The Little Book of C

Version 0.4.0

Duc-Tam Nguyen

2025-10-26

Table of contents

Content **6**

The Book **7**

Chapter 1. Getting Started **8**

1. What C Is and Why It Still Matters 8
2. Installing a C Compiler (gcc, clang, tinycc) 9
3. Writing and Running Your First Program 11
4. Anatomy of a C Program 13
5. Using Headers and the Preprocessor 15
6. Compiling, Linking, and Executing 17
7. Common Errors and Warnings 19
8. Command Line Basics for C Developers 22
9. Setting Up a Minimal Project Structure 25

Chapter 2. Language Basics **28**

11. Data Types and Variables 28
12. Constants, Literals, and Enumerations 31
13. Operators and Expressions 33
14. Control Flow: if, else, switch 37
15. Loops: for, while, do-while 41
16. Functions and Parameters 46
17. Scope and Lifetime of Variables 50
18. Return Values and Function Signatures 54
19. Static vs Dynamic Linking of Code Units 57
20. Practice: Build a Simple Calculator 61

Chapter 3. Working with Memory **66**

21. Understanding Memory Layout (Stack, Heap, Data, Code) 66
22. Pointers and Addresses 69
23. Arrays and Pointer Arithmetic 72
24. Strings as Character Arrays 76
25. Dynamic Memory Allocation (malloc, calloc, realloc, free) 80
26. Memory Leaks and Undefined Behavior 85
27. const and volatile Qualifiers 89
28. Function Pointers and Callbacks 93

29. Deep vs Shallow Copies . . . 97
30. Practice: Manual Memory Management . . . 101

Chapter 4. Structuring Data **106**
31. Structures and Nested Structures . . . 106
32. Unions and Type Reuse . . . 111
33. Typedef and Code Clarity . . . 116
34. Bitfields and Memory Packing . . . 121
35. Enumerations Revisited . . . 125
36. Linked Lists from Scratch . . . 129
37. Stacks and Queues with Structs . . . 134
38. Hash Tables and Function Pointers . . . 139
39. Minimal Object-Oriented Design in C . . . 144
40. Practice: Build a Tiny Library System . . . 149

Chapter 5. Input, Output and Files **155**
41. Standard I/O and printf/scanf . . . 155
42. File Pointers and `fopen` / `fclose` . . . 159
43. Reading and Writing Binary Files . . . 163
44. Working with `stdin`, `stdout`, and `stderr` . . . 168
45. Buffered I/O with `fgets` and `fputs` . . . 173
46. Error Checking with `errno` and `perror` . . . 177
47. Command-Line Arguments (`argc`, `argv`) . . . 182
48. Reading Configuration Files . . . 187
49. Serializing Structs to Disk . . . 192
50. Practice: Build a Log Reader and Writer . . . 197

Chapter 6. Compilation and the build process **203**
51. From Source to Executable: The Compilation Pipeline . . . 203
52. The Preprocessor and Macros . . . 208
53. Conditional Compilation (`#if`, `#ifdef`, `#ifndef`) . . . 213
54. Inline Functions and Header Hygiene . . . 219
55. Makefiles and Build Automation . . . 226
56. Linking Multiple Files . . . 232
57. Static and Shared Libraries . . . 239
58. Compiler Flags and Optimization Levels . . . 246
59. Understanding the Object File . . . 252
60. Practice: Write Your Own Makefile . . . 258

Chapter 7. Working Close to the System **266**
61. System Calls and the Standard Library . . . 266
62. Process Creation (fork, exec, wait) . . . 272
63. File Descriptors and open/read/write . . . 280

64. Pipes and Redirection 286
65. Signals and Signal Handlers 294
66. Memory Mapping (mmap) 301
67. Time and Clock APIs 307
68. Environment Variables 313
69. Error Handling and Return Codes 318
70. Practice: Mini Shell in C 324

Chapter 8. Debugging, Testing and Profiling **332**
71. Debugging with gdb 332
72. Using Valgrind for Memory Checking 337
73. Assertions and Defensive Programming 343
74. Unit Testing in C 348
75. Logging Systems 355
76. Profiling with gprof 360
77. Common Undefined Behaviors 365
78. Crash Analysis and Core Dumps 370
79. Code Review Checklist for C Projects 376
80. Practice: Fix Memory and Logic Bugs 380

Chapter 9. Portable and Modern C **388**
81. The C Standard Timeline (C89 to C23) 388
82. Portability and Endianness 394
83. Inline Assembly and Hardware Access 399
84. Cross-Compilation 404
85. Threading with pthreads 408
86. Atomic Operations and Memory Models 415
87. Using C with Other Languages (FFI) 420
88. Safer Alternatives (Bounds Checking, `_Static_assert`, and Modern C Safety Tools) 426
89. Modern Style: Clean and Readable C 431
90. Practice: Portable Multithreaded Program 436

Chapter 10. Building Real Projects **441**
91. Designing Small C Libraries 441
92. Building a Command-Line Tool 445
93. Tiny HTTP Server (Sockets and Threads) 449
94. Simple Key-Value Store 454
95. Implementing a Custom Allocator 462
96. Writing a Text Parser 466
97. Tiny Interpreter for an Expression Language 474
98. Interfacing with SQLite or LevelDB 482
99. Packaging, Versioning, and Documentation 488

100. Practice: Build Your Own Mini Project 494
Epilogue. The Spirit of C 499
The Path Beyond 500
A Note from the Author 500
Final Exercise 501

Content

A concise, structured guide to the C programming language.

- Download PDF - print-ready
- Download EPUB - e-reader friendly
- View LaTex - `.tex` source
- Source code (Github) - Markdown source
- Read on GitHub Pages - view online

The Book

Chapter 1. Getting Started

1. What C Is and Why It Still Matters

C is the language that sits closest to the machine while still feeling human to write. It's not the newest or the easiest, but it's one of the most powerful. Every modern operating system, compiler, and database has a core written in C, from Linux and Git to Python's interpreter and even parts of your browser.

Learning C gives you something no other language can: an understanding of *how computers actually work*. You'll see how memory is managed, how data moves, how the CPU runs your code, and how everything you write turns into tiny instructions that the machine understands.

C teaches discipline. There's no garbage collector or safety net. You decide when to allocate memory, when to free it, and what happens when you forget. You learn precision and control, the same skills that make great programmers in any language.

Tiny Code

```c
#include <stdio.h>

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

Run this and you've done what every C programmer starts with, printing your first line of text to the screen. It's small, but it carries the spirit of C: direct, explicit, and clear.

Why It Matters

C is the foundation of all systems programming. When you understand it, higher-level languages make more sense. You'll see why compilers work the way they do, why memory errors happen, and how performance decisions ripple through an entire program.

Even if you never write production C code, the mindset it builds, careful reasoning, attention to detail, respect for the machine, will shape how you write code in any language.

Try It Yourself

1. Install a C compiler like `gcc` or `clang`.
2. Save the code above into a file named `hello.c`.
3. Compile it:

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

4. Run it:

   ```
   ./hello
   ```

5. Modify the message and try printing more lines. You've just built your first C program.

2. Installing a C Compiler (gcc, clang, tinycc)

Before you can write and run C programs, you need a compiler. A compiler is a tool that *translates your human-readable code* into the machine instructions that your CPU understands. In C, this process is explicit, you see it, control it, and learn from it.

There are many compilers available, but three are most common:

- GCC (GNU Compiler Collection)**, The standard compiler on Linux and macOS, known for reliability and wide support.
- **Clang**, A modern compiler built for speed, cleaner diagnostics, and integration with tools like LLVM.
- **TinyCC (tcc)**, A super-lightweight compiler that's perfect for learning and quick testing.

Tiny Code

You can check if you already have a compiler installed by running one of these commands in your terminal:

```
gcc --version
clang --version
tcc --version
```

If you see a version number, you're ready. If not, you'll need to install one.

Installing on Different Systems

Linux (Debian/Ubuntu):

```
sudo apt update
sudo apt install build-essential
```

This installs GCC along with other useful tools like `make`.

macOS (with Xcode Command Line Tools):

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

This installs Clang and the developer toolchain.

Windows (via Mingw-w64):

1. Go to Mingw-w64.
2. Download and install it.
3. Add the compiler's `bin` folder to your system PATH.
4. Open `cmd` or PowerShell and run `gcc --version` to confirm.

Or, if you prefer an all-in-one environment, install **WSL (Windows Subsystem for Linux)** and use the Linux commands above.

Why It Matters

Installing a compiler is your first step toward understanding *how programs become executables*. In C, there's no hidden build system or automatic runtime, everything that happens between writing code and running it is visible. That clarity is part of what makes C such a powerful learning tool.

When you install your compiler, you're also installing the ability to explore how software really works.

Try It Yourself

1. Open your terminal or command prompt.
2. Type `gcc --version` or `clang --version` to confirm installation.
3. Create a simple file named `test.c`:

```
int main(void) { return 0; }
```

4. Compile it:

```
gcc test.c -o test
```

5. Run it:

```
./test
```

If it runs with no output, that's perfect, your compiler is ready. You've just built your very first executable program from source code.

3. Writing and Running Your First Program

Now that your compiler is ready, it's time to write your first real C program. This is where the magic happens, you'll write plain text, compile it into machine instructions, and watch your computer follow your commands exactly.

C doesn't hide what's happening under the hood. Every step, writing, compiling, linking, running, is visible and under your control.

Tiny Code

Create a new file named `hello.c` and type this code:

```c
#include <stdio.h>

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

Then, compile and run it:

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

You should see:

```
Hello, world!
```

Breaking It Down

- `#include <stdio.h>` This tells the compiler to use the *Standard Input/Output* library, which provides the `printf` function.
- `int main(void)` Every C program starts with a `main` function. It's the entry point, where execution begins.
- `printf("Hello, world!\n");` This prints text to the screen. The `\n` means "newline," so the next output starts on a new line.
- `return 0;` When `main` returns 0, it tells the operating system that your program finished successfully.

Why It Matters

Your "Hello, world" may look simple, but it represents an entire process:

1. **The compiler** translates your text (`hello.c`) into object code (`hello.o`).
2. **The linker** combines that code with standard libraries.
3. **The executable** (`hello`) is pure machine instructions.
4. **The operating system** loads and runs it.

Understanding this flow is what makes C special, it's not just about writing code, but about knowing *how* code becomes software.

Try It Yourself

1. Change the message to `"Hello, C learner!"` and recompile.
2. Add another line:

   ```c
   printf("This is my first C program.\n");
   ```

3. Try leaving out the semicolon, what error does the compiler show?
4. Try removing `#include <stdio.h>`, what happens then?
5. Experiment and break things. Every error teaches you how C thinks.

You've just written and run your first C program, a direct conversation between you and the machine. From here, every new piece of code builds on this simple moment of control and understanding.

4. Anatomy of a C Program

Now that your first program runs, let's open it up and look inside. Every C program follows a clear structure, a set of rules that tells both *you* and the *compiler* what each part means. Understanding this structure early will help you read, write, and debug code with confidence.

Tiny Code

Here's the same program, with comments explaining each piece:

```c
#include <stdio.h>      // 1. Preprocessor directive

// 2. Function definition
int main(void) {        // main: entry point of every C program
    printf("Hello, C!\n");  // 3. Statement: prints a message
    return 0;           // 4. Return statement: signals success
}
```

The Four Main Parts

1. **Preprocessor Directives** Lines that begin with `#` are handled before the code is even compiled. They include or define things that your program depends on. Example:

   ```c
   #include <stdio.h>
   #define PI 3.14159
   ```

2. **Functions** Every C program is made of functions. The function `main()` is special, it's where your program starts. You can define more functions to organize your code.

3. **Statements** Each instruction inside a function ends with a semicolon. These are the steps your program takes, one by one.

4. **Comments** Comments are ignored by the compiler but read by humans. Use them to explain *why* your code does something, not just *what* it does.

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

Why It Matters

C is a *structured* language. Every function, statement, and declaration lives inside a clear boundary. Unlike scripting languages, there's no automatic setup or hidden runtime, everything you see is everything that runs.

Learning the anatomy of a C program gives you a mental map:

- You know where execution begins (`main`).
- You know where code lives (inside functions).
- You know where libraries come from (via includes).

Once this map becomes natural, reading even large C programs starts to feel easy and logical.

Try It Yourself

1. Create a file `structure.c` with this content:

```c
#include <stdio.h>

void greet(void) {
    printf("Welcome to C programming!\n");
}

int main(void) {
    greet();
    return 0;
}
```

2. Compile and run it:

```
gcc structure.c -o structure
./structure
```

3. Add another function, maybe `void bye(void)` that prints a goodbye message, and call it after `greet()`.

4. Try removing `return 0;`, notice how the program still runs, but adding it makes your intent clear.

Every C program you'll ever write follows this basic shape. Once you can recognize these parts, you can start building programs that are longer, smarter, and closer to the system.

5. Using Headers and the Preprocessor

Every C program begins before it even starts running, with something called the *preprocessor*. Before the compiler turns your code into machine instructions, the preprocessor prepares it: it pulls in files, replaces macros, and sets up everything your program needs. This step is what makes `#include <stdio.h>` work, and it's key to understanding how larger C projects are organized.

Tiny Code

```c
#include <stdio.h>   // include the standard input/output header
#define PI 3.14159    // define a constant macro

int main(void) {
    printf("PI is approximately %.2f\n", PI);
    return 0;
}
```

When you compile this program, the preprocessor replaces `PI` with `3.14159` and includes the contents of the file `stdio.h` before the compiler even starts.

You can see the preprocessed result by running:

```
gcc -E program.c
```

It will output a much longer version of your code, showing all the lines that `stdio.h` added behind the scenes.

How Headers Work

Headers are **declaration files**. They tell the compiler *what exists*, like functions, constants, and types, without actually providing the code (the *definitions*). For example, `stdio.h` declares the function `printf()` so the compiler knows how to call it.

There are two main ways to include headers:

- **System headers:**

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

  The compiler looks for these in standard library directories.

- **User-defined headers:**

```
#include "myutils.h"
```

  The compiler looks for these in your project folder first.

This separation keeps large programs modular and readable.

Why It Matters

The preprocessor is like the "setup crew" for your program. It doesn't run your code, it *prepares* it. By understanding headers and macros, you can:

- Split big programs into smaller, reusable parts.
- Define constants in one place instead of repeating values.
- Write portable code that compiles on different systems.
- Avoid subtle bugs caused by missing declarations.

When you write `#include <stdio.h>`, you're tapping into decades of reliable, shared code, one of the greatest strengths of the C ecosystem.

Try It Yourself

1. Create two files:

   mathutils.h

```
#ifndef MATHUTILS_H
#define MATHUTILS_H

#define SQUARE(x) ((x) * (x))

#endif
```

   main.c

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

int main(void) {
    int n = 5;
    printf("The square of %d is %d\n", n, SQUARE(n));
    return 0;
}
```

2. Compile and run:

```
gcc main.c -o main
./main
```

3. Try editing the macro in `mathutils.h` to add a `CUBE(x)` function, and use it in `main.c`.
4. Then run:

```
gcc -E main.c | less
```

   to explore the preprocessed code and see how includes and macros expand.

Once you grasp headers and preprocessing, you'll understand how large C codebases stay organized, and how a simple `#include` line can unlock an entire library of functionality.

6. Compiling, Linking, and Executing

When you press *Enter* to compile your C program, a lot happens behind the scenes. Your source code goes through several stages before it becomes a runnable executable. Understanding these steps is essential, it turns compilation errors from mysteries into simple, fixable clues.

Tiny Code

Let's start with a familiar program:

```c
#include <stdio.h>

int main(void) {
    printf("Learning the C build process!\n");
    return 0;
}
```

Now compile and run it step by step:

```
# Step 1: Compile to object file
gcc -c hello.c -o hello.o

# Step 2: Link object file into executable
gcc hello.o -o hello

# Step 3: Run the program
./hello
```

Output:

```
Learning the C build process!
```

The Four Stages of Building a C Program

1. **Preprocessing** The preprocessor handles all lines starting with `#`. It expands macros, includes headers, and prepares code for compilation. Command to inspect:

   ```
   gcc -E hello.c | less
   ```

2. **Compilation** The compiler translates the preprocessed code into *assembly language*, and then into *object code*. Each source file (like `hello.c`) becomes an object file (`hello.o`). Command:

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

3. **Linking** The linker combines all object files and libraries into one final executable. For example, `printf` comes from the C standard library (`libc`), so the linker connects your code to it. Command:

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

4. **Execution** Once linked, your binary (`hello`) is loaded by the operating system and executed by the CPU. Command:

   ```
   ./hello
   ```

Why It Matters

C gives you control over every stage of this process. Most modern languages hide compilation or linking, but in C, these steps are transparent and configurable. When something goes wrong, a missing function, an undefined symbol, or a broken include, you'll know exactly *which stage* to look at.

Mastering the build process also opens the door to deeper skills:

- Creating reusable libraries (`.a` or `.so` files)
- Understanding Makefiles and build automation
- Debugging with symbols and optimized builds
- Writing portable programs that compile cleanly anywhere

Every system programmer eventually learns to *think like a compiler*, and this is where that thinking begins.

Try It Yourself

1. **Experiment with separate compilation:**

   Create two files:

   main.c

   ```c
   #include <stdio.h>
   void greet(void);

   int main(void) {
       greet();
       return 0;
   }
   ```

   greet.c

   ```c
   #include <stdio.h>

   void greet(void) {
       printf("Hello from another file!\n");
   }
   ```

   Then compile and link:

   ```
   gcc -c main.c
   gcc -c greet.c
   gcc main.o greet.o -o greetprog
   ./greetprog
   ```

2. **Try breaking it:** Delete the `void greet(void);` line in `main.c` and recompile, see how the compiler warns you about an implicit declaration.

3. **Observe the stages:** Add flags like `-Wall -O2 -v` to see detailed messages from the compiler and linker.

Once you understand compilation and linking, you've unlocked one of the most powerful parts of C, the ability to control exactly *how* your software is built, combined, and executed.

7. Common Errors and Warnings

No C programmer avoids errors. In fact, the compiler's messages are your best teachers. Each warning or error is the compiler's way of saying, "Something here doesn't make sense yet." Learning to read and fix them early will save you hours later and make debugging a natural part of your process.

Tiny Code

Let's look at a few examples:

```c
#include <stdio.h>

int main(void) {
    int a = 5
    printf("The value of a is %d\n", a);
    return 0;
}
```

Try compiling it:

```
gcc error_demo.c -o error_demo
```

Output:

```
error_demo.c: In function ‘main':
error_demo.c:4:13: error: expected ‘;' before ‘printf'
```

This means the compiler found a missing semicolon. The message even tells you where (`line 4`) and why (`expected ';' before 'printf'`).

Fix it by adding the missing semicolon:

```c
int a = 5;
```

Then compile again, clean output means success.

Common Types of Errors

1. **Syntax Errors** These are the easiest to fix. You've broken a grammar rule. Example: missing `;`, mismatched braces `{}`, or incorrect parentheses.
2. **Type Errors** You're using variables or functions in a way that doesn't match their type. Example:

   ```c
   int x = "hello"; // error: assigning string to int
   ```

3. **Undeclared Identifiers** You're using a variable or function that the compiler hasn't seen yet. Example:

```
printf("Value: %d\n", number); // error: ‘number' undeclared
```

4. **Linker Errors** Compilation succeeds, but linking fails because something is missing. Example:

   ```
   undefined reference to `greet'
   ```

   This means the compiler saw a declaration but couldn't find the actual function definition.

5. **Warnings** Warnings don't stop compilation, but they often point to potential bugs. Example:

   ```
   warning: variable ‘x' set but not used
   ```

   Always pay attention to warnings, clean builds (`no warnings`) are a mark of quality code.

Why It Matters

Every programmer makes mistakes. What matters is how fast you can understand what the compiler is saying. In C, error messages are usually precise and honest, they tell you exactly what broke. By learning to interpret them, you're training yourself to debug with logic, not luck.

Good habits:

- Compile often, don't wait to write 100 lines before testing.
- Use `-Wall -Wextra` to enable all useful warnings.
- Read errors *top to bottom*, the first one often causes the rest.
- Fix warnings even if the code still runs.

Try It Yourself

1. **Forgetting a return type:**

   ```
   main() {
       printf("No return type!\n");
   }
   ```

   Compile with:

   ```
   gcc -Wall test.c
   ```

   You'll get:

   ```
   warning: return type defaults to ‘int'
   ```

2. **Using an undeclared variable:**

```c
#include <stdio.h>
int main(void) {
    printf("%d\n", x);
    return 0;
}
```

This will produce:

```
error: ‘x' undeclared
```

3. **Fix each one** until your program compiles cleanly with no warnings.

Errors are not failures, they're the compiler's way of guiding you toward understanding. The more errors you fix, the better you become at speaking the language of the machine.

8. Command Line Basics for C Developers

C was born in the Unix world, and the command line is its natural home. If you can move comfortably in the terminal, you'll understand what your tools are doing, compiling, linking, and running programs directly. This section gives you the essential commands you'll need to build and explore C projects like a real systems programmer.

Tiny Code

Let's start with a quick refresher using a simple file `hello.c`:

```c
#include <stdio.h>

int main(void) {
    printf("Hello from the terminal!\n");
    return 0;
}
```

To build and run it from the command line:

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

Output:

```
Hello from the terminal!
```

That's the full cycle: write → compile → run. Now let's look at the basic tools that make that process smoother.

Essential Command Line Tools

1. **`ls`** – List files in the current directory

   ```
   ls
   ```

   You'll see files like `hello.c`, `hello.o`, or `hello`.

2. **`pwd`** – Print the current working directory

   ```
   pwd
   ```

3. **`cd`** – Change directories

   ```
   cd projects/c_programs
   ```

4. **`cat`** – Display file contents quickly

   ```
   cat hello.c
   ```

5. **`rm`** – Remove files

   ```
   rm hello
   ```

6. **`clear`** – Clear your terminal screen

   ```
   clear
   ```

7. **`man`** – Read the manual for a command

   ```
   man gcc
   ```

   Press `q` to exit.

8. **`echo`** – Print a message or variable

   ```
   echo "Compiling C!"
   ```

9. **`touch`** – Create a new empty file

   ```
   touch main.c
   ```

10. **`gcc`** – Compile your C source code

```
gcc main.c -o main
```

Why It Matters

The command line isn't just for building code, it teaches you how your tools actually work. In C, there's no hidden environment running behind a button click. Each command you type does exactly one job, and understanding those jobs gives you full control.

- You see the **build process** directly.
- You control **where outputs go**.
- You can **chain commands** for automation.

This mindset, knowing what happens under the hood, is what makes C programmers comfortable working close to the machine.

Try It Yourself

1. **Navigate and build manually:**

   ```
   mkdir myfirstc
   cd myfirstc
   touch hello.c
   ```

   Add code with your favorite text editor (like `nano hello.c`), then compile and run.

2. **Use compiler flags:**

   ```
   gcc -Wall -O2 hello.c -o hello
   ./hello
   ```

   - `-Wall` enables warnings.
   - `-O2` applies optimization.

3. **Use output redirection:**

   ```
   ./hello > output.txt
   cat output.txt
   ```

4. **Explore commands interactively:**

   ```
   man ls
   man gcc
   ```

   Read a few lines, knowing how to find help is as important as coding itself.

C and the command line grew up together. Once you get comfortable typing and compiling by hand, you'll start to feel how programs, files, and processes fit together. That's the real start of systems programming, not just writing code, but *commanding the computer directly.*

9. Setting Up a Minimal Project Structure

As your C programs grow, you'll quickly outgrow the single-file "hello.c" style. Real projects are made of multiple source files, headers, and sometimes libraries. A clear folder structure keeps your work clean, easy to build, and easy to maintain. In this section, you'll create a small, organized layout, the same structure used by professionals.

Tiny Code

Here's a minimal project layout:

```
my_project/
   include/
       greet.h
   src/
       greet.c
   main.c
   Makefile
```

include/greet.h

```c
#ifndef GREET_H
#define GREET_H

void greet(const char *name);

#endif
```

src/greet.c

```c
#include <stdio.h>
#include "greet.h"

void greet(const char *name) {
    printf("Hello, %s!\n", name);
}
```

main.c

```c
#include "greet.h"

int main(void) {
    greet("C Learner");
    return 0;
}
```

Makefile

```
CC = gcc
CFLAGS = -Wall -Iinclude
SRC = main.c src/greet.c
OUT = my_program

$(OUT): $(SRC)
    $(CC) $(CFLAGS) $(SRC) -o $(OUT)

clean:
    rm -f $(OUT)
```

Now build and run:

```
make
./my_program
```

Output:

```
Hello, C Learner!
```

Understanding the Structure

1. **`include/`** Holds header files (`.h`), declarations of functions, constants, and types. You include these in `.c` files using quotes:
   ```c
   #include "greet.h"
   ```
2. **`src/`** Contains source files (`.c`) that implement functions declared in headers.
3. **`main.c`** The entry point of your program, this file usually just calls functions from `src/`.
4. **`Makefile`** Defines how to build the program. You can run `make` instead of typing long `gcc` commands.
5. **Output binary** The compiled executable (here `my_program`) stays in the project's root for convenience.

Why It Matters

A clear structure helps you:

- Keep code modular and reusable
- Separate interface (`.h`) from implementation (`.c`)
- Make compilation faster and easier to manage
- Avoid name clashes in large projects
- Scale from one file to dozens without confusion

Even small C utilities benefit from structure, you'll thank yourself later when you revisit your code.

Try It Yourself

1. Create the directories and files shown above.
2. Type each file carefully and run `make`.
3. Modify `greet.c` to print a personalized message, e.g.

   ```
   printf("Welcome back, %s!\n", name);
   ```

   then rebuild.
4. Add another pair of files, `src/farewell.c` and `include/farewell.h`, with a goodbye function, and call it from `main.c`.
5. Run `make clean` to delete the binary and rebuild fresh.

This small structure is the seed of every serious C project. Once you can organize your files this way, you're ready to grow into larger systems, libraries, tools, and applications that others can use and build upon.

Chapter 2. Language Basics

11. Data Types and Variables

In C, everything begins with types. A **type** tells the compiler how much memory to reserve, how to interpret the bits stored there, and what operations are allowed. Understanding types is the foundation of writing safe and efficient C programs, it's how you speak the computer's native language precisely.

Tiny Code

```c
#include <stdio.h>

int main(void) {
    int age = 25;              // integer
    float height = 1.75;       // floating-point number
    char initial = 'A';        // single character
    double weight = 68.4;      // double-precision number

    printf("Age: %d\n", age);
    printf("Height: %.2f\n", height);
    printf("Initial: %c\n", initial);
    printf("Weight: %.1lf\n", weight);

    return 0;
}
```

Output:

```
Age: 25
Height: 1.75
Initial: A
Weight: 68.4
```

Core Built-in Types

Type	Size (Typical)	Description	Format Specifier
`char`	1 byte	Single character	`%c`
`int`	4 bytes	Whole number	`%d`
`float`	4 bytes	Decimal number (single precision)	`%f`
`double`	8 bytes	Decimal number (double precision)	`%lf`
`void`	–	No value or type	–

> Sizes may vary depending on system and compiler, but the relationships remain consistent.

You can also control how numbers behave using **modifiers**:

Modifier	Example	Meaning
`short`	`short int x;`	Smaller integer (often 2 bytes)
`long`	`long int y;`	Larger integer (often 8 bytes)
`unsigned`	`unsigned int z;`	Only non-negative values

Example:

```
unsigned int score = 100;
long long big_number = 1234567890123LL;
```

Declaring and Initializing Variables

A variable is simply a *named piece of memory*. You declare it by specifying its type and name:

```
int count;       // declaration
count = 5;       // assignment
```

Or both together:

```
int count = 5;   // initialization
```

Multiple declarations:

```c
int x = 1, y = 2, z = 3;
```

Variables must be declared *before* you use them, and their type cannot change.

Why It Matters

C is a **statically typed language**, meaning every variable's type is known at compile time. This makes programs faster and safer, because the compiler can:

- Catch type mismatches early
- Optimize memory layout
- Predict storage and alignment

When you understand data types, you understand how your code maps directly to the machine's memory.

C forces you to think carefully about *what kind of data* you're working with, a skill that improves every program you write, in any language.

Try It Yourself

1. Create a program that stores and prints:
   - Your age as an `int`
   - Your height as a `float`
   - Your name's first letter as a `char`
2. Add two integers and print the result:
   ```c
   int a = 10, b = 20;
   printf("Sum: %d\n", a + b);
   ```
3. Try using an `unsigned int` and print what happens if you assign a negative value.
4. Use `sizeof()` to inspect how big each type is on your system:
   ```c
   printf("Size of int: %zu bytes\n", sizeof(int));
   ```

Every number, character, and pointer in C starts here, in the precise world of types and variables. Once you're fluent in these, memory layout, structs, and pointers will make perfect sense.

12. Constants, Literals, and Enumerations

C programs often rely on values that never change, numbers, characters, or named constants used throughout your code. Instead of sprinkling magic numbers everywhere, you can give them meaningful names and keep your program readable, safe, and easy to maintain.

Tiny Code

```c
#include <stdio.h>

#define PI 3.14159           // preprocessor constant
const int DAYS_IN_WEEK = 7;  // read-only variable

enum Direction { NORTH, EAST, SOUTH, WEST }; // enumerated constants

int main(void) {
    printf("Pi: %.2f\n", PI);
    printf("Days in a week: %d\n", DAYS_IN_WEEK);
    printf("Direction EAST has value: %d\n", EAST);
    return 0;
}
```

Output:

```
Pi: 3.14
Days in a week: 7
Direction EAST has value: 1
```

Constants in C

1. **Preprocessor constants (`#define`)** These are replaced *before* compilation. Think of them as text substitutions, not variables.

   ```c
   #define MAX_USERS 100
   printf("Max users: %d\n", MAX_USERS);
   ```

   - No memory is used.
   - No type checking, the compiler just replaces the text.

2. **Constant variables (`const`)** These are real variables stored in memory but cannot be modified after initialization.

```
const double SPEED_OF_LIGHT = 299792458.0;
```

   - Type safe.
   - Preferred for constants in modern C code.

3. **Literals** These are fixed values written directly in your code:

```
42          // integer literal
3.14        // floating-point literal
'A'         // character literal
"Hello"     // string literal
0xFF        // hexadecimal literal (255)
075         // octal literal (61)
```

Enumerations

An **enum** (short for *enumeration*) defines a set of named integer constants. They make your code self-documenting and prevent mistakes with raw numbers.

```
enum TrafficLight {
    RED,     // 0
    YELLOW,  // 1
    GREEN    // 2
};

int main(void) {
    enum TrafficLight signal = GREEN;
    if (signal == GREEN)
        printf("Go!\n");
    return 0;
}
```

You can also assign custom values:

```
enum Month {
    JAN = 1, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
};
```

Now `JAN` starts at 1, and each value increments automatically.

Why It Matters

Constants and enums make your code **clearer and safer**:

- You can change one definition instead of many numbers.
- The compiler can enforce that constants are not modified.
- Enumerations group related values into a meaningful set.

Without them, large programs become fragile and full of unexplained numbers, a maintenance nightmare.

Good C programmers use constants to express **intent**, not just values.

Try It Yourself

1. Replace every numeric literal in your old programs with a `#define` or `const`. Example:
   ```
   #define MAX_SCORE 100
   const float TAX_RATE = 0.08;
   ```
2. Create an `enum` for days of the week, and print `MONDAY` and `FRIDAY`.
3. Assign custom values in your enum (e.g. start with `SUNDAY = 1`).
4. Experiment: try changing `const int x = 5; x = 10;` , notice the compiler stops you from modifying it.
5. Use `printf` to print literal values in different formats:
   ```
   printf("%d %x %o\n", 255, 255, 255); // decimal, hex, octal
   ```

Constants are how you make your C programs *speak clearly*. They turn numbers into ideas, and that's what transforms code from working to understandable.

13. Operators and Expressions

Operators are the building blocks of computation in C. They let you perform arithmetic, compare values, manipulate bits, and combine logic, all in concise expressions. Once you understand how operators work and how they interact through *precedence* and *associativity*, you can write clear, efficient code that behaves exactly as you expect.

Tiny Code

```c
#include <stdio.h>

int main(void) {
    int a = 10, b = 3;

    printf("a + b = %d\n", a + b);   // addition
    printf("a - b = %d\n", a - b);   // subtraction
    printf("a * b = %d\n", a * b);   // multiplication
    printf("a / b = %d\n", a / b);   // integer division
    printf("a %% b = %d\n", a % b);  // remainder (modulo)

    a += 5; // same as a = a + 5
    printf("a after += 5: %d\n", a);

    return 0;
}
```

Output:

```
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
a after += 5: 15
```

Arithmetic Operators

Operator	Meaning	Example	Result
`+`	Addition	`4 + 3`	7
`-`	Subtraction	`10 - 6`	4
`*`	Multiplication	`2 * 5`	10
`/`	Division	`7 / 2`	3 (integer division)
`%`	Modulo (remainder)	`7 % 2`	1

Tip: If you use floating-point numbers (`float`, `double`), division produces decimals.

Relational and Logical Operators

Type	Operator	Example	Meaning
Relational	==	a == b	Equal
	!=	a != b	Not equal
	<, >, <=, >=	a < b	Comparison
Logical	&&	a && b	Logical AND
	| |	a | | b	Logical OR
	!	!a	Logical NOT

Example:

```
int age = 20;
if (age >= 18 && age <= 60)
    printf("Adult\n");
```

Increment and Decrement

```
int x = 5;
printf("%d\n", ++x); // prefix: increments, then uses value (6)
printf("%d\n", x++); // postfix: uses value, then increments (6)
printf("%d\n", x);   // final value is 7
```

Assignment and Compound Operators

Operator	Meaning	Example
=	Assignment	x = 10
+=	Add and assign	x += 2
-=	Subtract and assign	x -= 3
*=	Multiply and assign	x *= 4
/=	Divide and assign	x /= 5
%=	Modulo and assign	x %= 6

These save typing and make intent clearer.

Bitwise Operators

C gives you direct access to bits, useful for systems, embedded, or optimization tasks.

Operator	Meaning	Example
`&`	AND	`a & b`
`|`	OR	`a | b`
`^`	XOR	`a ^ b`
`~`	NOT	`~a`
`<<`	Left shift	`a << 1`
`>>`	Right shift	`a >> 2`

Example:

```c
int mask = 0b0010;
int num = 0b1011;
int result = num & mask;  // checks if the 2nd bit is set
```

Precedence and Associativity

When you write complex expressions, C follows operator precedence rules. For example:

```c
int result = 2 + 3 * 4; // result is 14, not 20
```

`*` has higher precedence than `+`, so it runs first.

Use parentheses to make intentions clear:

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

Why It Matters

Operators are where **logic meets the machine**. They translate mathematical ideas and control decisions into instructions the CPU executes directly. Understanding how expressions are built and evaluated helps you:

- Write compact, efficient code
- Avoid precedence mistakes
- Control exactly what your program computes

In low-level work (like bitwise operations or embedded systems), operator mastery is essential.

Try It Yourself

1. Write a small program that takes two integers and prints their:
   - Sum
   - Difference
   - Product
   - Quotient
   - Remainder
2. Modify it to print results as floating-point values.
3. Use logical operators to test if both numbers are positive.
4. Try combining bitwise operations:

```
printf("%d\n", 5 & 3);  // AND
printf("%d\n", 5 | 3);  // OR
printf("%d\n", 5 ^ 3);  // XOR
```

5. Experiment with parentheses and operator order until you can predict every result.

Operators are where C's simplicity meets its power, a small set of symbols that give you total control over computation, logic, and even raw bits.

14. Control Flow: if, else, switch

Programs become powerful when they can *decide*, when they can choose one path or another depending on data or conditions. In C, **control flow** statements give you that power. They determine how your program moves through different parts of your code.

Tiny Code

```
#include <stdio.h>

int main(void) {
    int temperature = 30;

    if (temperature > 35) {
        printf("It's too hot!\n");
    } else if (temperature > 25) {
        printf("It's warm.\n");
    } else {
```

```
        printf("It's cool.\n");
    }

    return 0;
}
```

Output:

```
It's warm.
```

This is how you express logic in C: by checking conditions and executing only the code that matches.

The if and else Structure

The basic pattern looks like this:

```
if (condition) {
    // do something if true
} else if (another_condition) {
    // do something else
} else {
    // default action
}
```

Each `if` or `else if` checks a **condition** that must evaluate to `true` (non-zero) or `false` (zero).

Example:

```
int score = 85;
if (score >= 90)
    printf("Grade: A\n");
else if (score >= 80)
    printf("Grade: B\n");
else
    printf("Grade: C or below\n");
```

Comparison and Boolean Logic

C doesn't have a built-in `bool` type in older standards, but since C99, you can include it:

```c
#include <stdbool.h>
bool is_ready = true;
if (is_ready) printf("Let's go!\n");
```

Behind the scenes, `true` is just `1` and `false` is `0`.

Nested if Statements

You can nest decisions for more complex logic:

```c
if (x > 0) {
    if (x % 2 == 0)
        printf("Positive even number\n");
    else
        printf("Positive odd number\n");
}
```

Just be careful, too much nesting makes code harder to read. When logic gets complex, consider reorganizing or using a `switch` statement.

The switch Statement

`switch` is a clean way to test one variable against several fixed values.

```c
#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("Another day\n");
    }

    return 0;
}
```

Output:

```
Wednesday
```

Each `case` label marks a potential branch. `break` stops the switch from "falling through" into the next case.

You can group multiple cases:

```
switch (ch) {
    case 'a':
    case 'A':
        printf("Letter A detected\n");
        break;
}
```

The Ternary Operator

For quick decisions, you can use the **conditional operator**:

```
int age = 20;
printf("%s\n", (age >= 18) ? "Adult" : "Minor");
```

This is equivalent to:

```
if (age >= 18)
    printf("Adult\n");
else
    printf("Minor\n");
```

Why It Matters

Control flow gives your programs **intelligence**. Instead of running straight through, your code reacts to input, conditions, and data. C's branching statements are simple but flexible, they're the building blocks of everything from sorting algorithms to operating system schedulers.

When you understand how to control execution, you can shape your program's logic precisely.

Try It Yourself

1. Write a program that:
    - Reads an integer from the user.
    - Prints whether it's positive, negative, or zero.
2. Extend it:
    - If it's positive, print whether it's even or odd.
3. Use a `switch` statement:
    - Ask for a number 1–7.
    - Print the day of the week that matches.
4. Try replacing your `if` statements with a ternary operator where it makes sense.

Control flow is how you *think* in code, it's how you teach your program to make decisions just like you do.

15. Loops: for, while, do-while

Sometimes you need your program to repeat something, a calculation, a print statement, or a check, again and again. Instead of copying the same line of code many times, you use **loops**. Loops make your program efficient, compact, and able to handle dynamic data of any size.

The `for` Loop

A `for` loop repeats a block of code a fixed number of times.

```
for (initialization; condition; update) {
    // repeated statements
}
```

Example:

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

Explanation:

- **Initialization** runs once at the start (`int i = 0`)
- **Condition** is checked before every loop (`i < 10`)
- **Update** runs after each iteration (`i++`)
- The loop stops when the condition becomes false

The `while` Loop

The `while` loop repeats **while** a condition remains true.

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

This loop executes as long as `n` is greater than zero.

The `do-while` Loop

The `do-while` loop guarantees at least one execution, because the condition is checked *after* the body.

```c
int i = 0;
do {
    printf("Running once! i = %d\n", i);
    i++;
} while (i < 1);
```

It's useful for input validation or repeating tasks until the user chooses to stop.

Breaking and Continuing

Sometimes you want to **skip** or **stop** partway through a loop.

```c
for (int i = 1; i <= 10; i++) {
    if (i == 5) continue;   // skip this iteration
    if (i == 8) break;      // stop the loop
    printf("%d ", i);
}
```

Output:

```
1 2 3 4 6 7
```

Nested Loops

You can place one loop inside another to handle grids, tables, or multiple 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
```

Tiny Code

Here's a complete program that demonstrates all three types of loops and control flow features:

```c
#include <stdio.h>

int main(void) {
    // for loop
    printf("for loop:\n");
    for (int i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    printf("\n\n");

    // while loop
    printf("while loop:\n");
    int n = 3;
    while (n > 0) {
        printf("n = %d\n", n);
        n--;
    }
    printf("\n");

    // do-while loop
    printf("do-while loop:\n");
    int x = 0;
    do {
        printf("x = %d\n", x);
        x++;
    } while (x < 1);
    printf("\n");

    // break and continue
    printf("break and continue demo:\n");
    for (int i = 1; i <= 10; i++) {
        if (i == 5) continue; // skip 5
        if (i == 8) break;    // stop at 8
        printf("%d ", i);
    }
    printf("\n");

    // nested loop
    printf("\nnested loops:\n");
```

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

    return 0;
}
```

Compile and run:

```
gcc loops_demo.c -o loops_demo
./loops_demo
```

You'll see all types of loops in action.

Why It Matters

Loops are the **engine of repetition** in every C program. They make it possible to:

- Process arrays, files, and lists of data
- Run algorithms that iterate until a condition is met
- Automate repetitive tasks efficiently

In C, loops are close to how the CPU itself operates, each iteration is a direct cycle of logic and computation. By mastering them, you control how your program moves, stops, and repeats, the heartbeat of every algorithm.

Try It Yourself

1. Write a `for` loop that prints numbers 1 through 100.
2. Add an `if` inside it to print only even numbers.
3. Write a `while` loop that counts down from 10 to 1.
4. Create a nested loop that prints a 3×3 multiplication table.
5. Try an infinite loop safely:

```c
while (1) {
    printf("Press Ctrl+C to stop\n");
    break; // or add a condition to exit
}
```

Once you're comfortable with loops, you can build patterns, algorithms, and data processors, all by controlling how many times code repeats and under what conditions.

16. Functions and Parameters

Functions are how you break a program into smaller, reusable pieces. Each function performs one specific task, you call it when needed, pass in data (parameters), and get something back (a return value). Functions make your code organized, testable, and easier to understand.

The Structure of a Function

A function in C has four main parts:

```c
return_type function_name(parameter_list) {
    // body of the function
    return value;
}
```

Example:

```c
int add(int a, int b) {
    return a + b;
}
```

Here:

- `int` is the return type
- `add` is the name
- `(int a, int b)` are parameters
- `return a + b;` sends a result back to the caller

Declaring and Defining Functions

In C, you must *declare* a function before using it. The declaration tells the compiler what to expect.

```c
int add(int a, int b); // declaration (prototype)

int main(void) {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}

int add(int a, int b) { // definition
    return a + b;
}
```

Output:

```
Result: 7
```

Passing Parameters

When you call a function, the arguments are passed *by value*, a copy of each value is made. Changing parameters inside the function does **not** affect the original variables.

```c
void change(int x) {
    x = 10;
}

int main(void) {
    int a = 5;
    change(a);
    printf("%d\n", a); // still 5
    return 0;
}
```

If you want to modify the original variable, use **pointers** (you'll explore this in Chapter 3):

```c
void change(int *x) {
    *x = 10;
}
```

Return Values

A function can return a value using `return`. The type of the returned value must match the function's declared return type.

```c
double average(double a, double b) {
    return (a + b) / 2.0;
}
```

To return nothing, use `void`:

```c
void greet(void) {
    printf("Hello!\n");
}
```

Tiny Code

Here's a complete program that combines multiple functions and parameters:

```c
#include <stdio.h>

// function declarations
int add(int a, int b);
int subtract(int a, int b);
double divide(double a, double b);
void greet(const char *name);

// main function
int main(void) {
    greet("C Learner");

    int sum = add(10, 5);
    int diff = subtract(10, 5);
    double quotient = divide(10.0, 5.0);

    printf("Sum: %d\n", sum);
```

```c
    printf("Difference: %d\n", diff);
    printf("Quotient: %.2f\n", quotient);

    return 0;
}

// function definitions
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

double divide(double a, double b) {
    if (b == 0) {
        printf("Error: division by zero!\n");
        return 0.0;
    }
    return a / b;
}

void greet(const char *name) {
    printf("Hello, %s!\n", name);
}
```

Compile and run:

```
gcc functions_demo.c -o functions_demo
./functions_demo
```

Output:

```
Hello, C Learner!
Sum: 15
Difference: 5
Quotient: 2.00
```

Why It Matters

Functions are the **building blocks** of every program. They let you:

- Break large problems into smaller steps
- Reuse code instead of rewriting it
- Test each part independently
- Make programs easier to read and maintain

In C, you'll use functions for everything, from arithmetic helpers to memory allocators, system calls, and modular libraries.

Try It Yourself

1. Write a function that returns the larger of two numbers.
2. Create a `void` function that prints a welcome message.
3. Add a `multiply()` function and call it from `main()`.
4. Modify the program to read numbers from user input and pass them as parameters.
5. Experiment by removing the declaration at the top, see what compiler error appears, then fix it.

Functions are how C programs *grow*. Each one is a small tool, and together, they become complete systems.

17. Scope and Lifetime of Variables

Every variable in C exists within a specific *scope* (where it can be accessed) and has a *lifetime* (how long it exists in memory). Understanding both is essential to avoid common bugs, from name conflicts to mysterious "garbage values." Once you know where and how variables live, you'll start thinking like the compiler.

Variable Scope

Scope defines *where* a variable can be seen or used in your code.

1. **Block scope (local variables)** Declared inside a function or block `{ ... }`. Accessible only within that block.

```c
void example(void) {
    int x = 10;  // local to this function
    printf("%d\n", x);
}
```

You can't access `x` outside of `example()`.

2. **File scope (global variables)** Declared outside of all functions. Accessible anywhere in the file after declaration.

```
int counter = 0; // global variable

void increment(void) {
    counter++;
}
```

3. **Function parameter scope** Parameters behave like local variables, visible only within the function.

```
void greet(const char *name) {
    printf("Hello, %s!\n", name);
}
```

4. **Block shadowing** Inner variables can temporarily "hide" outer ones:

```
int x = 5;
{
    int x = 10;  // shadows the outer x
    printf("%d\n", x); // prints 10
}
printf("%d\n", x); // prints 5
```

Variable Lifetime

Lifetime determines *how long* a variable stays in memory.

1. **Automatic (default)** Local variables are created when a function starts and destroyed when it ends.

```
void demo(void) {
    int temp = 42;  // exists only while demo() runs
}
```

2. **Static** Declared with the `static` keyword, they keep their value between function calls.

```
void counter(void) {
    static int count = 0;  // initialized only once
    count++;
    printf("Count: %d\n", count);
}
```

Every call to `counter()` increases the same variable.

3. **Dynamic** Created manually using `malloc()` or `calloc()`, they live until you `free()` them. (You'll learn this in Chapter 3.)
4. **Global** Exist for the entire duration of the program.

Storage Classes in C

Keyword	Storage	Scope	Lifetime	Notes
`auto`	Stack	Block	Function call	Default for locals
`register`	CPU Register	Block	Function call	Hint to optimize speed
`static`	Static Memory	Block/File	Program	Retains value
`extern`	Static Memory	Global	Program	Declared elsewhere

Tiny Code

Here's a full program showing scope and lifetime in action:

```c
#include <stdio.h>

int global_var = 10; // global scope

void demo_scope(void) {
    int local_var = 5; // block scope
    static int persistent = 0; // retains value between calls

    printf("Global: %d, Local: %d, Static: %d\n", global_var, local_var, persistent);
    persistent++;
}

int main(void) {
    printf("First call:\n");
    demo_scope();

    printf("\nSecond call:\n");
    demo_scope();

    printf("\nAccessing global variable in main: %d\n", global_var);
    return 0;
}
```

Compile and run:

```
gcc scope_demo.c -o scope_demo
./scope_demo
```

Output:

```
First call:
Global: 10, Local: 5, Static: 0

Second call:
Global: 10, Local: 5, Static: 1

Accessing global variable in main: 10
```

Why It Matters

Scope and lifetime are the **invisible structure** beneath your code. They define what data is available where, and for how long. Without understanding them, you'll face bugs like:

- Uninitialized values after a function returns
- Variables mysteriously resetting
- Conflicts between local and global names

Once you know how the compiler manages variables, you can reason about memory, performance, and correctness with confidence.

Try It Yourself

1. Write a function with a static counter and call it three times. Observe how the count persists.
2. Add a global variable, modify it from two different functions, and print the result.
3. Create nested blocks with variables of the same name, see how shadowing behaves.
4. Move a variable outside a function and mark it `static`. Try accessing it from another function, what happens?
5. Rewrite your earlier "calculator" example using global and local variables to see the difference.

When you understand scope and lifetime, you gain control over how your program's data moves, lives, and dies, a skill every true C programmer needs.

18. Return Values and Function Signatures

Functions not only perform tasks but often *communicate results* back to the caller. They do this through **return values**. Every C function has a **signature**, a declaration that defines its return type, name, and parameters. Getting comfortable with signatures and return values helps you write clean, predictable, and modular programs.

Function Signatures Explained

A function signature looks like this:

```
return_type function_name(parameter_list);
```

It tells the compiler:

1. **What kind of value** the function returns (`int`, `double`, `void`, etc.)
2. **What the function is called**
3. **What arguments** it expects and their types

Example:

```
int max(int a, int b);
```

This says: "`max` is a function that takes two integers and returns an integer."

Returning Values

You use the `return` keyword to send a value back from a function.

```
int add(int x, int y) {
    return x + y;
}
```

The type of the value you return must match the function's declared return type.

If a function doesn't need to return anything, declare it as `void`:

```
void greet(void) {
    printf("Hello!\n");
}
```

A `void` function can still perform actions, it just doesn't produce a result.

Multiple Return Points

You can return early if certain conditions are met.

```
int divide(int a, int b) {
    if (b == 0) {
        printf("Error: division by zero!\n");
        return 0;
    }
    return a / b;
}
```

This is common for error handling or input validation.

Returning Different Data Types

You can return any type, not just integers.

```
double average(double a, double b) {
    return (a + b) / 2.0;
}

char first_letter(const char *word) {
    return word[0];
}

_Bool is_even(int n) {
    return n % 2 == 0;
}
```

For more complex data, you'll later learn how to return pointers or structs.

Tiny Code

Here's a complete example showing several return types and signatures:

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

// function declarations
```

```c
int add(int x, int y);
double divide(double a, double b);
bool is_even(int n);
void greet(const char *name);

int main(void) {
    greet("C Programmer");

    int sum = add(7, 3);
    double quotient = divide(10.0, 4.0);
    bool check = is_even(sum);

    printf("Sum: %d\n", sum);
    printf("Quotient: %.2f\n", quotient);
    printf("Is sum even? %s\n", check ? "Yes" : "No");

    return 0;
}

// function definitions
int add(int x, int y) {
    return x + y;
}

double divide(double a, double b) {
    if (b == 0.0) {
        printf("Cannot divide by zero.\n");
        return 0.0;
    }
    return a / b;
}

bool is_even(int n) {
    return n % 2 == 0;
}

void greet(const char *name) {
    printf("Hello, %s!\n", name);
}
```

Compile and run:

```
gcc return_demo.c -o return_demo
./return_demo
```

Output:

```
Hello, C Programmer!
Sum: 10
Quotient: 2.50
Is sum even? Yes
```

Why It Matters

Return values are how functions *communicate*. By designing clear and meaningful signatures:

- You make your code **predictable**, every function has a defined purpose and output.
- The compiler can **check correctness**, mismatched types raise warnings.
- You can **compose** functions, one function's return becomes another's input.

In large systems, consistent signatures and meaningful return types form the backbone of good API design.

Try It Yourself

1. Write a function `max(a, b)` that returns the larger of two integers.
2. Write a function `to_upper(char c)` that returns an uppercase version of a character.
3. Modify a `divide()` function to return `-1` if division by zero occurs.
4. Create a `sum_to_n(int n)` that returns the sum of all numbers from 1 to `n`.
5. Try using a `void` function that prints the result of another function call.

Return values give your functions purpose, they turn simple actions into reusable building blocks that make your programs expressive, modular, and alive.

19. Static vs Dynamic Linking of Code Units

When your program grows beyond a single file, you begin linking multiple *code units* together, functions and data that live in different files. This linking step decides how your program combines and shares code. There are two main ways to do it in C: **static linking** and **dynamic linking**. Understanding both is essential for building real-world software.

The Big Picture

When you compile a C program:

1. Each `.c` file becomes an **object file** (`.o`).
2. The **linker** combines all object files and libraries into one executable.
3. Depending on how you link, that executable may contain:
    - All required code inside (static linking), or
    - References to shared libraries on the system (dynamic linking).

Static Linking

Static linking copies all the necessary library code directly into your program at build time.

Example command:

```
gcc main.c mathutils.c -o app_static
```

Everything from `mathutils.c` gets *embedded* inside `app_static`.

Pros:

- No external dependencies at runtime
- Faster startup (everything is self-contained)
- Easier deployment

Cons:

- Larger executable file
- Updating a library means recompiling the program

Dynamic Linking

Dynamic linking (or *shared linking*) links your program to shared libraries (`.so` on Linux, `.dll` on Windows) at runtime instead of embedding them.

Example:

```
gcc main.c -o app_dynamic -lm
```

Here, `-lm` tells the linker to use the shared **math library** (`libm.so`).

Your program keeps the library *separate*, loading it when executed.

Pros:

- Smaller executables
- Libraries can be updated independently
- Multiple programs share the same library in memory

Cons:

- Requires the correct library to be available at runtime
- Slightly slower startup

Tiny Code

Let's demonstrate static vs dynamic linking using a simple math utility.

mathutils.c

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}
```

mathutils.h

```c
#ifndef MATHUTILS_H
#define MATHUTILS_H

int add(int a, int b);
int multiply(int a, int b);

#endif
```

main.c

```c
#include <stdio.h>
#include "mathutils.h"

int main(void) {
    int x = 4, y = 5;
    printf("Add: %d\n", add(x, y));
    printf("Multiply: %d\n", multiply(x, y));
    return 0;
}
```

Static Linking Build:

```
gcc main.c mathutils.c -o static_app
./static_app
```

Output:

```
Add: 9
Multiply: 20
```

Dynamic Linking Build (using shared library):

```
gcc -c -fPIC mathutils.c -o mathutils.o
gcc -shared -o libmathutils.so mathutils.o
gcc main.c -L. -lmathutils -o dynamic_app
export LD_LIBRARY_PATH=.
./dynamic_app
```

Output:

```
Add: 9
Multiply: 20
```

Now your executable depends on the shared `libmathutils.so`, the same library could be used by many other programs.

Why It Matters

Linking determines **how your software connects and shares code**. It affects:

- Performance and memory usage
- Deployment and portability
- How easily your program updates when libraries change

Static linking is great for small, standalone tools. Dynamic linking is better for large systems, shared components, or when you rely on system libraries (like `libc`, `libm`, `pthread`).

Understanding linking makes you a systems thinker, you'll know how the pieces of your program fit together at the binary level.

Try It Yourself

1. Create `mathutils.c` and `mathutils.h` as above.
2. Compile statically and dynamically; compare file sizes using `ls -lh`.
3. Move `libmathutils.so` out of the directory and run `./dynamic_app`, notice the missing library error.
4. Add `export LD_LIBRARY_PATH=.` and run again.
5. Modify `multiply()` to print a message, recompile *only* the shared library and see the change take effect instantly.

Static vs dynamic linking is where your C programs move from "source code" to real-world software, how your logic becomes part of an executable that lives, loads, and runs on any machine.

20. Practice: Build a Simple Calculator

Now that you've learned about functions, variables, loops, control flow, and linking, it's time to bring everything together. You'll build a simple calculator that performs basic arithmetic using clean modular code. This small project will reinforce everything from Chapters 11–19, data types, operators, control flow, and reusable functions.

Project Overview

You'll write a calculator that:

- Prompts the user for two numbers and an operator
- Performs the corresponding operation (+, -, *, /)
- Handles division by zero safely

- Repeats until the user chooses to quit

You'll structure it using multiple functions and a clean main loop.

Tiny Code

calculator.c

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

// Function declarations
double add(double a, double b);
double subtract(double a, double b);
double multiply(double a, double b);
double divide(double a, double b);
void print_menu(void);

int main(void) {
    double num1, num2, result;
    char op;
    bool running = true;

    printf("=== Simple C Calculator ===\n");

    while (running) {
        print_menu();
        printf("Enter an operator (+, -, *, /) or q to quit: ");
        scanf(" %c", &op);

        if (op == 'q' || op == 'Q') {
            running = false;
            printf("Goodbye!\n");
            break;
        }

        printf("Enter first number: ");
        scanf("%lf", &num1);
        printf("Enter second number: ");
        scanf("%lf", &num2);

        switch (op) {
```

```
            case '+':
                result = add(num1, num2);
                printf("Result: %.2f\n", result);
                break;
            case '-':
                result = subtract(num1, num2);
                printf("Result: %.2f\n", result);
                break;
            case '*':
                result = multiply(num1, num2);
                printf("Result: %.2f\n", result);
                break;
            case '/':
                if (num2 == 0) {
                    printf("Error: Division by zero!\n");
                } else {
                    result = divide(num1, num2);
                    printf("Result: %.2f\n", result);
                }
                break;
            default:
                printf("Unknown operator: %c\n", op);
        }

        printf("\n");
    }

    return 0;
}

// Function definitions
double add(double a, double b) {
    return a + b;
}

double subtract(double a, double b) {
    return a - b;
}

double multiply(double a, double b) {
    return a * b;
}
```

```c
double divide(double a, double b) {
    return a / b;
}

void print_menu(void) {
    printf("\nChoose an operation:\n");
    printf("  +  Addition\n");
    printf("  -  Subtraction\n");
    printf("  *  Multiplication\n");
    printf("  /  Division\n");
    printf("  q  Quit\n\n");
}
```

Compile and run:

```c
gcc calculator.c -o calculator
./calculator
```

Example session:

```
=== Simple C Calculator ===
Choose an operation:
  +  Addition
  -  Subtraction
  *  Multiplication
  /  Division
  q  Quit

Enter an operator (+, -, *, /) or q to quit: +
Enter first number: 5
Enter second number: 3
Result: 8.00

Enter an operator (+, -, *, /) or q to quit: /
Enter first number: 10
Enter second number: 0
Error: Division by zero!

Enter an operator (+, -, *, /) or q to quit: q
Goodbye!
```

Why It Matters

This simple project combines nearly everything you've learned in Chapter 2:

- **Data types** for representing numbers
- **Operators** for performing calculations
- **Control flow** for making decisions
- **Loops** for repeated interaction
- **Functions** for modular design

You've now moved beyond syntax, you've built a working, reusable C program that interacts with real users.

Try It Yourself

1. Add a new operator `%` for modulo (integer remainder).
2. Create separate files:
   - `calculator.c` for `main()`
   - `mathutils.c` and `mathutils.h` for the arithmetic functions Then compile using:

```
gcc calculator.c mathutils.c -o calculator
```

3. Extend the calculator to remember the last result and reuse it if the user enters a single operand.
4. Add input validation (e.g., check if `scanf` actually reads a number).
5. For a challenge, implement power (`^`) or square root (`sqrt`) using `<math.h>`.

This calculator marks the end of your **Language Basics** journey, from variables and control flow to full, interactive programs. In the next chapter, you'll dive into memory: how C stores your data, manages it, and lets you control it directly.

Chapter 3. Working with Memory

21. Understanding Memory Layout (Stack, Heap, Data, Code)

Before you can master pointers or dynamic memory, you need to understand **how memory is organized** in a running C program. C gives you a level of control that few languages allow, but to use it safely, you must know where your data lives and how long it stays there.

The Memory Segments

When a program runs, its memory is divided into several key sections:

Segment	Purpose	Example Data
Code (Text)	Stores compiled machine instructions	Functions, program logic
Data (Static)	Stores global and static variables with initialized values	`int count = 5;`
BSS (Uninitialized Data)	Holds global/static variables that start as zero or are uninitialized	`int buffer[256];`
Heap	Used for dynamic memory allocation (`malloc`, `calloc`)	Large, runtime-created data
Stack	Stores local variables, function parameters, return addresses	Function calls, recursion

These segments are managed differently by the operating system, and each has a different **lifetime** and **scope**.

Tiny Code

Here's a small program that prints the memory addresses of different variables to show where they live:

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

// Global variable (Data segment)
int global_var = 42;

// Uninitialized global variable (BSS segment)
int global_bss;

void show_addresses(void) {
    // Local variable (Stack)
    int local_var = 10;

    // Static variable (Data segment)
    static int static_var = 20;

    // Dynamic variable (Heap)
    int *heap_var = malloc(sizeof(int));
    *heap_var = 30;

    printf("Code (function) address:     %p\n", (void *)show_addresses);
    printf("Global variable address:     %p\n", (void *)&global_var);
    printf("Uninitialized global address:%p\n", (void *)&global_bss);
    printf("Static variable address:     %p\n", (void *)&static_var);
    printf("Stack variable address:      %p\n", (void *)&local_var);
    printf("Heap variable address:       %p\n", (void *)heap_var);

    free(heap_var);
}

int main(void) {
    show_addresses();
    return 0;
}
```

Compile and run:

```
gcc memory_layout.c -o memory_layout
./memory_layout
```

Example output (addresses vary by system):

```
Code (function) address:     0x561ce7348169
Global variable address:     0x561ce7546014
Uninitialized global address:0x561ce7546018
Static variable address:     0x561ce7546020
Stack variable address:      0x7ffc94b65a5c
Heap variable address:       0x561ce774b2a0
```

Notice how the stack address is much higher than the heap, the stack usually grows *downward*, and the heap grows *upward* in memory.

How It All Works

1. **Code segment**
    - Contains compiled instructions.
    - Usually marked as read-only to prevent accidental modification.
2. **Data segment**
    - Holds global and static variables initialized with values.
    - Exists for the entire program duration.
3. **BSS (Block Started by Symbol)**
    - Holds uninitialized global/static variables.
    - Automatically zero-initialized at runtime.
4. **Stack**
    - Used for local variables and function calls.
    - Automatically managed, grows and shrinks as functions are called and return.
5. **Heap**
    - Allocated manually at runtime.
    - Requires explicit management (`malloc` and `free`).

Why It Matters

Every time you write a variable, you're deciding, whether consciously or not, *where* in memory it lives. Understanding this layout helps you:

- Debug memory errors (segmentation faults, leaks, corruption).
- Reason about performance and function calls.
- Write correct code when using `malloc`, `free`, and pointers.
- Build real systems software like allocators or kernels.

Without this mental model, C memory bugs feel mysterious; with it, they become logical and fixable.

Try It Yourself

1. Add more global, local, and static variables to the example and print their addresses.
2. Allocate two blocks with `malloc()` and compare their addresses, the heap grows upward.
3. Call `show_addresses()` multiple times and notice how the stack variable's address changes each call.
4. Move a variable from global to local and observe how its memory segment changes.
5. Draw a diagram showing stack, heap, data, and code regions for your system.

Understanding memory layout is your first real step into **systems-level C**, it's how you begin to see your code not just as text, but as **structured bytes living inside memory**.

22. Pointers and Addresses

Pointers are at the heart of C programming. They give you direct access to memory, the power to read, write, and manipulate data stored anywhere in your program. Understanding pointers transforms how you think about variables, functions, and data structures.

You can't truly master C without mastering pointers.

What Is a Pointer?

A **pointer** is a variable that stores the *address* of another variable. Think of it as a reference to a specific spot in memory.

```
int value = 42;
int *ptr = &value; // pointer to int, stores the address of value
```

- `&value` gives the memory address of `value`.

- `ptr` *holds* that address.
- `*ptr` lets you *access* the value stored there (this is called *dereferencing*).

Tiny Code

```c
#include <stdio.h>

int main(void) {
    int number = 10;
    int *p = &number;  // pointer stores address of number

    printf("Value of number: %d\n", number);
    printf("Address of number: %p\n", (void *)&number);
    printf("Pointer p holds address: %p\n", (void *)p);
    printf("Value through pointer: %d\n", *p);

    *p = 20;  // modify the value via the pointer
    printf("New value of number: %d\n", number);

    return 0;
}
```

Compile and run:

```
gcc pointer_basics.c -o pointer_basics
./pointer_basics
```

Output (addresses vary):

```
Value of number: 10
Address of number: 0x7ffc8f4c9c4c
Pointer p holds address: 0x7ffc8f4c9c4c
Value through pointer: 10
New value of number: 20
```

Pointer Declaration and Dereferencing

Syntax	Meaning
`int *p;`	Pointer to an integer
`char *c;`	Pointer to a character
`float *f;`	Pointer to a float
`p = &x;`	Assigns address of variable `x` to pointer `p`
`*p`	Accesses (dereferences) the value stored at the address held by `p`

Dereferencing works both ways:

- Reading the value: `x = *p;`
- Writing to the address: `*p = 99;`

Null Pointers

A pointer that points to nothing should be set to `NULL`.

```
int *ptr = NULL;
if (ptr == NULL) {
    printf("Pointer is not initialized.\n");
}
```

Dereferencing a null pointer (`*ptr` when `ptr == NULL`) causes a **segmentation fault**, one of the most common C errors.

Pointer to Pointer

You can have pointers that store addresses of other pointers.

```
int x = 5;
int *p = &x;
int **pp = &p;

printf("x = %d\n", **pp);
```

This concept appears in multi-dimensional arrays and function pointers.

Why It Matters

Pointers are what make C powerful:

- They enable **dynamic memory allocation** (`malloc`, `calloc`, `free`).
- They allow **arrays, strings, and structures** to be passed efficiently.
- They are the foundation for **linked lists, trees, and system calls**.

But they also demand precision. A single misused pointer can cause crashes or memory corruption.

Mastering pointers means mastering both control and responsibility over memory.

Try It Yourself

1. Write a program that declares an integer and prints both its value and address.
2. Create a pointer to that integer and modify the variable's value through the pointer.
3. Try declaring `int *p = NULL;` and check it before dereferencing.
4. Print a pointer to a pointer (`int **`) and see how the addresses relate.
5. For fun, declare two variables and make one pointer swap their values using dereferencing.

Once you truly understand pointers, the rest of C, arrays, structs, dynamic memory, even function calls, begins to make sense. They are the bridge between your code and the machine's actual memory.

23. Arrays and Pointer Arithmetic

An **array** is a block of consecutive memory cells that hold elements of the same type. Arrays and pointers are deeply connected in C, in fact, an array's name often behaves like a pointer to its first element. Understanding how arrays and pointer arithmetic work together is key to writing fast, memory-efficient programs.

Declaring and Using Arrays

```c
#include <stdio.h>

int main(void) {
    int numbers[5] = {10, 20, 30, 40, 50};

    for (int i = 0; i < 5; i++) {
        printf("numbers[%d] = %d\n", i, numbers[i]);
```

```
    }

    return 0;
}
```

Output:

```
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
numbers[4] = 50
```

Here:

- `numbers` is an array of five integers.
- Each element is stored **next to each other in memory**.
- The compiler knows each `int` takes the same number of bytes, so it can find `numbers[i]` quickly using pointer arithmetic.

Array Name as a Pointer

When you use an array's name (without an index), it acts as a pointer to its first element.

```
int numbers[3] = {1, 2, 3};
int *p = numbers;  // same as &numbers[0]

printf("%d %d %d\n", *p, *(p + 1), *(p + 2));
```

Output:

```
1 2 3
```

Each time you add 1 to the pointer, it moves forward by one element, **not one byte**, but one *object* of that type.

Tiny Code

Here's a complete example showing array access and pointer arithmetic:

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {2, 4, 6, 8, 10};
    int *ptr = arr; // arr decays to pointer to arr[0]

    printf("Accessing with array index:\n");
    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    printf("\nAccessing with pointer arithmetic:\n");
    for (int i = 0; i < 5; i++) {
        printf("*(ptr + %d) = %d\n", i, *(ptr + i));
    }

    printf("\nAddresses in memory:\n");
    for (int i = 0; i < 5; i++) {
        printf("&arr[%d] = %p\n", i, (void *)&arr[i]);
    }

    return 0;
}
```

Compile and run:

```
gcc array_pointer.c -o array_pointer
./array_pointer
```

Output (addresses will differ):

```
Accessing with array index:
arr[0] = 2
arr[1] = 4
arr[2] = 6
arr[3] = 8
arr[4] = 10

Accessing with pointer arithmetic:
*(ptr + 0) = 2
*(ptr + 1) = 4
```

```
*(ptr + 2) = 6
*(ptr + 3) = 8
*(ptr + 4) = 10

Addresses in memory:
&arr[0] = 0x7ffcc73f9a60
&arr[1] = 0x7ffcc73f9a64
&arr[2] = 0x7ffcc73f9a68
&arr[3] = 0x7ffcc73f9a6c
&arr[4] = 0x7ffcc73f9a70
```

You can see that each element sits **4 bytes apart** (typical size of `int`).

Pointer Arithmetic Rules

When you move pointers, C automatically scales by the size of the type they point to:

Expression	Meaning
`p + 1`	Move to the next element
`p - 1`	Move to the previous element
`*(p + i)`	Access the i-th element after the current one
`p2 - p1`	Returns the number of elements between two pointers

Example:

```c
int *start = arr;
int *end = arr + 5;
printf("Array length: %ld\n", end - start); // prints 5
```

Common Pitfalls

1. **Out-of-bounds access** Accessing memory beyond an array's valid range leads to *undefined behavior*:

   ```c
   arr[5] = 99; // invalid! array has only indices 0-4
   ```

2. **Array decay** Arrays "decay" to pointers when passed to functions, they lose size information. You must pass the length manually.

```c
void print_array(int *arr, int len);
```

3. **Pointer confusion** Remember that `arr[i]` and `*(arr + i)` mean the same thing. Mixing them is fine, but be consistent for readability.

Why It Matters

Arrays and pointers form the **foundation of C data structures**. You'll use them to build:

- Strings (arrays of `char`)
- Matrices (arrays of arrays)
- Linked lists and trees (via pointer arithmetic)

Once you're comfortable thinking of arrays as *contiguous memory blocks* accessed through pointers, you can start designing your own data structures like a real systems programmer.

Try It Yourself

1. Write a function that prints an array using only pointers (no `[]` syntax).
2. Create an array of `char` and print it as a string and as separate characters.
3. Declare an array of 10 numbers, then use pointers to sum them.
4. Print the address difference between two elements.
5. Create a two-dimensional array and print it with nested loops.

Arrays and pointers are two sides of the same coin in C. Once you understand their connection, you'll see how powerful, and elegant, direct memory access can be.

24. Strings as Character Arrays

In C, a **string** is simply an array of characters ending with a special null character `'\0'`. Unlike higher-level languages, C doesn't have a built-in string type, just arrays and pointers. This simplicity gives you full control over text data but also demands care: every string operation must respect memory limits and null terminators.

How Strings Work

A string like `"Hello"` in C is represented internally as:

Character	H	e	l	l	o	\0
Index	0	1	2	3	4	5

The `'\0'` (ASCII 0) marks the end of the string, it's how functions like `printf` or `strlen` know where to stop.

Declaring Strings

There are two common ways to declare strings:

```
char greeting1[] = "Hello";         // automatic null terminator
char greeting2[6] = {'H','e','l','l','o','\0'}; // explicit
char *greeting3 = "Hello";          // pointer to string literal
```

`greeting1` is a mutable array you can modify. `greeting3` points to a **read-only** string literal stored in memory, modifying it causes undefined behavior.

Tiny Code

Here's a complete example that explores string declarations, iteration, and basic operations:

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

int main(void) {
    char msg[] = "C language";
    char *ptr = msg; // pointer to the first character

    printf("String: %s\n", msg);
    printf("Length: %zu\n", strlen(msg));

    printf("\nCharacters one by one:\n");
    for (int i = 0; msg[i] != '\0'; i++) {
        printf("msg[%d] = %c (address: %p)\n", i, msg[i], (void *)&msg[i]);
    }
```

```c
    printf("\nAccess via pointer arithmetic:\n");
    for (int i = 0; *(ptr + i) != '\0'; i++) {
        printf("*(ptr + %d) = %c\n", i, *(ptr + i));
    }

    // Modify string safely
    msg[0] = 'C';
    msg[1] = '+';
    printf("\nModified string: %s\n", msg);

    return 0;
}
```

Compile and run:

```
gcc strings_demo.c -o strings_demo
./strings_demo
```

Output:

```
String: C language
Length: 10

Characters one by one:
msg[0] = C (address: 0x7ffd29c4a0a0)
msg[1] =   (address: 0x7ffd29c4a0a1)
msg[2] = l (address: 0x7ffd29c4a0a2)
...

Access via pointer arithmetic:
*(ptr + 0) = C
*(ptr + 1) =
*(ptr + 2) = l
...

Modified string: C+anguage
```

Common String Operations

C provides several standard functions in `<string.h>`:

Function	Description	Example
`strlen(s)`	Get string length (excluding \0)	`strlen("Hi") == 2`
`strcpy(dest, src)`	Copy string	`strcpy(name, "Bob");`
`strcat(dest, src)`	Concatenate strings	`strcat(full, last);`
`strcmp(a, b)`	Compare strings (0 if equal)	`strcmp("a","b")`
`strchr(s, c)`	Find first occurrence of character	`strchr(word, 'a')`
`strstr(s, sub)`	Find substring	`strstr(text, "find")`

Example:

```
char a[20] = "Hello, ";
char b[] = "World!";
strcat(a, b);
printf("%s\n", a); // "Hello, World!"
```

Pointers and Strings

Because a string's name decays into a pointer, you can pass strings directly to functions:

```
void greet(const char *name) {
    printf("Hello, %s!\n", name);
}

int main(void) {
    greet("C Learner");
    return 0;
}
```

`const char *` prevents accidental modification of the string literal.

Common Pitfalls

1. **Forgetting `'\0'`:**

   ```
   char word[4] = {'T','e','s','t'}; // missing terminator, unsafe
   ```

2. **Buffer overflows:** Copying more characters than fit in the destination buffer leads to undefined behavior.

   ```
   char dest[5];
   strcpy(dest, "Too long!"); // dangerous
   ```

3. **Modifying string literals:**

```
char *s = "Hello";
s[0] = 'Y'; // crash or undefined behavior
```

Why It Matters

Strings are the foundation of **text processing, file handling, and user interfaces** in C. Because they're just arrays of characters, understanding strings forces you to think about:

- **Memory layout**
- **Null termination**
- **Buffer size and safety**

Once you internalize how C handles text at the byte level, you'll be ready to build real parsers, file readers, and command-line tools.

Try It Yourself

1. Write a function `void reverse(char *s)` that reverses a string in place.
2. Implement your own version of `strlen`.
3. Create a program that counts vowels in a string.
4. Concatenate two user-input strings using `strcat`.
5. Experiment by printing a string without `'\0'`, observe what happens.

Strings in C are both elegant and dangerous, a true test of precision. Once you master them, you'll understand how text truly exists in memory, one byte at a time.

25. Dynamic Memory Allocation (malloc, calloc, realloc, free)

Static arrays have fixed size, but real programs often need flexible data that grows or shrinks at runtime. Dynamic memory allocation lets you **request**, **use**, and **release** memory manually while your program is running. It's one of the most powerful and error-prone parts of C.

The Idea

C provides four key functions from `<stdlib.h>` for dynamic memory management:

Function	Purpose
`malloc(size)`	Allocates a block of memory
`calloc(n, size)`	Allocates and clears memory for `n` elements

Function	Purpose
`realloc(ptr, size)`	Changes the size of a previously allocated block
`free(ptr)`	Releases memory back to the system

These return a **pointer** to the allocated memory, or `NULL` if allocation fails.

Basic Example

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

int main(void) {
    int *p = malloc(sizeof(int));  // allocate space for one int
    if (p == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    *p = 42;  // store a value in allocated memory
    printf("Value: %d\n", *p);

    free(p);  // release the memory
    return 0;
}
```

Compile and run:

```
gcc malloc_demo.c -o malloc_demo
./malloc_demo
```

Output:

```
Value: 42
```

Allocating Arrays Dynamically

You can allocate arrays at runtime using `malloc()` or `calloc()`.

```c
int n;
printf("Enter number of elements: ");
scanf("%d", &n);

int *arr = malloc(n * sizeof(int));
if (arr == NULL) {
    printf("Out of memory.\n");
    return 1;
}

// Initialize and print
for (int i = 0; i < n; i++) {
    arr[i] = i * 10;
    printf("%d ", arr[i]);
}

free(arr);
```

Output example:

```
Enter number of elements: 5
0 10 20 30 40
```

`malloc()` leaves memory uninitialized, while `calloc()` clears it to zero:

```c
int *arr = calloc(n, sizeof(int)); // all elements start at 0
```

Changing Memory Size with realloc()

When you need to resize an allocated block, say, double an array's capacity, use `realloc()`.

```c
int *arr = malloc(3 * sizeof(int));
arr[0] = 1; arr[1] = 2; arr[2] = 3;

// grow array to 5 elements
int *temp = realloc(arr, 5 * sizeof(int));
if (temp == NULL) {
    printf("Reallocation failed!\n");
    free(arr);
    return 1;
```

```
}
arr = temp;

arr[3] = 4;
arr[4] = 5;

for (int i = 0; i < 5; i++)
    printf("%d ", arr[i]);

free(arr);
```

Output:

```
1 2 3 4 5
```

`realloc()` tries to expand the existing block if possible; if not, it allocates a new block, copies the data, and frees the old one automatically.

Tiny Code

Here's a complete program combining `malloc`, `calloc`, `realloc`, and `free`:

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

int main(void) {
    int n = 3;
    int *nums = calloc(n, sizeof(int));

    if (nums == NULL) {
        printf("Initial allocation failed.\n");
        return 1;
    }

    // Fill array
    for (int i = 0; i < n; i++) nums[i] = (i + 1) * 5;

    printf("Initial values: ");
    for (int i = 0; i < n; i++) printf("%d ", nums[i]);
    printf("\n");
```

```c
    // Resize
    n = 5;
    int *new_nums = realloc(nums, n * sizeof(int));
    if (new_nums == NULL) {
        printf("Reallocation failed.\n");
        free(nums);
        return 1;
    }
    nums = new_nums;

    // Fill new slots
    for (int i = 3; i < n; i++) nums[i] = (i + 1) * 5;

    printf("After realloc: ");
    for (int i = 0; i < n; i++) printf("%d ", nums[i]);
    printf("\n");

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

Output:

```
Initial values: 5 10 15
After realloc: 5 10 15 20 25
```

Memory Allocation Diagram

```
Stack   →   grows downward
Heap    →   grows upward
Data    →   global/static variables
Code    →   program instructions
```

Each call to `malloc` reserves space on the **heap**, which stays allocated until explicitly freed.

Why It Matters

Dynamic memory is the **backbone of all real systems programming**. Without it, you can't build:

- Variable-sized arrays
- Linked lists, trees, graphs
- Caches and databases
- File readers and parsers

It's also where most C bugs happen, dangling pointers, leaks, double frees, and buffer overruns, so disciplined management is crucial.

Try It Yourself

1. Allocate an array of 10 integers, fill it, print it, and free it.
2. Use `calloc` instead of `malloc` and observe the zero initialization.
3. Resize the array from 10 to 20 elements using `realloc`.
4. Forget to call `free()` and then run your program with **Valgrind**, see the memory leak report.
5. Write a function `int *make_array(int n)` that allocates and returns a pointer to a new array.

Dynamic allocation is where you start managing memory *by hand*. Done right, it gives you incredible control and efficiency, done wrong, it's chaos. Master it carefully: it's the essence of being a C programmer.

26. Memory Leaks and Undefined Behavior

C gives you total control over memory, which means you can do anything you want, including things that *should never be done*. Two of the biggest dangers are **memory leaks** (when memory is never released) and **undefined behavior** (when the program does something unpredictable). Learning to avoid these is the key to writing stable, safe, and correct C programs.

What Is a Memory Leak?

A **memory leak** happens when you allocate memory on the heap and never free it. The memory stays reserved even though you can't access it anymore.

Example:

```c
#include <stdlib.h>

void leak(void) {
    int *data = malloc(100 * sizeof(int)); // allocated
    data[0] = 42;
```

```
    // forgot to free(data); memory is now lost
}
```

If `leak()` runs many times, your program consumes more and more memory until it crashes or slows down. In long-running programs (like servers), this is deadly.

Rule: Every `malloc`, `calloc`, or `realloc` must eventually be paired with a matching `free()`.

Tiny Code

Let's see leaks and fixes in action.

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

void with_leak(void) {
    int *arr = malloc(5 * sizeof(int));
    for (int i = 0; i < 5; i++) arr[i] = i;
    printf("with_leak: allocated 5 ints, but not freed.\n");
}

void without_leak(void) {
    int *arr = malloc(5 * sizeof(int));
    for (int i = 0; i < 5; i++) arr[i] = i;
    printf("without_leak: freeing memory.\n");
    free(arr);
}

int main(void) {
    with_leak();
    without_leak();
    return 0;
}
```

Compile and run with **Valgrind** (a memory checker):

```
gcc leaks_demo.c -o leaks_demo
valgrind ./leaks_demo
```

Valgrind output (simplified):

```
==1234== HEAP SUMMARY:
==1234==    definitely lost: 20 bytes in 1 blocks
==1234==    indirectly lost: 0 bytes in 0 blocks
==1234== LEAK SUMMARY:
==1234==    1 blocks definitely lost
```

You can see the first function leaked memory, while the second freed it properly.

Dangling Pointers

A **dangling pointer** points to memory that has been freed or is otherwise invalid.

```c
int *p = malloc(sizeof(int));
*p = 10;
free(p);
printf("%d\n", *p); //   undefined behavior
```

After `free(p)`, the pointer `p` still holds the old address, but that memory no longer belongs to you. Accessing it may crash, or appear to work, or corrupt data, you can't rely on it.

Always nullify freed pointers:

```c
free(p);
p = NULL;
```

Double Free

Freeing the same memory twice also leads to undefined behavior:

```c
int *p = malloc(sizeof(int));
free(p);
free(p); //   double free error
```

Most modern OSes detect this and abort, but it's still a critical bug.

Use-After-Free

This is one of the worst kinds of memory errors. It happens when you access memory after it's been freed.

```c
int *arr = malloc(3 * sizeof(int));
arr[0] = 5;
free(arr);
arr[0] = 7; //  use-after-free
```

The compiler won't catch this, but Valgrind will warn you.

Uninitialized Memory

Reading memory you never wrote to is also undefined:

```c
int *arr = malloc(5 * sizeof(int));
printf("%d\n", arr[2]); //  uninitialized read
```

`malloc()` does **not** zero out memory, use `calloc()` if you need cleared data.

Common Causes of Undefined Behavior

Type	Example	Consequence
Out-of-bounds access	`arr[10]` in a 5-element array	Corrupts memory
Use-after-free	Dereferencing freed pointer	Crash or silent corruption
Null pointer dereference	`*NULL`	Crash
Division by zero	`x / 0`	Crash
Invalid pointer arithmetic	`(int *)0 + 1`	Undefined
Modifying string literal	`char *s = "hi"; s[0]='H';`	Crash

Why It Matters

Undefined behavior is not "just a bug." It means **anything** can happen:

- Your program may seem fine but fail later.
- Compiler optimizations may remove or reorder code unexpectedly.
- The same code might work on one system and crash on another.

In C, correctness is your responsibility. You must know when memory is valid, who owns it, and when to free it.

Defensive Techniques

1. **Always check `malloc()` return values.**
2. **Initialize pointers to `NULL`.**
3. **Set pointers to `NULL` after freeing.**
4. **Use Valgrind** (Linux) or **AddressSanitizer** (Clang/GCC) to detect leaks and invalid access.
5. **Prefer small, testable functions**, easier to verify memory ownership.
6. **Avoid mixing stack and heap memory** unless you're certain of lifetimes.

Try It Yourself

1. Write a small program that intentionally leaks memory. Run it under Valgrind.
2. Fix the leak by calling `free()` properly.
3. Create a dangling pointer and observe what happens (on some systems it crashes, on others not).
4. Experiment with `calloc()` to see how zero-initialized memory behaves.
5. Write a function that allocates memory and returns it, then ensure the caller frees it.

Final Thought

Memory errors are the hardest bugs to track because they may not appear right away. But once you understand **ownership**, who allocates and who frees, memory in C becomes predictable, even elegant. This discipline is what separates casual C users from real systems programmers.

27. const and volatile Qualifiers

C gives you fine-grained control over how variables are used through **type qualifiers**. Two of the most important are `const` and `volatile`. They look simple but play a crucial role in writing safe, predictable, and efficient code, especially in systems programming, embedded systems, and multithreaded environments.

The `const` Qualifier

`const` means *read-only*: once a variable is initialized, you cannot modify it.

```
const int x = 10;
x = 20; //  error: assignment of read-only variable
```

It's a promise to the compiler, and to other programmers, that the value won't change.

`const` can be applied to many things:

- Variables
- Function parameters
- Pointers
- Return types

const with Pointers

`const` with pointers can be tricky but follows consistent rules. The position of `const` determines what cannot change.

Declaration	Meaning
`const int *p;`	Pointer to **constant data**, data can't change
`int *const p;`	**Constant pointer**, pointer can't change, data can
`const int *const p;`	Both pointer and data are constant

Example:

```
int value = 42;
const int *p1 = &value;   // cannot modify *p1
int *const p2 = &value;   // cannot reassign p2
const int *const p3 = &value; // cannot change *p3 or p3
```

const in Function Parameters

Marking parameters as `const` helps prevent accidental modification and enables compiler optimizations.

```
void print_message(const char *msg) {
    printf("%s\n", msg);
}
```

Here, `msg` is read-only; the function can't modify the string it points to.

Tiny Code

Here's a program demonstrating `const` in action:

```
#include <stdio.h>

void show(const int *ptr) {
    // *ptr = 10; //   not allowed
    printf("Value: %d\n", *ptr);
}

int main(void) {
    int num = 5;
    const int *p = &num;
    int *const q = &num;

    printf("num = %d\n", num);

    // *p = 10; //   cannot modify value through const pointer
    *q = 15;   //   data modifiable through q
    printf("num after q change = %d\n", num);

    show(&num); // function accepts const pointer
    return 0;
}
```

Compile and run:

```
gcc const_demo.c -o const_demo
./const_demo
```

Output:

```
num = 5
num after q change = 15
Value: 15
```

The `volatile` Qualifier

`volatile` tells the compiler that a variable **can change at any time**, even if your code doesn't modify it. It prevents the compiler from optimizing out reads or writes.

Use `volatile` when:

- A variable can be changed by **hardware** (e.g., memory-mapped I/O registers).
- A variable can be modified by **another thread** or **signal handler**.
- You need to force an **actual memory read** each time, not a cached value.

Example:

```
volatile int sensor_value;
while (sensor_value < 100) {
    // without volatile, compiler might optimize this loop away
}
```

Here, `sensor_value` might be updated by hardware; `volatile` ensures each check re-reads memory instead of reusing a cached register value.

Combining const and volatile

Yes, you can use both together, a value that can change unexpectedly, but your code cannot modify it.

Example:

```
const volatile int status_register = 0x1234;
```

This is common in embedded systems, where a hardware register's bits may change due to external events.

Why It Matters

- `const` improves safety and clarity: makes interfaces self-documenting and helps the compiler catch mistakes.
- `volatile` preserves correctness in concurrent or hardware-driven systems.
- Together, they let you balance optimization with precision, critical in low-level C programming.

If you misuse or forget them:

- You risk accidental modification of data (`const`).
- You risk the compiler removing critical reads/writes (`volatile`).

Try It Yourself

1. Write a program that tries to modify a `const int` through a pointer, observe the compiler error.
2. Declare a variable as `volatile int counter` and increment it in a loop.
    - Then remove `volatile` and inspect the generated assembly with `gcc -S`.
3. Create a function with `const char *msg` and try to modify it, see why it's prohibited.
4. Experiment with `const int *p` vs `int *const p` to understand their difference.
5. Combine both: `const volatile int flag;` and print it in a loop.

In C, `const` and `volatile` are more than just keywords, they're contracts. They tell the compiler exactly how memory can be used, which helps both humans and machines reason safely about your code.

28. Function Pointers and Callbacks

Functions in C are values too, they live in memory and have addresses just like variables. A **function pointer** is a pointer that stores the address of a function, allowing you to call that function indirectly. This idea powers callbacks, event systems, custom sorters, and plug-in architectures in C.

What Is a Function Pointer?

Just like `int *` points to an integer, a **function pointer** points to a function.

Syntax:

```
return_type (*pointer_name)(parameter_types);
```

Example:

```
int add(int a, int b) {
    return a + b;
}

int (*func_ptr)(int, int) = add;
```

Now you can call the function through the pointer:

```
int result = func_ptr(2, 3); // same as add(2, 3)
```

Tiny Code

Here's a complete example showing how to declare, assign, and call function pointers:

```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }

void operate(int x, int y, int (*op)(int, int)) {
    printf("Result: %d\n", op(x, y)); // call through pointer
}

int main(void) {
    int (*f)(int, int); // declaration

    f = add;
    printf("Add via pointer: %d\n", f(5, 3));

    f = sub;
    printf("Subtract via pointer: %d\n", f(5, 3));

    printf("\nUsing callback function:\n");
    operate(4, 6, mul); // pass function pointer as argument

    return 0;
}
```

Compile and run:

```
gcc func_pointer_demo.c -o func_pointer_demo
./func_pointer_demo
```

Output:

```
Add via pointer: 8
Subtract via pointer: 2
```

```
Using callback function:
Result: 24
```

Function Pointers in Arrays

You can also store multiple function pointers in an array, useful for building tables of operations.

```c
int (*ops[3])(int, int) = {add, sub, mul};
for (int i = 0; i < 3; i++)
    printf("ops[%d](4, 2) = %d\n", i, ops[i](4, 2));
```

Output:

```
ops[0](4, 2) = 6
ops[1](4, 2) = 2
ops[2](4, 2) = 8
```

This pattern underlies dispatch tables, interpreters, and virtual function systems in C.

Callbacks

A **callback** is a function you pass as an argument to another function, letting the callee "call back" into user code. This pattern is essential in event-driven and modular designs.

Example: a simple iterator that accepts a callback

```c
#include <stdio.h>

void for_each(int *arr, int n, void (*callback)(int)) {
    for (int i = 0; i < n; i++)
        callback(arr[i]);
}

void print_square(int x) {
    printf("%d^2 = %d\n", x, x * x);
}

int main(void) {
    int nums[] = {1, 2, 3, 4, 5};
```

```
    for_each(nums, 5, print_square); // pass callback
    return 0;
}
```

Output:

```
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
```

Why It Matters

Function pointers let you:

- Select behavior at runtime (dynamic dispatch).
- Pass logic into libraries without recompiling them.
- Build frameworks, event handlers, interpreters, and plug-ins.
- Replace huge switch-case structures with elegant dispatch tables.

They are also how C implements:

- `qsort()` and `bsearch()` comparison functions,
- signal handlers (`signal(SIGINT, handler)`), and
- system callbacks in GUIs or kernels.

Try It Yourself

1. Write three arithmetic functions and store them in an array of function pointers.
2. Build a `calculate(a, b, char op)` function that picks the right function pointer based on `op`.
3. Implement a callback-style loop that calls a user-defined function for each array element.
4. Pass a function pointer to `qsort()` from `<stdlib.h>` to sort integers in descending order.
5. Write a small menu system that calls the right function based on user choice.

Function pointers and callbacks give your programs flexibility and abstraction without sacrificing speed. They're how C achieves dynamic behavior, the bridge between data and executable logic.

29. Deep vs Shallow Copies

When you assign one variable to another in C, you're often copying *addresses*, not *actual data*. This distinction between **shallow copies** and **deep copies** becomes critical when working with pointers, arrays, and dynamically allocated structures. Understanding it helps you prevent memory corruption, double frees, and mysterious bugs.

The Core Idea

- A **shallow copy** duplicates only the pointer, both variables refer to the *same memory*.
- A **deep copy** duplicates the *data itself*, each variable owns its own independent memory.

Simple Analogy

Think of shallow vs deep copy like two houses:

- **Shallow copy:** You hand someone your house key. You both open the same door.
- **Deep copy:** You build a new house that looks identical, but is separate.

Shallow Copy Example

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

int main(void) {
    char *original = malloc(10);
    strcpy(original, "Hello");

    // Shallow copy
    char *copy = original;

    printf("Before change: %s | %s\n", original, copy);

    copy[0] = 'J'; // modify one
    printf("After change: %s | %s\n", original, copy);

    free(original);
    // free(copy); //  would cause double free error!
```

```
    return 0;
}
```

Output:

```
Before change: Hello | Hello
After change: Jello | Jello
```

Explanation:

- `copy` points to the *same memory* as `original`.
- Changing one changes both.
- You must only `free()` it once, freeing both is a bug.

Deep Copy Example

A deep copy allocates new memory and copies the data over.

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

int main(void) {
    char *original = malloc(10);
    strcpy(original, "Hello");

    // Deep copy
    char *copy = malloc(strlen(original) + 1);
    strcpy(copy, original);

    printf("Before change: %s | %s\n", original, copy);

    copy[0] = 'J';
    printf("After change: %s | %s\n", original, copy);

    free(original);
    free(copy); //  both safely freed

    return 0;
}
```

Output:

```
Before change: Hello | Hello
After change: Hello | Jello
```

Now the two strings are completely independent, a true deep copy.

Shallow vs Deep in Structs

Consider this structure:

```c
typedef struct {
    char *name;
} Person;
```

If you assign one `Person` to another:

```c
Person a, b;
a.name = malloc(20);
strcpy(a.name, "Alice");

b = a; // shallow copy
b.name[0] = 'M'; // modifies a.name too!
```

Both `a` and `b` point to the same memory. To make a deep copy:

```c
b.name = malloc(strlen(a.name) + 1);
strcpy(b.name, a.name);
```

Now they're independent.

Tiny Code

Here's a full program demonstrating both copies with structs:

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

typedef struct {
    char *name;
    int age;
```

```c
} Person;

void print_person(const char *label, Person p) {
    printf("%s: name=%s age=%d\n", label, p.name, p.age);
}

int main(void) {
    Person p1;
    p1.name = malloc(20);
    strcpy(p1.name, "Alice");
    p1.age = 25;

    // Shallow copy
    Person p2 = p1;
    print_person("Before", p1);
    p2.name[0] = 'M'; // modifies same memory
    print_person("After shallow copy", p1);

    // Deep copy
    Person p3;
    p3.name = malloc(strlen(p1.name) + 1);
    strcpy(p3.name, p1.name);
    p3.age = p1.age;

    p3.name[0] = 'C'; // independent copy
    print_person("After deep copy", p1);
    print_person("Deep copy result", p3);

    free(p1.name);
    free(p3.name); //  safe
    return 0;
}
```

Output:

```
Before: name=Alice age=25
After shallow copy: name=Mlice age=25
After deep copy: name=Mlice age=25
Deep copy result: name=Clice age=25
```

Why It Matters

Shallow and deep copies determine **ownership** of memory:

- If two variables share the same pointer (shallow), freeing one invalidates the other.
- Deep copies isolate data, preventing interference but using more memory.

Getting this wrong leads to:

- Double free or dangling pointer errors
- Memory leaks
- Corrupted data in complex structures

Understanding these concepts is crucial for:

- Managing dynamic arrays and linked lists
- Designing APIs that safely return or duplicate data
- Writing custom copy constructors for structs

Try It Yourself

1. Create a struct with dynamically allocated fields (e.g., `name`, `address`) and write two copy functions: `copy_shallow()` and `copy_deep()`.
2. Modify one copy and observe the difference.
3. Call `free()` in the wrong order and note what happens.
4. Use Valgrind to verify that deep copies are properly freed.
5. Extend the concept to an array of structs, implement deep copy for each element.

When you understand deep vs shallow copies, you control **how memory ownership moves** in your program, a foundation for safe, modular, and leak-free C design.

30. Practice: Manual Memory Management

Now that you've learned how memory works, stack vs heap, allocation, freeing, leaks, deep vs shallow copies, it's time to practice **controlling memory manually**. This exercise ties together `malloc`, `free`, pointers, and struct management in a real, runnable program.

You'll build a small system that stores and manipulates dynamically allocated records, a tiny simulation of how databases or object systems manage memory in C.

Goal

Create a simple "student record manager" that can:

- Dynamically allocate memory for each student's name.
- Store and print student data.
- Free all allocated memory cleanly at the end.

Tiny Code

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

typedef struct {
    char *name;
    int age;
    float gpa;
} Student;

Student *create_student(const char *name, int age, float gpa) {
    Student *s = malloc(sizeof(Student));
    if (!s) {
        printf("Memory allocation failed for Student.\n");
        exit(1);
    }

    s->name = malloc(strlen(name) + 1);
    if (!s->name) {
        printf("Memory allocation failed for name.\n");
        free(s);
        exit(1);
    }

    strcpy(s->name, name);
    s->age = age;
    s->gpa = gpa;

    return s;
}
```

```c
void print_student(const Student *s) {
    printf("Name: %-10s | Age: %d | GPA: %.2f\n", s->name, s->age, s->gpa);
}

void free_student(Student *s) {
    free(s->name);
    free(s);
}

int main(void) {
    printf("=== Manual Memory Management Demo ===\n");

    // Create three students dynamically
    Student *a = create_student("Alice", 20, 3.8f);
    Student *b = create_student("Bob", 22, 3.4f);
    Student *c = create_student("Carol", 19, 3.9f);

    // Print their details
    print_student(a);
    print_student(b);
    print_student(c);

    // Modify dynamically allocated memory
    strcpy(b->name, "Bobby");
    b->gpa = 3.6f;
    printf("\nAfter update:\n");
    print_student(b);

    // Free memory
    free_student(a);
    free_student(b);
    free_student(c);

    printf("\nAll memory released.\n");
    return 0;
}
```

Compile and run:

```
gcc manual_memory.c -o manual_memory
./manual_memory
```

Output:

```
=== Manual Memory Management Demo ===
Name: Alice      | Age: 20 | GPA: 3.80
Name: Bob        | Age: 22 | GPA: 3.40
Name: Carol      | Age: 19 | GPA: 3.90

After update:
Name: Bobby      | Age: 22 | GPA: 3.60

All memory released.
```

How It Works

1. **Dynamic Allocation:** Each `Student` and its `name` field are created on the heap with `malloc()`. You control exactly when they exist and when to destroy them.
2. **Ownership:**
   - The program owns each student's memory.
   - Each `create_student()` call must later be matched by `free_student()`.
3. **Memory Safety:**
   - Every `malloc` result is checked.
   - Freed memory is properly released before exit.

Expanding the Example

Try these modifications:

1. **Dynamic Array of Students**

   ```c
   Student **students = malloc(3 * sizeof(Student *));
   students[0] = create_student("Ava", 21, 3.7f);
   students[1] = create_student("Ben", 20, 3.5f);
   students[2] = create_student("Cleo", 23, 3.9f);
   ```

   Iterate through them and print all details, then free each one.

2. **Reallocation (grow list)** Use `realloc()` to increase your array's capacity when adding more students dynamically.

3. **Deep Copy Function** Implement:

```
Student *copy_student(const Student *src);
```

   which performs a deep copy by allocating new memory for both the struct and its name.

4. **Leak Detection** Run your program with `valgrind ./manual_memory`, confirm that all memory is freed cleanly.

Why It Matters

This small example mirrors what real C systems do:

- Allocate complex data on demand.
- Manage lifetime explicitly.
- Clean up correctly.

Everything from operating systems to databases and compilers depends on this discipline. Once you can manage small dynamic structures like this confidently, you're ready to build larger systems safely, from allocators to object pools to file caches.

Try It Yourself

1. Add a new field (`major`) and handle it dynamically.
2. Add an array of grades and compute averages.
3. Convert your static list into a dynamically resizable array using `realloc`.
4. Intentionally omit a `free()` call, then detect the leak with Valgrind.
5. Write a `destroy_all()` function that frees an array of students safely.

You've now completed **Chapter 3: Working with Memory**. You understand how data lives, moves, and disappears in C, and you've practiced taking full control over it. From here, you'll learn how to **structure** that data elegantly using `struct`, `union`, and real-world data abstractions in Chapter 4.

Chapter 4. Structuring Data

31. Structures and Nested Structures

Real-world programs often deal with groups of related data, not just single variables. For example, a person has a name, an age, and an address. Instead of juggling separate variables, you can **combine them into a single structure** using `struct`.

`struct` is one of the most powerful features in C, it lets you define your own data types that group information logically and efficiently.

What Is a Structure?

A **structure** is a user-defined type that holds variables of different kinds under one name.

```
struct Person {
    char name[50];
    int age;
    float height;
};
```

This declares a *template* for a `Person` object. It doesn't create actual data yet, just the blueprint.

Declaring and Using Structures

You can now create variables of this new type:

```
#include <stdio.h>

struct Person {
    char name[50];
    int age;
    float height;
};
```

```c
int main(void) {
    struct Person p1 = {"Alice", 25, 1.65f};

    printf("Name: %s\n", p1.name);
    printf("Age: %d\n", p1.age);
    printf("Height: %.2f m\n", p1.height);

    return 0;
}
```

Output:

```
Name: Alice
Age: 25
Height: 1.65 m
```

Accessing Members

Use the dot operator `.` to access fields of a structure variable:

```c
p1.age = 26;
printf("Updated age: %d\n", p1.age);
```

If you have a pointer to a structure, use the arrow operator `->`:

```c
struct Person *ptr = &p1;
printf("Pointer access: %s is %d years old.\n", ptr->name, ptr->age);
```

Output:

```
Pointer access: Alice is 26 years old.
```

Initializing and Copying Structures

You can initialize a struct directly:

```c
struct Person p2 = {.name = "Bob", .age = 30, .height = 1.75f};
```

Structures can be assigned and copied by value:

```c
struct Person copy = p2;
printf("Copy: %s (%d)\n", copy.name, copy.age);
```

This performs a **shallow copy**, all fields are copied, but if any contain pointers, they'll still refer to the same memory (you'll learn how to make deep copies later).

Nested Structures

Structures can contain other structures. This helps you organize complex data clearly.

Example:

```c
#include <stdio.h>

struct Date {
    int day;
    int month;
    int year;
};

struct Student {
    char name[50];
    int id;
    struct Date birthdate; // nested structure
};

int main(void) {
    struct Student s = {
        .name = "Carol",
        .id = 1234,
        .birthdate = {15, 8, 2003}
    };

    printf("%s (ID %d) was born on %02d/%02d/%04d\n",
           s.name, s.id,
           s.birthdate.day, s.birthdate.month, s.birthdate.year);
    return 0;
}
```

Output:

```
Carol (ID 1234) was born on 15/08/2003
```

You access nested fields with the dot operator:

```
s.birthdate.year = 2004;
```

Structures and Functions

You can pass structs to functions by value or by pointer:

```
void print_person(struct Person p);
void update_age(struct Person *p, int new_age);
```

Example:

```
void update_age(struct Person *p, int new_age) {
    p->age = new_age;
}
```

Passing a pointer is more efficient, especially for large structures.

Tiny Code

Here's a full example combining everything above:

```
#include <stdio.h>

struct Date {
    int day;
    int month;
    int year;
};

struct Person {
    char name[50];
    int age;
    float height;
    struct Date birthdate;
};
```

```c
void print_person(const struct Person *p) {
    printf("%s, %d years old, born on %02d/%02d/%04d, height %.2fm\n",
           p->name, p->age,
           p->birthdate.day, p->birthdate.month, p->birthdate.year,
           p->height);
}

int main(void) {
    struct Person person = {"Alice", 25, 1.68f, {1, 2, 1999}};

    print_person(&person);

    person.age++;
    person.birthdate.year++;
    printf("After update:\n");
    print_person(&person);

    return 0;
}
```

Output:

```
Alice, 25 years old, born on 01/02/1999, height 1.68m
After update:
Alice, 26 years old, born on 01/02/2000, height 1.68m
```

Why It Matters

Structures let you:

- Combine related data into logical units.
- Model real-world entities directly in code.
- Pass data efficiently between functions.
- Build higher-level data abstractions like lists, trees, or objects.

They're the foundation of all complex C systems, files, network packets, kernel data, even database rows are built on top of `struct`.

Try It Yourself

1. Create a `Book` structure with `title`, `author`, and `year`.
2. Write a function to print all details of a `Book`.
3. Create a nested structure `Library` that contains multiple books.
4. Access a nested field (e.g., the title of the first book).
5. Modify the `Library` through a pointer using the `->` operator.

Structures are how C lets you **model the world**, compact, explicit, and fast. Next, you'll learn about **unions** and how C lets different data types share the same memory space efficiently.

32. Unions and Type Reuse

Sometimes you need a variable that can hold *different types of data* at different times, but you don't want to waste memory keeping all of them active at once. That's where **unions** come in.

A `union` lets multiple fields share the *same memory location*. It's a space-saving feature and a powerful tool for implementing type flexibility, variant data, and even low-level binary manipulation.

What Is a Union?

A **union** is like a structure, but instead of giving each member its own memory, *all members share the same memory block*. Only one field is valid at any moment.

Syntax:

```
union Data {
    int i;
    float f;
    char c;
};
```

Here, `i`, `f`, and `c` share the same storage. The size of the union is equal to the size of its *largest* member.

Using a Union

```c
#include <stdio.h>

union Data {
    int i;
    float f;
    char c;
};

int main(void) {
    union Data d;

    d.i = 42;
    printf("d.i = %d\n", d.i);

    d.f = 3.14f;
    printf("d.f = %.2f\n", d.f);

    d.c = 'A';
    printf("d.c = %c\n", d.c);

    // The last assignment overwrites the previous ones
    printf("After d.c = 'A', d.i = %d (corrupted)\n", d.i);

    return 0;
}
```

Output:

```
d.i = 42
d.f = 3.14
d.c = A
After d.c = 'A', d.i = 1094795585
```

Notice how writing to one member affects the others, because they occupy the same memory.

Memory Layout Illustration

```
+------------------+
| Shared Memory    |  <- same location for all fields
| (size = largest) |
+------------------+
```

```
| i: 4 bytes       |
| f: 4 bytes       |
| c: 1 byte        |
+------------------+
```

All fields overlap in the same storage area.

Tiny Code

Let's see a practical example where a union saves memory.

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

union Value {
    int i;
    float f;
    char str[20];
};

int main(void) {
    union Value v;

    v.i = 42;
    printf("As int: %d\n", v.i);

    v.f = 3.14f;
    printf("As float: %.2f\n", v.f);

    strcpy(v.str, "Hello");
    printf("As string: %s\n", v.str);

    printf("Union size: %zu bytes\n", sizeof(v));
    return 0;
}
```

Output:

```
As int: 42
As float: 3.14
As string: Hello
Union size: 20 bytes
```

Even though it contains an `int`, a `float`, and a `char[20]`, the total size is only 20 bytes, the size of the largest member.

Tagged Unions (Type-Safe Pattern)

In practice, you often use a **tag** (an enum or integer) to remember which member is active, this is known as a *tagged union* or *discriminated union*.

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

enum Type { INT, FLOAT, STRING };

struct Variant {
    enum Type type;
    union {
        int i;
        float f;
        char str[20];
    } data;
};

void print_variant(const struct Variant *v) {
    switch (v->type) {
        case INT:   printf("INT: %d\n", v->data.i); break;
        case FLOAT: printf("FLOAT: %.2f\n", v->data.f); break;
        case STRING:printf("STRING: %s\n", v->data.str); break;
    }
}

int main(void) {
    struct Variant v;

    v.type = STRING;
    strcpy(v.data.str, "C Language");
    print_variant(&v);

    v.type = INT;
    v.data.i = 123;
    print_variant(&v);

    v.type = FLOAT;
```

```c
    v.data.f = 9.81f;
    print_variant(&v);

    return 0;
}
```

Output:

```
STRING: C Language
INT: 123
FLOAT: 9.81
```

This is how you combine the flexibility of unions with the safety of knowing which field is currently valid.

Why It Matters

Unions are crucial for:

- **Saving memory**, only one field exists at a time.
- **Implementing variant data types**, e.g., JSON values, expression trees, network packets.
- **Working with hardware registers**, map one register into multiple view types.
- **Binary serialization**, reinterpret raw bytes as various data forms.

In low-level systems, they enable compact and flexible representations that C is famous for.

Try It Yourself

1. Write a union `Number` with `int`, `float`, and `double`.
    - Print its size and observe which member determines it.
2. Implement a tagged union `Message` with types TEXT, BINARY, and COMMAND.
3. Create a struct with an enum tag and a union, simulate how file formats (like PNG chunks) are parsed.
4. Write a function that prints the active union field using the tag.
5. Modify the previous example to store an array of tagged unions.

In C, **unions give you memory control and flexibility** that few languages allow. They're the foundation for advanced constructs like variant types, polymorphic structs, and even message protocols, used everywhere from the Linux kernel to embedded firmware.

33. Typedef and Code Clarity

C gives you the power to create your own type names using the `typedef` keyword. It doesn't create new types at runtime, instead, it creates **aliases** that make your code cleaner, more expressive, and easier to maintain.

If `struct`, `enum`, or pointer syntax ever feels cluttered, `typedef` is your best friend.

What Is `typedef`?

`typedef` gives an existing type a new name. It's like a *nickname* for a complex or frequently used declaration.

Syntax:

```
typedef existing_type new_name;
```

Example:

```
typedef unsigned long ulong;
```

Now `ulong` can be used wherever you'd normally write `unsigned long`.

Basic Examples

```c
#include <stdio.h>

typedef int Score;
typedef char Letter;

int main(void) {
    Score math = 95;
    Letter grade = 'A';
    printf("Math: %d, Grade: %c\n", math, grade);
    return 0;
}
```

Output:

```
Math: 95, Grade: A
```

It doesn't change how the compiler treats the variable, just makes the code more readable.

Typedef with Pointers

Pointer declarations can get messy. With `typedef`, you can simplify them.

```c
typedef int* IntPtr;

int main(void) {
    int x = 10;
    IntPtr p = &x; // same as int *p = &x;
    printf("Value: %d\n", *p);
    return 0;
}
```

Output:

```
Value: 10
```

> **Tip:** Be careful, `IntPtr a, b;` means *both* `a` and `b` are pointers, unlike plain `int *a, b;`.

Typedef with Structures

`typedef` shines with `struct`, `union`, and `enum` declarations. Without typedef:

```c
struct Point {
    int x;
    int y;
};

struct Point p1 = {3, 4};
```

With typedef:

```c
typedef struct {
    int x;
    int y;
} Point;

Point p1 = {3, 4};
```

No more need to repeat the word `struct` everywhere.

Combining Struct and Typedef

You can define and alias a struct in one line:

```c
typedef struct Person {
    char name[50];
    int age;
} Person;
```

Now you can write:

```c
Person p = {"Alice", 25};
```

instead of:

```c
struct Person p = {"Alice", 25};
```

Typedef with Function Pointers

Function pointer syntax can be hard to read, typedef simplifies it dramatically.

Without typedef:

```c
int (*operation)(int, int);
```

With typedef:

```c
typedef int (*Operation)(int, int);

int add(int a, int b) { return a + b; }

int main(void) {
    Operation op = add;
    printf("%d\n", op(2, 3));
}
```

Now `Operation` is a clean alias for a pointer to a function that takes two ints and returns an int.

Tiny Code

Here's a full example showing typedefs for structs, pointers, and function types:

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

typedef struct {
    char name[50];
    int age;
} Person;

typedef Person* PersonPtr;
typedef void (*Printer)(const Person*);

void print_person(const Person *p) {
    printf("%s (%d years old)\n", p->name, p->age);
}

int main(void) {
    Person p = {"Alice", 25};
    PersonPtr ptr = &p;
    Printer print = print_person;

    print(ptr);
    return 0;
}
```

Output:

```
Alice (25 years old)
```

Why It Matters

`typedef` improves:

- **Clarity:** Long or complex declarations become readable.
- **Consistency:** Standardize naming (e.g., `size_t`, `uint32_t`).
- **Maintainability:** Changing underlying types is easier, one typedef change updates all uses.
- **Abstraction:** Hides implementation details, especially in headers.

It's especially useful in large projects and system APIs, where naming conventions define clean boundaries.

Common Typedef Patterns

Purpose	Example
Standard aliases	`typedef unsigned int uint;`
Struct abstraction	`typedef struct Node Node;`
Function pointer type	`typedef void (*Handler)(int signal);`
Handle-like pattern	`typedef struct File* FileHandle;`
Platform types	`typedef long long int64; typedef unsigned int uint32;`

Try It Yourself

1. Define a `typedef` for `unsigned long long` called `u64`.
2. Create a `typedef struct` called `Book` with fields for title and pages.
3. Define a `typedef` for a function pointer `Comparator(int, int)`.
4. Write a program that passes a `Comparator` to a sorting function.
5. Modify one typedef and observe how the code compiles cleanly without further edits.

`typedef` may look simple, but it's one of the most powerful readability tools in C. It lets you design your own vocabulary for your system, a small step toward writing clean, self-documenting code that scales.

34. Bitfields and Memory Packing

C lets you control data layout down to the *bit level* using **bitfields**. They allow you to store small values compactly inside a struct, perfect for flags, configuration registers, or communication protocols. Combined with *packing*, you can squeeze data into minimal space while still keeping it easy to manipulate symbolically.

What Is a Bitfield?

A **bitfield** lets you define the exact number of bits to allocate for a field inside a `struct`.

Example:

```
struct Flags {
    unsigned int is_visible : 1;
    unsigned int is_enabled : 1;
    unsigned int has_error  : 1;
};
```

Here, each field uses just **1 bit** instead of a full 4-byte `int`. That means 8 such flags fit comfortably in one byte.

Declaring and Using Bitfields

```
#include <stdio.h>

struct Status {
    unsigned int connected : 1;
    unsigned int error     : 1;
    unsigned int active    : 1;
    unsigned int reserved  : 5; // padding bits
};

int main(void) {
    struct Status s = {1, 0, 1, 0};
    printf("Connected: %u, Active: %u\n", s.connected, s.active);
    s.error = 1;
    printf("Error now: %u\n", s.error);
    return 0;
}
```

Output:

```
Connected: 1, Active: 1
Error now: 1
```

Even though there are 4 fields, the entire struct typically occupies only 1 byte.

Tiny Code

Here's a complete example demonstrating packed flags and printing bit values:

```c
#include <stdio.h>

struct DeviceStatus {
    unsigned int powered_on : 1;
    unsigned int connected  : 1;
    unsigned int has_error  : 1;
    unsigned int battery_low: 1;
    unsigned int reserved   : 4;
};

void print_bits(unsigned char byte) {
    for (int i = 7; i >= 0; i--)
        printf("%d", (byte >> i) & 1);
    printf("\n");
}

int main(void) {
    struct DeviceStatus d = {1, 1, 0, 0, 0};
    printf("Size of DeviceStatus: %zu bytes\n", sizeof(d));

    unsigned char *raw = (unsigned char*)&d;
    printf("Binary layout: ");
    print_bits(*raw);

    d.has_error = 1;
    printf("Updated binary: ");
    print_bits(*raw);

    return 0;
}
```

Output (may vary by platform):

```
Size of DeviceStatus: 1 bytes
Binary layout: 00000011
Updated binary: 00000111
```

Nested Bitfields Example

You can even use bitfields inside nested structs to create compact yet expressive data models:

```
struct Sensor {
    unsigned id       : 4;  // 0-15
    unsigned type     : 3;  // 0-7
    unsigned active   : 1;  // boolean
};

struct Device {
    struct Sensor sensors[2];
};
```

Each sensor entry now fits neatly into a single byte.

Memory Packing

By default, compilers may insert **padding bytes** to align fields for faster access. If you want tighter packing, for example, when saving binary data to a file or sending over a network, you can request packed structs.

Compiler directives differ by system:

```
#pragma pack(push, 1)
struct Packet {
    char type;
    unsigned int length;
    short checksum;
};
#pragma pack(pop)
```

Now the struct is tightly packed without alignment padding between fields.

Bitfields in Real Systems

Bitfields are used everywhere in systems programming:

- **Hardware control registers:** Represent on/off bits for devices.
- **Network protocols:** Flags in TCP, UDP, or IP headers.
- **Compression and serialization:** Compact representation of status or metadata.
- **Embedded systems:** Save every byte of RAM in microcontrollers.

Limitations

- Bitfield ordering (which bit is "first") is **implementation-dependent**, varies by compiler and platform.
- They cannot cross word boundaries reliably across architectures.
- Bitfields cannot be directly addressed by pointers (`&field` not allowed).
- Endianness matters if transmitting across different systems.

  For portable bit-level control (especially in networking), many engineers use explicit bitwise operators instead.

Manual Bitwise Control

Sometimes you'll prefer manual masks and shifts:

```
unsigned char flags = 0;
flags |= (1 << 0); // set bit 0
flags |= (1 << 2); // set bit 2
flags &= ~(1 << 0); // clear bit 0
```

This approach is more portable and explicit, but less readable for large sets of flags.

Why It Matters

Bitfields give you **compact control** over memory layout and binary representation. They're critical in:

- Embedded firmware
- Network stacks
- Kernel drivers
- Compression libraries

They make your code expressive *and* efficient, as long as you understand alignment and portability issues.

Try It Yourself

1. Define a struct `Permissions` with 1-bit fields for `read`, `write`, and `execute`.
2. Print its size and check how compact it is.
3. Use a bitfield struct to represent a simplified TCP header (flags like SYN, ACK, FIN).
4. Use `#pragma pack(1)` and observe the size difference.
5. Write functions to set, clear, and toggle bits using bitwise operators.

Bitfields are where C meets the hardware. They let you talk to the machine not just in bytes, but in **bits**, the true language of computers. Next, you'll revisit enumerations and see how they complement these compact structures by giving symbolic meaning to values.

35. Enumerations Revisited

You've seen `enum` briefly when learning about constants, but now it's time to use it as a **first-class design tool**. Enumerations give names to sets of integer values, making code easier to read, maintain, and debug. They also pair beautifully with `struct`, `union`, and `bitfield` patterns from the previous sections.

What Is an Enumeration?

An **enumeration** (`enum`) defines a type whose values are limited to a specific list of named constants.

Example:

```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

Under the hood, `enum Color` is an integer type — `RED` = 0, `GREEN` = 1, `BLUE` = 2 by default.

Basic Usage

```c
#include <stdio.h>

enum Direction {
    NORTH,
    EAST,
    SOUTH,
    WEST
};

int main(void) {
    enum Direction d = EAST;
    printf("Current direction: %d\n", d);
    return 0;
}
```

Output:

```
Current direction: 1
```

Even though `EAST` prints as `1`, using a named constant makes your code far more meaningful.

Assigning Custom Values

You can specify explicit integer values, useful for compatibility or mapping to real-world codes.

```c
enum ErrorCode {
    OK = 0,
    FILE_NOT_FOUND = 404,
    SERVER_ERROR = 500
};
```

If you skip a value, enumeration continues counting from the last number:

```c
enum Level {
    LOW = 1,
    MEDIUM,
    HIGH
};
// HIGH = 3
```

Tiny Code

A small program that uses enums for clear program flow:

```c
#include <stdio.h>

enum Status {
    SUCCESS = 0,
    WARNING = 1,
    ERROR = 2
};

const char* status_to_string(enum Status s) {
    switch (s) {
        case SUCCESS: return "Success";
        case WARNING: return "Warning";
        case ERROR:   return "Error";
        default:      return "Unknown";
    }
}

int main(void) {
    enum Status s = WARNING;
    printf("Status: %s (%d)\n", status_to_string(s), s);
    return 0;
}
```

Output:

```
Status: Warning (1)
```

This pattern, `enum` + `switch`, is everywhere in C projects: error handling, state machines, network protocols, and more.

Enumerations with Structs

Combine `enum` with `struct` for self-describing data:

```c
#include <stdio.h>

enum ShapeType {
    CIRCLE,
    RECTANGLE
};

struct Shape {
    enum ShapeType type;
    union {
        struct { float radius; };
        struct { float width, height; };
    };
};

void print_shape(struct Shape s) {
    if (s.type == CIRCLE)
        printf("Circle with radius %.2f\n", s.radius);
    else
        printf("Rectangle %.2fx%.2f\n", s.width, s.height);
}

int main(void) {
    struct Shape c = {CIRCLE, .radius = 2.5f};
    struct Shape r = {RECTANGLE, .width = 3.0f, .height = 4.0f};
    print_shape(c);
    print_shape(r);
    return 0;
}
```

Output:

```
Circle with radius 2.50
Rectangle 3.00x4.00
```

This pairing of `enum` + `union` is a common pattern in real-world systems, known as a **tagged union**.

Scoped Enums in Modern C (C23)

C23 introduces a cleaner way to scope enum names:

```
enum class Mode { READ, WRITE, APPEND };
```

This avoids name collisions and allows better type checking — similar to `enum class` in C++. (Some compilers may not support this yet, but it's worth knowing.)

Why It Matters

Enumerations bring **semantic clarity** to your code:

- Replace "magic numbers" with meaningful names.
- Simplify debugging and logging.
- Enable compile-time checking, you can't assign invalid constants easily.
- Combine cleanly with structs, unions, and bitfields to express state machines or protocols.

They also improve portability, your program logic is described by **intent**, not arbitrary numbers.

Try It Yourself

1. Define an enum `TrafficLight { RED, YELLOW, GREEN }` and print messages based on its value.
2. Create an enum `FileType { TEXT, BINARY, UNKNOWN }` and use it inside a `struct FileInfo`.
3. Extend your tagged-union pattern: add `TRIANGLE` to the `Shape` enum.
4. Write a `switch` statement that maps `enum ErrorCode` to error messages.
5. Experiment with explicitly setting values and skipping a few, observe the auto-increment behavior.

Enumerations make your programs *speak in concepts, not numbers*. They are the key to clarity, readability, and robust design, the bridge between human meaning and machine representation.

36. Linked Lists from Scratch

Now that you understand how to group data with `struct`, it's time to make it **dynamic**. A **linked list** is one of the most fundamental data structures in C, built entirely with pointers and structs. It teaches you how memory, pointers, and iteration really work.

What Is a Linked List?

A **linked list** is a collection of nodes, where each node stores:

1. Data (of any type you choose), and
2. A pointer to the next node.

Unlike arrays, linked lists aren't fixed in size, you can add or remove nodes anytime without reallocating large blocks of memory.

Basic Node Structure

```
struct Node {
    int value;
    struct Node *next;
};
```

This defines a "node" that holds an integer and a pointer to the next node in the list. If `next` is `NULL`, it's the end of the list.

Creating and Traversing a Linked List

Let's build a simple three-node list:

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

struct Node {
    int value;
    struct Node *next;
};

int main(void) {
    // Create three nodes dynamically
    struct Node *a = malloc(sizeof(struct Node));
    struct Node *b = malloc(sizeof(struct Node));
    struct Node *c = malloc(sizeof(struct Node));

    a->value = 10; a->next = b;
    b->value = 20; b->next = c;
```

```
    c->value = 30; c->next = NULL;

    // Traverse and print
    struct Node *current = a;
    while (current != NULL) {
        printf("%d ", current->value);
        current = current->next;
    }
    printf("\n");

    // Free memory
    free(a); free(b); free(c);
    return 0;
}
```

Output:

```
10 20 30
```

Tiny Code: Build, Insert, Delete

Let's make it reusable, define helper functions for common operations.

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node* create_node(int value) {
    Node *n = malloc(sizeof(Node));
    n->value = value;
    n->next = NULL;
    return n;
}

void append(Node **head, int value) {
    Node *new_node = create_node(value);
    if (*head == NULL) {
```

```
        *head = new_node;
        return;
    }
    Node *cur = *head;
    while (cur->next) cur = cur->next;
    cur->next = new_node;
}

void print_list(const Node *head) {
    for (const Node *p = head; p != NULL; p = p->next)
        printf("%d -> ", p->value);
    printf("NULL\n");
}

void delete_list(Node *head) {
    while (head) {
        Node *next = head->next;
        free(head);
        head = next;
    }
}

int main(void) {
    Node *head = NULL;
    append(&head, 5);
    append(&head, 10);
    append(&head, 15);

    printf("Linked list contents:\n");
    print_list(head);

    delete_list(head);
    return 0;
}
```

Output:

```
Linked list contents:
5 -> 10 -> 15 -> NULL
```

Why Use Linked Lists?

- **Dynamic size:** Easily grow or shrink as needed.
- **Efficient insertions and deletions:** No need to shift elements as in arrays.
- **Great for learning memory handling:** You directly allocate and free each node.

But they also have trade-offs:

- Slower random access (must traverse from the head).
- Slightly higher memory usage due to pointer fields.

Variants You'll Meet Later

Type	Description
Singly Linked List	Each node points to the next one (like above).
Doubly Linked List	Each node has `prev` and `next` pointers.
Circular Linked List	The last node links back to the first.
Sentinel List	Uses dummy head/tail nodes to simplify logic.

Why It Matters

Linked lists are a window into **manual memory management**, you handle creation, traversal, and cleanup. They're used in:

- Kernels (e.g., Linux `list_head`)
- Compilers (symbol tables, token streams)
- Dynamic containers (queues, allocators)

You're not just learning a data structure, you're learning how to think in pointers.

Try It Yourself

1. Implement a function `int length(Node *head)` that counts the number of nodes.
2. Write `insert_front()` and `insert_after()` functions.
3. Implement a `find()` function that returns a pointer to a node with a given value.
4. Modify the `delete_list()` function to print which node is being freed.
5. Extend the struct to include a `char name[20]` and print both the name and value.

You've now built one of the most essential dynamic structures in computer science, entirely from scratch. Next, you'll build on this foundation to create **stacks** and **queues**, two of the most common and useful data abstractions in systems programming.

37. Stacks and Queues with Structs

You've learned how to build a **linked list**, now you'll use that foundation to create two classic data structures: **Stacks** (LIFO, Last In, First Out) and **Queues** (FIFO, First In, First Out). Both are essential for real-world programs, from parsing expressions to managing tasks and kernel scheduling.

1. The Stack

A **stack** is like a pile of plates. You add to the top (*push*), and remove from the top (*pop*).

Operations:

- `push(x)` → add an item to the top
- `pop()` → remove the top item
- `peek()` → look at the top item without removing it

Stack Implementation Using Linked List

Each stack node holds data and a pointer to the next node.

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

typedef struct {
    Node *top;
} Stack;

Stack* create_stack(void) {
    Stack *s = malloc(sizeof(Stack));
    s->top = NULL;
    return s;
}

void push(Stack *s, int value) {
    Node *n = malloc(sizeof(Node));
    n->value = value;
```

```
    n->next = s->top;
    s->top = n;
}

int pop(Stack *s) {
    if (!s->top) {
        printf("Stack underflow!\n");
        return -1;
    }
    Node *temp = s->top;
    int val = temp->value;
    s->top = temp->next;
    free(temp);
    return val;
}

int peek(const Stack *s) {
    return s->top ? s->top->value : -1;
}

void free_stack(Stack *s) {
    while (s->top) pop(s);
    free(s);
}

int main(void) {
    Stack *s = create_stack();
    push(s, 10);
    push(s, 20);
    push(s, 30);

    printf("Top: %d\n", peek(s));
    printf("Popped: %d\n", pop(s));
    printf("Popped: %d\n", pop(s));
    printf("Top now: %d\n", peek(s));

    free_stack(s);
    return 0;
}
```

Output:

```
Top: 30
Popped: 30
Popped: 20
Top now: 10
```

Why Use a Stack?

Stacks are used in:

- Function calls (the *call stack*)
- Undo/redo systems
- Parsing expressions (e.g., evaluating `(2 + 3) * 4`)
- Depth-first search (DFS) in graphs

2. The Queue

A **queue** is like a line at a store. You add to the back (*enqueue*), and remove from the front (*dequeue*).

Operations:

- `enqueue(x)` → add to the end
- `dequeue()` → remove from the front

Queue Implementation Using Linked List

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

typedef struct {
    Node *front;
    Node *rear;
} Queue;

Queue* create_queue(void) {
    Queue *q = malloc(sizeof(Queue));
```

```
    q->front = q->rear = NULL;
    return q;
}

void enqueue(Queue *q, int value) {
    Node *n = malloc(sizeof(Node));
    n->value = value;
    n->next = NULL;

    if (q->rear == NULL) {
        q->front = q->rear = n;
        return;
    }
    q->rear->next = n;
    q->rear = n;
}

int dequeue(Queue *q) {
    if (q->front == NULL) {
        printf("Queue underflow!\n");
        return -1;
    }
    Node *temp = q->front;
    int val = temp->value;
    q->front = temp->next;

    if (q->front == NULL)
        q->rear = NULL;

    free(temp);
    return val;
}

void print_queue(const Queue *q) {
    for (Node *p = q->front; p != NULL; p = p->next)
        printf("%d ", p->value);
    printf("\n");
}

void free_queue(Queue *q) {
    while (q->front) dequeue(q);
    free(q);
```

```
}

int main(void) {
    Queue *q = create_queue();
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    printf("Queue: ");
    print_queue(q);

    printf("Dequeued: %d\n", dequeue(q));
    printf("Dequeued: %d\n", dequeue(q));

    printf("Remaining: ");
    print_queue(q);

    free_queue(q);
    return 0;
}
```

Output:

```
Queue: 1 2 3
Dequeued: 1
Dequeued: 2
Remaining: 3
```

Stack vs Queue Summary

Feature	Stack	Queue
Access Order	LIFO (Last In, First Out)	FIFO (First In, First Out)
Main Operations	push / pop	enqueue / dequeue
Used For	Recursion, parsing, backtracking	Task scheduling, buffering
Example	Undo system	Printer jobs

Tiny Code Exercise: Dual Queue-Stack

Here's a minimal snippet that lets you switch between stack and queue mode:

```c
typedef enum { STACK_MODE, QUEUE_MODE } Mode;
```

You could use the same linked list logic but change whether new nodes are added at the head (stack) or tail (queue).

Why It Matters

Stack and queue behavior underlie **every major system abstraction**:

- CPU scheduling
- IO buffering
- Event loops
- Expression parsing
- Recursive algorithms

Building them in raw C solidifies your understanding of **pointer-based data structures** and **memory ownership**.

Try It Yourself

1. Implement `is_empty()` for both stack and queue.
2. Extend the queue to handle strings instead of ints.
3. Add a function `reverse_queue()` using a stack.
4. Implement a "bounded queue" that has a fixed maximum size.
5. Write a small program simulating customer arrivals using a queue.

Stacks and queues are the **control flow primitives of memory and time**. Next, you'll combine them with hashing and function pointers to build your own **hash table**, the basis for efficient lookups and symbol tables in C.

38. Hash Tables and Function Pointers

Hash tables are among the most important data structures in computing, fast, flexible, and foundational. They give you **average O(1)** lookup, insertion, and deletion by mapping *keys* to *values* through a hash function. In this section, you'll build a simple hash table from scratch in C using **structs**, **arrays**, and **function pointers** for hash and comparison operations.

What Is a Hash Table?

A **hash table** stores data as key–value pairs. When you insert a key:

1. The **hash function** converts it into an integer index.
2. The data is stored in that slot of an array.
3. When you search later, the key is hashed again to find the same index.

If multiple keys map to the same slot, that's called a **collision**, handled by *chaining* (linked lists) or *open addressing*.

Simple Design

We'll use **chaining**, each slot in the table is a linked list of key–value pairs that share the same hash.

```
typedef struct Entry {
    char *key;
    int value;
    struct Entry *next;
} Entry;

typedef struct {
    Entry **buckets; // array of linked lists
    size_t size;
} HashTable;
```

Hash Function

A basic string hash function (the djb2 algorithm):

```
#include <stddef.h>

unsigned long hash_string(const char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c; // hash * 33 + c
    return hash;
}
```

Tiny Code: Hash Table Implementation

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

typedef struct Entry {
    char *key;
    int value;
    struct Entry *next;
} Entry;

typedef struct {
    Entry **buckets;
    size_t size;
} HashTable;

unsigned long hash_string(const char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash;
}

HashTable* create_table(size_t size) {
    HashTable *t = malloc(sizeof(HashTable));
    t->size = size;
    t->buckets = calloc(size, sizeof(Entry*));
    return t;
}

Entry* create_entry(const char *key, int value) {
    Entry *e = malloc(sizeof(Entry));
    e->key = strdup(key);
    e->value = value;
    e->next = NULL;
    return e;
}

void insert(HashTable *t, const char *key, int value) {
    unsigned long index = hash_string(key) % t->size;
```

```
    Entry *new_entry = create_entry(key, value);
    new_entry->next = t->buckets[index];
    t->buckets[index] = new_entry;
}

Entry* search(HashTable *t, const char *key) {
    unsigned long index = hash_string(key) % t->size;
    for (Entry *e = t->buckets[index]; e; e = e->next)
        if (strcmp(e->key, key) == 0)
            return e;
    return NULL;
}

void free_table(HashTable *t) {
    for (size_t i = 0; i < t->size; i++) {
        Entry *e = t->buckets[i];
        while (e) {
            Entry *next = e->next;
            free(e->key);
            free(e);
            e = next;
        }
    }
    free(t->buckets);
    free(t);
}

int main(void) {
    HashTable *table = create_table(8);

    insert(table, "apple", 5);
    insert(table, "banana", 7);
    insert(table, "orange", 10);

    Entry *result = search(table, "banana");
    if (result)
        printf("banana = %d\n", result->value);
    else
        printf("Key not found\n");

    free_table(table);
    return 0;
```

```
}
```

Output:

```
banana = 7
```

Using Function Pointers for Genericity

We can make the hash table generic by letting users provide custom hash and compare functions:

```
typedef unsigned long (*HashFunc)(const void*);
typedef int (*CompareFunc)(const void*, const void*);
```

Then we embed them in the struct:

```
typedef struct {
    Entry **buckets;
    size_t size;
    HashFunc hash;
    CompareFunc compare;
} GenericTable;
```

This lets you reuse the same table for strings, integers, or structs, just provide the right hash and compare functions.

Example:

```
unsigned long hash_int(const void *p) {
    return (*(int*)p) * 2654435761u;
}
```

Why It Matters

Hash tables power:

- **Compilers** (symbol tables, variable scopes)
- **Databases and caches** (key-value stores)
- **Operating systems** (file descriptor maps, kernel objects)
- **Network stacks** (routing tables, ARP caches)

They balance **speed**, **simplicity**, and **control**, the heart of efficient system design in C.

Common Pitfalls

- Forgetting to handle collisions (loses data).
- Failing to free all nodes → memory leaks.
- Using poor hash functions → clustering, performance drops.
- Not resizing when full → reduced efficiency.

A well-designed hash table grows dynamically (doubling capacity and rehashing when load exceeds a threshold).

Try It Yourself

1. Modify the table to **update** existing keys instead of always inserting new ones.
2. Implement a **delete(key)** function that removes an entry.
3. Write a version with **integer keys**.
4. Implement **rehash()** that doubles table size when 75% full.
5. Replace function pointers with macros for performance comparison.

Hash tables are where C shows its full power: raw pointers, function indirection, and dynamic memory, all working together for blazing-fast lookups. Next, you'll take these ideas further and explore how to simulate **object-oriented design in C** using structs, function pointers, and encapsulation.

39. Minimal Object-Oriented Design in C

C doesn't have classes or inheritance, but it gives you **structs**, **function pointers**, and **encapsulation through conventions**. With these, you can build *object-oriented style* systems that are simple, fast, and explicit. You'll learn how to design data structures that "own" both **data and behavior**, like lightweight objects.

The Core Idea

In object-oriented design, an object combines:

- **Data** → the state
- **Functions** → the operations

In C, you can achieve this by placing **function pointers inside structs**, and treating them as "methods."

A Simple Example: A Counter Object

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

typedef struct Counter Counter; // forward declaration

struct Counter {
    int value;

    // methods (function pointers)
    void (*inc)(Counter *self);
    void (*reset)(Counter *self);
    void (*print)(const Counter *self);
};

void counter_inc(Counter *self) { self->value++; }
void counter_reset(Counter *self) { self->value = 0; }
void counter_print(const Counter *self) { printf("Value: %d\n", self->value); }

Counter* new_counter(void) {
    Counter *c = malloc(sizeof(Counter));
    c->value = 0;
    c->inc = counter_inc;
    c->reset = counter_reset;
    c->print = counter_print;
    return c;
}

void free_counter(Counter *c) { free(c); }

int main(void) {
    Counter *c = new_counter();
    c->inc(c);
    c->inc(c);
    c->print(c);
    c->reset(c);
    c->print(c);
    free_counter(c);
    return 0;
}
```

Output:

```
Value: 2
Value: 0
```

Here, `Counter` behaves like a small class: it stores both the **state** (`value`) and its **methods** (`inc`, `reset`, `print`).

How It Works

Concept (OOP)	Equivalent in C
Class	`struct` definition
Object	An instance (`malloc`ed struct)
Method	Function pointer
Constructor	`new_...()` function
Destructor	`free_...()` function
this	Pointer to the struct (`self`)

Example: Shape Interface (Polymorphism)

You can simulate polymorphism, the ability to call the same function name on different types, using function pointers.

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

typedef struct Shape Shape;
struct Shape {
    double (*area)(Shape *self);
    void (*print)(Shape *self);
};

typedef struct {
    Shape base;
    double radius;
} Circle;

typedef struct {
```

```
    Shape base;
    double width, height;
} Rectangle;

double circle_area(Shape *s) {
    Circle *c = (Circle*)s;
    return M_PI * c->radius * c->radius;
}

void circle_print(Shape *s) {
    Circle *c = (Circle*)s;
    printf("Circle (r=%.2f) area=%.2f\n", c->radius, circle_area(s));
}

double rect_area(Shape *s) {
    Rectangle *r = (Rectangle*)s;
    return r->width * r->height;
}

void rect_print(Shape *s) {
    Rectangle *r = (Rectangle*)s;
    printf("Rectangle (%.2fx%.2f) area=%.2f\n",
           r->width, r->height, rect_area(s));
}

Shape* new_circle(double r) {
    Circle *c = malloc(sizeof(Circle));
    c->radius = r;
    c->base.area = circle_area;
    c->base.print = circle_print;
    return (Shape*)c;
}

Shape* new_rectangle(double w, double h) {
    Rectangle *r = malloc(sizeof(Rectangle));
    r->width = w;
    r->height = h;
    r->base.area = rect_area;
    r->base.print = rect_print;
    return (Shape*)r;
}
```

```c
int main(void) {
    Shape *s1 = new_circle(2.5);
    Shape *s2 = new_rectangle(3.0, 4.0);

    s1->print(s1);
    s2->print(s2);

    free(s1);
    free(s2);
    return 0;
}
```

Output:

```
Circle (r=2.50) area=19.63
Rectangle (3.00x4.00) area=12.00
```

Both shapes share the same "interface" (`area`, `print`) but behave differently, classic polymorphism.

Why This Works

Every "object" stores pointers to its **methods**, so you can call them without knowing the exact type. The first field (`base`) in derived structs allows casting between the parent (`Shape*`) and child (`Circle*`, `Rectangle*`). This mimics *inheritance by composition*.

Benefits

- Provides **clear separation** between interface and implementation.
- Enables **runtime dispatch** (function behavior depends on type).
- Keeps code modular, functions can operate on abstract "objects."
- Used in major C projects like the Linux kernel, GTK, and SQLite.

Limitations

- No true type safety, casts can go wrong.
- No automatic destructors or constructors (you must manage memory).
- No inheritance syntax, everything is explicit.

But these are also *strengths*: nothing is hidden, and everything is under your control.

Try It Yourself

1. Add a new shape: `Triangle` with base and height.
2. Write a function `print_all(Shape **arr, int n)` that prints all shapes in an array.
3. Add a `destroy(Shape *s)` method pointer and implement type-specific cleanup.
4. Extend the `Counter` struct with a `decrement` method.
5. Try designing a small "interface" for `Animal` → `Dog`, `Cat` with a `speak()` function.

With structs and function pointers, C becomes a minimal but powerful **object system**. You now have everything needed to design reusable, modular code, without losing the clarity and efficiency that make C timeless.

Next, you'll finish this chapter by putting all these ideas together: building a small, **real-world system in C**, your own **Tiny Library System**, with data structures, memory management, and modular design.

40. Practice: Build a Tiny Library System

You've now learned every building block, structs, pointers, dynamic memory, linked lists, enums, and even object-style design with function pointers. It's time to combine them into a real mini-project: a **Tiny Library System**. This will be a full, runnable C program that manages books, authors, and borrowing records using everything you've learned so far.

Goal

Implement a minimal system that can:

1. Store book records dynamically
2. Add new books
3. Search books by title
4. Borrow and return books
5. Clean up all memory correctly

We'll use:

- `struct` for data models
- `enum` for status tracking
- linked lists for storage
- `typedef` and function pointers for clarity

Data Structures

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

typedef enum {
    AVAILABLE,
    BORROWED
} BookStatus;

typedef struct Book {
    char *title;
    char *author;
    int year;
    BookStatus status;
    struct Book *next;
} Book;

typedef struct {
    Book *head;
} Library;
```

Each book is one node in a linked list. The library owns the head pointer.

Core Functions

```
Book* create_book(const char *title, const char *author, int year) {
    Book *b = malloc(sizeof(Book));
    b->title = strdup(title);
    b->author = strdup(author);
    b->year = year;
    b->status = AVAILABLE;
    b->next = NULL;
    return b;
}

void add_book(Library *lib, Book *b) {
    b->next = lib->head;
```

```
    lib->head = b;
}

Book* find_book(Library *lib, const char *title) {
    for (Book *cur = lib->head; cur != NULL; cur = cur->next)
        if (strcmp(cur->title, title) == 0)
            return cur;
    return NULL;
}

void borrow_book(Library *lib, const char *title) {
    Book *b = find_book(lib, title);
    if (!b) {
        printf("Book not found: %s\n", title);
        return;
    }
    if (b->status == BORROWED)
        printf("Book already borrowed: %s\n", b->title);
    else {
        b->status = BORROWED;
        printf("You borrowed: %s\n", b->title);
    }
}

void return_book(Library *lib, const char *title) {
    Book *b = find_book(lib, title);
    if (!b) {
        printf("Book not found: %s\n", title);
        return;
    }
    if (b->status == AVAILABLE)
        printf("Book already returned: %s\n", b->title);
    else {
        b->status = AVAILABLE;
        printf("You returned: %s\n", b->title);
    }
}

void list_books(const Library *lib) {
    printf("\n--- Library Catalog ---\n");
    for (const Book *b = lib->head; b != NULL; b = b->next)
        printf("%-20s | %-15s | %d | %s\n",
```

```
               b->title, b->author, b->year,
               b->status == AVAILABLE ? "Available" : "Borrowed");
    printf("------------------------\n\n");
}

void free_library(Library *lib) {
    Book *cur = lib->head;
    while (cur) {
        Book *next = cur->next;
        free(cur->title);
        free(cur->author);
        free(cur);
        cur = next;
    }
    lib->head = NULL;
}
```

Tiny Code, Full Program

```
int main(void) {
    Library lib = {NULL};

    add_book(&lib, create_book("The C Programming Language", "Kernighan & Ritchie", 1988));
    add_book(&lib, create_book("Clean Code", "Robert C. Martin", 2008));
    add_book(&lib, create_book("Algorithms in C", "Sedgewick", 1998));

    list_books(&lib);

    borrow_book(&lib, "Clean Code");
    borrow_book(&lib, "Clean Code");  // test duplicate borrow
    return_book(&lib, "Clean Code");

    borrow_book(&lib, "Algorithms in C");
    list_books(&lib);

    free_library(&lib);
    return 0;
}
```

Compile and run:

```
gcc library_system.c -o library_system
./library_system
```

Output:

```
--- Library Catalog ---
Algorithms in C      | Sedgewick       | 1998 | Available
Clean Code           | Robert C. Martin | 2008 | Available
The C Programming Language | Kernighan & Ritchie | 1988 | Available
------------------------

You borrowed: Clean Code
Book already borrowed: Clean Code
You returned: Clean Code
You borrowed: Algorithms in C

--- Library Catalog ---
Algorithms in C      | Sedgewick       | 1998 | Borrowed
Clean Code           | Robert C. Martin | 2008 | Available
The C Programming Language | Kernighan & Ritchie | 1988 | Available
------------------------
```

What You Just Practiced

You've combined everything from this chapter:

Concept	How You Used It
Structs	For `Book` and `Library` models
Enums	For `BookStatus`
Dynamic memory	`malloc`, `free`, `strdup`
Linked lists	Dynamic collection of books
Pointers	Passing references between functions
Encapsulation	Each function hides internal details

Why It Matters

This "tiny library" is a microcosm of systems programming: you're managing memory, defining abstractions, and building a dynamic system with clear data ownership. From here, you can scale to databases, caches, or in-memory key-value stores, all built on the same principles.

Try It Yourself

1. Add `id` and `genre` fields to `Book`.
2. Implement `remove_book(title)` to delete a node safely.
3. Add a command interface (read from stdin) for interactive use.
4. Save and load the library to a file using `fwrite` and `fread`.
5. Write a function to count how many books are borrowed vs available.

You've completed **Chapter 4: Structuring Data**, the heart of understanding how C organizes the world. Next, you'll move from in-memory structures to **input, output, and files**, learning how to interact with the outside world through the standard I/O library in Chapter 5.

Chapter 5. Input, Output and Files

41. Standard I/O and printf/scanf

Input and output are how your programs talk to the outside world. In C, almost everything goes through the **Standard I/O library**, defined in `<stdio.h>`. You've already met `printf()` in "Hello, C World", now you'll learn how all these functions fit together, how they work, and how to use them safely.

The Standard Streams

Every C program starts with **three standard streams** automatically open:

Stream	Purpose	Example Function
`stdin`	Standard input (keyboard, or redirected file)	`scanf()`, `fgets()`
`stdout`	Standard output (screen)	`printf()`, `puts()`
`stderr`	Standard error (screen, for diagnostics)	`fprintf(stderr, ...)`

You can redirect these streams in the terminal:

```
./program < input.txt > output.txt 2> errors.log
```

Printing with printf()

`printf()` formats and prints data to `stdout`. Its power lies in **format specifiers**, which describe the type and layout of what to print.

Type	Format	Example
int	`%d`	`printf("%d", 42);`
float	`%f`	`printf("%.2f", 3.1415);`
char	`%c`	`printf("%c", 'A');`
string	`%s`	`printf("%s", "Hello");`
pointer	`%p`	`printf("%p", ptr);`

Type	Format	Example
hexadecimal	%x	printf("%x", 255);

You can control width, precision, and alignment:

```
printf("%-10s | %6.2f\n", "Price", 3.5);
```

Output:

```
Price      |   3.50
```

Tiny Code: Print Everything

```
#include <stdio.h>

int main(void) {
    int i = 42;
    float f = 3.1415;
    char c = 'C';
    char *s = "Hello, C!";

    printf("Integer: %d\n", i);
    printf("Float: %.2f\n", f);
    printf("Char: %c\n", c);
    printf("String: %s\n", s);
    printf("Pointer: %p\n", (void*)s);
    return 0;
}
```

Output:

```
Integer: 42
Float: 3.14
Char: C
String: Hello, C!
Pointer: 0x7ffeed001234
```

Reading with scanf()

`scanf()` reads formatted input from `stdin`. It's like `printf()` in reverse, you tell it the format, and it fills your variables.

```c
int age;
float height;
printf("Enter age and height: ");
scanf("%d %f", &age, &height);
printf("You are %d years old and %.1f meters tall.\n", age, height);
```

Input:

```
25 1.75
```

Output:

```
You are 25 years old and 1.8 meters tall.
```

> **Always use the `&` operator** for non-array variables — it passes the memory address where the value should be stored.

Safer Input: fgets() and sscanf()

`scanf()` is risky for strings, it doesn't prevent buffer overflow. Safer pattern: use `fgets()` to read a full line, then parse it.

```c
char buf[100];
printf("Enter your name: ");
fgets(buf, sizeof(buf), stdin);
buf[strcspn(buf, "\n")] = '\0'; // remove newline
printf("Hello, %s!\n", buf);
```

Combining printf and scanf

You can build interactive console tools easily:

```c
#include <stdio.h>

int main(void) {
    char name[50];
    int year;

    printf("Enter your name: ");
    scanf("%49s", name); // limit input to 49 chars + null
    printf("Enter your birth year: ");
    scanf("%d", &year);

    printf("Hi %s! You are about %d years old.\n", name, 2025 - year);
    return 0;
}
```

Output:

```
Enter your name: Alice
Enter your birth year: 2000
Hi Alice! You are about 25 years old.
```

Formatted Output to Files and Strings

You can redirect formatted output anywhere, not just the screen.

```c
fprintf(stderr, "Error: invalid input.\n"); // print to stderr

char buffer[50];
sprintf(buffer, "Pi = %.3f", 3.14159); // print to string
puts(buffer);
```

Output:

```
Pi = 3.142
```

Why It Matters

`printf()` and `scanf()` are the **workhorses of console I/O**. They teach you:

- How data moves between memory and streams.
- How to control numeric precision and layout.
- How input and output interact with the terminal or files.

Every C system, from tiny microcontrollers to full operating systems, uses these same foundations.

Try It Yourself

1. Print a table of numbers with two columns: number and its square.
2. Read three integers using `scanf` and print their average.
3. Use `fgets` and `sscanf` to safely parse `"42 3.14"` into int and float.
4. Write a small quiz app: ask a question, read input, print "Correct" or "Try again".
5. Experiment with printing to `stderr`, redirect errors to a file.

Mastering standard I/O is like mastering your program's **voice**, it's how your C code speaks and listens. Next, you'll move deeper into file handling, learning how to open, read, and write files with file pointers in Section 42.

42. File Pointers and `fopen` / `fclose`

Files let your C programs remember things beyond runtime. Unlike standard input and output, which disappear when the program ends, files provide **persistent storage**, you can read and write data between runs.

This section introduces the key file-handling API in C: `fopen()`, `fclose()`, `fprintf()`, `fscanf()`, and their relatives.

The Big Picture

File I/O in C uses a `FILE *` pointer to represent an open file. You don't manipulate the disk directly, instead, you read and write through a buffered file stream managed by the runtime.

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

This returns a pointer to a `FILE` object if successful, or `NULL` if the file can't be opened.

File Modes

When opening a file, you specify a **mode**, what you intend to do with it.

Mode	Meaning	Behavior
`"r"`	read	Fails if file doesn't exist
`"w"`	write	Creates new or truncates existing
`"a"`	append	Opens or creates; writes go to end
`"r+"`	read/write	Must exist
`"w+"`	read/write	Truncates if exists
`"a+"`	read/write	Appends; reading starts at beginning

Tiny Code: Write and Read a File

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("example.txt", "w");
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    fprintf(fp, "Hello, file world!\n");
    fprintf(fp, "C makes you closer to the machine.\n");
    fclose(fp);

    fp = fopen("example.txt", "r");
    if (!fp) {
        perror("Failed to reopen file");
        return 1;
    }

    char line[100];
    printf("--- File Content ---\n");
    while (fgets(line, sizeof(line), fp))
        printf("%s", line);

    fclose(fp);
```

```
    return 0;
}
```

Output:

```
--- File Content ---
Hello, file world!
C makes you closer to the machine.
```

How It Works

1. `fopen()` creates a connection to a file.
2. `fprintf()` writes formatted text, just like `printf()` but to a file stream.
3. `fclose()` flushes buffers and closes the file.
4. Reopen with `"r"` mode to read what you wrote.

You can also mix with `fscanf()` to read formatted data.

Checking for Errors

Always check file operations for errors. Use `if (!fp)` after `fopen()`, and use `perror()` to print the reason.

```
FILE *f = fopen("missing.txt", "r");
if (!f) {
    perror("Error opening file");
}
```

Example output:

```
Error opening file: No such file or directory
```

Reading Formatted Data with `fscanf`

You can parse text files using `fscanf()`, just like `scanf()`:

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("numbers.txt", "r");
    if (!fp) return 1;

    int a, b;
    while (fscanf(fp, "%d %d", &a, &b) == 2)
        printf("%d + %d = %d\n", a, b, a + b);

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

If `numbers.txt` contains:

```
2 3
10 15
7 9
```

Output:

```
2 + 3 = 5
10 + 15 = 25
7 + 9 = 16
```

File Position and Rewinding

You can move around a file using:

```c
fseek(fp, 0, SEEK_SET);  // go to beginning
fseek(fp, 0, SEEK_END);  // go to end
rewind(fp);              // reset to start
```

To know where you are:

```c
long pos = ftell(fp);
printf("Current position: %ld\n", pos);
```

Writing Binary Data

Text files are human-readable; binary files store raw bytes. You'll use `fwrite()` and `fread()` for that (covered more in the next section).

Example:

```
int numbers[] = {1, 2, 3, 4};
fwrite(numbers, sizeof(int), 4, fp);
```

Why It Matters

File I/O is the bridge between your C program and the real world:

- Configuration and log files
- Database storage
- Caches and serialization
- Operating system utilities (copy, move, grep, etc.)

It teaches **resource management**, always `fopen()` and `fclose()` in pairs, check errors, and handle failures gracefully.

Try It Yourself

1. Write a program that asks for your name and saves it to `user.txt`.
2. Append a timestamp each time the program runs.
3. Read all lines and count how many times your program has been executed.
4. Modify the example to reverse all lines read from a file.
5. Handle missing files gracefully using `perror()`.

You now know how to open, read, and write text files safely. Next, you'll go deeper into **binary files**, where data moves in raw bytes, perfect for storing structs and arrays efficiently.

43. Reading and Writing Binary Files

Text files are easy to read but not always efficient. Binary files, on the other hand, store raw bytes exactly as they exist in memory, no formatting, no conversions. They're ideal for saving arrays, structs, or any data that must be written and read back quickly without loss or rounding.

Text vs Binary

Aspect	Text File	Binary File
Format	Human-readable (ASCII)	Raw bytes
Size	Larger (extra characters, newlines)	Smaller (compact form)
Read/Write	`fprintf`, `fscanf`, `fgets`	`fwrite`, `fread`
Use Case	Logs, config, reports	Structs, images, executables, serialized data

When you open a file for binary I/O, add `b` to the mode:

```
FILE *fp = fopen("data.bin", "wb"); // write binary
FILE *fp = fopen("data.bin", "rb"); // read binary
```

Writing Binary Data

Let's write an array of integers directly to disk.

```
#include <stdio.h>

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

    FILE *fp = fopen("numbers.bin", "wb");
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    fwrite(numbers, sizeof(int), count, fp);
    fclose(fp);

    printf("Wrote %zu integers to numbers.bin\n", count);
    return 0;
}
```

This writes 5 integers (4 bytes each on most systems) directly to disk as raw bytes, no text conversion.

Reading Binary Data

Now let's read them back:

```c
#include <stdio.h>

int main(void) {
    int numbers[5];
    FILE *fp = fopen("numbers.bin", "rb");
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    size_t n = fread(numbers, sizeof(int), 5, fp);
    fclose(fp);

    printf("Read %zu integers:\n", n);
    for (size_t i = 0; i < n; i++)
        printf("%d ", numbers[i]);
    printf("\n");

    return 0;
}
```

Output:

```
Wrote 5 integers to numbers.bin
Read 5 integers:
10 20 30 40 50
```

Writing and Reading Structs

You can store whole structures directly using the same pattern.

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

typedef struct {
    int id;
    float price;
```

```
    char title[50];
} Book;

int main(void) {
    Book b1 = {1, 9.99, "The C Book"};
    Book b2 = {2, 15.49, "Algorithms in C"};

    FILE *fp = fopen("books.bin", "wb");
    if (!fp) return 1;
    fwrite(&b1, sizeof(Book), 1, fp);
    fwrite(&b2, sizeof(Book), 1, fp);
    fclose(fp);

    fp = fopen("books.bin", "rb");
    if (!fp) return 1;
    Book b;
    while (fread(&b, sizeof(Book), 1, fp) == 1)
        printf("%d | %s | %.2f\n", b.id, b.title, b.price);
    fclose(fp);
    return 0;
}
```

Output:

```
1 | The C Book | 9.99
2 | Algorithms in C | 15.49
```

Handling Endianness

Binary files depend on the CPU's byte order (endianness). If you write on a **little-endian** machine and read on a **big-endian** one, bytes may appear reversed.

For portable formats, you can:

- Use standardized serialization (like Protocol Buffers or MessagePack).
- Convert manually using bit shifts or network-byte-order functions (`htonl`, `ntohl`).

Example for manual conversion:

```c
unsigned int to_big_endian(unsigned int x) {
    return ((x & 0xFF) << 24) |
           ((x & 0xFF00) << 8) |
           ((x & 0xFF0000) >> 8) |
           ((x >> 24) & 0xFF);
}
```

Appending Binary Data

You can append more records with mode `"ab"`:

```c
Book b3 = {3, 21.75, "Advanced C"};
FILE *fp = fopen("books.bin", "ab");
fwrite(&b3, sizeof(Book), 1, fp);
fclose(fp);
```

Binary File Utilities

Function	Purpose
`fwrite(ptr, size, count, file)`	Write binary data
`fread(ptr, size, count, file)`	Read binary data
`fseek(file, offset, origin)`	Move position
`ftell(file)`	Get current position
`rewind(file)`	Go back to start

Why It Matters

Binary I/O is essential for:

- Saving large datasets efficiently
- Game save files, multimedia formats, or scientific data
- Databases and memory-mapped storage
- Embedded and system-level tools

It's the foundation of **serialization**, transforming data in memory into bytes that can travel or persist.

Try It Yourself

1. Save an array of `double` values and read them back.
2. Modify the struct example to include an `enum` field and test the binary result.
3. Implement a function `count_records(filename)` that counts how many structs are stored.
4. Use `fseek()` to jump to the third record and print only that one.
5. Write both text and binary versions of the same file and compare sizes.

Binary I/O connects C's low-level power to real-world storage efficiency. Next, you'll expand this further by understanding **standard streams**, how to use `stdin`, `stdout`, and `stderr` to build flexible, composable command-line tools.

44. Working with `stdin`, `stdout`, and `stderr`

Every C program automatically starts with three open streams connected to your environment, the keyboard, the terminal screen, and the error console. They are the **standard I/O streams** that make your programs flexible and scriptable.

Understanding these three streams is crucial for writing tools that can interact with files, pipes, and other programs, the essence of Unix-style design.

The Three Standard Streams

Stream	Purpose	Typical Device	Example Use
`stdin`	Standard Input	Keyboard (or file via <)	`scanf`, `fgets`
`stdout`	Standard Output	Screen (or file via >)	`printf`, `puts`, `fprintf(stdout, …)`
`stderr`	Standard Error	Screen (separate from stdout)	`fprintf(stderr, …)`

These are all of type `FILE *`. You can treat them like normal file pointers, reading, writing, or redirecting them.

Basic Example

```c
#include <stdio.h>

int main(void) {
    char name[50];
    printf("Enter your name: ");
    fgets(name, sizeof(name), stdin);

    fprintf(stdout, "Hello, %s", name);
    fprintf(stderr, "Note: This is an example error message.\n");
    return 0;
}
```

Output:

```
Enter your name: Alice
Hello, Alice
Note: This is an example error message.
```

Redirection in the Shell

You can redirect each stream separately:

```
./program < input.txt > output.txt 2> errors.log
```

- `<` replaces `stdin` (read input from file)
- `>` replaces `stdout` (write normal output to file)
- `2>` replaces `stderr` (write errors to file)

You can also combine them:

```
./program > all_output.txt 2>&1
```

This merges both output and error streams into one file.

Reading from stdin

You can build programs that process input dynamically, one line at a time:

```c
#include <stdio.h>

int main(void) {
    char line[100];
    printf("Enter text (Ctrl+D to stop):\n");
    while (fgets(line, sizeof(line), stdin))
        printf("You said: %s", line);
    return 0;
}
```

Now your program behaves like a Unix filter, it can read from a file, a pipe, or a keyboard input interchangeably.

Example:

```
echo "hello" | ./program
```

Output:

```
You said: hello
```

Writing to stdout and stderr

`stdout` is for normal program output, while `stderr` is for error messages or logs.

```c
#include <stdio.h>

int main(void) {
    fprintf(stdout, "Everything is fine.\n");
    fprintf(stderr, "Warning: something might be wrong.\n");
    return 0;
}
```

You can suppress normal output but keep errors:

```
./program > /dev/null
```

Output:

```
Warning: something might be wrong.
```

Tiny Code: Word Counter Using stdin/stdout

This small program mimics a simplified version of the Unix `wc` command.

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

int main(void) {
    int ch, words = 0, in_word = 0;

    while ((ch = getchar()) != EOF) {
        if (isspace(ch))
            in_word = 0;
        else if (!in_word) {
            in_word = 1;
            words++;
        }
    }

    printf("Word count: %d\n", words);
    return 0;
}
```

Try:

```
echo "C is small but powerful" | ./wordcount
```

Output:

```
Word count: 4
```

Mixing stdout and stderr

Sometimes you need to log progress to `stderr` while outputting results to `stdout`. That way, logs don't pollute the actual data.

```c
#include <stdio.h>

int main(void) {
    for (int i = 0; i < 3; i++) {
        fprintf(stderr, "Processing item %d...\n", i + 1);
```

```
        fprintf(stdout, "Item %d processed\n", i + 1);
    }
    return 0;
}
```

Redirect logs separately:

```
./program > result.txt 2> log.txt
```

Flushing Buffers

Output streams are **buffered**, data isn't written until the buffer is full or flushed. To ensure output appears immediately:

```
fflush(stdout);  // flush output buffer
```

This is useful for interactive programs.

Why It Matters

These three streams give your program flexibility:

- Work interactively (keyboard/screen)
- Work in batch (file input/output)
- Chain with other tools using pipes

They are the foundation of the **Unix philosophy**: small programs that do one thing well and can be composed together.

Try It Yourself

1. Write a program that reads from stdin and prints only lines containing a given keyword.
2. Print errors to stderr if no keyword is provided.
3. Redirect input and output from files using `<` and `>`.
4. Add progress messages to stderr and redirect them to a separate log.
5. Combine everything into a small "filter" tool that processes text from pipelines.

With `stdin`, `stdout`, and `stderr`, your C programs become tools that fit seamlessly into real workflows, able to interact with files, other programs, and users alike. Next, you'll explore **buffered I/O**, understanding how the C library optimizes performance through read and write buffers using `fgets`, `fputs`, and more.

45. Buffered I/O with `fgets` and `fputs`

When your program reads and writes data, it doesn't always go directly to disk or the terminal, instead, it uses **buffers**. Buffers are small chunks of memory that temporarily hold data, improving performance by reducing how often the system has to perform slow I/O operations.

C's Standard I/O library (`<stdio.h>`) handles this automatically for you. In this section, you'll learn how buffering works and how to use `fgets`, `fputs`, and related functions to manage it effectively.

What Is Buffered I/O?

Instead of reading or writing one character at a time, the C library:

- Fills a buffer (for input) or
- Collects a batch of characters (for output)

When the buffer is full or flushed, data moves between your program and the file or terminal.

This is why sometimes `printf()` output doesn't appear immediately, it's waiting in a buffer until a newline or flush occurs.

Input: `fgets()`

`fgets()` reads a full line from a stream (including spaces) and stores it in a string.

```
char *fgets(char *str, int size, FILE *stream);
```

- `str`: where to store the line
- `size`: maximum number of characters to read (including `\0`)
- `stream`: where to read from (e.g. `stdin` or a file pointer)

Example:

```
#include <stdio.h>

int main(void) {
    char line[100];
    printf("Enter a sentence: ");
    if (fgets(line, sizeof(line), stdin))
        printf("You said: %s", line);
    return 0;
}
```

Input:

```
C is beautiful.
```

Output:

```
You said: C is beautiful.
```

If the input exceeds the buffer, `fgets` stops reading after `size - 1` characters to prevent overflow, and automatically null-terminates the string.

Output: `fputs()`

`fputs()` writes a string to a stream.

```c
int fputs(const char *str, FILE *stream);
```

Example:

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("output.txt", "w");
    if (!fp) {
        perror("Open failed");
        return 1;
    }

    fputs("Buffered I/O makes C fast.\n", fp);
    fputs("fgets and fputs are line-based tools.\n", fp);

    fclose(fp);
    printf("Wrote to output.txt\n");
    return 0;
}
```

Output file:

```
Buffered I/O makes C fast.
fgets and fputs are line-based tools.
```

Why `fgets` Is Safer Than `scanf("%s", …)`

- `fgets()` respects buffer boundaries
- It reads spaces and tabs correctly
- It prevents undefined behavior from overflows

Avoid this:

```
scanf("%s", buffer); // stops at first space and may overflow
```

Prefer this:

```
fgets(buffer, sizeof(buffer), stdin);
```

Writing to `stdout` or `stderr` with `fputs`

`fputs()` works on any output stream, not just files.

```
fputs("Message to screen.\n", stdout);
fputs("Error to stderr!\n", stderr);
```

You can even redirect these in the shell:

```
./program > result.txt 2> error.log
```

Reading Files Line by Line

Here's how to read a file safely using `fgets()`:

```
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("poem.txt", "r");
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    char line[200];
    while (fgets(line, sizeof(line), fp))
```

```
        printf("%s", line);

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

Tiny Code: Copy a File

A minimal file copier using `fgets` and `fputs`:

```
#include <stdio.h>

int main(void) {
    FILE *in = fopen("input.txt", "r");
    FILE *out = fopen("copy.txt", "w");

    if (!in || !out) {
        perror("File error");
        return 1;
    }

    char buf[256];
    while (fgets(buf, sizeof(buf), in))
        fputs(buf, out);

    fclose(in);
    fclose(out);
    printf("Copied successfully.\n");
    return 0;
}
```

Buffer Flushing

Sometimes you need to manually flush output:

```
fflush(stdout);
```

This is useful for interactive programs that must show messages immediately.

To disable buffering entirely (e.g., for logging):

```c
setbuf(stdout, NULL);
```

Why It Matters

Buffered I/O balances **speed** and **safety**:

- `fgets` protects against overflow
- `fputs` ensures efficient output
- Buffering minimizes slow disk and console calls

It's what makes C both low-level and performant without forcing you to manage every byte yourself.

Try It Yourself

1. Write a program that reads lines from stdin and writes them to a new file.
2. Count how many lines you read before EOF.
3. Print each line with line numbers using `fgets` and `printf`.
4. Experiment with buffer sizes, try 16 vs 256 bytes and note the performance difference.
5. Flush output after every line for an interactive logging program.

`fgets` and `fputs` give you a safe, line-based foundation for file and console I/O. Next, you'll learn how to **handle errors correctly** using `errno`, `perror`, and `strerror`, essential tools for writing reliable system programs in C.

46. Error Checking with `errno` and `perror`

Even the best-written C programs can encounter errors, missing files, permission issues, division by zero, or failed memory allocations. Unlike some languages that throw exceptions, C reports errors **manually** using return values and a global variable named `errno`.

Understanding how to use `errno`, `perror()`, and `strerror()` is essential for writing **robust, production-grade** C programs that fail gracefully and informatively.

The Idea Behind `errno`

`errno` is a global integer (declared in `<errno.h>`) that stores an **error code** whenever a library function fails.

- On success, most functions leave `errno` unchanged.
- On failure, they typically set `errno` and return an error value (often `NULL`, `-1`, or `0` depending on the function).
- You must check the return value first, only then should you inspect `errno`.

Example:

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

int main(void) {
    FILE *fp = fopen("missing.txt", "r");
    if (!fp) {
        printf("Error code: %d\n", errno);
        perror("fopen failed");
    }
    return 0;
}
```

Output:

```
Error code: 2
fopen failed: No such file or directory
```

Common `errno` Codes

Code	Name	Meaning
2	`ENOENT`	No such file or directory
13	`EACCES`	Permission denied
12	`ENOMEM`	Not enough memory
22	`EINVAL`	Invalid argument
17	`EEXIST`	File already exists
5	`EIO`	Input/output error
111	`ECONNREFUSED`	Connection refused (network)

Include `<errno.h>` to use these symbolic names:

```c
if (errno == ENOENT) printf("File missing.\n");
```

Using `perror()`

`perror()` prints a human-readable error message to `stderr`, based on the current value of `errno`.

```c
perror("File open error");
```

Example:

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

int main(void) {
    FILE *f = fopen("ghost.txt", "r");
    if (!f)
        perror("Unable to open file");
    return 0;
}
```

Output:

```
Unable to open file: No such file or directory
```

Using `strerror()`

If you want to use the error message in your own formatted output, use `strerror()` from `<string.h>`:

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

int main(void) {
    FILE *f = fopen("/root/secret.txt", "r");
    if (!f)
        printf("Error (%d): %s\n", errno, strerror(errno));
    return 0;
}
```

Output:

```
Error (13): Permission denied
```

Tiny Code: Robust File Reader

Here's a simple file reader that checks for errors at every step:

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    FILE *fp = fopen(argv[1], "r");
    if (!fp) {
        fprintf(stderr, "Error opening %s: %s\n", argv[1], strerror(errno));
        return 1;
    }

    char buf[128];
    while (fgets(buf, sizeof(buf), fp))
        printf("%s", buf);

    if (ferror(fp)) {
        fprintf(stderr, "Error reading file: %s\n", strerror(errno));
    }

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

Usage:

```
./readfile poem.txt
```

If the file is missing:

```
Error opening poem.txt: No such file or directory
```

Clearing and Resetting `errno`

Some functions may set `errno` even if they succeed later. To be safe, you can clear it before a call:

```
#include <errno.h>

errno = 0;
FILE *f = fopen("file.txt", "r");
if (!f) perror("fopen");
```

This ensures you don't read a leftover error from an earlier operation.

Checking Other System Errors

`errno` isn't limited to file I/O, it applies to many system calls and library functions:

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

int main(void) {
    FILE *f = fopen("/dev/full", "w"); // special Linux device that fails on write
    if (f) {
        if (fputc('A', f) == EOF)
            perror("Write failed");
        fclose(f);
    }
    return 0;
}
```

Output:

```
Write failed: No space left on device
```

Why It Matters

`errno` and its helpers (`perror`, `strerror`) make your programs *explain themselves* when things go wrong. This is vital for:

- System tools that must report specific error causes
- Debugging production code
- Writing portable, maintainable programs

Good C developers never just "fail silently."

Try It Yourself

1. Open a file that doesn't exist, then create it and try again.
2. Simulate a read error by using `ferror()` after reading a closed file.
3. Try writing to a directory (`fopen("/tmp", "w")`) and inspect `errno`.
4. Print all known error codes and messages using a loop and `strerror()`.
5. Write a small "safe_open" function that wraps `fopen` with error reporting.

With error handling mastered, you now know how to make C programs both informative and reliable. Next, you'll explore **command-line arguments (`argc`, `argv`)**, the gateway to building flexible, scriptable tools that process user input dynamically.

47. Command-Line Arguments (`argc`, `argv`)

Every C program can receive input directly from the **command line**, no `scanf`, no `fgets`, just arguments passed when you run the executable. This is how professional C tools (like `gcc`, `ls`, and `grep`) receive filenames, options, and flags.

The Function Signature

Your `main` function can take **two parameters**:

```c
int main(int argc, char *argv[])
```

Parameter	Meaning
`argc`	Argument count (number of command-line arguments)
`argv`	Argument vector (array of C strings, each argument)

`argv[0]` is the program name itself, and `argv[1]` onward are the user-provided arguments.

Example:

```
./hello world test
```

Then:

- `argc == 3`
- `argv[0] = "./hello"`
- `argv[1] = "world"`
- `argv[2] = "test"`

Tiny Code: Print Command-Line Arguments

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Argument count: %d\n", argc);
    for (int i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

Run it:

```
./args foo bar 123
```

Output:

```
Argument count: 4
argv[0] = ./args
argv[1] = foo
argv[2] = bar
argv[3] = 123
```

Checking for Missing Arguments

If your program needs arguments, check `argc` before accessing them.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }
    printf("Opening file: %s\n", argv[1]);
    return 0;
}
```

Run it:

```
./fileop
```

Output:

```
Usage: ./fileop <filename>
```

Run again:

```
./fileop data.txt
```

Output:

```
Opening file: data.txt
```

Converting String Arguments to Numbers

All command-line arguments are **strings**. To use them as numbers, convert using:

- `atoi()` – string to int
- `atof()` – string to float
- `strtol()` / `strtod()` – safer and more flexible alternatives

Example:

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "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 10 25
```

Output:

```
10 + 25 = 35
```

Handling Options (Flags)

You can build simple command-line tools that handle options manually:

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

int main(int argc, char *argv[]) {
    int verbose = 0;

    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-v") == 0)
            verbose = 1;
    }

    if (verbose)
        printf("Verbose mode on\n");
```

```
    else
        printf("Run quietly\n");

    return 0;
}
```

Run:

```
./tool -v
```

Output:

```
Verbose mode on
```

Tiny Code: Mini File Echo Tool

This program prints the content of a file passed as an argument:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return 1;
    }

    FILE *fp = fopen(argv[1], "r");
    if (!fp) {
        perror("Error");
        return 1;
    }

    char buf[128];
    while (fgets(buf, sizeof(buf), fp))
        printf("%s", buf);

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

Run:

```
./echo mytext.txt
```

Why It Matters

`argc` and `argv` make your C programs scriptable and composable:

- Automate tasks from the command line
- Integrate with shell scripts or pipelines
- Process multiple input files
- Implement command-line flags and options

Every real-world C utility, from `ls` to `gcc`, depends on this pattern.

Try It Yourself

1. Write a program that takes a list of integers and prints their sum.
2. Add a `-r` flag to reverse the order of printed arguments.
3. Build a "greet" tool:

   ```
   ./greet Alice Bob Charlie
   ```

   → `Hello, Alice! Hello, Bob! Hello, Charlie!`
4. Write a "compare" tool that checks if two file names are identical.
5. Combine `argc` and file I/O: copy one file to another with

   ```
   ./copy source.txt dest.txt
   ```

With command-line arguments, your C programs evolve from static exercises to **flexible, real-world tools**. Next, you'll explore **reading configuration files**, a powerful way to let your programs adapt automatically without recompilation.

48. Reading Configuration Files

As your C programs grow, hardcoding settings like file paths, thresholds, or user preferences becomes limiting. Configuration files let your program read settings at runtime, a critical capability for tools, servers, and embedded systems.

You'll learn how to read and parse configuration files using standard I/O and string handling.

The Goal

A configuration file might look like this:

```
port=8080
host=localhost
max_clients=100
log_file=server.log
```

Your program should:

1. Open the file
2. Read it line by line
3. Split each line into `key` and `value`
4. Store or use those values

Step 1. Define a Structure for Config

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

#define MAX_LINE 128

typedef struct {
    int port;
    char host[64];
    int max_clients;
    char log_file[64];
} Config;
```

This `Config` struct will hold parsed values.

Step 2. Implement the Parser

```c
void load_config(const char *filename, Config *cfg) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        perror("Cannot open config file");
        exit(1);
    }

    char line[MAX_LINE];
    while (fgets(line, sizeof(line), fp)) {
        line[strcspn(line, "\n")] = '\0'; // remove newline

        if (line[0] == '#' || strlen(line) == 0)
            continue; // skip comments and blanks

        char key[64], value[64];
        if (sscanf(line, "%63[^=]=%63s", key, value) == 2) {
            if (strcmp(key, "port") == 0)
                cfg->port = atoi(value);
            else if (strcmp(key, "host") == 0)
                strncpy(cfg->host, value, sizeof(cfg->host));
            else if (strcmp(key, "max_clients") == 0)
                cfg->max_clients = atoi(value);
            else if (strcmp(key, "log_file") == 0)
                strncpy(cfg->log_file, value, sizeof(cfg->log_file));
        }
    }

    fclose(fp);
}
```

This function:

- Reads each line
- Ignores comments and empty lines
- Extracts key-value pairs using `sscanf()`
- Updates fields in `Config`

Step 3. Use the Configuration

```c
int main(void) {
    Config cfg = {0};
    load_config("config.txt", &cfg);

    printf("Server settings:\n");
    printf("Host: %s\n", cfg.host);
    printf("Port: %d\n", cfg.port);
    printf("Max clients: %d\n", cfg.max_clients);
    printf("Log file: %s\n", cfg.log_file);

    return 0;
}
```

Run with a `config.txt` file:

```
host=127.0.0.1
port=9090
max_clients=250
log_file=/tmp/server.log
```

Output:

```
Server settings:
Host: 127.0.0.1
Port: 9090
Max clients: 250
Log file: /tmp/server.log
```

Tiny Code: Default Fallbacks

You can initialize sensible defaults before reading the file:

```c
Config cfg = {
    .port = 8080,
    .host = "localhost",
    .max_clients = 100,
    .log_file = "server.log"
};
```

This ensures your program still works even if the file is missing some values.

Step 4. Optional: Handle Quoted Values

If you expect values with spaces (like `name="My Server"`), you can modify parsing logic:

```
if (sscanf(line, "%63[^=]=\"%63[^\"]\"", key, value) == 2) {
    // handle quoted strings
}
```

Step 5. Optional: Generic Storage

For more flexible systems, you can use a hash table or array of key-value pairs instead of fixed fields:

```
typedef struct {
    char key[64];
    char value[64];
} KVPair;

KVPair settings[100];
```

This allows loading arbitrary keys without recompiling the program.

Why It Matters

Configuration files let you:

- **Separate code from data**, no need to recompile to change behavior
- **Adapt to environments**, dev, test, production
- **Make your program reusable** by others

They're used everywhere, from `.ini` and `.conf` files to complex YAML/JSON formats in modern systems.

Try It Yourself

1. Add support for `#` comments and empty lines (skip them safely).
2. Make the parser print a warning for unknown keys.
3. Add a function `save_config()` that writes the struct back to a file.
4. Add `reload_config()` to update settings at runtime.
5. Implement your own `.ini` format parser supporting `[section]` headers.

With configuration files, your C programs gain flexibility and real-world usability, they can adapt, reload, and persist settings just like professional systems software. Next, you'll learn how to **serialize and deserialize structs to disk**, the next level of persistent data handling in Section 49.

49. Serializing Structs to Disk

So far, you've worked with text files, configuration files, and basic binary data. Now it's time to combine those ideas into something more powerful, **serialization**: saving complete C structs to disk and restoring them later, exactly as they were in memory.

This is the foundation for databases, caches, and persistent state in operating systems and games.

What Is Serialization?

Serialization means converting in-memory data into a format that can be stored or transmitted (like a file). **Deserialization** is the reverse: reconstructing that data from the file.

In C, this often means writing structs directly as binary data with `fwrite()` and reading them back with `fread()`.

Step 1. Define a Struct to Store

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

typedef struct {
    int id;
    char name[50];
    float price;
} Product;
```

Each field is fixed-size, which makes it safe to write directly to disk as binary.

Step 2. Write Structs to Disk

```c
void save_products(const char *filename, Product *arr, size_t count) {
    FILE *fp = fopen(filename, "wb");
    if (!fp) {
        perror("Cannot open file for writing");
        exit(1);
    }

    fwrite(arr, sizeof(Product), count, fp);
    fclose(fp);
}
```

Step 3. Read Structs from Disk

```c
size_t load_products(const char *filename, Product *arr, size_t max_count) {
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        perror("Cannot open file for reading");
        return 0;
    }

    size_t n = fread(arr, sizeof(Product), max_count, fp);
    fclose(fp);
    return n;
}
```

Tiny Code: Complete Example

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

typedef struct {
    int id;
    char name[50];
    float price;
} Product;

void save_products(const char *filename, Product *arr, size_t count) {
```

```c
    FILE *fp = fopen(filename, "wb");
    if (!fp) {
        perror("Cannot open file");
        exit(1);
    }
    fwrite(arr, sizeof(Product), count, fp);
    fclose(fp);
}

size_t load_products(const char *filename, Product *arr, size_t max_count) {
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        perror("Cannot open file");
        return 0;
    }
    size_t n = fread(arr, sizeof(Product), max_count, fp);
    fclose(fp);
    return n;
}

int main(void) {
    Product products[3] = {
        {1, "Notebook", 2.99},
        {2, "Pencil", 0.49},
        {3, "Backpack", 25.00}
    };

    save_products("store.bin", products, 3);
    printf("Products saved.\n");

    Product loaded[3];
    size_t n = load_products("store.bin", loaded, 3);
    printf("Loaded %zu products:\n", n);

    for (size_t i = 0; i < n; i++)
        printf("%d | %-10s | $%.2f\n", loaded[i].id, loaded[i].name, loaded[i].price);

    return 0;
}
```

Output:

```
Products saved.
Loaded 3 products:
1 | Notebook   | $2.99
2 | Pencil     | $0.49
3 | Backpack   | $25.00
```

Step 4. Appending Records

You can add more data without overwriting by using append mode `"ab"`:

```
Product p = {4, "Eraser", 0.99};
FILE *fp = fopen("store.bin", "ab");
fwrite(&p, sizeof(Product), 1, fp);
fclose(fp);
```

Step 5. Random Access to Records

You can use `fseek()` to jump to a specific record (useful for updating or reading one record at a time).

```
FILE *fp = fopen("store.bin", "rb");
fseek(fp, sizeof(Product) * 1, SEEK_SET);  // skip first record
Product p;
fread(&p, sizeof(Product), 1, fp);
printf("Record 2: %s\n", p.name);
fclose(fp);
```

Step 6. Portability Considerations

Serialization like this is **machine-dependent** because of:

- **Endianness** (byte order of integers/floats)
- **Structure padding** (compiler alignment)
- **Data type sizes**

To make it portable:

- Use `#pragma pack(1)` or `__attribute__((packed))` to disable padding.
- Convert integers to a standard byte order (e.g., use `htonl()` and `ntohl()`).
- Consider text-based or portable formats like CSV, JSON, or protobuf for cross-platform storage.

Step 7. Text-Based Alternative (Human-Readable)

```c
void save_as_text(const char *filename, Product *arr, size_t count) {
    FILE *fp = fopen(filename, "w");
    if (!fp) return;
    for (size_t i = 0; i < count; i++)
        fprintf(fp, "%d,%s,%.2f\n", arr[i].id, arr[i].name, arr[i].price);
    fclose(fp);
}
```

This produces:

```
1,Notebook,2.99
2,Pencil,0.49
3,Backpack,25.00
```

Easy to read, but slower to parse and less space-efficient.

Why It Matters

Serialization makes your C programs *stateful*, they can save progress, store data, or recover after restarts. It's the basis for:

- Databases and key-value stores
- Save files in games
- Checkpointing in scientific software
- System daemons and caches

You're now handling **real persistence** in C.

Try It Yourself

1. Add a function `add_product()` that appends new records safely.
2. Implement `list_products()` that prints all products from file.
3. Add a "delete by id" operation by copying all but one record to a new file.
4. Experiment with structure padding (`sizeof(Product)` may not be what you expect).
5. Add a checksum field to detect corrupted data.

You now know how to persist structured data in binary or text form. Next, you'll close Chapter 5 by combining all this knowledge, writing a **log reader and writer** system that records events, rotates files, and safely replays logs on startup.

50. Practice: Build a Log Reader and Writer

You've explored text and binary I/O, buffering, error handling, and configuration. Now it's time to bring everything together in one real-world practice project, a **Log Reader and Writer** in C.

This system will let you write structured logs to a file and later read them back, a foundation for tools like servers, daemons, and debugging utilities.

Project Overview

You'll build a minimal logging system with two main parts:

1. **Logger (Writer):**
    - Appends log messages to a file with timestamps and levels (INFO, WARN, ERROR).
    - Handles file opening, writing, and safe closure.
2. **Reader:**
    - Reads log entries line by line.
    - Filters by log level or keyword.

This project teaches structured file I/O, formatted output, parsing, and simple text search, all in clean C.

Step 1. Define the Log Format

A log line will look like this:

```
[2025-10-15 21:00:32] [INFO] Server started
[2025-10-15 21:01:05] [WARN] High CPU usage
[2025-10-15 21:02:10] [ERROR] Connection failed
```

Each entry includes:

- Timestamp
- Level (INFO/WARN/ERROR)
- Message

Step 2. Implement the Logger

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdarg.h>
#include <string.h>

typedef enum {
    INFO,
    WARN,
    ERROR
} LogLevel;

const char* level_to_string(LogLevel level) {
    switch (level) {
        case INFO: return "INFO";
        case WARN: return "WARN";
        case ERROR: return "ERROR";
        default: return "UNKNOWN";
    }
}

void write_log(FILE *fp, LogLevel level, const char *fmt, ...) {
    if (!fp) return;

    // Timestamp
    time_t t = time(NULL);
    struct tm *tm_info = localtime(&t);
    char timebuf[32];
    strftime(timebuf, sizeof(timebuf), "%Y-%m-%d %H:%M:%S", tm_info);

    // Format message
    va_list args;
    va_start(args, fmt);

    fprintf(fp, "[%s] [%s] ", timebuf, level_to_string(level));
    vfprintf(fp, fmt, args);
    fprintf(fp, "\n");
    fflush(fp); // flush immediately for safety
```

```
    va_end(args);
}
```

Step 3. Example Writer Program

```
int main(void) {
    FILE *log = fopen("system.log", "a");
    if (!log) {
        perror("Cannot open log file");
        return 1;
    }

    write_log(log, INFO, "System started");
    write_log(log, WARN, "Low disk space on /dev/sda1");
    write_log(log, ERROR, "Failed to connect to database");
    write_log(log, INFO, "Shutdown complete");

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

Run it:

```
[2025-10-15 21:00:32] [INFO] System started
[2025-10-15 21:00:35] [WARN] Low disk space on /dev/sda1
[2025-10-15 21:00:38] [ERROR] Failed to connect to database
[2025-10-15 21:01:00] [INFO] Shutdown complete
```

Step 4. Implement the Reader

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

void read_logs(const char *filename, const char *filter) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        perror("Cannot open log file");
        return;
```

```
    }

    char line[256];
    while (fgets(line, sizeof(line), fp)) {
        if (filter == NULL || strstr(line, filter))
            printf("%s", line);
    }

    fclose(fp);
}
```

Example usage:

```
int main(void) {
    printf("All logs:\n");
    read_logs("system.log", NULL);

    printf("\nOnly errors:\n");
    read_logs("system.log", "ERROR");
    return 0;
}
```

Output:

```
All logs:
[2025-10-15 21:00:32] [INFO] System started
[2025-10-15 21:00:35] [WARN] Low disk space on /dev/sda1
[2025-10-15 21:00:38] [ERROR] Failed to connect to database
[2025-10-15 21:01:00] [INFO] Shutdown complete

Only errors:
[2025-10-15 21:00:38] [ERROR] Failed to connect to database
```

Step 5. Add Command-Line Interface

Combine both features using `argc` and `argv`:

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s [write|read] [message/filter]\n", argv[0]);
```

```
        return 1;
    }

    if (strcmp(argv[1], "write") == 0) {
        FILE *log = fopen("system.log", "a");
        if (!log) {
            perror("open");
            return 1;
        }
        write_log(log, INFO, "%s", argc > 2 ? argv[2] : "Generic log entry");
        fclose(log);
    } else if (strcmp(argv[1], "read") == 0) {
        const char *filter = argc > 2 ? argv[2] : NULL;
        read_logs("system.log", filter);
    } else {
        fprintf(stderr, "Invalid command. Use write or read.\n");
    }
    return 0;
}
```

Usage:

```
./logger write "Hello world"
./logger read
./logger read ERROR
```

Step 6. Handling Errors Gracefully

- Always check return values from `fopen`, `fgets`, and `fprintf`.
- Use `perror()` for system-level diagnostics.
- Flush frequently or close properly to ensure logs persist after crashes.

Step 7. Optional Enhancements

- Add log rotation (rename or truncate after N lines).
- Add log levels (only write logs above a threshold).
- Implement `save_config()` to define a log file path and verbosity from a config file.
- Add timestamps in UTC or with milliseconds for precision.
- Write logs in binary format for higher speed, then parse later.

Why It Matters

Logging is a **core system capability**. This project reinforces:

- Structured file I/O
- Error handling (`errno`, `perror`)
- String parsing and filtering
- Command-line tool design

Every C-based system, from embedded devices to Linux daemons, relies on some form of logging.

Try It Yourself

1. Add a `LogLevel` threshold (ignore logs below WARN).
2. Implement a `rotate_logs()` that renames `system.log` to `system.log.1` when it exceeds 100 lines.
3. Add timestamps in UTC instead of localtime.
4. Use `argv` to let users specify a custom log file name.
5. Store logs both to file and to `stderr` simultaneously.

This completes **Chapter 5: Input, Output, and Files**, a milestone in your journey. You can now handle text, binary, streams, and persistent data with safety and clarity. Next, you'll step into **Chapter 6: Compilation and the Build Process**, where source code transforms into executable binaries through preprocessing, compilation, linking, and automation.

Chapter 6. Compilation and the build process

51. From Source to Executable: The Compilation Pipeline

Every time you run

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

you're launching a complex, multi-stage process that transforms human-readable C code into a machine-executable binary. Understanding this **compilation pipeline** is the heart of becoming a real systems programmer.

Let's unpack what happens between your `.c` file and the final executable.

Step 1. The Four Stages

A C compiler (like `gcc` or `clang`) performs four major stages internally:

Stage	Tool	Input	Output	Description
1. Preprocessing	`cpp`	`hello.c`	expanded source	Handles `#include`, `#define`, and macros
2. Compilation	`cc1`	expanded source	`hello.s`	Translates C into assembly
3. Assembly	`as`	`hello.s`	`hello.o`	Converts assembly into machine code
4. Linking	`ld`	`.o` + libraries	`hello`	Combines everything into a runnable program

You can stop at any step with compiler flags to see what's happening.

Tiny Code: Observe Each Stage

Let's use a simple program:

```c
// hello.c
#include <stdio.h>

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

Run these commands to inspect each stage:

```bash
# 1. Preprocessing
gcc -E hello.c -o hello.i

# 2. Compilation to Assembly
gcc -S hello.i -o hello.s

# 3. Assembly to Object File
gcc -c hello.s -o hello.o

# 4. Linking
gcc hello.o -o hello
```

Now check what each file looks like:

- `hello.i`: C code with headers expanded
- `hello.s`: Human-readable assembly instructions
- `hello.o`: Machine code (binary object)
- `hello`: Final executable

Run:

```bash
./hello
```

Output:

```
Hello, C!
```

Step 2. Preprocessing (`#include`, `#define`, `#if`)

The preprocessor handles all lines starting with `#`. It's purely textual, no code execution yet.

```
gcc -E hello.c -o hello.i
```

Open `hello.i` and you'll see thousands of lines from `stdio.h` inserted into your code. It also replaces macros and removes comments.

This is the stage where your **headers**, **macros**, and **conditional compilation** come to life.

Step 3. Compilation (to Assembly)

Next, the compiler translates your preprocessed C code into **assembly language** for your target CPU.

```
gcc -S hello.i -o hello.s
```

Open `hello.s` to peek at low-level instructions like:

```
mov     edi, OFFSET FLAT:.LC0
call    puts
```

These are CPU-specific, on x86, ARM, or RISC-V, they'll differ. This stage also performs **optimization**, **type checking**, and **error detection**.

Step 4. Assembly (to Object File)

The assembler converts the `.s` file into raw machine instructions and data structures, producing a **relocatable object file**.

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

You can inspect it with:

```
objdump -d hello.o
```

Each function in your code becomes a **symbol** in this object file.

Step 5. Linking

The linker (`ld`) combines object files and libraries into a single executable.

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

If your program uses external functions (like `printf`), the linker locates them in system libraries (e.g., `/usr/lib/libc.so`) and records their addresses.

The result: one self-contained executable ready to run.

Step 6. Inspect the Final Binary

Use tools like:

```
file hello
nm hello | head
readelf -h hello
```

These reveal:

- File type (ELF, Mach-O, etc.)
- Defined and undefined symbols
- Sections like `.text`, `.data`, `.bss`

You're now seeing your C code at the machine level.

Step 7. Cleanup and Automation

While it's educational to run each step manually, most of the time you'll rely on:

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

or a `Makefile` to orchestrate multiple files (we'll cover that in Section 55).

Tiny Code: Multi-File Example

Let's build a two-file program manually:

main.c

```c
#include "greet.h"

int main(void) {
    greet("C programmer");
    return 0;
}
```

greet.c

```c
#include <stdio.h>
void greet(const char *name) {
    printf("Hello, %s!\n", name);
}
```

greet.h

```c
void greet(const char *name);
```

Compile and link manually:

```
gcc -c main.c
gcc -c greet.c
gcc main.o greet.o -o app
./app
```

Output:

```
Hello, C programmer!
```

Why It Matters

Understanding the compilation pipeline helps you:

- Debug tricky build errors (`undefined reference`, `multiple definition`, etc.)
- Control optimization and debugging symbols
- Inspect intermediate stages for learning or tuning performance
- Build your own lightweight build systems or compilers

This is how **source code becomes machine reality**, step by step, precisely defined.

Try It Yourself

1. Generate all intermediate files (`.i`, `.s`, `.o`) for a few programs and inspect them.
2. Experiment with `gcc -O0`, `-O2`, and `-O3` and observe how assembly changes.
3. Add `-g` and explore the binary with `gdb`.
4. Build a program that spans multiple `.c` files.
5. Use `nm` and `objdump` to trace how symbols move through the stages.

Next, you'll explore the **preprocessor and macros**, the engine behind includes, constants, and compile-time code generation.

52. The Preprocessor and Macros

Before your C code is ever compiled, it passes through a powerful text-handling stage called the **preprocessor**. This is where headers are included, macros are expanded, and conditional compilation happens.

The preprocessor doesn't "understand" C, it performs text substitution and file inclusion, preparing your code for the compiler.

Step 1. What the Preprocessor Does

Every line that starts with `#` is a **preprocessor directive**. Common ones include:

Directive	Purpose
`#include`	Insert contents of a header file
`#define`	Define a macro or constant
`#undef`	Remove a macro definition
`#if`, `#ifdef`, `#ifndef`	Conditional compilation
`#else`, `#elif`, `#endif`	Branch logic for the preprocessor
`#error`	Stop compilation with a message
`#pragma`	Compiler-specific instruction

Tiny Code: See It in Action

Create `macro.c`:

```c
#include <stdio.h>

#define PI 3.14159
#define CIRCLE_AREA(r) (PI * (r) * (r))
#define SQUARE(x) ((x) * (x))

int main(void) {
    printf("PI = %.2f\n", PI);
    printf("Area of circle (r=2): %.2f\n", CIRCLE_AREA(2));
    printf("Square of 5: %d\n", SQUARE(5));
    return 0;
}
```

Compile and run:

```
gcc macro.c -o macro
./macro
```

Output:

```
PI = 3.14
Area of circle (r=2): 12.57
Square of 5: 25
```

Step 2. Expanding Macros

You can inspect the preprocessor output before compilation:

```
gcc -E macro.c -o macro.i
```

Open `macro.i`, you'll see all `#include` files expanded and macros replaced with their values.

This is a great way to **debug macro behavior** or check how large standard headers expand.

Step 3. Function-Like Macros

Macros can look like functions, but they are **expanded inline**, meaning no call overhead, but also no type safety.

```c
#define ADD(a, b) ((a) + (b))
```

Usage:

```c
printf("%d\n", ADD(2, 3));  // becomes ((2) + (3))
```

Be careful with missing parentheses:

```c
#define BAD_ADD(a, b) a + b
printf("%d\n", 2 * BAD_ADD(3, 4)); // expands to 2 * 3 + 4 → 10, not 14
```

Always wrap parameters and entire expressions in parentheses.

Step 4. Stringizing and Token Pasting

Macros can manipulate text using special operators.

Stringizing (#) turns an argument into a string literal:

```c
#define PRINT_EXPR(expr) printf(#expr " = %d\n", expr)
```

Usage:

```c
int x = 5, y = 10;
PRINT_EXPR(x + y);  // prints: x + y = 15
```

Token Pasting (##) concatenates tokens:

```c
#define MAKE_VAR(name, num) name##num
int MAKE_VAR(counter, 1) = 42;  // becomes int counter1 = 42;
```

Step 5. Conditional Compilation

You can include or exclude code based on conditions:

```
#define DEBUG 1

#if DEBUG
    #define LOG(msg) printf("DEBUG: %s\n", msg)
#else
    #define LOG(msg)
#endif
```

Usage:

```
int main(void) {
    LOG("Starting program");
    printf("Running main logic\n");
    return 0;
}
```

Compile with or without `-DDEBUG=1`:

```
gcc -DDEBUG=1 log.c -o log
```

You can also use:

```
#ifdef DEBUG
#ifndef RELEASE
```

Step 6. Header Guards

Prevent multiple inclusions of the same header file by using preprocessor guards:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

void greet(void);

#endif
```

If `MY_HEADER_H` is already defined, the contents are skipped. This prevents duplicate definitions across multiple includes.

Step 7. Built-in Macros

The compiler defines a few handy macros automatically:

Macro	Expands To
`__FILE__`	current filename
`__LINE__`	current line number
`__DATE__`	compilation date
`__TIME__`	compilation time
`__func__`	current function name (C99+)

Example:

```
printf("Error at %s:%d in %s()\n", __FILE__, __LINE__, __func__);
```

Output:

```
Error at macro.c:10 in main()
```

Tiny Code: Debug Macro

```
#define DEBUG_PRINT(fmt, ...) \
    fprintf(stderr, "[%s:%d] " fmt "\n", __FILE__, __LINE__, __VA_ARGS__)

int main(void) {
    int x = 10;
    DEBUG_PRINT("x = %d", x);
    return 0;
}
```

Output:

```
[macro.c:5] x = 10
```

This is how logging frameworks are implemented in C using macros.

Step 8. Undefining and Redefining

You can remove a macro with `#undef`:

```
#undef PI
#define PI 3.14
```

This is often used in large projects to avoid macro name collisions between libraries.

Why It Matters

The preprocessor gives C flexibility and power at **compile time**, enabling:

- Cross-platform builds (conditional compilation)
- Debug logging systems
- Inline performance optimizations
- Simplified configuration management

It's also a double-edged sword, overusing macros can make code hard to debug and maintain. Modern C favors **inline functions** for most use cases (see Section 54), but macros remain indispensable for low-level systems work.

Try It Yourself

1. Write a macro that swaps two variables without a temporary.
2. Implement a `LOG(level, msg)` macro that prints messages only if `level >= MIN_LOG_LEVEL`.
3. Use `__DATE__` and `__TIME__` to print build information.
4. Add header guards to all your `.h` files and test multiple inclusions.
5. Try `gcc -E` on different programs to understand how preprocessing changes the source.

In the next section, you'll go deeper into **conditional compilation**, controlling which parts of your program are built based on platform, features, or debugging needs.

53. Conditional Compilation (`#if`, `#ifdef`, `#ifndef`)

Conditional compilation lets you **control which code gets compiled**, not at runtime, but at compile time. This is how C programs adapt to different operating systems, architectures, or build configurations without changing source files manually.

Think of it as logic for the compiler's *eyes only*.

Step 1. Why Conditional Compilation Exists

Large C programs often need to handle differences such as:

- Platform (Windows, Linux, macOS, embedded)
- Compiler (gcc, clang, MSVC)
- Debug vs release builds
- Optional features or experimental modules

Instead of maintaining multiple versions of the same file, you can use conditional directives to selectively include or exclude code.

Step 2. The Core Directives

Directive	Purpose
`#if <expr>`	Compile code if expression is true
`#ifdef <macro>`	Compile if macro is defined
`#ifndef <macro>`	Compile if macro is not defined
`#else`	Alternate block
`#elif <expr>`	Else-if for preprocessor
`#endif`	Marks the end of a conditional block

These work only during **preprocessing**, before compilation starts.

Tiny Code: Platform-Specific Compilation

```c
#include <stdio.h>

int main(void) {
#ifdef _WIN32
    printf("Running on Windows\n");
#elif __linux__
    printf("Running on Linux\n");
#elif __APPLE__
    printf("Running on macOS\n");
#else
    printf("Unknown platform\n");
#endif
    return 0;
}
```

Compile and run on your system. The output will depend on which predefined macros your compiler sets automatically.

Step 3. Enabling and Disabling Features

You can define flags at compile time with `-D`:

```
gcc -DDEBUG log.c -o log
```

In your code:

```
#ifdef DEBUG
    printf("Debug mode: extra checks enabled\n");
#endif
```

No recompilation needed to switch, just re-run `gcc` with or without `-DDEBUG`.

You can also assign values:

```
gcc -DVERSION=2 main.c -o main
```

Then:

```
#if VERSION >= 2
    printf("New feature enabled!\n");
#endif
```

Step 4. Guarding Code with `#ifndef`

This is one of the most common idioms in C headers:

```
#ifndef CONFIG_H
#define CONFIG_H

#define MAX_CLIENTS 100
#define TIMEOUT_MS 3000

#endif
```

It ensures that if `config.h` is included multiple times, it only gets processed once. Every header in the C standard library uses this pattern.

Step 5. Excluding Experimental Code

```
#define ENABLE_EXPERIMENTAL 0

#if ENABLE_EXPERIMENTAL
void experimental_feature() {
    printf("Running experimental feature\n");
}
#endif
```

If `ENABLE_EXPERIMENTAL` is set to `0`, this code is completely removed before compilation, it doesn't even exist in the object file.

Tiny Code: Debug Mode Example

```
#include <stdio.h>

#define DEBUG_MODE 1

void compute(int x) {
#if DEBUG_MODE
    printf("[DEBUG] compute() called with x=%d\n", x);
#endif
    printf("Result: %d\n", x * x);
}

int main(void) {
    compute(5);
    return 0;
}
```

Output when `DEBUG_MODE` is 1:

```
[DEBUG] compute() called with x=5
Result: 25
```

Set `DEBUG_MODE` to 0, recompile, and the `[DEBUG]` message disappears entirely.

Step 6. Using `#elif` and `#else`

```c
#define OS 2

#if OS == 1
    #define OS_NAME "Windows"
#elif OS == 2
    #define OS_NAME "Linux"
#else
    #define OS_NAME "Unknown"
#endif

int main(void) {
    printf("OS: %s\n", OS_NAME);
    return 0;
}
```

Output:

```
OS: Linux
```

Step 7. Combining with Logical Operators

You can use `&&`, `||`, and `!` in preprocessor conditions.

```c
#if defined(DEBUG) && !defined(RELEASE)
    printf("Debug build only\n");
#endif
```

You can even use numeric comparisons:

```c
#if VERSION >= 3
    printf("Version 3+ detected\n");
#endif
```

Step 8. Forcing Compilation Errors

Sometimes you want to stop compilation if a required macro is missing:

```
#ifndef API_KEY
#error "API_KEY not defined! Please compile with -DAPI_KEY=your_key"
#endif
```

This is useful for configuration validation at build time.

Step 9. Compiler-Specific Macros

Compilers automatically define macros to identify themselves and the environment.

Macro	Meaning
`__GNUC__`	Defined by GCC
`__clang__`	Defined by Clang
`_MSC_VER`	Defined by MSVC
`__x86_64__`	64-bit architecture
`__arm__`, `__aarch64__`	ARM architectures
`__STDC__`	Conforms to ANSI C standard

You can use these to write portable, adaptive code:

```
#ifdef __clang__
    printf("Compiled with Clang\n");
#elif defined(__GNUC__)
    printf("Compiled with GCC\n");
#endif
```

Tiny Code: Portable Sleep Function

```
#include <stdio.h>

#ifdef _WIN32
    #include <windows.h>
    #define SLEEP(ms) Sleep(ms)
#else
    #include <unistd.h>
    #define SLEEP(ms) usleep((ms) * 1000)
#endif
```

```c
int main(void) {
    printf("Waiting...\n");
    SLEEP(1000);
    printf("Done!\n");
    return 0;
}
```

This compiles cleanly on both Windows and Linux with no code changes.

Why It Matters

Conditional compilation makes your C code:

- **Portable**, same code runs on multiple systems
- **Configurable**, features can be toggled at build time
- **Maintainable**, no need for multiple codebases
- **Efficient**, excluded code doesn't even enter the binary

In system software and embedded development, this is indispensable.

Try It Yourself

1. Write a program that prints a different greeting depending on the OS.
2. Use `#if` and `#error` to enforce that only one of `DEBUG` or `RELEASE` can be defined.
3. Write a header that defines constants for different CPU architectures.
4. Add a feature flag (`ENABLE_LOGGING`) that can be turned on/off via `gcc -D`.
5. Use `#ifdef` and `#ifndef` to create a lightweight build-time configuration system.

In the next section, you'll take the next step toward clean, maintainable C code by learning about **inline functions** and **header hygiene**, modern, safer replacements for many macro patterns.

54. Inline Functions and Header Hygiene

In early C, programmers often relied on macros for performance and reuse. But macros have big drawbacks, no type checking, no debugging symbols, and messy error messages.

Inline functions were introduced to solve this problem. They combine the efficiency of macros with the safety of real functions.

This section also covers **header hygiene**, or how to write clean, reusable `.h` files that scale safely across large projects.

Step 1. What Does "Inline" Mean?

Normally, calling a function like `add(a, b)` incurs a small overhead, the CPU jumps to the function and back. Inlining means the compiler **replaces the call with the function's code directly**, avoiding that jump.

You can suggest this with the `inline` keyword:

```
inline int add(int a, int b) {
    return a + b;
}
```

When used properly, it's as fast as a macro but behaves like a real function.

Step 2. Comparing Macros vs Inline Functions

Macro version:

```
#define ADD(a, b) ((a) + (b))
```

Inline version:

```
inline int add(int a, int b) {
    return a + b;
}
```

Macro:

- No type checking
- May cause multiple evaluations (e.g., `ADD(x++, y++)`)
- Harder to debug

Inline function:

- Type-checked
- Single evaluation
- Can be stepped through in a debugger

Step 3. Declaring Inline Functions in Headers

When defining inline functions in header files, add `static` to avoid multiple-definition errors:

```c
// math_utils.h
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

static inline int square(int x) {
    return x * x;
}

static inline int max(int a, int b) {
    return (a > b) ? a : b;
}

#endif
```

This ensures each `.c` file that includes the header gets its own copy, avoiding linker conflicts.

Tiny Code: Inline Utilities

```c
#include <stdio.h>

static inline int cube(int x) {
    return x * x * x;
}

int main(void) {
    int n = 3;
    printf("cube(%d) = %d\n", n, cube(n));
    return 0;
}
```

Output:

```
cube(3) = 27
```

The compiler will expand `cube(3)` directly into `3 * 3 * 3`, no function call overhead.

Step 4. Inline and the Compiler

`inline` is a *hint* to the compiler, not a command. The compiler decides whether inlining actually improves performance.

You can force inlining (non-portably) with attributes:

```
__attribute__((always_inline)) inline void fast_add(int *x, int y) {
    *x += y;
}
```

But it's best to let the optimizer choose. Inlining too much can increase binary size (known as "code bloat").

Step 5. Inline and Linkage

Inline functions behave differently depending on whether they're declared `static`, `extern`, or plain `inline`.

Keyword Combination	Meaning
`static inline`	Visible only in this translation unit (safe for headers)
`extern inline`	Shared across translation units (rarely needed)
`inline` (alone)	Behavior depends on compiler and standard version

Stick with `static inline` for header-defined helpers.

Step 6. Inline vs Macros: Debug Example

Macro:

```
#define PRINT(x) printf("%d\n", x)
```

Error output when debugging might show:

```
macro.c: In function 'main': macro.c:5: error: expected ';'
```

Inline:

```c
inline void print(int x) {
    printf("%d\n", x);
}
```

Now you get a clean message:

```
error: too few arguments to function 'print'
```

Inlining makes error handling and debugging much cleaner.

Step 7. Header Hygiene, The Rules of Clean Headers

Headers define your program's public interface. Poorly written headers cause multiple-definition errors, redefinition warnings, and broken builds.

Follow these guidelines:

1. **Use header guards**

   ```c
   #ifndef MYLIB_H
   #define MYLIB_H
   // contents
   #endif
   ```

2. **Keep headers minimal** Only include what's necessary, use forward declarations when possible.
3. **Don't put function definitions unless they're `static inline`.**
4. **Never use `using namespace`, global variables, or large macros** in headers.
5. **Group related declarations together**:

   ```c
   typedef struct Point { int x, y; } Point;
   void move(Point *p, int dx, int dy);
   ```

6. **Include standard headers only when required**:

   ```c
   #include <stdio.h>  // only if you use FILE*
   ```

Tiny Code: Clean Header + Implementation Example

mathlib.h

```c
#ifndef MATHLIB_H
#define MATHLIB_H

typedef struct {
    int x, y;
} Point;

static inline int add(int a, int b) { return a + b; }
void print_point(Point p);

#endif
```

mathlib.c

```c
#include <stdio.h>
#include "mathlib.h"

void print_point(Point p) {
    printf("(%d, %d)\n", p.x, p.y);
}
```

main.c

```c
#include "mathlib.h"

int main(void) {
    Point p = {2, 3};
    print_point(p);
    printf("Sum = %d\n", add(2, 5));
    return 0;
}
```

Build:

```
gcc main.c mathlib.c -o demo
```

Output:

```
(2, 3)
Sum = 7
```

This structure mirrors real-world C libraries, headers for declarations, `.c` files for definitions, and inline helpers where performance matters.

Step 8. Inline and Optimization Flags

Use `-O2` or higher optimization to let the compiler inline aggressively:

```
gcc -O2 main.c -o main
```

At `-O0` (no optimization), even inline functions may not be expanded.

Step 9. Inline in C99 and Beyond

Inline semantics were standardized in **C99**. Older compilers (pre-C99) treated inline inconsistently. Always compile with `-std=c99` or later for predictable behavior:

```
gcc -std=c99 main.c -o main
```

Why It Matters

Inline functions give you:

- Performance like macros
- Type safety and cleaner debugging
- Reusable logic in headers
- Safer and smaller helper functions

They are a modern C programmer's best tool for writing efficient yet maintainable code.

Try It Yourself

1. Replace three of your macros from previous exercises with inline functions.
2. Benchmark your program with and without `-O2` to see the difference.
3. Write a header-only math library using `static inline` functions.
4. Add header guards and check with multiple includes.
5. Use `objdump -d` to confirm whether your inline code actually got expanded.

Next, you'll automate your growing C projects with **Makefiles and build systems**, the tools that manage compilation, linking, and dependencies efficiently.

55. Makefiles and Build Automation

Compiling one or two C files by hand is fine, but real projects quickly grow to dozens or hundreds of files. Typing long `gcc` commands every time becomes tedious, error-prone, and inconsistent across environments.

That's where **Makefiles** come in. They automate the build process, track dependencies, and rebuild only what changed.

Let's build a complete understanding of how to use `make` and write simple but powerful Makefiles.

Step 1. What Is `make`?

`make` is a tool that reads a file called **Makefile** and executes the build rules it defines.

Each rule describes:

1. A **target** (the thing you want to build)
2. Its **dependencies** (what it needs)
3. The **commands** to build it

Basic syntax:

```
target: dependencies
<TAB>command
```

Yes, the indentation **must** be a real tab, not spaces.

Step 2. The Simplest Makefile

Suppose your project has:

```
main.c
math.c
math.h
```

Makefile:

```
app: main.c math.c
    gcc main.c math.c -o app
```

Build:

```
make
```

Output:

```
gcc main.c math.c -o app
```

Run:

```
./app
```

Now if you run `make` again, nothing happens, because `make` sees that the output (`app`) is newer than the sources. That's the magic of dependency tracking.

Step 3. Split into Compilation Steps

A better version builds `.o` files separately:

```
app: main.o math.o
    gcc main.o math.o -o app

main.o: main.c math.h
    gcc -c main.c

math.o: math.c math.h
    gcc -c math.c
```

Now when you change only `math.c`, only `math.o` recompiles.

Tiny Code: Minimal Project

main.c

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

int main(void) {
    printf("2 + 3 = %d\n", add(2, 3));
    return 0;
}
```

math.c

```
int add(int a, int b) { return a + b; }
```

math.h

```
int add(int a, int b);
```

Makefile

```
app: main.o math.o
    gcc main.o math.o -o app

main.o: main.c math.h
    gcc -c main.c

math.o: math.c math.h
    gcc -c math.c

clean:
    rm -f *.o app
```

Run:

```
make
./app
make clean
```

Step 4. Use Variables

Makefiles support variables to avoid repetition:

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
OBJ = main.o math.o

app: $(OBJ)
    $(CC) $(OBJ) -o app

%.o: %.c
```

```
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) app
```

Here:

- `$<` = first dependency
- `$@` = target name
- `%.o: %.c` = pattern rule (applies to all matching files)

Step 5. Add Debug and Release Modes

```
CC = gcc
CFLAGS = -Wall -std=c99
DEBUG_FLAGS = -g -O0
RELEASE_FLAGS = -O2

OBJ = main.o math.o
TARGET = app

all: release

debug: CFLAGS += $(DEBUG_FLAGS)
debug: $(TARGET)

release: CFLAGS += $(RELEASE_FLAGS)
release: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) -o $(TARGET)

clean:
    rm -f $(OBJ) $(TARGET)
```

Build in debug mode:

```
make debug
```

Build optimized release:

```
make release
```

Step 6. Automatic Dependencies

You can have `gcc` generate dependency files automatically:

```
gcc -MMD -c main.c
```

This creates `main.d` which tracks included headers. You can include these files in your Makefile for automatic rebuilds:

```
-include $(OBJ:.o=.d)
```

That's how professional build systems keep dependencies accurate.

Step 7. Phony Targets

Targets that don't produce actual files should be marked **phony**:

```
.PHONY: clean all debug release
```

This prevents file name collisions (e.g., if a file named `clean` exists).

Step 8. Organize Larger Projects

For multi-directory projects:

```
src/
    main.c
    util.c
include/
    util.h
```

You can structure your Makefile like:

```
SRC = src/main.c src/util.c
OBJ = $(SRC:.c=.o)
CFLAGS = -Iinclude -Wall
TARGET = app

$(TARGET): $(OBJ)
    $(CC) $(OBJ) -o $@

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

`-Iinclude` tells the compiler where to find header files.

Step 9. Use Built-in Rules

`make` already knows how to build `.o` from `.c`. A minimalist Makefile can be:

```
app: main.o math.o
    gcc $^ -o $@
```

Where `$^` expands to all dependencies (`main.o math.o`).

Step 10. Tiny Code: Library Build Example

```
CC = gcc
CFLAGS = -Wall -std=c99
OBJ = util.o io.o
LIB = libtools.a

$(LIB): $(OBJ)
    ar rcs $(LIB) $(OBJ)

clean:
    rm -f $(OBJ) $(LIB)
```

Build your static library:

```
make
```

Now link it:

```
gcc main.c -L. -ltools -o app
```

Why It Matters

Makefiles give you:

- **Reproducible builds**, same commands every time
- **Incremental recompilation**, only changed files rebuild
- **Multiple configurations**, debug, release, test
- **Extensibility**, can run scripts, code generation, packaging, etc.

Every serious C project, from the Linux kernel to tiny embedded tools, relies on `make` or its descendants (like CMake, Ninja, Meson).

Try It Yourself

1. Create a project with 3 `.c` files and 2 `.h` files.
2. Write a Makefile that supports `make debug`, `make release`, and `make clean`.
3. Add a `make install` target that copies the binary to `/usr/local/bin`.
4. Use variables like `CC`, `CFLAGS`, and pattern rules to simplify your file list.
5. Run `make -n` to preview commands without executing them.

In the next section, you'll learn how to **link multiple files and libraries**, understanding object files, symbols, and how your code connects together during the build process.

56. Linking Multiple Files

When a program grows beyond one `.c` file, the compiler must combine them into a single executable. This process, **linking**, is what joins all your functions, variables, and library references into one binary.

You've already seen snippets of it with:

```
gcc main.o math.o -o app
```

But now we'll go deeper into *how* linking works and what happens when things go wrong.

Step 1. The Two Compilation Phases

Every C build has two main phases:

1. **Compilation** – each `.c` file becomes a `.o` (object file).

   ```
   gcc -c main.c -o main.o
   gcc -c math.c -o math.o
   ```

   Each `.o` file contains *machine code* and *symbol tables* (lists of what it defines and what it needs).

2. **Linking** – the linker (`ld`) merges all `.o` files and libraries into an executable:

   ```
   gcc main.o math.o -o app
   ```

If any symbol is missing (like an undefined function), the linker fails.

Step 2. What Are Symbols?

Symbols are names the compiler uses to track functions and global variables. There are two kinds:

- **Defined symbols** – functions or variables provided by this file.
- **Undefined symbols** – things it *needs* from another file.

Example:

math.c

```c
int add(int a, int b) { return a + b; }
```

main.c

```c
#include <stdio.h>
int add(int, int);

int main(void) {
    printf("%d\n", add(2, 3));
    return 0;
}
```

Compile and link:

```
gcc -c main.c
gcc -c math.c
gcc main.o math.o -o app
```

Run:

```
5
```

If you forget to link `math.o`:

```
gcc main.o -o app
```

You'll get:

```
undefined reference to `add'
```

Step 3. Using Header Files for Declarations

Each `.c` file should declare (not define) external functions in a header:

math.h

```
#ifndef MATH_H
#define MATH_H
int add(int a, int b);
#endif
```

Then include it in both `main.c` and `math.c`:

```
#include "math.h"
```

This ensures consistency between the declaration and definition.

Step 4. Linking Object Files and Libraries

You can link any number of `.o` files:

```
gcc main.o math.o util.o io.o -o app
```

You can also link with libraries:

```
gcc main.o -lm -o app
```

(`-lm` links the math library that provides functions like `sqrt`, `sin`, etc.)

The `-l` flag searches `/usr/lib` and `/lib` by default.

Custom library example:

```
gcc main.o -L. -lmyutils -o app
```

Here, `-L.` adds the current directory to the library search path, and `-lmyutils` links `libmyutils.a` or `libmyutils.so`.

Step 5. The Order of Linking Matters

The linker reads from **left to right**. If a symbol is used before its definition appears, it might fail.

Example:

```
gcc -lm main.o -o app    #  wrong order
gcc main.o -lm -o app    #  correct
```

Always list libraries **after** the object files that need them.

Step 6. Splitting and Linking a Multi-File Project

```
project/
  main.c
  math.c
  io.c
  math.h
  io.h
  Makefile
```

Makefile

```
CC = gcc
CFLAGS = -Wall -std=c99
OBJ = main.o math.o io.o
TARGET = app

$(TARGET): $(OBJ)
    $(CC) $(OBJ) -o $(TARGET)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

Now, just run:

```
make
./app
```

The Makefile takes care of compiling and linking everything in order.

Tiny Code: Building a Small Modular Program

math.c

```
#include "math.h"
int square(int x) { return x * x; }
int cube(int x) { return x * x * x; }
```

math.h

```
#ifndef MATH_H
#define MATH_H
int square(int x);
int cube(int x);
#endif
```

main.c

```c
#include <stdio.h>
#include "math.h"

int main(void) {
    printf("square(3) = %d\n", square(3));
    printf("cube(2) = %d\n", cube(2));
    return 0;
}
```

Build manually:

```
gcc -c math.c -o math.o
gcc -c main.c -o main.o
gcc main.o math.o -o app
./app
```

Output:

```
square(3) = 9
cube(2) = 8
```

Step 7. Static vs Dynamic Linking

Type	File	Description
Static	`.a`	Code is copied into the executable
Dynamic	`.so`	Code is shared at runtime via system libraries

Static linking:

```
gcc main.o -L. -lmath -static -o app
```

Dynamic linking (default):

```
gcc main.o -L. -lmath -o app
```

Dynamic executables are smaller and share libraries across programs.

Step 8. Inspecting Linked Binaries

To see which symbols are defined or missing:

```
nm main.o | head
```

Or for a full binary:

```
objdump -t app | less
ldd app
```

`ldd` shows which shared libraries the program depends on.

Step 9. Common Linking Errors

Error	Meaning	Fix
`undefined reference to <symbol>`	Missing `.o` or library	Add all object files or correct `-l` flags
`multiple definition of <symbol>`	Same function defined in multiple files	Use `extern` in declarations
`cannot find -lfoo`	Missing library file	Check `-L` paths or install dev package
`relocation truncated`	Mismatched architectures	Ensure all files are built for same target

Step 10. Inline vs External Linking

Inline functions defined as `static inline` in headers do *not* require linking, each `.c` file gets its own copy. But normal functions in `.c` files must be linked exactly once.

Why It Matters

Linking is where your program becomes *whole*. It teaches you:

- How `.o` and `.h` files interact
- How libraries integrate with your binaries
- How to debug missing symbols and multiple definitions
- How modular design affects build architecture

Understanding the linker is essential for building scalable, multi-file systems, from small utilities to entire kernels.

Try It Yourself

1. Create a program with 3 `.c` files and 3 `.h` files.
2. Intentionally omit one object file and observe the linker error.
3. Use `nm` to inspect which functions each `.o` defines and references.
4. Build a static library (`ar rcs libutils.a util.o`) and link it manually.
5. Use `ldd` to list shared libraries your compiled program depends on.

Next, you'll learn how to create and use **static and shared libraries**, the modular building blocks that every serious C project relies on for reusability and scalability.

57. Static and Shared Libraries

In large C projects, you often want to **reuse code** across multiple programs, without copying the same `.c` files everywhere. That's exactly what **libraries** are for.

A **library** is a collection of precompiled object files (`.o`) packaged together. There are two main kinds:

Type	Extension	Loaded	Typical Use
Static Library	`.a`	At compile time	Simpler, self-contained executables
Shared Library	`.so`	At runtime	Smaller binaries, reusable system-wide

Let's see how to build and use both.

Step 1. Build a Static Library (`.a`)

A static library is just an archive of object files.

Example project:

```
math.c
string_utils.c
main.c
```

math.c

```c
int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }
```

string_utils.c

```c
#include <string.h>

int str_eq(const char *a, const char *b) {
    return strcmp(a, b) == 0;
}
```

math.h

```c
int add(int a, int b);
int mul(int a, int b);
```

string_utils.h

```c
int str_eq(const char *a, const char *b);
```

Compile the object files:

```
gcc -c math.c string_utils.c
```

Create a static library:

```
ar rcs libmylib.a math.o string_utils.o
```

Now you have `libmylib.a`, your first static library.

Step 2. Link the Library

main.c

```c
#include <stdio.h>
#include "math.h"
#include "string_utils.h"

int main(void) {
    printf("3 + 4 = %d\n", add(3, 4));
    printf("Equal? %d\n", str_eq("abc", "abc"));
    return 0;
}
```

Link it:

```
gcc main.c -L. -lmylib -o app
```

Output:

```
3 + 4 = 7
Equal? 1
```

Here:

- `-L.` tells the compiler to look in the current directory
- `-lmylib` links against `libmylib.a`

The `.a` file is copied into your executable, you can now delete it and your program will still run.

Step 3. Inspect the Static Library

List its contents:

```
ar -t libmylib.a
```

Output:

```
math.o
string_utils.o
```

Extract a file:

```
ar -x libmylib.a math.o
```

You can think of `.a` files like a ".zip" of `.o` modules.

Step 4. Build a Shared Library (`.so`)

A shared library is loaded dynamically at runtime, not compiled into the executable. They're what you see in `/usr/lib` as `.so` (Linux) or `.dll` (Windows).

Build one:

```
gcc -fPIC -c math.c string_utils.c
gcc -shared -o libmylib.so math.o string_utils.o
```

Now you have a shared library:

```
libmylib.so
```

`-fPIC` means "position-independent code", required for shared libraries.

Step 5. Link a Program Against It

```
gcc main.c -L. -lmylib -o app
```

Run it:

```
./app
```

If you get:

```
error while loading shared libraries: libmylib.so: cannot open shared object file
```

You need to add the current directory to the runtime library path:

```
export LD_LIBRARY_PATH=.
./app
```

Output:

```
3 + 4 = 7
Equal? 1
```

Step 6. Verify the Linking Type

Check whether your program is statically or dynamically linked:

```
ldd app
```

Output for dynamic linking:

```
libmylib.so => ./libmylib.so (0x00007f8e8b...)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (...)
```

For static linking, `libmylib.a` won't appear, it's baked into the executable.

Step 7. Installing Libraries System-Wide

To make your library accessible globally:

Copy the `.so` file to `/usr/local/lib`:

```
sudo cp libmylib.so /usr/local/lib
sudo ldconfig
```

Copy headers to `/usr/local/include`:

```
sudo cp math.h string_utils.h /usr/local/include
```

Now you can compile anywhere with:

```
gcc main.c -lmylib -o app
```

Step 8. Versioning Shared Libraries

Real-world shared libraries include version numbers for compatibility:

```
libmylib.so.1.0.0
libmylib.so -> libmylib.so.1.0.0  (symlink)
```

You can set this during build:

```
gcc -shared -Wl,-soname,libmylib.so.1 -o libmylib.so.1.0.0 math.o string_utils.o
ln -sf libmylib.so.1.0.0 libmylib.so
```

Tiny Code: Combined Example

Makefile

```
CC = gcc
CFLAGS = -Wall -fPIC
OBJS = math.o string_utils.o
TARGET_STATIC = libmylib.a
TARGET_SHARED = libmylib.so

all: $(TARGET_STATIC) $(TARGET_SHARED)

$(TARGET_STATIC): $(OBJS)
    ar rcs $@ $^

$(TARGET_SHARED): $(OBJS)
    $(CC) -shared -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET_STATIC) $(TARGET_SHARED)
```

Build:

```
make
```

Link app:

```
gcc main.c -L. -lmylib -o app
```

Run:

```
LD_LIBRARY_PATH=. ./app
```

Step 9. Static vs Shared Tradeoffs

Feature	Static (`.a`)	Shared (`.so`)
Linking	At compile time	At runtime
File size	Larger executable	Smaller executable
Update	Recompile required	Replace `.so` file
Portability	Fully self-contained	Needs library present
Speed	Slightly faster	Slight load delay

For small tools or embedded systems, use static. For large or updatable software, prefer shared.

Step 10. Inspecting Library Symbols

To see what functions are exported:

```
nm -D libmylib.so | grep add
```

To check dynamic symbol resolution:

```
objdump -T libmylib.so | head
```

Why It Matters

Libraries make your C code modular, maintainable, and reusable. They are the foundation of every serious C ecosystem, from `libc` to OpenSSL to SDL.

Once you understand how to build and link your own `.a` and `.so` files, you can:

- Ship APIs others can use
- Organize large codebases cleanly
- Understand how system libraries integrate into every program you compile

Try It Yourself

1. Build a small math library (`libmathx.a`, `libmathx.so`).
2. Create a header-only helper and test it in multiple programs.
3. Try mixing static and shared libraries in one build.
4. Inspect your shared library with `ldd` and `nm`.
5. Version your `.so` file using symbolic links and test compatibility.

Next, you'll explore how **compiler flags and optimization levels** affect performance, safety, and debugging, learning how to tune `gcc` for both development and release builds.

58. Compiler Flags and Optimization Levels

Once your program compiles and links correctly, the next step is mastering **compiler flags**, the switches that control warnings, debugging info, optimization, and performance.

Using the right flags can make your C code **safer**, **faster**, and **easier to debug**.

Let's go through the essential `gcc` and `clang` options every C developer should know.

Step 1. The Basic Compilation Command

The most common compile command looks like:

```
gcc main.c -o main
```

This compiles `main.c` into an executable called `main` using **default settings**, minimal warnings, no optimization, and no debug info.

For serious development, you'll want more control.

Step 2. Warning Flags

Warnings are the compiler's early-warning system. They catch mistakes before they become bugs.

Flag	Meaning
`-Wall`	Enable most common warnings
`-Wextra`	Enable additional, stricter warnings
`-Werror`	Treat warnings as errors
`-Wpedantic`	Enforce strict ISO C compliance
`-Wshadow`	Warn if a local variable hides another variable
`-Wconversion`	Warn about implicit type conversions
`-Wunused`	Warn about unused variables or functions

Example:

```
gcc -Wall -Wextra -Werror main.c -o main
```

If your code has:

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

You'll get:

```
warning: 'x' is used uninitialized
```

With `-Werror`, that warning becomes a build-stopping error, a good habit for clean codebases.

Step 3. Debugging Flags

Debugging information allows tools like `gdb` or `lldb` to map machine code back to your C source.

Flag	Description
`-g`	Include debug symbols (file names, line numbers)
`-ggdb`	Include GNU-specific symbols for gdb
`-O0`	Disable optimization (makes debugging easier)

Example:

```c
gcc -g -O0 main.c -o main
```

Now run:

```c
gdb ./main
```

You'll be able to inspect variables and step through source lines.

Step 4. Optimization Levels

Optimization tells the compiler how aggressively to transform your code for speed or size.

Flag	Description
`-O0`	No optimization (fast compile, easy to debug)
`-O1`	Basic optimization
`-O2`	General speed optimization (default for most builds)

Flag	Description
`-O3`	Aggressive optimization (may increase size)
`-Os`	Optimize for size
`-Ofast`	Ignore strict standards for speed (dangerous)

Example:

```
gcc -O2 main.c -o main
```

Compare sizes:

```
gcc -O0 main.c -o slow
gcc -O2 main.c -o fast
ls -lh slow fast
```

`fast` will be smaller and run faster, the compiler reorders code, inlines functions, and removes dead logic.

Step 5. Profiling and Instrumentation Flags

Profiling helps measure which parts of your program consume the most CPU time.

Flag	Purpose
`-pg`	Generate profiling data for `gprof`
`-fprofile-generate` / `-fprofile-use`	Use profile-guided optimization (PGO)
`-ftime-report`	Show how long each compilation phase took

Example:

```
gcc -pg main.c -o main
./main
gprof main gmon.out > report.txt
```

Step 6. Standards Compliance Flags

The C language evolves, you can specify which version to follow.

Flag	Meaning
`-std=c89`	ANSI C (1989)
`-std=c99`	Modern C with `inline`, `bool`, `// comments`
`-std=c11`	Adds `_Generic`, `_Thread_local`, safer atomics
`-std=c17`	Minor cleanup
`-std=c23`	Latest (adds `typeof`, safer macros, etc.)

Example:

```
gcc -std=c99 -Wall main.c -o main
```

Always choose a consistent standard for your project.

Step 7. Platform and Architecture Flags

Flag	Description
`-m32` / `-m64`	Compile for 32-bit or 64-bit architecture
`-march=native`	Optimize for the host CPU
`-fPIC`	Position-independent code (required for shared libraries)
`-static`	Fully static linking
`-DNAME=value`	Define a macro (same as `#define` in code)

Example:

```
gcc -DDEBUG=1 -O2 -m64 main.c -o main
```

Step 8. Linking Flags

Flag	Description
`-L<dir>`	Add library search path
`-l<name>`	Link with library (e.g., `-lm` for math)
`-static`	Force static linking
`-shared`	Build a shared library
`-rpath <dir>`	Add runtime library search path

Example:

```
gcc main.o -L. -lmylib -Wl,-rpath=. -o app
```

`-Wl,` passes options directly to the linker (`ld`).

Tiny Code: Debug and Release Builds

Makefile

```
CC = gcc
CFLAGS = -Wall -std=c99
DEBUG_FLAGS = -g -O0 -DDEBUG
RELEASE_FLAGS = -O2 -DNDEBUG
SRC = main.c util.c
OBJ = $(SRC:.c=.o)
TARGET = app

debug: CFLAGS += $(DEBUG_FLAGS)
debug: $(TARGET)

release: CFLAGS += $(RELEASE_FLAGS)
release: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) -o $(TARGET)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

Run:

```
make debug
make release
```

- Debug build: full symbols, no optimization.
- Release build: optimized, no debugging info.

Step 9. Sanitizers (Runtime Safety Tools)

Modern compilers include built-in **sanitizers** to detect memory and thread errors:

Flag	Detects
`-fsanitize=address`	Memory leaks, buffer overflows
`-fsanitize=undefined`	Undefined behavior
`-fsanitize=thread`	Data races in multithreaded code

Example:

```
gcc -g -fsanitize=address main.c -o main
./main
```

If your code writes past an array boundary, you'll get an instant, readable report, no guessing.

Step 10. Combining Flags

Typical **development build:**

```
gcc -Wall -Wextra -Werror -g -O0 -std=c99 main.c -o debug_app
```

Typical **release build:**

```
gcc -O2 -march=native -flto -DNDEBUG main.c -o fast_app
```

- `-flto` enables **link-time optimization** (LTO), optimizing across files.
- `-DNDEBUG` disables `assert()` calls.

Why It Matters

Compiler flags are how professionals control:

- **Safety** (warnings, sanitizers)
- **Performance** (optimization levels)
- **Debuggability** (symbols and checks)
- **Portability** (standards and architectures)

Mastering them gives you precise control over how your code behaves, builds, and performs, essential for reliable systems programming.

Try It Yourself

1. Compile the same program with `-O0`, `-O2`, and `-O3`, time each run.
2. Add `-fsanitize=address` and find hidden memory bugs.
3. Compare binary sizes between `-g` and `-s`.
4. Add `-Wall -Wextra -Werror` to your Makefile and fix every warning.
5. Explore `gcc --help=optimizers` to see all available optimization passes.

Next, you'll peek **inside the object file itself**, learning what's stored inside `.o` binaries and how the linker stitches them together to form a complete executable.

59. Understanding the Object File

By now, you've seen `.o` files appear in every build step, the intermediate products between source and executable. But what exactly is *inside* them?

Object files are the compiler's way of packaging machine code, symbol tables, and metadata, ready for the linker to assemble into a final program. Understanding object files helps you debug linking errors, inspect performance, and even reverse-engineer compiled code.

Step 1. The Lifecycle Recap

The compilation pipeline looks like this:

```
source.c  > preprocessor  > compiler  > assembler  > linker
     |                          |            |          |
     |                       hello.s       hello.o    a.out
```

Your `.c` file becomes a `.o` file after **assembly**, but before **linking**.

Each `.o` is self-contained, it knows what it defines, what it needs, and where its code lives in memory.

Step 2. Object File Formats

Different operating systems use different binary formats:

OS	Format	Typical Extension
Linux	ELF (Executable and Linkable Format)	`.o`, `.so`, executable
macOS	Mach-O	`.o`, `.dylib`
Windows	COFF/PE	`.obj`, `.dll`, `.exe`

OS	Format	Typical Extension

On Linux, ELF dominates, so we'll use it as our reference.

Step 3. Sections Inside an Object File

Run:

```
gcc -c main.c -o main.o
readelf -S main.o
```

You'll see output like:

```
[ 1] .text     PROGBITS  code and functions
[ 2] .data     PROGBITS  initialized global variables
[ 3] .bss      NOBITS    uninitialized globals
[ 4] .rodata   PROGBITS  constants (e.g., string literals)
[ 5] .symtab   SYMTAB    symbol table
[ 6] .strtab   STRTAB    string table
[ 7] .rel.text RELA      relocation info
```

Section	Contents
`.text`	Compiled machine code (functions)
`.data`	Global variables with initial values
`.bss`	Global variables without initial values
`.rodata`	Constants, `const` variables, string literals
`.symtab`	Symbol table: function and variable metadata
`.rel*`	Relocation info, how to connect this file to others

Step 4. Inspecting Symbols

Every `.o` file contains **symbols** that describe its functions and variables. List them:

```
nm main.o
```

Output:

```
0000000000000000 T main
                 U printf
```

Symbol	Meaning
`T`	Defined in the text (code) section
`U`	Undefined, must be provided by another file or library
`D`	Defined in data section
`B`	Defined in bss section
`R`	Defined in read-only data
`W`	Weak symbol (can be overridden)

Here, `main` is defined, `printf` is undefined, meaning the linker must find it in the C standard library.

Tiny Code: Inspecting a Multi-File Example

math.c

```c
int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }
```

main.c

```c
#include <stdio.h>
int add(int, int);
int mul(int, int);

int main(void) {
    printf("%d\n", add(2, 3) * mul(1, 4));
    return 0;
}
```

Compile but don't link:

```
gcc -c main.c
gcc -c math.c
```

Inspect:

```
nm main.o
```

```
U add
U mul
U printf
T main
```

`U` means *undefined*, the linker must resolve these.

Now inspect `math.o`:

```
T add
T mul
```

These are *definitions* that will satisfy the linker.

Link and inspect the result:

```
gcc main.o math.o -o app
nm app | grep add
```

Now you'll see:

```
0000000000401136 T add
```

The linker relocated `add()` into its final address.

Step 5. Symbol Visibility

By default, every function and global variable has **external linkage**, visible to the linker.

Use `static` to limit visibility to the current file:

```
static int hidden_func(void) { return 42; }
```

Now `nm` will not list it as an exported symbol. This keeps your binary clean and prevents name collisions across files.

Step 6. Inspecting Relocations

Object files can't know final addresses yet, so they store **relocation entries**: placeholders for addresses that the linker must fill later.

Check them:

```
readelf -r main.o
```

Output:

```
Relocation section '.rela.text':
0000000000000010 R_X86_64_PC32 printf-0x4
```

This tells the linker:

> "When linking, replace this placeholder with the address of `printf()`."

Step 7. Disassemble the Code

You can see the actual machine instructions:

```
objdump -d main.o
```

Output snippet:

```
0000000000000000 <main>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: b8 00 00 00 00        mov    $0x0,%eax
```

Each instruction corresponds to compiled C code. This is how you verify optimizations, inspect inlining, or study generated assembly.

Step 8. Mixing Object Files

Because `.o` files contain clear symbol metadata, you can mix object files from different languages, for example, C and assembly.

sum.s

```
.globl sum
sum:
    addq %rsi, %rdi
    movq %rdi, %rax
    ret
```

Compile and link:

```
as sum.s -o sum.o
gcc main.c sum.o -o app
```

This is how C integrates cleanly with low-level code.

Step 9. Object File Size and Contents

Check file size:

```
size main.o
```

Output:

```
text    data     bss     dec     hex filename
45      4        0       49      31  main.o
```

- `text` → code size
- `data` → initialized global variables
- `bss` → uninitialized variables

Step 10. Tiny Code: Investigate Everything

```
gcc -c hello.c -o hello.o
readelf -h hello.o          # ELF header
readelf -S hello.o          # Sections
readelf -s hello.o | head   # Symbol table
readelf -r hello.o          # Relocations
objdump -d hello.o | head   # Disassembly
```

You'll get a complete low-level view of how your C code looks to the compiler.

Why It Matters

Understanding object files helps you:

- Debug linking errors and symbol conflicts
- Inspect compiler optimizations
- Integrate C with assembly
- Build static and shared libraries manually
- See what's actually inside the binary

In systems programming, this insight separates code *users* from code *engineers*.

Try It Yourself

1. Create two `.o` files that depend on each other and inspect their undefined symbols.
2. Use `readelf -S` to compare `.text`, `.data`, and `.bss` for different programs.
3. Add a global variable and see how it appears in `.data` or `.bss`.
4. Mark a function as `static` and confirm it disappears from `nm` output.
5. Compile with `-O2` and observe changes in disassembly with `objdump -d`.

Next, you'll complete Chapter 6 by building your own **Makefile-based compilation pipeline** from scratch, writing every stage explicitly to transform `.c` files into `.o`, `.a`, and `.so` artifacts just like a real compiler toolchain.

60. Practice: Write Your Own Makefile

Now that you understand how the C build process works, preprocessing, compiling, linking, and libraries, it's time to **tie everything together** with your own **Makefile**.

`make` is one of the oldest and most powerful automation tools in C development. It watches file timestamps, builds only what has changed, and lets you define build rules in a concise way.

By writing your own Makefile, you'll automate your entire compilation workflow like a professional.

Step 1. Create the Project

Let's build a small multi-file project:

```
project/
   Makefile
   main.c
   math.c
   math.h
   string_utils.c
```

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", mul(2, 3));
    return 0;
}
```

math.c

```c
#include "math.h"

int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }
```

math.h

```c
#ifndef MATH_H
#define MATH_H

int add(int a, int b);
int mul(int a, int b);

#endif
```

Step 2. Write the Simplest Makefile

Makefile

```
main: main.c math.c
    gcc main.c math.c -o main
```

Run:

```
make
./main
```

Output:

```
2 + 3 = 5
2 * 3 = 6
```

This works, but `make` will rebuild everything every time, even if only one file changed.

Let's make it smarter.

Step 3. Split the Compilation Steps

Separate compilation into object files:

```
CC = gcc
CFLAGS = -Wall -std=c99

main: main.o math.o
    $(CC) $(CFLAGS) main.o math.o -o main

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 main
```

Now when you run `make`, it builds `.o` files only once, and recompiles only what changed.

Test it:

```
make
touch math.c
make
```

You'll see that only `math.o` is rebuilt.

Step 4. Add Automatic Dependency Handling

Use **pattern rules** to avoid repeating commands for every `.c` file:

```
CC = gcc
CFLAGS = -Wall -std=c99
OBJS = main.o math.o string_utils.o
TARGET = app

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $(TARGET)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)
```

`$<` means "the first dependency" (like `main.c`). `$@` means "the target" (like `main.o`).

Now the Makefile works for any `.c` file automatically.

Step 5. Add Debug and Release Targets

Real projects have multiple build modes:

```
CC = gcc
CFLAGS = -Wall -std=c99
DEBUG_FLAGS = -g -O0 -DDEBUG
RELEASE_FLAGS = -O2 -DNDEBUG
OBJS = main.o math.o
TARGET = app

.PHONY: all clean debug release
```

```
all: release

debug: CFLAGS += $(DEBUG_FLAGS)
debug: $(TARGET)

release: CFLAGS += $(RELEASE_FLAGS)
release: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $(TARGET)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)
```

Run:

```
make debug
./app
```

Then:

```
make clean
make release
```

The debug build has symbols for `gdb`; the release build is optimized.

Step 6. Add Static and Shared Library Targets

Add these to your Makefile:

```
libmylib.a: math.o
    ar rcs libmylib.a math.o

libmylib.so: math.o
    $(CC) -shared -o libmylib.so math.o
```

Now you can build a reusable library:

```
make libmylib.a
make libmylib.so
```

Step 7. Add Installation and Help

```
install:
    cp app /usr/local/bin/

help:
    @echo "make [target]"
    @echo "Targets: all, debug, release, clean, install, libmylib.a, libmylib.so"
```

The `@` suppresses command echoing, only prints your messages.

Run:

```
make help
```

Step 8. Use Variables for Paths and Options

Clean up your Makefile by grouping related flags:

```
CC = gcc
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
CFLAGS = -Wall -Wextra -std=c99
LDFLAGS = -lm
TARGET = app

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) -o $(TARGET) $(LDFLAGS)

clean:
    rm -f $(OBJ) $(TARGET)
```

`wildcard` and `patsubst` let you automatically include new `.c` files as the project grows.

Tiny Code: Final Polished Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -O2
LDFLAGS = -lm
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
TARGET = app

all: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) -o $(TARGET) $(LDFLAGS)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

Run:

```
make
./app
```

This pattern is simple, robust, and scalable, the foundation of nearly all C build systems.

Step 9. Makefile Debugging

To see what commands `make` is running:

```
make VERBOSE=1
```

Or trace variable expansions:

```
make -p
```

Add `@echo "Building $@"` before commands for clarity.

Step 10. Why It Matters

A well-crafted Makefile:

- Automates your entire C build workflow
- Avoids recompiling unchanged files
- Scales to large multi-directory projects
- Makes your builds reproducible across systems

It's your first step toward professional build systems like **CMake**, **Meson**, or **Bazel**, all of which build on these principles.

Try It Yourself

1. Add a new `.c` file and watch your Makefile compile it automatically.
2. Add a `test` target that compiles and runs unit tests.
3. Add colorized output using `tput` or ANSI escapes.
4. Build both static and shared libraries in one run.
5. Convert your Makefile to use multiple directories (`src/`, `include/`, `build/`).

Congratulations! You've completed **Chapter 6 – Compilation and the Build Process**. You now understand how source becomes an executable, every stage from preprocessor to linker, and every tool in between.

Next, we'll step deeper into **Chapter 7: Working Close to the System**, where your programs start interacting directly with the operating system through system calls, processes, and files.

Chapter 7. Working Close to the System

61. System Calls and the Standard Library

When you write C programs that touch files, processes, or devices, you're talking to the **operating system**, not directly to hardware. That communication happens through **system calls**.

System calls are the lowest-level interface between user-space programs and the OS kernel. C's **standard library (libc)** is a thin layer of wrappers built on top of those calls, making them easier to use and more portable.

Let's explore how this works, and how to use system calls directly from your C code.

Step 1. What Is a System Call?

A **system call** (syscall) lets a program request a service from the OS kernel, like reading a file, creating a process, or allocating memory.

Examples:

- `read()`, `write()` – access files and devices
- `fork()`, `exec()` – create and manage processes
- `open()`, `close()` – handle file descriptors
- `mmap()` – map files into memory
- `socket()` – network communication

When you call a system function, control passes from **user space** to **kernel space**, then back again.

Step 2. The Role of libc

The C standard library (glibc, musl, etc.) provides *wrappers* around these system calls.

Example:

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("test.txt", "r");
    if (!f) {
        perror("fopen failed");
        return 1;
    }
    fclose(f);
}
```

Under the hood, `fopen()` eventually calls `open()`, a system call defined in `<fcntl.h>`. You can call it directly too.

Step 3. Calling System Calls Directly

Here's the same operation without stdio helpers:

```c
#include <fcntl.h>     // open
#include <unistd.h>    // read, write, close
#include <stdio.h>     // perror

int main(void) {
    int fd = open("test.txt", O_RDONLY);
    if (fd == -1) {
        perror("open failed");
        return 1;
    }

    char buf[128];
    ssize_t n = read(fd, buf, sizeof(buf) - 1);
    if (n >= 0) {
        buf[n] = '\0';
        write(STDOUT_FILENO, buf, n);
    }
    close(fd);
}
```

Compile and run:

```
gcc sysread.c -o sysread
./sysread
```

This prints the first 128 bytes of `test.txt` directly using system calls, no `fopen()` or `printf()` involved.

Tiny Code: Minimal System Call Example

Let's drop even the C library and use a **raw syscall** interface.

```
#include <unistd.h>

int main(void) {
    const char msg[] = "Hello via system call\n";
    write(1, msg, sizeof(msg) - 1);  // 1 = STDOUT
    _exit(0);
}
```

Compile:

```
gcc -nostdlib -static syshello.c -o syshello
```

Run:

```
Hello via system call
```

You just executed a system call directly, bypassing the standard library entirely.

Step 4. File Descriptors

System calls like `read()` and `write()` work with **file descriptors**, small integer handles managed by the OS.

Descriptor	Meaning
0	Standard input (`stdin`)
1	Standard output (`stdout`)
2	Standard error (`stderr`)

Every open file, socket, or pipe has a unique descriptor.

Example:

```
write(1, "Output to stdout\n", 17);
write(2, "Output to stderr\n", 17);
```

Step 5. Inspecting System Calls

You can watch your program's system calls using `strace`:

```
strace ./sysread
```

Example output:

```
open("test.txt", O_RDONLY) = 3
read(3, "Hello World\n", 12) = 12
write(1, "Hello World\n", 12) = 12
close(3) = 0
```

This shows the real kernel-level operations, a great debugging and learning tool.

Step 6. Return Values and Errors

System calls usually return:

- **0** → success (bytes read, process ID, etc.)
- **-1** → error (with `errno` set to an error code)

Example:

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

int fd = open("missing.txt", O_RDONLY);
if (fd == -1)
    fprintf(stderr, "Error: %s\n", strerror(errno));
```

Common error codes:

Code	Meaning
ENOENT	File not found
EACCES	Permission denied
EBADF	Invalid descriptor
EINTR	Interrupted system call

Step 7. Mixing System Calls and stdio

You can combine both layers safely, just don't mix them on the *same* file descriptor.

Example (safe):

```
FILE *f = fopen("log.txt", "w");
write(STDOUT_FILENO, "Console log\n", 12);
fprintf(f, "File log\n");
fclose(f);
```

Example (unsafe):

```
FILE *f = fopen("data.txt", "w");
write(fileno(f), "mixed output\n", 13);  // may confuse buffering
```

Step 8. System Call Wrappers

The Linux kernel provides hundreds of system calls. You can call most through `<unistd.h>`, but for rare ones, you can use `syscall()`:

```
#include <sys/syscall.h>
#include <unistd.h>

int main(void) {
    syscall(SYS_write, 1, "Hello syscall\n", 14);
    return 0;
}
```

Step 9. Viewing Available Syscalls

Check all available system calls for your platform:

```
man 2 intro
```

or

```
ausyscall --dump | head
```

You'll see a list like:

```
read, write, open, close, stat, fork, execve, mmap, ...
```

Step 10. Why It Matters

System calls are the *foundation* of everything in C and Unix-like systems.

They give your program direct access to:

- Files and devices
- Processes and signals
- Memory and networking
- Time and environment

Learning to use them directly is essential for understanding how higher-level abstractions (like `stdio`, `pthreads`, or sockets) are built.

Try It Yourself

1. Use `strace` on `ls` or `cat` to see how system calls drive them.
2. Replace a `fopen()`/`fread()` pair with direct `open()` and `read()` calls.
3. Write a tiny file copier using only `open`, `read`, `write`, and `close`.
4. Experiment with invalid file descriptors and print out `errno`.
5. Build a version of `echo` that uses only raw system calls.

Next, you'll take this a step further: learning how to **create and manage processes** with `fork()` and `exec()`, the heart of Unix multitasking.

62. Process Creation (fork, exec, wait)

Every program in a Unix-like system runs inside a **process**, a running instance of a program with its own memory, file descriptors, and environment. When you type `ls` or `cat`, the shell doesn't just "jump" into those programs. It **creates a new process** to run them.

In C, you can do exactly the same thing, create new processes, run other programs, and synchronize them.

This section teaches you how **`fork()`**, **`exec()`**, and **`wait()`** work together, the three essential building blocks of process control.

Step 1. The Idea of a Process

When your program starts, it runs as **one process**, with:

- a **PID** (process ID),
- its own **memory space**,
- **file descriptors** (stdin, stdout, stderr),
- and environment variables.

You can check your own PID:

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

int main(void) {
    printf("My PID is %d\n", getpid());
    return 0;
}
```

Compile and run:

```
gcc pid.c -o pid
./pid
```

Output:

```
My PID is 5231
```

Step 2. Creating a New Process with `fork()`

`fork()` creates a **new process** by duplicating the current one.

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

int main(void) {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    }
    if (pid == 0) {
        printf("Child process! PID = %d\n", getpid());
    } else {
        printf("Parent process! PID = %d, child PID = %d\n", getpid(), pid);
    }
    return 0;
}
```

Compile and run:

```
gcc fork_demo.c -o fork_demo
./fork_demo
```

Example output:

```
Parent process! PID = 5231, child PID = 5232
Child process! PID = 5232
```

- `fork()` returns **0** in the child process
- `fork()` returns **child PID** in the parent process
- Both processes continue executing from the same point

Step 3. Independent Memory After `fork()`

Each process gets a **copy** of the parent's memory. Changing a variable in the child doesn't affect the parent.

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

int main(void) {
    int counter = 0;
    pid_t pid = fork();

    if (pid == 0) {
        counter += 10;
        printf("Child counter: %d\n", counter);
    } else {
        counter += 1;
        printf("Parent counter: %d\n", counter);
    }
    return 0;
}
```

Output:

```
Parent counter: 1
Child counter: 10
```

Each process has its own copy of `counter`.

Step 4. Replacing a Process Image with `exec()`

After `fork()`, the child can **replace itself** with a new program using `exec()`.

There are multiple versions:

- `execl(path, arg0, arg1, ..., NULL)`
- `execv(path, argv[])`
- `execlp(file, arg0, arg1, ..., NULL)`, searches `$PATH`
- `execvp(file, argv[])`, most commonly used

Example:

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

int main(void) {
    printf("Before exec\n");
```

```
    execlp("ls", "ls", "-l", NULL);
    printf("This will not run if exec succeeds\n");
    return 0;
}
```

Compile and run:

```
gcc exec_demo.c -o exec_demo
./exec_demo
```

Output:

```
Before exec
(total listing from `ls`)
```

After `exec`, the current process image is replaced, the PID stays the same, but the program running inside changes.

Step 5. Combining `fork()` and `exec()`

This is how your shell launches commands.

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

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        execlp("echo", "echo", "Hello from child", NULL);
        perror("exec failed");
    } else {
        printf("Parent is waiting...\n");
    }
    return 0;
}
```

Output:

```
Parent is waiting...
Hello from child
```

The parent forks; the child replaces itself with `echo`.

Step 6. Waiting for the Child (`wait()` and `waitpid()`)

The parent can **wait** for its child process to finish.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child running\n");
        execlp("sleep", "sleep", "1", NULL);
    } else {
        printf("Parent waiting for child...\n");
        wait(NULL);
        printf("Child finished\n");
    }
    return 0;
}
```

Compile and run:

```
gcc wait_demo.c -o wait_demo
./wait_demo
```

Output:

```
Parent waiting for child...
Child running
Child finished
```

Step 7. Checking Exit Status

You can get the child's **exit code** using `waitpid()`.

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
    pid_t pid = fork();
    if (pid == 0) {
        _exit(42);  // child exits with status 42
    } else {
        int status;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status))
            printf("Child exited with code %d\n", WEXITSTATUS(status));
    }
}
```

Output:

```
Child exited with code 42
```

Step 8. Multiple Children

You can spawn multiple processes and wait for them all:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    for (int i = 0; i < 3; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            printf("Child %d PID %d\n", i, getpid());
            _exit(0);
        }
    }
```

```
    for (int i = 0; i < 3; i++)
        wait(NULL);
    printf("All children done\n");
}
```

Output:

```
Child 0 PID 5321
Child 1 PID 5322
Child 2 PID 5323
All children done
```

Step 9. Orphan and Zombie Processes

If the parent doesn't call `wait()`, the child becomes a **zombie** (terminated, but still in process table). If the parent terminates before the child, the child becomes an **orphan** and gets adopted by `init` (PID 1).

Run this:

```
ps -l | grep Z
```

You'll see zombie processes marked with a `Z`.

Tiny Code: Minimal Shell Launcher

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    char *argv[] = {"ls", "-1", NULL};
    pid_t pid = fork();

    if (pid == 0) {
        execvp(argv[0], argv);
        perror("exec failed");
    } else {
        wait(NULL);
```

```
        printf("Command finished\n");
    }
}
```

Compile and run:

```
gcc mini_shell.c -o mini_shell
./mini_shell
```

Output:

```
(file listing)
Command finished
```

Step 10. Why It Matters

`fork()`, `exec()`, and `wait()` form the **core process model** of Unix. Every command-line program, daemon, and service uses these under the hood.

They let you:

- Launch other programs
- Build parallel workers
- Implement your own shell
- Control process trees and jobs

Once you understand these, you're ready to dive into **inter-process communication**, making your processes talk via **pipes and redirection**.

Try It Yourself

1. Write a program that forks two children, one runs `date`, one runs `whoami`.
2. Modify it to wait for both children to finish.
3. Create a program that forks a child, but the parent exits immediately (observe orphan adoption).
4. Write your own `run(command)` function using `fork()`, `execvp()`, and `waitpid()`.
5. Combine all this into a tiny shell that accepts commands and executes them interactively.

Next, you'll learn how these processes can **communicate and share data**, using **file descriptors, pipes, and redirection** in the next section.

63. File Descriptors and open/read/write

Now that you can create and manage processes, let's explore how those processes **communicate with files, devices, and even each other**, through **file descriptors**.

File descriptors (FDs) are one of the simplest yet most powerful abstractions in Unix and C. Everything, files, pipes, sockets, terminals, is represented by a small integer handle. Once you understand how to open, read, write, and close file descriptors, you can interact with any I/O system on a Unix machine.

Step 1. What Is a File Descriptor?

A **file descriptor** is an integer that identifies an open resource in your process. Every process starts with three open descriptors by default:

FD	Symbolic Name	Description
0	`STDIN_FILENO`	Standard input (keyboard)
1	`STDOUT_FILENO`	Standard output (screen)
2	`STDERR_FILENO`	Standard error (screen)

Each time you open a file, socket, or pipe, the kernel gives you the **lowest unused FD**.

Step 2. Opening Files with `open()`

You can open files directly using the **system call layer**, instead of `fopen()` from stdio.

```c
#include <fcntl.h>     // open
#include <unistd.h>    // close, read, write
#include <stdio.h>     // perror

int main(void) {
    int fd = open("data.txt", O_RDONLY);
    if (fd == -1) {
        perror("open failed");
        return 1;
    }

    printf("File descriptor: %d\n", fd);
    close(fd);
}
```

Compile and run:

```
gcc open_demo.c -o open_demo
./open_demo
```

Output example:

```
File descriptor: 3
```

Step 3. Reading from a File

`read(fd, buffer, size)` reads raw bytes into a memory buffer.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int fd = open("data.txt", O_RDONLY);
    if (fd == -1) return 1;

    char buf[64];
    ssize_t n = read(fd, buf, sizeof(buf) - 1);
    if (n > 0) {
        buf[n] = '\0';
        printf("Read %zd bytes: %s\n", n, buf);
    }

    close(fd);
}
```

Output:

```
Read 12 bytes: Hello world
```

Step 4. Writing to a File

`write(fd, buffer, size)` writes raw bytes from memory to a file descriptor.

```c
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(void) {
    int fd = open("out.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    const char msg[] = "Writing from C using write()\n";
    write(fd, msg, strlen(msg));
    close(fd);
}
```

This overwrites `out.txt` with your message. Flags:

- `O_WRONLY` → write-only
- `O_CREAT` → create if it doesn't exist
- `O_TRUNC` → truncate (clear) existing contents

The final argument `0644` sets Unix permissions:

- Owner can read/write,
- Group/others can read.

Step 5. Append and Non-blocking Modes

You can combine flags using bitwise OR:

```c
int fd = open("log.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
```

`O_APPEND` moves the file offset to the end before every write, ideal for logs.

You can also open files as **non-blocking**:

```c
int fd = open("pipe", O_RDONLY | O_NONBLOCK);
```

Useful for I/O on sockets or named pipes.

Step 6. Duplicating Descriptors

You can duplicate an FD using `dup()` or `dup2()`. This is how **redirection** works (`>` in shells).

Example:

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(void) {
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(fd, STDOUT_FILENO);  // redirect stdout to file
    close(fd);

    printf("This goes into output.txt!\n");
}
```

After `dup2`, everything printed to stdout goes to `output.txt`.

Run and inspect:

```
./redir_demo
cat output.txt
```

Output:

```
This goes into output.txt!
```

Step 7. Offsets and `lseek()`

You can move around inside a file using `lseek(fd, offset, whence)`.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int fd = open("data.txt", O_RDONLY);
    lseek(fd, 5, SEEK_SET); // move to byte 5
    char buf[16];
    read(fd, buf, 10);
```

```
    buf[10] = '\0';
    printf("Chunk: %s\n", buf);
    close(fd);
}
```

`whence` can be:

- `SEEK_SET` (from start)
- `SEEK_CUR` (from current)
- `SEEK_END` (from end)

Step 8. Error Checking and Return Values

All system calls return:

- A nonnegative value → success
- `-1` → error (check `errno`)

Example:

```
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd = open("missing.txt", O_RDONLY);
    if (fd == -1)
        fprintf(stderr, "Error: %s\n", strerror(errno));
}
```

Output:

```
Error: No such file or directory
```

Tiny Code: Copy File Using System Calls

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int src = open("source.txt", O_RDONLY);
    int dst = open("copy.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if (src == -1 || dst == -1) {
        perror("open failed");
        return 1;
    }

    char buf[256];
    ssize_t n;
    while ((n = read(src, buf, sizeof(buf))) > 0)
        write(dst, buf, n);

    close(src);
    close(dst);
    return 0;
}
```

Compile and run:

```c
gcc copy.c -o copy
./copy
```

Now `copy.txt` is identical to `source.txt`, using pure syscalls.

Step 9. Reading from STDIN and Writing to STDOUT

You can use `read(0, ...)` and `write(1, ...)` directly for console I/O.

```c
#include <unistd.h>

int main(void) {
    char buf[64];
    ssize_t n = read(STDIN_FILENO, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, n);
}
```

Run:

```
./echo_demo
hello world
```

Output:

```
hello world
```

That's the essence of every shell command.

Step 10. Why It Matters

File descriptors unify I/O across everything:

- Regular files
- Pipes and sockets
- Devices and terminals

They let you control exactly how data flows in and out of your program, a foundation for system tools, servers, and OS-level programming.

Once you understand these primitives, you can build your own versions of tools like `cat`, `tee`, and even simple shells.

Try It Yourself

1. Write a mini `cat` clone using `read()` and `write()`.
2. Use `dup2()` to redirect both stdout and stderr to a file.
3. Add error messages using `perror()` and handle `EINTR`.
4. Use `lseek()` to skip the first N bytes of a file before printing.
5. Implement a simple file appender with `O_APPEND`.

Next, you'll use these file descriptors to make **processes communicate**, building **pipes and redirection**, the same mechanisms shells use to connect commands like `ls | grep`.

64. Pipes and Redirection

Now that you can read and write with file descriptors, you can connect **two processes** so that one's output becomes the other's input, just like `ls | grep c` in a shell.

This magic happens through **pipes**, one of Unix's simplest and most elegant inter-process communication (IPC) mechanisms.

Step 1. What Is a Pipe?

A **pipe** is a unidirectional data channel between two file descriptors, one for reading, one for writing.

In the shell:

```
ls | grep main
```

is equivalent to:

- Process A (`ls`) writes into the pipe.
- Process B (`grep`) reads from the pipe.

In C, you can do the same thing using `pipe()` and `fork()`.

Step 2. Creating a Pipe

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

int main(void) {
    int fds[2];
    if (pipe(fds) == -1) {
        perror("pipe failed");
        return 1;
    }
    printf("Read end: %d, Write end: %d\n", fds[0], fds[1]);
    return 0;
}
```

Compile and run:

```
gcc pipe_demo.c -o pipe_demo
./pipe_demo
```

Output example:

```
Read end: 3, Write end: 4
```

You now have two connected file descriptors:

- `fds[0]`: read end
- `fds[1]`: write end

Whatever you write into `fds[1]` can be read from `fds[0]`.

Step 3. Writing and Reading Through a Pipe

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

int main(void) {
    int fds[2];
    pipe(fds);

    const char msg[] = "hello through pipe";
    write(fds[1], msg, strlen(msg));

    char buf[64];
    ssize_t n = read(fds[0], buf, sizeof(buf) - 1);
    buf[n] = '\0';
    printf("Received: %s\n", buf);

    close(fds[0]);
    close(fds[1]);
}
```

Output:

```
Received: hello through pipe
```

You've just communicated through memory between two file descriptors, no files, no network.

Step 4. Pipes Between Parent and Child

Here's where it gets powerful: **pipes can connect processes**.

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

int main(void) {
    int fds[2];
    pipe(fds);

    pid_t pid = fork();
    if (pid == 0) {
        // Child
        close(fds[1]); // close write end
        char buf[64];
        ssize_t n = read(fds[0], buf, sizeof(buf) - 1);
        buf[n] = '\0';
        printf("Child got: %s\n", buf);
    } else {
        // Parent
        close(fds[0]); // close read end
        const char msg[] = "Hi from parent!";
        write(fds[1], msg, strlen(msg));
        close(fds[1]);
    }
}
```

Output:

```
Child got: Hi from parent!
```

Parent writes, child reads, a clean data channel between two processes.

Step 5. Redirecting STDIN/STDOUT to a Pipe

You can use `dup2()` to connect a pipe directly to standard input/output.

Example: connect parent's write end to child's stdin.

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

int main(void) {
```

```
    int fds[2];
    pipe(fds);

    pid_t pid = fork();

    if (pid == 0) {
        dup2(fds[0], STDIN_FILENO);
        close(fds[1]);
        execlp("wc", "wc", "-w", NULL);
    } else {
        close(fds[0]);
        write(fds[1], "Hello from parent\nThis is a pipe test\n", 39);
        close(fds[1]);
    }
}
```

Output:

```
6
```

Here's what happens:

- Parent writes data into the pipe.
- Child's stdin is connected to that pipe.
- `wc -w` counts the words received.

This is exactly how the shell implements pipelines like `echo "hi" | wc -w`.

Step 6. Chaining Multiple Commands

You can chain multiple commands by creating multiple pipes and connecting them in series.

Example concept:

```
cat file.txt | grep "error" | wc -l
```

Each process reads from the previous pipe and writes to the next, the shell's fundamental design.

You can implement the same concept in C by:

1. Creating a pipe between each process pair.
2. Forking a new process for each command.
3. Redirecting its stdin/stdout via `dup2()`.

Step 7. Named Pipes (FIFOs)

Pipes normally exist only between related processes. To share data between unrelated programs, you can use **named pipes** (FIFOs).

Create one:

```
mkfifo mypipe
```

Then in one terminal:

```
cat > mypipe
```

And in another:

```
cat < mypipe
```

Data flows through the named pipe like a file.

In C:

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main(void) {
    mkfifo("mypipe", 0666);
    int fd = open("mypipe", O_WRONLY);
    write(fd, "Hello FIFO\n", 11);
    close(fd);
}
```

Step 8. Error Handling and EOF

If all write ends of a pipe are closed, `read()` returns `0`, indicating EOF.

```c
int fds[2];
pipe(fds);
close(fds[1]); // no writers
char buf[10];
ssize_t n = read(fds[0], buf, 10); // n == 0 => EOF
```

If you try to write after all readers are gone, you'll get `SIGPIPE`.

Tiny Code: Minimal Shell Pipeline

```c
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    int fds[2];
    pipe(fds);

    if (fork() == 0) {
        dup2(fds[1], STDOUT_FILENO);
        close(fds[0]);
        execlp("ls", "ls", NULL);
    }

    if (fork() == 0) {
        dup2(fds[0], STDIN_FILENO);
        close(fds[1]);
        execlp("wc", "wc", "-l", NULL);
    }

    close(fds[0]);
    close(fds[1]);
    wait(NULL);
    wait(NULL);
}
```

Run:

```
gcc pipe_chain.c -o pipe_chain
./pipe_chain
```

Output:

```
(number of files in directory)
```

You just recreated `ls | wc -l` in C.

Step 9. Combining Redirection and Files

You can redirect stdout or stderr to files the same way:

```
int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
dup2(fd, STDOUT_FILENO);
close(fd);
execlp("ls", "ls", NULL);
```

Now the output of `ls` goes to `output.txt`.

Step 10. Why It Matters

Pipes and redirection are the **heart of Unix philosophy**:

- Programs do one thing well.
- Communicate through text streams.
- Compose complex workflows by chaining simple tools.

Understanding how to implement pipes makes you capable of:

- Building your own shell
- Connecting processes dynamically
- Implementing inter-process communication safely

Try It Yourself

1. Recreate `ls | grep c` using two `fork()` calls and one pipe.
2. Build `cat file | wc -l`.
3. Implement a "tee" program that duplicates output to both stdout and a file.
4. Create a named pipe and write to it from one process, read from another.
5. Extend your shell to support pipelines of any length.

Next, you'll explore how processes **signal and interrupt each other**, using **signals** and **signal handlers**, a crucial concept for handling interrupts, timeouts, and graceful termination.

65. Signals and Signal Handlers

When you press **Ctrl+C** and your program stops, that's not magic, it's a **signal**. Signals are how the operating system tells your process that something important has happened.

They're asynchronous, lightweight messages from the kernel or other processes. Your C program can catch, ignore, or handle them, giving you full control over shutdowns, interrupts, and errors.

Step 1. What Is a Signal?

A **signal** is an integer code sent to a process to notify it of an event.

Signal	Meaning	Default Action
`SIGINT`	Interrupt (Ctrl+C)	Terminate
`SIGTERM`	Termination request	Terminate
`SIGKILL`	Forced kill (cannot catch)	Terminate immediately
`SIGSEGV`	Invalid memory access	Core dump
`SIGCHLD`	Child process exited	Ignore or handle
`SIGALRM`	Timer expired	Terminate
`SIGUSR1`, `SIGUSR2`	User-defined	Terminate (unless handled)

You can list all signals:

```
kill -l
```

Output example:

```
1) SIGHUP  2) SIGINT  3) SIGQUIT  9) SIGKILL  15) SIGTERM  ...
```

Step 2. Sending Signals

Any process can send a signal to another using the `kill()` system call.

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

int main(void) {
    pid_t pid = getpid();
```

```
    printf("My PID: %d\n", pid);
    pause();  // wait for signal
}
```

Run this in one terminal:

```
./signal_wait
```

Then in another:

```
kill -SIGUSR1 <pid>
```

The program will wake up from `pause()` and terminate (default behavior for SIGUSR1).

Step 3. Installing a Signal Handler

You can **override** the default action by installing a **handler function**.

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

void handle_sigint(int sig) {
    printf("\nCaught signal %d (SIGINT). Exiting gracefully.\n", sig);
    _exit(0);
}

int main(void) {
    signal(SIGINT, handle_sigint);
    while (1) {
        printf("Running... Press Ctrl+C to stop.\n");
        sleep(1);
    }
}
```

Compile and run:

```
gcc sigint_demo.c -o sigint_demo
./sigint_demo
```

Output:

```
Running... Press Ctrl+C to stop.
Running... Press Ctrl+C to stop.
^C
Caught signal 2 (SIGINT). Exiting gracefully.
```

Step 4. Using `sigaction()` (Modern API)

`signal()` is simple but inconsistent across systems. The **recommended** modern interface is `sigaction()`.

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int sig) {
    write(STDOUT_FILENO, "Caught signal\n", 14);
}

int main(void) {
    struct sigaction sa = {0};
    sa.sa_handler = handler;
    sigaction(SIGUSR1, &sa, NULL);

    printf("PID: %d\n", getpid());
    while (1) pause();
}
```

Send a signal:

```
kill -SIGUSR1 <pid>
```

Output:

```
Caught signal
```

Unlike `signal()`, this version is **reliable and reentrant-safe** (you can call only async-safe functions like `write()` inside handlers).

Step 5. Ignoring and Resetting Signals

You can **ignore** a signal:

```
signal(SIGINT, SIG_IGN);
```

Or **reset** to default behavior:

```
signal(SIGINT, SIG_DFL);
```

This can be useful if you don't want Ctrl+C to interrupt certain sections of code.

Step 6. Sending Signals to Other Processes

Example: parent signaling its child.

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

void child_handler(int sig) {
    printf("Child got signal %d\n", sig);
}

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        signal(SIGUSR1, child_handler);
        while (1) pause();
    } else {
        sleep(1);
        printf("Parent sending SIGUSR1\n");
        kill(pid, SIGUSR1);
        sleep(1);
        kill(pid, SIGTERM);
    }
}
```

Output:

```
Parent sending SIGUSR1
Child got signal 10
```

Step 7. Blocking and Unblocking Signals

Sometimes you want to delay signal handling. You can use `sigprocmask()` to block signals temporarily.

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);

    sigprocmask(SIG_BLOCK, &set, NULL);
    printf("SIGINT blocked for 5 seconds...\n");
    sleep(5);
    printf("Unblocking now.\n");

    sigprocmask(SIG_UNBLOCK, &set, NULL);
    while (1) pause();
}
```

Press Ctrl+C during the block, nothing happens. Once unblocked, it terminates normally.

Step 8. Timers with `alarm()` and `SIGALRM`

You can set a timer that sends a signal after a delay.

```c
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void handler(int sig) {
    printf("Timer expired!\n");
}

int main(void) {
    signal(SIGALRM, handler);
    alarm(3);  // after 3 seconds, send SIGALRM
    printf("Waiting...\n");
```

```
    pause();
}
```

Output:

```
Waiting...
Timer expired!
```

Step 9. Cleaning Up on Exit

Signals let you implement graceful cleanup (e.g., close files, delete temp files).

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

void cleanup(int sig) {
    printf("\nCleaning up before exit...\n");
    unlink("tempfile.tmp");
    _exit(0);
}

int main(void) {
    signal(SIGINT, cleanup);
    open("tempfile.tmp", O_CREAT | O_WRONLY, 0644);
    while (1) {
        printf("Running... (Ctrl+C to exit)\n");
        sleep(1);
    }
}
```

Now pressing Ctrl+C removes the temporary file safely.

Tiny Code: Graceful Shutdown Server

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

```c
volatile sig_atomic_t running = 1;

void stop(int sig) {
    running = 0;
}

int main(void) {
    signal(SIGINT, stop);
    printf("Server running. Press Ctrl+C to stop.\n");

    while (running) {
        printf("Handling request...\n");
        sleep(1);
    }

    printf("Server shutting down cleanly.\n");
}
```

Step 10. Why It Matters

Signals are essential for:

- Graceful termination (Ctrl+C)
- Timeouts and alarms
- Child process monitoring (`SIGCHLD`)
- Error handling (segmentation faults)
- Daemon and server control

Every real-world Unix program, from editors to web servers, depends on correct signal handling for stability.

Try It Yourself

1. Write a program that ignores SIGINT for 5 seconds, then restores default behavior.
2. Catch SIGTERM and print "Termination requested".
3. Make a parent send SIGUSR1 to its child every second.
4. Use `alarm()` to implement a timeout for user input.
5. Add a signal handler to your shell that cleans up child processes before exit.

Next, you'll learn how programs **share and map memory directly**, using `mmap()`, a system call that powers databases, shared memory, and file-backed data structures.

66. Memory Mapping (mmap)

In previous sections, you learned how to read and write files using `read()` and `write()`. Those system calls move data between **files** and **user-space buffers** in RAM.

But what if you could **map a file directly into memory**, and then treat it as part of your process's address space?

That's exactly what **memory mapping** (via `mmap`) does, it's faster, more flexible, and forms the backbone of databases, shared memory systems, and even virtual memory itself.

Step 1. What Is `mmap`?

`mmap()` maps a file or device into memory so you can access it directly, as if it were an array in RAM.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Parameter	Description
`addr`	Hint for mapping address (usually `NULL`)
`length`	Number of bytes to map
`prot`	Protection: `PROT_READ`, `PROT_WRITE`, etc.
`flags`	Type: `MAP_PRIVATE`, `MAP_SHARED`, etc.
`fd`	File descriptor to map
`offset`	Start offset in file (must be multiple of page size)

Step 2. Simple File Mapping Example

Let's map a file into memory and print its contents.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd = open("data.txt", O_RDONLY);
```

```
    if (fd == -1) { perror("open"); return 1; }

    struct stat st;
    fstat(fd, &st);

    char *data = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (data == MAP_FAILED) { perror("mmap"); return 1; }

    write(STDOUT_FILENO, data, st.st_size);
    munmap(data, st.st_size);
    close(fd);
}
```

Compile and run:

```
gcc mmap_read.c -o mmap_read
./mmap_read
```

This directly prints the file contents, no loops, no `read()` calls.

Step 3. Reading vs. Writing

If you want to modify a file through memory, you must open it read-write and use `PROT_WRITE`.

```
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>

int main(void) {
    int fd = open("memo.txt", O_RDWR | O_CREAT, 0666);
    ftruncate(fd, 64); // ensure file has enough size

    char *map = mmap(NULL, 64, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (map == MAP_FAILED) return 1;

    strcpy(map, "Hello, memory-mapped file!");
    msync(map, 64, MS_SYNC);

    munmap(map, 64);
```

```
    close(fd);
}
```

Open `memo.txt`, you'll see the written text instantly.

- `MAP_SHARED`: changes are written back to the file.
- `MAP_PRIVATE`: copy-on-write (changes visible only to this process).

Step 4. Anonymous Mappings

You can create memory that isn't tied to any file, purely in RAM.

```
#include <sys/mman.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    size_t len = 4096;
    int *arr = mmap(NULL, len, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (arr == MAP_FAILED) return 1;

    arr[0] = 1234;
    printf("arr[0] = %d\n", arr[0]);

    munmap(arr, len);
}
```

Output:

```
arr[0] = 1234
```

Anonymous mappings are commonly used for dynamic memory regions or shared memory between processes.

Step 5. Shared Memory Between Processes

You can use `MAP_SHARED` and `fork()` to let parent and child processes share the same mapped memory.

```c
#include <sys/mman.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int *shared = mmap(NULL, sizeof(int),
                       PROT_READ | PROT_WRITE,
                       MAP_SHARED | MAP_ANONYMOUS,
                       -1, 0);

    *shared = 0;
    pid_t pid = fork();

    if (pid == 0) {
        (*shared)++;
        printf("Child: shared = %d\n", *shared);
    } else {
        sleep(1);
        printf("Parent: shared = %d\n", *shared);
    }

    munmap(shared, sizeof(int));
}
```

Output:

```
Child: shared = 1
Parent: shared = 1
```

Both processes see the same memory, no pipes or sockets needed.

Step 6. Memory Protections

Use `PROT_*` flags to control access:

- `PROT_READ` → read allowed
- `PROT_WRITE` → write allowed
- `PROT_EXEC` → executable
- `PROT_NONE` → no access

You can change permissions later:

```
mprotect(ptr, len, PROT_READ);
```

This helps you simulate "read-only" data regions or test segmentation faults intentionally.

Step 7. Page Size and Alignment

Memory is mapped in units of **pages** (usually 4096 bytes). You can get your system's page size:

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

int main(void) {
    printf("Page size: %ld bytes\n", sysconf(_SC_PAGESIZE));
}
```

Offsets in `mmap` must be **aligned** to page size.

Step 8. Unmapping and Syncing

When you're done with a mapped region, always call:

```
munmap(addr, length);
```

If you modified the data:

```
msync(addr, length, MS_SYNC);
```

This ensures changes are written back to disk.

Step 9. Using `mmap` for Performance

Advantages over `read()` and `write()`:

- Avoids extra data copies between kernel and user space.
- The OS loads only the pages you touch (lazy loading).
- Efficient for random access to large files.

Databases, editors, and browsers (like SQLite, Vim, Chrome) rely heavily on `mmap` for performance.

Tiny Code: Count Lines in a Large File

```c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int fd = open("bigfile.txt", O_RDONLY);
    struct stat st;
    fstat(fd, &st);

    char *data = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    size_t lines = 0;

    for (size_t i = 0; i < st.st_size; i++)
        if (data[i] == '\n') lines++;

    printf("Lines: %zu\n", lines);
    munmap(data, st.st_size);
    close(fd);
}
```

Run it on a multi-GB file, it'll perform incredibly fast.

Step 10. Why It Matters

`mmap` opens a new world of **memory-driven file access**:

- Used by OSes to load executables, shared libs, and pages.
- Powers databases, compilers, and search engines.
- Enables shared memory between processes.
- Reduces I/O overhead for large files.

It's the bridge between **files** and **memory**, unifying two key abstractions in C and Unix.

Try It Yourself

1. Write a file and modify it in place using `mmap`.

2. Create shared memory between parent and child processes with `MAP_SHARED`.
3. Measure performance difference between `read()` and `mmap`.
4. Map only part of a file using an offset aligned to page size.
5. Implement a small in-memory key-value store backed by `mmap`.

Next, you'll explore how to work with **time and clocks in C**, retrieving system timestamps, measuring durations, and implementing timers with precision.

67. Time and Clock APIs

Time is one of the simplest things humans understand, and one of the trickiest things for computers to handle correctly. In C, time is represented in **seconds since the Unix epoch (Jan 1, 1970)**, and you can work with it at various levels: wall-clock time, process time, and high-precision timers.

Let's explore how to **get, format, and measure time** in C.

Step 1. The Basics: `time()`

The simplest way to get the current time is with the `time()` function.

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

int main(void) {
    time_t now = time(NULL);
    printf("Seconds since epoch: %ld\n", now);
}
```

Output:

```
Seconds since epoch: 1739709201
```

That's the number of seconds since **1970-01-01 00:00:00 UTC**.

Step 2. Converting to Human-Readable Format

You can convert `time_t` into a calendar date using `localtime()` or `gmtime()`.

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

int main(void) {
    time_t now = time(NULL);
    struct tm *t = localtime(&now);
    printf("Local time: %02d-%02d-%04d %02d:%02d:%02d\n",
           t->tm_mday, t->tm_mon + 1, t->tm_year + 1900,
           t->tm_hour, t->tm_min, t->tm_sec);
}
```

Output:

```
Local time: 16-10-2025 09:32:10
```

Step 3. Formatting Dates with `strftime()`

`strftime()` lets you format time into a string safely.

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

int main(void) {
    char buf[100];
    time_t now = time(NULL);
    struct tm *t = localtime(&now);
    strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", t);
    printf("Formatted: %s\n", buf);
}
```

Output:

```
Formatted: 2025-10-16 09:32:10
```

Step 4. Measuring Elapsed Time

To measure how long something takes, use `clock()` from `<time.h>`.

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

int main(void) {
    clock_t start = clock();
    for (volatile long i = 0; i < 100000000; i++);
    clock_t end = clock();
    double seconds = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Elapsed time: %.3f seconds\n", seconds);
}
```

Output:

```
Elapsed time: 0.520 seconds
```

`clock()` measures **CPU time**, not real elapsed time, so it excludes time spent waiting for I/O or sleeping.

Step 5. High-Resolution Timing with `clock_gettime()`

For precise measurements, use `clock_gettime()`.

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

int main(void) {
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (volatile long i = 0; i < 100000000; i++);

    clock_gettime(CLOCK_MONOTONIC, &end);

    double elapsed = (end.tv_sec - start.tv_sec)
                   + (end.tv_nsec - start.tv_nsec) / 1e9;
    printf("Elapsed: %.6f seconds\n", elapsed);
}
```

Output:

```
Elapsed: 0.515421 seconds
```

This measures **real elapsed time**, immune to system clock changes.

Step 6. Sleeping for a Duration

You can make your program pause using `sleep()` or `nanosleep()`.

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

int main(void) {
    printf("Sleeping for 2 seconds...\n");
    sleep(2);
    printf("Awake!\n");
}
```

For sub-second precision:

```c
#include <time.h>

int main(void) {
    struct timespec ts = {0, 500000000}; // 0.5 seconds
    nanosleep(&ts, NULL);
}
```

Step 7. Getting UTC and Local Time Zones

`gmtime()` gives you UTC, while `localtime()` converts to your system's timezone.

```c
time_t now = time(NULL);
printf("UTC:   %s", asctime(gmtime(&now)));
printf("Local: %s", asctime(localtime(&now)));
```

You can change timezone behavior via the `TZ` environment variable and `tzset()`.

Step 8. Process Time and Resource Usage

You can inspect how much CPU time your program used with `getrusage()`.

```c
#include <sys/resource.h>
#include <stdio.h>

int main(void) {
    struct rusage usage;
    getrusage(RUSAGE_SELF, &usage);
    printf("User CPU time: %ld.%06lds\n",
           usage.ru_utime.tv_sec, usage.ru_utime.tv_usec);
    printf("System CPU time: %ld.%06lds\n",
           usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
}
```

This is how profilers measure CPU consumption.

Step 9. Time Differences

You can subtract two `time_t` values using `difftime()`.

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

int main(void) {
    time_t start = time(NULL);
    sleep(2);
    time_t end = time(NULL);
    printf("Elapsed: %.0f seconds\n", difftime(end, start));
}
```

Output:

```
Elapsed: 2 seconds
```

Step 10. Tiny Code: Countdown Timer

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

int main(void) {
    for (int i = 5; i > 0; i--) {
```

```
        printf("%d...\n", i);
        sleep(1);
    }
    printf("Time's up!\n");
}
```

Output:

```
5...
4...
3...
2...
1...
Time's up!
```

Why It Matters

Time functions are critical in:

- Logging and timestamps
- Benchmarking and profiling
- Scheduling events
- Measuring performance of algorithms
- Synchronizing distributed systems

Every systems program eventually needs **accurate, reliable time measurement**, and C gives you all the low-level tools to do it efficiently.

Try It Yourself

1. Print the current date in ISO 8601 format.
2. Measure how long it takes to read a large file.
3. Build a stopwatch that measures elapsed time with `clock_gettime()`.
4. Make a countdown timer that updates in place on the terminal.
5. Display both UTC and local time, formatted nicely.

Next, you'll learn how to **access and modify environment variables**, another key part of how Unix programs communicate with their runtime environment.

68. Environment Variables

Every program in Unix inherits a **set of key–value pairs** called **environment variables**. They store information about your shell, system configuration, and runtime behavior, such as your username, home directory, and compiler paths.

C gives you full control to **read, modify, and define** these variables directly.

Step 1. What Are Environment Variables?

Environment variables are strings of the form:

```
KEY=VALUE
```

You can view them in your shell:

```
printenv
```

Common examples:

```
HOME=/home/user
PATH=/usr/local/bin:/usr/bin:/bin
USER=alice
LANG=en_US.UTF-8
SHELL=/bin/bash
```

These values are passed to every program when you run it.

Step 2. Accessing Environment Variables in C

You can use the standard library function `getenv()` to read a variable.

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

int main(void) {
    const char *path = getenv("PATH");
    if (path)
        printf("PATH = %s\n", path);
    else
        printf("PATH not found.\n");
}
```

Output example:

```
PATH = /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

If the variable doesn't exist, `getenv()` returns `NULL`.

Step 3. Setting Environment Variables

To define or change a variable inside your program, use `setenv()`.

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

int main(void) {
    setenv("GREETING", "Hello from C!", 1);
    printf("%s\n", getenv("GREETING"));
}
```

`setenv(name, value, overwrite)`

- `overwrite = 0`: don't overwrite existing variable.
- `overwrite = 1`: replace it if it exists.

Output:

```
Hello from C!
```

Step 4. Removing Environment Variables

Use `unsetenv()` to delete a variable.

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

int main(void) {
    setenv("TEMPVAR", "temporary", 1);
    printf("Before unset: %s\n", getenv("TEMPVAR"));
    unsetenv("TEMPVAR");
    printf("After unset: %s\n", getenv("TEMPVAR"));
}
```

Output:

```
Before unset: temporary
After unset: (null)
```

Step 5. Accessing All Environment Variables

The `environ` global variable gives you access to the entire environment list.

```c
#include <stdio.h>

extern char **environ;

int main(void) {
    for (char **env = environ; *env != NULL; env++) {
        printf("%s\n", *env);
    }
}
```

This prints all environment variables currently active in your process.

Step 6. Passing Environment Variables to Child Processes

When you use `fork()` and `exec()`, environment variables are **inherited** by the child process automatically.

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

int main(void) {
    setenv("HELLO", "world", 1);
    execlp("printenv", "printenv", "HELLO", NULL);
    perror("execlp");
}
```

Output:

```
world
```

You can also provide a custom environment list using `execle()` or `execve()`.

Step 7. Custom Environment for a New Program

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

int main(void) {
    char *newenv[] = { "MODE=debug", "VERSION=1.0", NULL };
    execle("/usr/bin/env", "env", NULL, newenv);
    perror("execle");
}
```

Output:

```
MODE=debug
VERSION=1.0
```

Only these two variables exist for the new process, everything else is discarded.

Step 8. Why Environment Variables Matter

They are a key part of **Unix's design philosophy**:

- Programs should be **configurable without recompilation**.
- Environment variables provide **global, process-level configuration**.

For example:

- `PATH` controls where executables are searched.
- `HOME` defines user directories.
- `LANG` defines locale settings.
- `LD_LIBRARY_PATH` controls dynamic linking.

Step 9. Security Considerations

Environment variables are inherited automatically, so they can be a security risk if not handled carefully:

- Avoid trusting environment variables for authentication.
- Always validate `PATH`, `HOME`, and `TMPDIR`.
- Use `unsetenv()` for sensitive contexts (e.g., setuid programs).

Tiny Code: Mini Shell with `PATH` Lookup

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

int main(void) {
    const char *path = getenv("PATH");
    if (!path) path = "(none)";
    printf("Current PATH:\n%s\n", path);
    setenv("PATH", "/usr/local/bin:/usr/bin", 1);
    printf("\nUpdated PATH:\n%s\n", getenv("PATH"));
}
```

Output:

```
Current PATH:
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin

Updated PATH:
/usr/local/bin:/usr/bin
```

Step 10. Why It Matters

Environment variables give your programs **context**:

- They tell you where files, libraries, and configs live.
- They let you control runtime behavior dynamically.
- They're essential for system tools, shells, daemons, and tests.

Understanding how to read and modify them is key to mastering Unix programming in C.

Try It Yourself

1. Print all environment variables sorted alphabetically.
2. Implement your own `env` command in C.
3. Write a program that modifies `PATH` and launches another program.
4. Create a child process that inherits modified variables.
5. Combine `setenv()`, `execle()`, and `getenv()` to simulate sandboxed runs.

Next, you'll learn about **error handling and return codes**, the invisible signals that every Unix process uses to tell the system whether it succeeded or failed.

69. Error Handling and Return Codes

Every C program, from the tiniest script to the Linux kernel itself, relies on **error codes** and **return values** to communicate success or failure. Unlike higher-level languages, C gives you **no exceptions**, only clear, explicit status codes and `errno`.

Mastering these patterns will make your programs robust, predictable, and professional.

Step 1. Exit Codes and `main()`

Every process returns an integer exit code to the operating system. Conventionally:

- `0` → success
- nonzero → failure or specific error

```c
#include <stdio.h>

int main(void) {
    printf("Everything OK!\n");
    return 0; // exit success
}
```

Check it in your shell:

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

Output:

```
Everything OK!
0
```

If you return a nonzero value:

```c
return 1;
```

then `$?` will be `1`, meaning failure.

Step 2. Using `EXIT_SUCCESS` and `EXIT_FAILURE`

Instead of hardcoded numbers, use standard macros from `<stdlib.h>`:

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

int main(void) {
    printf("Failed to open file.\n");
    return EXIT_FAILURE;
}
```

These improve portability and readability.

Step 3. The Global `errno`

When a library or system call fails, it usually sets a global variable named **`errno`**. It's declared in `<errno.h>`.

```c
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd = open("nonexistent.txt", O_RDONLY);
    if (fd == -1) {
        printf("Error opening file: %s\n", strerror(errno));
    }
}
```

Output:

```
Error opening file: No such file or directory
```

`errno` stores an integer code, but `strerror()` converts it into a readable message.

Step 4. Common `errno` Values

Code	Macro	Meaning
2	`ENOENT`	No such file or directory
13	`EACCES`	Permission denied
12	`ENOMEM`	Out of memory
22	`EINVAL`	Invalid argument
9	`EBADF`	Bad file descriptor
11	`EAGAIN`	Resource temporarily unavailable

You can check them directly:

```
if (errno == EACCES) { ... }
```

Step 5. The `perror()` Function

A simpler way to print error messages is `perror()`, it automatically uses the current `errno`.

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

int main(void) {
    int fd = open("nothing.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
    }
}
```

Output:

```
open: No such file or directory
```

Step 6. Returning Meaningful Codes

Good C programs **translate internal errors into meaningful exit codes**.

Example: file copy program

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s src dest\n", argv[0]);
        return 1;
    }
    FILE *src = fopen(argv[1], "r");
    if (!src) {
        perror("fopen src");
        return 2;
    }
    FILE *dst = fopen(argv[2], "w");
    if (!dst) {
        perror("fopen dst");
        fclose(src);
        return 3;
    }
    fclose(src);
    fclose(dst);
    return 0;
}
```

Now each exit code represents a specific type of failure.

Step 7. Resetting and Checking `errno`

Some system calls set `errno` only when they fail. So always reset it before use if you plan to inspect it later:

```c
#include <errno.h>
errno = 0;
if (some_syscall() == -1) {
    perror("syscall failed");
}
```

Step 8. Custom Error Handling Helpers

You can create your own function to simplify error handling.

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

void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

int main(void) {
    FILE *f = fopen("no.txt", "r");
    if (!f) die("fopen");
}
```

Output:

```
fopen: No such file or directory
```

This pattern appears throughout Unix utilities.

Step 9. Handling Partial Failures

Not all errors should abort your program. Sometimes you should log, retry, or ignore.

```c
FILE *f = fopen("optional.conf", "r");
if (!f) {
    fprintf(stderr, "Warning: config file missing, using defaults.\n");
}
```

This kind of **graceful degradation** is good design.

Step 10. Tiny Code: Safe File Reader

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

int main(void) {
    FILE *f = fopen("data.txt", "r");
```

```
    if (!f) {
        fprintf(stderr, "Error: %s\n", strerror(errno));
        return EXIT_FAILURE;
    }

    char buf[64];
    while (fgets(buf, sizeof(buf), f))
        printf("%s", buf);

    fclose(f);
    return EXIT_SUCCESS;
}
```

Output (if missing file):

```
Error: No such file or directory
```

Why It Matters

Error handling separates **toy programs** from **real software**:

- Every syscall can fail, be prepared.
- Always check return values.
- Always report why it failed.

By convention:

- Return `0` on success.
- Return nonzero for recoverable or fatal errors.
- Print messages to `stderr`, not `stdout`.

Try It Yourself

1. Open a nonexistent file and print the full `errno` value.
2. Build a small utility that returns specific codes for specific problems.
3. Use `perror()` vs `strerror()` and compare their outputs.
4. Add a `die()` helper to your previous exercises.
5. Write a safe wrapper that retries system calls when `errno == EAGAIN`.

Next, you'll put all of this together in **Practice 70: Building a Mini Shell in C**, where you'll handle processes, pipes, and signals to create your own working Unix shell prototype.

70. Practice: Mini Shell in C

It's time to bring together everything you've learned so far, **system calls**, **process creation**, **pipes**, **redirection**, and **signal handling**, into one cohesive project.

In this section, you'll build a **minimal interactive shell**, just like `bash` or `zsh`, but stripped down to the essentials. It will run commands, handle input/output redirection, and even support pipelines.

Step 1. What You'll Build

Your mini shell will:

1. Display a prompt like `$`
2. Read user input (e.g., `ls -l`, `cat file.txt`)
3. Parse it into command and arguments
4. Create a new process using `fork()`
5. Replace the process image with the requested command using `execvp()`
6. Wait for the child to finish

Optional extensions:

- Handle signals (Ctrl+C) gracefully
- Redirect output to a file (`>` redirection)
- Chain commands using pipes (`|`)

Step 2. Core Loop Skeleton

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 1024

int main(void) {
    char input[MAX];

    while (1) {
        printf("$ ");
```

```
        fflush(stdout);

        if (!fgets(input, sizeof(input), stdin))
            break;

        // Remove newline
        input[strcspn(input, "\n")] = 0;

        // Exit command
        if (strcmp(input, "exit") == 0)
            break;

        // Tokenize input
        char *args[64];
        int i = 0;
        char *token = strtok(input, " ");
        while (token) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        // Fork and execute
        pid_t pid = fork();
        if (pid == 0) {
            execvp(args[0], args);
            perror("execvp");
            exit(1);
        } else if (pid > 0) {
            wait(NULL);
        } else {
            perror("fork");
        }
    }

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

Compile and run:

```
gcc mini_shell.c -o mini_shell
./mini_shell
```

Try commands:

```
$ ls
$ pwd
$ echo hello world
$ exit
```

Step 3. Handling Errors Gracefully

If you enter a wrong command:

```
$ xyz
```

Output:

```
execvp: No such file or directory
```

This happens because the program handles `execvp()` failure properly with `perror()`, just as you learned in section 69.

Step 4. Adding Signal Handling

Let's make Ctrl+C stop the running command, but not kill the shell itself.

```
#include <signal.h>

void sigint_handler(int sig) {
    printf("\nType 'exit' to quit.\n$ ");
    fflush(stdout);
}

int main(void) {
    signal(SIGINT, sigint_handler);
    ...
}
```

Now the shell ignores Ctrl+C while waiting for input, instead of terminating.

Step 5. Supporting Output Redirection

We'll add `>` redirection like:

```
$ echo Hello > out.txt
```

Add this before the `execvp()` call in the child:

```
#include <fcntl.h>

for (int j = 0; args[j]; j++) {
    if (strcmp(args[j], ">") == 0) {
        args[j] = NULL;
        int fd = open(args[j + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
        dup2(fd, STDOUT_FILENO);
        close(fd);
        break;
    }
}
```

Now `stdout` of the command goes to the file instead of the screen.

Step 6. Supporting Input Redirection

Similarly, for `<` redirection:

```
$ cat < in.txt
```

Add:

```
if (strcmp(args[j], "<") == 0) {
    args[j] = NULL;
    int fd = open(args[j + 1], O_RDONLY);
    dup2(fd, STDIN_FILENO);
    close(fd);
    break;
}
```

Step 7. Adding Pipe Support

To handle commands like:

```
$ ls | wc -l
```

We create two processes connected by a pipe.

```
int pipefd[2];
pipe(pipefd);

pid_t p1 = fork();
if (p1 == 0) {
    dup2(pipefd[1], STDOUT_FILENO);
    close(pipefd[0]);
    execlp("ls", "ls", NULL);
}

pid_t p2 = fork();
if (p2 == 0) {
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[1]);
    execlp("wc", "wc", "-l", NULL);
}

close(pipefd[0]);
close(pipefd[1]);
wait(NULL);
wait(NULL);
```

That's the same pattern you saw in section 64, `ls | wc -l`.

Step 8. Combining All Features

Your shell now:

- Parses user input
- Spawns child processes
- Handles I/O redirection
- Supports Ctrl+C interruption
- Runs simple pipelines

With ~150 lines of code, you have a **working Unix shell prototype**.

Step 9. Tiny Code: Full Mini Shell

Here's the clean, minimal version:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <signal.h>

void sigint_handler(int sig) {
    printf("\n$ ");
    fflush(stdout);
}

int main(void) {
    signal(SIGINT, sigint_handler);
    char input[1024];

    while (1) {
        printf("$ ");
        fflush(stdout);
        if (!fgets(input, sizeof(input), stdin)) break;
        input[strcspn(input, "\n")] = 0;
        if (strcmp(input, "exit") == 0) break;

        char *args[64];
        int i = 0;
        char *token = strtok(input, " ");
        while (token) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;

        pid_t pid = fork();
        if (pid == 0) {
            for (int j = 0; args[j]; j++) {
                if (strcmp(args[j], ">") == 0) {
                    int fd = open(args[j + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
                    dup2(fd, STDOUT_FILENO);
```

```
                    close(fd);
                    args[j] = NULL;
                } else if (strcmp(args[j], "<") == 0) {
                    int fd = open(args[j + 1], O_RDONLY);
                    dup2(fd, STDIN_FILENO);
                    close(fd);
                    args[j] = NULL;
                }
            }
            execvp(args[0], args);
            perror("execvp");
            exit(1);
        } else if (pid > 0) {
            wait(NULL);
        }
    }

    printf("Exiting shell.\n");
    return 0;
}
```

Try it out, it's a real, interactive shell.

Step 10. Why It Matters

This exercise combines everything you've learned in Chapter 7:

- System calls (`fork`, `exec`, `wait`, `pipe`, `dup2`)
- Signals (`SIGINT`)
- File descriptors and redirection
- Error handling with `errno`
- Environment inheritance

You've just built a simplified version of the core that powers **every Unix shell**, from `bash` to `zsh` to `fish`.

Try It Yourself

1. Add support for pipelines (`|`) by chaining multiple commands.
2. Implement background processes with `&`.
3. Add `cd` and `pwd` as built-in commands.

4. Display the exit code after each command.
5. Handle multiple spaces and quoted arguments.

Next, we'll move into **Chapter 8: Debugging, Testing, and Profiling**, starting with **gdb**, your most powerful ally in understanding and fixing C programs.

Chapter 8. Debugging, Testing and Profiling

71. Debugging with gdb

Every C programmer eventually meets a segmentation fault, and that's when you discover your most powerful companion: **gdb**, the GNU Debugger. Debugging isn't about luck; it's about learning to inspect a program as it runs, to pause time, and to see what the computer is really doing.

Let's learn how to use gdb to **find bugs, inspect memory, trace crashes**, and truly understand your code.

Step 1. Compiling with Debug Information

Before you can debug, you need to tell the compiler to include **symbol information** (variable names, line numbers, etc.). Use the `-g` flag:

```
gcc -g main.c -o main
```

You can now open it in gdb:

```
gdb ./main
```

Step 2. Starting and Running Your Program

Inside gdb, you can run your program just like normal:

```
(gdb) run
```

If it crashes, you'll get something like:

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000040114a in buggy_function () at main.c:12
12        *ptr = 42;
```

You now know exactly where it failed.

Step 3. Setting Breakpoints

A **breakpoint** tells gdb to pause before executing a specific line or function.

```
(gdb) break main
Breakpoint 1 at 0x40113b: file main.c, line 5.
(gdb) run
```

When the program stops, you can inspect state:

```
(gdb) print variable_name
(gdb) next
(gdb) step
```

- `next` executes the next line (skipping function calls).
- `step` goes *into* a function call.

Step 4. Example: A Simple Bug

Here's a program with a classic segmentation fault.

```c
#include <stdio.h>

void buggy(void) {
    int *p = NULL;
    *p = 10;
}

int main(void) {
    printf("Before crash\n");
    buggy();
    printf("After crash\n");
}
```

Compile and debug:

```
gcc -g bug.c -o bug
gdb ./bug
(gdb) run
```

Output:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555159 in buggy () at bug.c:5
5        *p = 10;
```

Now inspect:

```
(gdb) backtrace
#0 buggy () at bug.c:5
#1 main () at bug.c:10
```

You've just **traced the crash** from main to the exact faulty line.

Step 5. Inspecting Variables

You can view variable values anytime:

```
(gdb) print x
(gdb) print *ptr
```

You can also modify them:

```
(gdb) set variable x = 100
```

To list all local variables:

```
(gdb) info locals
```

Step 6. Navigating Execution

Common gdb commands:

Command	Action
`run`	Start the program
`break N`	Stop at line N
`next`	Run next line
`step`	Step into function
`continue`	Resume until next breakpoint
`finish`	Run until function returns

Command	Action
`backtrace`	Show call stack
`info locals`	List local vars
`print VAR`	Show value
`quit`	Exit gdb

Step 7. Watching Variables

You can set **watchpoints**, gdb stops when a variable changes.

```
int counter = 0;
for (int i = 0; i < 10; i++)
    counter += i;
```

In gdb:

```
(gdb) watch counter
(gdb) run
```

Every time `counter` changes, the program pauses, showing where it happened.

Step 8. Conditional Breakpoints

Sometimes you only want to stop under specific conditions:

```
(gdb) break loop.c:25 if i > 100
```

This breakpoint triggers only when `i` exceeds 100.

Step 9. Inspecting Memory and Registers

You can inspect raw memory:

```
(gdb) x/10x &array
```

This prints 10 hexadecimal words starting at `&array`.

Or view registers:

```
(gdb) info registers
```

This is useful for low-level debugging (e.g., compilers, OS kernels, embedded code).

Step 10. Tiny Code: Debugging a Logic Bug

```c
#include <stdio.h>

int main(void) {
    int sum = 0;
    for (int i = 1; i <= 5; i++) {
        sum = sum + i;
    }
    printf("Sum = %d\n", sum); // should be 15
}
```

Introduce a bug:

```c
for (int i = 1; i <= 5; i++); // extra semicolon!
```

Compile and debug:

```
gcc -g bug.c -o bug
gdb ./bug
(gdb) break main
(gdb) run
(gdb) print sum
(gdb) next
(gdb) print sum
```

You'll see that `sum` never changes, because the loop body was empty.

Why It Matters

Debugging is how you **learn to think like the machine**:

- You can inspect what happens between lines.
- You can see every variable's value.
- You can find segmentation faults in seconds.

Learning gdb teaches you how C really runs, from stack frames to pointers.

Try It Yourself

1. Write a program that crashes (e.g., use a null pointer) and trace the cause with gdb.
2. Use `next` and `step` to trace a recursive function.
3. Set a watchpoint on a variable in a loop.
4. Add a conditional breakpoint that triggers only when a value exceeds a limit.
5. Explore `backtrace` and `info locals` after a crash.

Next, you'll learn how to **detect hidden memory errors**, leaks, invalid frees, and buffer overflows, using the indispensable **Valgrind** tool.

72. Using Valgrind for Memory Checking

If gdb helps you *see* how your program runs, **Valgrind** helps you *see where it leaks*. C gives you raw control over memory, and that means you're responsible for every allocation, deallocation, and pointer access.

Valgrind is your best friend when you need to find:

- Memory leaks (forgotten `free()`)
- Invalid reads/writes
- Double frees
- Use-after-free errors

Let's learn how to use it to make your programs solid and leak-free.

Step 1. Installing Valgrind

On Linux:

```
sudo apt install valgrind
```

On macOS (with Homebrew):

```
brew install valgrind
```

Compile your program **with debug symbols**:

```
gcc -g memory.c -o memory
```

Step 2. Running with Valgrind

Run your program under Valgrind:

```
valgrind ./memory
```

Valgrind runs your program inside a virtual CPU and monitors every memory operation. At the end, it prints a detailed report of allocations and leaks.

Step 3. A Simple Example

Here's a program with two common mistakes: a leak and an invalid free.

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

int main(void) {
    int *p = malloc(10 * sizeof(int));
    p[10] = 42;  // invalid write (out of bounds)
    return 0;    // forgot to free(p)
}
```

Run it:

```
gcc -g mem_bug.c -o mem_bug
valgrind ./mem_bug
```

Output:

```
==1234== Invalid write of size 4
==1234==    at 0x1091A: main (mem_bug.c:6)
==1234==  Address 0x5201048 is 0 bytes after a block of size 40 alloc'd
==1234==    at 0x484186F: malloc (vg_replace_malloc.c:380)
==1234==
==1234== HEAP SUMMARY:
==1234==    definitely lost: 40 bytes in 1 blocks
==1234== LEAK SUMMARY:
==1234==    definitely lost: 40 bytes in 1 blocks
```

Valgrind caught both the invalid access **and** the leak.

Step 4. Fixing the Errors

Correct version:

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

int main(void) {
    int *p = malloc(10 * sizeof(int));
    if (!p) return 1;
    p[9] = 42;   // valid index
    free(p);
}
```

Run again:

```
valgrind ./mem_bug
```

Output:

```
== All heap blocks were freed -- no leaks are possible
== ERROR SUMMARY: 0 errors from 0 contexts
```

Clean and perfect.

Step 5. Detecting Use-After-Free

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

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 5;
    free(p);
    printf("%d\n", *p); // using freed memory
}
```

Valgrind says:

```
==1234== Invalid read of size 4
==1234==    at 0x1091A: main (use_after_free.c:7)
==1234==  Address 0x5201040 is 0 bytes inside a block of size 4 free'd
```

It even shows *where* the block was freed.

Step 6. Detecting Double Free

```c
#include <stdlib.h>

int main(void) {
    int *p = malloc(4);
    free(p);
    free(p);
}
```

Valgrind output:

```
==1234== Invalid free() / delete / delete[]
==1234==    at 0x4845DEF: free (vg_replace_malloc.c:872)
==1234==  Address 0x5201040 was freed previously
```

Step 7. Memory Leak Categories

Valgrind divides leaks into categories:

Type	Meaning
definitely lost	No pointer to the block remains, true leak
indirectly lost	Referenced by a leaked block
possibly lost	Pointer may exist but Valgrind can't confirm
still reachable	Program ended but memory wasn't freed (often harmless)

Step 8. Getting a Clean Report

To check for true leaks only:

```
valgrind --leak-check=full --show-leak-kinds=definite ./program
```

For more verbose tracking:

```
valgrind --track-origins=yes ./program
```

That flag tells you *where* an uninitialized variable first appeared.

Step 9. Checking for Stack or Uninitialized Values

```
#include <stdio.h>

int main(void) {
    int x;
    printf("%d\n", x); // uninitialized read
}
```

Valgrind output:

```
==1234== Use of uninitialised value of size 4
==1234==    at 0x1091A: main (uninit.c:5)
```

Always initialize your variables!

Step 10. Tiny Code: Leak Detector

```
#include <stdlib.h>

void leak1(void) { malloc(100); }
void leak2(void) { char *p = malloc(50); free(p); }

int main(void) {
    leak1();
    leak2();
}
```

Run:

```
valgrind --leak-check=full ./leaks
```

Output:

```
==1234== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1234== LEAK SUMMARY:
==1234==    definitely lost: 100 bytes in 1 blocks
```

Only `leak1()` forgot to `free()`, Valgrind pinpointed it exactly.

Why It Matters

Valgrind helps you:

- Find hidden memory leaks
- Detect invalid pointer usage
- Catch uninitialized values
- Write safer, cleaner, more reliable C

It's an essential tool in your workflow, especially for long-running programs, servers, or systems software.

Try It Yourself

1. Write a program that allocates multiple blocks and forgets to free one.
2. Intentionally use `p[10]` on a `malloc(10)` block.
3. Trigger a use-after-free and find it in Valgrind.
4. Use `--track-origins=yes` to trace uninitialized data.
5. Refactor your code until Valgrind reports:

   ```
   All heap blocks were freed -- no leaks are possible
   ```

Next, you'll explore **Assertions and Defensive Programming**, techniques to catch logic errors *before* they reach runtime crashes.

73. Assertions and Defensive Programming

Bugs are inevitable, but crashes don't have to be. C gives you direct power over the machine, which means **you must protect your own assumptions**. That's where **assertions** and **defensive programming** come in: they help you catch mistakes early, fail fast, and make your code predictable.

Step 1. What Is an Assertion?

An **assertion** is a sanity check built into your code. It tests whether something you believe to be true *actually is*. If not, the program immediately stops with an error message, before things get worse.

Include the header:

```
#include <assert.h>
```

Example:

```
int divide(int a, int b) {
    assert(b != 0);  // ensure no division by zero
    return a / b;
}
```

If `b` is zero, the program aborts:

```
Assertion failed: (b != 0), function divide, file main.c, line 3.
```

Step 2. How Assertions Work

`assert(expression)` expands to something like:

```
if (!(expression)) {
    fprintf(stderr, "Assertion failed: %s, file %s, line %d\n",
            "expression", __FILE__, __LINE__);
    abort();
}
```

When compiled normally, it checks the condition. When compiled with `-DNDEBUG`, assertions are disabled.

Step 3. Enabling or Disabling Assertions

Default: assertions are active. To disable them (for production builds):

```
gcc -DNDEBUG program.c -o program
```

In that build, `assert()` statements are removed.

This lets you keep your debug checks without slowing down release binaries.

Step 4. Practical Example

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

int find_max(int *arr, int n) {
    assert(arr != NULL);
    assert(n > 0);
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max) max = arr[i];
    return max;
}

int main(void) {
    int data[] = {3, 5, 7, 2, 8};
    printf("Max: %d\n", find_max(data, 5));
}
```

If you pass a `NULL` pointer or invalid length, the program fails immediately.

Step 5. Writing Good Assertions

Assertions should test *internal assumptions*, not user input. Bad:

```
assert(argc == 3);
```

Good:

```c
if (argc != 3) {
    fprintf(stderr, "Usage: %s input output\n", argv[0]);
    return 1;
}
```

Use assertions to check invariants inside your logic, things that *should never happen* unless there's a bug.

Step 6. Defensive Programming Techniques

Defensive programming goes beyond assertions, it's about **writing code that assumes mistakes will happen**.

Check every function return value:

```c
FILE *f = fopen("file.txt", "r");
if (!f) {
    perror("fopen");
    return 1;
}
```

Validate inputs:

```c
int divide(int a, int b) {
    if (b == 0) {
        fprintf(stderr, "Division by zero!\n");
        return 0;
    }
    return a / b;
}
```

Avoid undefined behavior:

- Initialize all variables.
- Don't access freed memory.
- Check array bounds.
- Always match `malloc()` with `free()`.

Step 7. Assertions in Complex Systems

In large programs, assertions act like *tripwires* to detect when state becomes inconsistent.

Example: a queue

```c
#include <assert.h>

void enqueue(int *queue, int *count, int value, int max) {
    assert(*count < max);
    queue[*count] = value;
    (*count)++;
}
```

If something goes wrong in your logic, the assertion will tell you immediately, before memory corruption happens.

Step 8. Logging vs Assertions

- **Assertions**: catch programming errors.
- **Logging**: record runtime information.

Combine both:

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

int read_value(int *arr, int n, int index) {
    assert(index >= 0 && index < n);
    printf("Reading arr[%d] = %d\n", index, arr[index]);
    return arr[index];
}
```

Step 9. Using `static_assert` for Compile-Time Checks (C11+)

C11 introduced `_Static_assert`, which checks conditions **during compilation**.

```c
#include <assert.h>
_Static_assert(sizeof(int) == 4, "int must be 4 bytes");
```

If the condition fails, the compiler stops with:

```
error: static assertion failed: "int must be 4 bytes"
```

This is perfect for configuration or architecture assumptions.

Step 10. Tiny Code: Safe Array Access

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

#define MAX 5

int safe_get(int arr[], int n, int i) {
    assert(i >= 0 && i < n);
    return arr[i];
}

int main(void) {
    int nums[MAX] = {1, 2, 3, 4, 5};
    printf("%d\n", safe_get(nums, MAX, 2));  // OK
    printf("%d\n", safe_get(nums, MAX, 10)); // triggers assertion
}
```

Output:

```
Assertion failed: (i >= 0 && i < n), function safe_get, file main.c, line 7.
```

Why It Matters

Assertions and defensive coding make your software:

- **Safer**, catch bugs early.
- **Easier to debug**, fail at the source, not later.
- **More maintainable**, documents assumptions clearly.

In C, a single bad pointer can crash your system. Assertions are your safety net.

Try It Yourself

1. Add assertions to your stack or linked list implementation.
2. Write a function that validates all arguments before proceeding.
3. Use `_Static_assert` to check type sizes in a cross-platform program.
4. Combine assertions with logging for detailed error reports.
5. Run your program with invalid input and see how quickly assertions detect issues.

Next, you'll move into **unit testing in C**, building small, automated tests to ensure every function works exactly as intended.

74. Unit Testing in C

Testing isn't just something you do at the end, it's how you **build confidence** in every line of code. Unit testing means checking small, isolated pieces (functions, modules) automatically, so you can change your code without fear.

C doesn't come with a built-in testing framework, but it's easy to build lightweight ones, and several excellent libraries exist if you want more power.

Let's walk through how to design and run **unit tests in plain C**.

Step 1. What Is Unit Testing?

A *unit test* verifies a single behavior:

> Given an input, does this function produce the correct output?

For example:

```c
int add(int a, int b) { return a + b; }

void test_add(void) {
    if (add(2, 3) != 5) printf("test_add failed!\n");
    else printf("test_add passed!\n");
}
```

Run this test:

```c
int main(void) {
    test_add();
}
```

Output:

```
test_add passed!
```

Simple, but powerful.

Step 2. Organizing Tests

Keep tests separate from production code. A typical layout:

```
src/
  math.c
  math.h
tests/
  test_math.c
Makefile
```

Your Makefile might build both:

```
all:
    gcc -g -Wall -I../src ../src/math.c test_math.c -o test_math
```

Step 3. Writing Reusable Test Helpers

Define macros to simplify your checks.

```
#include <stdio.h>

#define ASSERT_EQ_INT(expected, actual) \
    if ((expected) != (actual)) \
        printf("FAIL: %s:%d: expected %d, got %d\n", __FILE__, __LINE__, (expected), (actual)
    else \
        printf("PASS: %s:%d\n", __FILE__, __LINE__);
```

Now:

```c
int add(int a, int b) { return a + b; }

void test_add(void) {
    ASSERT_EQ_INT(5, add(2, 3));
    ASSERT_EQ_INT(0, add(-1, 1));
}

int main(void) { test_add(); }
```

Output:

```
PASS: test_math.c:10
PASS: test_math.c:11
```

Step 4. Testing Multiple Functions

Add more test functions and call them in sequence:

```c
void test_subtract(void) { ASSERT_EQ_INT(1, 3 - 2); }

int main(void) {
    test_add();
    test_subtract();
}
```

Now your program automatically verifies both functions.

Step 5. Handling Floating-Point Comparisons

Floating-point values rarely match exactly, use a tolerance.

```c
#include <math.h>
#define ASSERT_EQ_FLOAT(expected, actual, eps) \
    if (fabs((expected) - (actual)) > (eps)) \
        printf("FAIL: expected %.3f, got %.3f\n", (expected), (actual)); \
    else \
        printf("PASS\n");
```

Step 6. Using Return Codes to Mark Failures

Instead of just printing results, you can make the test binary return `EXIT_FAILURE` if any test fails.

```
int fails = 0;
#define TEST(cond) \
    do { if (!(cond)) { \
        printf("FAIL: %s:%d: %s\n", __FILE__, __LINE__, #cond); \
        fails++; \
    } else { \
        printf("PASS: %s:%d\n", __FILE__, __LINE__); \
    } } while (0)
```

At the end:

```
return fails ? EXIT_FAILURE : EXIT_SUCCESS;
```

Now your CI or shell can detect test results via `$?`.

Step 7. Minimal Testing Framework: TinyTest

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

int tests_run = 0, tests_failed = 0;

#define TEST(name) void name(void)
#define RUN(test) do { \
    printf("Running %s... ", #test); \
    test(); \
    tests_run++; \
    printf("OK\n"); \
} while(0)
#define ASSERT_TRUE(cond) if (!(cond)) { \
    printf("\n  Assertion failed: %s\n", #cond); \
    tests_failed++; \
    return; \
}
```

```c
TEST(test_addition) {
    int sum = 2 + 3;
    ASSERT_TRUE(sum == 5);
}

int main(void) {
    RUN(test_addition);
    printf("\nTests run: %d, failed: %d\n", tests_run, tests_failed);
    return tests_failed ? EXIT_FAILURE : EXIT_SUCCESS;
}
```

Output:

```
Running test_addition... OK

Tests run: 1, failed: 0
```

Step 8. Using Real Testing Libraries

When your project grows, you can use frameworks like:

- **Check** (POSIX-compliant)
- **Unity** (embedded-friendly)
- **CMocka**
- **Criterion**

Example with *Check*:

```
sudo apt install check
```

```c
#include <check.h>

START_TEST(test_add)
{
    ck_assert_int_eq(2 + 3, 5);
}
END_TEST
```

Then compile with:

```
gcc test.c -lcheck -o test
```

Step 9. Automating Tests with Makefile

Add a test target:

```
test:
    gcc -Wall -g src/*.c tests/*.c -o tests/run_tests
    ./tests/run_tests
```

Now you can just run:

```
make test
```

and see your full suite's results.

Step 10. Tiny Code: Testing a Linked List

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

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *push(Node *head, int val) {
    Node *n = malloc(sizeof(Node));
    n->value = val;
    n->next = head;
    return n;
}

void test_push(void) {
    Node *head = NULL;
    head = push(head, 10);
    head = push(head, 20);
    assert(head->value == 20);
```

```c
    assert(head->next->value == 10);
    printf("test_push passed\n");
}

int main(void) {
    test_push();
    printf("All tests passed.\n");
}
```

Output:

```
test_push passed
All tests passed.
```

Why It Matters

Unit testing turns code into **verifiable logic**:

- Prevents regressions.
- Encourages small, clean functions.
- Makes debugging faster.
- Builds confidence before refactoring.

When you trust your tests, you can rewrite your code fearlessly.

Try It Yourself

1. Write a test suite for your dynamic array implementation.
2. Add `ASSERT_EQ_FLOAT` and `ASSERT_EQ_STR` macros.
3. Automate tests using `make test`.
4. Add a `fails` counter and colorize your results.
5. Use a testing library like *Criterion* or *Unity* and compare styles.

Next, you'll learn how to **add logging systems** to your C programs, to record what's happening under the hood in a controlled, readable way.

75. Logging Systems

As your programs grow, **printf debugging** quickly becomes messy. You need a way to *see inside your program*, what it's doing, what went wrong, and why, without flooding your terminal with random messages.

That's where **logging systems** come in. A good log system helps you trace execution, record errors, and understand how your program behaves over time.

Step 1. What Is Logging?

Logging is like keeping a diary for your program. Instead of printing everything to the screen, you log structured messages with **levels** (INFO, WARN, ERROR) and **timestamps**.

It's essential for:

- Debugging complex flows.
- Auditing events and errors.
- Monitoring long-running services.
- Diagnosing crashes and performance issues.

Step 2. The Simplest Logger, printf with Context

```c
#include <stdio.h>

int main(void) {
    printf("[INFO] Starting program\n");
    printf("[WARN] Low memory\n");
    printf("[ERROR] Failed to open file\n");
}
```

This works, but there's no timestamp, file name, or severity control.

Step 3. Adding Log Levels and Macros

We can make this structured and reusable using macros.

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

#define LOG(level, msg, ...) do { \
    time_t t = time(NULL); \
    struct tm *tm_info = localtime(&t); \
    char buf[20]; \
    strftime(buf, 20, "%Y-%m-%d %H:%M:%S", tm_info); \
    fprintf(stderr, "[%s] [%s] " msg "\n", buf, level, ##__VA_ARGS__); \
} while (0)

#define LOG_INFO(msg, ...)  LOG("INFO", msg, ##__VA_ARGS__)
#define LOG_WARN(msg, ...)  LOG("WARN", msg, ##__VA_ARGS__)
#define LOG_ERROR(msg, ...) LOG("ERROR", msg, ##__VA_ARGS__)

int main(void) {
    LOG_INFO("Program started");
    LOG_WARN("Memory usage at %d%%", 80);
    LOG_ERROR("File %s not found", "data.txt");
}
```

Output:

```
[2025-10-16 23:41:09] [INFO] Program started
[2025-10-16 23:41:09] [WARN] Memory usage at 80%
[2025-10-16 23:41:09] [ERROR] File data.txt not found
```

Now your logs have consistent structure and useful context.

Step 4. Controlling Log Verbosity

Add a global log level:

```c
enum { LOG_LEVEL_INFO, LOG_LEVEL_WARN, LOG_LEVEL_ERROR };
int CURRENT_LOG_LEVEL = LOG_LEVEL_INFO;

#define SHOULD_LOG(level) ((level) >= CURRENT_LOG_LEVEL)
#define LOGX(level, tag, msg, ...) do { \
    if (SHOULD_LOG(level)) { \
        time_t t = time(NULL); \
```

```
        struct tm *tm_info = localtime(&t); \
        char buf[20]; \
        strftime(buf, 20, "%H:%M:%S", tm_info); \
        fprintf(stderr, "[%s] [%s] " msg "\n", buf, tag, ##__VA_ARGS__); \
    } \
} while (0)
```

Now:

```
LOGX(LOG_LEVEL_INFO, "INFO", "Running");
LOGX(LOG_LEVEL_WARN, "WARN", "Low disk space");
LOGX(LOG_LEVEL_ERROR, "ERROR", "Crash at line %d", __LINE__);
```

Change verbosity dynamically:

```
CURRENT_LOG_LEVEL = LOG_LEVEL_WARN;
```

Now INFO messages are skipped.

Step 5. Logging to a File

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

void log_to_file(const char *filename, const char *msg) {
    FILE *f = fopen(filename, "a");
    if (!f) return;
    time_t t = time(NULL);
    fprintf(f, "%s: %s\n", ctime(&t), msg);
    fclose(f);
}

int main(void) {
    log_to_file("log.txt", "Program started");
    log_to_file("log.txt", "Action complete");
}
```

The log file will contain:

```
Thu Oct 16 23:41:09 2025: Program started
Thu Oct 16 23:41:10 2025: Action complete
```

Step 6. Including File and Line Information

You can include source info automatically using `__FILE__` and `__LINE__`:

```
#define LOG_DEBUG(msg, ...) \
    fprintf(stderr, "[DEBUG] %s:%d " msg "\n", __FILE__, __LINE__, ##__VA_ARGS__)
```

Example:

```
LOG_DEBUG("x = %d", x);
```

Output:

```
[DEBUG] main.c:42 x = 10
```

Step 7. Rotating or Limiting Logs

For long-running programs, you don't want logs to grow forever. You can:

- Truncate or rename old files.
- Only keep N entries.
- Write daily logs (`log_2025-10-16.txt`).

Example:

```
char filename[64];
time_t now = time(NULL);
strftime(filename, sizeof(filename), "log_%Y-%m-%d.txt", localtime(&now));
log_to_file(filename, "Daily entry");
```

Step 8. Adding Colors (Optional)

Make console logs easier to read with ANSI color codes:

```
#define RED   "\x1b[31m"
#define YEL   "\x1b[33m"
#define GRN   "\x1b[32m"
#define RST   "\x1b[0m"

#define LOGC(level, color, msg, ...) \
    fprintf(stderr, color "[%s] " msg RST "\n", level, ##__VA_ARGS__)
```

Example:

```
LOGC("INFO", GRN, "Server started");
LOGC("WARN", YEL, "High CPU usage");
LOGC("ERROR", RED, "Out of memory");
```

Step 9. Combining Logging and Assertions

You can combine assertions with logs for extra safety:

```
#include <assert.h>

#define SAFE_LOG(cond, msg, ...) \
    if (!(cond)) { \
        LOG_ERROR(msg, ##__VA_ARGS__); \
        assert(cond); \
    }
```

If something fails, it both logs and triggers an assertion.

Step 10. Tiny Code: Minimal Logger

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

#define LOG(fmt, ...) do { \
    time_t now = time(NULL); \
    char buf[20]; \
    strftime(buf, sizeof(buf), "%H:%M:%S", localtime(&now)); \
    fprintf(stderr, "[%s] " fmt "\n", buf, ##__VA_ARGS__); \
} while (0)

int main(void) {
    LOG("Starting program");
    LOG("Loading config");
    LOG("Finished setup");
}
```

Output:

```
[23:42:00] Starting program
[23:42:01] Loading config
[23:42:02] Finished setup
```

Why It Matters

Logging makes invisible processes visible. It helps you:

- Trace execution flow.
- Debug production code.
- Audit errors and warnings.
- Understand system performance over time.

In real systems, servers, compilers, databases, **logs are your lifeline** when things go wrong.

Try It Yourself

1. Add `LOG_INFO`, `LOG_WARN`, and `LOG_ERROR` macros to one of your C projects.
2. Write logs to both `stderr` and a file.
3. Add timestamps and line numbers automatically.
4. Add colors for each level.
5. Implement a rotating file log system that keeps only today's file active.

Next, you'll learn about **profiling with gprof**, how to measure where your program spends its time, and how to make it faster.

76. Profiling with gprof

When your program works but feels *slow*, guessing isn't enough, you need to **measure**. Profiling shows you *where* your program spends its time, which functions are hot, and where optimization truly matters.

C gives you a lot of control, but performance tuning without profiling is like driving blindfolded. That's why we use **gprof**, the GNU profiler, a tool that measures how long your code spends in each function.

Step 1. What Is Profiling?

Profiling is the process of recording runtime statistics such as:

- How many times each function runs.
- How much CPU time each function uses.
- Which functions call which others.

It helps you find bottlenecks, functions that dominate runtime, and focus your optimization there.

Step 2. Enabling Profiling with gprof

Compile your program with the `-pg` flag to enable profiling hooks:

```
gcc -pg main.c -o program
```

Run the program normally:

```
./program
```

After it finishes, a file named `gmon.out` is created. This file contains execution data collected during runtime.

Generate a report:

```
gprof program gmon.out > analysis.txt
```

Now open `analysis.txt` to see where your program spent its time.

Step 3. Example Program

Here's a small example to demonstrate profiling:

```
#include <stdio.h>

void slow_function(void) {
    for (volatile long i = 0; i < 50000000; i++);
}

void fast_function(void) {
```

```
    for (volatile long i = 0; i < 5000000; i++);
}

int main(void) {
    slow_function();
    fast_function();
    slow_function();
    return 0;
}
```

Compile and run:

```
gcc -pg main.c -o profile_me
./profile_me
gprof profile_me gmon.out > report.txt
```

Step 4. Reading the gprof Report

The report has two key sections:

1. Flat Profile

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   ms/call  ms/call  name
 66.7      0.20     0.20        2   100.00   100.00  slow_function
 33.3      0.30     0.10        1   100.00   100.00  fast_function
```

This tells you:

- `slow_function()` took 66% of total time.
- `fast_function()` took 33%.

2. Call Graph

```
index % time    self  children    called     name
                0.20    0.00       2/2       slow_function
                0.10    0.00       1/1       fast_function
```

This shows relationships, which functions called which, and how time was distributed among them.

Step 5. Profiling Multi-File Programs

When working on multiple `.c` files, just compile each with `-pg`:

```
gcc -pg -c foo.c
gcc -pg -c bar.c
gcc -pg foo.o bar.o -o app
```

Then run and analyze as before.

Step 6. Ignoring Initialization or Short Runs

Profiling works best for *real workloads*. Avoid profiling tiny runs, because initialization costs can dominate and distort results.

For example, a 10 ms startup delay might look huge if your program only runs 20 ms in total. Use representative input and real loops to get meaningful data.

Step 7. Focusing on Hotspots

When you know which functions dominate runtime (often the top 5%), you can:

- Inline them manually.
- Use better data structures.
- Reduce allocations.
- Simplify inner loops.

Optimization is about precision, don't guess where your code is slow; let the profiler prove it.

Step 8. Combining gprof with Compiler Optimizations

Compare your performance before and after adding optimization flags:

```
gcc -pg -O0 main.c -o slow
gcc -pg -O3 main.c -o fast
```

Then run both and inspect the reports. You'll see dramatic changes in timing distribution, sometimes even inlined functions disappear entirely from the profile.

Step 9. Visualizing Profiles

You can visualize gprof results using **gprof2dot** and **Graphviz**:

```
pip install gprof2dot
gprof program gmon.out | gprof2dot | dot -Tpng -o profile.png
```

This generates a **call graph image**, showing which functions dominate. The thicker the arrow, the more time is spent there.

Step 10. Tiny Code: Measuring a Sorting Algorithm

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

#define N 100000

void bubble_sort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1]) {
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
}

void fill_random(int *arr, int n) {
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100000;
}

int main(void) {
    int *arr = malloc(N * sizeof(int));
    fill_random(arr, N);
    bubble_sort(arr, N);
    free(arr);
}
```

Compile and run:

```
gcc -pg main.c -O0 -o sort_profile
./sort_profile
gprof sort_profile gmon.out | head -n 20
```

Output (excerpt):

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   ms/call  ms/call  name
 95.0      0.95     0.95        1   950.00   950.00  bubble_sort
```

This tells you 95% of time is spent in `bubble_sort()`, confirming the algorithmic bottleneck.

Why It Matters

Profiling bridges the gap between "feels slow" and **knowing why**. It helps you:

- Focus optimization effort where it matters most.
- Verify improvements quantitatively.
- Eliminate guesswork.

In performance engineering, **measurement beats intuition** every time.

Try It Yourself

1. Profile your own sorting or search algorithm.
2. Compare results between `-O0`, `-O2`, and `-O3`.
3. Profile a multithreaded or I/O-bound program.
4. Visualize results with `gprof2dot`.
5. Identify one bottleneck and fix it, then re-profile to see the difference.

Next, you'll explore **common undefined behaviors** in C, the silent bugs that can make your perfectly profiled program crash unpredictably.

77. Common Undefined Behaviors

C gives you freedom, but also **responsibility**. Unlike higher-level languages, C doesn't protect you from dangerous mistakes. Some actions cause **undefined behavior (UB)**: the compiler is allowed to do *anything* in response, crash, hang, or even appear to work fine until it doesn't.

Undefined behavior is what makes C both powerful and perilous. Let's explore what causes it, how to recognize it, and how to write code that never falls into its traps.

Step 1. What Is Undefined Behavior?

In the C standard, **undefined behavior** means "no rules apply." If your program does something the language doesn't define, the compiler can assume it never happens and optimize freely.

This means your program might:

- Crash immediately.
- Produce wrong results.
- Behave differently each time.
- Work fine on one compiler and fail on another.

Example:

```
int x = 5 / 0;  // UB: division by zero
```

The compiler is *not required* to warn you or handle this safely.

Step 2. Common Sources of UB

Here are the most frequent offenders every C programmer must know:

Category	Example
Out-of-bounds access	`arr[10]` when the array has 10 elements
Use of uninitialized variable	`int x; printf("%d", x);`
Dangling pointer access	Use memory after `free()`
Invalid pointer arithmetic	`(p + 5)` when `p` doesn't point into an array
Signed integer overflow	`int x = INT_MAX + 1;`
Modifying and reading same variable	`i = i++;` or `a[i] = i++;`
Null pointer dereference	`int *p = NULL; *p = 5;`
Incorrect type punning	Accessing a float as int through wrong pointer type
Mismatched `malloc`/`free`	`free()` memory not allocated by `malloc()`
Violating `const` or `volatile` contracts	Writing to a `const` variable

Step 3. Out-of-Bounds Access

```
int arr[3] = {1, 2, 3};
printf("%d\n", arr[3]);  // UB: index 3 is past the end
```

C doesn't check bounds, you're responsible for it. You might print garbage, crash, or accidentally overwrite another variable.

Always check:

```
if (index >= 0 && index < size)
    printf("%d\n", arr[index]);
```

Step 4. Using Uninitialized Variables

```
int x;
printf("%d\n", x);  // UB: x is uninitialized
```

Even if it prints `0`, that's luck, not correctness. Always initialize your variables explicitly:

```
int x = 0;
```

Step 5. Dangling Pointers

```
int *p = malloc(sizeof(int));
*p = 10;
free(p);
printf("%d\n", *p);  // UB: accessing freed memory
```

After `free()`, the pointer still *exists* but the memory doesn't belong to you. Set it to `NULL`:

```
free(p);
p = NULL;
```

Step 6. Signed Integer Overflow

In C, **signed overflow is undefined**, but unsigned overflow wraps around predictably.

```
int x = 2147483647;
x = x + 1;  // UB
```

Unsigned version:

```c
unsigned int x = 4294967295;
x = x + 1;  // wraps to 0 (defined)
```

To check overflow safely:

```c
if (a > INT_MAX - b) {
    printf("overflow\n");
} else {
    x = a + b;
}
```

Step 7. Modifying and Reading in One Expression

```c
int i = 0;
i = i++ + 1;  // UB: reading and writing i without sequence point
```

Avoid combining side effects. Write clean code:

```c
i++;
i = i + 1;
```

Step 8. Null Pointer Dereference

```c
int *p = NULL;
*p = 10;  // UB
```

Always validate pointers:

```c
if (p != NULL) *p = 10;
```

Step 9. Type Punning and Aliasing

```c
float f = 3.14;
int *ip = (int *)&f;  // UB: violates strict aliasing
printf("%d\n", *ip);
```

If you must reinterpret bytes, use `memcpy`:

```
int i;
memcpy(&i, &f, sizeof(i));
```

This avoids aliasing violations and is safe across compilers.

Step 10. Tiny Code: Detecting UB with Tools

Use **compilers and runtime checkers** to detect UB before it hits production.

```
gcc -fsanitize=undefined -g ub_example.c -o ub_example
./ub_example
```

Sample program:

```
#include <stdio.h>

int main(void) {
    int x = 2147483647;
    x++;
    printf("%d\n", x);
}
```

Output:

```
runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in type 'int'
```

This is the **Undefined Behavior Sanitizer (UBSan)** in action, your best friend for finding invisible bugs.

Why It Matters

Undefined behavior is silent corruption. It can:

- Work on your machine but fail elsewhere.
- Break when you change compiler flags.
- Cause subtle, unpredictable bugs.

Avoiding UB is the foundation of reliable systems programming. In C, correctness comes from *discipline*.

Try It Yourself

1. Write a small program with deliberate UB (like using uninitialized variables).
2. Run it with `-fsanitize=undefined`.
3. Fix each issue until UBSan runs clean.
4. Check array access and pointer validity.
5. Refactor old C code to avoid UB, it's a great debugging exercise.

Next, you'll learn how to perform **crash analysis and read core dumps**, so even when your program fails, you can find out *exactly why*.

78. Crash Analysis and Core Dumps

Even with careful coding and testing, programs crash. In C, a crash is your system's way of saying *"you touched something you shouldn't have."* The good news is you can **analyze crashes** scientifically using **core dumps**, snapshots of your program's memory at the moment of failure.

Learning how to read them is an essential skill for every systems programmer.

Step 1. What Is a Core Dump?

A **core dump** is a file that captures your program's state (stack, registers, memory) at the time it crashed. You can inspect it later using a debugger like **gdb** to see what went wrong.

Common crash signals that generate core dumps:

- **SIGSEGV** – invalid memory access (segmentation fault)
- **SIGABRT** – failed assertion
- **SIGFPE** – arithmetic error (like division by zero)
- **SIGILL** – illegal instruction

Step 2. Enabling Core Dumps

By default, modern systems limit or disable them. Enable them using:

```
ulimit -c unlimited
```

Check:

```
ulimit -a | grep core
```

Now when your program crashes, a file named `core` or `core.<pid>` will appear.

Step 3. A Crashing Example

Let's make a simple crash:

```c
#include <stdio.h>

int main(void) {
    int *p = NULL;
    *p = 42;  // crash: dereferencing null pointer
    return 0;
}
```

Compile and run:

```
gcc crash.c -g -o crash
./crash
```

Output:

```
Segmentation fault (core dumped)
```

A `core` file is now created in your directory.

Step 4. Opening the Core Dump in gdb

Use gdb to inspect what happened:

```
gdb ./crash core
```

You'll see:

```
Core was generated by `./crash'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000000000401136 in main () at crash.c:5
5       *p = 42;
```

This tells you the **exact line** where the program crashed.

Step 5. Inspecting Variables and Stack

Within gdb:

```
(gdb) bt
```

Shows the **backtrace**, the call stack at the moment of crash.

```
#0  main () at crash.c:5
```

You can inspect variables:

```
(gdb) info locals
(gdb) print p
```

Output:

```
$1 = (int *) 0x0
```

Step 6. A More Complex Example

```c
#include <stdio.h>

void f3(int *p) {
    *p = 7;  // likely crash here
}

void f2(int *p) {
    f3(p);
}

void f1(void) {
    int *x = NULL;
    f2(x);
}

int main(void) {
    f1();
}
```

Run it, crash it, then:

```
gdb ./crash core
```

Inside gdb:

```
(gdb) bt
#0  f3 (p=0x0) at crash.c:4
#1  f2 (p=0x0) at crash.c:8
#2  f1 () at crash.c:12
#3  main () at crash.c:16
```

You can see the entire call chain that led to the null dereference.

Step 7. Debugging Optimized Builds

When you compile with `-O2` or `-O3`, the compiler may inline or reorder code, making debugging harder. For debugging, always use:

```
gcc -g -O0 crash.c -o crash
```

The `-g` flag keeps symbol information (file names, line numbers). Without it, gdb can't tell you much beyond addresses.

Step 8. Controlling Core Dump Location

Change where dumps are stored:

```
sudo sysctl -w kernel.core_pattern=/tmp/core.%e.%p
```

This example saves them in `/tmp` with program name and process ID:

```
/tmp/core.crash.1234
```

Step 9. Crash Analysis Workflow

1. **Enable dumps**: `ulimit -c unlimited`
2. **Run program until it crashes**
3. **Locate core file**
4. **Analyze with gdb**:
    - `bt` for stack trace
    - `info locals` for local vars
    - `frame <n>` to inspect each stack frame
    - `list` to see nearby code

This gives you a full picture of what happened just before failure.

Step 10. Tiny Code: Assertion Failure Analysis

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

int main(void) {
    int x = 5;
    assert(x == 10);  // fails
    printf("Done\n");
}
```

Compile and run:

```
gcc -g assert_fail.c -o assert_fail
ulimit -c unlimited
./assert_fail
```

Output:

```
assert_fail: assert_fail.c:5: main: Assertion `x == 10' failed.
Aborted (core dumped)
```

Now inspect:

```
gdb ./assert_fail core
(gdb) bt
```

Output:

```
#0  0x00007f0a9a4b9187 in raise () from /lib64/libc.so.6
#1  0x00007f0a9a4c53e8 in abort () from /lib64/libc.so.6
#2  0x00007f0a9a4b0246 in __assert_fail_base () from /lib64/libc.so.6
#3  0x00007f0a9a4b02f2 in __assert_fail () from /lib64/libc.so.6
#4  0x0000000000401136 in main () at assert_fail.c:5
```

You can trace exactly how the assertion caused the abort signal.

Why It Matters

Crash analysis turns chaos into clarity. Instead of guessing, you can:

- See which line caused the crash.
- Inspect all variables at that point.
- Reproduce and fix the bug quickly.

Every serious C programmer must master this, it's how systems engineers debug everything from user tools to kernels.

Try It Yourself

1. Write a small program that dereferences a null pointer.
2. Enable core dumps (`ulimit -c unlimited`).
3. Crash it, then analyze the dump in gdb.
4. Add one more function layer and check the call chain.
5. Experiment with `SIGFPE` (divide by zero) and see how the core dump differs.

Next, you'll build a **code review checklist for C projects**, habits and principles that help prevent these crashes before they ever happen.

79. Code Review Checklist for C Projects

Before your C program ships to production (or even your homework submission), it should survive one last test, **a code review**. This is where you or your teammates look at the code not just for correctness, but for *clarity, safety, and maintainability.*

Think of this as your personal pilot checklist before takeoff. Every great C programmer has one.

Step 1. Readability and Structure

- Are files organized logically (`src/`, `include/`, `tests/`)?
- Are headers clean, with include guards?
- Are functions short and focused (one purpose each)?
- Are names meaningful (`count_users()` is better than `doit()`)?
- Is indentation consistent and readable?
- Are comments clear and relevant, not just noise?

Tiny Code Example:

```
// Bad
void d(int a, int b) { printf("%d\n", a+b); }

// Good
void print_sum(int a, int b) {
    printf("%d\n", a + b);
}
```

Readable code *explains itself.*

Step 2. Header Hygiene

- Each `.h` file must have an **include guard**:

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
// declarations
#endif
```

- Headers should declare, not define.
- No global variables unless justified.
- Use `static inline` carefully (for small utilities only).

Step 3. Memory Safety

- Every `malloc` must have a corresponding `free`.
- Check all allocation results:

```
p = malloc(size);
if (!p) { perror("malloc"); exit(1); }
```

- Avoid dangling pointers:

```
free(p);
p = NULL;
```

- Use Valgrind or AddressSanitizer to ensure no leaks.

Step 4. Pointer Discipline

- Check for null pointers before dereferencing.
- Don't return pointers to local variables:

```
int* bad(void) {
    int x = 10;
    return &x; // wrong: stack memory
}
```

- Document ownership: who allocates, who frees.

Step 5. Error Handling

- Always check function return values:

```
if (fwrite(buf, 1, len, file) != len) {
    perror("fwrite");
}
```

- Use meaningful error messages.
- Prefer returning error codes over silently ignoring failures.
- For libraries, use `errno` or your own error enums.

Step 6. Undefined Behavior Prevention

- No uninitialized variables.
- No out-of-bounds array access.
- No signed integer overflow.
- No use-after-free.
- Compile with:

```
gcc -Wall -Wextra -Wpedantic -fsanitize=undefined,address
```

Fix all warnings, treat them as errors.

Step 7. Portability

- Don't assume `int` is 32 bits or `char` is signed.
- Use `<stdint.h>` types (`int32_t`, `uint64_t`).
- Use `size_t` for sizes and indexing.
- Avoid platform-specific APIs unless wrapped.
- Test on multiple compilers (`gcc`, `clang`, `tinycc`).

Step 8. Testing and Validation

- Every function that can fail must have at least one test.
- Edge cases: empty input, zero values, large input.
- Tests should run automatically:

```
make test
```

- Compare results with known-good output.
- Document how to reproduce test results.

Step 9. Documentation

- Add a short header to every file:

```
/* math_utils.c
 * Simple math helpers.
 * Author: Your Name
 * License: MIT
 */
```

- Comment tricky code, but not the obvious.
- Maintain a `README.md` explaining build and run steps.
- Version your code (Git). Write meaningful commit messages.

Step 10. Tiny Code: Applying the Checklist

Let's check a small example:

Original:

```
int *make_array(int n){
  int arr[n]; for(int i=0;i<n;i++) arr[i]=i; return arr;
}
```

Review notes:

- Returns pointer to local array → UB
- Magic loop style → unreadable
- Missing input validation

Fixed:

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

int *make_array(int n) {
    if (n <= 0) return NULL;
    int *arr = malloc(n * sizeof(int));
    if (!arr) { perror("malloc"); return NULL; }
    for (int i = 0; i < n; i++) arr[i] = i;
    return arr;
}
```

Passes checklist

- Safe, clear, and portable.

Why It Matters

A checklist builds **discipline and consistency**. It ensures:

- Clean, maintainable code.
- Fewer crashes and leaks.
- Easier debugging and onboarding.
- Long-term stability in complex systems.

Every high-quality C project uses one, from open-source libraries to kernels.

Try It Yourself

1. Take one of your old C programs and review it using this checklist.
2. Fix memory leaks, add error checks, clean up naming.
3. Compile with all warnings on.
4. Run it through AddressSanitizer or Valgrind.
5. Document everything, then repeat for your next project.

Next, you'll wrap up this debugging chapter with a **hands-on practice session**: fixing real memory and logic bugs step by step.

80. Practice: Fix Memory and Logic Bugs

Now it's time to apply everything you've learned, debugging, testing, assertions, logging, and analysis, to *real code that's broken*. This section walks you through a handful of small, common C bugs that new programmers (and even experienced ones) run into, showing how to find, understand, and fix them.

Step 1. Bug #1, Segmentation Fault from a Bad Pointer

Buggy Code:

```c
#include <stdio.h>

int main(void) {
    int *p;
    *p = 10;  // writing to uninitialized pointer
    printf("%d\n", *p);
}
```

Symptom:

```
Segmentation fault (core dumped)
```

Diagnosis:

- The pointer `p` is never initialized.
- It points to an undefined address.

Fix:

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    int *p = &x;
    printf("%d\n", *p);
}
```

Lesson: Always initialize pointers before use. If dynamic, allocate with `malloc()` and check for `NULL`.

Step 2. Bug #2, Memory Leak

Buggy Code:

```c
#include <stdlib.h>

void leak(void) {
    int *arr = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++) arr[i] = i;
    // forgot to free
}

int main(void) {
    for (int i = 0; i < 10000; i++) leak();
}
```

Diagnosis: Each call to `leak()` allocates memory and never frees it. Use Valgrind to confirm:

```
valgrind ./a.out
```

Fix:

```c
void leak(void) {
    int *arr = malloc(10 * sizeof(int));
    if (!arr) return;
    for (int i = 0; i < 10; i++) arr[i] = i;
    free(arr);
}
```

Lesson: Every `malloc()` needs a matching `free()`, no exceptions.

Step 3. Bug #3, Off-by-One Error

Buggy Code:

```c
#include <stdio.h>

int main(void) {
    int nums[5] = {0, 1, 2, 3, 4};
    for (int i = 0; i <= 5; i++)  //   should be < 5
        printf("%d ", nums[i]);
}
```

Symptom: Sometimes prints garbage or segfaults.

Fix:

```c
for (int i = 0; i < 5; i++)
```

Lesson: Off-by-one errors are the most common bug in loops. Always check your boundary conditions carefully.

Step 4. Bug #4, Use-After-Free

Buggy Code:

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

int main(void) {
    int *x = malloc(sizeof(int));
    *x = 5;
```

```
    free(x);
    printf("%d\n", *x);  //  accessing freed memory
}
```

Fix:

```
free(x);
x = NULL;
```

Now:

```
if (x) printf("%d\n", *x);
```

Lesson: Once you free memory, it's no longer yours, never touch it again.

Step 5. Bug #5, Stack Variable Escaping Scope

Buggy Code:

```
int *make_ptr(void) {
    int x = 10;
    return &x;  //  pointer to local variable
}

int main(void) {
    int *p = make_ptr();
    printf("%d\n", *p);  // UB
}
```

Fix:

```
int *make_ptr(void) {
    int *x = malloc(sizeof(int));
    *x = 10;
    return x;
}
```

and remember to `free(p)` later.

Lesson: Never return the address of a local variable, its lifetime ends when the function returns.

Step 6. Bug #6, Missing Return Statement

Buggy Code:

```
int add(int a, int b) {
    int c = a + b;
    // forgot to return
}

int main(void) {
    printf("%d\n", add(2, 3));
}
```

Fix:

```
return c;
```

Lesson: If the function's return type is non-`void`, always return a value. Compile with `-Wall -Wextra` to catch this automatically.

Step 7. Bug #7, Uninitialized Variable

Buggy Code:

```
int sum(void) {
    int s;
    for (int i = 0; i < 3; i++) s += i;  // s not initialized
    return s;
}
```

Fix:

```
int s = 0;
```

Lesson: Initialize all variables before using them, especially accumulators.

Step 8. Bug #8, Mixing Signed and Unsigned

Buggy Code:

```c
#include <stdio.h>

int main(void) {
    int a = -1;
    unsigned int b = 1;
    if (a < b) printf("less\n"); else printf("greater\n");
}
```

Output:

```
greater
```

Explanation: `a` is converted to unsigned, so it becomes a large positive number.

Fix: Avoid mixing signed and unsigned types. Use explicit casts or consistent types.

Step 9. Bug #9, Buffer Overflow

Buggy Code:

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

int main(void) {
    char name[8];
    strcpy(name, "Superlongname");  //  too big
    printf("%s\n", name);
}
```

Fix:

```c
strncpy(name, "Superlongname", sizeof(name) - 1);
name[7] = '\0';
```

Lesson: Never trust input size, always use bounded functions.

Step 10. Bug #10, Floating-Point Comparison

Buggy Code:

```c
#include <stdio.h>

int main(void) {
    float a = 0.1f * 3;
    if (a == 0.3f) printf("Equal\n");
    else printf("Not equal\n");
}
```

Output:

```
Not equal
```

Fix:

```c
if (fabsf(a - 0.3f) < 1e-6) printf("Equal\n");
```

Lesson: Floating-point math is approximate, always compare with a tolerance.

Putting It All Together

You can combine all these techniques:

- Use **assertions** to catch impossible states.
- Use **logging** to trace events.
- Run **Valgrind** or **ASan** to detect memory bugs.
- Use **unit tests** to verify correctness.
- And if it still crashes, **analyze the core dump**.

Every debugging tool is a lens. Use them together to see clearly.

Why It Matters

Debugging teaches you *how programs fail*. Each bug fixed makes you a more confident systems engineer. C doesn't forgive mistakes, but it rewards precision.

Try It Yourself

1. Write a program containing at least 3 of these bugs.
2. Run it under AddressSanitizer (`-fsanitize=address`).
3. Fix each bug one by one.
4. Document what caused it and what fixed it.
5. Make this a personal debugging kata, practice until no bug survives longer than 10 minutes.

Next, we'll begin **Chapter 9: Portable and Modern C**, where you'll learn how to write C that runs everywhere, from embedded chips to modern servers.

Chapter 9. Portable and Modern C

81. The C Standard Timeline (C89 to C23)

C has been around for more than fifty years, and it has evolved slowly and carefully. Every version of the C standard improves the language while keeping backward compatibility with decades of existing code.

Understanding the **timeline of C standards** helps you write portable, modern code and know which features are safe to use in your target environments.

Step 1. The Beginning, K&R C (1972–1989)

C was born at Bell Labs in the early 1970s, developed by **Dennis Ritchie** as a systems programming language for **Unix**. The first book, *The C Programming Language* by Kernighan and Ritchie (1978), informally defined "K&R C."

Key Traits:

- No standardization yet.
- Implicit function declarations.
- No `void` type for functions without return.
- No function prototypes (parameters not type-checked).
- Header files were optional.

Example:

```
main() {
    printf("Hello, world\n");
}
```

It was simple, direct, and dangerous, but it worked.

Step 2. ANSI C (C89 / C90)

In 1989, C became standardized by ANSI (and in 1990 by ISO). This version, **C89/C90**, unified compiler behavior and made C portable across systems.

Key Features:

- Function prototypes (`int add(int, int);`)
- Standard headers (`<stdio.h>`, `<stdlib.h>`, `<string.h>`)
- `void` keyword
- Type qualifiers: `const`, `volatile`
- New library functions (`memcpy`, `qsort`, `assert`)
- Formalized the standard library
- Single-line comments were still not supported (use `/* */`)

Tiny Code:

```c
#include <stdio.h>

int add(int a, int b) { return a + b; }

int main(void) {
    printf("%d\n", add(2, 3));
}
```

Step 3. C95 (ISO Amendment)

A minor update that refined C90, rarely mentioned but still significant.

Added:

- Wide character support (`<wchar.h>`)
- Multibyte strings
- More internationalization utilities
- Macros like `__STDC_VERSION__`

It paved the way for Unicode support in later versions.

Step 4. C99, Modernization Begins

C99 (published in 1999) was the biggest update since the beginning.

Major Improvements:

- `//` single-line comments
- Variable declarations anywhere
- Inline functions
- `long long` (64-bit integer)
- `stdbool.h` for `bool`, `true`, `false`
- `stdint.h` for fixed-width integers (`int32_t`, `uint64_t`)
- Designated initializers and compound literals
- Flexible array members
- `snprintf` safer string formatting
- Variable-length arrays (VLAs)

Tiny Code:

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

int main(void) {
    bool done = false;
    uint64_t sum = 0;
    for (int i = 0; i < 5; i++)
        sum += i;
    printf("%llu\n", (unsigned long long)sum);
    return done;
}
```

C99 made C feel modern, introducing safer and more expressive syntax.

Step 5. C11, Concurrency and Safety

Released in 2011, **C11** added better threading and safety mechanisms.

Key Additions:

- `_Thread_local` storage specifier
- `<threads.h>` for portable threads, mutexes, condition variables
- `_Atomic` for atomic operations

- `_Static_assert` for compile-time checks
- Bounds-checked functions (`strcpy_s`, `memcpy_s`)
- Optional Annex K for safer standard library functions
- Improved Unicode and wide character support

Tiny Code:

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

int run(void *arg) {
    printf("Hello from thread %d\n", *(int *)arg);
    return 0;
}

int main(void) {
    int id = 1;
    thrd_t t;
    thrd_create(&t, run, &id);
    thrd_join(t, NULL);
}
```

C11 made C safer and concurrency-aware, though not all compilers implemented `<threads.h>` fully.

Step 6. C17 (a.k.a. C18), The Refinement

Officially ISO/IEC 9899:2018 (published in 2018), C17 fixed inconsistencies and bugs in C11 but didn't add new features.

Highlights:

- Clarifications to atomics, macros, and UB rules.
- Improved compatibility with C++ compilers.
- Bug fixes in the standard library.
- `__STDC_VERSION__` is `201710L`.

It's the default "stable" standard for modern C codebases.

Step 7. C23, The Latest Standard

C23 is the most recent (published in 2024), continuing modernization without breaking backward compatibility.

Major Features:

- `typeof` (like in GCC)
- `nullptr` keyword
- `static_assert` (alias for `_Static_assert`)
- UTF-8 string literals: `u8"Hello"`
- New standard attributes (`[[maybe_unused]]`, `[[nodiscard]]`)
- `constexpr`-like features (`constexpr` functions are planned)
- Better Unicode and formatting APIs
- Safer library extensions
- Improved interoperability with C++

Tiny Code:

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    [[maybe_unused]] int y = 20;
    static_assert(sizeof(int) == 4, "Expected 4-byte int");
    printf("%d\n", x);
}
```

C23 brings C closer to modern C++ and Rust-style safety while staying simple and lightweight.

Step 8. Checking Your Compiler's Version

You can check your compiler's supported C standard using:

```
gcc -dM -E - < /dev/null | grep __STDC_VERSION__
```

Common outputs:

```
199901L  → C99
201112L  → C11
201710L  → C17
202311L  → C23
```

Or compile with:

```
gcc -std=c99 program.c
gcc -std=c11 program.c
gcc -std=c23 program.c
```

Step 9. Compatibility and Portability Tips

- Always declare the standard explicitly: `-std=c11` or `-std=c17`.
- Avoid compiler-specific extensions unless guarded with `#ifdef __GNUC__`.
- Use standard headers like `<stdint.h>` and `<stdbool.h>`.
- When writing libraries, prefer the lowest standard that supports your needs.
- Add `_Static_assert` or `#error` for unsupported standards.

Step 10. Tiny Code: Version Detector

```
#include <stdio.h>

int main(void) {
#if __STDC_VERSION__ >= 202311L
    printf("C23 or newer\n");
#elif __STDC_VERSION__ >= 201710L
    printf("C17\n");
#elif __STDC_VERSION__ >= 201112L
    printf("C11\n");
#elif __STDC_VERSION__ >= 199901L
    printf("C99\n");
#else
    printf("C90 or earlier\n");
#endif
}
```

Compile and run to see what your compiler supports.

Why It Matters

C's evolution shows its unique philosophy: **change slowly, but never break old code.** Knowing which standard you target means you can use modern features confidently, without losing portability.

Try It Yourself

1. Write the version detector program above and run it with `-std=c99`, `-std=c11`, and `-std=c23`.
2. Experiment with `_Static_assert` and `_Thread_local`, see which standards support them.
3. Try compiling a small thread example using `<threads.h>`.
4. Look up your compiler's documentation to see which features of C23 are implemented.
5. Pick one feature (like `[[nodiscard]]`) and use it in a tiny project.

Next, you'll explore **portability and endianness**, the invisible details that determine how your C programs behave across different machines and architectures.

82. Portability and Endianness

Portability means your C program behaves the same way everywhere, on Linux, Windows, ARM, x86, or even a tiny microcontroller. Writing portable code is one of the hardest and most important skills in systems programming.

This section helps you understand the biggest low-level trap of all: **endianness**, and how to write code that runs safely across architectures.

Step 1. What Is Portability?

A **portable C program** is one that:

- Compiles cleanly with different compilers.
- Runs correctly on 32-bit, 64-bit, and embedded systems.
- Does not assume details of CPU, OS, or compiler behavior.

Portability depends on respecting what the C standard guarantees, and avoiding assumptions that might be true only on *your* machine.

Step 2. Why Portability Matters

You might write a C program on macOS (little-endian x86_64) and later need to run it on:

- A Raspberry Pi (ARM, also little-endian)
- A big-endian PowerPC router
- An embedded MIPS controller

If your program reads or writes binary data, it must handle **endianness**, or the same file may be misread on another architecture.

Step 3. Understanding Endianness

Endianness defines how bytes of multibyte values are stored in memory.

Type	Description	Memory (4-byte int = 0x12345678)
Little-endian	Least significant byte first	`78 56 34 12`
Big-endian	Most significant byte first	`12 34 56 78`

Intel and ARM (in most modes) are **little-endian**. Many older CPUs (PowerPC, SPARC) are **big-endian**.

C does not define the byte order, it depends on the platform.

Step 4. Checking Endianness at Runtime

```c
#include <stdio.h>

int main(void) {
    unsigned int x = 0x12345678;
    unsigned char *p = (unsigned char *)&x;

    if (*p == 0x78)
        printf("Little-endian\n");
    else
        printf("Big-endian\n");
}
```

Explanation: The pointer `p` reads the lowest memory byte. If it contains the least significant byte (`0x78`), it's little-endian.

Step 5. Converting Between Endiannesses

Use standard POSIX functions to handle conversions safely:

```c
#include <arpa/inet.h>  // or <winsock2.h> on Windows
#include <stdint.h>
#include <stdio.h>

int main(void) {
```

```
    uint32_t x = 0x12345678;
    uint32_t y = htonl(x); // Host to Network Long (big-endian)
    printf("0x%x -> 0x%x\n", x, y);
}
```

Functions:

- `htons` – host to network short (16-bit)
- `htonl` – host to network long (32-bit)
- `ntohs` – network to host short
- `ntohl` – network to host long

Network byte order is always **big-endian**.

Step 6. Handling Portability in File Formats

If you serialize structs directly to disk:

```
fwrite(&header, sizeof(header), 1, file);
```

you're likely writing machine-dependent data:

- Endianness may differ.
- Padding and alignment may differ.
- Structure layout can vary by compiler.

Better approach: Write each field individually in a well-defined order:

```
uint32_t size_net = htonl(header.size);
fwrite(&size_net, sizeof(size_net), 1, file);
```

Now, any machine can read your file by reversing the conversion (`ntohl`).

Step 7. Data Type Size Differences

Type sizes vary across systems:

Type	Typical 32-bit	Typical 64-bit
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes

Type	Typical 32-bit	Typical 64-bit
`int`	4 bytes	4 bytes
`long`	4 bytes	8 bytes
`long long`	8 bytes	8 bytes
`void*`	4 bytes	8 bytes

Use `<stdint.h>` types (`int32_t`, `uint64_t`, etc.) for predictable sizes.

Step 8. Alignment and Padding

The compiler may insert padding between structure fields for speed or alignment.

Example:

```
struct Example {
    char a;
    int b;
};
```

On most systems:

- `sizeof(struct Example)` = 8, not 5 (3 bytes of padding).

To make portable formats:

- Use `#pragma pack(1)` (non-standard) or serialize field-by-field.
- Never assume `sizeof(struct)` is the same across systems.

Step 9. Compiler and OS Differences

Be careful with:

- Path separators (`/` vs `\\`)
- Newline conventions (`\n` vs `\r\n`)
- `#include <unistd.h>` (POSIX only)
- `system()` commands (OS-specific)
- Thread APIs (`pthread` vs Windows threads)
- Socket APIs (`<arpa/inet.h>` vs `<winsock2.h>`)

Use conditional compilation:

```
#ifdef _WIN32
#include <winsock2.h>
#else
#include <arpa/inet.h>
#endif
```

Step 10. Tiny Code: Writing Portable Binary I/O

```
#include <stdio.h>
#include <stdint.h>
#include <arpa/inet.h>

int main(void) {
    FILE *f = fopen("num.bin", "wb");
    uint32_t n = 0x12345678;
    uint32_t net = htonl(n);
    fwrite(&net, sizeof(net), 1, f);
    fclose(f);

    f = fopen("num.bin", "rb");
    uint32_t read_net;
    fread(&read_net, sizeof(read_net), 1, f);
    fclose(f);
    printf("Read back: 0x%x\n", ntohl(read_net));
}
```

This program writes and reads a 32-bit integer in **portable big-endian form**, the same bytes on any machine.

Why It Matters

Portability ensures your software lives longer than your hardware. A portable program:

- Runs on different CPUs and OSes.
- Shares data safely across architectures.
- Builds trust in your code across teams and systems.

Portability is a kind of professionalism, future-proofing your code.

Try It Yourself

1. Write a program that detects and prints system endianness.
2. Serialize a struct to a binary file, then deserialize it on another system.
3. Use `htonl` and `ntohl` to ensure data stays consistent.
4. Compile your code with both GCC and Clang.
5. Test it on both 32-bit and 64-bit architectures.

Next, you'll explore **inline assembly and hardware access**, the bridge between pure C and the underlying CPU instructions.

83. Inline Assembly and Hardware Access

C gives you precise control over memory and performance, but sometimes you need to go one level deeper, directly to the **CPU**. That's where **inline assembly** comes in: embedding assembly language inside your C code to optimize performance or access hardware-level features.

This chapter will show how to mix C and assembly safely, portably, and meaningfully.

Step 1. What Is Inline Assembly?

Inline assembly lets you insert small snippets of machine instructions into your C program. You can use it to:

- Access CPU instructions not exposed by C.
- Optimize performance-critical paths.
- Implement hardware drivers or low-level routines.

However, it's also **non-portable** and compiler-specific, so use it sparingly and isolate it behind clean C interfaces.

Step 2. Two Common Flavors

1. **GCC / Clang syntax (AT&T or Intel style)** Uses the `asm` or `__asm__` keyword.
2. **MSVC syntax** Uses `__asm { ... }` inside functions.

We'll focus on GCC/Clang syntax, since it's used in most systems programming contexts.

Step 3. Basic Inline Assembly Example

Tiny Code: Print CPU ID register (x86 only)

```
#include <stdio.h>

int main(void) {
    unsigned int eax, ebx, ecx, edx;
    eax = 0;
    __asm__ __volatile__(
        "cpuid"
        : "=a"(eax), "=b"(ebx), "=c"(ecx), "=d"(edx)
        : "a"(0)
    );
    printf("CPU Vendor: %.4s%.4s%.4s\n",
           (char*)&ebx, (char*)&edx, (char*)&ecx);
}
```

Explanation:

- `cpuid` is a CPU instruction that fills registers with information.
- `"=a"(eax)` means "store the output of register `eax` into variable `eax`."
- `: "a"(0)` means "put 0 into `eax` before running the instruction."
- The `__volatile__` keyword tells the compiler not to optimize it away.

Step 4. GCC Inline Assembly Syntax

General form:

```
asm volatile ("instruction list"
              : output_operands
              : input_operands
              : clobbered_registers);
```

Example:

```
asm volatile ("addl %%ebx, %%eax"
              : "=a"(result)
              : "a"(x), "b"(y));
```

Explanation:

- `"addl %%ebx, %%eax"`, assembly instruction
- `"=a"(result)`, output in `eax` goes to `result`
- `"a"(x), "b"(y)`, inputs: put `x` in `eax`, `y` in `ebx`

Step 5. Reading CPU Cycle Counters

Tiny Code: Measure CPU cycles between operations

```
#include <stdio.h>

unsigned long long rdtsc(void) {
    unsigned int lo, hi;
    __asm__ __volatile__("rdtsc" : "=a"(lo), "=d"(hi));
    return ((unsigned long long)hi << 32) | lo;
}

int main(void) {
    unsigned long long start = rdtsc();
    for (volatile int i = 0; i < 1000000; i++);
    unsigned long long end = rdtsc();
    printf("Cycles: %llu\n", end - start);
}
```

Explanation:

- `rdtsc` reads the CPU's timestamp counter.
- It's a precise measure of time in CPU cycles, great for microbenchmarking.

Step 6. Writing to I/O Ports (Embedded or Kernel Context)

If you're writing embedded code or OS kernels, you often interact with hardware registers directly.

Example (x86, privileged mode only):

```
static inline void outb(unsigned short port, unsigned char value) {
    __asm__ __volatile__("outb %0, %1" : : "a"(value), "Nd"(port));
}
```

This writes a byte to an I/O port, used for devices like serial ports, timers, or PIC controllers.

In user-space, you generally can't do this (needs kernel privileges).

Step 7. Memory Barriers and CPU Fences

When working with concurrency or hardware, you may need to control instruction ordering.

```
__asm__ __volatile__("mfence" ::: "memory");
```

This tells the CPU and compiler not to reorder memory operations, essential for writing thread-safe or device-control code at the hardware level.

Step 8. Register Constraints

GCC lets you specify which registers to use.

Constraint	Register	Meaning
`"a"`	`eax`	accumulator
`"b"`	`ebx`	base
`"c"`	`ecx`	counter
`"d"`	`edx`	data
`"S"`	`esi`	source index
`"D"`	`edi`	destination index

Example:

```
asm("mul %1" : "=a"(res) : "r"(x));
```

The `"r"` constraint lets the compiler choose any register.

Step 9. Mixing Assembly and C Functions

You can write small routines in separate `.S` files (pure assembly) and call them from C:

```
# file: add.S
.global add_two
add_two:
    addl %esi, %edi
    movl %edi, %eax
    ret
```

Then in C:

```c
int add_two(int a, int b);
int main(void) {
    printf("%d\n", add_two(5, 7));
}
```

This hybrid style is used in OS kernels, bootloaders, and math libraries.

Step 10. Tiny Code: Inline Assembly Add Function

```c
#include <stdio.h>

int add_fast(int a, int b) {
    int result;
    __asm__ ("addl %1, %0" : "=r"(result) : "r"(b), "0"(a));
    return result;
}

int main(void) {
    printf("%d\n", add_fast(3, 5));
}
```

The `"0"(a)` constraint tells the compiler to use the same register for input and output.

Why It Matters

Inline assembly teaches you what really happens beneath your C code. Even if you rarely use it, understanding it helps you:

- Read compiler-generated assembly (`gcc -S`)
- Optimize performance-critical code
- Understand how system calls, context switches, and kernel traps work

It's where software meets hardware, the true metal of computing.

Try It Yourself

1. Write a small inline assembly snippet that swaps two integers.
2. Print the CPU vendor string with `cpuid`.
3. Use `rdtsc()` to benchmark your function.

4. Inspect compiler-generated assembly using `gcc -S`.
5. Try to reimplement a basic math operation in assembly and compare performance.

Next, you'll learn **cross-compilation**, how to build your C programs for *other architectures and systems*, from your own machine.

84. Cross-Compilation

Cross-compilation means **building a program on one machine so it runs on another**. If you've ever compiled a C program on your laptop and deployed it to a Raspberry Pi, an ESP32 board, or even a custom Linux image, you've done cross-compilation.

This is an essential skill for systems programmers, embedded developers, and anyone who builds for multiple architectures or operating systems.

Step 1. What Is a Cross-Compiler?

A **cross-compiler** is a compiler that produces executables for a *target platform* different from the *host platform*.

Term	Meaning
Host	The system where you build the code
Target	The system where the program will run
Build	The system where the compiler itself was built (often same as host)

Example: You're on macOS (x86_64) and want to compile for a Raspberry Pi (ARM). Your toolchain must translate x86 instructions into ARM ones.

Step 2. Why Cross-Compile?

- Deploy software to embedded devices without compiling directly on them.
- Build for multiple architectures from one workstation.
- Generate portable binaries (for ARM, MIPS, RISC-V, etc.).
- Prepare static binaries for minimal systems or containers.

Cross-compilation is the foundation of **embedded Linux**, **IoT**, and **firmware development**.

Step 3. Installing a Cross-Compiler

On Linux, install a toolchain package for your target architecture. Examples:

```
sudo apt install gcc-arm-linux-gnueabihf
sudo apt install gcc-aarch64-linux-gnu
sudo apt install gcc-riscv64-linux-gnu
```

Each toolchain contains:

- `gcc` or `clang` cross-compiler
- `as` (assembler)
- `ld` (linker)
- target system headers and libraries

Step 4. Verifying the Target

Check your compiler's target triple:

```
arm-linux-gnueabihf-gcc -v
```

Output example:

```
Target: arm-linux-gnueabihf
```

The **triple** encodes:

```
<architecture>-<vendor>-<OS>-<ABI>
```

For instance:

- `x86_64-pc-linux-gnu`
- `arm-none-eabi` (bare-metal, no OS)
- `aarch64-linux-gnu`

Step 5. Compiling for Another Platform

Tiny Code:

```c
#include <stdio.h>

int main(void) {
    printf("Hello from cross-compiled C!\n");
}
```

Compile for ARM:

```
arm-linux-gnueabihf-gcc hello.c -o hello_arm
```

Transfer it to your ARM device:

```
scp hello_arm user@raspberrypi.local:/home/user/
```

Then run on the Pi:

```
./hello_arm
```

If everything's configured correctly, you'll see:

```
Hello from cross-compiled C!
```

Step 6. Static vs Dynamic Linking

When cross-compiling, your target system might not have the same libraries. You can link everything into one binary:

```
arm-linux-gnueabihf-gcc -static hello.c -o hello_static
```

Static linking ensures the binary runs even if the target lacks shared libraries, useful for minimal or embedded systems.

Step 7. Using Clang for Cross-Compilation

Clang simplifies multi-target builds through `--target` and `--sysroot` options:

```
clang --target=aarch64-linux-gnu --sysroot=/path/to/sysroot hello.c -o hello_arm64
```

`--sysroot` points to a directory that mimics the target's filesystem, containing its headers and libraries.

Step 8. Building for Windows or macOS from Linux

You can also cross-compile across operating systems:

Linux → Windows:

```
sudo apt install mingw-w64
x86_64-w64-mingw32-gcc hello.c -o hello.exe
```

Linux → macOS: More complex, usually requires Clang with Apple SDKs or `osxcross`.

Step 9. Automating with CMake or Makefiles

CMake makes multi-platform builds easy.

`toolchain-arm.cmake`

```
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR arm)
SET(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
```

Then:

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchain-arm.cmake ..
make
```

Your build system now knows it's cross-compiling for ARM.

Step 10. Tiny Code: Detect and Print Target Architecture

```
#include <stdio.h>

int main(void) {
#if defined(__x86_64__)
    printf("x86_64\n");
#elif defined(__aarch64__)
    printf("ARM64\n");
#elif defined(__arm__)
    printf("ARM 32-bit\n");
#elif defined(__riscv)
```

```
    printf("RISC-V\n");
#else
    printf("Unknown architecture\n");
#endif
}
```

Compile for different targets and observe the output.

Why It Matters

Cross-compilation connects your laptop to every other device you'll ever program. It's how kernel modules, embedded systems, and even Android apps are built. Once you learn it, you can build **anywhere, for anything**.

Try It Yourself

1. Install an ARM or RISC-V cross-compiler on your host system.
2. Cross-compile and run a simple "Hello" binary on an emulator (`qemu-arm`).
3. Use the `-static` flag to make it self-contained.
4. Add a Makefile with variables `CC`, `CFLAGS`, and `LDFLAGS` for easy reuse.
5. Try building both for Linux and Windows from the same source.

Next, you'll explore **threading with pthreads**, how to run multiple parts of your program at the same time using standard C threads.

85. Threading with pthreads

Modern computers run many things at once. Your web browser, text editor, and compiler all share CPU time through **threads**. In C, the most widely used threading API is **POSIX threads**, or **pthreads**. It's low-level, portable, and gives you fine-grained control over parallel execution.

This section will teach you how to create, manage, and synchronize threads safely.

Step 1. What Is a Thread?

A **thread** is a lightweight execution unit that shares the same memory space as other threads in a process.

Process	Thread
Has its own memory (stack, heap, code)	Shares memory with other threads
Created by OS	Created by process
Expensive to start	Cheap and fast to start
Communicates via IPC	Communicates via shared memory

Threads are ideal for tasks like handling multiple network requests, performing parallel computation, or keeping a UI responsive.

Step 2. Including pthreads

To use pthreads, include the header:

```
#include <pthread.h>
```

When compiling, link with the pthread library:

```
gcc program.c -o program -lpthread
```

Step 3. Creating Threads

Each thread runs a separate function. The function must take and return `void *`.

Tiny Code: Basic Thread Creation

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

void* task(void* arg) {
    printf("Hello from thread! Arg = %d\n", *(int*)arg);
    return NULL;
}

int main(void) {
    pthread_t thread;
    int value = 42;

    pthread_create(&thread, NULL, task, &value);
    pthread_join(thread, NULL);
```

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

Output:

```
Hello from thread! Arg = 42
Main thread finished.
```

Explanation:

- `pthread_create` starts a new thread.
- `pthread_join` waits for it to finish.

Step 4. Multiple Threads

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

void* work(void* arg) {
    int id = *(int*)arg;
    printf("Thread %d running\n", id);
    return NULL;
}

int main(void) {
    pthread_t threads[3];
    int ids[] = {1, 2, 3};

    for (int i = 0; i < 3; i++)
        pthread_create(&threads[i], NULL, work, &ids[i]);

    for (int i = 0; i < 3; i++)
        pthread_join(threads[i], NULL);

    printf("All threads done.\n");
}
```

Output order may vary, threads run concurrently.

Step 5. Race Conditions

When two threads modify the same variable at the same time, bad things happen. This is called a **race condition**.

Example (unsafe):

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

int counter = 0;

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++)
        counter++;
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d\n", counter);
}
```

Expected: `200000` Actual: unpredictable (e.g. `137421`), because increments overlap.

Step 6. Using Mutexes (Mutual Exclusion Locks)

A **mutex** ensures that only one thread modifies shared data at a time.

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

int counter = 0;
pthread_mutex_t lock;

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);
```

```
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&lock);
    printf("Counter = %d\n", counter);
}
```

Now the output will consistently be 200000.

Step 7. Condition Variables

Condition variables let threads wait for a signal. They're used to coordinate producer–consumer models.

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

pthread_mutex_t lock;
pthread_cond_t cond;
int ready = 0;

void* worker(void* arg) {
    pthread_mutex_lock(&lock);
    while (!ready)
        pthread_cond_wait(&cond, &lock);
    printf("Worker got the signal!\n");
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
```

```
    pthread_t t;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_create(&t, NULL, worker, NULL);

    sleep(1);
    pthread_mutex_lock(&lock);
    ready = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&lock);

    pthread_join(t, NULL);
}
```

Step 8. Thread Attributes

You can control thread behavior using `pthread_attr_t`:

- Stack size
- Detach state (joinable or detached)
- Scheduling policy

Example:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&thread, &attr, task, NULL);
pthread_attr_destroy(&attr);
```

Detached threads free resources automatically when done.

Step 9. Thread Safety and Best Practices

- Protect all shared data with mutexes.
- Avoid global variables when possible.
- Use thread-safe functions (`strtok_r` instead of `strtok`).
- Keep critical sections short.
- Join or detach all threads before program exit.

Step 10. Tiny Code: Parallel Sum

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

#define N 4

int partial[4];

void* compute(void* arg) {
    int id = *(int*)arg;
    int start = id * 25;
    int sum = 0;
    for (int i = start; i < start + 25; i++)
        sum += i;
    partial[id] = sum;
    return NULL;
}

int main(void) {
    pthread_t threads[N];
    int ids[N];
    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, compute, &ids[i]);
    }

    int total = 0;
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
        total += partial[i];
    }

    printf("Total sum = %d\n", total);
}
```

This program splits a task across multiple threads and combines results.

Why It Matters

Threads make your programs faster, more responsive, and scalable. They allow C to fully exploit modern multi-core CPUs, from servers to embedded systems. Learning pthreads means learning how real systems multitask efficiently and safely.

Try It Yourself

1. Write a program that starts 5 threads, each printing its ID.
2. Add a shared counter and protect it with a mutex.
3. Implement a producer–consumer queue using condition variables.
4. Use `pthread_attr_t` to create detached worker threads.
5. Profile your program's performance as you increase the thread count.

Next, you'll explore **atomic operations and memory models**, how modern CPUs ensure consistency when multiple threads share data without locks.

86. Atomic Operations and Memory Models

When multiple threads share data, you usually protect that data with **locks** like `pthread_mutex_t`. But sometimes, you need something faster, a way to perform an update that can't be interrupted, even across threads. That's where **atomic operations** come in.

This section introduces atomic operations in C and how the **memory model** ensures your program behaves predictably across cores.

Step 1. What Does "Atomic" Mean?

An **atomic operation** is one that happens all at once, it can't be divided or interrupted.

Example idea: If two threads both run `counter++` at the same time:

- Without atomicity → race condition.
- With atomicity → one thread's update completes fully before the other starts.

Atomic operations are essential in lock-free algorithms, concurrent queues, and reference counters.

Step 2. The Problem with `counter++`

This line looks simple:

```
counter++;
```

But under the hood, it's three separate steps:

1. Load `counter` from memory.
2. Increment it.
3. Store it back.

Two threads doing this at once can lose updates:

```
Thread A: load(5)
Thread B: load(5)
Thread A: store(6)
Thread B: store(6)
```

Result: one increment lost, final value should be 7 but ends up 6.

Step 3. Using Atomic Types

C11 introduced `<stdatomic.h>`, a portable way to use atomic operations.

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

int main(void) {
    atomic_int counter = 0;
    atomic_fetch_add(&counter, 1);
    atomic_fetch_add(&counter, 1);
    printf("%d\n", counter); // 2
}
```

No locks. No race conditions. The `atomic_*` functions guarantee the operations are atomic at the hardware level.

Step 4. Common Atomic Functions

Function	Description
`atomic_load`	Read atomically
`atomic_store`	Write atomically
`atomic_fetch_add`	Add and return old value
`atomic_fetch_sub`	Subtract and return old value
`atomic_exchange`	Replace and return old value
`atomic_compare_exchange_strong`	Compare-and-swap

Example:

```
atomic_compare_exchange_strong(&counter, &expected, desired);
```

If `counter == expected`, replace it with `desired`. Otherwise, update `expected` with the current value.

Step 5. Tiny Code: Atomic Counter with Threads

```c
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>

atomic_int counter = 0;

void* work(void* arg) {
    for (int i = 0; i < 100000; i++)
        atomic_fetch_add(&counter, 1);
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, work, NULL);
    pthread_create(&t2, NULL, work, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d\n", counter);
}
```

Output is always 200000, with no mutexes and no data races.

Step 6. Relaxed vs Sequential Consistency

Atomic operations can have **different memory orders**. By default, they're *sequentially consistent*, the strongest and safest ordering.

Memory Order	Meaning
`memory_order_seq_cst`	Global consistent order (default)
`memory_order_relaxed`	Only atomicity guaranteed
`memory_order_acquire`	Prevent reordering before load
`memory_order_release`	Prevent reordering after store
`memory_order_acq_rel`	Acquire + Release combo

Example:

```
atomic_fetch_add_explicit(&counter, 1, memory_order_relaxed);
```

This is faster but weaker, use only when you understand your memory model.

Step 7. Memory Barriers and Visibility

Modern CPUs reorder reads/writes for performance. Atomics, fences, and locks control *when* updates become visible to other threads.

Example: Thread A writes `ready = 1`. Thread B waits until it sees `ready == 1`. If the compiler reorders memory operations, Thread B might not see the change.

Use:

```
atomic_thread_fence(memory_order_seq_cst);
```

to prevent reordering across the fence.

Step 8. Compare-and-Swap (CAS)

CAS is the backbone of lock-free data structures.

```
int expected = 0;
int desired = 1;
if (atomic_compare_exchange_strong(&counter, &expected, desired)) {
    printf("Swapped!\n");
}
```

It atomically checks if `counter == expected` and updates it, all in one instruction. This is used to build things like spinlocks, queues, and reference counters.

Step 9. Spinlocks with Atomics

A **spinlock** keeps checking until it can acquire the lock.

```
#include <stdatomic.h>
#include <unistd.h>

atomic_flag lock = ATOMIC_FLAG_INIT;

void lock_spin(void) {
    while (atomic_flag_test_and_set(&lock))
        ; // busy wait
}

void unlock_spin(void) {
    atomic_flag_clear(&lock);
}
```

This is efficient when the lock is held for a very short time. For longer waits, use `pthread_mutex_t` instead.

Step 10. Tiny Code: Atomic Reference Counter

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

typedef struct {
    atomic_int refcount;
} Object;

void retain(Object* obj) {
    atomic_fetch_add(&obj->refcount, 1);
}

void release(Object* obj) {
    if (atomic_fetch_sub(&obj->refcount, 1) == 1)
        printf("Object freed\n");
```

```
}

int main(void) {
    Object obj = { .refcount = 1 };
    retain(&obj);
    release(&obj);
    release(&obj);
}
```

Output:

```
Object freed
```

This is how many real-world systems (e.g. file handles, shared memory) track usage.

Why It Matters

Atomic operations are the building blocks of lock-free programming. They allow you to write high-performance concurrent code without blocking other threads. The C memory model gives you guarantees to reason about correctness even across multiple CPU cores.

Try It Yourself

1. Replace a mutex counter with an atomic counter.
2. Implement a spinlock using `atomic_flag`.
3. Use `atomic_compare_exchange_strong` to build a simple CAS loop.
4. Test the difference between `memory_order_relaxed` and `seq_cst`.
5. Build a reference-counted structure using atomics.

Next, you'll explore **using C with other languages (FFI)**, how to make C libraries callable from Python, Rust, and Go.

87. Using C with Other Languages (FFI)

C is often called the *universal assembly language*, nearly every modern language can call into it. This is made possible through the **Foreign Function Interface (FFI)**, which defines how different languages talk to C code.

In this section, you'll learn how to expose your C functions to Python, Rust, and Go, and how to call functions from those languages *inside* C.

Step 1. What Is an FFI?

An **FFI (Foreign Function Interface)** is a bridge that lets programs written in one language use code written in another.

Why FFI matters:

- Reuse fast, low-level C libraries (e.g., OpenSSL, SQLite).
- Integrate C modules into higher-level languages like Python or Go.
- Extend existing programs without rewriting everything.
- Combine system-level control with productivity.

Step 2. The Foundation: C ABI

The **ABI (Application Binary Interface)** defines how function calls, parameters, and data structures are represented in memory. The FFI works because C has a stable and simple ABI.

Rules include:

- How arguments are passed (registers or stack).
- How return values are handled.
- How data types are aligned in memory.

That's why almost every language provides a way to "speak" the C ABI.

Step 3. Exposing C Functions to Other Languages

You can make your C functions callable by other languages by marking them with `extern "C"` (if compiling as C++) or just regular C functions otherwise.

Tiny Code: Shared C Library

```c
// file: mathlib.c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}
```

Compile it into a shared library:

```
gcc -shared -fPIC -o libmathlib.so mathlib.c
```

This creates a `.so` (Linux) or `.dll` (Windows) or `.dylib` (macOS) file you can load in other languages.

Step 4. Using C in Python (ctypes)

Python can call C functions directly using the `ctypes` module.

```
import ctypes

lib = ctypes.CDLL("./libmathlib.so")
print(lib.add(2, 3))
print(lib.multiply(4, 5))
```

Output:

```
5
20
```

Python automatically converts standard types (`int`, `float`, `char *`) to C equivalents.

For more complex types, you can define `ctypes.Structure` classes matching your C structs.

Step 5. Using C in Rust

Rust has a built-in `extern "C"` block for FFI.

Rust Example:

```
#[link(name = "mathlib")]
extern "C" {
    fn add(a: i32, b: i32) -> i32;
}

fn main() {
    unsafe {
        println!("{}", add(2, 3));
    }
}
```

Compile with:

```
rustc main.rs -L .
```

Rust enforces `unsafe` because it can't verify what happens inside the C function.

Step 6. Using C in Go

Go uses the `import "C"` directive for seamless C integration.

Go Example:

```
/*
#include "mathlib.c"
*/
import "C"
import "fmt"

func main() {
    fmt.Println(C.add(2, 3))
}
```

Compile and run:

```
go run main.go
```

Go will compile your C code behind the scenes and link it automatically.

Step 7. Calling Foreign Code from C

You can also go the other way, call functions from another language *inside C*.

Example: C calling Python

```
#include <Python.h>

int main(void) {
    Py_Initialize();
    PyRun_SimpleString("print('Hello from Python in C!')");
    Py_Finalize();
}
```

Compile with:

```
gcc main.c -o main $(python3-config --cflags --ldflags)
```

This embeds a Python interpreter in your C program, powerful for scripting or AI integration.

Step 8. Data Structures Across Languages

FFI works best with **simple, C-compatible types**:

- `int`, `double`, `char *`, and flat structs. Avoid C++ classes, pointers to complex structs, or variable-length arrays, they often don't translate cleanly.

Example:

```
typedef struct {
    int id;
    double score;
} Record;
```

You can use this struct easily from Python (`ctypes.Structure`) or Rust (`#[repr(C)] struct`).

Step 9. Memory Ownership Rules

Always define *who allocates and who frees memory.*

If C allocates something:

```
char* greet(void) {
    char* s = malloc(32);
    sprintf(s, "Hello from C!");
    return s;
}
```

Then the caller (e.g., Python) must call `free()` via FFI to avoid leaks. Never assume the garbage collector of another language will clean up C memory.

Step 10. Tiny Code: C Shared Library + Python

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

char* greet(const char* name) {
    static char buf[64];
    snprintf(buf, sizeof(buf), "Hello, %s!", name);
    return buf;
}
```

Compile:

```
gcc -shared -fPIC -o libgreet.so greet.c
```

Python:

```
import ctypes
lib = ctypes.CDLL("./libgreet.so")
lib.greet.restype = ctypes.c_char_p
print(lib.greet(b"World"))
```

Output:

```
Hello, World!
```

Why It Matters

FFI turns C into the foundation of the software world, your C code can power systems written in any language. This is how databases, OS kernels, and AI frameworks expose APIs across ecosystems. Understanding FFI means you can build *language bridges*, not just programs.

Try It Yourself

1. Write a simple C library (math, strings, or sorting).
2. Load it in Python using `ctypes` and call its functions.
3. Reuse the same library from Rust using `extern "C"`.
4. Embed Python in C for a scripting layer.
5. Think about which side should own and free memory.

Next, you'll explore **safer alternatives and modern C features**, bounds checking, static assertions, and ways to make C code more reliable.

88. Safer Alternatives (Bounds Checking, `_Static_assert`, and Modern C Safety Tools)

C gives you power and control, but also responsibility. Because C does not automatically protect you from memory errors, buffer overflows, or type misuse, you must add safety at the language and tool level.

This section explores modern safety features in **C11 to C23**, including **bounds checking**, **static assertions**, and practical habits for writing safer C.

Step 1. Why Safety Matters in C

C is fast because it trusts the programmer. That means:

- It doesn't check array bounds.
- It doesn't initialize memory automatically.
- It doesn't manage memory for you.

That trust is both the reason C is used for kernels and the reason it causes so many bugs. The goal is not to make C "safe by default," but to make your *use of C safe by design*.

Step 2. Safer Bounds Handling

A classic error in C:

```
char name[8];
strcpy(name, "HelloWorld"); // buffer overflow
```

This overwrites memory past `name` and causes undefined behavior.

Fix 1: Use bounded versions of functions

```
strncpy(name, "HelloWorld", sizeof(name) - 1);
name[sizeof(name) - 1] = '\0';
```

Fix 2: Use safer alternatives introduced in C11 Annex K (if your compiler supports them):

```
strcpy_s(name, sizeof(name), "Hello");
```

They automatically check bounds and return error codes. However, Annex K is *optional*, so not all compilers implement it.

Step 3. Tiny Code: Safe String Copy

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

int main(void) {
    char dst[8];
    strncpy(dst, "Example", sizeof(dst) - 1);
    dst[sizeof(dst) - 1] = '\0';
    printf("Safe copy: %s\n", dst);
}
```

Compile with:

```
gcc safe_copy.c -o safe_copy -Wall -Wextra -O2
```

The `-Wall -Wextra` flags warn about suspicious behavior early, one of your best "safety tools."

Step 4. `_Static_assert`: Compile-Time Checking

Introduced in **C11**, `_Static_assert` lets you validate conditions *before* the program even compiles.

Example:

```c
_Static_assert(sizeof(int) == 4, "This code requires 32-bit int");
```

If the condition fails, compilation stops with a clear message.

You can use it for:

- Checking structure layout
- Ensuring type sizes
- Verifying array lengths
- Enforcing invariants

Step 5. Safer Integer Operations

Integer overflow is undefined behavior in C. Example:

```c
int x = 2147483647 + 1; // overflow
```

Safer options:

- Use `unsigned` types when wrapping is intentional.
- Use compiler flags:
  - `-ftrapv` (GCC/Clang): trap on overflow.
  - `-fsanitize=undefined`: detect overflow at runtime.

Example:

```c
gcc -fsanitize=undefined -O2 -g check.c -o check
```

This will abort your program the moment an overflow occurs.

Step 6. Null Pointer and Resource Safety

Always check return values:

```c
FILE *f = fopen("data.txt", "r");
if (!f) {
    perror("Failed to open file");
    return 1;
}
```

For dynamic memory:

```c
char *p = malloc(100);
if (!p) {
    fprintf(stderr, "Out of memory\n");
    exit(1);
}
```

And always `free()` when done.

Step 7. Tools for Safety

Modern compilers and tools help you detect bugs early:

Tool	Purpose
AddressSanitizer (`-fsanitize=address`)	Detects buffer overflows, use-after-free
UndefinedBehaviorSanitizer (`-fsanitize=undefined`)	Detects integer and type errors
Valgrind	Checks for memory leaks and invalid accesses
clang-tidy	Static analysis and style checking
cppcheck	Portable static analyzer for C/C++

Example:

```
clang -fsanitize=address safe.c -o safe
./safe
```

If there's a bug, you'll get a detailed memory trace.

Step 8. Struct and Alignment Checks

Unintended padding can cause issues when serializing or working with hardware. You can assert layout at compile time:

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

struct Packet {
    char type;
    int id;
};

_Static_assert(offsetof(struct Packet, id) == 4, "Alignment mismatch");
```

This ensures your assumptions about memory layout are correct.

Step 9. Defensive Macros and Compile Flags

Protect yourself with compile-time options:

Flag	Purpose
`-Wall -Wextra`	Enable important warnings
`-Werror`	Treat warnings as errors
`-Wconversion`	Warn on implicit type conversions
`-fsanitize=address`	Detect memory safety issues
`-D_FORTIFY_SOURCE=2`	Add runtime buffer checks (glibc)
`-fstack-protector-strong`	Detect stack corruption
`-O2`	Optimize safely without risky transformations

Example:

```
gcc -Wall -Wextra -Werror -O2 -fstack-protector-strong -D_FORTIFY_SOURCE=2 main.c -o main
```

Step 10. Tiny Code: Using Static Assertions and Sanitizers

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

struct Data {
    int id;
    char name[16];
};

_Static_assert(sizeof(struct Data) <= 32, "Struct too large");

int main(void) {
    struct Data d = {42, "C Safety"};
    printf("%d %s\n", d.id, d.name);

    char buf[8];
    strncpy(buf, "Safe", sizeof(buf) - 1);
    buf[sizeof(buf) - 1] = '\0';

    printf("Buffer: %s\n", buf);
}
```

Compile with:

```
gcc -Wall -Wextra -fsanitize=address -O2 safe_program.c -o safe_program
```

This program will abort if memory safety is violated, giving you immediate feedback during testing.

Why It Matters

Safety doesn't make your code slower, it makes your software *trustworthy*. Even though C gives you sharp tools, the combination of static checks, compiler warnings, and runtime sanitizers can make your programs robust enough for production systems.

Try It Yourself

1. Add `_Static_assert` checks in your structs and constants.
2. Compile with `-Wall -Wextra -Werror` and fix all warnings.
3. Use AddressSanitizer to catch out-of-bounds bugs.
4. Test your program under Valgrind for leaks.
5. Try writing the same buggy code twice, once raw, once safe, and compare behavior.

Next, you'll explore **modern C style**, how to write clear, maintainable, and idiomatic code in the C23 era.

89. Modern Style: Clean and Readable C

C has been around for over 50 years, and yet it keeps evolving. Modern C (C11–C23) combines the power of low-level programming with safer syntax, cleaner idioms, and new features that make code easier to reason about.

This section will help you write **modern, readable, and maintainable C**, the kind of C that feels timeless.

Step 1. Think "Clarity Over Cleverness"

The golden rule of modern C is:

> Write for humans, not compilers.

Compilers can handle complexity, your teammates (and future you) can't.

Bad:

```
for (i = n; i--; ) a[i] = 0;
```

Good:

```
for (int i = 0; i < n; i++)
    a[i] = 0;
```

Readability and simplicity always win.

Step 2. Prefer Explicit Initialization

Always initialize your variables. Uninitialized memory is one of the biggest sources of bugs.

Bad:

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

Good:

```
int x = 0;
printf("%d\n", x);
```

Also initialize arrays and structs explicitly:

```
int arr[10] = {0};
struct Point p = { .x = 10, .y = 20 };
```

Step 3. Use `const` Generously

`const` communicates intent, "this value shouldn't change."

```
const double PI = 3.14159;
void print(const char* message);
```

This helps the compiler optimize, prevents accidental modification, and improves clarity.

Step 4. Prefer Modern Standard Headers

Use standard headers like `<stdint.h>`, `<stdbool.h>`, and `<stddef.h>` for clear, portable code.

Example:

```
#include <stdint.h>
#include <stdbool.h>

bool is_even(uint32_t n) {
    return (n % 2) == 0;
}
```

Avoid using old-style typedefs like `typedef unsigned long ulong;` unless it improves meaning.

Step 5. Use `bool` Instead of `int` for Logic

In old C, people used `int` for true/false. Modern C gives you `_Bool` via `<stdbool.h>`:

```
#include <stdbool.h>

bool done = false;
if (!done) {
    done = true;
}
```

This improves clarity and makes your code self-documenting.

Step 6. Write Small, Focused Functions

Keep functions short, ideally one purpose per function.

Bad:

```
void handle_all() { /* does 10 things */ }
```

Good:

```c
void read_input(void);
void process_data(void);
void write_output(void);
```

This makes testing and debugging far easier.

Step 7. Avoid Macros for Everything

In early C, macros were overused for constants and functions. Today, prefer inline functions and `const` instead.

Bad:

```c
#define SQUARE(x) ((x) * (x))
```

Good:

```c
static inline int square(int x) { return x * x; }
```

Inline functions are type-safe and debug-friendly.

Step 8. Use Scoped Variables and Declarations

Since C99, you can declare variables close to where they're used:

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

Avoid keeping variables alive longer than necessary, this reduces bugs and clarifies scope.

Step 9. Embrace C23 Features

C23 modernizes syntax and makes C safer and more expressive.

Highlights:

- `typeof`, reuse variable types automatically
- `nullptr`, replaces `NULL`
- `[[nodiscard]]`, warn if function return is ignored

- `auto`, type inference for local variables
- UTF-8 character support and string literals
- `alignof` / `alignas` for precise memory layout

Example:

```
[[nodiscard]] int divide(int a, int b) {
    if (b == 0) return 0;
    return a / b;
}
```

Step 10. Tiny Code: Modern C23 Example

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

[[nodiscard]] static inline uint32_t add(uint32_t a, uint32_t b) {
    return a + b;
}

int main(void) {
    const uint32_t x = 10, y = 20;
    uint32_t sum = add(x, y);

    bool valid = (sum > 0);
    if (valid)
        printf("Sum = %u\n", sum);

    return 0;
}
```

Compile with a modern compiler (GCC 13+ or Clang 17+):

```
gcc -std=c23 modern.c -o modern
```

Output:

```
Sum = 30
```

This code uses `[[nodiscard]]`, `bool`, and `const`, small touches that improve both style and safety.

Why It Matters

Readable C code lasts for decades. The best systems code, in kernels, compilers, and libraries, looks simple because it follows clear patterns:

- **Small, pure functions.**
- **Explicit types.**
- **No surprises in memory handling.**

Modern C doesn't mean rewriting everything. It means writing *intentional C*, clear, correct, and expressive.

Try It Yourself

1. Refactor one of your old programs to use `<stdint.h>` and `<stdbool.h>`.
2. Replace macros with inline functions.
3. Add `const` wherever possible.
4. Try compiling with `-std=c23` and explore new warnings.
5. Make your functions pure and side-effect free where possible.

Next, you'll conclude this journey with **Practice: Portable Multithreaded Program (90)**, a hands-on project that combines everything from memory management to threading and portability.

90. Practice: Portable Multithreaded Program

It's time to bring together everything you've learned, memory management, threads, synchronization, and portability, into one cohesive program.

In this final section of **Chapter 9**, you'll build a **portable multithreaded counter** that runs correctly across architectures, compilers, and systems, demonstrating clean, safe, and modern C in practice.

Step 1. The Goal

We'll write a program that:

- Spawns multiple threads using `pthread` (POSIX standard).
- Uses atomic operations for safe concurrent updates.
- Prints consistent results regardless of CPU or endianness.
- Compiles cleanly across Linux, macOS, and Windows (via MinGW).
- Uses modern, readable C11–C23 style.

Step 2. Plan the Design

1. **Shared Counter:** `atomic_int` for thread-safe increments.
2. **Thread Function:** Each thread performs a loop of increments.
3. **Timing:** Measure elapsed time for performance insight.
4. **Portability:** Use `#ifdef` for cross-platform compatibility.
5. **Final Validation:** Ensure result equals total increments.

Step 3. Full Tiny Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdatomic.h>
#include <pthread.h>
#include <stdint.h>
#include <time.h>

#ifdef _WIN32
#include <windows.h>
#define SLEEP(ms) Sleep(ms)
#else
#include <unistd.h>
#define SLEEP(ms) usleep((ms) * 1000)
#endif

#define THREADS 4
#define ITERATIONS 250000

atomic_int counter = 0;

void* worker(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < ITERATIONS; i++) {
        atomic_fetch_add(&counter, 1);
        if (i % 100000 == 0 && id == 0)
            SLEEP(1);
    }
    return NULL;
}

double now(void) {
```

```
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec / 1e9;
}

int main(void) {
    pthread_t threads[THREADS];
    int ids[THREADS];
    double start = now();

    for (int i = 0; i < THREADS; i++) {
        ids[i] = i;
        if (pthread_create(&threads[i], NULL, worker, &ids[i]) != 0) {
            perror("pthread_create failed");
            return 1;
        }
    }

    for (int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);

    double end = now();
    printf("Counter = %d (expected %d)\n", counter, THREADS * ITERATIONS);
    printf("Elapsed time: %.3f seconds\n", end - start);
    return 0;
}
```

Step 4. How It Works

- **Atomic Counter:** `atomic_fetch_add` ensures that increments are atomic and race-free without using a mutex.
- **Thread Creation:** Each thread runs the `worker()` function independently.
- **Synchronization:** `pthread_join` ensures all threads finish before printing results.
- **Timing:** Uses `clock_gettime()` for precise cross-platform timing.
- **Sleep Macro:** `SLEEP(ms)` abstracts away platform differences between Windows and POSIX.

Step 5. Compile and Run

On Linux or macOS:

```
gcc -std=c23 -pthread -O2 -Wall -Wextra portable_threads.c -o portable_threads
./portable_threads
```

On Windows (MinGW):

```
gcc -std=c23 -O2 -Wall -Wextra portable_threads.c -o portable_threads.exe -lws2_32
portable_threads.exe
```

Expected output:

```
Counter = 1000000 (expected 1000000)
Elapsed time: 0.134 seconds
```

The program finishes with perfect accuracy, no race conditions, and works across platforms.

Step 6. Improving Portability

- **Replace pthreads with C11 `<threads.h>`** if you want standard-only C:

  ```
  #include <threads.h>
  ```

  Use `thrd_create` and `thrd_join` instead of `pthread_create` and `pthread_join`.

- **Use static assertions for validation:**

  ```
  _Static_assert(THREADS > 0, "Must have at least one thread");
  ```

- **Use conditional macros** for system differences (`_WIN32`, `__linux__`, `__APPLE__`).

Step 7. Safety and Clarity Checklist

No raw pointers shared unsafely Atomic operations prevent races Sleep and timing are cross-platform Clean, modern syntax with C23 support Easy to modify (e.g., change thread count or workload)

Step 8. Why It's Portable

- Uses only standard C and POSIX APIs.
- Avoids endian-dependent or undefined behavior.
- Has clear abstractions for platform-specific code.
- Relies on atomic types, not CPU-specific intrinsics.
- Runs on x86, ARM, RISC-V, and others without changes.

Step 9. Why It Matters

This tiny program embodies what C is best at:

- **Speed:** Threaded performance close to hardware.
- **Control:** Explicit memory and concurrency.
- **Clarity:** Modern C syntax keeps it readable.
- **Portability:** Runs everywhere a compiler exists.

This is the C of today, minimal, precise, and reliable.

Step 10. Try It Yourself

1. Change `THREADS` and observe performance scaling.
2. Replace the atomic counter with a mutex, compare speed.
3. Port it to Windows and verify output.
4. Add timing to measure each thread's duration.
5. Experiment with C11 `<threads.h>` API for pure standard C.

You've completed **Chapter 9, Portable and Modern C**. Next comes **Chapter 10: Building Real Projects**, where you'll apply these foundations to construct real-world systems, libraries, servers, and interpreters, all in clean, idiomatic C.

Chapter 10. Building Real Projects

91. Designing Small C Libraries

Writing libraries is how you make your C code reusable, modular, and easy to maintain. In this section, you'll learn how to design and structure a **small, portable, and well-documented C library**, the kind used in real systems for decades.

Step 1. What Is a Library in C?

A library in C is a **collection of functions and data types** that can be used by multiple programs.

There are two kinds of libraries:

- **Static libraries** (`.a` or `.lib`) – compiled into the final program at build time.
- **Shared libraries** (`.so` or `.dll`) – loaded dynamically at runtime.

You'll start by building a small **static library** that provides reusable math utilities.

Step 2. Plan the Library

Let's design a library called **simplemath**, which provides:

- `add`, `subtract`, `multiply`, `divide`
- Error handling for divide-by-zero
- Clean, consistent naming

Structure:

```
simplemath/
    include/
        simplemath.h
    src/
        simplemath.c
    Makefile
```

Step 3. The Header File (`simplemath.h`)

```
#ifndef SIMPLEMATH_H
#define SIMPLEMATH_H

#ifdef __cplusplus
extern "C" {
#endif

double sm_add(double a, double b);
double sm_sub(double a, double b);
double sm_mul(double a, double b);
double sm_div(double a, double b, int *error);

#ifdef __cplusplus
}
#endif

#endif
```

Notes:

- Include guards prevent double inclusion.
- `extern "C"` allows usage in C++ projects.
- Prefix (`sm_`) prevents naming conflicts.

Step 4. The Implementation File (`simplemath.c`)

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

double sm_add(double a, double b) { return a + b; }
double sm_sub(double a, double b) { return a - b; }
double sm_mul(double a, double b) { return a * b; }

double sm_div(double a, double b, int *error) {
    if (b == 0) {
        if (error) *error = 1;
        fprintf(stderr, "Division by zero\n");
        return 0.0;
```

```
    }
    if (error) *error = 0;
    return a / b;
}
```

Step 5. Tiny Code: Example Program Using the Library

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

int main(void) {
    int err;
    double x = sm_div(10, 2, &err);
    printf("10 / 2 = %.2f\n", x);

    x = sm_div(10, 0, &err);
    if (err) printf("Error detected during division.\n");
    return 0;
}
```

Step 6. Makefile to Build the Library

```
CC = gcc
CFLAGS = -std=c23 -O2 -Wall -Wextra -Iinclude

all: libsimplemath.a test

libsimplemath.a: src/simplemath.o
    ar rcs libsimplemath.a src/simplemath.o

src/simplemath.o: src/simplemath.c include/simplemath.h
    $(CC) $(CFLAGS) -c src/simplemath.c -o src/simplemath.o

test: test.c libsimplemath.a
    $(CC) $(CFLAGS) test.c -L. -lsimplemath -o test

clean:
    rm -f src/*.o *.a test
```

Build it:

```
make
```

Run:

```
./test
```

Output:

```
10 / 2 = 5.00
Division by zero
Error detected during division.
```

Step 7. Design Guidelines for Clean C Libraries

Principle	Description
Prefix all symbols	Avoid global name clashes (e.g., `sm_add`)
Single responsibility	Each function should do one clear thing
Minimal dependencies	Don't rely on non-standard headers
Use header guards	Prevent duplicate inclusion
Provide error handling	Return codes, `errno`, or out parameters
Write documentation	Use Doxygen or simple comment blocks
Version your API	Track breaking changes cleanly

Step 8. Adding Versioning and Metadata

Add this to your header:

```
#define SIMPLEMATH_VERSION "1.0.0"
```

In your CMake or Makefile build scripts, you can propagate this version into your packaging system or documentation.

Step 9. Why It Matters

Writing a library transforms you from a script author into a systems builder. It teaches **API design, separation of interface and implementation, and long-term maintenance**, the same principles used in real-world software like glibc, SQLite, and curl.

Step 10. Try It Yourself

1. Add new functions (`sm_pow`, `sm_mod`, `sm_avg`).
2. Create a shared version of the library (`libsimplemath.so`).
3. Document your API using Doxygen-style comments.
4. Write a header-only version (`static inline` functions).
5. Package your library with versioning and examples.

Next, you'll learn how to **build a full command-line tool in C (92)**, connecting your reusable libraries to practical, user-facing applications.

92. Building a Command-Line Tool

Command-line tools are where most C programmers begin building *real software*. They are fast, portable, and integrate naturally with Unix-like environments. In this section, you'll build a **small, self-contained CLI tool** that processes input arguments, reads files, and outputs results, the same pattern used by tools like `grep`, `cat`, and `wc`.

Step 1. The Goal

We'll build a simple command-line tool called **linestat** that:

- Counts lines, words, and characters in a text file (like a mini `wc`).
- Takes input from a file or standard input.
- Accepts flags like `-l`, `-w`, `-c`.
- Uses clean error handling and modular functions.

Step 2. Project Layout

```
linestat/
    linestat.c
    Makefile
    README.md
```

Step 3. Core Concepts

Command-line programs follow a few timeless patterns:

1. **Read arguments** with `argc` and `argv`.
2. **Validate inputs** before processing.
3. **Open files** safely with `fopen` or use `stdin`.

4. **Process data line-by-line**.
5. **Report results** clearly and consistently.

Step 4. Tiny Code: linestat.c

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

static void print_usage(const char *prog) {
    printf("Usage: %s [-l] [-w] [-c] [file]\n", prog);
    printf("Options:\n");
    printf("  -l   count lines\n");
    printf("  -w   count words\n");
    printf("  -c   count characters\n");
}

int main(int argc, char *argv[]) {
    int count_lines = 0, count_words = 0, count_chars = 0;
    const char *filename = NULL;

    // Parse arguments
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-l") == 0) count_lines = 1;
        else if (strcmp(argv[i], "-w") == 0) count_words = 1;
        else if (strcmp(argv[i], "-c") == 0) count_chars = 1;
        else if (argv[i][0] != '-') filename = argv[i];
        else {
            print_usage(argv[0]);
            return 1;
        }
    }

    FILE *fp = filename ? fopen(filename, "r") : stdin;
    if (!fp) {
        perror("Error opening file");
        return 1;
    }

    long lines = 0, words = 0, chars = 0;
    int in_word = 0;
```

```
    int ch;

    while ((ch = fgetc(fp)) != EOF) {
        chars++;
        if (ch == '\n') lines++;
        if (ch == ' ' || ch == '\n' || ch == '\t') in_word = 0;
        else if (!in_word) { words++; in_word = 1; }
    }

    fclose(fp);

    if (!count_lines && !count_words && !count_chars) {
        count_lines = count_words = count_chars = 1; // Default all
    }

    if (count_lines) printf("Lines: %ld\n", lines);
    if (count_words) printf("Words: %ld\n", words);
    if (count_chars) printf("Chars: %ld\n", chars);

    return 0;
}
```

Step 5. Build and Run

Makefile

```
CC = gcc
CFLAGS = -std=c23 -O2 -Wall -Wextra

linestat: linestat.c
    $(CC) $(CFLAGS) linestat.c -o linestat

clean:
    rm -f linestat
```

Build it:

```
make
```

Run it:

```
./linestat -l -w -c example.txt
```

Or from a pipeline:

```
cat example.txt | ./linestat -w
```

Example output:

```
Lines: 12
Words: 85
Chars: 430
```

Step 6. Breaking Down the Code

- **Argument Parsing:** Loops through `argv` to detect flags.
- **Input Handling:** Reads from `stdin` when no file is given.
- **Counting Logic:** Tracks transitions between spaces and characters to count words.
- **Graceful Exit:** Uses `fclose` and `perror` for error reporting.
- **Default Behavior:** When no flags are passed, all counts are printed.

Step 7. Making It More Robust

You can extend this program easily:

- Add `-q` for quiet mode (only print totals).
- Add `--help` for extended usage info.
- Use `getline()` for reading full lines (C POSIX).
- Print counts side by side in a single line:

  ```
  12  85  430  example.txt
  ```

Step 8. Cross-Platform Considerations

- Use `#ifdef _WIN32` to handle file paths and newline differences.
- Always open files in text mode: `fopen(filename, "r")`.
- Use `size_t` instead of `long` for portability.

Step 9. Why It Matters

Writing a CLI teaches key systems skills:

- Argument parsing and I/O
- File handling and error checking
- Performance thinking (streaming, buffering)
- Modular design for future features

Every developer who writes in C eventually writes a CLI, it's how tools like Git, Curl, and GCC were born.

Step 10. Try It Yourself

1. Add a `-v` flag that shows program version.
2. Support reading multiple files.
3. Add timing (use `clock()` to measure runtime).
4. Print totals across all files.
5. Integrate your **simplemath** library to compute average words per line.

Next, you'll move to **93. Tiny HTTP Server (Sockets and Threads)**, where your command-line skills evolve into network programming: accepting connections, handling requests, and serving content in pure C.

93. Tiny HTTP Server (Sockets and Threads)

Now that you know how to build command-line tools, it's time to make your program talk to the network. In this section, you'll build a **tiny multithreaded HTTP server**, a small, minimal clone of what powers the web.

You'll learn sockets, threading, request parsing, and response generation, all from first principles.

Step 1. The Goal

We'll create a simple HTTP server that:

- Listens on port `8080`
- Accepts multiple connections (one per thread)
- Parses a minimal HTTP request
- Responds with a static HTML page

This project combines file I/O, networking, and concurrency, three of C's most powerful capabilities.

Step 2. Project Layout

```
tinyhttp/
    server.c
    Makefile
    index.html
```

Step 3. The Core Idea

The server will:

1. **Create a socket** and bind it to port 8080.
2. **Listen** for connections.
3. **Accept** a client.
4. **Handle** the request in a new thread.
5. **Send** an HTTP response.
6. **Close** the socket and repeat.

Step 4. Tiny Code: server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define PORT 8080
#define BUF_SIZE 4096

void *handle_client(void *arg) {
    int client_fd = *(int *)arg;
    free(arg);

    char buffer[BUF_SIZE];
    int bytes = read(client_fd, buffer, sizeof(buffer) - 1);
    if (bytes <= 0) {
```

```
        close(client_fd);
        return NULL;
    }
    buffer[bytes] = '\0';

    // Basic HTTP response
    const char *body = "<html><body><h1>Hello from TinyHTTP!</h1></body></html>";
    char response[BUF_SIZE];
    snprintf(response, sizeof(response),
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html\r\n"
        "Content-Length: %zu\r\n"
        "Connection: close\r\n\r\n"
        "%s", strlen(body), body);

    write(client_fd, response, strlen(response));
    close(client_fd);
    return NULL;
}

int main(void) {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 10) < 0) {
```

```
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("TinyHTTP running on http://localhost:%d\n", PORT);

    while (1) {
        int client_fd;
        struct sockaddr_in client;
        socklen_t len = sizeof(client);
        client_fd = accept(server_fd, (struct sockaddr *)&client, &len);
        if (client_fd < 0) {
            perror("accept failed");
            continue;
        }

        int *pclient = malloc(sizeof(int));
        *pclient = client_fd;

        pthread_t tid;
        pthread_create(&tid, NULL, handle_client, pclient);
        pthread_detach(tid);
    }

    close(server_fd);
    return 0;
}
```

Step 5. Build and Run

Makefile

```
CC = gcc
CFLAGS = -std=c23 -pthread -O2 -Wall -Wextra

all: server

server: server.c
    $(CC) $(CFLAGS) server.c -o server
```

```
clean:
    rm -f server
```

Build and run:

```
make
./server
```

Open your browser and visit:

`http://localhost:8080`

You should see:

`Hello from TinyHTTP!`

Step 6. How It Works

1. **Socket setup:** The server creates a TCP socket (`socket()`), binds it to port 8080, and listens.
2. **Accept loop:** The main thread waits for connections.
3. **Threading:** Each connection is handled by a new thread (`pthread_create`), allowing multiple clients at once.
4. **HTTP parsing:** Minimal, just reads the request header and ignores the rest for now.
5. **Response:** A static HTML body is written to the socket.
6. **Cleanup:** Each thread closes its client socket after responding.

Step 7. Extend It

To make it more realistic, add:

- Serve static files:
  ```
  FILE *f = fopen("index.html", "r");
  ```
- Parse the first line of the request to get the path.
- Return 404 if the file doesn't exist.
- Add MIME types for `.html`, `.css`, `.js`, `.png`.
- Add logging with timestamps.

Step 8. Cross-Platform Notes

- Use `#ifdef _WIN32` to include `<winsock2.h>` and initialize with `WSAStartup()`.
- Replace `close()` with `closesocket()` on Windows.
- Use threads from `<threads.h>` for C11-only builds.

Step 9. Why It Matters

Building an HTTP server from scratch teaches you how the web really works:

- **Sockets:** the foundation of all network software.
- **Concurrency:** how to handle many users at once.
- **Protocols:** understanding request/response formats.
- **Systems thinking:** combining multiple low-level C features cleanly.

You're no longer just writing programs, you're shaping communication between machines.

Step 10. Try It Yourself

1. Add logging for each client connection.
2. Serve static files (`index.html`, `style.css`).
3. Implement a `/time` endpoint returning the system time.
4. Benchmark with `curl` or `ab`.
5. Extend to HTTP/1.1 persistent connections.

Next, you'll build **94. A Simple Key-Value Store**, where you'll learn file-based persistence, indexing, and serialization, the first step toward writing databases in pure C.

94. Simple Key-Value Store

Databases look scary until you build one yourself. In this section you will write a tiny **append only key value store** that persists data to disk, loads an in memory index on startup, and supports `get` and `set` from a simple CLI.

You will learn files, serialization, indexing, and crash safety basics.

Step 1. Design the file format

Keep it simple and binary. Each record is append only:

```
[ u32 key_len ][ u32 val_len ][ key bytes ][ value bytes ]
```

- All integers are stored as big endian so the file is portable.
- No in place updates. Setting the same key again appends a new record.

Step 2. Endianness helpers

We will use `htonl` and `ntohl` to encode and decode 32 bit lengths.

```c
#include <arpa/inet.h>   // Windows: winsock2.h

static inline uint32_t be32(uint32_t x)  { return htonl(x);  }
static inline uint32_t from_be32(uint32_t x) { return ntohl(x); }
```

Step 3. The in memory index

On startup, scan the log file once and build a hash map of key -> file offset of the newest record. We will implement a simple open addressing hash table for clarity.

Index entry:

```c
typedef struct {
    uint64_t offset;   // file position of record start
    uint32_t key_hash; // cached hash for quick probing
    uint32_t key_len;  // used to confirm match
} kv_slot;
```

Step 4. Hashing

Use a compact 32 bit FNV-1a hash for strings.

```c
static uint32_t fnv1a(const unsigned char *s, size_t n) {
    uint32_t h = 2166136261u;
    for (size_t i = 0; i < n; i++) {
        h ^= s[i];
        h *= 16777619u;
```

```
    }
    return h;
}
```

Step 5. Tiny Code: core implementation

A single file version to keep things approachable.

```
// file: kv.c
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>   // Windows: include <winsock2.h> and link Ws2_32
#include <errno.h>

typedef struct {
    FILE *f;
    char *path;
    // simple hash table index
    struct slot { uint64_t off; uint32_t h, klen; } *tab;
    size_t cap, used;
} kv_db;

static uint32_t fnv1a(const unsigned char *s, size_t n) {
    uint32_t h = 2166136261u;
    for (size_t i = 0; i < n; i++) { h ^= s[i]; h *= 16777619u; }
    return h;
}

static size_t next_pow2(size_t n) { size_t p = 1; while (p < n) p <<= 1; return p; }

static int kv_index_put(kv_db *db, const unsigned char *key, uint32_t klen, uint64_t off) {
    if (db->used * 2 >= db->cap) { // grow
        size_t ncap = db->cap ? db->cap * 2 : 1024;
        struct slot *old = db->tab;
        size_t oldcap = db->cap;
        db->tab = calloc(ncap, sizeof(*db->tab));
        if (!db->tab) return -1;
        db->cap = ncap; db->used = 0;
```

```
        for (size_t i = 0; i < oldcap; i++) if (old[i].off) {
            // reinsert based on stored key hash and key length
            size_t m = ncap - 1, j = old[i].h & m;
            while (db->tab[j].off) j = (j + 1) & m;
            db->tab[j] = old[i];
            db->used++;
        }
        free(old);
    }
    uint32_t h = fnv1a(key, klen);
    size_t m = db->cap - 1, i = h & m;
    while (db->tab[i].off) {
        if (db->tab[i].h == h && db->tab[i].klen == klen) { db->tab[i].off = off; return 0; }
        i = (i + 1) & m;
    }
    db->tab[i].off = off; db->tab[i].h = h; db->tab[i].klen = klen; db->used++;
    return 0;
}

static long kv_index_find_slot(kv_db *db, const unsigned char *key, uint32_t klen) {
    if (db->cap == 0) return -1;
    uint32_t h = fnv1a(key, klen);
    size_t m = db->cap - 1, i = h & m, steps = 0;
    while (db->tab[i].off && steps <= db->cap) {
        if (db->tab[i].h == h && db->tab[i].klen == klen) return (long)i;
        i = (i + 1) & m; steps++;
    }
    return -1;
}

static int kv_open(kv_db *db, const char *path) {
    memset(db, 0, sizeof(*db));
    db->path = strdup(path);
    db->f = fopen(path, "ab+");
    if (!db->f) return -1;
    fflush(db->f);
    // build index by scanning from start
    FILE *r = fopen(path, "rb");
    if (!r) return -1;
    // start with some capacity
    db->cap = 1024; db->tab = calloc(db->cap, sizeof(*db->tab));
    if (!db->tab) return -1;
```

```
    uint64_t off = 0;
    for (;;) {
        uint32_t klen_be, vlen_be;
        if (fread(&klen_be, 4, 1, r) != 1) break;
        if (fread(&vlen_be, 4, 1, r) != 1) break;
        uint32_t klen = ntohl(klen_be), vlen = ntohl(vlen_be);
        unsigned char *k = malloc(klen);
        if (!k) break;
        if (fread(k, 1, klen, r) != klen) { free(k); break; }
        if (fseek(r, vlen, SEEK_CUR) != 0) { free(k); break; }
        kv_index_put(db, k, klen, off);
        free(k);
        off += 8u + klen + vlen;
    }
    fclose(r);
    return 0;
}

static int kv_set(kv_db *db, const unsigned char *key, uint32_t klen,
                  const unsigned char *val, uint32_t vlen) {
    uint32_t klen_be = htonl(klen), vlen_be = htonl(vlen);
    if (fwrite(&klen_be, 4, 1, db->f) != 1) return -1;
    if (fwrite(&vlen_be, 4, 1, db->f) != 1) return -1;
    if (fwrite(key, 1, klen, db->f) != klen) return -1;
    if (fwrite(val, 1, vlen, db->f) != vlen) return -1;
    fflush(db->f); // durability: fsync would be stronger
    // compute offset of the record we just wrote
    long end = ftell(db->f);
    if (end < 0) return -1;
    uint64_t off = (uint64_t)end - (8u + klen + vlen);
    return kv_index_put(db, key, klen, off);
}

static int kv_get(kv_db *db, const unsigned char *key, uint32_t klen,
                  unsigned char **out, uint32_t *outlen) {
    long s = kv_index_find_slot(db, key, klen);
    if (s < 0) return -1;
    uint64_t off = db->tab[s].off;
    if (fseek(db->f, (long)off, SEEK_SET) != 0) return -1;
    uint32_t klen_be, vlen_be;
    if (fread(&klen_be, 4, 1, db->f) != 1) return -1;
    if (fread(&vlen_be, 4, 1, db->f) != 1) return -1;
```

```
    uint32_t kL = ntohl(klen_be), vL = ntohl(vlen_be);
    unsigned char *kbuf = malloc(kL);
    if (!kbuf) return -1;
    if (fread(kbuf, 1, kL, db->f) != kL) { free(kbuf); return -1; }
    // confirm key match to be safe
    if (kL != klen || memcmp(kbuf, key, klen) != 0) { free(kbuf); return -1; }
    free(kbuf);
    unsigned char *v = malloc(vL + 1);
    if (!v) return -1;
    if (fread(v, 1, vL, db->f) != vL) { free(v); return -1; }
    v[vL] = 0; // NUL terminate for convenience
    *out = v; *outlen = vL;
    return 0;
}

static void kv_close(kv_db *db) {
    if (!db) return;
    if (db->f) fclose(db->f);
    free(db->tab);
    free(db->path);
}

static void usage(const char *p) {
    fprintf(stderr, "Usage: %s <file> get <key>\n", p);
    fprintf(stderr, "       %s <file> set <key> <value>\n", p);
}

int main(int argc, char **argv) {
    if (argc < 4) { usage(argv[0]); return 1; }
    kv_db db;
    if (kv_open(&db, argv[1]) != 0) { perror("open"); return 1; }

    const char *cmd = argv[2];
    if (strcmp(cmd, "set") == 0) {
        if (argc < 5) { usage(argv[0]); kv_close(&db); return 1; }
        const unsigned char *k = (const unsigned char *)argv[3];
        const unsigned char *v = (const unsigned char *)argv[4];
        if (kv_set(&db, k, (uint32_t)strlen((char*)k), v, (uint32_t)strlen((char*)v)) != 0)
            perror("set");
    } else if (strcmp(cmd, "get") == 0) {
        const unsigned char *k = (const unsigned char *)argv[3];
        unsigned char *out = NULL; uint32_t n = 0;
```

```
        if (kv_get(&db, k, (uint32_t)strlen((char*)k), &out, &n) == 0) {
            fwrite(out, 1, n, stdout);
            fputc('\n', stdout);
            free(out);
        } else {
            fprintf(stderr, "not found\n");
        }
    } else {
        usage(argv[0]);
    }

    kv_close(&db);
    return 0;
}
```

Build:

```
gcc -std=c23 -O2 -Wall -Wextra kv.c -o kv
```

Run:

```
./kv store.log set color blue
./kv store.log get color
# prints: blue
```

Step 6. Compaction

Because we append forever, the log grows. Implement a simple **compact** command that rewrites only the latest version of each key to a new file, then swaps files.

Idea:

1. Iterate index
2. Read the newest record for each key
3. Append it to `store.log.new`
4. Replace the old file

This keeps disk usage under control and speeds up startup scanning.

Step 7. Crash safety basics

- Always `fflush` after appending a record.
- For stronger durability call `fsync(fileno(db->f))` on POSIX after `fflush`.
- Write whole records or none. Length headers first, then key, then value.
- Consider a checksum per record to detect torn writes.

Step 8. CLI improvements

Add subcommands:

```
kv <file> set <k> <v>
kv <file> get <k>
kv <file> compact
kv <file> stats
```

`stats` can print number of keys, file size, load factor, and index capacity.

Step 9. Testing

- Insert 10k keys, then get a random 100 keys and verify values.
- Overwrite the same key many times and ensure `get` returns the latest one.
- Kill the program during writes and ensure the log is still readable.
- Run with AddressSanitizer to catch memory bugs:

```
clang -std=c23 -O1 -g -fsanitize=address,undefined kv.c -o kv_asan
```

Step 10. Why it matters

This tiny store teaches the core database loop:

- **Log structured storage** for durability
- **In memory index** for speed
- **Compaction** for space and locality
- **Portable encoding** for cross platform reads

You just built the foundation that many production systems use at larger scale.

Try it yourself

1. Add a delete tombstone record type and have `get` respect it.
2. Store expiration timestamps and implement a `purge` command.
3. Use memory mapped I O for reads to speed up lookups.
4. Replace the linear probing table with a chained hash or hopscotch hashing.
5. Add a simple checksum per record and verify on read.

Next you will implement **95. Implementing a Custom Allocator** where you will learn how `malloc` like systems manage the heap, and write a tiny arena allocator you can drop into small C projects.

95. Implementing a Custom Allocator

Every C program eventually asks the operating system for memory, but `malloc` and `free` are not magic—they are layers above system calls like `brk` and `mmap`. In this section, you will build your own **custom memory allocator**—a simple arena allocator that grabs a large block of memory once and doles it out efficiently.

You'll see how real allocators work inside kernels, games, and embedded systems.

Step 1. The Goal

We'll implement a minimal **arena allocator** that:

- Allocates from a preallocated block
- Never frees individual objects
- Resets all memory at once when the arena is cleared

This model is perfect for short-lived data structures, parsing, and high-performance applications.

Step 2. Design

An arena allocator tracks:

- The **base pointer** (start of memory)
- The **current pointer** (next free position)
- The **capacity** (total size of the arena)

When you allocate, it simply bumps the pointer forward.

Structure:

```
typedef struct {
    unsigned char *base;
    size_t capacity;
    size_t offset;
} Arena;
```

Step 3. Tiny Code: Minimal Arena

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

typedef struct {
    unsigned char *base;
    size_t capacity;
    size_t offset;
} Arena;

Arena *arena_create(size_t capacity) {
    Arena *a = malloc(sizeof(Arena));
    if (!a) return NULL;
    a->base = malloc(capacity);
    if (!a->base) { free(a); return NULL; }
    a->capacity = capacity;
    a->offset = 0;
    return a;
}

void *arena_alloc(Arena *a, size_t size) {
    if (a->offset + size > a->capacity) return NULL;
    void *ptr = a->base + a->offset;
    a->offset += size;
    return ptr;
}

void arena_reset(Arena *a) {
```

```
    a->offset = 0;
}

void arena_free(Arena *a) {
    free(a->base);
    free(a);
}
```

Step 4. Example Use

```
int main(void) {
    Arena *arena = arena_create(1024);
    if (!arena) {
        fprintf(stderr, "Failed to create arena\n");
        return 1;
    }

    int *arr = arena_alloc(arena, 10 * sizeof(int));
    if (!arr) {
        fprintf(stderr, "Allocation failed\n");
        arena_free(arena);
        return 1;
    }

    for (int i = 0; i < 10; i++) arr[i] = i * i;

    printf("Squares: ");
    for (int i = 0; i < 10; i++) printf("%d ", arr[i]);
    printf("\n");

    arena_reset(arena);  // all memory reused
    arena_free(arena);
    return 0;
}
```

Step 5. How It Works

1. `arena_create` grabs one large block from `malloc`.
2. `arena_alloc` hands out memory by increasing an offset—no per-object metadata.
3. `arena_reset` rewinds the arena to reuse the memory instantly.

4. `arena_free` releases the entire block in one call.

This is *O(1)* for every allocation, with zero fragmentation.

Step 6. Adding Alignment

Sometimes allocations must be aligned (for example, 16-byte alignment for SIMD). We can round up the offset to the nearest alignment boundary.

```
static size_t align_up(size_t n, size_t align) {
    return (n + (align - 1)) & ~(align - 1);
}

void *arena_alloc_aligned(Arena *a, size_t size, size_t align) {
    size_t new_offset = align_up(a->offset, align);
    if (new_offset + size > a->capacity) return NULL;
    void *ptr = a->base + new_offset;
    a->offset = new_offset + size;
    return ptr;
}
```

Step 7. Debugging Helpers

Add diagnostic printing to understand usage:

```
void arena_stats(Arena *a) {
    printf("Arena used: %zu / %zu bytes (%.1f%%)\n",
           a->offset, a->capacity,
           (a->offset * 100.0) / a->capacity);
}
```

Step 8. Advanced Idea: Nested Arenas

You can make **sub-arenas** for scoped memory:

```
typedef struct {
    Arena *parent;
    size_t start;
} ArenaScope;
```

```c
ArenaScope arena_push(Arena *a) {
    return (ArenaScope){ .parent = a, .start = a->offset };
}

void arena_pop(ArenaScope s) {
    s.parent->offset = s.start;
}
```

This lets you "temporarily allocate" for a function or block and reset automatically.

Step 9. Why It Matters

Allocators define how performance feels in large systems. By writing one, you understand:

- How `malloc` and `free` manage metadata
- How fragmentation occurs
- How specialized allocators (arenas, pools, slabs) achieve speed and predictability

Games, web servers, and compilers all use custom allocators to control lifetime and avoid overhead.

Step 10. Try It Yourself

1. Add bounds checking that prints errors when overrun.
2. Implement a **pool allocator** for fixed-size objects (e.g., `struct Node`).
3. Use `mmap` to request anonymous memory directly from the OS.
4. Add a leak detector that reports unfreed bytes at shutdown.
5. Combine multiple arenas into a hierarchical allocator.

Next you'll build **96. Writing a Text Parser**, using your allocator to manage short-lived strings and tokens as you build a mini lexer and parser in pure C.

96. Writing a Text Parser

Time to turn raw text into structure. In this section you will write a **tiny expression parser** that converts strings like `3 + 4*2 - (1 + 5)` into an **AST** (abstract syntax tree). We will build a simple tokenizer, a recursive descent parser with precedence, and a pretty printer to check the result. In the next section you can add an evaluator to run it.

Step 1. Goal and scope

We will parse integer arithmetic with these features:

- Integers: `0`, `42`, `1234`
- Operators: `+`, `-`, `*`, `/`
- Parentheses: `( ... )`
- Whitespace ignored

Output: an AST you can traverse or evaluate later.

Step 2. Grammar (informal)

We will use the classic precedence rules:

```
expr   -> term (('+' | '-') term)*
term   -> factor (('*' | '/') factor)*
factor -> INT | '(' expr ')'
```

Parsing follows these functions in order: `parse_expr`, `parse_term`, `parse_factor`.

Step 3. Tokens

We first scan characters into tokens:

- `TOK_INT` with a numeric value
- `TOK_PLUS`, `TOK_MINUS`, `TOK_STAR`, `TOK_SLASH`
- `TOK_LPAREN`, `TOK_RPAREN`
- `TOK_EOF` to mark the end

Step 4. AST nodes

Use a compact node type:

```c
typedef enum { N_INT, N_ADD, N_SUB, N_MUL, N_DIV } NodeKind;

typedef struct Node {
    NodeKind kind;
    struct Node *left, *right;  // for binary ops
    long value;                 // for integers
} Node;
```

Binary nodes use `left` and `right`. Integer nodes use `value`.

Step 5. A tiny arena for nodes

Allocating nodes frequently with `malloc` is noisy. Use a tiny arena so each node is just a bump allocation and everything frees at once when you are done.

Step 6. Error handling strategy

Keep it simple:

- If an unexpected token appears, print a message with the position.
- Stop parsing and return `NULL`.
- The pretty printer will only run if the tree is not `NULL`.

Step 7. Pretty printing the AST

To verify parsing, print with minimal parentheses:

- For binary nodes, print `(left op right)`.
- For integers, print the number. This is enough to confirm shape and precedence.

Step 8. Tiny Code: self contained lexer + parser + printer

```c
// file: expr_parser.c
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

/* ---------- tiny arena ---------- */
typedef struct {
    unsigned char *base;
    size_t cap, off;
} Arena;

static Arena *arena_new(size_t cap) {
    Arena *a = malloc(sizeof(*a));
```

```
    if (!a) return NULL;
    a->base = malloc(cap);
    if (!a->base) { free(a); return NULL; }
    a->cap = cap; a->off = 0;
    return a;
}
static void *arena_alloc(Arena *a, size_t n, size_t align) {
    size_t p = (a->off + (align - 1)) & ~(align - 1);
    if (p + n > a->cap) return NULL;
    void *ptr = a->base + p; a->off = p + n; return ptr;
}
static void arena_free(Arena *a) { if (!a) return; free(a->base); free(a); }

/* ---------- tokens ---------- */
typedef enum {
    TOK_INT, TOK_PLUS, TOK_MINUS, TOK_STAR, TOK_SLASH,
    TOK_LPAREN, TOK_RPAREN, TOK_EOF, TOK_ERR
} TokKind;

typedef struct {
    TokKind kind;
    long    ival;
    const char *start;  // for error messages
    const char *end;
} Token;

typedef struct {
    const char *src;
    const char *cur;
    Token       look;   // one-token lookahead
} Lexer;

static void skip_ws(Lexer *L) {
    while (isspace((unsigned char)*L->cur)) L->cur++;
}

static Token make(Lexer *L, TokKind k, const char *s, const char *e, long v) {
    Token t = {k, v, s, e};
    return t;
}

static Token next_token_raw(Lexer *L) {
```

```
    skip_ws(L);
    const char *s = L->cur;
    if (*L->cur == 0) return make(L, TOK_EOF, s, s, 0);

    char c = *L->cur++;
    switch (c) {
        case '+': return make(L, TOK_PLUS,   s, L->cur, 0);
        case '-': return make(L, TOK_MINUS,  s, L->cur, 0);
        case '*': return make(L, TOK_STAR,   s, L->cur, 0);
        case '/': return make(L, TOK_SLASH,  s, L->cur, 0);
        case '(': return make(L, TOK_LPAREN, s, L->cur, 0);
        case ')': return make(L, TOK_RPAREN, s, L->cur, 0);
        default:
            if (isdigit((unsigned char)c)) {
                long v = c - '0';
                const char *p = L->cur;
                while (isdigit((unsigned char)*p)) {
                    v = v * 10 + (*p - '0');
                    p++;
                }
                Token t = make(L, TOK_INT, s, p, v);
                L->cur = p;
                return t;
            }
            return make(L, TOK_ERR, s, L->cur, 0);
    }
}

static void lexer_init(Lexer *L, const char *src) {
    L->src = src; L->cur = src; L->look = next_token_raw(L);
}
static Token peek(Lexer *L) { return L->look; }
static Token take(Lexer *L) { Token t = L->look; L->look = next_token_raw(L); return t; }

/* ---------- AST ---------- */
typedef enum { N_INT, N_ADD, N_SUB, N_MUL, N_DIV } NodeKind;

typedef struct Node {
    NodeKind kind;
    struct Node *l, *r;
    long value;
} Node;
```

```c
static Node *node_new_int(Arena *A, long v) {
    Node *n = arena_alloc(A, sizeof(*n), _Alignof(Node));
    if (!n) return NULL;
    n->kind = N_INT; n->l = n->r = NULL; n->value = v; return n;
}
static Node *node_new_bin(Arena *A, NodeKind k, Node *l, Node *r) {
    Node *n = arena_alloc(A, sizeof(*n), _Alignof(Node));
    if (!n) return NULL;
    n->kind = k; n->l = l; n->r = r; n->value = 0; return n;
}

/* ---------- parser: expr -> term -> factor ---------- */
typedef struct { Lexer *L; Arena *A; int ok; } Parser;

static void fail(Parser *P, const char *msg, Token t) {
    P->ok = 0;
    size_t pos = (size_t)(t.start - P->L->src);
    fprintf(stderr, "Parse error at pos %zu: %s\n", pos, msg);
}

static Node *parse_expr(Parser *P); // forward

static Node *parse_factor(Parser *P) {
    Token t = peek(P->L);
    if (t.kind == TOK_INT) { take(P->L); return node_new_int(P->A, t.ival); }
    if (t.kind == TOK_LPAREN) {
        take(P->L);
        Node *inside = parse_expr(P);
        if (!P->ok) return NULL;
        if (peek(P->L).kind != TOK_RPAREN) { fail(P, "expected ')'", peek(P->L)); return NULL
        take(P->L);
        return inside;
    }
    fail(P, "expected number or '('", t);
    return NULL;
}

static Node *parse_term(Parser *P) {
    Node *n = parse_factor(P);
    while (P->ok) {
        TokKind k = peek(P->L).kind;
        if (k != TOK_STAR && k != TOK_SLASH) break;
```

```
        take(P->L);
        Node *r = parse_factor(P);
        if (!r) return NULL;
        n = node_new_bin(P->A, k == TOK_STAR ? N_MUL : N_DIV, n, r);
        if (!n) { P->ok = 0; return NULL; }
    }
    return n;
}

static Node *parse_expr(Parser *P) {
    Node *n = parse_term(P);
    while (P->ok) {
        TokKind k = peek(P->L).kind;
        if (k != TOK_PLUS && k != TOK_MINUS) break;
        take(P->L);
        Node *r = parse_term(P);
        if (!r) return NULL;
        n = node_new_bin(P->A, k == TOK_PLUS ? N_ADD : N_SUB, n, r);
        if (!n) { P->ok = 0; return NULL; }
    }
    return n;
}

/* ---------- printer ---------- */
static void print_ast(Node *n) {
    if (!n) return;
    switch (n->kind) {
        case N_INT: printf("%ld", n->value); break;
        case N_ADD: printf("("); print_ast(n->l); printf(" + "); print_ast(n->r); printf(")"
        case N_SUB: printf("("); print_ast(n->l); printf(" - "); print_ast(n->r); printf(")"
        case N_MUL: printf("("); print_ast(n->l); printf(" * "); print_ast(n->r); printf(")"
        case N_DIV: printf("("); print_ast(n->l); printf(" / "); print_ast(n->r); printf(")"
    }
}

/* ---------- main for quick testing ---------- */
int main(int argc, char **argv) {
    const char *src = (argc > 1) ? argv[1] : "3 + 4*2 - (1 + 5)";
    Lexer L; lexer_init(&L, src);

    Arena *A = arena_new(1 << 16);
    if (!A) { fprintf(stderr, "arena alloc failed\n"); return 1; }
```

```
    Parser P = { .L = &L, .A = A, .ok = 1 };
    Node *root = parse_expr(&P);

    if (P.ok && peek(&L).kind == TOK_EOF && root) {
        print_ast(root);
        printf("\n");
    } else {
        fprintf(stderr, "Failed to parse input.\n");
    }

    arena_free(A);
    return P.ok ? 0 : 1;
}
```

Build and run:

```
gcc -std=c23 -O2 -Wall -Wextra expr_parser.c -o expr_parser
./expr_parser "3 + 4*2 - (1 + 5)"
```

Example output:

```
((3 + (4 * 2)) - (1 + 5))
```

This shows correct precedence and grouping.

Step 9. How to extend it

- Add unary `+` and unary `-` in `parse_factor`.
- Support integers in hex and underscores like `1_000`.
- Add power operator `^` with higher precedence.
- Record source spans on each node for better error messages.
- Swap the pretty printer for a JSON dump of the AST.

Step 10. Why this matters

Parsing transforms bytes into meaning. With a tokenizer, a clean grammar, and a small AST, you can build:

- Expression evaluators

- Config file readers
- Query languages
- Full interpreters

In the next section you will use this AST to build a **tiny interpreter** that evaluates expressions at runtime.

97. Tiny Interpreter for an Expression Language

You already built a tokenizer, parser, and AST. Now you will **evaluate** that AST so `3 + 4*2 - (1 + 5)` produces `5` at runtime. We will add a tiny environment for variables, a few built in functions, and a simple REPL.

Step 1. Evaluation model

We will walk the AST recursively:

- `N_INT` returns its value
- `N_ADD` returns `eval(left) + eval(right)`
- `N_SUB`, `N_MUL`, `N_DIV` similar, with divide by zero checks

Keep everything in `long` for now. You can switch to `double` later if you want decimals.

Step 2. Variables and assignments

Extend the grammar slightly:

```
stmt   -> IDENT '=' expr | expr
expr   -> term (('+' | '-') term)*
term   -> factor (('*' | '/') factor)*
factor -> INT | IDENT | '(' expr ')'
```

- If input contains `x = 10`, store `x -> 10` in the environment
- If input contains `x + 2`, look up `x` then evaluate

Step 3. Environment

Use a tiny linear table for clarity:

```c
typedef struct { const char *name; long value; } Binding;

typedef struct {
    Binding *items;
    size_t   count, cap;
} Env;

static long *env_get(Env *E, const char *name) {
    for (size_t i = 0; i < E->count; i++)
        if (strcmp(E->items[i].name, name) == 0) return &E->items[i].value;
    return NULL;
}

static int env_set(Env *E, const char *name, long v) {
    long *p = env_get(E, name);
    if (p) { *p = v; return 1; }
    if (E->count == E->cap) {
        size_t ncap = E->cap ? E->cap * 2 : 16;
        Binding *n = realloc(E->items, ncap * sizeof(*n));
        if (!n) return 0;
        E->items = n; E->cap = ncap;
    }
    E->items[E->count++] = (Binding){ strdup(name), v };
    return 1;
}
```

This is not the fastest map, but it is simple and good for a tiny interpreter.

Step 4. Extend tokens for identifiers and equals

Add two kinds:

- `TOK_IDENT` for variable names
- `TOK_EQ` for `=`

Identifier rule: start with letter or `_`, continue with letter, digit, `_`.

Step 5. AST nodes for identifiers and assignment

```c
typedef enum { N_INT, N_ADD, N_SUB, N_MUL, N_DIV, N_IDENT, N_ASSIGN } NodeKind;

typedef struct Node {
    NodeKind kind;
    struct Node *l, *r;  // for binary ops and assignment
    long value;          // for integers
    const char *name;    // for identifiers
} Node;
```

- `N_IDENT` uses `name`
- `N_ASSIGN` uses `l` as name node and `r` as expression node

Step 6. Parser changes

- In `factor`, if token is `IDENT`, return `N_IDENT`
- Add `parse_stmt`:
  - If lookahead is `IDENT` then `=` then parse `expr` and build `N_ASSIGN`
  - Else parse `expr`

Step 7. Tiny Code: evaluator + minimal REPL

Below is a compact interpreter that builds on the earlier parser. For brevity, the lexer and arena are trimmed to only the new bits you need here.

```c
// file: tiny_interp.c
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

/* ----- tiny arena ----- */
typedef struct { unsigned char *base; size_t cap, off; } Arena;
static Arena *arena_new(size_t cap){ Arena*a=malloc(sizeof(*a)); if(!a)return NULL; a->base=
static void *arena_alloc(Arena*a,size_t n,size_t al){ size_t p=(a->off+(al-1))&~(al-1); if(p
static void arena_free(Arena*a){ if(!a)return; free(a->base); free(a); }

/* ----- tokens ----- */
typedef enum { TOK_INT, TOK_PLUS, TOK_MINUS, TOK_STAR, TOK_SLASH,
```

```
               TOK_LPAREN, TOK_RPAREN, TOK_IDENT, TOK_EQ, TOK_EOF, TOK_ERR } TokKind;

typedef struct { TokKind kind; long ival; const char *s,*e; char *lexeme; } Token;

typedef struct { const char *src,*cur; Token look; } Lexer;

static void skip_ws(Lexer*L){ while(isspace((unsigned char)*L->cur)) L->cur++; }
static Token make_tok(TokKind k,const char*s,const char*e,long v,char*lex){ return (Token){k

static Token next_raw(Lexer*L){
    skip_ws(L); const char*s=L->cur; if(*L->cur==0) return make_tok(TOK_EOF,s,s,0,NULL);
    char c=*L->cur++;
    if (c=='+') return make_tok(TOK_PLUS, s,L->cur,0,NULL);
    if (c=='-') return make_tok(TOK_MINUS,s,L->cur,0,NULL);
    if (c=='*') return make_tok(TOK_STAR, s,L->cur,0,NULL);
    if (c=='/') return make_tok(TOK_SLASH,s,L->cur,0,NULL);
    if (c=='(') return make_tok(TOK_LPAREN,s,L->cur,0,NULL);
    if (c==')') return make_tok(TOK_RPAREN,s,L->cur,0,NULL);
    if (c=='=') return make_tok(TOK_EQ,    s,L->cur,0,NULL);
    if (isdigit((unsigned char)c)) {
        long v = c - '0'; const char*p=L->cur;
        while (isdigit((unsigned char)*p)) { v = v*10 + (*p - '0'); p++; }
        L->cur = p; return make_tok(TOK_INT, s,p,v,NULL);
    }
    if (isalpha((unsigned char)c) || c=='_') {
        const char*p=L->cur; while (isalnum((unsigned char)*p) || *p=='_') p++;
        size_t n = (size_t)(p - (L->cur-1)); // include first char
        char *lex = malloc(n+1);
        memcpy(lex, s, n); lex[n] = 0;
        L->cur = p; return make_tok(TOK_IDENT,s,p,0,lex);
    }
    return make_tok(TOK_ERR, s,L->cur,0,NULL);
}
static void lex_init(Lexer*L,const char*src){ L->src=src; L->cur=src; L->look=next_raw(L); }
static Token peek(Lexer*L){ return L->look; }
static Token take(Lexer*L){ Token t=L->look; L->look=next_raw(L); return t; }

/* ----- AST ----- */
typedef enum { N_INT, N_ADD, N_SUB, N_MUL, N_DIV, N_IDENT, N_ASSIGN } NodeKind;
typedef struct Node { NodeKind k; struct Node *l,*r; long v; const char*name; } Node;

static Node* new_int(Arena*A,long v){ Node*n=arena_alloc(A,sizeof(*n),_Alignof(Node)); if(!n
```

```c
static Node* new_ident(Arena*A,const char*name){ Node*n=arena_alloc(A,sizeof(*n),_Alignof(No
static Node* new_bin(Arena*A,NodeKind k,Node*l,Node*r){ Node*n=arena_alloc(A,sizeof(*n),_Ali
static Node* new_assign(Arena*A,Node*name,Node*expr){ Node*n=arena_alloc(A,sizeof(*n),_Align

/* ----- parser ----- */
typedef struct { Lexer*L; Arena*A; int ok; } Parser;
static void fail(Parser*P,const char*msg,Token t){ P->ok=0; size_t pos=(size_t)(t.s - P->L->

static Node* parse_expr(Parser*P); // forward

static Node* parse_factor(Parser*P){
    Token t = peek(P->L);
    if (t.kind==TOK_INT){ take(P->L); return new_int(P->A, t.ival); }
    if (t.kind==TOK_IDENT){ take(P->L); return new_ident(P->A, t.lexeme); }
    if (t.kind==TOK_LPAREN){ take(P->L); Node*e=parse_expr(P); if(peek(P->L).kind!=TOK_RPAREN
    fail(P,"expected number, name, or '('", t); return NULL;
}
static Node* parse_term(Parser*P){
    Node*n=parse_factor(P);
    while(P->ok){
        TokKind k = peek(P->L).kind;
        if(k!=TOK_STAR && k!=TOK_SLASH) break;
        take(P->L);
        Node*r=parse_factor(P); if(!r) return NULL;
        n=new_bin(P->A, k==TOK_STAR?N_MUL:N_DIV, n, r);
    }
    return n;
}
static Node* parse_expr(Parser*P){
    Node*n=parse_term(P);
    while(P->ok){
        TokKind k = peek(P->L).kind;
        if(k!=TOK_PLUS && k!=TOK_MINUS) break;
        take(P->L);
        Node*r=parse_term(P); if(!r) return NULL;
        n=new_bin(P->A, k==TOK_PLUS?N_ADD:N_SUB, n, r);
    }
    return n;
}
static Node* parse_stmt(Parser*P){
    if (peek(P->L).kind==TOK_IDENT){
        // look ahead for '='
```

```
        Token save = P->L->look;
        Token ident = take(P->L);
        if (peek(P->L).kind==TOK_EQ){
            take(P->L); // consume '='
            Node*rhs = parse_expr(P);
            if (!rhs) return NULL;
            return new_assign(P->A, new_ident(P->A, ident.lexeme), rhs);
        }
        // no '=', rewind
        P->L->look = save;
    }
    return parse_expr(P);
}

/* ----- environment ----- */
typedef struct { const char*name; long value; } Binding;
typedef struct { Binding *items; size_t count, cap; } Env;
static long* env_get(Env*E,const char*name){ for(size_t i=0;i<E->count;i++) if(strcmp(E->iten
static int env_set(Env*E,const char*name,long v){
    long*p=env_get(E,name); if(p){*p=v; return 1;}
    if(E->count==E->cap){ size_t ncap=E->cap?E->cap*2:16; Binding*n=realloc(E->items,ncap*siz
    E->items[E->count++] = (Binding){ strdup(name), v };
    return 1;
}

/* ----- evaluator ----- */
static int eval(Node*n, Env*E, long *out){
    if(!n) return 0;
    switch(n->k){
        case N_INT: *out = n->v; return 1;
        case N_IDENT: {
            long *p = env_get(E, n->name);
            if(!p){ fprintf(stderr,"Name not found: %s\n", n->name); return 0; }
            *out = *p; return 1;
        }
        case N_ADD: { long a,b; if(!eval(n->l,E,&a)||!eval(n->r,E,&b)) return 0; *out=a+b; re
        case N_SUB: { long a,b; if(!eval(n->l,E,&a)||!eval(n->r,E,&b)) return 0; *out=a-b; re
        case N_MUL: { long a,b; if(!eval(n->l,E,&a)||!eval(n->r,E,&b)) return 0; *out=a*b; re
        case N_DIV: {
            long a,b; if(!eval(n->l,E,&a)||!eval(n->r,E,&b)) return 0;
            if(b==0){ fprintf(stderr,"Divide by zero\n"); return 0; }
            *out=a/b; return 1;
```

```
        }
        case N_ASSIGN: {
            long v; if(!eval(n->r,E,&v)) return 0;
            if(!n->l || n->l->k != N_IDENT){ fprintf(stderr,"Left side of '=' must be a name
            if(!env_set(E, n->l->name, v)){ fprintf(stderr,"Env set failed\n"); return 0; }
            *out = v; return 1;
        }
    }
    return 0;
}

/* ----- simple REPL ----- */
int main(void){
    char *line = NULL; size_t n = 0;
    Env env = {0};
    puts("tiny repl. enter expressions or assignments. Ctrl D to exit.");
    while (1){
        printf("> "); fflush(stdout);
        ssize_t m = getline(&line, &n, stdin);
        if (m <= 0) break;

        // strip newline
        if (m>0 && line[m-1]=='\n') line[m-1]=0;

        Arena *A = arena_new(1<<16);
        if(!A){ fprintf(stderr,"arena failed\n"); break; }

        Lexer L; lex_init(&L, line);
        Parser P = { .L=&L, .A=A, .ok=1 };
        Node* root = parse_stmt(&P);

        long result = 0;
        if (P.ok && root && eval(root, &env, &result))
            printf("%ld\n", result);
        else
            fprintf(stderr,"Error\n");

        arena_free(A);
    }
    free(line);
    // free env bindings
    for(size_t i=0;i<env.count;i++) free((void*)env.items[i].name);
```

```
    free(env.items);
    return 0;
}
```

Build and try:

```
gcc -std=c23 -O2 -Wall -Wextra tiny_interp.c -o tiny_interp
./tiny_interp
```

Example session:

```
> 3 + 4*2 - (1 + 5)
5
> x = 10
10
> x + 7
17
> y = x * 3
30
> y / 5
6
```

Step 8. Add built in functions (optional)

You can recognize identifiers like `max` or `min` and parse a function call form `name '(' args ')'`. Then implement small handlers in the evaluator that pop evaluated arguments and return a result.

Step 9. Better numbers

Switch to `double` if you want division with fractions:

- Change value type to `double`
- Print with `%.6g`
- Update divide by zero checks accordingly

Step 10. Why this matters

You now have a complete loop:

- Text
- Tokens
- AST
- Evaluation

This is the heart of configuration languages, query languages, calculators, and many scripting systems. In the next section you will connect this skill to external data by **interfacing with SQLite or LevelDB** from C and building a tiny query tool.

98. Interfacing with SQLite or LevelDB

Time to connect your C programs to real data. In this section you will talk to two popular embeddable databases:

- **SQLite**: relational, SQL queries, ACID transactions in a single file
- **LevelDB**: key value store, ordered by key, fast reads and writes

You will write tiny programs that insert and query data with both engines.

Step 1. When to choose which

- Choose **SQLite** when you want tables, indexes, SQL, and transactions
- Choose **LevelDB** when you want a simple sorted key value store, no SQL, and you control schema in your app

Both are embeddable and require no separate server process.

Step 2. Install headers and libs

On Linux or macOS with Homebrew or apt:

```
# SQLite
sudo apt install libsqlite3-dev        # Debian based
# or
brew install sqlite                    # macOS

# LevelDB
sudo apt install libleveldb-dev        # Debian based
```

```
# or
brew install leveldb                    # macOS
```

Windows users can grab prebuilt binaries or build from source and link the .lib files.

Step 3. Tiny Code for SQLite: create, insert, query

```
// file: sqlite_demo.c
#include <stdio.h>
#include <sqlite3.h>

static int print_row(void *unused, int argc, char **argv, char **col) {
    for (int i = 0; i < argc; i++)
        printf("%s = %s\n", col[i], argv[i] ? argv[i] : "NULL");
    puts("---");
    return 0;
}

int main(void) {
    sqlite3 *db = NULL;
    if (sqlite3_open("people.db", &db) != SQLITE_OK) {
        fprintf(stderr, "open: %s\n", sqlite3_errmsg(db));
        return 1;
    }

    const char *ddl =
        "CREATE TABLE IF NOT EXISTS people ("
        " id INTEGER PRIMARY KEY AUTOINCREMENT,"
        " name TEXT NOT NULL,"
        " age INTEGER NOT NULL"
        ");";
    if (sqlite3_exec(db, ddl, NULL, NULL, NULL) != SQLITE_OK) {
        fprintf(stderr, "ddl: %s\n", sqlite3_errmsg(db));
        return 1;
    }

    // Use prepared statements for safety and speed
    const char *ins = "INSERT INTO people(name, age) VALUES(?, ?);";
    sqlite3_stmt *stmt = NULL;
    if (sqlite3_prepare_v2(db, ins, -1, &stmt, NULL) != SQLITE_OK) {
```

```
        fprintf(stderr, "prepare: %s\n", sqlite3_errmsg(db));
        return 1;
    }

    struct { const char *name; int age; } rows[] = {
        {"Ada", 36}, {"Linus", 55}, {"Grace", 61}
    };
    for (int i = 0; i < 3; i++) {
        sqlite3_reset(stmt);
        sqlite3_clear_bindings(stmt);
        sqlite3_bind_text(stmt, 1, rows[i].name, -1, SQLITE_TRANSIENT);
        sqlite3_bind_int(stmt,  2, rows[i].age);
        if (sqlite3_step(stmt) != SQLITE_DONE) {
            fprintf(stderr, "insert: %s\n", sqlite3_errmsg(db));
            return 1;
        }
    }
    sqlite3_finalize(stmt);

    const char *q = "SELECT id, name, age FROM people WHERE age >= ? ORDER BY age DESC;";
    if (sqlite3_prepare_v2(db, q, -1, &stmt, NULL) != SQLITE_OK) {
        fprintf(stderr, "prepare q: %s\n", sqlite3_errmsg(db));
        return 1;
    }
    sqlite3_bind_int(stmt, 1, 40);

    while (sqlite3_step(stmt) == SQLITE_ROW) {
        int id = sqlite3_column_int(stmt, 0);
        const unsigned char *name = sqlite3_column_text(stmt, 1);
        int age = sqlite3_column_int(stmt, 2);
        printf("id=%d name=%s age=%d\n", id, name, age);
    }
    sqlite3_finalize(stmt);

    sqlite3_close(db);
    return 0;
}
```

Build and run:

```
gcc -std=c23 -O2 sqlite_demo.c -lsqlite3 -o sqlite_demo
./sqlite_demo
```

You should see rows printed for people with age 40 or higher.

Step 4. SQLite best practices in C

- Always use **prepared statements** with `?` placeholders
- Always call `sqlite3_finalize` on statements
- Wrap batches in `BEGIN` and `COMMIT` for speed
- Check every return code and print `sqlite3_errmsg(db)` on error
- Use `sqlite3_last_insert_rowid` to fetch new primary keys

Step 5. Tiny Code for LevelDB: open, put, get, iterate

LevelDB has a C API that mirrors the C++ API.

```
// file: leveldb_demo.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leveldb/c.h>

int main(void) {
    char *err = NULL;

    leveldb_options_t *opts = leveldb_options_create();
    leveldb_options_set_create_if_missing(opts, 1);

    leveldb_t *db = leveldb_open(opts, "kvdb", &err);
    if (err) { fprintf(stderr, "open: %s\n", err); leveldb_free(err); return 1; }

    leveldb_writeoptions_t *wopt = leveldb_writeoptions_create();
    leveldb_readoptions_t  *ropt = leveldb_readoptions_create();

    // Put some keys
    leveldb_put(db, wopt, "name", 4, "Ada", 3, &err);
    if (err) { fprintf(stderr, "put: %s\n", err); leveldb_free(err); err = NULL; }
    leveldb_put(db, wopt, "lang", 4, "C", 1, &err);
    leveldb_put(db, wopt, "year", 4, "1972", 4, &err);
```

```
    // Get a value
    size_t vlen = 0;
    char *val = leveldb_get(db, ropt, "name", 4, &vlen, &err);
    if (err) { fprintf(stderr, "get: %s\n", err); leveldb_free(err); err = NULL; }
    if (val) { printf("name=%.*s\n", (int)vlen, val); leveldb_free(val); }

    // Iterate in key order
    leveldb_iterator_t *it = leveldb_create_iterator(db, ropt);
    leveldb_iter_seek_to_first(it);
    while (leveldb_iter_valid(it)) {
        size_t klen, vlen2;
        const char *k = leveldb_iter_key(it, &klen);
        const char *v = leveldb_iter_value(it, &vlen2);
        printf("%.*s=%.*s\n", (int)klen, k, (int)vlen2, v);
        leveldb_iter_next(it);
    }
    if ((err = (char*)leveldb_iter_get_error(it)) && *err) {
        fprintf(stderr, "iter: %s\n", err);
    }
    leveldb_iter_destroy(it);

    // Clean up
    leveldb_readoptions_destroy(ropt);
    leveldb_writeoptions_destroy(wopt);
    leveldb_close(db);
    leveldb_options_destroy(opts);
    return 0;
}
```

Build and run:

```
gcc -std=c23 -O2 leveldb_demo.c -lleveldb -o leveldb_demo
./leveldb_demo
```

You should see key value pairs in sorted key order.

Step 6. Transactions and durability

- **SQLite** has full transactions
  - Use `BEGIN IMMEDIATE;` then your inserts then `COMMIT;`

  - For crash safety use the default rollback journal or WAL mode

- **LevelDB** has atomic writes per key and write batches

  - Use `leveldb_writebatch_t` to group puts and deletes atomically
  - Sync to disk with `leveldb_writeoptions_set_sync(wopt, 1)`

Step 7. Parameter binding and type safety with SQLite

Use the correct bind and column functions:

- `sqlite3_bind_int`, `sqlite3_bind_int64`, `sqlite3_bind_double`, `sqlite3_bind_text`
- `sqlite3_column_int`, `sqlite3_column_int64`, `sqlite3_column_double`, `sqlite3_column_text`

Never build SQL by string concatenation with user input. Bindings prevent SQL injection and handle escaping for you.

Step 8. Working with binary data

- **SQLite**: use `sqlite3_bind_blob` and `sqlite3_column_blob` with a separate length
- **LevelDB**: keys and values are raw byte spans `(ptr, length)`, so binary is natural

You can store serialized structs, protobufs, or JSON. Remember to define your own versioning for compatibility.

Step 9. Schema and indexing ideas

- **SQLite**

  - Normalize into tables with primary keys and foreign keys
  - Create indexes for frequent lookups
  - Use `PRAGMA foreign_keys = ON;` to enforce constraints

- **LevelDB**

  - Design composite keys to encode access patterns
  - Example: `user:<id>` for user row, `user_email:<email>` points to `<id>`
  - Range scans are easy: store keys like `post:<yyyy-mm>:<id>` and iterate by prefix

Step 10. Why this matters

Embedding a database takes your C program from toy to tool. You now know how to:

- Execute SQL queries and prepared statements with SQLite
- Use a sorted key value engine with LevelDB
- Choose the right storage model for each problem
- Handle durability, binary data, and iteration from C

Try it yourself

1. Extend the SQLite demo with a `BEGIN` and `COMMIT` around a loop of 10000 inserts and measure time.
2. Add an index on `age` and compare query performance.
3. In the LevelDB demo add a write batch that inserts 1000 sequential keys.
4. Store binary blobs in both systems and read them back.
5. Build a tiny CLI that routes `sql ...` lines to SQLite and `kv ...` lines to LevelDB.

Next up is **99. Packaging, Versioning, and Documentation** where you will learn how to ship your code like a pro with Makefiles, pkg config, semantic versioning, and clean README docs.

99. Packaging, Versioning, and Documentation

You've written real C programs—now it's time to **package, version, and document** them like a professional. This is what makes your code usable by others and maintainable by your future self.

Step 1. The goal of packaging

Packaging is about making your project easy to:

- build (`make`, `cmake`, or `meson`)
- install (`make install`)
- link (`pkg-config`)
- use (`#include "yourlib.h"`)

You'll create a structure that helps others build and use your code without guessing.

Step 2. Standard project layout

A simple, conventional layout for a C project:

```
myproject/
  include/
     myproject.h
  src/
     main.c
     util.c
  tests/
     test_basic.c
  Makefile
  README.md
  LICENSE
```

- `include/` holds headers that others can include
- `src/` holds your implementation files
- `tests/` holds unit tests
- `Makefile` defines how to build and install

Step 3. Tiny Code: a simple reusable library

```c
// include/myproject.h
#ifndef MYPROJECT_H
#define MYPROJECT_H

int add(int a, int b);
int sub(int a, int b);

#endif
```

```c
// src/myproject.c
#include "myproject.h"

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
```

Step 4. Minimal Makefile

```
CC      = gcc
CFLAGS  = -std=c23 -O2 -Wall -Iinclude
LDFLAGS =

SRC = $(wildcard src/*.c)
OBJ = $(SRC:.c=.o)
LIB = libmyproject.a

.PHONY: all clean install uninstall

all: $(LIB)

$(LIB): $(OBJ)
    ar rcs $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

install:
    mkdir -p /usr/local/include/myproject
    cp include/*.h /usr/local/include/myproject/
    cp $(LIB) /usr/local/lib/

uninstall:
    rm -f /usr/local/lib/$(LIB)
    rm -rf /usr/local/include/myproject

clean:
    rm -f $(OBJ) $(LIB)
```

Build the library:

```
make
sudo make install
```

Then another project can link against it:

```
gcc main.c -lmyproject -L/usr/local/lib -I/usr/local/include/myproject
```

Step 5. Versioning your releases

Follow **semantic versioning**:

```
vMAJOR.MINOR.PATCH
```

Examples:

- `v1.0.0` – stable release
- `v1.1.0` – new feature, backward compatible
- `v1.1.1` – bug fix, no API change
- `v2.0.0` – breaking API change

Tag your releases in git:

```
git tag -a v1.0.0 -m "First stable release"
git push origin v1.0.0
```

Step 6. Create a pkg-config file

`pkg-config` lets others compile your library easily.

Create `myproject.pc`:

```
prefix=/usr/local
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/myproject

Name: myproject
Description: Tiny math helper library
Version: 1.0.0
Libs: -L${libdir} -lmyproject
Cflags: -I${includedir}
```

Install it in `/usr/local/lib/pkgconfig/` and test:

```
pkg-config --cflags --libs myproject
```

Step 7. Documentation with Markdown and Doxygen

Keep a clear **README.md** at the root:

```
# myproject

A tiny example C library for arithmetic functions.

## Build
```

make sudo make install

````
## Usage
```c
#include <myproject.h>

int main() {
    printf("%d\n", add(3, 4));
}


For API documentation, use **Doxygen**:

```bash
sudo apt install doxygen
doxygen -g
````

Edit `Doxyfile` to include your source paths, then run:

```
doxygen Doxyfile
```

Docs will appear in `html/` or `latex/`.

Step 8. Licensing

Add a `LICENSE` file so others know how they can use your code. Common ones:

- **MIT License**: simple, permissive
- **Apache 2.0**: adds patent protection
- **GPLv3**: ensures derivatives remain open

Example MIT License header for your source files:

```
/*
 * Copyright (c) 2025 Your Name
 * Licensed under the MIT License.
 */
```

Step 9. Continuous Integration (optional)

Add GitHub Actions or another CI service:

```
# .github/workflows/build.yml
name: Build and Test
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - run: make
    - run: make test || echo "No tests yet"
```

Now every push builds automatically.

Step 10. Why this matters

Professional packaging is part of being a systems engineer:

- Your projects build reproducibly.
- Others can install, link, and use them easily.
- Documentation and version tags create confidence.
- Licensing clarifies ownership.

You have now moved from *C programmer* to *C maintainer*, the person others trust to deliver solid, reusable, and well-documented software.

Next is **100. Practice: Build Your Own Mini Project**, where you will bring everything together, writing, building, debugging, and packaging a complete small system in pure C.

100. Practice: Build Your Own Mini Project

You've walked through all the essential layers of C, syntax, memory, data structures, file I/O, compilation, debugging, and even packaging. Now you'll bring it all together by building a complete mini project from scratch.

This final section is a synthesis: plan, design, implement, test, and document a small, useful system in pure C.

Step 1. Choose your project scope

Pick something small enough to finish but rich enough to touch multiple topics. Here are three good options:

Option A: A Tiny Note Manager

- Command line tool to add, list, and delete notes
- Stores data in a simple binary file
- Indexes notes by ID

Option B: A Simple HTTP Server

- Serves static files from a directory
- Uses sockets (`socket`, `bind`, `listen`, `accept`)
- Logs each request to a file

Option C: A Tiny Key-Value Store

- Command line tool with commands `put`, `get`, `list`, `delete`
- Uses `fopen`, `fread`, and `fwrite`
- Optional: add LevelDB or SQLite backend

Step 2. Plan your structure

Example: for the **Tiny Note Manager**

```
tinynotes/
  include/
     tinynotes.h
  src/
     main.c
     notes.c
     util.c
```

```
data/
    notes.bin
Makefile
README.md
LICENSE
```

Step 3. Tiny Code: minimal working version

Below is a working **Tiny Note Manager** in under 150 lines.

```c
// file: tinynotes.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_NOTE_LEN 256
#define DATA_FILE "notes.bin"

typedef struct {
    int id;
    char text[MAX_NOTE_LEN];
} Note;

static void add_note(const char *msg) {
    FILE *f = fopen(DATA_FILE, "ab");
    if (!f) { perror("open"); exit(1); }

    Note n = {0};
    fseek(f, 0, SEEK_END);
    long size = ftell(f);
    n.id = (int)(size / sizeof(Note)) + 1;
    strncpy(n.text, msg, MAX_NOTE_LEN - 1);

    fwrite(&n, sizeof(n), 1, f);
    fclose(f);
    printf("Added note %d: %s\n", n.id, n.text);
}

static void list_notes(void) {
    FILE *f = fopen(DATA_FILE, "rb");
    if (!f) { puts("No notes yet."); return; }
```

```
    Note n;
    while (fread(&n, sizeof(n), 1, f) == 1)
        printf("%d: %s\n", n.id, n.text);
    fclose(f);
}

static void delete_all(void) {
    if (remove(DATA_FILE) == 0)
        puts("All notes deleted.");
    else
        puts("No notes to delete.");
}

int main(int argc, char **argv) {
    if (argc < 2) {
        puts("Usage: tinynotes <add|list|clear> [message]");
        return 0;
    }

    if (strcmp(argv[1], "add") == 0 && argc >= 3) {
        add_note(argv[2]);
    } else if (strcmp(argv[1], "list") == 0) {
        list_notes();
    } else if (strcmp(argv[1], "clear") == 0) {
        delete_all();
    } else {
        puts("Invalid command.");
    }
    return 0;
}
```

Build it:

```
gcc -std=c23 -O2 tinynotes.c -o tinynotes
```

Try it:

```
./tinynotes add "Learn C deeply"
./tinynotes add "Write clear code"
./tinynotes list
```

Output:

```
Added note 1: Learn C deeply
Added note 2: Write clear code
1: Learn C deeply
2: Write clear code
```

Step 4. Extend it

Add these small improvements:

- `tinynotes delete <id>` to remove a note by index
- Store creation time (`time_t`)
- Save to a user-specific directory (`~/.tinynotes/`)
- Encrypt notes with XOR or AES before writing (optional)
- Add JSON export using `fprintf`

Step 5. Package it

Add a Makefile:

```
CC=gcc
CFLAGS=-std=c23 -O2 -Wall
TARGET=tinynotes

all:
    $(CC) $(CFLAGS) tinynotes.c -o $(TARGET)

install:
    cp $(TARGET) /usr/local/bin/

clean:
    rm -f $(TARGET)
```

Install:

```
sudo make install
```

Now you can type `tinynotes list` from anywhere.

Step 6. Document it

README.md

```
# tinynotes

A simple command-line note manager written in C.

## Build
make
sudo make install

## Usage
tinynotes add "hello world"
tinynotes list
tinynotes clear
```

Step 7. Version and license

Tag your release:

```
git tag -a v0.1.0 -m "first public release"
```

Add `LICENSE` (MIT, Apache, or GPL). Publish it on GitHub if you want others to use or contribute.

Step 8. Why this matters

This is how small programs grow into tools:

- Real file I/O
- Error handling
- Build automation
- Documentation and versioning

C gives you the power to build precise, fast, and minimal software. You now know every layer—from compiler to system call.

Step 9. Try it yourself

1. Replace file storage with SQLite or LevelDB
2. Add `search` and `sort`
3. Build a networked version that syncs notes over sockets
4. Add a unit test suite with assertions
5. Package your project as a `.deb` or `.tar.gz`

Step 10. Congratulations

You've reached the end of **The Little Book of C**.

You started from `printf("Hello, World");` and finished with building, packaging, and documenting full working systems.

You now have the foundation to explore:

- **Operating Systems**
- **Compilers and Interpreters**
- **Embedded Systems**
- **Databases and Networking**

C is not just a language. It is the foundation of computing. You now speak it fluently, like a systems engineer.

You've reached the final page, the quiet epilogue of *The Little Book of C.*

Let's close this journey the same way C programs begin: with clarity, purpose, and curiosity.

Epilogue. The Spirit of C

C is not just about syntax, pointers, or the compiler. It is a mindset, one that teaches you to think about **how machines actually work**.

When you write in C, you're speaking the native tongue of computers. You tell the processor what to do, byte by byte, without any illusion between you and the hardware.

You've learned that:

- Every variable has a precise place in memory
- Every function call has a cost on the stack
- Every pointer is a promise to be careful
- Every line you write translates into instructions and data

C rewards those who think deeply and punishes those who guess. But when you master it, you gain a kind of freedom that few languages can match.

The Path Beyond

Now that you can code confidently in C, here are natural next steps:

1. Systems Programming Explore Linux internals, system calls, and kernel modules. Books like *The Linux Programming Interface* or your future "Little Book of System Programming with C" are perfect companions.

2. Compilers and Language Tools Write your own parser or bytecode interpreter. C gives you the precision to build new languages from scratch.

3. Operating Systems and Embedded Try building a tiny OS (like *xv6*), or program microcontrollers with bare-metal C. You'll see how C shapes the firmware world.

4. Libraries and Open Source Contribute to open-source projects written in C, from SQLite to Redis to Git. You'll read world-class C and learn design by example.

5. Build Your Own X in C You can build your own database, HTTP server, shell, or compiler. Each one is a new chance to reapply what you've learned here.

A Note from the Author

When Dennis Ritchie designed C in the 1970s, he wasn't just inventing a language. He was inventing a way to think, about **data, control, and abstraction** at the same time.

Fifty years later, the same clarity still matters. C is timeless because it stays close to truth.

Every byte you allocate, every loop you write, every segmentation fault you fix, it all teaches you how computers *really* work.

So keep experimenting. Break things. Fix them. Rebuild.

That's how every systems engineer begins.

Final Exercise

Before you leave this book, write one last C program. It doesn't need to do anything fancy, just something that reminds you why you love building things from first principles.

```c
#include <stdio.h>

int main(void) {
    printf("I learned to think in C.\n");
    return 0;
}
```

Compile it. Run it. Smile. You now speak the language of the machine.

The Little Book of C *End of Volume*