```

Output:

```
3 cubed = 27
```

Why It Matters

Splitting code into `.c` and `.h` files is a professional habit. It makes your code modular, reusable, and easy to scale. Every serious C project, from operating systems to libraries, follows this pattern.

Once you get used to it, you'll never want to cram everything into one file again.

Try It Yourself

1. Create a `geometry.h` and `geometry.c` with functions `area_circle` and `area_square`.
2. Include `geometry.h` in `main.c` and print both areas.
3. Add a second module `converter.h` with `to_fahrenheit()` and `to_celsius()`.
4. Use both modules together.
5. Recompile only the files you change.

You're now building multi-file C programs, a big step toward professional software development.

Chapter 8. The Power of the Processor

71. What Is the Preprocessor

Before your C code is turned into a program, it passes through a special stage called the preprocessor. Think of it as a helper that prepares your code before the compiler starts its real work.

This step might seem invisible, but it's powerful. It can insert code, replace text, and control which parts of your program are even compiled, all before a single line is turned into machine instructions.

Let's see how it works.

71.1 The Compilation Pipeline

When you compile a C program, it doesn't happen all at once. There are four stages:

1. Preprocessing, handles `#include`, `#define`, and `#if`
2. Compilation, translates C into assembly
3. Assembly, turns assembly into machine code
4. Linking, connects everything into a final executable

The preprocessor is step one, it's like a text editor that edits your code automatically before compiling.

71.2 What the Preprocessor Does

The preprocessor reads your source file and performs directives, commands that start with `#`. These aren't normal C statements; they're special instructions to the preprocessor.

Common examples:

- `#include`, insert another file's contents
- `#define`, create a constant or macro
- `#if`, `#ifdef`, `#endif`, control what code is included

It's like giving your code a quick makeover before sending it off to the compiler.

71.3 Example: Expanding Includes

If you write:

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

The preprocessor replaces that line with the entire contents of `stdio.h`. This is how your program learns about `printf` and other standard functions.

That means after preprocessing, your code is much bigger, but you never see it unless you ask for it.

You can check it by running:

```
gcc -E main.c
```

This shows you the preprocessed output, all macros expanded, all includes inserted.

71.4 Example: Using #define

With `#define`, you can create symbolic names or macros:

```
#define PI 3.14159
```

Now, whenever the preprocessor sees `PI`, it replaces it with `3.14159`. It's simple text replacement, not a variable, just a shortcut.

You'll use `#define` for constants, compile-time flags, or small helper macros (we'll explore more in Section 73).

71.5 Example: Conditional Compilation

You can also include or skip parts of your program depending on certain conditions.

```
#define DEBUG 1

#if DEBUG
    printf("Debug mode on\n");
#endif
```

If `DEBUG` is defined, the code inside runs; if not, it's ignored entirely. This is useful for debugging or platform-specific code.

71.6 Preprocessor vs Compiler

It's important to remember: The preprocessor is not the compiler, it doesn't check syntax, types, or logic.

It only rewrites text. The compiler only sees the *final result* after preprocessing.

So if you have:

```
#define PI 3.14
float area = PI * r * r;
```

The compiler actually sees:

```
float area = 3.14 * r * r;
```

71.7 Visual Picture

You can imagine the preprocessor as a "recipe preparer":

- It reads your recipe (code)
- Substitutes ingredients (macros)
- Adds missing steps (includes)
- Removes parts you don't need (conditionals)

Only after all that does the compiler step in to cook the final dish (binary).

71.8 Tiny Code Example

```c
#include <stdio.h>
#define NAME "Alice"
#define TIMES 3

int main(void) {
#if TIMES > 2
    printf("Hello, %s! You're learning C!\n", NAME);
#endif
    return 0;
}
```

Output:

```
Hello, Alice! You're learning C!
```

If you changed `#define TIMES 1`, that line wouldn't even be compiled.

71.9 Common Mistakes

1. Forgetting `#`, directives must start with it.
2. Thinking `#define` makes a variable, it doesn't! It's just text substitution.
3. Overusing macros when normal variables or functions would be better.
4. Missing `#endif` after a conditional block.

We'll explore each in detail soon.

71.10 Why It Matters

The preprocessor is your first gatekeeper, it controls what the compiler actually sees. Once you understand it, you'll know how to:

- Split code into multiple files
- Use constants and macros effectively
- Include only what you need
- Build code that adapts to different systems

This gives you flexibility and power, and helps you debug smarter, too.

Try It Yourself

1. Write a simple program with `#define NAME "World"` and `printf("Hello, %s!\n", NAME)`.
2. Use `#if` and `#endif` to print a message only when `DEBUG` is defined.
3. Add two `#include` headers and run `gcc -E` to see what preprocessing does.
4. Experiment by defining your own constant with `#define`.
5. Comment out an `#include` and see what breaks, you'll learn why includes matter!

Once you get comfortable with the preprocessor, you'll see it as your first teammate in every C project, preparing your code before the compiler even begins.

72. `#include` and Header Guards

You've already seen `#include` a few times, it's one of the most common preprocessor directives in C. It's how you bring code from one file into another, like borrowing tools from a shared toolbox.

In this section, you'll learn exactly how `#include` works, why we use it, and how header guards keep your code safe from duplication.

72.1 What `#include` Does

The `#include` directive literally copies and pastes the content of another file into your source file, *before* compilation begins.

For example:

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

When the preprocessor runs, it replaces this line with the entire text of `stdio.h`. That's how your program learns about `printf`, `scanf`, and other standard library functions.

It's as if you typed out all the contents of `stdio.h` yourself, but thankfully, you don't have to!

72.2 Two Kinds of Includes

There are two styles of `#include`:

1. Angle brackets `<...>`

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

Used for system headers (from the compiler or standard library). The compiler searches system directories for the file.

2. Double quotes `"..."`

```
#include "math_utils.h"
```

Used for your own project headers. The compiler looks in the current directory first, then system paths.

If your program includes its own files, always use quotes.

72.3 Why We Use Header Files

Header files (`.h`) let you share declarations, functions, structures, constants, across multiple `.c` files.

Instead of repeating code everywhere, you declare things once in a header and include it wherever needed.

Example:

`math_utils.h`

```
int add(int a, int b);
```

`main.c`

```
#include "math_utils.h"
int main(void) {
    printf("%d\n", add(3, 4));
}
```

`math_utils.c`

```
int add(int a, int b) {
    return a + b;
}
```

All files share the same declaration from the header, no duplication.

72.4 The Problem: Multiple Inclusion

Sometimes the same header can be included more than once, directly or indirectly. That causes redefinition errors during compilation.

For example:

```
#include "math_utils.h"
#include "geometry.h"  // geometry.h also includes math_utils.h
```

Now `math_utils.h` is included twice, and the compiler complains.

To fix this, C programmers use header guards.

72.5 What Are Header Guards

A header guard is a simple pattern that prevents a file from being included more than once.

At the top of your header:

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
```

At the bottom:

```
#endif
```

Together, these lines mean:

- If `MATH_UTILS_H` is not yet defined, define it and include this file.
- If it's already defined, skip it, because we've seen this file before.

So the full header looks like:

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);

#endif
```

This simple trick prevents duplicate definitions.

72.6 How It Works

Let's walk through it:

1. The first time the file is included, `MATH_UTILS_H` is not defined → file contents are added.
2. The preprocessor defines `MATH_UTILS_H`.
3. The second time, `MATH_UTILS_H` is defined → the preprocessor skips the file.

So no matter how many times it's included, it's only processed once.

72.7 Choosing a Guard Name

The name after `#ifndef` should be unique, usually based on the file name:

- `math_utils.h` → `MATH_UTILS_H`
- `student_record.h` → `STUDENT_RECORD_H`

Avoid generic names like `HEADER_H`, they might clash with others.

72.8 Example with Header Guard

`student.h`

```
#ifndef STUDENT_H
#define STUDENT_H

typedef struct {
    int id;
    char name[50];
} Student;

void print_student(Student s);

#endif
```

`main.c`

```
#include <stdio.h>
#include "student.h"
#include "student.h"  // accidentally included twice

int main(void) {
```

```
    Student s = {1, "Alice"};
    print_student(s);
    return 0;
}
```

student.c

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

void print_student(Student s) {
    printf("%d - %s\n", s.id, s.name);
}
```

Even though it's included twice, the header guard protects you.

72.9 Tiny Code Example

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

int main(void) {
    printf("3 + 4 = %d\n", add(3, 4));
    return 0;
}
```

math_utils.h

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
int add(int a, int b);
#endif
```

math_utils.c

```
#include "math_utils.h"
int add(int a, int b) { return a + b; }
```

Compile:

```
gcc main.c math_utils.c -o app
```

Output:

```
3 + 4 = 7
```

Safe, simple, and modular.

72.10 Why It Matters

`#include` and header guards are how C keeps big programs organized. They let you:

- Share code safely across files
- Avoid redefinition errors
- Manage large projects with confidence

Every serious C program, from a simple calculator to a full operating system, uses header guards. They're part of writing professional, maintainable C.

Try It Yourself

1. Create a header file `greetings.h` with a function `void say_hello(void);`
2. Add a guard called `GREETINGS_H`.
3. Write `greetings.c` that defines `say_hello`.
4. Include it twice in `main.c`, it should still compile fine.
5. Try removing the guard to see what error appears.

Once you see how `#include` and header guards work together, you'll understand how large C programs stay neat, safe, and error-free.

73. Defining Macros with `#define`

In C, the `#define` directive lets you create macros, short, simple replacements that happen *before* your code is compiled. You can use them for constants, code shortcuts, and even small function-like snippets.

They're part of the preprocessor's job: before the compiler ever sees your code, the preprocessor replaces every macro with what it stands for.

Think of them as little "search and replace" helpers that make your code more readable and flexible.

73.1 What Is a Macro

A macro is a name that stands for something else. It doesn't take up memory or store a value, it's just text substitution.

Basic form:

```
#define NAME value
```

Whenever the preprocessor sees `NAME`, it replaces it with `value`.

Example:

```
#define PI 3.14159
#define MAX_SIZE 100
```

Now `PI` and `MAX_SIZE` are symbolic constants. If you write:

```
float area = PI * r * r;
```

the preprocessor changes it to:

```
float area = 3.14159 * r * r;
```

You don't need semicolons or an equals sign, `#define` is not a statement, it's a directive.

73.2 Why Use Macros for Constants

Macros make code easier to read and maintain.

If you ever want to change a value, you change it once:

```
#define TAX_RATE 0.15
```

Then use `TAX_RATE` everywhere. No need to hunt down every `0.15` in your code.

This also reduces mistakes, it's clear what each constant means.

73.3 No Semicolon Needed

A common beginner mistake is adding a semicolon:

```c
#define PI 3.14; //   Wrong
```

That semicolon becomes part of the replacement, which breaks your code.

Correct version:

```c
#define PI 3.14  //   Right
```

Remember: `#define` is a replacement rule, not a statement.

73.4 Macros Are Not Variables

Macros don't have types or addresses. You can't use `&PI` or `scanf("%f", &PI)`, that doesn't make sense.

If you need a modifiable value, use a variable:

```c
float pi = 3.14;
```

If it's a fixed constant, `#define` is fine, or better yet, use `const` (we'll compare soon).

73.5 Function-like Macros

You can also make macros that behave like tiny functions:

```c
#define SQUARE(x) ((x) * (x))
```

When you write:

```c
int n = SQUARE(5);
```

the preprocessor turns it into:

```c
int n = ((5) * (5));
```

This happens before compilation, there's no function call at runtime, just a direct substitution.

73.6 Be Careful with Parentheses

Macros don't understand math, they're just text substitution. So always use parentheses to avoid surprises.

Example:

```
#define DOUBLE(x) x + x  //  Dangerous
int a = 3 * DOUBLE(2);  // expands to 3 * 2 + 2 → 8, not 12
```

Fix:

```
#define DOUBLE(x) ((x) + (x))  //  Safe
```

Now it expands to `3 * ((2) + (2)) → 12`.

This is one of the most important macro rules: wrap everything in parentheses.

73.7 Multi-line Macros

You can write macros that span multiple lines using a backslash \ at the end of each line:

```
#define DEBUG_LOG(msg) \
    printf("Debug: %s (line %d)\n", msg, __LINE__);
```

This macro prints a message along with the current line number, useful for quick debugging.

73.8 Built-in Macro Helpers

C provides special predefined macros you can use:

- `__FILE__` → current file name
- `__LINE__` → current line number
- `__DATE__` → date of compilation
- `__TIME__` → time of compilation

Example:

```
printf("Compiled from %s at line %d\n", __FILE__, __LINE__);
```

These are great for debugging or logging.

73.9 Macros vs `const`

In modern C, `const` variables are often better than macros for constants:

```
const float PI = 3.14159;
```

Why?

- `const` has a type (safer)
- Errors are easier to debug
- Works with debuggers and type checkers

Use `#define` when:

- You need conditional compilation
- You want a function-like macro
- You're defining something not tied to a type

Otherwise, prefer `const` for plain constants.

73.10 Tiny Code Example

```c
#include <stdio.h>

#define PI 3.14159
#define SQUARE(x) ((x) * (x))
#define AREA_CIRCLE(r) (PI * SQUARE(r))

int main(void) {
    float r = 2.0;
    printf("Radius: %.2f\n", r);
    printf("Area: %.2f\n", AREA_CIRCLE(r));
    return 0;
}
```

Output:

```
Radius: 2.00
Area: 12.57
```

The macros expand before compilation:

```
AREA_CIRCLE(r) → (3.14159 * ((r) * (r)))
```

No function calls, just substitution.

Why It Matters

Macros are like lightweight tools that make your code flexible and expressive. You can define constants, create shortcuts, and even write mini-functions, all handled before your code compiles.

But remember, they're text replacements, not smart code. Use parentheses, keep them simple, and prefer `const` when you need real variables.

Try It Yourself

1. Define `PI` and `GRAVITY` with `#define` and print them.
2. Write a macro `CUBE(x)` that calculates `x * x * x`.
3. Write a macro `MAX(a, b)` that returns the larger of two numbers.
4. Add a `DEBUG_LOG(msg)` macro using `__FILE__` and `__LINE__`.
5. Compare a `#define` constant with a `const` variable, notice the difference.

Once you master `#define`, you'll see it everywhere, in headers, libraries, and your own reusable code.

74. Working with Paths and Filenames

Whenever you include a header file or open a file in C, you're dealing with paths, addresses that tell your program where to look. Getting paths right is important because your compiler and your program both need to know exactly where files live.

Let's break it down step by step so it feels simple and natural.

74.1 What Is a Path

A path is just the *location* of a file in your system. When you tell C to include or open something, you're really saying,

> "Go find this file at this address."

There are two main kinds of paths:

- Absolute path, starts from the root (like `/home/user/project/math_utils.h`)
- Relative path, starts from where your program is (like `./math_utils.h` or `../include/math_utils.h`)

Think of it like giving directions:

- Absolute path → the full address.
- Relative path → directions from where you currently are.

74.2 Paths in `#include`

When you use `#include`, you're asking the preprocessor to pull another file into your source code. C supports two ways to specify paths here:

1. Angle brackets `<...>` for system or library headers

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

   This tells the compiler,

   > "Search the system include directories."

2. Double quotes `"..."` for your own files

   ```
   #include "math_utils.h"
   ```

   This tells the compiler,

   > "Look in the current folder first, then in the system directories."

74.3 Including Files from Subfolders

If your project grows, you'll likely organize it into folders:

```
project/
   src/
       main.c
   include/
       math_utils.h
```

From `main.c`, you can include the header using a relative path:

```
#include "../include/math_utils.h"
```

Or you can add `include/` to your compiler's search path:

```
gcc src/main.c -I include -o app
```

Now you can simply write:

```
#include "math_utils.h"
```

The `-I` option tells the compiler,

> "Also look inside this directory for headers."

74.4 Why Folder Organization Matters

Separating headers and source files makes projects easier to manage. A common layout looks like this:

```
/project
    /include    → header files (.h)
    /src        → source files (.c)
    /build      → compiled output
```

With this setup:

- All your interfaces live in `/include`
- All your logic lives in `/src`
- You include headers using `-I include`

Clean and predictable, perfect for bigger programs.

74.5 Common Relative Path Symbols

In file paths:

- `.` means "current directory"
- `..` means "one directory up"

Examples:

```
#include "../common/util.h"  // go up one level, then into common/
#include "./math_utils.h"    // same directory
```

You can use these in both `#include` and file I/O.

74.6 Paths in File I/O

When you open files in C using `fopen`, paths matter there too.

```
FILE *f = fopen("data.txt", "r");
```

This looks for `data.txt` in the current working directory, usually where you run your program from, not necessarily where the source code is.

If your file is elsewhere, use a relative or absolute path:

```
FILE *f = fopen("../resources/data.txt", "r");
```

If the file can't be found, `fopen` returns `NULL`. Always check before using it:

```
if (f == NULL) {
    perror("Could not open file");
    return 1;
}
```

74.7 Avoid Hardcoding Absolute Paths

It's tempting to write something like:

```
fopen("/Users/alice/Desktop/data.txt", "r");
```

But that only works on *your* computer. If someone else runs it, their path will be different.

Prefer relative paths (like `"../data.txt"`) or configuration options so your code is portable.

74.8 Using Paths in Larger Projects

As projects grow, you'll often:

- Put headers in an `include/` folder
- Add `-I include` to compiler commands
- Put sources in `src/`
- Include headers with simple names like `"math_utils.h"`

You can even have nested includes like:

```
#include "geometry/circle.h"
```

if your folder structure is:

```
include/
    geometry/
        circle.h
```

C's preprocessor will treat the `/` as a path separator, no special syntax needed.

74.9 Tiny Code Example

```
project/
   include/
      math_utils.h
   src/
       main.c
       math_utils.c
```

include/math_utils.h

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);

#endif
```

src/math_utils.c

```c
#include "math_utils.h"
int add(int a, int b) { return a + b; }
```

src/main.c

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    printf("3 + 4 = %d\n", add(3, 4));
    return 0;
}
```

Compile from the project root:

```
gcc src/*.c -I include -o app
```

Output:

```
3 + 4 = 7
```

Your program finds headers correctly and stays neatly organized.

74.10 Why It Matters

Working with paths is about clarity and portability. When your project is well organized and your includes are clean:

- You always know where things live
- The compiler always knows where to look
- Your code runs smoothly on any system

Once you start managing multi-file projects, clean paths become a habit, and your programs start to look truly professional.

Try It Yourself

1. Create a folder `project/` with subfolders `src/` and `include/`.
2. Write `math_utils.h` and `math_utils.c` and include them in `main.c`.
3. Compile using `-I include`.
4. Add another folder `geometry/` inside `include/` and make a `circle.h`.
5. Try including it using `#include "geometry/circle.h"`.

Once you're comfortable with paths and filenames, you'll never feel lost in multi-file C programs again.

75. Conditional Compilation with `#if` and `#ifdef`

Sometimes, you want your program to include or skip certain parts of code, depending on the situation. Maybe you're adding debugging messages, or you need to compile differently for Windows vs Linux.

That's where conditional compilation comes in. It lets you control what gets compiled, all through the preprocessor.

75.1 What Conditional Compilation Means

Normally, the compiler reads every line of your program. But with conditional compilation, you can tell it:

> "Only compile this part if a certain condition is true."

That's handled by preprocessor directives like:

- `#if`
- `#ifdef`
- `#ifndef`
- `#else`
- `#elif`
- `#endif`

Think of them like `if` statements for the compiler itself, deciding which lines *exist* in your final program.

75.2 The Basic `#if`

You can test simple conditions using `#if` and `#endif`:

```
#define DEBUG 1

#if DEBUG
    printf("Debugging is on!\n");
#endif
```

If `DEBUG` is defined as `1` (true), the code inside runs. If it's `0`, the preprocessor skips it entirely.

You can also combine conditions:

```
#if DEBUG && VERBOSE
    printf("Extra details...\n");
#endif
```

The preprocessor understands basic arithmetic and logic:

- `==`, `!=`, `<`, `>`, `&&`, `||`, `!`
- Numbers only, not variables or functions

75.3 `#ifdef` and `#ifndef`

These two check whether a macro exists, not its value.

- `#ifdef` means "if defined"
- `#ifndef` means "if not defined"

Examples:

```
#ifdef DEBUG
    printf("Debug mode enabled\n");
#endif
```

```
#ifndef DEBUG
    #define DEBUG 1
#endif
```

These are great for feature flags or header guards.

75.4 Combining with `#else` and `#elif`

You can also give alternatives:

```
#define OS_WINDOWS 1

#if OS_WINDOWS
    printf("Running on Windows\n");
#else
    printf("Running on another OS\n");
#endif
```

Or chain conditions:

```
#define OS 2

#if OS == 1
    printf("Windows\n");
#elif OS == 2
    printf("Linux\n");
#else
    printf("Other\n");
#endif
```

This way, your program adapts automatically depending on what's defined.

75.5 Example: Debug Mode

```
#include <stdio.h>

#define DEBUG 1

int main(void) {
#if DEBUG
    printf("Debug info: starting program...\n");
#endif
    printf("Hello, world!\n");
    return 0;
}
```

If you set `DEBUG` to `0`, the debug message disappears completely from the compiled program. It's as if it never existed.

75.6 Defining Macros from the Command Line

You don't even have to edit your code, you can define macros right from the compiler command:

```
gcc main.c -DDEBUG=1 -o app
```

Now `DEBUG` is defined for this build. You can toggle features just by changing the build command, handy for testing or releases.

75.7 Example: Cross-Platform Code

You can use macros to handle different systems:

```c
#include <stdio.h>

int main(void) {
#ifdef _WIN32
    printf("Running on Windows\n");
#elif __linux__
    printf("Running on Linux\n");
#else
    printf("Unknown system\n");
#endif
    return 0;
}
```

Compilers often define platform macros automatically. So your code can adapt to the environment without manual changes.

75.8 Conditional Function Definitions

Sometimes, you may want to include different functions for different cases:

```c
#if DEBUG
void log_message(const char *msg) {
    printf("LOG: %s\n", msg);
}
#else
void log_message(const char *msg) {
```

```
    // do nothing in release mode
}
#endif
```

This keeps your interface consistent (`log_message` always exists), but its behavior depends on the build.

75.9 Common Pitfalls

1. Forgetting `#endif`, always close your condition.
2. Mixing preprocessor with runtime logic, remember, these are checked *before compilation*.
3. Overcomplicating, keep conditions clear and short.
4. No semicolons, these are not C statements, so don't add `;` after them.

75.10 Tiny Code Example

```c
#include <stdio.h>

#define DEBUG 1
#define VERSION 2

int main(void) {
#if DEBUG
    printf("Debug mode active\n");
#endif

#if VERSION == 1
    printf("Version 1 features\n");
#elif VERSION == 2
    printf("Version 2 features\n");
#else
    printf("Unknown version\n");
#endif

    return 0;
}
```

Change the macro values and recompile, watch how the output changes.

Why It Matters

Conditional compilation gives you control. You decide what's built and what's ignored, all without touching the main logic.

It's essential for:

- Debugging
- Cross-platform builds
- Optional features
- Release vs development modes

Once you understand it, you can make one codebase that adapts to many situations.

Try It Yourself

1. Define `DEBUG` and print messages only when it's on.
2. Create `VERSION` and switch between "Lite" and "Pro" features.
3. Use `#ifndef` to add a header guard.
4. Define a macro from the command line (`-DDEBUG=1`).
5. Build once with `DEBUG=1` and again with `DEBUG=0`, compare outputs.

With these tools, you're now ready to write flexible, configurable C programs that fit every build and platform.

76. Function-like Macros

So far, you've seen `#define` used for simple constants, like

```
#define PI 3.14159
```

But macros can do more than stand in for numbers, they can act like tiny functions. These are called function-like macros, and they're a powerful part of C's preprocessor.

They don't actually create a function. Instead, they perform text substitution, replacing code before compilation. Used carefully, they can make your programs shorter and more flexible.

76.1 What Is a Function-like Macro

A function-like macro looks like a function, but it's really just a shortcut:

```c
#define SQUARE(x) ((x) * (x))
```

When the preprocessor sees `SQUARE(5)`, it replaces it with:

```c
((5) * (5))
```

This happens before the compiler runs, so there's no function call overhead, just plain code after expansion.

76.2 Why Use Function-like Macros

They can:

- Save typing for short, repeated operations
- Make simple inline code clearer
- Avoid runtime overhead

For example:

```c
#define CUBE(x) ((x) * (x) * (x))
```

You can now write `CUBE(n)` instead of `n * n * n`.

But because macros are text substitutions, not real functions, you need to be careful with how you write them.

76.3 Always Use Parentheses

This is the golden rule of macros. Wrap every parameter and the whole expression in parentheses.

Without parentheses, operator precedence can break your code.

Bad:

```c
#define DOUBLE(x) x + x
int y = 3 * DOUBLE(2);  // expands to 3 * 2 + 2 = 8
```

Good:

```c
#define DOUBLE(x) ((x) + (x))
int y = 3 * DOUBLE(2);  // expands to 3 * ((2) + (2)) = 12
```

The parentheses keep operations safe and predictable.

76.4 Macros with Multiple Arguments

Macros can take more than one argument:

```c
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Now `MAX(3, 5)` becomes `((3) > (5) ? (3) : (5))`, which evaluates to `5`.

You can use this anywhere a normal expression would fit.

76.5 Example: Area and Perimeter

Let's use macros to calculate shapes:

```c
#define AREA_RECT(w, h) ((w) * (h))
#define PERIM_RECT(w, h) (2 * ((w) + (h)))
```

Then:

```c
int w = 5, h = 3;
printf("Area: %d, Perimeter: %d\n", AREA_RECT(w, h), PERIM_RECT(w, h));
```

Output:

```
Area: 15, Perimeter: 16
```

Simple, readable, and fast.

76.6 Beware of Side Effects

Because macros duplicate arguments, using them with functions or increments can cause surprises.

Example:

```c
#define SQUARE(x) ((x) * (x))
int n = 3;
int result = SQUARE(n++);  // expands to ((n++) * (n++))
```

This increments `n` twice, unexpected behavior! Avoid using macros with expressions that change values (like `++`, `--`, or function calls).

If you need safety, write a real function instead.

76.7 Macro Naming Conventions

To make macros stand out, many programmers use uppercase names:

```c
#define ABS(x) ((x) < 0 ? -(x) : (x))
```

This helps you quickly recognize them in code and avoid naming conflicts with variables or functions.

76.8 Debugging Macro Expansions

You can inspect what macros expand to by using:

```
gcc -E file.c
```

This runs only the preprocessor and shows the expanded source. It's helpful when something behaves unexpectedly.

76.9 Tiny Code Example

```c
#include <stdio.h>

#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main(void) {
    int a = 4, b = 6;
    printf("SQUARE(%d) = %d\n", a, SQUARE(a));
    printf("MAX(%d, %d) = %d\n", a, b, MAX(a, b));
    return 0;
}
```

Output:

```
SQUARE(4) = 16
MAX(4, 6) = 6
```

Everything happens at compile time, no function calls, just expanded code.

76.10 When to Use (and When Not To)

Use function-like macros for:

- Small, simple, pure expressions
- Operations that won't cause side effects
- Cases where performance matters and inline code helps

Avoid them when:

- You need type checking
- You need complex logic
- You're calling functions or using `++` inside

If in doubt, use a `static inline` function instead, safer, type-checked, and just as fast.

Why It Matters

Function-like macros can make your code shorter and faster, but they also demand care.

With parentheses and caution, they're a handy tool. Once you understand their behavior, you'll know when a macro is helpful, and when a function is the better choice.

Try It Yourself

1. Write macros `CUBE(x)` and `ABS(x)`, test with negative numbers.
2. Write `MAX(a, b)` and `MIN(a, b)` and try different pairs.
3. Experiment with bad parentheses, see what goes wrong!
4. Try expanding your code with `gcc -E` to see macro replacements.
5. Compare a macro and a real function, see which is safer or clearer.

Soon you'll be writing macros like a pro, simple, safe, and easy to read.

77. Debugging with `#error` and `#warning`

Sometimes, your program might compile when it shouldn't, maybe a setting is wrong, a platform isn't supported, or a constant is missing. Wouldn't it be nice if the compiler could stop and tell you exactly what's wrong?

That's exactly what **`#error`** and **`#warning`** do. They're special preprocessor directives that let you display messages (and even halt compilation) when conditions aren't met.

Think of them as little safety guards that speak up before the compiler makes a mistake.

77.1 What Is `#error`

The **`#error`** directive stops compilation immediately and prints a custom message.

Syntax:

```
#error "Your message here"
```

When the preprocessor sees this line, it refuses to continue, it shows your message and exits with an error.

Example:

```
#if __STDC_VERSION__ < 202000L
#error "This program requires a modern C compiler"
#endif
```

If your compiler doesn't meet that version, you'll get:

```
error: This program requires a modern C compiler
```

The program won't compile until you fix the issue.

77.2 Why Use `#error`

You can use it to:

- Enforce minimum compiler versions
- Prevent unsupported builds
- Catch missing definitions or bad configurations
- Warn other developers when something's misused

It's a friendly way to fail early, better to stop at compile time than to run broken code.

77.3 Example: Required Macro

```
#ifndef DEBUG
#error "DEBUG must be defined (use -DDEBUG=1)"
#endif
```

If you forget to define `DEBUG`, compilation stops and reminds you how to fix it.

You can define it like this:

```
gcc main.c -DDEBUG=1 -o app
```

77.4 Example: Unsupported Platform

```
#if !defined(_WIN32) && !defined(__linux__)
#error "Unsupported platform: please compile on Windows or Linux"
#endif
```

If someone tries to compile on macOS (or another OS), the preprocessor politely refuses.

77.5 What Is `#warning`

`#warning` works like `#error`, but it doesn't stop compilation. It just prints a compiler warning message, useful when you want to inform, not block.

Syntax:

```
#warning "This feature is experimental"
```

Output:

```
warning: This feature is experimental
```

Your program will still compile and run, but you'll see the note.

77.6 Example: Version Notice

```
#if __STDC_VERSION__ < 201112L
#warning "Consider updating to C11 or newer"
#endif
```

This helps nudge users toward better compilers, without stopping their build.

77.7 Combining with Conditions

You can pair `#error` and `#warning` with any condition:

```
#define VERSION 2

#if VERSION == 1
#warning "Version 1 is outdated"
#elif VERSION > 3
#error "Version not supported"
#endif
```

Messages appear depending on what's defined.

77.8 Example: Safety Checks

```
#define MAX_USERS 0

#if MAX_USERS <= 0
#error "MAX_USERS must be greater than zero"
#endif
```

If a configuration is invalid, the build fails before it reaches the compiler.

This is a great habit, let the build process catch mistakes early.

77.9 Tiny Code Example

```c
#include <stdio.h>

#define MODE 2

#if MODE == 1
#warning "Mode 1 is for testing only"
#elif MODE > 3
#error "Invalid MODE setting"
#endif

int main(void) {
    printf("Running in mode %d\n", MODE);
    return 0;
}
```

Compile it and try different `MODE` values, see how messages change.

77.10 Best Practices

- Use `#error` for serious misconfigurations that must stop the build.
- Use `#warning` for friendly reminders or temporary notes.
- Always write clear, actionable messages, say *what went wrong* and *how to fix it*.
- Avoid spamming warnings; too many messages hide the important ones.

Why It Matters

`#error` and `#warning` give your code a voice during compilation. They turn silent mistakes into visible messages, helping you and others fix issues early.

In bigger projects, these checks become part of a build safety net — so your program never compiles in an unsupported or broken state.

Try It Yourself

1. Add a `#error` if `DEBUG` isn't defined.
2. Add a `#warning` if you're using an old C version.
3. Add a `#error` when `MAX_SIZE` is over 1000.
4. Define macros from the command line (`-DDEBUG=1`) and test again.
5. Practice writing helpful messages, imagine another beginner reading them.

Once you start using these directives, your compiler becomes more than a tool — it becomes a teammate that tells you when something's not right.

78. Built-in Macros: `__FILE__`, `__LINE__`, and Friends

C gives you a handful of built-in macros that always exist, no matter what you write. They're like little bookmarks the compiler fills in automatically, showing where and when your code was compiled.

You don't define them yourself. They're always there, ready to help with debugging, logging, and version tracking.

78.1 What Are Built-in Macros

Built-in macros are special names that start and end with double underscores (`__`). When the preprocessor runs, it replaces each one with useful information.

They're not variables or constants, they're compile-time values inserted directly into your code.

78.2 The Most Common Built-ins

Here are the ones you'll use most often:

Macro	Expands To	Description
`__FILE__`	A string with the current filename	Where this code lives
`__LINE__`	The current line number	Where this code appears
`__DATE__`	The compilation date	When it was compiled
`__TIME__`	The compilation time	What time it was compiled
`__STDC__`	`1` if you're using a standard C compiler	Confirms compliance
`__STDC_VERSION__`	Version number (like `201112L` for C11)	Which C standard

78.3 Example: Printing File and Line

Here's a simple example that prints where a line comes from:

```c
#include <stdio.h>

int main(void) {
    printf("This code is from file: %s, line: %d\n", __FILE__, __LINE__);
    return 0;
}
```

Output:

```
This code is from file: main.c, line: 4
```

If you move the `printf` to a different line, the number changes automatically.

78.4 Example: Compilation Info

You can show when your program was built:

```c
#include <stdio.h>

int main(void) {
    printf("Compiled on %s at %s\n", __DATE__, __TIME__);
    return 0;
}
```

Output might look like:

```
Compiled on Oct 2 2025 at 10:42:15
```

This is useful for version tracking, you'll always know which build you're running.

78.5 Example: Logging Helper

You can combine these macros to make helpful debug messages:

```c
#define LOG(msg) \
    printf("[%s:%d] %s\n", __FILE__, __LINE__, msg)
```

Then use it:

```c
int main(void) {
    LOG("Starting program");
    LOG("Initialization complete");
    return 0;
}
```

Output:

```
[main.c:6] Starting program
[main.c:7] Initialization complete
```

Now every log tells you exactly where it came from, no guessing!

78.6 Checking Compiler Version

You can use `__STDC_VERSION__` to check which C standard is being used:

```c
#if __STDC_VERSION__ >= 201112L
    printf("Using C11 or newer\n");
#else
    printf("Older C version detected\n");
#endif
```

This helps you write portable code that adjusts to the compiler.

78.7 Using Macros for Debug Builds

You can mix built-ins with `#ifdef DEBUG` to show info only in debug mode:

```c
#ifdef DEBUG
#define DEBUG_LOG(msg) \
    printf("[DEBUG] %s:%d, %s\n", __FILE__, __LINE__, msg)
#else
#define DEBUG_LOG(msg)
#endif
```

Now:

```c
int main(void) {
    DEBUG_LOG("Running diagnostics...");
    return 0;
}
```

If you compile with `-DDEBUG`, the logs appear; otherwise, they vanish.

78.8 Why Double Underscores

Names like `__FILE__` and `__LINE__` are reserved by the C standard. That's why user-defined macros shouldn't start or end with underscores, to avoid conflicts with the language itself.

Stick to uppercase names without double underscores for your own macros:

```c
#define MAX_SIZE 100
```

78.9 Tiny Code Example

```c
#include <stdio.h>

int main(void) {
    printf("File: %s\n", __FILE__);
    printf("Line: %d\n", __LINE__);
    printf("Compiled on: %s at %s\n", __DATE__, __TIME__);
    printf("C Standard: %ld\n", __STDC_VERSION__);
    return 0;
}
```

Output:

```
File: main.c
Line: 5
Compiled on: Oct 2 2025 at 10:42:15
C Standard: 202000
```

Now your program tells its own story, where, when, and how it was built.

78.10 Why It Matters

Built-in macros are small but powerful. They let your code:

- Report its source
- Show compile-time info
- Log helpful debug messages
- Adjust to the compiler version

You'll use them most in debugging and logging, especially in bigger projects.

Try It Yourself

1. Print `__FILE__` and `__LINE__` in your main function.
2. Add a `LOG` macro that includes file and line automatically.
3. Display `__DATE__` and `__TIME__` at program startup.
4. Use `__STDC_VERSION__` to show your C version.
5. Try moving your log statements around, watch the line numbers change.

With these macros, your code becomes self-aware, it knows when, where, and how it was built, and it can tell you too!

79. The Compilation Pipeline: Preprocess → Compile → Link

Every time you run `gcc` (or any C compiler), a lot happens behind the scenes. Your code doesn't jump straight to an executable, it goes through several distinct stages, each transforming your program step by step.

Understanding these stages helps you debug build errors, organize large projects, and control your builds like a pro.

Let's walk through the journey your code takes from `.c` files to a working program.

79.1 The Big Picture

Think of compilation as a four-step pipeline:

1. Preprocessing, expands macros, includes headers, and cleans up code
2. Compilation, turns C code into assembly
3. Assembly, converts assembly into machine code (object files)
4. Linking, combines all object files into a final executable

Every time you type something like:

```
gcc main.c -o app
```

these four steps happen automatically.

We'll unpack each one so you can see exactly what's going on.

79.2 Step 1: Preprocessing

The preprocessor handles all lines starting with `#`, like:

- `#include`, inserts header files
- `#define`, expands macros
- `#if`, `#ifdef`, includes or skips code conditionally

At this stage, your code is pure text manipulation. No syntax checking yet, it's just assembling the full code.

You can see the result using:

```
gcc -E main.c
```

This outputs the preprocessed source, where all includes and macros have been expanded.

79.3 Step 2: Compilation

Next, the compiler translates your C source into assembly language, a lower-level representation still readable by humans.

It checks for:

- Syntax errors
- Type mismatches
- Undeclared variables
- Invalid expressions

You can stop after this step to inspect the assembly code:

```
gcc -S main.c
```

This produces `main.s`, a plain text file with assembly instructions.

79.4 Step 3: Assembly

Now the assembler takes that assembly (`.s`) file and converts it into machine code, creating an object file (`.o` or `.obj`).

Command:

```
gcc -c main.c
```

This skips linking and stops at the object file. Object files are binary, not human-readable, and contain compiled functions waiting to be linked.

If you have multiple source files, you'll get one `.o` per `.c`:

```
main.o
math_utils.o
student.o
```

79.5 Step 4: Linking

Finally, the linker takes all the object files and glues them together, along with any needed libraries (like `libc`), to form one executable.

For example:

```
gcc main.o math_utils.o -o app
```

The linker checks that:

- Every function call matches a defined function
- External symbols (like `printf`) are found in libraries

If something's missing, you'll see "undefined reference" errors, those come from this stage.

79.6 Why Separate Steps Matter

Splitting compilation into steps gives you flexibility:

- Faster builds, only recompile changed files
- Modular design, work on parts independently
- Custom control, inspect or optimize each stage

In big projects, tools like `make` automate this process, only rebuilding what's necessary.

79.7 Summary of Commands

Step	Description	Command	Output
1. Preprocess	Expand macros, includes	`gcc -E main.c`	Preprocessed text
2. Compile	Translate to assembly	`gcc -S main.c`	`main.s`
3. Assemble	Turn into machine code	`gcc -c main.c`	`main.o`
4. Link	Combine all objects	`gcc main.o -o app`	`app` executable

Usually, `gcc main.c -o app` runs all four in one go.

79.8 Example: Multi-file Build

```
project/
  main.c
  math_utils.c
  math_utils.h
```

Compile separately:

```
gcc -c main.c       // main.o
gcc -c math_utils.c // math_utils.o
```

Link together:

```
gcc main.o math_utils.o -o app
```

You now have a clean, modular build, if you change only `math_utils.c`, recompile just that file.

79.9 Visualizing the Pipeline

```
 main.c


[Preprocess]   → expanded source


[Compile]      → main.s (assembly)
```

```
[Assemble]     → main.o (object file)


[Link]         → app (executable)
```

Each step builds on the last, just like an assembly line.

79.10 Tiny Code Example

main.c

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    printf("3 + 4 = %d\n", add(3, 4));
    return 0;
}
```

math_utils.h

```c
int add(int a, int b);
```

math_utils.c

```c
int add(int a, int b) {
    return a + b;
}
```

Build it step by step:

```
gcc -c main.c
gcc -c math_utils.c
gcc main.o math_utils.o -o app
```

Run it:

```
./app
```

Output:

```
3 + 4 = 7
```

Why It Matters

Understanding the compilation pipeline turns you from a user of the compiler into a builder who controls it. You'll be able to:

- Debug build errors with confidence
- Rebuild only what's needed
- Organize large projects cleanly
- Appreciate how source code becomes machine code

Once you see how it all fits together, you'll never look at `gcc` the same way again.

Try It Yourself

1. Run `gcc -E main.c > expanded.c` and inspect the result.
2. Generate assembly with `gcc -S main.c` and open `main.s`.
3. Create object files using `-c` and link them manually.
4. Introduce an undefined function, see the linker error.
5. Rebuild after changing one file, notice how much faster it is!

By understanding each stage, you gain control over your builds, and that's a superpower every C programmer needs.

80. Balancing Macros and Functions

By now, you've seen that macros can act like quick shortcuts, they're handled by the preprocessor, before the compiler even sees your code. You've also written functions, which the compiler fully understands and checks for types and safety.

So when should you use a macro, and when should you use a function? This section helps you decide the right tool for the job.

80.1 Macros vs. Functions: The Core Difference

Let's start with what makes them different.

Feature	Macro (`#define`)	Function
Processed by	Preprocessor	Compiler
Type Checking	None	Full type checking
Parameters	Just text	Typed arguments
Overhead	None (inline code)	Slight (function call)
Safety	Risky if misused	Safe and predictable
Debugging	Harder	Easier

Macros are like *smart copy-paste shortcuts*. Functions are *real code blocks* that the compiler understands.

80.2 When to Use a Macro

Macros are good for small, simple, type-agnostic operations that don't cause side effects.

Use them when:

- You want a compile-time substitution
- You need something to work with any type
- You want performance with no function call

Example:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

This works for `int`, `float`, or even `char`, the compiler just substitutes text.

80.3 When to Use a Function

Functions are better when:

- You need type safety
- The logic is more than one line
- You want to debug easily
- You use side effects or changing values

Example:

```
int max_int(int a, int b) {
    return (a > b) ? a : b;
}
```

If you accidentally pass wrong types, the compiler warns you. Functions are also easier to read and maintain, especially for beginners.

80.4 Why Macros Can Be Risky

Macros don't check types or parentheses. They can expand in surprising ways.

Example:

```
#define SQUARE(x) (x * x)
int n = SQUARE(1 + 2);  // expands to (1 + 2 * 1 + 2) = 5, not 9
```

Fix with parentheses:

```
#define SQUARE(x) ((x) * (x))
```

Still, a function is safer:

```
int square(int x) { return x * x; }
```

The compiler won't let you misuse it.

80.5 Inline Functions: The Best of Both Worlds

Modern C lets you mark a function as inline:

```
inline int square(int x) { return x * x; }
```

This tells the compiler:

> "Replace this call with the function's code, if it's faster."

You get:

- Type checking
- Readable code
- Possible performance gain

It's like a safe macro, usually the best choice.

80.6 Example: Macro vs Function

Macro

```
#define CUBE(x) ((x) * (x) * (x))
```

Function

```
int cube(int x) {
    return x * x * x;
}
```

Usage

```
printf("%d\n", CUBE(2)); // 8
printf("%d\n", cube(2)); // 8
```

Both give the same result, but the function is safer and easier to debug.

80.7 Mixing Both: Constants and Helpers

Macros are still perfect for constants and compile-time options:

```
#define PI 3.14159
#define DEBUG 1
```

But for logic, especially if it's more than a single expression, prefer a function.

80.8 Example: Logging

Let's say you want a quick logging tool. You can combine a macro and a function safely:

```
#include <stdio.h>

inline void log_message(const char *msg) {
    printf("[LOG] %s\n", msg);
}

#ifdef DEBUG
#define LOG(msg) log_message(msg)
```

```
#else
#define LOG(msg)
#endif
```

When `DEBUG` is defined, `LOG()` calls the real function. When it's off, the macro removes the call entirely.

That's the balance: macro for control, function for behavior.

80.9 Tiny Code Example

```c
#include <stdio.h>

#define MAX(a, b) ((a) > (b) ? (a) : (b))

inline int square(int x) {
    return x * x;
}

int main(void) {
    int a = 3, b = 5;
    printf("Max of %d and %d is %d\n", a, b, MAX(a, b));
    printf("Square of %d is %d\n", a, square(a));
    return 0;
}
```

Output:

```
Max of 3 and 5 is 5
Square of 3 is 9
```

Here, both macro and function shine in their own way.

80.10 Quick Guidelines

Use a macro when:

- It's a simple one-line expression
- It works for multiple types
- You need compile-time substitution

Use a function when:

- You want type safety
- The code has logic or side effects
- You need to debug or step through it

Use `inline` when:

- You want the speed of macros
- But the safety of functions

Why It Matters

Choosing between macros and functions is part of learning to write clean, safe C code. Once you know the trade-offs, you'll make smart choices automatically, balancing speed, safety, and readability.

Try It Yourself

1. Write a macro `ABS(x)` and a function `abs_val(int x)`, test both.
2. Try `SQUARE(x)` as a macro and as an `inline` function, compare safety.
3. Add a `MAX(a, b)` macro, and see if it works with `float` too.
4. Intentionally break a macro, then fix it with parentheses.
5. Mix both in one program: macros for constants, functions for logic.

Once you practice both sides, you'll feel confident choosing the right tool every time.

Chapter 9. Files, tools, and concurrency

81. File I/O Basics: `fopen` and `fclose`

So far, everything your programs have done has lived in memory, once your program ends, all that data disappears. To make your programs more useful, you need a way to save information permanently. That's where files come in.

In C, you can read and write files just like you read and write variables, you just need to open a connection first. This process is called file I/O (input/output).

Let's start with the basics: opening and closing files.

81.1 Why Work with Files

Files let you store data between runs, like saving a score, a log, or a list of names. Once you can read and write files, you can:

- Save user data
- Process large text files
- Read configuration settings
- Generate reports or logs

In C, you control all of this manually, which means you get full power (and full responsibility).

81.2 The `FILE*` Pointer

C uses a special type called `FILE` to represent a file. You don't create one directly, instead, you ask C to open a file for you.

When you open a file, C gives you back a pointer to a `FILE` object:

```
FILE *fp;
```

This pointer works like a handle, it represents the open connection to your file.

81.3 Opening a File with `fopen`

To open a file, call `fopen`:

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

Here:

- `"data.txt"` is the file name (path)
- `"r"` means open for reading

If it succeeds, `fp` will point to the open file. If it fails (say the file doesn't exist), `fp` will be `NULL`.

Always check:

```
if (fp == NULL) {
    printf("Could not open file.\n");
    return 1;
}
```

81.4 File Open Modes

The second argument to `fopen` tells C how to open the file.

Mode	Meaning	Creates New File?
`"r"`	Read	No
`"w"`	Write (overwrite existing)	Yes
`"a"`	Append (add to end)	Yes
`"r+"`	Read and write	No
`"w+"`	Read and write (overwrite)	Yes
`"a+"`	Read and write (append)	Yes

If you open with `"w"`, be careful, it will erase existing content.

81.5 Closing a File with `fclose`

Once you're done, close the file to free resources:

```
fclose(fp);
```

If you forget to close files, you might lose data or run out of file handles.

Make it a habit:

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
    printf("Error opening file.\n");
    return 1;
}

// ... use file ...

fclose(fp);
```

81.6 Example: Opening and Closing

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("hello.txt", "w");

    if (fp == NULL) {
        printf("Failed to open file.\n");
        return 1;
    }

    printf("File opened successfully!\n");

    fclose(fp);
    printf("File closed.\n");

    return 0;
}
```

Run it, and you'll find a new file named `hello.txt` in your folder. Even if it's empty, the file was created and closed cleanly.

81.7 Check Before You Read or Write

You should always check the pointer before using it:

```c
FILE *fp = fopen("input.txt", "r");
if (fp == NULL) {
    printf("Error: cannot open input.txt\n");
    return 1;
}
```

If you try to use a `NULL` pointer, your program will crash. So this small check saves you a lot of debugging later.

81.8 File Paths and Locations

When you write `"data.txt"`, C looks in your current working directory, usually the same folder where you run your program.

To open a file elsewhere, use a relative or absolute path:

```
fopen("../resources/data.txt", "r");
fopen("/home/user/documents/data.txt", "r");
```

Use double backslashes on Windows:

```
fopen("C:\\Users\\Alice\\Desktop\\data.txt", "r");
```

81.9 Common Mistakes

1. Forgetting to close the file
2. Not checking `fopen` return value
3. Using wrong mode (`"r"` for reading non-existent file)
4. Writing to a read-only file
5. Mixing reading and writing modes incorrectly

These small checks make your file I/O safe and predictable.

81.10 Tiny Code Example

```
#include <stdio.h>

int main(void) {
    FILE *file = fopen("example.txt", "w");

    if (file == NULL) {
        printf("Failed to open file.\n");
        return 1;
    }

    fprintf(file, "Hello, file I/O!\n");
    fclose(file);

    printf("Done writing!\n");
    return 0;
}
```

Output in terminal:

```
Done writing!
```

Contents of `example.txt`:

```
Hello, file I/O!
```

Why It Matters

Opening and closing files is the first step in persistent programming. Once you know how to do this, you can start reading input files, saving logs, and building tools that work with real-world data.

Try It Yourself

1. Open a file called `notes.txt` in `"w"` mode and write one line.
2. Close it, then open it again in `"r"` mode and read it (coming up next).
3. Try opening a file that doesn't exist in `"r"` mode, watch for `NULL`.
4. Experiment with `"a"` mode, write multiple times and see what happens.
5. Always remember to `fclose` every opened file.

Once you're comfortable with `fopen` and `fclose`, you're ready to read and write data, and that's coming up next.

82. Reading and Writing Files

Now that you know how to open and close files, it's time to learn how to actually put data into them and read it back out. This is one of the most powerful skills in C, it's how programs remember things after they stop running.

Think of a file as a container of text or data. You open it, use special functions to read or write, and close it when you're done, just like handling a notebook.

82.1 Two Directions: Input and Output

File I/O is really just about two directions:

- Output → writing data *to* a file
- Input → reading data *from* a file

Each direction uses its own set of functions, but the workflow is always the same:

1. `fopen` the file
2. `fprintf` or `fscanf` (or similar)
3. `fclose` when finished

82.2 Writing to a File with `fprintf`

The simplest way to write text to a file is `fprintf`. It works just like `printf`, but writes to a file instead of the screen.

Example:

```
FILE *fp = fopen("output.txt", "w");
if (fp == NULL) {
    printf("Could not open file for writing.\n");
    return 1;
}

fprintf(fp, "Hello, file!\n");
fprintf(fp, "The answer is %d\n", 42);

fclose(fp);
```

After running this, open `output.txt`, you'll see:

```
Hello, file!
The answer is 42
```

You can write any format, integers, floats, strings, just like `printf`.

82.3 Appending to a File

If you open in `"a"` mode, your file keeps its old content, and new text is added to the end.

```
FILE *fp = fopen("log.txt", "a");
fprintf(fp, "New log entry\n");
fclose(fp);
```

Each run adds another line, perfect for logs or cumulative data.

82.4 Reading from a File with `fscanf`

To read text back, use `fscanf`. It works like `scanf`, but reads from a file.

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

int number;
char word[20];
fscanf(fp, "%d %s", &number, word);

printf("Number: %d, Word: %s\n", number, word);

fclose(fp);
```

If `input.txt` contains:

```
42 hello
```

The program prints:

```
Number: 42, Word: hello
```

82.5 Reading Lines with `fgets`

If you want to read an entire line of text, use `fgets`.

```c
char buffer[100];
fgets(buffer, sizeof(buffer), fp);
printf("Line: %s", buffer);
```

It reads up to one line (or until the buffer is full). You'll often use `fgets` in loops to process text files line by line.

82.6 End of File (EOF)

When reading, you often need to know when to stop. C signals this with a special value: `EOF`.

You can loop until you hit the end:

```c
while (fscanf(fp, "%d", &n) == 1) {
    printf("Read: %d\n", n);
}
```

Or use `feof(fp)` to check if you've reached the end.

82.7 Example: Copy Text from One File to Another

Here's a simple program that copies a file's contents:

```c
#include <stdio.h>

int main(void) {
    FILE *in = fopen("input.txt", "r");
    FILE *out = fopen("output.txt", "w");

    if (in == NULL || out == NULL) {
        printf("Error opening files.\n");
        return 1;
    }

    char ch;
    while ((ch = fgetc(in)) != EOF) {
        fputc(ch, out);
    }

    fclose(in);
    fclose(out);

    printf("Copy complete!\n");
    return 0;
}
```

Try it, you'll create an exact text copy.

82.8 Common Reading and Writing Functions

Function	Purpose
`fprintf`	Write formatted text

Function	Purpose
`fscanf`	Read formatted text
`fputs`	Write a string
`fgets`	Read a line
`fputc`	Write one character
`fgetc`	Read one character

These let you work at different levels, line by line, word by word, or character by character.

82.9 Don't Forget `fclose`

Always close your file when done:

```
fclose(fp);
```

If you forget, data might not be fully saved, it could still be sitting in a buffer.

82.10 Tiny Code Example

```
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("greetings.txt", "w");

    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(fp, "Hello from C!\n");
    fprintf(fp, "This is another line.\n");
    fclose(fp);

    fp = fopen("greetings.txt", "r");
    char line[100];

    printf("File contents:\n");
    while (fgets(line, sizeof(line), fp)) {
```

```
        printf("%s", line);
    }

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

Output:

```
File contents:
Hello from C!
This is another line.
```

Why It Matters

File I/O lets your programs remember, log, and communicate. It's how real software saves progress, stores data, and reads configuration files.

Once you master reading and writing, you can build text analyzers, loggers, and even small databases.

Try It Yourself

1. Create `numbers.txt` with some numbers and read them with `fscanf`.
2. Write a program that saves names entered by the user.
3. Build a logger using `"a"` mode.
4. Write and then read back your own "quote of the day."
5. Try copying one text file to another, one character at a time.

You've now learned how to talk to files, the foundation of every program that remembers anything.

83. Working with Binary Files

So far, you've been working with text files, reading and writing readable characters like letters and numbers. But sometimes, you'll want to work with binary files, files that store raw bytes instead of text.

Why does this matter? Because binary files are faster, smaller, and more precise. They're perfect for saving things like images, game data, or arrays of numbers exactly as they are in memory.

Let's explore how they work and how to use them safely.

83.1 What Is a Binary File

A binary file is just a sequence of bytes, no hidden formatting, no line breaks, no text encoding. It's how computers store information natively.

When you write a number like `42` to a text file, it saves `'4'` and `'2'`. When you write to a binary file, it saves the byte representation of the number, the same bits your CPU uses.

Binary files aren't meant to be opened with a text editor. You'll see strange characters, that's normal.

83.2 Opening a Binary File

You open binary files just like text files, but add a `b` to the mode:

Mode	Meaning
`"rb"`	Read binary
`"wb"`	Write binary (overwrite)
`"ab"`	Append binary
`"rb+"`	Read and write
`"wb+"`	Write and read (overwrite)

Example:

```
FILE *fp = fopen("data.bin", "wb");
```

83.3 Writing Binary Data: `fwrite`

To write binary data, use `fwrite`. It writes a block of memory directly into the file.

Syntax:

```
fwrite(pointer, size_of_each, count, file_pointer);
```

Example:

```c
int numbers[] = {10, 20, 30};
FILE *fp = fopen("data.bin", "wb");
fwrite(numbers, sizeof(int), 3, fp);
fclose(fp);
```

This writes all three integers as raw bytes. Each `int` takes up 4 bytes (on most systems), so the file will be 12 bytes long.

83.4 Reading Binary Data: `fread`

To read binary data back, use `fread`:

```c
int numbers[3];
FILE *fp = fopen("data.bin", "rb");
fread(numbers, sizeof(int), 3, fp);
fclose(fp);
```

Now the `numbers` array contains `{10, 20, 30}` again, exactly as before.

83.5 Checking Results

You can check how many items were read or written, both functions return that count:

```c
size_t written = fwrite(numbers, sizeof(int), 3, fp);
if (written != 3) printf("Write error!\n");
```

Same for `fread`:

```c
size_t read = fread(numbers, sizeof(int), 3, fp);
if (read != 3) printf("Read error or early EOF.\n");
```

This helps you catch incomplete reads or writes.

83.6 Example: Saving a Structure

You can write and read structs directly, no need to format them as text.

```c
typedef struct {
    int id;
    float score;
} Record;

Record r1 = {1, 95.5};

FILE *fp = fopen("record.bin", "wb");
fwrite(&r1, sizeof(Record), 1, fp);
fclose(fp);
```

Then read it back:

```c
Record r2;
FILE *fp = fopen("record.bin", "rb");
fread(&r2, sizeof(Record), 1, fp);
fclose(fp);

printf("ID: %d, Score: %.1f\n", r2.id, r2.score);
```

Output:

```
ID: 1, Score: 95.5
```

Binary I/O makes saving entire structs quick and easy.

83.7 Example: Reading/Writing Arrays

You can save whole arrays in one go:

```c
float data[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
FILE *fp = fopen("floats.bin", "wb");
fwrite(data, sizeof(float), 5, fp);
fclose(fp);

// Reading back
float readback[5];
fp = fopen("floats.bin", "rb");
fread(readback, sizeof(float), 5, fp);
fclose(fp);
```

```c
for (int i = 0; i < 5; i++)
    printf("%.1f ", readback[i]);
```

Output:

```
1.1 2.2 3.3 4.4 5.5
```

All values preserved exactly, no rounding or text formatting errors.

83.8 Why Use Binary Files

Binary files are:

- Compact, no wasted space on text formatting
- Fast, read/write blocks directly
- Precise, no rounding or parsing issues

But they're not human-readable, so use them when your program, not you, needs to interpret the data.

83.9 Common Mistakes

1. Forgetting `"b"` in mode (`"w"` instead of `"wb"`)
2. Mismatched types on `fread`/`fwrite`
3. Writing structs with pointers (they won't serialize correctly)
4. Reading into too-small arrays
5. Forgetting to `fclose`

Always match what you write with what you read, same size, same order.

83.10 Tiny Code Example

```c
#include <stdio.h>

int main(void) {
    int nums[3] = {5, 10, 15};
    FILE *fp = fopen("nums.bin", "wb");
    if (fp == NULL) return 1;
```

```
    fwrite(nums, sizeof(int), 3, fp);
    fclose(fp);

    int read_nums[3];
    fp = fopen("nums.bin", "rb");
    fread(read_nums, sizeof(int), 3, fp);
    fclose(fp);

    printf("Numbers read: %d %d %d\n", read_nums[0], read_nums[1], read_nums[2]);
    return 0;
}
```

Output:

```
Numbers read: 5 10 15
```

The file looks like gibberish if opened in a text editor, but it's exactly what your program expects.

Why It Matters

Binary files let you store data exactly as it exists in memory, no conversions, no formatting. They're ideal for performance, precision, and saving complex data structures.

Once you learn this, you can build fast and efficient storage systems right inside your programs.

Try It Yourself

1. Save an array of 10 integers into `data.bin` and read it back.
2. Create a struct `Person` with `name` and `age` and save one record.
3. Try opening the binary file in a text editor, what do you see?
4. Write two structs back-to-back and read them into an array.
5. Compare the file sizes of text vs binary, which is smaller?

You now have the power to work with raw data, the language your computer speaks natively!

84. Error Handling in File Operations

When you work with files, things can go wrong, a file might not exist, a disk might be full, or you might not have permission to open it. That's normal. Every good C program checks for these problems and handles them gracefully.

In this section, you'll learn how to detect and respond to file errors so your program never crashes unexpectedly.

84.1 Why Error Handling Matters

If you try to use a file that didn't open correctly, your program could:

- Crash
- Print garbage
- Corrupt data

Instead, you should always check for errors and handle them politely. A simple check and a friendly message go a long way.

84.2 Checking `fopen`

The most common error happens when a file fails to open. Whenever you call `fopen`, it returns `NULL` if something goes wrong.

Example:

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
    printf("Error: could not open file.\n");
    return 1;
}
```

Maybe the file doesn't exist, or maybe you don't have permission, either way, `fp` will be `NULL`. Always check before using it.

84.3 Using `perror`

The function `perror` prints a system message describing the last error. It's more helpful than just "Error opening file".

```c
FILE *fp = fopen("missing.txt", "r");
if (fp == NULL) {
    perror("fopen");
    return 1;
}
```

Output:

```
fopen: No such file or directory
```

Now you know exactly what went wrong.

84.4 Checking Other Operations

Other file functions also report problems. For example, `fread` and `fwrite` return how many items were processed.

```c
size_t written = fwrite(data, sizeof(int), 5, fp);
if (written < 5) {
    printf("Write failed!\n");
}
```

Similarly:

```c
size_t read = fread(buffer, sizeof(int), 5, fp);
if (read < 5) {
    if (feof(fp)) printf("End of file reached.\n");
    else printf("Read error!\n");
}
```

This way you can tell whether you hit the end or a real error occurred.

84.5 Using `feof` and `ferror`

C gives you two handy helpers:

- `feof(fp)` returns true if end-of-file reached
- `ferror(fp)` returns true if a read/write error occurred

You can use them after operations:

```
if (feof(fp)) printf("End of file.\n");
if (ferror(fp)) printf("A file error occurred.\n");
```

Together, these cover almost all file problems.

84.6 Resetting Errors with `clearerr`

If you want to reuse the same file pointer after an error, call:

```
clearerr(fp);
```

This clears the `feof` and `ferror` flags so you can try again. You don't need this often, but it's useful in loops or retries.

84.7 Example: Safe File Open

```
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("data.txt", "r");
    if (fp == NULL) {
        perror("Could not open data.txt");
        return 1;
    }

    printf("File opened successfully.\n");
    fclose(fp);
    return 0;
}
```

If `data.txt` doesn't exist, you'll see:

```
Could not open data.txt: No such file or directory
```

If it does, the program runs smoothly.

84.8 Example: Safe Reading Loop

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("numbers.txt", "r");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    int num;
    while (fscanf(fp, "%d", &num) == 1) {
        printf("Read: %d\n", num);
    }

    if (ferror(fp)) printf("Read error occurred.\n");
    else if (feof(fp)) printf("End of file reached.\n");

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

This program safely handles every case, missing file, bad data, or end-of-file.

84.9 When to Stop and When to Recover

Sometimes the right move is to stop the program (for critical errors). Other times, you can recover and move on (like skipping bad lines).

Example recovery:

```c
if (ferror(fp)) {
    clearerr(fp);
    printf("Error ignored, continuing...\n");
}
```

You get to decide based on the situation.

84.10 Tiny Code Example

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("output.txt", "w");
    if (fp == NULL) {
        perror("Failed to open file");
        return 1;
    }

    int result = fprintf(fp, "Hello, world!\n");
    if (result < 0) {
        perror("Write failed");
        fclose(fp);
        return 1;
    }

    if (fclose(fp) == EOF) {
        perror("Error closing file");
        return 1;
    }

    printf("All good! File written and closed.\n");
    return 0;
}
```

This program checks every step, open, write, close, and handles any failure.

Why It Matters

Error handling makes your programs reliable and professional. Even if something goes wrong, your program stays calm and explains the problem clearly.

It's a small habit that makes a big difference.

Try It Yourself

1. Open a file that doesn't exist, print a clear error.
2. Try writing to a read-only file, catch and report it.
3. Read past the end of a file, detect `feof`.

4. Intentionally trigger `ferror`, then clear it with `clearerr`.
5. Wrap your file code in safety checks, one by one.

Once you handle errors gracefully, you'll feel in full control, your programs won't just run, they'll respond intelligently when something goes wrong.

85. Command-Line Arguments

Up to now, your programs have always started the same way, no matter what you type when you run them. But sometimes, you want your program to behave differently depending on user input from the command line.

That's where command-line arguments come in. They let users pass information directly to `main` when starting the program.

This is how real-world programs take options like `ls -l` or `gcc main.c -o app`. Let's learn how you can do the same in your own programs.

85.1 The Special Form of `main`

Until now, you've written:

```
int main(void)
```

To accept arguments, you'll use:

```
int main(int argc, char *argv[])
```

Here's what those mean:

- `argc` = argument count (how many items are on the command line)
- `argv` = argument vector (an array of strings containing each argument)

Think of `argv` as a list of words typed after your program's name.

85.2 How It Works

Let's look at a simple example:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("You passed %d arguments.\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

If you compile this as `args`, then run:

```
./args hello world 123
```

You'll see:

```
You passed 4 arguments.
argv[0] = ./args
argv[1] = hello
argv[2] = world
argv[3] = 123
```

Notice how `argv[0]` is always your program's name. Everything after it is user input.

85.3 Using Arguments

You can use arguments to make your program flexible.

Example:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    printf("Opening file: %s\n", argv[1]);
    return 0;
}
```

Run it like this:

```
./program data.txt
```

If you forget the filename, it shows a friendly usage message.

85.4 Converting Strings to Numbers

Command-line arguments are always strings. If you want numbers, you'll need to convert them.

Use `atoi` (ASCII to integer) or `atof` (to float):

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <a> <b>\n", argv[0]);
        return 1;
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);

    printf("%d + %d = %d\n", a, b, a + b);
    return 0;
}
```

Run:

```
./sum 5 7
```

Output:

```
5 + 7 = 12
```

85.5 Handling Too Few Arguments

If users forget to pass arguments, don't crash, explain what to do.

Bad:

```
printf("%s\n", argv[1]); //  might crash if argc < 2
```

Good:

```
if (argc < 2) {
    printf("Missing argument!\n");
    return 1;
}
```

Always check before accessing `argv[i]`.

85.6 Example: Echo Program

Here's a small program that repeats whatever the user types:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return 0;
}
```

Run:

```
./echo Hello from C
```

Output:

```
Hello from C
```

Simple, clean, and useful.

85.7 Why It's Useful

Command-line arguments let you:

- Pass filenames, numbers, or options
- Build flexible tools
- Run scripts with parameters
- Automate tests or batch jobs

You'll use them in almost every real C program you write.

85.8 Quick Recap

- `argc` = how many arguments
- `argv` = array of strings
- `argv[0]` = program name
- Always check `argc` before using `argv[i]`
- Convert strings with `atoi`, `atof`, or `strtol`

85.9 Tiny Code Example

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <num1> <num2>\n", argv[0]);
        return 1;
    }

    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    printf("Sum: %d\n", x + y);
    return 0;
}
```

Try:

```
./sum 10 20
```

Output:

```
Sum: 30
```

85.10 Why It Matters

Command-line arguments turn your program into a tool, something that responds to the user's input. You're no longer limited to hard-coded data, now, your program listens to what's typed at launch.

Try It Yourself

1. Write a program that takes one filename and prints "Opening file: name".
2. Create a calculator that adds two numbers from the command line.
3. Make a greeting tool: `./greet Alice` prints "Hello, Alice!"
4. Try running with no arguments, handle the error.
5. Print all arguments except `argv[0]` on one line.

Once you get used to `argc` and `argv`, you'll feel like you're giving your programs a voice, they can now respond to you right from the terminal.

86. Using `make` and Makefiles

As your programs grow beyond one file, typing long compile commands becomes tiring, and easy to mess up. Wouldn't it be nice if you could just type one simple word, like `make`, and your whole program compiled automatically?

That's exactly what the `make` tool does. It reads a special file called a Makefile, learns how your project is built, and only recompiles what has changed. It's one of the most useful tools you'll ever learn.

86.1 What Is `make`

`make` is a build automation tool. You describe how to build your program once, and `make` takes care of the rest.

Instead of typing:

```
gcc main.c helper.c math.c -o app
```

You can just type:

```
make
```

And `make` will figure out the right commands to run.

86.2 The Makefile

A Makefile is a plain text file (named `Makefile` or `makefile`) that lists:

- Targets (what to build)
- Dependencies (what files it needs)
- Commands (how to build it)

Each line that starts with a tab is a command.

Here's the simplest example:

```
app: main.c
    gcc main.c -o app
```

Run:

```
make
```

and `make` will compile `main.c` into `app`.

86.3 Targets, Dependencies, Commands

A rule has three parts:

```
target: dependencies
<TAB> command
```

In our example:

- Target: `app` (the program you want to build)
- Dependency: `main.c` (file needed to build it)
- Command: `gcc main.c -o app` (how to build it)

Make sure you use a real tab before the command, not spaces!

86.4 Adding Multiple Files

If your program has more than one source file:

```
app: main.c helper.c
    gcc main.c helper.c -o app
```

Now, if you change only `helper.c`, `make` knows to rebuild `app`.

You don't need to retype the whole command every time, just run `make`.

86.5 Cleaning Up

You can add a special clean target to remove compiled files:

```
clean:
    rm -f app
```

Run:

```
make clean
```

and it deletes the program, letting you start fresh.

86.6 Using Variables

To avoid repeating yourself, define variables:

```
CC = gcc
CFLAGS = -Wall -Wextra

app: main.c helper.c
    $(CC) $(CFLAGS) main.c helper.c -o app
```

Now you can change `CC` or `CFLAGS` in one place.

`$(CC)` means "insert the value of `CC`".

86.7 Example: Simple Project

Say you have:

```
main.c
math.c
math.h
```

You can write:

```
CC = gcc
CFLAGS = -Wall -Wextra

app: main.o math.o
    $(CC) $(CFLAGS) main.o math.o -o app

main.o: main.c math.h
    $(CC) $(CFLAGS) -c main.c

math.o: math.c math.h
    $(CC) $(CFLAGS) -c math.c

clean:
    rm -f *.o app
```

Then just run:

```
make
```

to build, and:

```
make clean
```

to tidy up.

86.8 Incremental Builds

The best part about `make` is speed. It checks file timestamps, only recompiles what's changed, and skips the rest.

So if you edit `math.c`, only `math.o` is rebuilt. This saves time in big projects.

86.9 Default Target

The first rule in your Makefile is the default target. That's what runs when you type `make` with no arguments.

You can also specify others:

```
make clean
make app
```

Each target is like a mini command.

86.10 Tiny Example

Here's a full Makefile for a two-file program:

```
CC = gcc
CFLAGS = -Wall

app: main.o utils.o
    $(CC) $(CFLAGS) main.o utils.o -o app

main.o: main.c utils.h
    $(CC) $(CFLAGS) -c main.c

utils.o: utils.c utils.h
    $(CC) $(CFLAGS) -c utils.c

clean:
    rm -f *.o app
```

Now run:

```
make
```

to build, and:

```
make clean
```

to remove everything.

Why It Matters

Makefiles turn messy compile commands into one simple word: `make`. They save you time, prevent errors, and are used in almost every C project. Learning `make` is like learning a superpower for managing your code.

Try It Yourself

1. Create a program split into `main.c` and `helper.c`.
2. Write a simple Makefile to build them into one executable.
3. Add a `clean` target to delete all `.o` files.
4. Use variables `CC` and `CFLAGS`.
5. Edit one file, see how `make` rebuilds only what changed.

Once you start using Makefiles, you'll wonder how you ever compiled without them.

87. Debugging with `gdb`

Even the best programmers make mistakes, bugs happen. What matters is how you find and fix them.

Instead of just guessing, you can use a debugger, a tool that lets you run your program step by step, see variables, and stop exactly where things go wrong. In C, the most common debugger is `gdb`, the GNU Debugger.

Let's learn how to use it to actually *see* what your program is doing.

87.1 What Is a Debugger

A debugger is like a microscope for your code. You can:

- Run your program one line at a time
- Inspect variable values
- Pause when certain conditions happen
- Jump back and forth through the flow

It's one of the most powerful tools for understanding your code.

87.2 Compile with Debug Info

To debug properly, compile your program with the `-g` flag:

```
gcc -g main.c -o main
```

This tells the compiler to include *debug symbols*, information about line numbers and variable names, so `gdb` knows what's going on.

Without `-g`, `gdb` can still run your program, but you won't see meaningful info.

87.3 Starting `gdb`

Launch your program under the debugger like this:

```
gdb ./main
```

This opens the `gdb` interface. You'll see a prompt that looks like this:

```
(gdb)
```

From here, you can type commands to control your program.

87.4 Running the Program

To start the program:

```
(gdb) run
```

Your program executes just like normal. If it crashes, `gdb` pauses and shows where.

If your program needs arguments:

```
(gdb) run arg1 arg2
```

This works just like command-line arguments outside the debugger.

87.5 Setting Breakpoints

A breakpoint tells `gdb` where to pause execution. You can then inspect values before continuing.

Set a breakpoint at a line number:

```
(gdb) break 10
```

or at a function:

```
(gdb) break main
(gdb) break compute_sum
```

When your program hits that line or function, it stops, right before running it.

87.6 Running Step by Step

Once paused, you can move through code line by line:

Command	Action
`next`	Run next line (skip over function calls)
`step`	Step into a function call
`continue`	Resume execution until next breakpoint
`finish`	Run until current function returns

This helps you follow exactly what your code does.

87.7 Inspecting Variables

You can check what's inside any variable with `print`:

```
(gdb) print x
$1 = 42
```

You can also watch complex expressions:

```
(gdb) print a + b
```

Or list all locals:

```
(gdb) info locals
```

This is great for catching logic errors, when a variable isn't what you expect.

87.8 Example Session

Imagine this buggy code:

```c
#include <stdio.h>

int divide(int a, int b) {
    return a / b;
}

int main(void) {
    int x = 10;
    int y = 0;
    int z = divide(x, y);
    printf("%d\n", z);
    return 0;
}
```

Compile with debug info:

```
gcc -g main.c -o main
```

Run under gdb:

```
gdb ./main
```

Then:

```
(gdb) break divide
(gdb) run
```

When it stops:

```
Breakpoint 1, divide (a=10, b=0)
```

Check `b`:

```
(gdb) print b
$1 = 0
```

You just caught the bug, division by zero, before it crashed!

87.9 Quitting `gdb`

When you're done:

```
(gdb) quit
```

Press `y` if it asks to confirm.

87.10 Tiny Example

```
gcc -g bug.c -o bug
gdb ./bug
```

Inside `gdb`:

```
(gdb) break main
(gdb) run
(gdb) next
(gdb) print x
(gdb) continue
(gdb) quit
```

You've just stepped through a program, watched a variable, and exited cleanly.

Why It Matters

A debugger saves you hours of frustration. Instead of printing variables everywhere with `printf`, you can pause and look directly at what's happening inside your program.

Once you learn `gdb`, you'll debug smarter, not harder.

Try It Yourself

1. Write a small program with a bug (like division by zero).
2. Compile with `-g` and open in `gdb`.
3. Set a breakpoint at the function where the bug happens.
4. Run, step through, and inspect variables.
5. Fix the bug, recompile, and confirm it's gone.

Debugging isn't just for fixing mistakes, it's for *understanding* your code deeply. Once you get comfortable with `gdb`, you'll feel like you can see inside your program's mind.

88. Understanding Linking and Libraries

When you write a program in C, you rarely work alone. Your code often depends on functions from other files or pre-built libraries, like `printf`, `sqrt`, or even your own helper modules.

The process that brings all these pieces together into a single program is called linking. It's what happens *after* compilation, turning your `.o` files into a real executable.

Let's explore how linking works, why it matters, and how to use libraries with confidence.

88.1 The Two-Step Build Process

When you compile C code, two main steps happen:

1. Compilation Each `.c` file becomes an object file (`.o`):

   ```
   gcc -c main.c   →  main.o
   ```

2. Linking All object files (and libraries) combine into one program:

   ```
   gcc main.o math.o -o app
   ```

If the linker can't find a function (like `printf`), you'll see an undefined reference error. That means you forgot to link in the file or library that provides it.

88.2 What Is a Library

A library is a collection of precompiled code, a set of `.o` files bundled together. Instead of writing `printf` yourself, you just link against the standard library.

There are two main kinds:

- Static libraries (`.a`), code is copied into your program
- Shared libraries (`.so` on Linux, `.dll` on Windows), code is loaded at runtime

Both let you reuse code without rewriting it.

88.3 Linking Multiple Files

If you split your program across several files:

```
main.c  helper.c
```

You compile them separately:

```
gcc -c main.c
gcc -c helper.c
```

Then link:

```
gcc main.o helper.o -o app
```

If you skip `helper.o`, you'll get an error:

```
undefined reference to 'helper_function'
```

That's the linker saying: "I see the call, but where's the definition?"

88.4 The Order Matters

When linking manually, order can be important. The linker reads left to right, it needs to see object files before libraries that use them.

Example:

```
gcc main.o helper.o -o app
```

works, but

```
gcc -o app main.o
```

without `helper.o` doesn't.

So always include all needed files and libraries.

88.5 Using the Math Library

Some functions, like `sqrt` or `pow`, live in special libraries (like `libm`). To use them, you need to link with `-lm`:

```
gcc mathdemo.c -o mathdemo -lm
```

If you forget `-lm`, you'll see:

```
undefined reference to 'sqrt'
```

Adding `-lm` fixes it. Think of `-l<name>` as "link library named `<name>`".

88.6 Static vs Shared Libraries

- Static (`.a`): included in your program at build time. Result: one big standalone file.
- Shared (`.so` / `.dll`): loaded dynamically at runtime. Result: smaller executable, but needs the library present on the system.

For beginners, you don't need to build your own yet, just know that most system libraries are shared.

88.7 Example: Your Own Library

Let's say you have:

```
// mathutils.c
int square(int x) { return x * x; }
```

Compile into a library:

```
gcc -c mathutils.c
ar rcs libmathutils.a mathutils.o
```

Now link with it:

```
gcc main.c -L. -lmathutils -o app
```

`-L.` means "look in current directory", `-lmathutils` means "use libmathutils.a".

Now your app uses your own static library!

88.8 Common Linker Errors

Error	Meaning	Fix
`undefined reference`	Function called but not found	Add missing `.o` or `-l` flag
`cannot find -lmylib`	Library file missing	Check `-L` path
`multiple definition`	Same function defined twice	Remove duplicate or use `extern`

The linker doesn't compile, it just matches names. So missing or duplicated definitions cause trouble.

88.9 Linking with Makefiles

Makefiles make linking easy. You can describe dependencies once, and `make` handles them.

Example:

```
app: main.o helper.o
    gcc main.o helper.o -o app
```

If `main.c` or `helper.c` changes, `make` recompiles and relinks automatically.

88.10 Tiny Code Example

```c
// helper.c
#include <stdio.h>
void greet(void) {
    printf("Hello from helper!\n");
}
```

```c
// main.c
void greet(void);

int main(void) {
    greet();
    return 0;
}
```

Compile and link:

```
gcc -c main.c
gcc -c helper.c
gcc main.o helper.o -o app
./app
```

Output:

```
Hello from helper!
```

Without linking both `.o` files, the call to `greet` wouldn't work.

Why It Matters

Linking is the bridge between your code and other code. It's what turns individual `.c` files, or entire libraries, into a single working program. Once you understand it, multi-file projects and libraries stop being mysterious.

Try It Yourself

1. Split a program into `main.c` and `helper.c`, link them together.
2. Use `sqrt`, try compiling without and with `-lm`.
3. Build your own static library with `ar` and link to it.
4. Trigger an "undefined reference" error, then fix it.
5. Write a Makefile that builds and links everything automatically.

After this, you'll see linking not as magic, but as a simple, logical final step, connecting all your code into one.

89. Simple Threads with `<threads.h>`

So far, all your programs have done one thing at a time, one main path, one sequence of instructions. But many real-world programs need to do multiple things at once: downloading files while updating a progress bar, handling many connections, or computing parts of a problem in parallel.

To do that, you use threads, lightweight "mini-programs" that run side by side inside the same process. And starting with modern C, you have a built-in way to use them through `<threads.h>`.

Let's explore how to create, run, and join threads safely, step by step.

89.1 What Is a Thread

A thread is a path of execution inside a program. Every program starts with one, the main thread. When you create new threads, they share the same memory but run independently.

You can think of threads like helpers: each one works on a task while the others do something else.

89.2 Including the Thread Library

To use threads in modern C, include:

```
#include <threads.h>
```

This gives you access to:

- `thrd_t` → the thread type
- `thrd_create()` → to start a new thread

- `thrd_join()` → to wait for it to finish

It's simple, portable, and standardized, no special libraries needed.

89.3 A Thread Function

Each thread runs a function. That function must take a single `void*` argument and return an `int`.

Example:

```
int work(void *arg) {
    printf("Hello from a thread!\n");
    return 0;
}
```

89.4 Creating a Thread

You create a thread with `thrd_create`:

```
thrd_t t;
thrd_create(&t, work, NULL);
```

This starts a new thread that runs `work(NULL)`.

If you want to pass data, replace `NULL` with a pointer:

```
int value = 42;
thrd_create(&t, work, &value);
```

The thread will receive that pointer as its `arg`.

89.5 Waiting for a Thread

If you want to wait for a thread to finish before continuing, call `thrd_join`:

```
thrd_join(t, NULL);
```

This blocks the main thread until `t` completes. Without joining, your program might end before the thread finishes.

89.6 Example: One Thread

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

int work(void *arg) {
    printf("Running in another thread!\n");
    return 0;
}

int main(void) {
    thrd_t t;
    thrd_create(&t, work, NULL);
    thrd_join(t, NULL);
    printf("Back in main.\n");
    return 0;
}
```

Output:

```
Running in another thread!
Back in main.
```

You just launched your first parallel task!

89.7 Example: Passing Data

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

int print_number(void *arg) {
    int num = *(int *)arg;
    printf("Number: %d\n", num);
    return 0;
}

int main(void) {
    int x = 7;
    thrd_t t;
```

```
    thrd_create(&t, print_number, &x);
    thrd_join(t, NULL);
    return 0;
}
```

This shows how to share simple data. The thread prints whatever number it's given.

89.8 Multiple Threads

You can launch several threads at once:

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

int hello(void *arg) {
    int id = *(int *)arg;
    printf("Hello from thread %d\n", id);
    return 0;
}

int main(void) {
    thrd_t threads[3];
    int ids[3] = {1, 2, 3};

    for (int i = 0; i < 3; i++)
        thrd_create(&threads[i], hello, &ids[i]);

    for (int i = 0; i < 3; i++)
        thrd_join(threads[i], NULL);

    printf("All threads finished.\n");
    return 0;
}
```

Output (order may vary):

```
Hello from thread 2
Hello from thread 1
Hello from thread 3
All threads finished.
```

Threads run concurrently, so order isn't guaranteed.

89.9 Thread Safety Tips

Threads share memory, so they can modify the same data at the same time. That's powerful but dangerous, it can cause race conditions.

Simple rules for now:

- Don't change the same variable from two threads
- Pass separate data to each thread
- Use `mtx_t` (mutex) if you need to share (coming next)

Keep it simple: one thread per independent task.

89.10 Tiny Code Example

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

int greet(void *arg) {
    printf("Hello from thread!\n");
    return 0;
}

int main(void) {
    thrd_t t;
    if (thrd_create(&t, greet, NULL) != thrd_success) {
        printf("Failed to create thread.\n");
        return 1;
    }
    thrd_join(t, NULL);
    printf("Main finished.\n");
    return 0;
}
```

Output:

```
Hello from thread!
Main finished.
```

Why It Matters

Threads let your programs do more than one thing at a time, making them faster and more responsive. You'll see them everywhere: in servers, games, data processing, and UI applications.

And with `<threads.h>`, you can use them in clean, standard C.

Try It Yourself

1. Create one thread that prints a message.
2. Launch three threads that each print their ID.
3. Pass a number to a thread and print its square.
4. Try removing `thrd_join`, what happens?
5. Experiment with random delays to see thread interleaving.

Once you get comfortable, you'll see that threads are just like helpers, small, independent workers that share your program's memory and time.

90. Synchronization and Data Safety

Now that you've met threads, you know they can run at the same time and even share memory. That's powerful, but it can also be dangerous if two threads try to change the same variable at once.

To keep your data safe and your program stable, you need synchronization, tools that help threads take turns and avoid stepping on each other's work.

In this section, you'll learn the basics of synchronization with mutexes and locks. Don't worry, we'll go slowly and keep it simple.

90.1 The Problem: Race Conditions

A race condition happens when two threads try to change shared data at the same time.

Example:

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

int counter = 0;

int add(void *arg) {
```

```
    for (int i = 0; i < 100000; i++) {
        counter++;
    }
    return 0;
}

int main(void) {
    thrd_t t1, t2;
    thrd_create(&t1, add, NULL);
    thrd_create(&t2, add, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    printf("Counter: %d\n", counter);
    return 0;
}
```

You might expect `200000`, but you'll often get a smaller number. That's because the two threads race to update `counter`, and one overwrites the other's work.

We need a way to make sure only one thread updates at a time.

90.2 Meet the Mutex

A mutex (mutual exclusion) is like a lock on a door, only one thread can hold it at a time. When one thread locks it, others must wait until it unlocks.

C provides a mutex type in `<threads.h>`:

```
mtx_t lock;
```

You create it with:

```
mtx_init(&lock, mtx_plain);
```

And use it like this:

```
mtx_lock(&lock);   // enter critical section
// do safe work
mtx_unlock(&lock); // leave critical section
```

This guarantees that only one thread at a time runs the protected code.

90.3 Fixing the Race

Let's fix the counter example with a mutex.

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

int counter = 0;
mtx_t lock;

int add(void *arg) {
    for (int i = 0; i < 100000; i++) {
        mtx_lock(&lock);
        counter++;
        mtx_unlock(&lock);
    }
    return 0;
}

int main(void) {
    mtx_init(&lock, mtx_plain);

    thrd_t t1, t2;
    thrd_create(&t1, add, NULL);
    thrd_create(&t2, add, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("Counter: %d\n", counter);

    mtx_destroy(&lock);
    return 0;
}
```

Now the program always prints:

```
Counter: 200000
```

The mutex makes the increment atomic, one thread finishes before the next begins.

90.4 Critical Sections

The code between `mtx_lock` and `mtx_unlock` is called a critical section. Only one thread may be inside it at a time.

Use critical sections to:

- Update shared variables
- Write to shared files
- Modify shared data structures

Keep them short, locking too long can make your program slower.

90.5 Initializing and Destroying

Always remember to:

- `mtx_init` before first use
- `mtx_destroy` at the end

It's good practice, even for small programs.

90.6 Example: Bank Account

Let's see a practical use. Two threads deposit money into the same account:

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

int balance = 0;
mtx_t lock;

int deposit(void *arg) {
    for (int i = 0; i < 100000; i++) {
        mtx_lock(&lock);
        balance++;
        mtx_unlock(&lock);
    }
    return 0;
}

int main(void) {
    mtx_init(&lock, mtx_plain);
```

```
    thrd_t t1, t2;
    thrd_create(&t1, deposit, NULL);
    thrd_create(&t2, deposit, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("Final balance: %d\n", balance);

    mtx_destroy(&lock);
    return 0;
}
```

No matter how the threads interleave, the balance is correct, thanks to the mutex.

90.7 Try Without Lock

If you remove the lock/unlock lines, the balance will often be wrong. Try it once, you'll see how unpredictable shared data can become. That's why synchronization is essential.

90.8 Other Synchronization Tools

Mutexes are just the start. C's `<threads.h>` also offers:

- `cnd_t` for condition variables (wait/notify)
- `once_flag` for one-time initialization

But for most beginner programs, mutexes are all you need.

90.9 Common Mistakes

1. Forgetting to `mtx_init` → crash
2. Forgetting to `mtx_destroy` → resource leak
3. Locking twice without unlocking → deadlock
4. Unlocking from a different thread → undefined behavior
5. Holding a lock too long → performance drop

Keep it simple: one lock, one unlock, short critical section.

90.10 Tiny Code Example

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

int value = 0;
mtx_t m;

int increment(void *arg) {
    mtx_lock(&m);
    value++;
    printf("Value now: %d\n", value);
    mtx_unlock(&m);
    return 0;
}

int main(void) {
    mtx_init(&m, mtx_plain);
    thrd_t t1, t2;
    thrd_create(&t1, increment, NULL);
    thrd_create(&t2, increment, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    mtx_destroy(&m);
    return 0;
}
```

Output:

```
Value now: 1
Value now: 2
```

Safe, predictable, and fully synchronized.

Why It Matters

Without synchronization, multi-threaded programs become unreliable and hard to debug. With mutexes, you control when and how threads access shared data, keeping everything consistent and correct.

You've just learned the secret to safe concurrency.

Try It Yourself

1. Re-run your counter example with and without `mtx_lock`. Compare results.
2. Create a shared array and have two threads fill different halves.
3. Add a mutex to protect a shared sum variable.
4. Try locking twice in a row, see what happens (then fix it).
5. Measure how many iterations per second you get with and without locking.

Once you master synchronization, you'll be ready to write safe, concurrent C programs, fast and correct at the same time.

Chapter 10. Putting it all together

91. Mini Project 1: Text Analyzer

Congratulations, you've made it all the way here! Now it's time to bring everything together.

In this mini project, you'll build a Text Analyzer, a simple tool that reads a text file and reports basic statistics, like how many lines, words, and characters it contains.

This project combines everything you've learned: file I/O, loops, conditionals, functions, and basic data handling. Let's take it step by step.

91.1 Project Goal

Write a program that:

1. Takes a filename as a command-line argument.
2. Opens the file.
3. Reads it line by line.
4. Counts:
    - Total characters
    - Total words
    - Total lines
5. Prints a summary at the end.

This is a classic utility, similar to the Unix `wc` command (word count).

91.2 Planning the Program

Let's think before coding. We'll need:

- A function to open the file.
- A loop to read it line by line.
- Logic to count words (detect spaces and newlines).
- A final summary.

We'll count words by checking when a sequence of letters starts.

91.3 Handling Input

Our program takes the filename from the command line:

```
if (argc < 2) {
    printf("Usage: %s <filename>\n", argv[0]);
    return 1;
}
```

Then we'll open it safely:

```
FILE *fp = fopen(argv[1], "r");
if (fp == NULL) {
    perror("Error opening file");
    return 1;
}
```

91.4 Reading the File

We can read one character at a time with `fgetc()`, this makes counting easier.

We'll track:

- `chars` for total characters
- `words` for total words
- `lines` for total lines
- `in_word` flag to check if we're inside a word

91.5 Core Counting Logic

Here's the main counting loop:

```
int chars = 0, words = 0, lines = 0;
int c, in_word = 0;

while ((c = fgetc(fp)) != EOF) {
    chars++;

    if (c == '\n')
        lines++;

    if (c == ' ' || c == '\n' || c == '\t') {
        in_word = 0;
    } else if (!in_word) {
        in_word = 1;
        words++;
    }
}
```

This loop goes through each character, tracks lines when it sees `'\n'`, and increments `words` each time a new word starts.

91.6 Displaying the Results

When done, print the summary:

```
printf("Lines: %d\n", lines);
printf("Words: %d\n", words);
printf("Characters: %d\n", chars);
```

Finally, don't forget to close the file:

```
fclose(fp);
```

91.7 Full Program

Here's the complete version:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    FILE *fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }

    int chars = 0, words = 0, lines = 0;
    int c, in_word = 0;

    while ((c = fgetc(fp)) != EOF) {
        chars++;

        if (c == '\n')
            lines++;

        if (c == ' ' || c == '\n' || c == '\t') {
            in_word = 0;
        } else if (!in_word) {
            in_word = 1;
            words++;
        }
    }

    fclose(fp);

    printf("File: %s\n", argv[1]);
    printf("Lines: %d\n", lines);
    printf("Words: %d\n", words);
    printf("Characters: %d\n", chars);

    return 0;
}
```

91.8 Example Run

Say you have a file called `sample.txt`:

```
Hello world!
This is a test.
```

Run:

```
./textanalyzer sample.txt
```

Output:

```
File: sample.txt
Lines: 2
Words: 5
Characters: 27
```

91.9 Tiny Improvements

You can enhance it later:

- Add error messages for empty files.
- Support reading from standard input (`stdin`).
- Print average word length.
- Use `fgets()` instead of `fgetc()` for performance.

Each improvement builds on what you know.

91.10 Why It Matters

This project ties together files, loops, and logic in one useful tool. It shows how simple building blocks can become a real program.

You're not just learning C, you're learning how to think like a software builder.

Try It Yourself

1. Run the program on different text files.
2. Add a counter for blank lines.
3. Modify it to count digits or punctuation marks.
4. Use `fgets()` and `strlen()` instead of `fgetc()`.
5. Print a summary table for multiple files (bonus).

You've just written your first utility, a small but mighty C program that reads real data and analyzes it like a pro.

92. Mini Project 2: Guessing Game

Let's take a break from files and build something fun, a Guessing Game! This little project will help you practice loops, conditionals, random numbers, and user input.

Your program will pick a secret number, and you'll try to guess it. After each guess, it tells you if you're too high, too low, or exactly right.

This is one of the best beginner projects, simple, interactive, and great for mastering logic.

92.1 Project Goal

Write a program that:

1. Chooses a random number between 1 and 100.
2. Prompts the user to guess the number.
3. Tells the user if the guess is too high, too low, or correct.
4. Counts how many guesses it took.
5. Ends when the user guesses correctly.

92.2 What You'll Learn

This project helps you practice:

- Generating random numbers
- Reading user input safely
- Using loops and conditionals
- Giving clear feedback to the user

You'll also get to see how programs can interact, almost like a game.

92.3 Setting Up Random Numbers

C provides random numbers with `rand()`, but to make it different each time, you seed it with the current time.

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

srand(time(NULL)); // set the seed
int secret = rand() % 100 + 1; // number from 1 to 100
```

`rand() % 100` gives a value between 0 and 99, so we add 1 to shift it into 1–100.

92.4 Getting the User's Guess

We'll use `scanf` to get the user's guess:

```
int guess;
printf("Enter your guess: ");
scanf("%d", &guess);
```

Always check the input is valid, but for now, we'll keep it simple.

92.5 The Game Loop

We'll keep asking until the user guesses correctly. A `while` loop is perfect:

```
int guess = 0;
int tries = 0;

while (guess != secret) {
    printf("Enter your guess: ");
    scanf("%d", &guess);
    tries++;

    if (guess < secret)
        printf("Too low! Try again.\n");
    else if (guess > secret)
        printf("Too high! Try again.\n");
    else
        printf("Correct! You guessed it in %d tries.\n", tries);
}
```

92.6 Putting It Together

Here's the complete program:

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

int main(void) {
    srand(time(NULL));
    int secret = rand() % 100 + 1;
    int guess = 0;
    int tries = 0;

    printf("I'm thinking of a number between 1 and 100.\n");

    while (guess != secret) {
        printf("Enter your guess: ");
        scanf("%d", &guess);
        tries++;

        if (guess < secret)
            printf("Too low! Try again.\n");
        else if (guess > secret)
            printf("Too high! Try again.\n");
        else
            printf("Correct! You guessed it in %d tries.\n", tries);
    }

    return 0;
}
```

92.7 Example Run

```
I'm thinking of a number between 1 and 100.
Enter your guess: 50
Too low! Try again.
Enter your guess: 75
Too high! Try again.
Enter your guess: 63
Too low! Try again.
Enter your guess: 69
```

```
Correct! You guessed it in 4 tries.
```

Every run is different because of the random seed.

92.8 Tiny Improvements

Once you've got it working, you can make it more fun:

- Add input validation (`if (guess < 1 || guess > 100)`)
- Show a "hint" if the guess is very close
- Let the user choose the range (1–50, 1–1000, etc.)
- Ask if they want to play again

Each of these adds a little more logic and creativity.

92.9 Common Mistakes

1. Forgetting `srand(time(NULL))`, the number will be the same every run.
2. Using `rand() % 100` but forgetting `+ 1` (you'll never guess 100).
3. Not updating `tries` each loop.
4. Using `=` instead of `==` in comparisons.
5. Forgetting to handle invalid input (try entering a letter!).

Don't worry, these are easy to fix once you know them.

92.10 Why It Matters

This project blends logic and interaction, a perfect match for beginners. It's simple enough to build in minutes, but rich enough to teach important ideas like loops, conditionals, and randomness.

You're not just printing text anymore, you're creating a tiny game!

Try It Yourself

1. Change the range to 1–50.
2. Add a "hint" if the user is within 10 of the secret.
3. Keep track of best score (fewest guesses).
4. Ask if the user wants to play again.
5. Use a `for` loop with a max number of guesses (like 10).

You've now built your first interactive game, one that listens, thinks, and responds. This is where programming really starts to feel magical.

93. Mini Project 3: Calculator

It's time to build something every programmer tries at least once, a Calculator! This project gives you great practice with user input, operators, switch statements, and functions.

You'll create a small program that reads two numbers and an operator, performs the calculation, and prints the result. It's simple, useful, and a perfect way to pull together what you've learned so far.

93.1 Project Goal

Build a calculator that:

1. Asks the user for two numbers.
2. Asks what operation to perform (+, -, *, /).
3. Performs that operation.
4. Prints the result.
5. Handles invalid operators gracefully.

This is a classic practice project for learning input, branching, and math.

93.2 Plan the Steps

Let's outline what the program will do:

1. Ask for first number.
2. Ask for second number.
3. Ask for operator (`+`, `-`, `*`, `/`).
4. Use a `switch` to decide which math to do.
5. Display the result.
6. Handle division by zero or invalid inputs.

This clear sequence keeps your program simple and readable.

93.3 Getting User Input

We'll use `scanf` to read the values:

```
double a, b;
char op;

printf("Enter first number: ");
scanf("%lf", &a);

printf("Enter an operator (+, -, *, /): ");
scanf(" %c", &op); // note the space before %c

printf("Enter second number: ");
scanf("%lf", &b);
```

The space before `%c` makes sure we skip leftover newlines from earlier input.

93.4 Deciding What to Do

We'll use a `switch` on the operator:

```
switch (op) {
    case '+':
        printf("%.2f + %.2f = %.2f\n", a, b, a + b);
        break;
    case '-':
        printf("%.2f - %.2f = %.2f\n", a, b, a - b);
        break;
    case '*':
        printf("%.2f * %.2f = %.2f\n", a, b, a * b);
        break;
    case '/':
        if (b != 0)
            printf("%.2f / %.2f = %.2f\n", a, b, a / b);
        else
            printf("Error: division by zero!\n");
        break;
    default:
        printf("Unknown operator: %c\n", op);
}
```

This structure makes it easy to add more operations later.

93.5 Full Program

Here's your complete calculator:

```c
#include <stdio.h>

int main(void) {
    double a, b;
    char op;

    printf("Enter first number: ");
    scanf("%lf", &a);

    printf("Enter an operator (+, -, *, /): ");
    scanf(" %c", &op);

    printf("Enter second number: ");
    scanf("%lf", &b);

    switch (op) {
        case '+':
            printf("%.2f + %.2f = %.2f\n", a, b, a + b);
            break;
        case '-':
            printf("%.2f - %.2f = %.2f\n", a, b, a - b);
            break;
        case '*':
            printf("%.2f * %.2f = %.2f\n", a, b, a * b);
            break;
        case '/':
            if (b != 0)
                printf("%.2f / %.2f = %.2f\n", a, b, a / b);
            else
                printf("Error: division by zero!\n");
            break;
        default:
            printf("Unknown operator: %c\n", op);
    }
```

```
    return 0;
}
```

93.6 Example Run

```
Enter first number: 10
Enter an operator (+, -, *, /): *
Enter second number: 5
10.00 * 5.00 = 50.00
```

Another run:

```
Enter first number: 7
Enter an operator (+, -, *, /): /
Enter second number: 0
Error: division by zero!
```

Perfect, your program reacts exactly as it should.

93.7 Improving It with a Loop

Want to keep calculating until the user quits? Add a loop:

```
char cont = 'y';

while (cont == 'y' || cont == 'Y') {
    // (all calculator code here)
    printf("Do another calculation? (y/n): ");
    scanf(" %c", &cont);
}
```

Now it's a reusable calculator!

93.8 Moving to Functions

You can move the math logic into a function:

```c
double calculate(double a, double b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return (b != 0) ? a / b : 0;
        default: return 0;
    }
}
```

This makes your code modular and easier to extend.

93.9 Common Mistakes

1. Forgetting space before `%c` in `scanf` → operator input is skipped.
2. Division by zero not handled → crash or `inf`.
3. Mixing up `%d` and `%lf` → wrong output for doubles.
4. No `break` in `switch` → falls through to next case.
5. Forgetting `default:` → unknown operators go unhandled.

Pay attention to these, and your calculator will be solid.

93.10 Why It Matters

This project shows how small, clear logic can build something genuinely useful. It's an excellent exercise in control flow, input validation, and user interaction.

You've built your own math tool, one you fully understand.

Try It Yourself

1. Add support for modulus (`%`) with integers.
2. Add exponentiation (use `pow()` from `<math.h>`).
3. Display all operations in one loop until user quits.
4. Use a function `calculate(a, b, op)` for cleaner structure.
5. Add error messages for invalid input or bad operators.

With each improvement, you'll turn your simple calculator into a more polished tool, one step closer to real-world software.

94. Mini Project 4: File Copy Utility

Now that you're comfortable reading and writing files, let's build something truly practical, a File Copy Utility.

This program copies the contents of one file into another, just like the `cp` command on Linux or the "Copy → Paste" action in your file explorer.

It's simple, yet powerful: you'll learn how to read from one file, write to another, and handle errors safely.

94.1 Project Goal

Your program should:

1. Take two filenames from the command line, the source and the destination.
2. Open the source file for reading.
3. Open (or create) the destination file for writing.
4. Copy all the contents from source to destination.
5. Close both files and confirm success.

94.2 Why This Project Matters

File copying is one of the most common tasks in programming. It combines everything you've learned about file I/O, error handling, loops, and command-line arguments, all in one program.

And best of all, it's a real, useful tool!

94.3 Plan the Steps

Let's break it down step by step:

1. Check that the user provided two filenames.
2. Open the source file (`"r"`) and the destination file (`"w"`).
3. Read the source file one character at a time using `fgetc`.
4. Write each character to the destination using `fputc`.
5. Close both files and print a success message.

Simple, right?

94.4 Handling Input

We'll start with checking command-line arguments:

```
if (argc < 3) {
    printf("Usage: %s <source> <destination>\n", argv[0]);
    return 1;
}
```

That ensures we always have both filenames ready.

94.5 Opening Files Safely

Next, open both files carefully:

```
FILE *src = fopen(argv[1], "r");
if (src == NULL) {
    perror("Error opening source file");
    return 1;
}

FILE *dst = fopen(argv[2], "w");
if (dst == NULL) {
    perror("Error opening destination file");
    fclose(src);
    return 1;
}
```

Always check for errors, missing files, wrong permissions, etc. If opening the destination fails, remember to close the source file first.

94.6 Copying Data

Now, let's copy the contents character by character:

```
int ch;
while ((ch = fgetc(src)) != EOF) {
    fputc(ch, dst);
}
```

This reads one byte at a time and writes it immediately. It works for any text file, and even small binary files.

94.7 Closing and Confirming

Once copying is done, close both files:

```
fclose(src);
fclose(dst);

printf("File copied successfully from %s to %s\n", argv[1], argv[2]);
```

That's it, your copy utility is complete!

94.8 Full Program

Here's the full code:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <source> <destination>\n", argv[0]);
        return 1;
    }

    FILE *src = fopen(argv[1], "r");
    if (src == NULL) {
        perror("Error opening source file");
        return 1;
    }

    FILE *dst = fopen(argv[2], "w");
    if (dst == NULL) {
        perror("Error opening destination file");
        fclose(src);
        return 1;
    }

    int ch;
    while ((ch = fgetc(src)) != EOF) {
        fputc(ch, dst);
    }
```

```
    fclose(src);
    fclose(dst);

    printf("File copied successfully from %s to %s\n", argv[1], argv[2]);

    return 0;
}
```

94.9 Example Run

```
$ ./filecopy input.txt output.txt
File copied successfully from input.txt to output.txt
```

If `input.txt` contains:

```
Hello world!
```

Then `output.txt` will now have exactly the same contents.

94.10 Tiny Improvements

Once your basic version works, you can make it more robust:

- Copy binary files too → use `"rb"` and `"wb"` modes.
- Show progress by counting bytes copied.
- Print file sizes before and after copying.
- Add error messages if read or write fails.
- Ask before overwriting an existing file.

Each small upgrade makes it closer to a real utility.

Why It Matters

This project gives you hands-on experience with file manipulation, one of the most common real-world programming tasks. You're building something every operating system depends on: safe, reliable file operations.

Try It Yourself

1. Copy a small text file and compare contents.
2. Add a byte counter (`int bytes = 0;`) and print how many were copied.
3. Modify the program to handle binary files using `"rb"` and `"wb"`.
4. Test with a large file, see how fast it runs!
5. Try error cases, missing file, no permissions, etc.

Once you've done this, you'll know exactly how a copy command works behind the scenes, because you built one yourself!

95. Mini Project 5: Simple Logger

Let's build another handy tool, a Simple Logger.

Logging is one of the most common patterns in programming. It's how programs record what happened, messages, errors, or progress, into a file so you can review them later.

In this project, you'll create a small program that appends messages to a log file with timestamps. You'll learn how to open files in append mode, work with time, and handle repeated writes safely.

95.1 Project Goal

Your logger will:

1. Ask the user for a message.
2. Add the message to a log file (`log.txt`).
3. Prepend a timestamp to each message.
4. Keep all previous entries (append, not overwrite).
5. Allow multiple entries in one run.

You'll end up with a file full of useful logs, like a little notebook for your program.

95.2 What You'll Practice

- File I/O (open, write, close)
- Append mode (`"a"`)
- Time functions from `<time.h>`
- Loops and user input

You've already seen all of these before, now we'll combine them into something practical.

95.3 Opening the Log File

We'll open the file in append mode, which means new lines are added to the end without erasing the old ones.

```
FILE *log = fopen("log.txt", "a");
if (log == NULL) {
    perror("Error opening log file");
    return 1;
}
```

If the file doesn't exist, `"a"` mode creates it automatically.

95.4 Getting the Current Time

We'll add a timestamp for each entry using `<time.h>`:

```
#include <time.h>

time_t now = time(NULL);
struct tm *t = localtime(&now);
fprintf(log, "[%04d-%02d-%02d %02d:%02d:%02d] ",
        t->tm_year + 1900,
        t->tm_mon + 1,
        t->tm_mday,
        t->tm_hour,
        t->tm_min,
        t->tm_sec);
```

This prints the date and time in a readable format, like:

```
[2025-10-02 09:15:30]
```

Perfect for a log entry.

95.5 Reading a Message

We'll use `fgets()` to read the message (it handles spaces too):

```c
char message[256];
printf("Enter a log message (or 'quit' to stop): ");
fgets(message, sizeof(message), stdin);
```

95.6 Writing to the Log

Once we have the timestamp and message, we just write them:

```c
fprintf(log, "%s", message);
```

You can add a newline if needed, `fgets` usually includes it.

Then, close the file at the end:

```c
fclose(log);
```

95.7 Putting It All Together

Here's the full logger:

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

int main(void) {
    char message[256];

    while (1) {
        printf("Enter a log message (or 'quit' to stop): ");
        fgets(message, sizeof(message), stdin);

        // Remove trailing newline
        message[strcspn(message, "\n")] = '\0';

        if (strcmp(message, "quit") == 0)
            break;

        FILE *log = fopen("log.txt", "a");
        if (log == NULL) {
            perror("Error opening log file");
```

```
            return 1;
        }

        time_t now = time(NULL);
        struct tm *t = localtime(&now);

        fprintf(log, "[%04d-%02d-%02d %02d:%02d:%02d] %s\n",
                t->tm_year + 1900,
                t->tm_mon + 1,
                t->tm_mday,
                t->tm_hour,
                t->tm_min,
                t->tm_sec,
                message);

        fclose(log);

        printf("Logged: %s\n", message);
    }

    printf("Goodbye! Check log.txt for your messages.\n");
    return 0;
}
```

95.8 Example Run

```
Enter a log message (or 'quit' to stop): Program started
Logged: Program started
Enter a log message (or 'quit' to stop): Something went wrong
Logged: Something went wrong
Enter a log message (or 'quit' to stop): quit
Goodbye! Check log.txt for your messages.
```

log.txt now contains:

```
[2025-10-02 09:15:30] Program started
[2025-10-02 09:15:45] Something went wrong
```

95.9 Tiny Improvements

Try adding:

- A custom filename (`./logger mylog.txt`)
- Levels like INFO, WARNING, ERROR
- Session header when program starts
- Log rotation (create new file if too large)

Each idea helps you learn how real-world loggers evolve.

95.10 Common Mistakes

1. Forgetting `"a"` mode, `"w"` will erase your log!
2. Not closing the file after each write, data may not be saved.
3. Forgetting to strip the newline from `fgets()`.
4. Not handling `quit`, infinite loop!

Watch out for these, and your logger will be reliable and clean.

Why It Matters

Logging turns a silent program into one that tells its story. It's how developers track what's happening inside, for debugging, monitoring, and auditing.

With this simple tool, you can record events, errors, or notes, a small step toward professional software design.

Try It Yourself

1. Run the logger and add a few entries.
2. Open `log.txt` and confirm the format.
3. Add a `[INFO]`, `[WARN]`, or `[ERROR]` tag before the message.
4. Ask the user for a custom filename.
5. Combine this with earlier projects, e.g., log game results or file copies.

You've just built your own logging system, simple, safe, and endlessly useful.

96. Mini Project 6: Contact Book

Let's build something a bit more like an application, a Contact Book.

This project teaches you how to store, search, and display structured data, a list of names, phone numbers, and emails, using structures, arrays, and file storage.

By the end, you'll have a small program that can add, list, and save contacts to a file. Think of it as a simple digital notebook built entirely in C.

96.1 Project Goal

Your Contact Book will:

1. Store a list of contacts (name, phone, email).
2. Let the user add new contacts.
3. List all saved contacts.
4. Save contacts to a file.
5. Load contacts when the program starts.

This combines structs, arrays, files, and menus, everything you've learned so far.

96.2 Designing the Structure

Each contact has three pieces of data: a name, a phone number, and an email. Let's define a structure to hold them:

```
struct Contact {
    char name[50];
    char phone[20];
    char email[50];
};
```

We'll keep an array of contacts in memory, like:

```
struct Contact contacts[100];
int count = 0;
```

This gives space for up to 100 entries, enough for a starter project.

96.3 Showing the Menu

The user will see a simple text menu:

```
1. Add new contact
2. List contacts
3. Save and exit
```

We'll use a loop and a `switch` to handle choices.

96.4 Adding a Contact

Here's how we'll gather info:

```
printf("Enter name: ");
fgets(contacts[count].name, sizeof(contacts[count].name), stdin);
contacts[count].name[strcspn(contacts[count].name, "\n")] = '\0';

printf("Enter phone: ");
fgets(contacts[count].phone, sizeof(contacts[count].phone), stdin);
contacts[count].phone[strcspn(contacts[count].phone, "\n")] = '\0';

printf("Enter email: ");
fgets(contacts[count].email, sizeof(contacts[count].email), stdin);
contacts[count].email[strcspn(contacts[count].email, "\n")] = '\0';

count++;
```

We remove the trailing newline from `fgets` so the strings are clean.

96.5 Listing Contacts

A simple loop prints all contacts:

```
for (int i = 0; i < count; i++) {
    printf("%d. %s | %s | %s\n", i + 1,
           contacts[i].name,
           contacts[i].phone,
           contacts[i].email);
}
```

If there are none yet, show a friendly message:

```
if (count == 0)
    printf("No contacts found.\n");
```

96.6 Saving to File

We'll save the contacts to a text file called `contacts.txt`:

```
FILE *f = fopen("contacts.txt", "w");
if (f == NULL) {
    perror("Error saving file");
    return 1;
}

for (int i = 0; i < count; i++) {
    fprintf(f, "%s;%s;%s\n",
            contacts[i].name,
            contacts[i].phone,
            contacts[i].email);
}

fclose(f);
```

We use `;` to separate fields so they're easy to parse later.

96.7 Loading from File

When the program starts, it can read back previous contacts:

```
FILE *f = fopen("contacts.txt", "r");
if (f != NULL) {
    while (fscanf(f, "%49[^;];%19[^;];%49[^\n]\n",
                  contacts[count].name,
                  contacts[count].phone,
                  contacts[count].email) == 3) {
        count++;
    }
    fclose(f);
}
```

This reads each line and fills your array again, simple persistence!

96.8 Full Program

Here's the complete version:

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

struct Contact {
    char name[50];
    char phone[20];
    char email[50];
};

int main(void) {
    struct Contact contacts[100];
    int count = 0;
    int choice;

    // Load contacts
    FILE *f = fopen("contacts.txt", "r");
    if (f != NULL) {
        while (fscanf(f, "%49[^;];%19[^;];%49[^\n]\n",
                      contacts[count].name,
                      contacts[count].phone,
                      contacts[count].email) == 3) {
            count++;
        }
        fclose(f);
    }

    while (1) {
        printf("\n--- Contact Book ---\n");
        printf("1. Add new contact\n");
        printf("2. List contacts\n");
        printf("3. Save and exit\n");
        printf("Choose an option: ");
        scanf("%d", &choice);
        getchar(); // clear newline

        if (choice == 1) {
            if (count >= 100) {
                printf("Contact list full!\n");
                continue;
```

```
            }
            printf("Enter name: ");
            fgets(contacts[count].name, sizeof(contacts[count].name), stdin);
            contacts[count].name[strcspn(contacts[count].name, "\n")] = '\0';

            printf("Enter phone: ");
            fgets(contacts[count].phone, sizeof(contacts[count].phone), stdin);
            contacts[count].phone[strcspn(contacts[count].phone, "\n")] = '\0';

            printf("Enter email: ");
            fgets(contacts[count].email, sizeof(contacts[count].email), stdin);
            contacts[count].email[strcspn(contacts[count].email, "\n")] = '\0';

            count++;
            printf("Contact added!\n");
        } else if (choice == 2) {
            if (count == 0) {
                printf("No contacts found.\n");
            } else {
                printf("\n--- Contact List ---\n");
                for (int i = 0; i < count; i++) {
                    printf("%d. %s | %s | %s\n", i + 1,
                           contacts[i].name,
                           contacts[i].phone,
                           contacts[i].email);
                }
            }
        } else if (choice == 3) {
            FILE *out = fopen("contacts.txt", "w");
            if (out == NULL) {
                perror("Error saving file");
                return 1;
            }
            for (int i = 0; i < count; i++) {
                fprintf(out, "%s;%s;%s\n",
                        contacts[i].name,
                        contacts[i].phone,
                        contacts[i].email);
            }
            fclose(out);
            printf("Contacts saved. Goodbye!\n");
            break;
```

```
        } else {
            printf("Invalid option. Try again.\n");
        }
    }

    return 0;
}
```

96.9 Example Run

```
--- Contact Book ---
1. Add new contact
2. List contacts
3. Save and exit
Choose an option: 1
Enter name: Alice
Enter phone: 123-456
Enter email: alice@example.com
Contact added!

--- Contact Book ---
1. Add new contact
2. List contacts
3. Save and exit
Choose an option: 2
--- Contact List ---
1. Alice | 123-456 | alice@example.com
```

96.10 Tiny Improvements

You can make your Contact Book even better:

- Add a search function (by name).
- Allow deleting contacts.
- Save in CSV or JSON format.
- Store more fields (address, notes).
- Sort by name before listing.

Each small step makes it closer to a real app.

Why It Matters

You've now built your first data-driven program, one that reads, stores, and saves real information. This is the heart of all database systems, from phone apps to contact managers.

You're not just coding anymore, you're building software that remembers.

Try It Yourself

1. Add a search option to find contacts by name.
2. Add a delete option using an index number.
3. Sort the list alphabetically before displaying.
4. Save to a custom filename entered by the user.
5. Limit contact names to unique entries (no duplicates).

This project is a big leap, a true foundation for working with structured data and files.

97. Mini Project 7: Matrix Operations

Time to step into the world of mathematical programming, let's build a Matrix Operations tool.

In this project, you'll create a small program that performs basic operations on 2D matrices: addition, subtraction, and multiplication.

It's a great exercise to practice arrays, loops, and functions, and to see how math comes alive through code.

97.1 Project Goal

Your program will:

1. Ask for the size of the matrices (rows and columns).
2. Ask the user to input two matrices.
3. Perform operations (add, subtract, multiply).
4. Display the results neatly.

You'll get to see how code can work just like a calculator for grids of numbers.

97.2 What's a Matrix?

A matrix is a rectangular array of numbers. Example (2×3 matrix):

```
1 2 3
4 5 6
```

You can think of it like a 2D array: `matrix[row][column]`

C handles these easily with nested arrays.

97.3 Declaring Matrices

Let's set up some matrices. For simplicity, we'll limit them to size 10×10:

```
int A[10][10], B[10][10], C[10][10];
int rows, cols;
```

Then ask the user for the size:

```
printf("Enter rows and columns (max 10): ");
scanf("%d %d", &rows, &cols);
```

97.4 Inputting Matrix Values

Use nested loops to read each element:

```
printf("Enter elements of Matrix A:\n");
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("A[%d][%d]: ", i, j);
        scanf("%d", &A[i][j]);
    }
}

printf("Enter elements of Matrix B:\n");
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("B[%d][%d]: ", i, j);
        scanf("%d", &B[i][j]);
    }
}
```

This gives you two matrices filled with user input.

97.5 Displaying a Matrix

Let's write a helper function to print any matrix:

```
void printMatrix(int M[10][10], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%4d", M[i][j]);
        }
        printf("\n");
    }
}
```

Now you can call `printMatrix(A, rows, cols)` whenever you want to show results.

97.6 Adding and Subtracting

Addition and subtraction are element-wise:

```
C[i][j] = A[i][j] + B[i][j];
C[i][j] = A[i][j] - B[i][j];
```

Code:

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        C[i][j] = A[i][j] + B[i][j];
    }
}
```

You can wrap this in a function like `addMatrices()`.

97.7 Multiplying Matrices

Matrix multiplication is trickier. For multiplication, the columns of A must equal the rows of B. If A is (m × n), B must be (n × p), and result C is (m × p).

Formula:

```
C[i][j] = sum(A[i][k] * B[k][j]) for k = 0..n-1
```

Code:

```c
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        C[i][j] = 0;
        for (int k = 0; k < cols; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

97.8 Full Program

Here's a simple version that performs addition, subtraction, and multiplication:

```c
#include <stdio.h>

void printMatrix(int M[10][10], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%4d", M[i][j]);
        }
        printf("\n");
    }
}

int main(void) {
    int A[10][10], B[10][10], C[10][10];
    int rows, cols;

    printf("Enter rows and columns (max 10): ");
    scanf("%d %d", &rows, &cols);
```

```
    printf("Enter elements of Matrix A:\n");
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            scanf("%d", &A[i][j]);

    printf("Enter elements of Matrix B:\n");
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            scanf("%d", &B[i][j]);

    // Addition
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            C[i][j] = A[i][j] + B[i][j];
    printf("\nMatrix Addition:\n");
    printMatrix(C, rows, cols);

    // Subtraction
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            C[i][j] = A[i][j] - B[i][j];
    printf("\nMatrix Subtraction:\n");
    printMatrix(C, rows, cols);

    // Multiplication
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++) {
            C[i][j] = 0;
            for (int k = 0; k < cols; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    printf("\nMatrix Multiplication:\n");
    printMatrix(C, rows, cols);

    return 0;
}
```

97.9 Example Run

```
Enter rows and columns (max 10): 2 2
Enter elements of Matrix A:
```

```
1 2
3 4
Enter elements of Matrix B:
5 6
7 8

Matrix Addition:
   6   8
  10  12

Matrix Subtraction:
  -4  -4
  -4  -4

Matrix Multiplication:
  19  22
  43  50
```

97.10 Tiny Improvements

- Add menu options for which operation to perform.
- Save results to a file.
- Handle rectangular (non-square) matrices.
- Support scalar multiplication (multiply by a number).
- Display formatted output (row by row).

Each upgrade helps you understand both math and memory layout in C.

Why It Matters

This project shows how data structures, loops, and logic combine to solve real math problems. You're building a mini tool that's part of the foundation of scientific computing, graphics, and machine learning.

Try It Yourself

1. Add a menu to choose between add, subtract, multiply.
2. Add a function `inputMatrix()` to reduce repeated code.
3. Support matrices of different shapes for multiplication.
4. Save results into a file called `matrix_output.txt`.
5. Try extending to floating-point matrices (`double` instead of `int`).

With this project, you're not just manipulating numbers, you're building a tiny math engine of your own!

98. Mini Project 8: JSON-like Parser

Let's take a small step into parsing, the process of reading structured text and turning it into data a program can use.

In this project, you'll build a simple JSON-like parser. It won't handle every detail of JSON, but it will read key–value pairs from a file and store them in memory.

By the end, you'll have a program that can load a configuration-style file like this:

```
name: Alice
age: 25
language: C
```

…and print out the keys and values neatly.

This is your first taste of how real programs read settings, configs, and even APIs!

98.1 Project Goal

You'll build a parser that:

1. Reads a text file line by line.
2. Splits each line into a key and value at `:`.
3. Stores them in an array of structs.
4. Prints all key–value pairs.

This is a simple, human-readable data format, perfect for beginners to parsing.

98.2 Why JSON-like

Real JSON uses braces and quotes, like `{ "name": "Alice" }`. But parsing JSON fully is a big job. So we'll start simpler, one `key: value` per line, then later you can extend it.

98.3 Data Structure

We'll store each key–value pair in a struct:

```c
struct Entry {
    char key[50];
    char value[100];
};
```

Then keep them in an array:

```c
struct Entry entries[100];
int count = 0;
```

98.4 Input File Example

Save this to a file called `data.txt`:

```
name: Alice
age: 25
language: C
editor: vim
```

Your program will read it, store each pair, and print them out.

98.5 Reading the File

Open it safely:

```c
FILE *f = fopen("data.txt", "r");
if (f == NULL) {
    perror("Error opening file");
    return 1;
}
```

Now we can read each line with `fgets`:

```c
char line[200];
while (fgets(line, sizeof(line), f) != NULL) {
    // process each line
}
```

98.6 Splitting Key and Value

Each line has `key: value`. We can use `strtok` to split:

```
char *key = strtok(line, ":");
char *value = strtok(NULL, "\n");
```

We'll clean up extra spaces:

```
if (key) {
    while (*key == ' ') key++; // skip spaces
}
if (value) {
    while (*value == ' ') value++; // skip spaces
}
```

Then store:

```
strcpy(entries[count].key, key);
strcpy(entries[count].value, value);
count++;
```

98.7 Printing Results

After reading all lines:

```
printf("\nParsed key-value pairs:\n");
for (int i = 0; i < count; i++) {
    printf("%s = %s\n", entries[i].key, entries[i].value);
}
```

That's it, a simple parser!

98.8 Full Program

Here's the complete code:

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

struct Entry {
    char key[50];
    char value[100];
};

int main(void) {
    struct Entry entries[100];
    int count = 0;

    FILE *f = fopen("data.txt", "r");
    if (f == NULL) {
        perror("Error opening file");
        return 1;
    }

    char line[200];
    while (fgets(line, sizeof(line), f) != NULL) {
        char *key = strtok(line, ":");
        char *value = strtok(NULL, "\n");

        if (key && value) {
            while (*key == ' ') key++;
            while (*value == ' ') value++;
            strcpy(entries[count].key, key);
            strcpy(entries[count].value, value);
            count++;
        }
    }

    fclose(f);

    printf("\nParsed key-value pairs:\n");
    for (int i = 0; i < count; i++) {
        printf("%s = %s\n", entries[i].key, entries[i].value);
    }

    return 0;
}
```

98.9 Example Run

If `data.txt` contains:

```
name: Alice
age: 25
language: C
```

You'll see:

```
Parsed key-value pairs:
name = Alice
age = 25
language = C
```

98.10 Tiny Improvements

Once your parser works, try these upgrades:

- Ask for a filename (`./parser settings.txt`)
- Ignore empty lines or comments starting with `#`
- Save parsed entries to another file
- Let users search by key
- Trim whitespace more carefully

Each feature brings you closer to a real-world config reader.

Why It Matters

Parsing is how programs understand text. From config files to JSON APIs, the same core idea applies, read, split, store, and use.

You're now building a foundation for handling structured data, one of the most important skills in programming.

Try It Yourself

1. Add support for comments starting with `#`.
2. Skip blank lines.
3. Ask for a key and print its value if found.
4. Add error handling for malformed lines.
5. Save results to a `parsed.txt` file.

You've just built your first parser, simple, useful, and a great step toward working with real-world data formats.

99. Mini Project 9: Mini Shell

Now it's time to make something that feels truly interactive, a Mini Shell.

A shell is a program that takes user commands, runs them, and shows the results. You've already been using one every time you type commands like `gcc main.c` or `ls` in your terminal.

In this project, you'll build a small version of that, a program that reads commands from the user and executes them using C's system calls.

It's a fantastic way to practice loops, strings, and system interaction.

99.1 Project Goal

Your mini shell will:

1. Display a prompt (like `$`).
2. Read a command from the user.
3. Run the command using the system.
4. Repeat until the user types `exit`.

It's simple, powerful, and gives you a peek behind the curtain of how real shells like bash or zsh work.

99.2 What You'll Practice

- Working with strings and `fgets`
- Using the system() function
- Building a command loop
- Handling special commands like `exit`

This project shows how a small program can act as a gateway to the whole operating system.

99.3 Starting with the Prompt

Let's start with a friendly prompt and input loop:

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

int main(void) {
    char command[100];

    while (1) {
        printf("$ ");                    // prompt
        fgets(command, sizeof(command), stdin);

        // remove newline at end
        command[strcspn(command, "\n")] = '\0';

        // check for exit
        if (strcmp(command, "exit") == 0)
            break;

        // execute the command
        system(command);
    }

    printf("Goodbye!\n");
    return 0;
}
```

That's already a working mini shell!

99.4 Example Run

```
$ ls
main.c  notes.txt  program
$ date
Thu Oct  2 10:30:21 2025
$ echo Hello World
Hello World
$ exit
Goodbye!
```

Every line you type is passed to your operating system's shell via `system()`, your program is acting as a middleman.

99.5 Understanding `system()`

The `system()` function runs any command exactly as if you'd typed it in a normal terminal.

```
system("ls");
system("gcc main.c -o main");
```

It's great for quick experiments, but always be careful with untrusted input in real-world apps.

99.6 Adding a Welcome Message

Let's greet the user:

```
printf("Welcome to MiniShell! Type 'exit' to quit.\n");
```

So the program feels more complete.

99.7 Handling Empty Input

If the user presses Enter without typing anything, we don't need to run a command. Add a quick check:

```
if (strlen(command) == 0)
    continue;
```

This avoids running blank lines.

99.8 Ignoring Leading Spaces

We can also skip leading spaces:

```
char *cmd = command;
while (*cmd == ' ') cmd++;
if (*cmd == '\0') continue;
system(cmd);
```

This small fix makes your shell a bit smarter.

99.9 Full Program

Here's your improved Mini Shell:

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

int main(void) {
    char command[100];
    printf("Welcome to MiniShell! Type 'exit' to quit.\n");

    while (1) {
        printf("$ ");
        fgets(command, sizeof(command), stdin);
        command[strcspn(command, "\n")] = '\0';

        // Skip empty input
        char *cmd = command;
        while (*cmd == ' ') cmd++;
        if (*cmd == '\0')
            continue;

        if (strcmp(cmd, "exit") == 0)
            break;

        system(cmd);
    }

    printf("Goodbye!\n");
    return 0;
}
```

99.10 Tiny Improvements

Try enhancing your shell with new features:

- Built-in commands: implement `help`, `clear`, or `version`.
- History: store recent commands in an array.
- Custom prompt: show username or path.
- Error handling: check `system()` return values.
- Chaining: allow `;` to run multiple commands.

Each addition teaches you more about how real shells are built.

Why It Matters

This project connects C with your operating system. You're using C not just for math or data, but to talk directly to the machine.

This is the essence of systems programming, giving you control over how software and the OS interact.

Try It Yourself

1. Add a `help` command that lists built-in features.
2. Create a `clear` command that runs `system("clear")`.
3. Count how many commands the user has run.
4. Print the current working directory in the prompt.
5. Combine commands like `echo hi; date`.

You've just built your first interactive shell, small but mighty. Every line you type goes straight from your code to your computer, and that's a powerful feeling.

100. Mini Project 10: Tiny HTTP Server

You've come a long way, now let's finish with something truly exciting: a Tiny HTTP Server.

This project will show you how to make your computer respond to web requests, just like a real website does! You'll learn how servers listen on a port, accept connections, and send back responses, all using plain C.

Don't worry, we'll keep it simple. By the end, you'll be able to open your browser, type `http://localhost:8080`, and see a message served by your C program.

100.1 Project Goal

Your tiny server will:

1. Open a network socket on port 8080.
2. Wait for a browser (or client) to connect.
3. Read the incoming request.
4. Send a simple HTTP response.
5. Close the connection.

It's a small step into network programming, and your first taste of backend development.

100.2 What You'll Learn

- How sockets let programs talk over the network
- How to read and write data between server and client
- What an HTTP request and response look like
- How to test your server with a browser

You'll see how low-level code powers every web request.

100.3 What Is HTTP?

HTTP (HyperText Transfer Protocol) is the language of the web.

When you visit a page, your browser sends a request:

```
GET / HTTP/1.1
Host: localhost
```

And the server replies with a response:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, world!
```

We'll build the simplest possible server that does exactly this.

100.4 Including the Right Headers

Networking in C uses `<sys/socket.h>` and `<netinet/in.h>`:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
```

You'll need these to open sockets and handle connections.

100.5 Setting Up the Server Socket

Let's start a server on port 8080:

```
int server_fd = socket(AF_INET, SOCK_STREAM, 0);
if (server_fd == -1) {
    perror("socket failed");
    return 1;
}
```

Then set up the server address:

```
struct sockaddr_in address;
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(8080);
```

Bind the socket and start listening:

```
if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
    perror("bind failed");
    return 1;
}

if (listen(server_fd, 3) < 0) {
    perror("listen failed");
    return 1;
}
```

100.6 Accepting a Connection

Once a client connects (like your browser), accept it:

```
int addrlen = sizeof(address);
int new_socket = accept(server_fd, (struct sockaddr*)&address, (socklen_t*)&addrlen);
if (new_socket < 0) {
    perror("accept failed");
    return 1;
}
```

Now you can read what the client sent!

100.7 Reading the Request

We'll store the request in a buffer:

```
char buffer[1024] = {0};
read(new_socket, buffer, sizeof(buffer));
printf("Request:\n%s\n", buffer);
```

This shows the raw HTTP request from your browser, a great learning moment.

100.8 Sending the Response

Now send a minimal HTTP reply:

```
char response[] =
    "HTTP/1.1 200 OK\r\n"
    "Content-Type: text/plain\r\n"
    "Content-Length: 14\r\n"
    "\r\n"
    "Hello, world!\n";

write(new_socket, response, strlen(response));
```

Then close the connection:

```
close(new_socket);
```

And keep listening for the next client if you want.

100.9 Full Program

Here's the complete tiny HTTP server:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(void) {
```

```c
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // 1. Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket failed");
        return 1;
    }

    // 2. Bind to port 8080
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);

    if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
        perror("bind failed");
        return 1;
    }

    // 3. Start listening
    if (listen(server_fd, 3) < 0) {
        perror("listen failed");
        return 1;
    }

    printf("Tiny HTTP Server running on http://localhost:8080\n");

    // 4. Accept a single connection
    new_socket = accept(server_fd, (struct sockaddr*)&address, (socklen_t*)&addrlen);
    if (new_socket < 0) {
        perror("accept failed");
        return 1;
    }

    // 5. Read the request
    char buffer[1024] = {0};
    read(new_socket, buffer, sizeof(buffer));
    printf("Request received:\n%s\n", buffer);

    // 6. Send a response
```

```
    char response[] =
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/plain\r\n"
        "Content-Length: 14\r\n"
        "\r\n"
        "Hello, world!\n";
    write(new_socket, response, strlen(response));

    // 7. Close sockets
    close(new_socket);
    close(server_fd);

    printf("Response sent. Goodbye!\n");
    return 0;
}
```

100.10 Example Run

1. Compile it:

```
gcc tiny_http.c -o tiny_http
```

2. Run it:

```
./tiny_http
Tiny HTTP Server running on http://localhost:8080
```

3. Open your browser and go to:

```
http://localhost:8080
```

You'll see:

```
Hello, world!
```

Tiny Improvements

- Serve an HTML file instead of plain text.
- Handle multiple requests in a loop.
- Add a log message for each connection.
- Serve different responses for different URLs.
- Experiment with ports (e.g. 3000, 5000).

Each change brings you closer to a real web server.

Why It Matters

This is a huge milestone, your code just talked to a browser! You've stepped into network programming, the world of servers, APIs, and the internet itself.

Everything from simple websites to large cloud systems starts here.

Try It Yourself

1. Replace the text with a short HTML page.
2. Print the client's IP address.
3. Add a loop to handle more than one request.
4. Save each request into a log file.
5. Return different messages for `/hello` and `/bye`.

With this tiny server, you've closed the loop: from your terminal to the web. You've built software that listens, responds, and communicates, the heart of modern computing.