A COMPLETE COURSE

Programming with C

Fifty-six weeks, from the first program to production code

Sercan Külcü

Giresun University · Department of Computer Engineering

Programming with C

Sercan Külcü
Giresun University, Department of Computer Engineering
Giresun, Türkiye

This book is the printed form of the course notes published at
`sercankulcu.github.io/teaching/programming-with-c/`

Every program in it is in the companion repository at
`github.com/sercankulcu/programming-with-c`, where continuous integration compiles it on Linux, macOS and Windows so the listings here stay honest.

Generated from the notes of 14 November 2026.

Preface

C is small enough to learn completely and unforgiving enough that learning it incompletely shows. This book takes a reader from no prior programming experience to competent, independent C development, in fifty-six chapters meant to be read in order.

It is organised in four parts. **Basic** builds the language core, the early habits, and the first command-line programs, ending with pointers and strings. **Intermediate** covers memory, aggregate data, files, error handling and modular design. **Advanced** covers libraries and tooling, data structures, performance, undefined behavior, portability, the low-level systems interfaces, network programming and concurrency. **Professional** covers working inside an existing codebase, collaboration and review, distributing libraries, interoperating with other languages, scalable I/O, security testing, bare-metal development, and long-term maintenance.

Each chapter assumes only what earlier chapters have established, and nothing is used before it has been explained. Each is anchored to a worked example that is built and explained rather than assigned. The examples form a chain instead of a set of isolated exercises: the growable array of chapter 19 becomes the reusable module of chapter 28, the library of chapter 29, the installable package of chapter 45 and the target of a foreign-language binding in chapter 46; the socket server of chapter 40 becomes the event loop of chapter 47. The embedded chapters target an ARM Cortex-M image running under QEMU, so every example can be built and run without buying hardware.

Each chapter ends with a short set of questions and their answers. They are not homework. They are there because a reader who cannot answer them has not finished the chapter, and the fastest way to find that out is to try.

The code is not decoration. Every listing is in the companion repository, one directory per chapter, and is compiled on three operating systems and two compilers on every change. Where a listing is deliberately wrong — and several are, because a defect you have watched fail teaches more than one you have read about — the text says so.

Sercan Külcü
Giresun

Contents

BASIC *The language itself*

1 Computers, Programs, and the C Toolchain 2
2 History, Standards, and the Compilation Model 12
3 Program Structure, Style, and Basic Output 20
4 Data Representation 30
5 Variables, Types, and Constants 40
6 Operators and Expressions 51
7 Type Conversions 61
8 Formatted Input and Output 73
9 Conditional Statements 84
10 Loops and Flow Control 96
11 Functions 108
12 One- and Multi-Dimensional Arrays 120
13 Introduction to Pointers 131
14 Strings and Character Handling 143

INTERMEDIATE *Memory, data, and programs that last*

15 Pointer Arithmetic and Arrays 155
16 Pointers as Function Parameters 166
17 Scope, Lifetime, and Program Memory Layout 179
18 Recursion 191
19 Dynamic Memory Allocation 205
20 Memory Errors and Their Diagnosis 219
21 Structures 232
22 Unions, Enumerations, and Type Aliases 245
23 Program Arguments and Input 258
24 The Standard Library Toolbox 269
25 Bit-Level Operations 281
26 Files and Streams 294
27 Error Handling Strategies 307

28 The Preprocessor and Modular Programming 322

ADVANCED *Tooling, machines, and systems*

29 Building Programs and Libraries 337
30 Function Pointers, Callbacks, and Generic Programming 348
31 Command-Line Interfaces and the Environment 365
32 Linked Data Structures 379
33 Trees, Hash Tables, and Abstract Data Types 394
34 Performance and Memory Layout 413
35 Debugging, Testing, and Analysis 426
36 Undefined Behavior and Secure Coding 441
37 Portability and the Machine Model 454
38 Type Qualifiers, Inlining, and Modern C 468
39 Processes, Signals, and Low-Level I/O 481
40 Network Programming with Sockets 497
41 Concurrency and the C Memory Model 513
42 Reading Excellent C 529

PROFESSIONAL *Working as a C programmer*

43 Working in an Existing Codebase 540
44 Collaborative Development and Code Review 551
45 Packaging and Distributing C Libraries 563
46 ABI and Interfacing with Other Languages 576
47 Event-Driven and Scalable I/O 590
48 Fuzzing and Security Testing 607
49 Embedded C and Cross-Compilation 620
50 Memory-Mapped I/O and Hardware Registers 632
51 Interrupts and Real-Time Behavior 644
52 Linker Scripts and Firmware Memory Layout 660
53 Constrained-Resource Programming 673
54 Real-Time Operating Systems 691
55 Embedded Debugging and Testing 707
56 Maintaining Long-Lived C Codebases 724

APPENDICES

Appendix A: Prerequisites for Programming with C 740

Appendix B: Important People in C & Procedural Programming 748

PART 1

Basic

The language itself

Chapters 1–14

BASIC · CHAPTER 1

Computers, Programs, and the C Toolchain

Before writing C, it helps to know what a program actually is to a machine, and what the compiler does to your text file to produce one. This chapter builds that picture and then puts a working toolchain in your hands.

BY THE END OF THIS CHAPTER YOU CAN

- Explain what the processor, main memory, and storage each contribute when a program runs.
- Describe the difference between a compiler, an interpreter, and an assembler, and say where C sits.
- Install a C compiler and confirm it works from a terminal.
- Write, compile, and run a first C program from the command line.
- Read a compiler diagnostic, find the line it refers to, and distinguish an error from a warning.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 How a computer executes a program

A running program is not the file you wrote. The file is text. What runs is a sequence of numeric instructions that a particular processor knows how to carry out. Understanding the three pieces of hardware involved makes almost everything later in this course easier to reason about.

The processor

The central processing unit repeats one simple cycle, billions of times per second: fetch the next instruction from memory, decode what it means, execute it, and move on. Each instruction is tiny — add two numbers, copy a value from one place to another, jump to a different instruction if some condition holds. Nothing more sophisticated than that ever happens. Everything a computer does is built from these steps.

The processor keeps a small number of extremely fast storage slots called *registers*. There are only a few dozen of them, and they hold the values the processor is working with right now. A key fact for later: a register holds a value, and it has no name of its own. When you write `int`

`count = 0;` in C, the compiler decides whether `count` lives in a register or in memory, and it may change its mind partway through a function.

Main memory

Main memory, or RAM, is a very long numbered list of bytes. Every byte has an address — a plain integer. That is the whole model. There are no variables in memory, no types, no names; there are only bytes at addresses. Your program's instructions live there, and so does its data.

This is the single most important idea to carry forward. C is unusual among modern languages precisely because it lets you work with those addresses directly. Chapters 13 through 20 are, in a sense, one long consequence of this paragraph.

Storage

Disks and solid-state drives keep files when the power is off. They are enormously larger than memory and enormously slower. Your source file lives there; so does the executable the compiler produces. When you run a program, the operating system copies it from storage into memory and then points the processor at its first instruction.

> **Orders of magnitude.** A register access takes well under a nanosecond. A main-memory access takes tens of nanoseconds. A solid-state drive read takes tens of microseconds — roughly a thousand times longer again. These ratios are why chapter 34 spends a whole session on memory layout and cache behavior.

2 Compilers, interpreters, and assemblers

The processor understands only machine code: raw numbers encoding its instruction set. Humans do not write machine code. Several kinds of programs exist to close that gap, and they close it in different ways.

Assembler

An assembler translates *assembly language* — a thin, human-readable naming of machine instructions — into machine code. The mapping is almost one instruction to one instruction. Assembly is specific to one processor family; assembly written for an ARM chip means nothing to an x86 chip.

Compiler

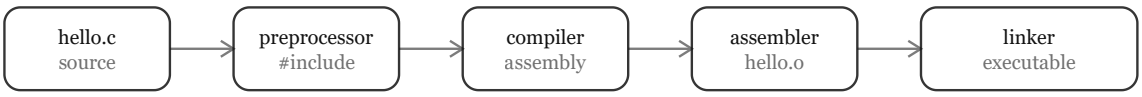
A compiler translates a whole source file written in a higher-level language into machine code, ahead of time. The result is a standalone executable file. Compilation happens once; the

program then runs at full speed, with no translator present. C is a compiled language, and this is why C programs are fast and why the compiler is such a central character in this course.

Interpreter

An interpreter reads source code and carries out its meaning directly, statement by statement, every time the program runs. There is no separate executable. This is convenient — no build step — but slower, because translation happens repeatedly while the program runs. Python and JavaScript are usually run this way.

The practical consequence for you is that C has a *build step*. You will edit a file, run a command, and get a second file. If you change the source and forget to rebuild, you will run the old program and be thoroughly confused. This happens to everyone at least once.



What `gcc hello.c` actually runs. Chapter 2 takes each of these stages apart and inspects its output.

3 Installing a toolchain

You need two things: a compiler and a text editor. Both are free on every platform.

Linux

The compiler is almost certainly one command away. On Debian or Ubuntu:

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

On Fedora, `sudo dnf install gcc gdb make`. On Arch, `sudo pacman -S base-devel gdb`.

macOS

Install the Command Line Tools, which provide Clang under the name `gcc`:

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

Windows

The most useful option is the Windows Subsystem for Linux, because it gives you the same environment the rest of this course assumes:

```
wsl --install
```

Then follow the Linux instructions inside the resulting Ubuntu shell. The alternatives are MSYS2 or MinGW-w64 if you prefer to stay on native Windows.

Confirming it works

```
gcc --version
```

You should see a version banner. If instead you see `command not found`, the compiler is not installed or not on your `PATH`; do not continue until that line produces output.

For an editor, anything that saves plain text will do. Visual Studio Code with the C/C++ extension is a reasonable default. Avoid word processors — a `.c` file must be plain text with no formatting.

> **Use the terminal, at least at first.** An IDE hides the build step behind a button. That is convenient later and harmful now: the entire point of chapters 1 and 2 is to see what the button does. Type the commands yourself until they are boring.

4 Your first program

Create a file called `hello.c`:

```
#include <stdio.h>


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

Compile it and run it:

```
gcc -Wall -Wextra -g -o hello hello.c
./hello
```

On Windows outside WSL the executable will be `hello.exe` and you run it as `hello.exe`.

Line by line

LINE	WHAT IT DOES
`#include <stdio.h>`	Tells the preprocessor to paste in the declarations of the standard input and output library, which is where `printf` comes from. Without it the compiler does not know what `printf` is.
`int main(void)`	Declares the function the operating system calls to start your program. It returns an `int`, and `(void)` says it takes no parameters. Every C program has exactly one `main`.
`{ … }`	Braces group the statements that make up the function body.
`printf("Hello, world\n");`	Calls the library function that writes text to the terminal. `\n` is one character — a newline — not a backslash followed by an `n`. The semicolon ends the statement.
`return 0;`	Ends `main` and hands 0 back to the operating system, conventionally meaning success. Chapter 3 returns to this.

About those flags

Use `-Wall -Wextra` from your very first compilation and never stop. They turn on warnings about code that is legal C but almost certainly a mistake. A large fraction of the bugs this course spends chapters diagnosing are bugs the compiler would have mentioned if asked. `-g` includes debugging information, which you will want from chapter 18 onward, and `-o hello` names the output file.

Without `-o`, the compiler writes a file called `a.out` — a name inherited from 1970s Unix that has outlived every reason for existing.

5 Worked example: breaking it on purpose

The goal this chapter is not to write a correct program. It is to become unafraid of the compiler's output. So take the working `hello.c` and damage it, one change at a time, rebuilding after each.

Experiment 1 — remove the semicolon

```
    printf("Hello, world\n")
    return 0;
```

GCC reports something close to:

```
hello.c: In function 'main':
hello.c:5:29: error: expected ';' before 'return'
    5 |     printf("Hello, world\n")
      |                             ^
      |                             ;
    6 |     return 0;
      |     ~~~~~~
```

Read that carefully, because every diagnostic you will ever see has the same shape. `hello.c:5:29` is file, line, column. `error` is the severity. Then the message, then the offending source line with a caret under the exact position — and here GCC even suggests the fix.

Notice that the compiler blames line 5 but only noticed on line 6. It read `printf(...)`, found no semicolon, kept reading, hit `return`, and only then knew something was wrong. **The reported line is where the compiler gave up, not always where you erred.** Look one line above as a reflex.

Experiment 2 — remove the include

Delete the `#include` line. Depending on your compiler and standard you will get either a warning about an implicit declaration or a hard error. Under a modern compiler:

```
hello.c:4:5: error: implicit declaration of function 'printf' [-Wimplicit-function-
declaration]
```

The compiler has no idea what `printf` is, because nothing told it. This is your first encounter with the fact that C requires everything to be declared before use, which is what headers exist for and what chapter 28 is about.

Experiment 3 — misspell a name

Change `printf` to `printff` while keeping the include. An older compiler only warns about the implicit declaration and proceeds, and then the *linker* fails (GCC 14 and later stop earlier,

with the implicit-declaration error from Experiment 2):

```
/usr/bin/ld: /tmp/ccXXXXXX.o: in function `main':
hello.c:(.text+0xe): undefined reference to `printff'
collect2: error: ld returned 1 exit status
```

This message looks nothing like the earlier ones — different program, different vocabulary, no line-and-column caret. Recognizing *which tool* is complaining is a skill in itself. "undefined reference" always means the linker could not find the body of a function you called. Chapter 29 covers this in depth.

Experiment 4 — an unused variable

Add `int x = 5;` to `main` and rebuild. The program still compiles and runs, but:

```
hello.c:5:9: warning: unused variable 'x' [-Wunused-variable]
```

A **warning** does not stop the build. The compiler is saying that this is legal C but probably not what you meant. Treat warnings as errors in your own habits — later, in chapter 29, you will make the build system enforce it with `-Werror`.

6 Reading compiler messages

Three rules cover most of the first chapters:

Fix the first error only, then rebuild. One mistake often cascades into a dozen messages. The second through twelfth are usually fiction produced by a confused parser. Fixing the first frequently clears them all.

Read the word after the colon. `error` means no program was produced. `warning` means a program was produced that the compiler doubts. `note` is extra context attached to the message above it, not a separate problem.

Look at the caret, then one line up. The caret points at the token where the compiler noticed, which is frequently one token past the real mistake.

MESSAGE	USUAL CAUSE
`expected ';' before …`	Missing semicolon at the end of the previous statement.
`implicit declaration of function 'X'`	Missing `#include`, or the name is misspelled.
`undefined reference to 'X'`	Linker stage: the function was declared but its body was never found. Misspelling, or a missing library such as `-lm`.
`'X' undeclared (first use in this function)`	Using a variable that was never declared, or declared in a different scope.
`expected declaration or statement at end of input`	An unclosed brace. Check the indentation of the whole file.
`control reaches end of non-void function`	A function that promises to return a value has a path that does not.
`format '%d' expects argument of type 'int'`	A `printf` format specifier does not match the argument given. Chapter 8.

> **The classic first-chapter trap.** You fix the source, rebuild in one terminal, and run the old binary from another directory — or you edit `hello.c` but compile `hello2.c`. When a change appears to have no effect whatsoever, the first thing to doubt is whether you actually rebuilt and ran what you think you did.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Running `gcc hello.c` and then `./hello`	`No such file or directory`	Without `-o hello` the output is `a.out`. Always name your output.
Saving the file as `hello.c.txt`	Compiler complains the file type is unrecognized	Turn off "hide known file extensions" in your file manager.
Using curly quotation marks copied from a web page	`stray '\342' in program`	Type the code rather than pasting it, or paste into a plain-text editor first.
Writing `Printf` or `Main`	Implicit declaration, or the program does not link	C is case-sensitive everywhere, always.
Expecting output without `\n`	Text appears late, or merges with the shell prompt	Output is buffered; the newline usually flushes it. Chapter 26 explains why.
Ignoring warnings because the program runs	A bug survives into chapter 20 and costs an evening	Fix every warning the day it appears.

8 Check yourself

Answer before opening each one.

Why does a C program need a build step when a Python program does not?

C is compiled ahead of time into machine code for a specific processor, so translation happens once, before the program runs, and produces a separate executable file. Python is normally interpreted: the translation happens while the program runs, every time. The C approach costs you a build step and buys you speed and a program that runs with no interpreter present.

The compiler reports an error on line 12, but line 12 looks perfectly fine. What should you suspect?

That the actual mistake is on line 11 or earlier. The compiler reports where it could no longer make sense of the text, which is often one token past the omission — most commonly a missing semicolon or an unclosed brace or parenthesis further up.

What is the difference between an error and a warning?

An error means compilation failed and no executable was produced. A warning means an executable *was* produced, but the compiler noticed something that is legal C and probably wrong. Warnings are advice from a tool that has read millions of programs; the professional habit is to treat them as errors.

You get `undefined reference to 'sqrt'` even though you included `<math.h>`. What is going on?

The header told the compiler what `sqrt` looks like, so the compile stage succeeded. But the header does not contain the function's body — that lives in the math library, which the linker must be told to search. Add `-lm` to the command line. The distinction between declaration and definition is exactly what chapter 28 is about.

Where does a variable actually live while a program runs?

In a register, or in main memory at some address — and the compiler decides which, sometimes changing its mind within a single function. The name you wrote does not exist at run time. Holding on to this idea now will make chapters 13 and 17 much easier.

9 Where this leads

You now have a working toolchain and a program that runs. Chapter 2 opens up the single command you have been typing: `gcc -o hello hello.c` silently runs four separate programs in sequence, and you will inspect the output of each one. That chapter also places C historically and explains what changed between C89 and C23 — which matters more than it sounds, because the compiler's behavior depends on which standard you ask for.

BASIC · CHAPTER 2

History, Standards, and the Compilation Model

Last chapter you typed one command and got a program. This chapter you open that command up. Along the way, the history explains why C makes the choices it does, and the standards explain why the same source file can mean different things to different compilers.

BY THE END OF THIS CHAPTER YOU CAN

- Explain why C was created and what problem it solved for the Unix authors.
- Name the major C standards and say what each one added.
- Run the preprocessor, compiler, assembler, and linker separately and inspect each output.
- Choose a language standard deliberately with `-std=` instead of accepting the default.
- Explain why a program that compiles for a classmate might not compile for you.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Where C came from

In 1969 the Bell Labs researchers Ken Thompson and Dennis Ritchie were building an operating system on a spare minicomputer. That system became Unix, and it was written, as most operating systems then were, in assembly language. Assembly is fast and precise, but it is tied to one processor: rewriting Unix for a different machine would have meant rewriting it entirely.

Thompson had already written a small language called **B**, itself a stripped-down version of **BCPL**, which came from Cambridge. B had one significant limitation: every value was a single machine word. It had no notion of a byte, a character, or a floating-point number. When the lab acquired a PDP-11 — a machine with byte addressing — B no longer fit the hardware.

Ritchie's response, between 1971 and 1973, was to add a type system to B. Values now had sizes: `char`, `int`, `float`, `double`. Structures arrived. The result was called C, and in 1973 Unix was rewritten in it. That was the decisive moment: an operating system in a portable language could be moved to a new machine by writing a new compiler rather than a new operating system.

> **Why this matters for you.** C was designed to be a portable assembler for systems programming in the early 1970s, on machines with a few dozen kilobytes of memory. Almost every feature you will find odd — no bounds checking, null-terminated strings, arrays decaying into pointers, undefined behavior — follows from that goal and those constraints. Chapter 42 returns to this once you have enough C to appreciate the reasoning.

2 From a book to a standard

For its first decade, C had no formal definition. What it had instead was a book: *The C Programming Language* by Brian Kernighan and Dennis Ritchie, published in 1978. Programmers call that dialect **K&R C**. The book described the language accurately, but a book is not a specification: where it was silent or ambiguous, compiler vendors each decided for themselves, and C began to fragment.

ANSI convened a committee in 1983 to settle the question. The result, published in 1989, is **ANSI C**, adopted internationally the following year as ISO C. Everyone calls it **C89** or **C90**; they are the same language. This standard introduced the function prototype — declaring a function's parameter types so the compiler can check calls against them — which is the single largest safety improvement in C's history.

Since then the language has been revised roughly once a decade by working group WG14. Revisions are conservative: C is a language where a twenty-year-old source file is expected to still compile.

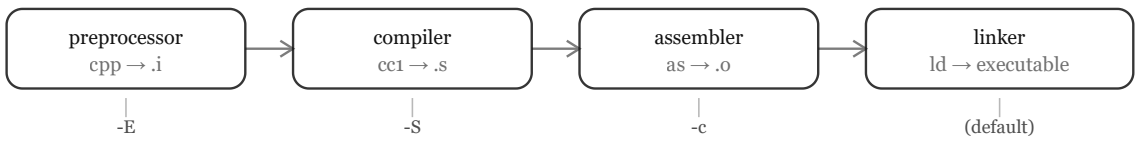
What each revision added

STANDARD	YEAR	NOTABLE ADDITIONS
C89 / C90	1989	Function prototypes, `void`, `const`, `volatile`, a standard library definition. The baseline every C compiler supports.
C99	1999	`//` comments, declarations anywhere in a block, `long long`, `_Bool` with `<stdbool.h>`, fixed-width types in `<stdint.h>`, designated initializers, compound literals, variable-length arrays, `inline`, `restrict`, `snprintf`.
C11	2011	Threads (`<threads.h>`), atomics (`<stdatomic.h>`), `_Generic`, `_Static_assert`, `_Alignas` and `_Alignof`, `_Noreturn`, anonymous structures and unions.
C17 / C18	2018	No new features. A maintenance release that fixed defects in C11. If you target C17 you are targeting C11 with the bugs corrected.
C23	2024	`bool`, `true`, and `false` as keywords, `nullptr`, `constexpr`, binary literals such as `0b1010`, digit separators, `typeof`, attributes such as `[[nodiscard]]`, and the removal of old K&R-style function definitions.

A practical consequence: most of the "modern C" conveniences you will read about online — declaring a loop variable inside the `for`, using `//` for a comment — are C99 features. They are twenty-five years old and universally available, but they are not in C89, and some embedded toolchains still default to C89.

3 The four translation phases

When you run `gcc -o hello hello.c`, GCC is acting as a driver: it runs four separate programs in sequence and quietly deletes the intermediate files. Here is what each one does, and how to stop after it.



Each flag stops the driver after one stage and leaves the intermediate file behind.

Phase 1 — the preprocessor

The preprocessor is a text manipulator. It knows nothing about C. It obeys the lines beginning with `#`: it pastes in the contents of included files, substitutes macros, and strips comments. Its output is still C source — just much longer.

```
gcc -E hello.c -o hello.i
wc -l hello.i
```

Expect several hundred to a few thousand lines from a program you wrote five of. Everything after the last `#` directive is your code; everything before it came from `stdio.h` and the headers it includes in turn. Scroll to the bottom and you will find your `main` unchanged.

Phase 2 — the compiler proper

This is the stage people mean by "compiling": preprocessed C in, assembly language out. All the type checking, all the optimization, and all the interesting diagnostics happen here.

```
gcc -S hello.c -o hello.s
cat hello.s
```

You are not expected to read assembly yet. Look at it anyway. Find the string `Hello, world` sitting in a data section, and find the `call` instruction that invokes `printf`. Recognizing that your program became *this* is worth more than understanding every line. Chapter 34 comes back and reads it properly.

Phase 3 — the assembler

The assembler turns assembly text into a binary object file containing real machine instructions.

```
gcc -c hello.c -o hello.o
file hello.o
nm hello.o
```

`nm` lists the symbols in the object file. You will see `T main` — a symbol this file *defines* in the text section — and `U printf`, a symbol this file *uses* but does not define. That single letter `U` is the reason the next phase exists.

Phase 4 — the linker

The linker collects object files and libraries, resolves every undefined symbol against a definition somewhere, and writes one executable.

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

This is where `printf` gets connected to the real implementation inside the C standard library. It is also where the `undefined reference` error from chapter 1 came from: the linker searched everywhere it was told to look and found no definition.

4 Worked example: the same file, two standards

Save this as `era.c`. It deliberately uses features from different decades.

```
#include <stdio.h>

int main(void)
{
    /* A C89-era comment. */
    int total = 0;

    for (int i = 0; i < 5; i++) {   /* declaration inside for: C99 */
        total += i;
    }

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

Compile it as C89:

```
gcc -std=c89 -Wall -Wextra -o era era.c
```

```
era.c: In function 'main':
era.c:8:5: error: 'for' loop initial declarations are only allowed in C99 or C11 mode
    8 |     for (int i = 0; i < 5; i++) {
      |     ^~~
era.c:8:5: note: use option '-std=c99', '-std=c11' or '-std=gnu99' to compile your code
```

Now as C23:

```
gcc -std=c23 -Wall -Wextra -o era era.c
./era
```

```
total = 10
```

The file did not change. The *language* changed. This is worth sitting with: "valid C" is not a single fixed thing, and the question "does this compile?" is incomplete until you say which standard you meant.

Now walk it through the phases

```
gcc -std=c23 -E era.c -o era.i     && wc -l era.i
gcc -std=c23 -S era.c -o era.s     && grep -c . era.s
gcc -std=c23 -c era.c -o era.o     && nm era.o
gcc era.o -o era                   && ./era
```

Two things to notice. First, the comment you wrote is gone by the time you look at `era.i` — the preprocessor removed it, which is why comments cost nothing at run time. Second, the loop may have vanished from `era.s` entirely if you add `-O2`: the compiler can compute `0+1+2+3+4` at compile time and emit the constant 10. Try it.

```
gcc -std=c23 -O2 -S era.c -o era-opt.s
grep -n '10\|\$10' era-opt.s
```

> **What just happened.** The compiler is not a transcriber. It is allowed to produce any machine code whose observable behavior matches what the standard says your program means. That freedom is what makes C fast, and it is also the reason undefined behavior is so dangerous — a topic chapter 36 is entirely about.

5 Choosing your flags

Settle on a command line now and use it for the rest of the course:

```
gcc -std=c17 -Wall -Wextra -g -o prog prog.c
```

FLAG	WHY
`-std=c17`	States the language explicitly. Without it you get the compiler's default, which differs between compilers and versions, and which is usually a GNU dialect rather than standard C.
`-Wall`	Turns on the common warnings. The name is misleading: it is not all of them.
`-Wextra`	Turns on a further set that `-Wall` omits. Together these two catch a large share of beginner bugs.
`-g`	Embeds debugging information so GDB can show you source lines and variable names. Costs nothing at run time.
`-O0` … `-O2`	Optimization level. `-O0` (the default) while developing, because the debugger behaves predictably; `-O2` when measuring speed.
`-pedantic`	Warns about anything outside the standard you named. Worth switching on once you are comfortable.

> **gnu17 is not c17.** GCC's default is `-std=gnu17`, which is C17 plus GNU extensions. Code that relies on those extensions compiles happily for you and fails for someone using a different compiler. Naming the standard explicitly turns a confusing future bug report into a compile error you see today.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Using `//` comments under `-std=c89`	`C++ style comments are not allowed in ISO C90`	Use `/* … */`, or target C99 or later.
Declaring a variable in the middle of a block under C89	`ISO C90 forbids mixed declarations and code`	Same fix. This is the most common reason old toolchains reject modern code.
Assuming `-E` output is an error	Thousands of lines scroll past	That is the point. Redirect it: `-o file.i`.
Forgetting `-o` with `-c`	Output lands in `hello.o` anyway	Harmless here, but be deliberate: name your outputs.
Debugging a `-O2` build and finding variables "optimized out"	GDB cannot show a value that no longer exists	Debug at `-O0`. Chapter 35 explains why optimized builds confuse debuggers.
Believing the C standard specifies `int` as 32 bits	Code breaks on another platform	It specifies a *minimum* range. Chapter 37 covers what is actually guaranteed.

7 Check yourself

Why was rewriting Unix in C such a significant event?

Because it decoupled the operating system from the processor. An assembly-language system must be rewritten for each new machine; a C system needs only a C compiler for that machine. Portability through a language, rather than through discipline, is the idea C was built to deliver.

Your classmate's program compiles on their laptop and fails on yours with "'for' loop initial declarations are only allowed in C99 or C11 mode". What differs?

The language standard each compiler defaulted to, not the code. Their compiler defaults to a C99-or-later dialect; yours is being invoked in C89 mode. The real fix is not to change the code but for both of you to pass `-std=` explicitly so the question never arises.

What is the difference between `-c` and `-S`?

`-S` stops after the compiler proper and leaves human-readable assembly in a `.s` file. `-c` goes one stage further, running the assembler, and leaves a binary object file `.o`. Neither runs the linker, so neither produces a program you can execute.

`nm hello.o` shows `U printf`. What does the `U` mean and what will resolve it?

`U` means undefined: this object file refers to `printf` but does not contain its body. The linker resolves it by finding the definition in the C standard library, which it searches automatically. For other libraries you must say so — `-lm` for the math library, for example.

You compile with `-O2` and the loop disappears from the assembly. Is this a bug?

No. The compiler must preserve your program's observable behavior, not its literal structure. If it can prove the loop always produces 10, it may emit the constant. This latitude is what makes optimizing compilers effective — and what makes undefined behavior so hazardous, because a program with undefined behavior gives the compiler a false premise to reason from.

8 Where this leads

You now know what the build does and which language you are writing. Chapter 3 turns to the program itself: the anatomy of `main`, what its return value is for, how to write code that another person can read, and how to put your work under version control so that chapter 43's lesson on navigating someone else's codebase starts from a habit you already have.

BASIC · CHAPTER 3

Program Structure, Style, and Basic Output

Every C program has the same skeleton. This chapter takes it apart piece by piece, introduces the output you will use for the rest of the course, and establishes two habits — readable formatting and version control — that separate people who finish projects from people who abandon them.

BY THE END OF THIS CHAPTER YOU CAN

- Explain every line of a minimal C program, including why `main` returns a value and who reads it.
- Use `EXIT_SUCCESS` and `EXIT_FAILURE`, and inspect a program's exit status from the shell.
- Print text and values with `printf`.
- Format and name code so that a stranger can read it, and run a formatter to enforce it.
- Create a Git repository, commit your work, and ignore build products.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The shape of a C program

Here is the smallest useful program, annotated:

```
#include <stdio.h>      /* 1. bring in declarations           */
#include <stdlib.h>     /*    for EXIT_SUCCESS                */

int main(void)          /* 2. where execution begins          */
{                       /* 3. the function body opens         */
    printf("ready\n");  /* 4. one statement, ends in ;        */
    return EXIT_SUCCESS;/* 5. hand a result back to the shell */
}                       /*    the body closes                 */
```

A C source file is a sequence of *declarations* and *definitions* at the outermost level. You cannot put a bare statement outside a function; there is no top-level script. Everything executable lives inside a function body.

`#include` and where names come from

C does not know what `printf` is. Nothing is built in except the language itself — the keywords, the operators, the control structures. Every function you call comes from a library, and before you can call it, the compiler needs its *declaration*: its name, its return type, and the types of its parameters.

A header file is a text file full of such declarations. `#include <stdio.h>` tells the preprocessor to paste that file in at this point, which is why removing the line produced the "implicit declaration" error in chapter 1.

The angle brackets mean "look in the system include directories". Quotation marks — `#include "myheader.h"` — mean "look next to this source file first". You will write your own headers in chapter 28.

HEADER	WHAT IT DECLARES	FIRST NEEDED
`<stdio.h>`	`printf`, `scanf`, `fgets`, file handling	Chapter 3
`<stdlib.h>`	`EXIT_SUCCESS`, `malloc`, `free`, `strtol`, `exit`	Chapter 3
`<string.h>`	`strlen`, `strcpy`, `memcpy`	Chapter 14
`<stdbool.h>`	`bool`, `true`, `false`	Chapter 5
`<math.h>`	`sqrt`, `pow`, `fabs` (link with `-lm`)	Chapter 24

2 `main` and its return value

When the operating system starts your program, it calls `main`. When `main` returns, the program ends and the returned integer is handed back to whatever launched it — usually your shell. This value is the program's **exit status**, and it is not decoration: it is how programs tell scripts whether they succeeded.

```
./hello
echo $?
```

`$?` holds the exit status of the last command. Run it after a successful program and you get `0`. Run it after a failing one and you get something else.

By convention, **0 means success and any non-zero value means failure**. This is backwards from most people's intuition, and there is a reason: there is one way to succeed and many ways to fail, so the non-zero values are free to distinguish between them.

`<stdlib.h>` gives you two names for the common cases:

```
return EXIT_SUCCESS;   /* success; 0 on common platforms */
return EXIT_FAILURE;   /* an implementation-defined non-zero value */
```

Use them. `return 0;` is correct and extremely common, but the named constants say what you mean, and `EXIT_FAILURE` has no fixed numeric spelling you could write by hand portably.

Why this matters early

Shell scripts, build systems, and continuous integration all branch on exit status:

```
./myprogram && echo "worked"
./myprogram || echo "failed"
```

A program that always returns 0, even when it could not open its input file, is a program that silently breaks every automation around it. Chapter 31 designs command-line tools properly; the habit starts here.

> **The two legal forms of `main`.** `int main(void)` and `int main(int argc, char *argv[])`. The second receives the command-line arguments and appears in chapter 23. Anything else — notably `void main()`, which some old textbooks show — is not standard C, even where a compiler tolerates it.

3 Statements, blocks, and punctuation

A **statement** is one step. It ends with a semicolon. The semicolon is a terminator, not a separator — every statement gets one, including the last one in a block.

A **block** is a sequence of statements wrapped in braces. A block can appear anywhere a statement can, and it creates a new scope, which chapter 17 examines in detail.

```
{
    int inner = 1;      /* visible only inside these braces */
    printf("%d\n", inner);
}
/* inner does not exist here */
```

C ignores whitespace almost everywhere. The following is legal and compiles to exactly the same program as the annotated version above:

```
#include <stdio.h>
int main(void){printf("ready\n");return 0;}
```

The compiler does not care. Everyone who ever reads your code does. Which brings us to the real subject of this chapter.

> **The stray semicolon.** `if (x > 0);` is legal C. The semicolon is an empty statement, so the `if` controls nothing and the block after it always runs. The compiler will not stop you, though `-Wextra` may warn. This bug has cost people entire afternoons.

4 Printing with `printf`

`printf` takes a *format string* and then whatever values that string refers to. Inside the format string, a `%` introduces a conversion specifier — a placeholder that says "insert the next argument here, formatted this way".

```
int   count = 42;
double ratio = 0.5;
char  grade = 'A';

printf("count = %d\n", count);
printf("ratio = %f\n", ratio);
printf("grade = %c\n", grade);
printf("text  = %s\n", "literal");
printf("%d items at %f each\n", count, ratio);
```

SPECIFIER	ARGUMENT TYPE
`%d`	`int`
`%u`	`unsigned int`
`%f`	`double` (and `float`, which is promoted)
`%c`	a single character
`%s`	a string
`%zu`	`size_t` — you will need this from chapter 4
`%%`	a literal percent sign

Escape sequences let you write characters you cannot type directly. `\n` is a newline, `\t` a tab, `\\` a backslash, `\"` a double quote. Each is *one* character, not two.

> **Mismatched specifiers are not caught by the language.** `printf("%d\n", 3.5)` is undefined behavior: `printf` will read an `int`-sized piece of whatever was passed and print nonsense, or worse. GCC with `-Wall` catches the common cases — another reason that flag is not optional. Chapter 8 covers the full mechanism and chapter 30 explains why `printf` cannot check for itself.

5 Style: writing for the next reader

You will spend far more time reading code than writing it, and most of what you read will have been written by someone else — including yourself six months ago. Style is not decoration; it is the difference between code you can change safely and code you are afraid of.

Indentation

Indent one level for each nested block. Four spaces is the most common choice; the Linux kernel uses tabs of width eight. Pick one and never mix them within a file — mixed indentation looks correct in your editor and wrong in everyone else's.

Naming

GUIDELINE	GOOD	POOR
Say what it holds	`student_count`	`n2`
Length matches scope	`i` as a loop index	`i` as a global
Consistent word separation	`read_line`, `line_length`	`read_line`, `lineLength`
Constants stand out	`MAX_STUDENTS`	`maxstudents`
No abbreviation games	`average`	`avrg`

C traditionally uses `snake_case` for functions and variables and `UPPER_CASE` for macros and constants. The standard library follows it; so should you.

Comments

Comments should explain *why*, not *what*. The code already says what it does.

```
x = x + 1;              /* add one to x          — worthless */
retries = retries + 1;  /* the modem drops the first byte
                           after power-on, so we retry once */
```

A comment that restates the code is worse than none: it doubles the maintenance burden and eventually contradicts the code it describes.

Let a tool do it

Arguing about brace placement is a waste of a life. Install `clang-format` and let it decide:

```
clang-format -style=llvm -i hello.c
```

Put a `.clang-format` file at the root of a project and every file in it is formatted identically, forever, with no discussion.

6 Worked example: from unreadable to readable

Here is a working program. It compiles cleanly and produces correct output.

```
#include <stdio.h>
int main(void){int a=5,b=3,c;c=a*b;printf("%d\n",c);
int d=0;for(int i=1;i<=c;i++){d+=i;}printf("%d\n",d);return 0;}
```

Read it and answer: what does it print, and what are the two numbers for? The program works and tells you nothing.

Now the same computation, rewritten. Nothing about the behavior changed.

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

int main(void)
{
    const int rows = 5;
    const int columns = 3;

    const int cell_count = rows * columns;
    printf("cells in the grid: %d\n", cell_count);
```

```
    /* Sum 1..cell_count, used later to size the report buffer. */
    int running_total = 0;
    for (int i = 1; i <= cell_count; i++) {
        running_total += i;
    }
    printf("triangular number for %d: %d\n", cell_count, running_total);

    return EXIT_SUCCESS;
}
```

What changed, concretely:

- `a`, `b`, `c`, `d` became names that say what the value is.
- Values that never change are marked `const`, so the compiler enforces that.
- Each statement is on its own line, indented consistently.
- The output labels itself, so the program is diagnosable without reading its source.
- The one comment explains a purpose that is not visible from the code.
- `EXIT_SUCCESS` states the intent of the return value.

Verify that the rewrite really is equivalent — the discipline of checking matters more than the belief:

```
gcc -std=c17 -Wall -Wextra -g -o ugly ugly.c
gcc -std=c17 -Wall -Wextra -g -o clean clean.c
./ugly > a.txt
./clean | grep -o '[0-9]*$' > b.txt
diff a.txt b.txt && echo "same numbers"
```

7 Putting it under version control

From this chapter on, keep your work in Git. Not because the course requires it, but because you will want to know what you changed when something that worked yesterday does not work today — and because chapter 43's exercise in finding a regression with `git bisect` only makes sense if you have a history to bisect.

```
mkdir c-course && cd c-course
git init
git config user.name  "Your Name"
git config user.email "you@example.com"
```

Create a `.gitignore` before your first commit. Compiled output does not belong in a repository: it is large, it changes on every build, and it is reproducible from the source.

```
# Build products
*.o
*.obj
*.a
*.so
*.exe
a.out

# Binaries without an extension are hard to match by pattern,
# so keep them in one directory and ignore that.
/build/
/bin/

# Editor and OS noise
*.swp
.DS_Store
.vscode/
```

Then the loop you will repeat for the next fifty-three chapters:

```
git status                      # what changed
git add week03/clean.c          # stage this file
git commit -m "Rewrite the grid example for readability"
git log --oneline               # what has happened so far
```

Write commit messages in the imperative mood, describing the change rather than the file: `Fix off-by-one in row count`, not `updated main.c`. Your future self is the primary audience, and your future self will be in a hurry.

> **Commit small and often.** A commit should be one idea. That makes history readable, makes a bad change easy to isolate, and makes `git bisect` actually useful. Branching, merging, and reviewing arrive in chapter 44; the single-user habits are enough for now.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Writing `void main()`	Works on some compilers, rejected by others	`int main(void)`. It is the only portable spelling for the no-argument form.
Forgetting to return a status on the failure path	The shell sees success even though the program failed	Return `EXIT_FAILURE` from every path that did not do its job.
`printf(count)` instead of `printf("%d\n", count)`	Compiler error, or a format-string vulnerability if `count` were a string	The first argument is always a format string. Chapter 36 explains why this is a security issue.
Mixing tabs and spaces	Code looks aligned for you, ragged for everyone else	Configure your editor to insert spaces, and run `clang-format`.
Committing `a.out` and `.o` files	Repository bloats; every build shows as a change	Write `.gitignore` before the first commit.
One enormous commit at the end of the chapter	History says nothing; a bad change cannot be isolated	Commit each working step.

9 Check yourself

Why does `main` return a value, and who reads it?

The value is the program's exit status, returned to whatever started the program — normally the shell. Zero conventionally means success and non-zero means failure, which lets scripts and build systems branch on whether your program worked. Inspect it with `echo $?`.

Your program prints the right answer but returns `EXIT_FAILURE`. Does it matter?

To a human watching the terminal, no. To anything automated, yes — a build script, a test harness, or a shell pipeline will treat the run as failed and may stop. Exit status is part of your program's interface, not an afterthought.

What does `#include <stdio.h>` actually do?

It instructs the preprocessor to insert the contents of that header at that point in the file. The header contains *declarations* — names, return types, parameter types — not the function bodies. The bodies live in the C library and are attached later by the linker.

If the compiler ignores whitespace, why bother indenting?

Because the compiler is not the audience. Code is read many more times than it is written, and badly formatted code hides bugs — a misplaced brace or an accidental empty statement is invisible in a dense line and obvious in a formatted one. Delegate the decision to `clang-format` and stop thinking about it.

Why should compiled output be kept out of version control?

It is derived data: fully reproducible from the source by running the build. Storing it makes the repository large, makes every build appear as a change, and creates merge conflicts in binary files that cannot be merged. Track the inputs, not the outputs.

10 Where this leads

You can write, build, format, and version a program. Chapter 4 goes underneath it: how the machine actually stores the numbers and characters you have been printing. That is the foundation for types in chapter 5 and for the conversion pitfalls in chapter 7, and it is what makes the bit-level work of chapter 25 feel obvious rather than arbitrary.

BASIC · CHAPTER 4

Data Representation

Memory holds bytes, and a byte is just eight bits. Everything else — integers, decimals, letters — is an agreement about how to interpret those bits. C exposes that agreement more directly than most languages, so learning it now prevents a long list of surprises later.

BY THE END OF THIS CHAPTER YOU CAN

- Convert between binary, octal, hexadecimal, and decimal, and say why hexadecimal is used at all.
- Explain two's complement and why it is the representation every modern machine uses.
- Describe the three fields of an IEEE 754 floating-point number and explain why `0.1 + 0.2 != 0.3`.
- Explain what a character really is to a C program.
- Write a program that prints the underlying bytes of any value.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Why this chapter exists

A student who skips this material can still write working C for a while. The trouble arrives later, all at once, as a set of apparently unrelated mysteries:

- Why does adding 1 to the largest `int` produce a large negative number?
- Why is `0.1 + 0.2 == 0.3` false?
- Why does `'A' + 1` give `'B'`?
- Why is `if (x < y)` sometimes wrong when one of them is unsigned?
- Why do people write `0xFF` instead of `255`?

Every one of these has the same answer: because of how the value is stored. Half an hour on representation now saves a chapter of confusion in chapters 7, 12, and 25.

2 Positional number systems

Decimal is positional: each digit's contribution depends on its place, and each place is a power of ten. `247` means $2\times10^2 + 4\times10^1 + 7\times10^0$. Nothing about that scheme requires ten. Any base works.

BASE	DIGITS	C NOTATION	USED FOR
Binary (2)	0 1	`0b1010` (C23)	What the hardware actually stores
Octal (8)	0–7	`0755` — a leading zero	Unix file permissions, and little else
Decimal (10)	0–9	`255`	Humans
Hexadecimal (16)	0–9 A–F	`0xFF`	Addresses, bit patterns, colours

Reading binary

`1101` in binary is 1×8 + 1×4 + 0×2 + 1×1 = 13. Going the other way, repeatedly divide by two and collect the remainders from last to first: 13 ÷ 2 = 6 r1, 6 ÷ 2 = 3 r0, 3 ÷ 2 = 1 r1, 1 ÷ 2 = 0 r1 → 1101.

Why hexadecimal

Because sixteen is two to the fourth, **one hex digit is exactly four bits**, always, with no arithmetic required. This makes hex a compact shorthand for bit patterns that a person can convert at sight.

HEX	BITS	HEX	BITS	HEX	BITS	HEX	BITS
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

So `0x3A` is `0011 1010` without thinking, and one byte is always exactly two hex digits. When chapter 25 asks you to set bit 5 of a register, you will write `0x20` and see `0010 0000` in your head.

> **A leading zero means octal.** In C, `012` is not twelve — it is ten. This bites people writing zero-padded constants such as dates or times. `int minutes = 08;` does not even compile, because 8 is not an octal digit.

3 Bits, bytes, and words

A **bit** is one binary digit. A **byte** is the smallest individually addressable unit of memory — eight bits on every machine you will realistically meet. Eight bits give $2^8 = 256$ distinct patterns.

A **word** is the natural size the processor works with: 64 bits on a modern desktop, 32 on many embedded parts. The C type `int` was meant to be one word, but on 64-bit desktops it is still 32 bits.

In C, `sizeof(char)` is 1 *by definition*. All other sizes are measured in units of `char`. The standard guarantees only minimum ranges, not exact sizes — a point chapter 37 makes a great deal of. For now:

TYPE	TYPICAL SIZE	TYPICAL RANGE
`char`	1 byte	−128 … 127, or 0 … 255
`short`	2 bytes	−32 768 … 32 767
`int`	4 bytes	−2 147 483 648 … 2 147 483 647
`long`	8 bytes (4 on Windows)	platform dependent
`long long`	8 bytes	about $\pm 9.2 \times 10^{18}$

4 Signed integers and two's complement

Unsigned numbers are easy: the bits are the number. Negative numbers need a convention, and several have been tried.

The obvious idea, and why it fails

Sign-magnitude reserves the top bit for the sign and treats the rest as the magnitude. It is easy for a human to read, and bad for hardware: it gives two representations of zero (`0000` and `1000`), and addition needs special-case logic depending on the signs.

Two's complement

The representation every modern machine uses — and, since C23, the only one the C standard permits — is **two's complement**. The rule: the most significant bit carries a *negative* weight.

For a 4-bit number the place values are −8, 4, 2, 1:

BITS	ARITHMETIC	VALUE
`0000`	0	0
`0111`	4+2+1	7 (largest)
`1000`	−8	−8 (smallest)
`1111`	−8+4+2+1	−1
`1110`	−8+4+2	−2

Two properties make this the winner. There is exactly **one zero**. And **subtraction is just addition**: the same adder circuit handles signed and unsigned operands with no sign test at all. That is why the hardware does it this way, and therefore why C behaves this way.

Negating a number

Flip every bit, then add one. For 4-bit 5 = `0101`: flip to `1010`, add one → `1011` = −8+2+1 = −5. Correct.

Overflow

The range is asymmetric: there is one more negative value than positive, because zero occupies a slot on the positive side. For a 32-bit `int` the top value is 2 147 483 647. Add one:

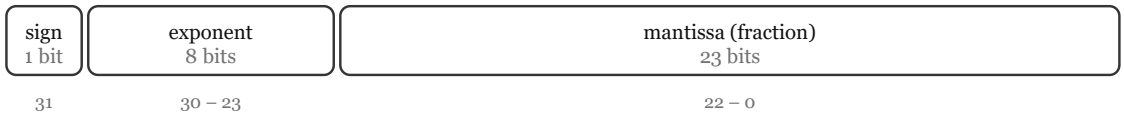
```
0111 1111 … 1111   =  2147483647
             + 1
1000 0000 … 0000   = -2147483648
```

The bits simply carried into the sign position. The value wrapped around.

> **This wrap-around is undefined behavior for signed types.** Unsigned overflow is defined to wrap; signed overflow is not defined at all, and an optimizing compiler is entitled to assume it never happens. That assumption can delete your bounds check. Chapter 36 shows a program that behaves differently at `-O0` and `-O2` for exactly this reason.

5 Floating point: IEEE 754

Real numbers are stored in a binary version of scientific notation. A `float` is 32 bits split into three fields:



A `double` uses the same scheme with 1 + 11 + 52 bits.

The value is roughly $sign \times 1.mantissa \times 2^{exponent - \text{bias}}$. The leading 1 is implicit and not stored, which buys one extra bit of precision for free.

Why 0.1 is not 0.1

In decimal, one third cannot be written exactly: 0.3333… goes on forever. The same thing happens in binary, but for a different set of numbers. One tenth in binary is `0.0001100110011…` repeating forever. With only 23 or 52 bits available, it must be cut off. What gets stored is the nearest representable value, which is *close to* 0.1 but not equal to it.

```
#include <stdio.h>


int main(void)
{
    double a = 0.1, b = 0.2;
    printf("%.20f\n", a + b);
    printf("%s\n", (a + b == 0.3) ? "equal" : "not equal");
    return 0;
}
```

```
0.30000000000000004441
not equal
```

This is not a bug in C, in your compiler, or in your computer. It is a consequence of storing infinitely many real numbers in finitely many bits, and every language using IEEE 754 — Python, Java, JavaScript — behaves identically.

The practical rule, which chapter 7 develops: **never compare floating-point values with `==`**. Compare against a tolerance instead.

The special values

Some bit patterns are reserved. Infinity results from overflow or division by zero; NaN — Not a Number — results from an undefined operation such as 0.0/0.0. NaN has a property that catches everyone once: **it is not equal to itself**, so `x != x` is a valid NaN test.

6 Characters

There is no character type in the hardware. A `char` is a small integer, and the mapping from integer to glyph is a convention. The traditional one is ASCII, which assigns codes 0–127.

CHARACTER	DECIMAL	HEX
`'\0'` (null)	0	0x00
`'\n'` (newline)	10	0x0A
space	32	0x20
`'0'`	48	0x30
`'A'`	65	0x41
`'a'`	97	0x61

Because a character is a number, arithmetic on characters works and is occasionally useful. `'A' + 1` is 66, which is `'B'`. `'7' - '0'` is 7, converting a digit character to its value. The gap between upper and lower case is exactly 32, which is a single bit — `0x20` — and that is why case conversion used to be written as a bitwise operation.

Two things to carry forward. `'\0'`, the character with value zero, marks the end of every C string; that is chapter 14. And ASCII covers only English — Turkish characters such as ş and ğ need more than 127 codes, which is what UTF-8 is for and what chapter 37 covers.

> **Plain `char` has no fixed signedness.** Whether `char` is signed or unsigned is implementation-defined: it is signed on x86 Linux and unsigned on ARM Linux. Code that stores values above 127 in a plain `char` and compares them behaves differently on different machines. Say `signed char` or `unsigned char` when you mean a small number, and reserve plain `char` for text.

7 Worked example: looking at the bytes

Nothing makes this concrete like printing the actual bits. Save as `bits.c`:

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

/* Print the bits of an int, most significant first. */
static void print_int_bits(int value)
{
    for (int bit = 31; bit >= 0; bit--) {
        putchar((value >> bit) & 1 ? '1' : '0');
        if (bit % 8 == 0 && bit != 0) {
            putchar(' ');
        }
    }
    putchar('\n');
}

/* Print the raw bytes of any object, lowest address first. */
static void print_raw_bytes(const void *object, size_t size)
{
    const unsigned char *byte = object;
    for (size_t i = 0; i < size; i++) {
        printf("%02X ", byte[i]);
    }
    putchar('\n');
}

int main(void)
{
    int positive = 13;
    int negative = -13;

    printf("13  decimal = %d, octal = %o, hex = %X\n", positive, positive, positive);
    printf("13  bits    = ");  print_int_bits(positive);
    printf("-13 bits    = ");  print_int_bits(negative);

    printf("\nsum of the two: %d\n", positive + negative);

    float f = 0.1f;
```

```
    printf("\n0.1f occupies %zu bytes: ", sizeof f);
    print_raw_bytes(&f, sizeof f);
    printf("0.1f printed to 20 places: %.20f\n", (double)f);

    printf("\n'A' = %d, 'A' + 1 = %c, '7' - '0' = %d\n",
           'A', 'A' + 1, '7' - '0');

    printf("\nlargest int  = %d\n", 2147483647);
    printf("plus one     = %d\n", 2147483647 + 1);  /* see the note below */

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o bits bits.c
./bits
```

What to look for

Compare the bit patterns of 13 and −13. Flip every bit of the first and add one, and you get the second — the negation rule from section 4, visible.

Look at the four bytes of `0.1f`. On a typical x86 machine you will see `CD CC CC 3D`, which is the pattern `3DCCCCCD` stored *backwards*. That is little-endian byte order, and it is the subject of chapter 37.

Finally, note that GCC warns about `2147483647 + 1`:

```
bits.c:47:46: warning: integer overflow in expression of type 'int'
              results in '-2147483648' [-Woverflow]
```

The compiler computed the constant at compile time, noticed it overflowed, and told you. This is the warning doing exactly the job chapter 1 promised it would.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Writing `010` meaning ten	The value is 8	A leading zero means octal. Drop it.
Comparing floats with `==`	Mathematically equal values compare unequal	Compare `fabs(a - b) < tolerance`. Chapter 7 covers choosing the tolerance.
Expecting signed overflow to wrap	Works at `-O0`, breaks at `-O2`	Signed overflow is undefined. Use a wider type or check before the operation.
Storing a value above 127 in a plain `char`	Behaves differently on x86 and ARM	Use `unsigned char` for byte values.
Printing a `float` with `%d`	Undefined behavior; nonsense output	Use `%f`. Floats are promoted to `double` when passed to `printf`.
Assuming `int` is always 32 bits	Code breaks on another platform	Use `<stdint.h>` types when the width matters. Chapter 37.

9 Check yourself

Why is hexadecimal used so heavily in systems programming rather than decimal?

Because 16 is a power of 2, one hex digit corresponds to exactly four bits with no arithmetic. A byte is always two hex digits, so a hex constant is a compact, directly readable bit pattern. Decimal has no such alignment: converting 200 to bits requires work, while `0xC8` is `1100 1000` at sight.

What are the two properties that made two's complement win over sign-magnitude?

There is a single representation of zero, and addition and subtraction use the same circuitry for signed and unsigned values with no special case for sign. Hardware simplicity drove the choice, and C's arithmetic behavior follows from it.

Is `0.1 + 0.2 == 0.3` false because of a flaw in C?

No. Neither 0.1 nor 0.2 nor 0.3 has an exact binary representation in a finite number of bits, so each is stored as a nearby value and the rounding errors do not cancel. Every language that uses IEEE 754 shows the same result. Compare with a tolerance instead of `==`.

Why does `'7' - '0'` give 7?

Characters are small integers. In ASCII the digit characters are consecutive starting at 48, so `'7'` is 55 and `'0'` is 48; the difference is the digit's numeric value. The standard guarantees the digits are consecutive, so this idiom is portable — unlike the same trick on letters.

Adding 1 to the largest `int` gives a negative number. Can you rely on that?

No. The bit-level explanation is real — the carry lands in the sign position — but signed overflow is *undefined behavior* in C, not defined wrap-around. The compiler may assume it cannot happen and optimize accordingly, so the observed result can change with optimization level. Unsigned types are defined to wrap; signed types are not.

10 Where this leads

You now know what is actually in a byte. Chapter 5 puts names and types on those bytes: how C declares a variable, what each basic type guarantees, and how `sizeof` and `<limits.h>` let a program report its own machine's properties. Chapter 7 then returns to this material to explain what happens when values of different types meet in one expression — which is where the conversions bite.

BASIC · CHAPTER 5

Variables, Types, and Constants

Last chapter you saw what a byte holds. This chapter you give those bytes names and types. A type in C is a promise about two things: how many bytes, and how to interpret them — and the C standard is far looser about the first than most people expect.

BY THE END OF THIS CHAPTER YOU CAN

- Declare and initialize variables of every basic type, and explain what an uninitialized variable contains.
- Choose between `int`, `long`, `unsigned`, `float`, `double`, and `bool` deliberately.
- Use `sizeof` correctly and explain why its result has type `size_t`.
- Report your machine's actual type limits using `<limits.h>` and `<float.h>`.
- Say why `const` and `#define` are not the same thing, and when to use each.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Declarations and initialization

A declaration introduces a name and gives it a type:

```
int   count;          /* declared, not initialized */
int   total = 0;      /* declared and initialized  */
double average = 0.0;
char  grade = 'B';
```

The type comes first, then the name. You can declare several names of the same type at once, though it is usually clearer not to:

```
int width = 10, height = 20;   /* legal */
```

Identifiers

A name may contain letters, digits, and underscores, and may not begin with a digit. C is case-sensitive: `total`, `Total`, and `TOTAL` are three different names. Keywords such as `int`, `return`, and `while` are reserved.

Two conventions worth respecting. Names beginning with an underscore followed by a capital letter, and names beginning with a double underscore, are reserved for the implementation — do not invent your own. And the standard library has already taken a great many ordinary words, so a global variable called `index` or `time` can collide in surprising ways.

The uninitialized variable

This is the first genuinely dangerous thing in C:

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

C does not zero your variables. A local variable that you do not initialize contains whatever bytes happened to be at that memory location — leftovers from a previous function call. The value is not random in any useful sense; it is arbitrary, it may be different on each run, and reading it is undefined behavior.

> **Why this bites so hard.** Uninitialized memory is often zero by accident during early development, so the bug hides. It surfaces later, on a different machine or after an unrelated change, as behavior nobody can reproduce. `-Wall` catches many cases (`-Wmaybe-uninitialized`), and chapter 20 shows how a sanitizer catches the rest. The cheap defence is to initialize every variable at the point of declaration.

2 The basic types

Integers

C has one integer type, `int` , plus modifiers that adjust its size and signedness.

DECLARATION	MEANING	GUARANTEED MINIMUM RANGE
`signed char`	smallest integer	−127 … 127
`unsigned char`	non-negative byte	0 … 255
`short`	at least 16 bits	−32 767 … 32 767
`int`	the natural size	−32 767 … 32 767
`long`	at least 32 bits	−2 147 483 647 … 2 147 483 647
`long long`	at least 64 bits	about ±9.2 × 10^{18}
`unsigned int`	no negatives, one extra bit of range	0 … 65 535

Read that table carefully. The standard guarantees `int` can hold only ±32 767. On your laptop it will be 32 bits, but the guarantee is what portable code may rely on — a distinction chapter 37 turns into a working discipline with `<stdint.h>`.

`signed` is the default for `int`, `short`, `long`, so `signed int` and `int` are the same type. Plain `char` is the exception: whether it is signed is implementation-defined, as chapter 4 noted.

Floating point

TYPE	TYPICAL SIZE	DECIMAL DIGITS OF PRECISION
`float`	4 bytes	about 7
`double`	8 bytes	about 15
`long double`	8, 12, or 16 bytes	platform dependent

Default to `double`. A floating-point literal such as `3.14` is already a `double`; writing `3.14f` makes it a `float`. Modern processors compute in double precision anyway, so `float` buys memory, not speed, and costs you half your significant digits. Use `float` when you have millions of values to store or you are on a small embedded target — chapter 53.

Boolean

C had no boolean type until C99. It has one now:

```
#include <stdbool.h>


bool ready = true;
bool done  = false;
```

Under the hood this is `_Bool`, and `<stdbool.h>` provides the friendlier spellings. In C23 `bool`, `true`, and `false` became real keywords and the header is no longer needed — but including it is harmless and keeps the code compiling under C17.

Before C99, and still in a great deal of existing code, truth is expressed with `int`: **zero is false, every other value is true**. That rule still governs every `if` and `while` you will write, so it is worth internalizing rather than hiding behind `bool`.

3 `sizeof` and `size_t`

`sizeof` reports how many bytes an object or type occupies. It is an operator, not a function, and it is evaluated at compile time.

```
printf("%zu\n", sizeof(int));    /* parentheses required for a type */
printf("%zu\n", sizeof count);   /* optional for an expression      */
```

`sizeof(char)` is 1 by definition, and every other size is a multiple of it. So `sizeof` answers "how many `char`-sized units", which on all mainstream machines means bytes.

Why `%zu` and not `%d`

`sizeof` yields a value of type `size_t` — an unsigned integer type large enough to hold the size of any object. On a 64-bit machine it is typically 64 bits wide, whereas `int` is 32. Printing it with `%d` is a mismatch, which is undefined behavior; `%zu` is the correct specifier.

`size_t` is worth taking seriously now, because it is everywhere from here on: it is the type of every array index you should use (chapter 12), the parameter type of `malloc` (chapter 19), and the return type of `strlen` (chapter 14).

> `size_t` **is unsigned.** A countdown loop written as `for (size_t i = n - 1; i >= 0; i--)` never terminates: an unsigned value is always greater than or equal to zero, so when `i` is 0 and decrements, it wraps to an enormous number. You will meet this again in chapter 7 and again, painfully, in chapter 12.

4 Knowing your machine: `<limits.h>` and `<float.h>`

Rather than guessing what your types can hold, ask. Two headers define the answers as compile-time constants.

CONSTANT	MEANING
`CHAR_BIT`	bits in a `char` — 8 everywhere you will meet
`INT_MIN`, `INT_MAX`	range of `int`
`UINT_MAX`	largest `unsigned int`
`LONG_MIN`, `LONG_MAX`	range of `long`
`LLONG_MAX`	largest `long long`
`FLT_DIG`, `DBL_DIG`	decimal digits of precision
`FLT_EPSILON`, `DBL_EPSILON`	gap between 1.0 and the next larger representable value

`DBL_EPSILON` deserves a note now and a full treatment in chapter 7: it is the standard starting point for choosing the tolerance in a floating-point comparison, because it quantifies exactly how much precision you have.

5 Literals and escape sequences

A literal is a value written directly in the source. Its *type* is determined by how you write it, which matters more than beginners expect.

LITERAL	TYPE	NOTE
`42`	`int`	
`42U`	`unsigned int`	
`42L`	`long`	
`42LL`	`long long`	
`3.14`	`double`	the default for decimals
`3.14f`	`float`	
`0x2A`	`int`	hexadecimal 42
`052`	`int`	octal 42 — leading zero
`0b101010`	`int`	binary, C23
`'A'`	`int` in C, not `char`	a character constant
`"text"`	`char[5]`	four characters plus `'\0'`

The type of a literal decides the type of the expression it sits in. `1 / 2` is integer division and gives 0; `1.0 / 2` gives 0.5. This is the most common arithmetic surprise in the language, and chapter 7 explains the rule behind it.

Escape sequences

ESCAPE	CHARACTER
`\n`	newline
`\t`	tab
`\\`	backslash
`\'` `\"`	quote characters
`\0`	the null character, value 0
`\x41`	the character with hex value 41 — `'A'`

6 Three ways to say "this never changes"

`#define` — textual substitution

```
#define MAX_STUDENTS 100
```

The preprocessor replaces every occurrence of `MAX_STUDENTS` with the text `100` before the compiler runs. There is no variable, no type, and no memory. It works anywhere, including array sizes and `#if` conditions.

`const` — a typed, read-only object

```
const int max_students = 100;
```

This is a real object with a real type that the compiler refuses to let you assign to. It appears in the debugger, participates in type checking, and obeys scope.

`enum` — named integer constants

```
enum { MAX_STUDENTS = 100, MAX_COURSES = 20 };
```

Integer constants with no memory cost, usable where a compile-time constant is required. Chapter 22 covers enumerations properly.

	#DEFINE	CONST	ENUM
Type-checked	no	yes	integers only
Respects scope	no	yes	yes
Visible in a debugger	no	yes	usually
Usable as an array size	yes	C99 VLA only	yes
Works for non-integers	yes	yes	no

Prefer `const` for values with a type, and `enum` for a family of related integers. Reserve `#define` for things the other two cannot do — which, before chapter 28, is mostly array sizes in C89-style code.

> **A `#define` is text, and text has no arithmetic.** `#define HALF 1/2` followed by `double x = HALF;` gives 0, not 0.5, because the substitution produces integer division. Worse, `#define SQUARE(x) x*x` makes `SQUARE(2+3)` expand to `2+3*2+3` = 11. Chapter 28 explains the parentheses that fix this and why a function is usually better.

7 Worked example: a program that describes its own machine

Save as `types.c`. It prints what your compiler actually does, next to what the standard merely promises.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>
#include <stdbool.h>

int main(void)
{
    printf("CHAR_BIT = %d  (bits per byte on this machine)\n\n", CHAR_BIT);

    printf("%-20s %6s  %s\n", "type", "bytes", "range on this machine");
    printf("%-20s %6zu  %d .. %d\n",
           "char",           sizeof(char),      CHAR_MIN,  CHAR_MAX);
    printf("%-20s %6zu  %d .. %d\n",
           "short",          sizeof(short),     SHRT_MIN,  SHRT_MAX);
    printf("%-20s %6zu  %d .. %d\n",
```

```
           "int",            sizeof(int),       INT_MIN,   INT_MAX);
    printf("%-20s %6zu  %ld .. %ld\n",
           "long",           sizeof(long),      LONG_MIN,  LONG_MAX);
    printf("%-20s %6zu  %lld .. %lld\n",
           "long long",      sizeof(long long), LLONG_MIN, LLONG_MAX);
    printf("%-20s %6zu  0 .. %u\n",
           "unsigned int",   sizeof(unsigned),  UINT_MAX);
    printf("%-20s %6zu\n", "float",       sizeof(float));
    printf("%-20s %6zu\n", "double",      sizeof(double));
    printf("%-20s %6zu\n", "long double", sizeof(long double));
    printf("%-20s %6zu\n", "bool",        sizeof(bool));
    printf("%-20s %6zu\n", "size_t",      sizeof(size_t));
    printf("%-20s %6zu\n", "void *",      sizeof(void *));

    printf("\nfloat  : %d significant decimal digits, epsilon = %g\n",
           FLT_DIG, FLT_EPSILON);
    printf("double : %d significant decimal digits, epsilon = %g\n",
           DBL_DIG, DBL_EPSILON);

    puts("\n-- what the standard actually guarantees --");
    puts("char       at least  -127 .. 127");
    puts("short      at least  -32767 .. 32767");
    puts("int        at least  -32767 .. 32767");
    puts("long       at least  -2147483647 .. 2147483647");
    puts("long long  at least  about +/- 9.2e18");

    puts("\n-- the classic surprise --");
    printf("1 / 2       = %d\n",  1 / 2);
    printf("1.0 / 2     = %g\n",  1.0 / 2);
    printf("7 / 2       = %d\n",  7 / 2);
    printf("7 %% 2       = %d\n", 7 % 2);

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o types types.c
./types
```

What to notice

`int` **is 4 bytes but only 2 are guaranteed.** The gap between the two halves of the output is the entire subject of chapter 37. Code that assumes the left column instead of the right is code that breaks when it moves.

`long` **is where portability goes wrong.** On 64-bit Linux and macOS it is 8 bytes; on 64-bit Windows it is 4. A great deal of otherwise-correct code has failed on exactly this.

`sizeof(void *)` **tells you the word size.** 8 on a 64-bit machine, 4 on a 32-bit one. This is the number that decides how much memory a process can address.

`1 / 2` **is 0.** Both operands are `int`, so C performs integer division and discards the remainder. Writing `1.0 / 2` makes one operand a `double` and the whole expression follows. Remember this the next time an average comes out wrong.

An experiment worth running

Add these two lines and rebuild:

```
    int uninitialized;
    printf("uninitialized = %d\n", uninitialized);
```

GCC will warn:

```
types.c:52:5: warning: 'uninitialized' is used uninitialized
              [-Wuninitialized]
```

Run it several times, and at different optimization levels. The value may be 0, may be garbage, and may change. That instability is exactly why the language calls it undefined rather than unspecified.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Reading an uninitialized variable	Undefined behavior; unreproducible bugs	Initialize at the point of declaration, always.
Printing `sizeof` with `%d`	Undefined behavior; wrong number on 64-bit	Use `%zu`.
Counting down with an unsigned index	Infinite loop	Loop upward, or use a signed type, or restructure the condition.
Expecting `1 / 2` to be 0.5	Silent integer division	Make one operand floating point: `1.0 / 2`.
Using `float` by habit	Seven digits of precision where you needed fifteen	Default to `double`.
`#define HALF 1/2`	Textual substitution yields integer division	Parenthesize, or use `const double half = 0.5;`.
Assuming `long` is 64 bits	Works on Linux, breaks on Windows	Use `long long`, or `int64_t` from chapter 37.

9 Check yourself

What does an uninitialized local variable contain?

Whatever bytes were already at that memory location — typically leftovers from earlier function calls. It is not zero, not random in any dependable sense, and reading it is undefined behavior. Initialize every variable where you declare it.

Why does `sizeof` return `size_t` rather than `int`?

Because a size cannot be negative and can exceed the range of `int` on a 64-bit machine. `size_t` is an unsigned type guaranteed to hold the size of any object. Print it with `%zu`; using `%d` is a format mismatch and undefined behavior.

Why does `for (size_t i = n - 1; i >= 0; i--)` never end?

`size_t` is unsigned, so `i >= 0` is always true. When `i` reaches 0 and is decremented it wraps to the largest representable value rather than becoming −1. Either loop upward, or write the condition as `i--> 0` using the value before the decrement.

Give one thing `const` does that `#define` cannot, and one thing `#define` does that `const` cannot.

`const` has a type, so the compiler type-checks its use, it obeys scope, and a debugger can show it. `#define` produces a compile-time constant expression usable as a fixed array size in standard C89, where a `const int` cannot be. That is essentially the only remaining reason to prefer it for plain values.

Your program computes an average as `int total / int count` and gets 3 instead of 3.5. Why, and what is the minimal fix?

Both operands are integers, so C performs integer division and truncates. The minimal fix is to make one operand floating point — `(double)total / count`. Casting the *result* instead, as in `(double)(total / count)`, is too late: the truncation has already happened.

10 Where this leads

You can now name and size every basic value. Chapter 6 puts those values into expressions: what the operators do, how precedence decides the shape of an expression, and why the order in which C evaluates operands is less fixed than it looks. Chapter 7 then combines chapters 4, 5, and 6 into the single topic that produces more quiet bugs than any other in the language — what happens when types mix.

BASIC · CHAPTER 6

Operators and Expressions

Operators are the part of C that looks most familiar and hides the most surprises. Precedence decides what an expression means; evaluation order decides when each piece happens — and C deliberately leaves much of the second unspecified.

BY THE END OF THIS CHAPTER YOU CAN

- Use every arithmetic, relational, and logical operator correctly, including `%` with negative operands.
- Predict the result of an expression from the precedence and associativity tables.
- Explain the difference between `i++` and `++i` and when it matters.
- Explain why short-circuit evaluation is a guarantee you can rely on, and argument evaluation order is not.
- Recognize an expression whose behavior the standard does not define.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Arithmetic operators

OPERATOR	MEANING	EXAMPLE	RESULT
`+`	addition	`7 + 2`	9
`-`	subtraction	`7 - 2`	5
`*`	multiplication	`7 * 2`	14
`/`	division	`7 / 2`	3 (integers!)
`%`	remainder	`7 % 2`	1

Integer division truncates

If both operands are integers, `/` performs integer division and discards the fractional part. `7 / 2` is 3, not 3.5, and not 4 — the result is truncated toward zero, never rounded.

To get a real quotient, at least one operand must be floating point:

```
7 / 2          /* 3     */
7.0 / 2        /* 3.5   */
(double)7 / 2  /* 3.5   */
(double)(7/2)  /* 3.0 — too late, truncation already happened */
```

% works on integers only

`7.5 % 2` does not compile. For floating point, use `fmod` from `<math.h>`.

The sign of the remainder follows the *dividend*: `-7 % 2` is −1, not 1. This is a consequence of truncating division, and it means `%` is not a modulo operator in the mathematical sense. A common test for evenness, `n % 2 == 1`, therefore fails for negative odd numbers; write `n % 2 != 0` instead.

> **Division by zero is undefined behavior, not an error value.** Integer `x / 0` typically crashes the program with a hardware exception. Floating-point division by zero is different: IEEE 754 defines it, and `1.0 / 0.0` yields infinity. Never assume C will hand you a convenient sentinel — check the divisor.

2 Relational and logical operators

OPERATOR	MEANING
`==` `!=`	equal, not equal
`<` `<=` `>` `>=`	ordering
`&&`	logical and
`||`	logical or
`!`	logical not

A relational operator produces `int` 1 for true and 0 for false. In a condition, **zero is false and everything else is true** — so `if (count)` means "if count is not zero".

Short-circuit evaluation is guaranteed

`&&` and `||` evaluate their left operand first, and if that settles the answer, **the right operand is never evaluated at all**. This is not an optimization; it is required by the standard, and correct C relies on it constantly.

```
if (divisor != 0 && total / divisor > 10) { … }
```

If `divisor` is zero, the division never happens. Reverse the two tests and the program crashes. From chapter 13 onward the same pattern guards pointers:

```
if (p != NULL && p->count > 0) { … }
```

The comma operator and the conditional operator also impose an order. Everything else — notably the operands of `+`, `*`, and the arguments of a function call — does not.

> **`=` is assignment; `==` is comparison.** `if (x = 5)` assigns 5 to `x` and then tests 5, which is true — always. It is legal C, so only a warning saves you. `-Wall` emits *suggest parentheses around assignment used as truth value*. Some programmers write `if (5 == x)` so that a typo becomes a compile error; the modern answer is to trust the warning and never ignore it.

3 Assignment, compound assignment, increment

Assignment in C is an *expression*, not a statement: it has a value, which is the value assigned. That is why `a = b = 0` works — it groups right to left.

Compound assignment

```
total += 5;     /* total = total + 5  */
total -= 5;
total *= 2;
total /= 2;
total %= 3;
```

These are not merely shorthand. In `arr[next_index()] += 1` the left operand is evaluated once; writing it out longhand would call `next_index()` twice.

Increment and decrement

`++` and `--` each come in two forms that differ in what the *expression* yields:

```
int i = 5;
int a = i++;   /* a = 5, i = 6 — post: yields the old value */

int j = 5;
int b = ++j;   /* b = 6, j = 6 — pre:  yields the new value */
```

When the value is discarded — as in the third clause of a `for` loop — the two are identical, and `i++` is conventional. When the value is used, choose deliberately.

> **An old myth.** You may read that `++i` is faster than `i++` because the latter must keep a copy. For an `int` on any compiler released this century, the generated code is identical — you can confirm this yourself with the `-S` technique from chapter 2. The advice has real force in C++ for iterator types; in C it does not.

4 Precedence and associativity

Precedence decides which operator binds tighter: in `a + b * c`, `*` binds tighter, so it is `a + (b * c)`. **Associativity** decides the grouping among operators of equal precedence: `a - b - c` is `(a - b) - c` because `-` is left-associative.

PRECEDENCE	OPERATORS	ASSOCIATIVITY
highest	`()` `[]` `->` `.` postfix `++` `--`	left to right
	unary `!` `~` `++` `--` `+` `-` `*` `&` `sizeof` casts	right to left
	`*` `/` `%`	left to right
	`+` `-`	left to right
	`<<` `>>`	left to right
	`<` `<=` `>` `>=`	left to right
	`==` `!=`	left to right
	`&` then `^` then `|`	left to right
	`&&` then `||`	left to right
	`?:`	right to left
	`=` `+=` `-=` …	right to left
lowest	`,`	left to right

Two rows in that table are responsible for a disproportionate share of real bugs.

Bitwise operators bind looser than comparison

This is widely regarded as a design mistake in C, preserved for compatibility:

```
if (flags & MASK == 0)     /* means: flags & (MASK == 0)  — wrong */
if ((flags & MASK) == 0)   /* what you meant                      */
```

Chapter 25 uses this constantly. Parenthesize bitwise expressions as a reflex.

Assignment binds looser than almost everything

```
x = a < b;      /* x = (a < b) — x gets 0 or 1 */
```

> **The practical rule.** Memorize five levels — unary tightest, then arithmetic, then comparison, then logical, then assignment — and parenthesize everything else. Nobody is impressed by an unparenthesized expression that requires the table to read. Clear beats clever, and the compiler produces identical code either way.

5 Evaluation order: the part that is not defined

Precedence tells you how an expression is *grouped*. It says nothing about the *order in which the pieces are computed*. These are different questions, and conflating them is a classic source of confusion.

In `f() + g()`, precedence guarantees the two results are added. It does not guarantee that `f` runs before `g`. The compiler may call them in either order, and may choose differently at different optimization levels.

That is merely *unspecified* — one of a set of allowed behaviors. Worse is available:

```
int i = 0;
int a = i++ + i++;        /* undefined behavior */
arr[i] = i++;             /* undefined behavior */
printf("%d %d\n", i++, i++);  /* undefined behavior */
```

The rule: if an expression modifies an object more than once, or modifies it and separately reads it for a purpose other than computing the new value, without an intervening *sequence*

point, the behavior is undefined. Not implementation-defined — undefined. The compiler may produce any result at all.

Where the sequence points are

CONSTRUCT	ORDERING GUARANTEE
`;` end of a full expression	everything before completes first
`&&` and `||`	left fully evaluated before right; right may be skipped
`?:`	condition evaluated first; exactly one branch runs
`,` (the comma operator)	left fully evaluated, its value discarded, then right
function call	all arguments evaluated before the body runs — in *unspecified* order

Enable `-Wall` and GCC warns about the obvious cases: *operation on 'i' may be undefined*. Do not treat that as advisory.

> **Why the language is like this.** Leaving evaluation order open lets the compiler reorder computation to use registers and pipelines efficiently — and in 1972, on a PDP-11, that mattered enormously. The cost is that a small set of expressions have no defined meaning. The fix costs nothing: put each side effect in its own statement.

6 Worked example: predict, then verify

Write your answers down before running this. The point of the exercise is the gap between your prediction and the output.

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

static int trace(const char *label, int value)
{
    printf("  [evaluating %s]\n", label);
    return value;
}

int main(void)
{
    puts("== integer division and remainder ==");
```

```
    printf("7 / 2        = %d\n",  7 / 2);
    printf("-7 / 2       = %d\n",  -7 / 2);
    printf("7 %% 2        = %d\n", 7 % 2);
    printf("-7 %% 2       = %d\n", -7 % 2);
    printf("7.0 / 2      = %g\n",  7.0 / 2);

    puts("\n== precedence ==");
    printf("2 + 3 * 4        = %d\n", 2 + 3 * 4);
    printf("(2 + 3) * 4      = %d\n", (2 + 3) * 4);
    printf("10 - 4 - 3       = %d\n", 10 - 4 - 3);
    printf("1 << 2 + 3       = %d\n", 1 << 2 + 3);   /* 1 << (2+3) */
    printf("(1 << 2) + 3     = %d\n", (1 << 2) + 3);

    puts("\n== the bitwise-versus-comparison trap ==");
    int flags = 0x0C;          /* 0000 1100 */
    int mask  = 0x04;          /* 0000 0100 */
    printf("flags & mask == 0    -> %d   (this is flags & (mask == 0))\n",
           flags & mask == 0);
    printf("(flags & mask) == 0  -> %d   (what you meant)\n",
           (flags & mask) == 0);

    puts("\n== pre versus post increment ==");
    int i = 5;
    printf("i++ yields %d, i is now %d\n", i++, i);
    int j = 5;
    printf("++j yields %d, j is now %d\n", ++j, j);

    puts("\n== short-circuit is guaranteed ==");
    int divisor = 0;
    if (divisor != 0 && 100 / divisor > 10) {
        puts("  not reached");
    }
    puts("  survived a division by zero that never happened");

    puts("\n== argument order is NOT guaranteed ==");
    int sum = trace("left", 1) + trace("right", 2);
    printf("  sum = %d\n", sum);
```

```
    printf("  rerun with -O2 and the two lines may swap\n");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o expr expr.c
./expr
```

What the compiler says before you even run it

```
expr.c:30:18: warning: suggest parentheses around comparison in operand of '&'
              [-Wparentheses]
expr.c:36:5: warning: operation on 'i' may be undefined [-Wsequence-point]
expr.c:38:5: warning: operation on 'j' may be undefined [-Wsequence-point]
```

All of these warnings point at real defects. The sequence-point one is worth dwelling on: `printf("… %d … %d", i++, i)` reads `i` and modifies it within one expression with no sequence point between, so the output is not merely platform-dependent — it is undefined. Fix it by separating the statements:

```
int old = i;
i++;
printf("i++ yields %d, i is now %d\n", old, i);
```

The experiment that makes the point

Rebuild at two optimization levels and compare the last section's output:

```
gcc -std=c17 -O0 -o expr0 expr.c && ./expr0 | tail -4
gcc -std=c17 -O2 -o expr2 expr.c && ./expr2 | tail -4
```

Here the two calls are operands of `+`, not function arguments, and GCC and Clang usually evaluate them left to right, so you will likely see `[evaluating left]` first on both builds — but nothing in the standard promises it. Either outcome teaches the lesson: the sum is always 3, but *when* each side effect happens is the compiler's business, not yours.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`if (x = 5)`	Assigns, then always tests true	`==` for comparison. Never ignore `-Wparentheses`.
`flags & MASK == 0`	Parses as `flags & (MASK == 0)`	Parenthesize bitwise expressions always.
`n % 2 == 1` to test odd	False for negative odd numbers	`n % 2 != 0`.
`average = total / count` with two `int`s	Truncated result	`(double)total / count`.
`a = i++ + i++`	Undefined behavior	One side effect per statement.
Relying on left-to-right argument evaluation	Works on one compiler, breaks on another	Compute into named variables first.
Dividing without checking the divisor	Crash on integer division by zero	Guard with `&&`, relying on short-circuit.

8 Check yourself

What is `-7 / 2` and `-7 % 2` in C, and why?

−3 and −1. Integer division truncates toward zero rather than rounding down, so −3.5 becomes −3; the remainder then takes the sign of the dividend so that `(a / b) * b + a % b == a` still holds. This is why `%` is a remainder operator, not a mathematical modulo.

Why is `if (p != NULL && p->count > 0)` safe but the reverse order is not?

Because `&&` guarantees short-circuit evaluation: if the left operand is false, the right is never evaluated, so the null pointer is never dereferenced. Written the other way round, the dereference happens first and the program crashes. This is a language guarantee, not an optimization.

What is the difference between precedence and evaluation order?

Precedence is about grouping — which operands belong to which operator, and therefore what the expression *means*. Evaluation order is about timing — when each subexpression is actually computed. C fixes the first completely and leaves much of the second to the compiler, which is why `f() + g()` has a defined value but no defined call order.

Why is `arr[i] = i++;` undefined rather than merely unpredictable?

Within one expression it both modifies `i` and reads `i` for a purpose unrelated to computing the new value, with no sequence point between. The standard does not enumerate the permitted outcomes; it withdraws all requirements, so the compiler may produce anything. Split it into two statements.

When do `i++` and `++i` behave identically?

Whenever the value of the expression is discarded — most commonly the third clause of a `for` loop, or a statement consisting of the increment alone. They differ only when the resulting value is used. For an `int`, the generated machine code is the same either way.

9 Where this leads

Every example this chapter used operands of the same type. Chapter 7 removes that restriction and asks what C does when an `int` meets an `unsigned`, or a `char` meets a `double`. The answer is a fixed set of conversion rules that run silently before every operation — and the single most common source of bugs that compile without a warning and behave correctly right up until they do not.

Type Conversions

When operands of different types meet, C converts them before doing anything. The rules are fixed, silent, and applied to every expression you write. This is the chapter that explains bugs which compile without a warning and behave perfectly until the day they do not.

BY THE END OF THIS CHAPTER YOU CAN

- Apply integer promotion and the usual arithmetic conversions to predict an expression's type.
- Explain why comparing a signed and an unsigned value can produce a mathematically wrong answer.
- Use a cast deliberately, and recognize when a cast is hiding a bug rather than fixing one.
- Describe what happens when a value does not fit the type it is assigned to.
- Compare two floating-point numbers correctly, and detect `NaN` and infinity.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Conversions happen whether you ask or not

C is a statically typed language with a permissive attitude: rather than rejecting an expression whose operands differ, it converts them to a common type and proceeds. These *implicit conversions* occur in four places:

- In an arithmetic or comparison expression, to bring both operands to one type.
- On assignment, to convert the right-hand value to the left-hand type.
- When passing an argument to a function, to match the parameter type.
- On `return`, to match the function's declared return type.

Most of the time this is convenient and correct. The remainder of this chapter is about the cases where it is not.

2 Integer promotion

Before any arithmetic, operands smaller than `int` are converted to `int`. This applies to `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, and `_Bool`.

The reason is historical and practical: early processors had no arithmetic on sub-word operands, so the language defined arithmetic to happen at word width.

```
char a = 100;
char b = 100;
char c = a + b;      /* a and b promoted to int: 200. Stored back
                        into a char, which cannot hold 200. */
```

The addition itself is fine — it happens in `int` and yields 200. The damage is done on assignment, where 200 is squeezed back into a `char`, which cannot hold it if plain `char` is signed.

Promotion also explains something that puzzles beginners:

```
printf("%zu\n", sizeof(char));      /* 1 */
printf("%zu\n", sizeof('A'));       /* 4 — in C, 'A' is an int  */
printf("%zu\n", sizeof(char) + 1);  /* 2, computed in size_t, not int */
```

A character constant such as `'A'` has type `int` in C, not `char`. This differs from C++, and it occasionally matters.

3 The usual arithmetic conversions

Once both operands are at least `int`, C brings them to a common type by walking down this list and stopping at the first case that applies:

STEP	RULE
1	If either is `long double`, convert the other to `long double`.
2	Otherwise if either is `double`, convert the other to `double`.
3	Otherwise if either is `float`, convert the other to `float`.
4	Otherwise both are integers: apply integer promotion, then convert toward the type of greater rank.
5	If the two have equal rank but differ in signedness, **the signed one is converted to unsigned**.

Step 5 is the one that draws blood. The rest are intuitive: mixing an `int` with a `double` gives a `double`, which is what anyone would expect.

Worked through:

```
int    i = 3;
double d = 2.0;
i / 2          /* int / int    -> int    -> 1   */
i / 2.0        /* int / double -> double -> 1.5 */
i / d          /* int / double -> double -> 1.5 */
(i + 1) / 2    /* 4 / 2        -> int    -> 2   */
```

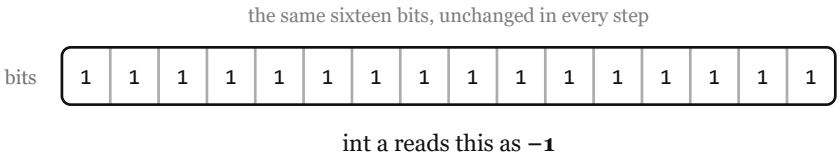
4 The signed-unsigned trap

Here is the single most notorious conversion bug in C:

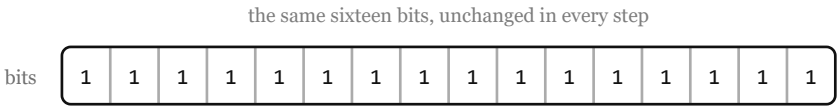
```
int      a = -1;
unsigned b = 1;

if (a < b) {
    puts("-1 is less than 1");     /* never printed */
} else {
    puts("-1 is NOT less than 1"); /* this one runs */
}
```

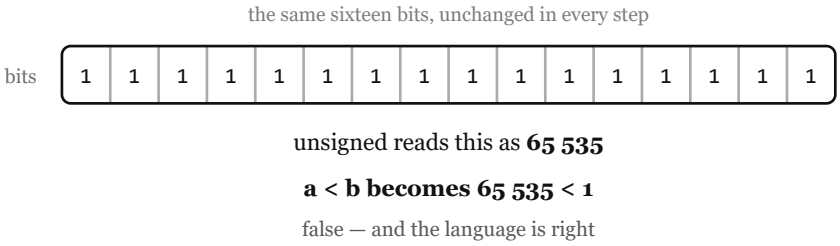
Both operands have rank `int`, and one is unsigned, so rule 5 applies: `a` is converted to `unsigned int`. In two's complement, −1 becomes 4 294 967 295. That number is not less than 1, so the comparison is false — and it is *correct* according to the language. The bug is in the program, not the compiler.



`int a = -1;` — in two's complement that is every bit set. (Sixteen bits drawn; a real `int` has thirty-two.)



Now compare it with an `unsigned`. Both operands have rank `int` and one is unsigned, so `a` is converted to unsigned. **Nothing in memory changes.** Only the reading does.



So `a < b` asks whether 65 535 is less than 1. It is not. The comparison is false, the compiler is correct, and the bug is in the program — which is exactly why it compiles without complaint.

The same mechanism turns loops into infinite loops:

```
size_t n = 0;
for (size_t i = 0; i < n - 1; i++) {
    /* n - 1 is unsigned. 0 - 1 wraps to SIZE_MAX.
       This loop runs about 18 quintillion times. */
}
```

And it is why `strlen` returning `size_t` deserves care:

```
if (strlen(s) - 1 >= 0)   /* always true: unsigned is never negative */
```

Defences

- Compile with `-Wsign-compare`, which `-Wextra` enables. It flags exactly this.
- Do not mix signedness in one expression. Pick a convention per function and convert at the boundary.
- Rearrange rather than subtract: write `i + 1 < n` instead of `i < n - 1`.

- Use unsigned types for quantities that genuinely cannot be negative — sizes, indices, bit patterns — and signed types for everything else.

> **Unsigned arithmetic never overflows; it wraps.** That is defined behavior, computed modulo 2^N. It is not an error, nothing is reported, and the program continues with a wrong value. Signed overflow, by contrast, is undefined behavior — see chapter 36. Neither is a safety net.

5 Explicit conversion: the cast

A cast forces a conversion:

```
double average = (double)total / count;
```

Placement matters entirely. `(double)total / count` converts `total` first, so the division is `double / int` → `double`. `(double)(total / count)` performs integer division first and then widens the already-truncated result. The parentheses are the whole difference between a correct average and a wrong one.

When a cast is right

- Forcing floating-point division between integers, as above.
- Silencing a legitimate narrowing where you have already checked the range.
- Converting `void *` to a typed pointer — chapter 30.
- Matching the type a variadic function expects, for example `(double)x` before a `printf("%f")`.

When a cast is wrong

A cast tells the compiler "I know what I am doing." If you do not, the cast merely deletes the warning that would have told you. A cast added to make a warning go away, without a reason you could state out loud, is a bug with the alarm disconnected.

```
/* Wrong: hides a real narrowing problem */
short s = (short)some_large_int;

/* Right: the check makes the cast honest */
if (some_large_int >= SHRT_MIN && some_large_int <= SHRT_MAX) {
    short s = (short)some_large_int;
}
```

6 When the value does not fit

CONVERSION	WHAT HAPPENS
Larger integer → smaller integer	The high-order bits are discarded. For unsigned targets this is defined modulo arithmetic; for signed targets it is implementation-defined.
Negative → unsigned	Defined: add 2^N. −1 becomes the maximum value.
`double` → integer	The fractional part is discarded — truncated toward zero, not rounded. If the value is out of range, the behavior is undefined.
Integer → `float`	May lose precision. A 32-bit `int` has more significant digits than a `float` can hold.
`double` → `float`	Rounded to the nearest representable `float`; may become infinity.

Note the truncation rule for floating point to integer: `(int)3.99` is 3 and `(int)-3.99` is −3. To round, use `round`, `floor`, or `ceil` from `<math.h>` — and remember to link with `-lm`.

7 Comparing floating-point values

Chapter 4 established that most decimal fractions have no exact binary representation. The consequence for comparison is direct: **never use `==` on floating-point values** unless you are comparing against a value you know to be exactly representable, such as 0.0 assigned literally.

The naive fix, and why it is not enough

```
if (fabs(a - b) < 1e-9) { … }   /* absolute tolerance */
```

This works when the numbers are near 1. It fails at both extremes: for values around 10^{-15} every pair compares equal, and for values around 10^{15} no pair ever does, because the gap between adjacent representable doubles is already larger than 10^{-9}.

A tolerance that scales

```
#include <math.h>
#include <float.h>
#include <stdbool.h>

static bool nearly_equal(double a, double b, double relative)
{
    double diff = fabs(a - b);
    if (diff <= DBL_EPSILON) {      /* handles a == b and both near zero */
        return true;
    }
    double scale = fmax(fabs(a), fabs(b));
    return diff <= relative * scale;
}
```

Choosing `relative` is a judgement about your problem, not a property of the language. For values derived from a handful of arithmetic steps, something around `1e-9` for `double` is reasonable. For a long accumulation, error grows and the tolerance must grow with it.

Accumulated error

```
double sum = 0.0;
for (int i = 0; i < 10; i++) {
    sum += 0.1;
}
/* sum is 0.99999999999999988898, not 1.0 */
```

Each addition rounds, and the errors do not cancel. Where exactness matters — money, above all — do not use floating point. Store amounts as integer minor units (kuruş, cents) and format for display.

NaN and infinity

```
#include <math.h>

double x = 0.0 / 0.0;    /* NaN      */
double y = 1.0 / 0.0;    /* infinity */

isnan(x)      /* true  */
isinf(y)      /* true  */
isfinite(y)   /* false */
(x == x)      /* false — NaN is not equal to itself */
```

That last line is the defining property of `NaN` and a legitimate test for it, though `isnan` says what you mean. It also means a `NaN` quietly poisons every comparison: `a < b`, `a > b`, and `a == b` are all false when either is `NaN`, so a sort can behave bizarrely if one slips into the data.

8 Worked example: two bugs that compile cleanly

Save as `convert.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include <limits.h>

static bool nearly_equal(double a, double b, double relative)
{
    double diff = fabs(a - b);
    if (diff <= DBL_EPSILON) {
        return true;
    }
    double scale = fmax(fabs(a), fabs(b));
    return diff <= relative * scale;
}

int main(void)
```

```
{
    puts("== bug 1: a comparison that is wrong but not incorrect ==");
    int      signed_value   = -1;
    unsigned unsigned_value = 1;
    printf("  -1 < 1u  evaluates to %d\n", signed_value < unsigned_value);
    printf("  because -1 converted to unsigned is %u\n",
           (unsigned)signed_value);
    printf("  fix: compare in a signed type -> %d\n",
           signed_value < (int)unsigned_value);

    puts("\n== the loop that would never end ==");
    const char *empty = "";
    size_t length = strlen(empty);
    printf("  strlen(\"\") = %zu\n", length);
    printf("  length - 1 = %zu   <-- wrapped\n", length - 1);
    puts("  so  for (i = 0; i < length - 1; i++)  would run ~1.8e19 times");
    puts("  fix: write  i + 1 < length  instead");

    puts("\n== bug 2: floating point equality ==");
    double a = 0.1, b = 0.2;
    printf("  0.1 + 0.2        = %.20f\n", a + b);
    printf("  == 0.3           -> %s\n", (a + b == 0.3) ? "true" : "false");
    printf("  nearly_equal     -> %s\n",
           nearly_equal(a + b, 0.3, 1e-9) ? "true" : "false");

    double sum = 0.0;
    for (int i = 0; i < 10; i++) {
        sum += 0.1;
    }
    printf("  0.1 added ten times = %.20f\n", sum);
    printf("  == 1.0           -> %s\n", (sum == 1.0) ? "true" : "false");
    printf("  nearly_equal     -> %s\n",
           nearly_equal(sum, 1.0, 1e-9) ? "true" : "false");

    puts("\n== promotion and narrowing ==");
    char small_a = 100, small_b = 100;
    printf("  100 + 100 as int  = %d\n", small_a + small_b);
    printf("  stored in a char  = %d   <-- did not fit\n",
           (char)(small_a + small_b));
```

```
    printf("\n  (int)3.99  = %d      (truncates, never rounds)\n", (int)3.99);
    printf("  (int)-3.99 = %d\n", (int)-3.99);
    printf("  lround(3.99) = %ld\n", lround(3.99));

    puts("\n== the average, done wrong and right ==");
    int total = 7, count = 2;
    printf("  total / count             = %d\n", total / count);
    printf("  (double)(total / count)   = %g   <-- cast too late\n",
           (double)(total / count));
    printf("  (double)total / count     = %g   <-- correct\n",
           (double)total / count);

    puts("\n== NaN and infinity ==");
    double nan_value = 0.0 / 0.0;
    double inf_value = 1.0 / 0.0;
    printf("  0.0/0.0 = %f, isnan -> %d, equals itself -> %d\n",
           nan_value, isnan(nan_value), nan_value == nan_value);
    printf("  1.0/0.0 = %f, isinf -> %d\n", inf_value, isinf(inf_value));

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o convert convert.c -lm
./convert
```

The `-lm` is required: `<math.h>` declares `fmax` and `lround`, but their implementations live in the math library, which the linker does not search unless told. Omit it and you get the `undefined reference` error from chapter 1.

Read the warnings first

```
convert.c:24:44: warning: comparison of integer expressions of different
                 signedness: 'int' and 'unsigned int' [-Wsign-compare]
```

That one warning is the entire subject of section 4, delivered by the compiler for free. A project that compiles with warnings switched on and then ignores them has paid the cost of the tooling without collecting the benefit.

9 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Comparing `int` with `size_t`	Negative values become huge; comparison inverts	Keep one signedness per expression; heed `-Wsign-compare`.
`i < length - 1` when `length` is 0	Wraps to `SIZE_MAX`; loop never ends	Write `i + 1 < length`.
`(double)(a / b)`	Integer division already truncated	`(double)a / b`.
`if (x == 0.1)`	Almost always false	Compare with a scaled tolerance.
Adding a cast to silence a warning	The bug stays, the alarm is gone	Understand the warning; cast only when you can state why.
Using `double` for money	Accumulated rounding error	Store integer minor units.
Forgetting `-lm`	`undefined reference to 'fabs'`	Link the math library.
Assuming `(int)x` rounds	3.99 becomes 3	Use `lround`, `floor`, or `ceil`.

10 Check yourself

Why is `-1 < 1u` false?

Both operands have the rank of `int` and one is unsigned, so the usual arithmetic conversions convert the signed operand to unsigned. In two's complement −1 becomes 4 294 967 295, which is not less than 1. The comparison is behaving exactly as specified; the defect is mixing signedness in the expression.

What is the difference between `(double)total / count` and `(double)(total / count)`?

The first converts `total` before dividing, so the division is performed in `double` and keeps the fraction. The second divides two integers — truncating immediately — and then widens the already-wrong result. Only the first computes an average correctly.

Is `char c = a + b;` with `a` and `b` both 100 an error?

Not a compile error. The operands are promoted to `int`, the sum 200 is computed correctly, and then it is converted back to `char` on assignment. If plain `char` is signed and 8 bits, 200 does not fit and the result is implementation-defined. The arithmetic was never the problem; the storage was.

Why is an absolute tolerance such as `fabs(a-b) < 1e-9` not a general solution?

Because the spacing between representable doubles grows with magnitude. Near 10^{15} that spacing already exceeds 10^{-9}, so no two distinct values ever compare equal; near 10^{-15} almost every pair does. A tolerance must scale with the size of the values being compared.

A sort of your `double` array produces nonsense. One element is `NaN`. Why does that break it?

Every comparison involving `NaN` is false — `<`, `>`, and `==` alike. A comparison function therefore reports that the `NaN` is neither before nor after nor equal to anything, which violates the ordering the sorting algorithm assumes and can corrupt the whole result. Filter or reject `NaN` before sorting.

11 Where this leads

You can now predict the type and value of any expression. Chapter 8 uses that directly: `printf` and `scanf` take a format string whose specifiers must match the argument types exactly, and the conversions from this chapter are precisely what decides whether they do. It is also where you meet the first function that can corrupt memory if you get an argument wrong.

BASIC · CHAPTER 8

Formatted Input and Output

Output is straightforward once you know the specifiers. Input is not. `scanf` *is the function most likely to make a beginner's program hang, loop forever, or silently read nothing — and understanding why teaches you more about C than the function itself is worth.*

BY THE END OF THIS CHAPTER YOU CAN

- Format output precisely with width, precision, and flags.
- Match every specifier to its argument type, and explain why a mismatch is undefined behavior.
- Explain what the input buffer is and why `scanf` leaves things in it.
- Check the return value of `scanf` and recover from malformed input.
- Write an input loop that cannot be made to hang or spin by any input at all.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 `printf` in detail

A conversion specification has up to five parts, in this order:

```
%[flags][width][.precision][length]conversion
```

Conversions

SPECIFIER	ARGUMENT TYPE	PRINTS
`%d` `%i`	`int`	signed decimal
`%u`	`unsigned int`	unsigned decimal
`%x` `%X`	`unsigned int`	hexadecimal, lower or upper case
`%o`	`unsigned int`	octal
`%f`	`double`	fixed notation, 6 decimals by default
`%e` `%E`	`double`	scientific notation
`%g` `%G`	`double`	whichever of `%f` / `%e` is shorter
`%c`	`int` (promoted char)	one character
`%s`	`char *`	characters up to the terminating `'\0'`
`%p`	`void *`	an address
`%%`	—	a literal `%`

Length modifiers

These adjust the expected argument size and are not optional when the type is not `int` or `double`.

WRITE	FOR
`%ld`	`long`
`%lld`	`long long`
`%zu`	`size_t`
`%hd`	`short`
`%Lf`	`long double`

There is no modifier for `float`, because a `float` argument is automatically promoted to `double` when passed to a variadic function. `%f` is correct for both.

Width, precision, and flags

```
printf("[%5d]\n",    42);       /* [   42]  width 5, right aligned */
printf("[%-5d]\n",   42);       /* [42   ]  left aligned           */
printf("[%05d]\n",   42);       /* [00042]  zero padded            */
printf("[%+d]\n",    42);       /* [+42]    always show the sign   */
printf("[%8.3f]\n",  3.14159);  /* [   3.142]                      */
printf("[%.2f]\n",   3.14159);  /* [3.14]   two decimals           */
printf("[%.3s]\n",   "abcdef"); /* [abc]    first three characters */
printf("[%*d]\n", 6, 42);       /* [    42] width from an argument */
```

For `%f` precision means digits after the point; for `%g` it means total significant digits; for `%s` it means a maximum number of characters. Columns line up when you combine width with left alignment, which is how the type table in chapter 5 was printed.

> **A mismatched specifier is undefined behavior, not a formatting quirk.** `printf("%d", 3.14)` makes `printf` read an `int`-sized piece of a `double`'s bits. `printf("%s", 42)` makes it treat 42 as an address and read memory there — usually a crash. `printf` cannot check: it receives no type information at run time, only the promise your format string made. Chapter 30 explains the variadic mechanism that makes this unavoidable.

The compiler does check, though, when it can see the format string as a literal. This is one of the most valuable warnings `-Wall` provides:

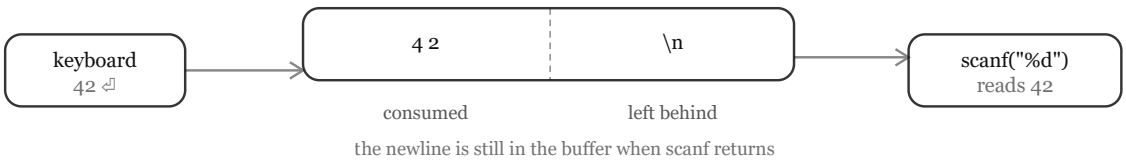
```
warning: format '%d' expects argument of type 'int',
         but argument 2 has type 'double' [-Wformat=]
```

2 The input buffer

Before `scanf` makes sense, the buffer does.

When you type at a terminal, your keystrokes do not reach the program one at a time. The operating system collects a whole line and hands it over only when you press Enter. That line — *including the newline character* — sits in a buffer belonging to `stdin`. Input functions consume characters from that buffer; anything they do not consume stays there for the next call.

Almost every `scanf` problem is a consequence of that last sentence.



the newline is still in the buffer when scanf returns

`scanf("%d")` stops at the first character that cannot be part of a number — and the newline is one.

3 `scanf` and its three failure modes

`scanf` takes a format string and the *addresses* of the variables to fill:

```
int age;
scanf("%d", &age);      /* note the & */
```

The `&` is the address-of operator from chapter 13. For now, take it as required: `scanf` must be able to write into your variable, so it needs to know where the variable lives. Forgetting it is the single most common `scanf` bug, and it corrupts memory rather than failing cleanly.

It returns a count, and you must check it

`scanf` returns the number of items it successfully assigned, or `EOF` if the input ended first.

```
int age;
if (scanf("%d", &age) != 1) {
    fprintf(stderr, "that was not a number\n");
    return EXIT_FAILURE;
}
```

A program that ignores this return value has no idea whether its variables were filled. If the user types `abc`, `scanf` returns 0, `age` keeps whatever it had before — possibly garbage, per chapter 5 — and the program proceeds on nonsense.

Failure mode 1 — the leftover newline

```
int age;
char initial;

scanf("%d", &age);      /* consumes 42, leaves '\n'        */
scanf("%c", &initial);  /* reads '\n' instantly, no pause  */
```

The second prompt appears to be skipped. It was not: it read the newline the first call left behind. `%d`, `%f`, and `%s` all skip leading whitespace, but `%c` does not — it takes whatever is next, including a newline.

The conventional patch is a space in the format string, which means "skip any whitespace here":

```
scanf(" %c", &initial);   /* the leading space fixes it */
```

Failure mode 2 — the infinite loop

```
int n;
while (scanf("%d", &n) != 1) {
    puts("please enter a number");
}
```

Type `abc` and this loop runs forever. `scanf` encounters `a`, finds it cannot be part of an integer, returns 0 — and **leaves `abc` in the buffer**. The next iteration meets the same characters and fails identically, as fast as the processor can manage.

Recovery requires explicitly discarding the bad input:

```
static void discard_line(void)
{
    int c;
    while ((c = getchar()) != '\n' && c != EOF) {
        /* throw away the rest of the line */
    }
}
```

Failure mode 3 — `%s` and buffer overflow

```
char name[10];
scanf("%s", name);      /* no bound: 50 characters will be written */
```

This is the same class of defect as `gets`, a function so unsafe that C11 removed it. `%s` writes until it meets whitespace, with no regard for how large your array is. Always give a width, one less than the array size to leave room for the terminator:

```
char name[10];
scanf("%9s", name);     /* at most 9 characters plus '\0' */
```

> **What professionals actually do.** Most production C reads a whole line with `fgets` and then parses it with `strtol` or `sscanf`. That separates "get the input" from "interpret the input", which makes error handling straightforward and leaves nothing unpredictable in the buffer. Chapter 23 builds exactly that. Learn `scanf` because you will meet it in every textbook and much existing code — then prefer the alternative.

4 Character-at-a-time input and output

```
int c = getchar();      /* one character, or EOF */
putchar(c);             /* one character out     */
```

Note the type: `getchar` returns `int`, not `char`. It must be able to return every possible character *and* the distinct value `EOF`, which is negative. Storing the result in a `char` destroys that distinction and can make the end-of-input test fail — on a machine where plain `char` is unsigned, it can loop forever.

```
int c;                          /* correct */
while ((c = getchar()) != EOF) {
    putchar(c);
}
```

That loop copies standard input to standard output — the whole of the Unix `cat` command, and a useful shape to recognize. The extra parentheses around the assignment are required, because `=` binds looser than `!=` (chapter 6).

End of input from a terminal is Ctrl-D on Linux and macOS, Ctrl-Z then Enter on Windows.

5 Worked example: an input loop that cannot be broken

The goal: read an integer between 1 and 100. Accept correct input, reject everything else with a clear message, and never hang or spin regardless of what is typed — including no input at all.

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

/* Throw away the remainder of the current input line.
   Returns false if input ended while doing so. */
static bool discard_line(void)
{
    int c;
    while ((c = getchar()) != '\n') {
        if (c == EOF) {
            return false;
        }
    }
    return true;
}

/* Read one integer in [low, high].
   Returns false only if input ends; otherwise keeps asking. */
static bool read_int_in_range(const char *prompt, int low, int high, int *out)
{
    for (;;) {
        printf("%s (%d-%d): ", prompt, low, high);
        fflush(stdout);

        int value;
        int converted = scanf("%d", &value);

        if (converted == EOF) {
            putchar('\n');
            return false;                    /* Ctrl-D, or piped input ran out */
        }
```

```
        if (converted == 0) {
            fprintf(stderr, "  not a number — try again\n");
            if (!discard_line()) {           /* essential: remove the bad text */
                return false;
            }
            continue;
        }

        if (!discard_line()) {               /* remove the trailing newline
                                                and anything after the number */
            return false;
        }

        if (value < low || value > high) {
            fprintf(stderr, "  %d is out of range — try again\n", value);
            continue;
        }

        *out = value;
        return true;
    }
}

int main(void)
{
    int score;
    if (!read_int_in_range("Enter a score", 1, 100, &score)) {
        fprintf(stderr, "no input; giving up\n");
        return EXIT_FAILURE;
    }

    printf("\n%-12s %s\n", "score", "bar");
    printf("%-12d ", score);
    for (int i = 0; i < score / 5; i++) {
        putchar('#');
    }
```

```
    printf("\n%-12s %6.1f%%\n", "as percent", (double)score);

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o prompt prompt.c
./prompt
```

Test it properly

Try each of these. A correct program survives all six.

INPUT	EXPECTED
`42`	accepted
`abc`	rejected once, prompts again — does not spin
`999`	rejected as out of range
`42abc`	accepts 42, discards the rest of the line
Enter on its own	keeps waiting for a number (%d skips the blank line)
Ctrl-D	exits cleanly with a failure status

And from a script, which is how real programs are usually driven:

```
echo "42" | ./prompt          ; echo "exit status $?"
echo "abc" | ./prompt         ; echo "exit status $?"
printf "" | ./prompt          ; echo "exit status $?"
```

The third case — empty input — is the one hand-testing always misses. `scanf` returns `EOF` immediately, and a loop that only checks for 0 would never terminate.

Three details worth noting

`fflush(stdout)` **after the prompt.** Standard output to a terminal is line-buffered, so a prompt with no trailing newline may not appear before the program blocks waiting for input. Flushing forces it out. Chapter 26 covers buffering properly.

Errors go to `stderr`**, results to** `stdout`**.** That separation lets a user redirect the data without losing the diagnostics — `./prompt > results.txt` still shows the error messages. Chapter 31 makes this a design rule.

Three distinct outcomes, three distinct responses. `EOF`, a conversion failure, and a valid-but-out-of-range value are different situations and the code treats them differently. Collapsing them into one "bad input" branch is how programs end up hanging.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`scanf("%d", age)` without `&`	Memory corruption or a crash	Pass the address. `-Wall` catches this one.
Ignoring `scanf`'s return value	Program proceeds on an unassigned variable	Compare it with the number of items you asked for.
Retrying `scanf` without discarding input	Infinite loop at full speed	Consume the rest of the line before retrying.
`scanf("%c", &ch)` after `%d`	Reads the leftover newline	`scanf(" %c", &ch)` — note the space.
`scanf("%s", buf)` with no width	Buffer overflow	`%9s` for a 10-byte array.
`char c = getchar();`	`EOF` becomes indistinguishable from a character	Declare `int c`.
Prompt not appearing before input	Program looks frozen	`fflush(stdout)`, or end the prompt with `\n`.
Checking only for 0, never `EOF`	Infinite loop on empty or piped input	Handle `EOF` as its own case.

7 Check yourself

Why does the second prompt seem to be skipped when a `%c` read follows a `%d` read?

Because `%d` stops at the newline and leaves it in the input buffer. `%c` does not skip whitespace, so it immediately consumes that leftover newline and returns without waiting. Writing `" %c"` — with a leading space — tells `scanf` to skip whitespace first.

Why does `while (scanf("%d", &n) != 1) puts("try again");` spin forever on letters?

`scanf` stops at the first character it cannot convert and leaves that character in the buffer. Each iteration sees the same offending text, fails identically, and loops. You must explicitly consume the rest of the line before retrying.

Why must `getchar`'s result be stored in an `int`?

It returns every possible character value plus a distinct `EOF`, which is negative. A `char` cannot represent all of those distinctly. Where plain `char` is unsigned, `EOF` becomes a positive value and the end-of-input test never succeeds.

Why can `printf` not detect that you passed a `double` where `%d` was written?

Because it is a variadic function: at run time it receives no type information about the extra arguments, only the format string's claim about them. It reads bytes according to that claim. The compiler can check when the format string is a literal — which is what `-Wformat` does — but the function itself cannot. Chapter 30 builds a variadic function and shows why.

Your interactive program works, but breaks when someone pipes input to it. What did you probably forget?

To handle `EOF`. Interactively the user keeps typing until the program is satisfied; piped input simply runs out, and `scanf` returns `EOF` rather than 0. Code that only distinguishes success from conversion failure loops forever once input ends. Always test with `printf "" | ./prog`.

8 Where this leads

You can now get data in and out of a program and defend it against bad input. Chapters 9 and 10 give you the control structures to act on that data — conditionals and loops — which turns these fragments into programs that decide and repeat. The input-validation pattern built here returns in chapter 23, rewritten with `fgets` and `strtol`, and again in chapter 48, where a fuzzer attacks it.

BASIC · CHAPTER 9

Conditional Statements

A program that cannot choose is a calculator. Conditionals are how a program decides — and in C they carry a handful of sharp edges that have each caused real, expensive failures.

BY THE END OF THIS CHAPTER YOU CAN

- Write `if`, `else if`, and `else` chains that are correct and readable.
- Explain what C actually treats as true, and why `if (x)` works.
- Use the conditional operator where it clarifies and avoid it where it does not.
- Write a `switch` correctly, including deliberate fall-through.
- Recognize the dangling `else`, the stray semicolon, and the missing `break` on sight.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 `if`, `else if`, `else`

```
if (temperature > 30) {
    puts("hot");
} else if (temperature > 20) {
    puts("warm");
} else if (temperature > 10) {
    puts("mild");
} else {
    puts("cold");
}
```

The conditions are tested in order and **at most one branch runs**. As soon as one test succeeds, the rest are skipped entirely. That ordering is what makes the chain above correct with such simple tests: by the time the second condition is reached, the program already knows the temperature is not above 30.

Write the same chain in the wrong order and it silently produces nonsense:

```
if (temperature > 10) {
    puts("mild");          /* 35 degrees reports "mild" */
} else if (temperature > 30) {
    puts("hot");           /* unreachable */
}
```

The compiler will not warn. Ordering the tests from most specific to least specific is a habit worth forming now.

There is no `elseif` keyword

`else if` is not special syntax. It is an `if` statement that happens to be the single statement belonging to an `else`. C allows any statement there, and an `if` is a statement. Writing the braces out makes this visible:

```
else {
    if (temperature > 20) { … }
}
```

Nobody writes it that way, but knowing that is what it means explains the dangling-`else` problem in section 4.

2 What C considers true

There is no boolean requirement. A condition is any expression that yields a number, and the rule is the one from chapter 6: **zero is false, everything else is true**.

```
if (count)        /* true when count is not zero        */
if (!count)       /* true when count is zero            */
if (pointer)      /* true when pointer is not NULL      */
if (-1)           /* true — non-zero, even though negative */
```

These idioms are pervasive in real C, so you must be able to read them. Whether to write them is a style question. `if (count != 0)` states the intent; `if (count)` is shorter. For pointers, `if (p)` is nearly universal and worth adopting; for counts, the explicit comparison usually reads better.

One place the shorthand is genuinely wrong: comparing floating-point values. `if (x)` on a `double` asks whether it is exactly zero, which chapter 7 warned against.

> **`=` versus `==`, again.** `if (status = 0)` assigns zero and then tests zero — the branch never runs, and `status` has been destroyed. It is legal C, so only `-Wall` saves you. This mistake was used in a 2003 attempt to plant a backdoor in the Linux kernel; it is not a beginner-only problem.

3 The conditional operator

C's only ternary operator chooses between two *expressions*:

```
int larger = (a > b) ? a : b;


printf("%d item%s\n", n, (n == 1) ? "" : "s");
```

It is an expression, so it can appear where a statement cannot — inside a function call, in an initializer, in a `return`. That is its real value. Exactly one of the two branches is evaluated, like `&&` and `||`.

It stops being helpful when nested:

```
/* Do not do this */
char *grade = (s >= 90) ? "A" : (s >= 80) ? "B" : (s >= 70) ? "C" : "F";
```

An `if`/`else if` chain expresses the same logic in a form that can be read, debugged line by line, and extended. Use the conditional operator for a single either-or choice; reach for `if` beyond that.

4 Three classic traps

The stray semicolon

```
if (x > 0);
{
    puts("positive");     /* runs unconditionally */
}
```

The semicolon is an empty statement, and it is the entire body of the `if`. The block that follows is just a block, executed always. `-Wextra` emits *suggest braces around empty body*, which is another reason that flag is not optional.

The dangling `else`

```
if (a > 0)
    if (b > 0)
        puts("both positive");
else
    puts("a is not positive");   /* WRONG: binds to the inner if */
```

An `else` attaches to the *nearest unmatched* `if`, regardless of how you indent. Here it belongs to `if (b > 0)`, so the message prints when `a` is positive and `b` is not — the opposite of what the layout suggests. Braces remove the ambiguity permanently:

```
if (a > 0) {
    if (b > 0) {
        puts("both positive");
    }
} else {
    puts("a is not positive");
}
```

Omitted braces

```
if (error)
    log_error();
    return -1;        /* always runs — not part of the if */
```

Indentation is not syntax in C. The `return` executes unconditionally. This is the shape of Apple's "goto fail" bug, and the reason many style guides — MISRA C and the CERT C rules among them — now require braces without exception.

> **Use braces always.** Even for a single statement. It costs two characters, it survives every future edit that adds a second statement, and it eliminates two of the three traps above outright. Let `clang-format` from chapter 3 enforce it.

5 `switch`

`switch` compares one integer expression against a list of constant values:

```
switch (command) {
case 'a':
    add_item();
    break;
case 'd':
    delete_item();
    break;
case 'q':
    quit();
    break;
default:
    fprintf(stderr, "unknown command '%c'\n", command);
    break;
}
```

The rules

- The controlling expression must be an integer type — `int`, `char`, an enumeration. Not a `double`, not a string.
- Each `case` label must be a *constant expression*, known at compile time. `case x:` where `x` is a variable does not compile.
- Labels must be distinct.
- `default` is optional and may appear anywhere, though the end is conventional.

Fall-through

This is the part that surprises everyone. A `case` label is a *jump target*, not a block. Execution enters at the matching label and then continues straight through every following label until it hits a `break` or the closing brace.

```
switch (grade) {
case 'A':
    puts("excellent");
    /* no break — falls through */
case 'B':
    puts("good");
    break;
}
```

An `'A'` prints both lines. Forgetting a `break` is one of the most common C bugs, and `-Wimplicit-fallthrough` exists specifically to catch it.

Deliberate fall-through is genuinely useful for grouping labels:

```
switch (c) {
case 'a': case 'e': case 'i': case 'o': case 'u':
    vowels++;
    break;
default:
    consonants++;
    break;
}
```

When you fall through on purpose across statements, say so. C23 provides an attribute; a comment has long been the convention and GCC recognizes it:

```
case 'A':
    bonus += 10;
    [[fallthrough]];        /* C23 */
case 'B':
    passed = true;
    break;
```

Declaring a variable inside a `switch`

```
switch (n) {
case 1:
    int x = 5;        /* error before C23: a label cannot precede a declaration */
    break;
}
```

Wrap the case body in braces, which also scopes the variable properly:

```
case 1: {
    int x = 5;
    …
    break;
}
```

6 `switch` or `if`?

USE `SWITCH` WHEN	USE `IF` WHEN
Testing one value against several constants	Testing ranges, or several different variables
The value is an integer or enumeration	The value is a string, a float, or a pointer
The cases are a closed, enumerable set	The conditions are arbitrary expressions
You want the compiler to warn about an unhandled `enum` value	Order of evaluation matters

That last row is a real advantage. With an enumeration and no `default`, `-Wswitch` warns when you add a new enumerator and forget to handle it — a small piece of static checking the language rarely offers. Chapter 22 relies on it.

The performance argument you may read — that `switch` compiles to a jump table and is therefore faster — is real but almost never decisive. The compiler builds a jump table only when the labels are dense, and for a handful of cases the difference is unmeasurable. Choose on clarity.

7 Worked example: the same classifier, twice

A program that reports a letter grade and a short description. Written first with `if`, then with `switch`, so the trade-off is concrete rather than theoretical.

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

/* Version 1: ranges. An if chain is the natural fit. */
static char grade_from_score(int score)
{
```

```
    if (score >= 90) {
        return 'A';
    } else if (score >= 80) {
        return 'B';
    } else if (score >= 70) {
        return 'C';
    } else if (score >= 60) {
        return 'D';
    } else {
        return 'F';
    }
}

/* Version 2: a fixed set of discrete values. switch is the natural fit. */
static const char *describe(char grade)
{
    switch (grade) {
    case 'A':
        return "excellent";
    case 'B':
        return "good";
    case 'C':
        return "satisfactory";
    case 'D':
        return "marginal";
    case 'F':
        return "failing";
    default:
        return "unknown";
    }
}

/* Deliberate fall-through, grouping labels that share one action. */
static bool is_passing(char grade)
{
    switch (grade) {
    case 'A':
    case 'B':
    case 'C':
```

```
    case 'D':
        return true;
    default:
        return false;
    }
}

int main(void)
{
    const int scores[] = { 95, 83, 71, 64, 42, 100, 0 };
    const size_t count = sizeof scores / sizeof scores[0];

    printf("%-7s %-7s %-15s %s\n", "score", "grade", "description", "passing");
    for (size_t i = 0; i < count; i++) {
        char g = grade_from_score(scores[i]);
        printf("%-7d %-7c %-15s %s\n",
               scores[i], g, describe(g), is_passing(g) ? "yes" : "no");
    }

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o grades grades.c
./grades
```

Why each construct fits where it does

`grade_from_score` tests *ranges*. A `switch` cannot express `score >= 90`; you would need a separate label for every score from 90 to 100, and the same for every other grade. And the chain depends on ordering — each test is written assuming the earlier ones already failed, which is why `score >= 80` does not need an upper bound.

`describe` maps a small closed set of values to results. Every branch does the same kind of thing with a different constant, which is exactly the shape `switch` was designed for. Written as an `if` chain it would repeat `grade ==` five times.

`is_passing` uses fall-through the way it is meant to be used: four labels sharing one action, no statements between them, no comment needed because the intent is unmistakable.

Now introduce the bugs and watch the compiler

Make these four changes one at a time and rebuild.

```
/* 1. drop a break */
case 'A':
    return "excellent";
case 'B':
    puts("good");        /* change return to puts, remove break */
case 'C':
    puts("satisfactory");
```

```
warning: this statement may fall through [-Wimplicit-fallthrough=]
```

```
/* 2. stray semicolon */
if (score >= 90);
    return 'A';
```

```
warning: suggest braces around empty body in an 'if' statement [-Wempty-body]
```

```
/* 3. assignment in a condition */
if (score = 90) { … }
```

```
warning: suggest parentheses around assignment used as truth value [-Wparentheses]
```

```
/* 4. reordered chain — the silent one */
if (score >= 60)      return 'D';
else if (score >= 90) return 'A';   /* unreachable */
```

```
(no warning at all)
```

Three of the four are caught for free. The fourth — wrong ordering — is perfectly valid C and produces wrong answers silently. That is the category of bug tests exist for, and it is why chapter 35 spends a session on testing rather than trusting the compiler to find everything.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`if (x = 5)`	Assigns, then always true	Use `==`; never ignore `-Wparentheses`.
`if (x > 0);`	Empty body; block runs unconditionally	Delete the semicolon; heed `-Wempty-body`.
Omitting braces on a single statement	A later edit silently falls outside the `if`	Always use braces.
Dangling `else`	`else` binds to the inner `if`	Braces on every nested `if`.
Missing `break` in a `switch`	Falls through to the next case	`break` in every case; mark intentional fall-through.
`switch` on a `double` or a string	Compile error	Use `if`, or `strcmp` for strings — chapter 14.
Ordering an `else if` chain loosest-first	Later branches unreachable; no warning	Order most specific first, and test the boundaries.
`if (0.1 + 0.2 == 0.3)`	False	Compare with a tolerance — chapter 7.

9 Check yourself

What does `if (count)` test?

Whether `count` is non-zero. C has no boolean requirement for conditions: any numeric expression works, zero is false and every other value — including negatives — is true. It is equivalent to `if (count != 0)`.

Which `if` does an `else` belong to when the nesting is ambiguous?

The nearest preceding `if` that does not already have an `else`, regardless of indentation. Indentation has no syntactic meaning in C, so code that looks correct can bind the opposite way. Braces on every nested `if` remove the question entirely.

Why does a `case` without `break` continue into the next one?

Because `case` is a label — a jump target — not a block. `switch` transfers control to the matching label and then execution proceeds normally through whatever follows, including subsequent labels, until a `break` or the closing brace. This allows label grouping, at the cost of making the missing `break` an easy mistake.

When does `switch` give you a safety guarantee that `if` does not?

When switching on an enumeration with no `default` label: `-Wswitch` then warns about any enumerator you failed to handle. Add a new value to the enum later and the compiler points at every switch that needs updating. An `if` chain offers no equivalent check.

Your grade chain returns 'D' for a score of 95 and the compiler said nothing. What happened?

The tests are ordered from loosest to strictest, so `score >= 60` matches first and the later branches are unreachable. This is valid C — the compiler has no way to know the intended semantics — so nothing is reported. Order conditions most specific first, and write tests that check the boundary values.

10 Where this leads

A program can now choose. Chapter 10 gives it the ability to repeat, which is where loop invariants, off-by-one errors, and the two uses of `goto` that survive in modern C appear. Together, conditionals and loops are all the control flow C has — everything after chapter 10 is about organizing data rather than organizing execution.

BASIC · CHAPTER 10

Loops and Flow Control

Three loop forms, and they are interchangeable — any one can express what the others do. What is not interchangeable is the discipline of knowing why a loop is correct, which is the difference between code that works on your test case and code that works.

BY THE END OF THIS CHAPTER YOU CAN

- Choose between `for`, `while`, and `do-while` for a given task.
- State a loop's invariant and use it to argue the loop is correct.
- Reason about half-open ranges and stop making off-by-one errors.
- Use `break` and `continue`, and escape a nested loop cleanly.
- Say when `goto` is defensible in C, and why chapter 27 comes back to it.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The three forms

`while` — test first

```
while (condition) {
    body
}
```

Test, then run the body, then test again. If the condition is false initially, the body never runs. Use it when the number of iterations is not known in advance.

`for` — the counted loop

```
for (initialization; condition; update) {
    body
}
```

Equivalent to (except that `continue` in a `for` still runs the update):

```
initialization;
while (condition) {
    body
    update;
}
```

The value of `for` is that all three pieces of loop bookkeeping sit on one line where a reader can check them together. Use it whenever there is a counter.

All three clauses are optional. `for (;;)` is the conventional infinite loop — an empty condition is treated as true.

`do-while` — test after

```
do {
    body
} while (condition);      /* note the semicolon */
```

The body always runs at least once. This is the right shape for "ask, then check whether the answer was acceptable" — the input loop from chapter 8 is naturally a `do-while`. It is the least used of the three, and the trailing semicolon is easy to forget.

Declaring the counter inside

```
for (int i = 0; i < n; i++) { … }
/* i does not exist here */
```

A C99 feature, and the right default: the counter is scoped to the loop, so it cannot be accidentally reused afterwards and the name is free for the next loop. Chapter 2 showed this failing under `-std=c89`.

2 Half-open ranges and off-by-one errors

The canonical C loop counts from 0 while the index is *less than* the count:

```
for (size_t i = 0; i < n; i++) { … }
```

This is a **half-open range**, written [0, n): it includes 0 and excludes n. It is not arbitrary convention — it has three properties that eliminate whole categories of mistake.

PROPERTY	WHY IT HELPS
The number of iterations is `n - 0 = n`	The count is visible without arithmetic
An empty range is `i < 0`, which is simply false	Zero elements needs no special case
Adjacent ranges join without gaps or overlap	[0,k) and [k,n) cover [0,n) exactly

Compare with the closed alternative:

```
for (size_t i = 0; i <= n - 1; i++) { … }   /* same idea, three defects */
```

It needs the reader to compute `n - 1`; it breaks when `n` is 0, because `0 - 1` on an unsigned type wraps to an enormous value (chapter 7); and two adjacent ranges written this way are easy to overlap by one.

> **The off-by-one checklist.** When a loop is wrong by one, ask three questions in this order. Does the first iteration handle the first element? Does the last iteration handle the last element? Does the loop do the right thing when there are zero elements? Almost every off-by-one error is one of those three, and checking them takes ten seconds.

3 Loop invariants

An invariant is a statement that is true before the loop starts, true after every iteration, and therefore true when the loop ends. Combine it with the exit condition and you have an argument that the loop is correct — not a test that it happened to work.

```
int maximum = values[0];
/* invariant: maximum equals the largest of values[0..i-1] */
for (size_t i = 1; i < n; i++) {
    if (values[i] > maximum) {
        maximum = values[i];
    }
}
/* loop ended with i == n, so by the invariant:
   maximum is the largest of values[0..n-1] — the whole array */
```

Writing the invariant as a comment is a genuine technique, not a classroom exercise. It forces the two questions that catch real bugs: is it true before the first iteration — here, that `maximum`

is the largest of a one-element range, which is why the initialization reads `values[0]` and the loop starts at 1 — and does the body preserve it?

It also exposes an edge case immediately. The invariant assumes `values[0]` exists, so this loop is wrong for `n == 0`. The fix has to be a guard before the loop, and the invariant is what told you so.

Termination

An invariant shows the loop computes the right thing *if* it stops. Termination is a separate obligation: identify a quantity that strictly decreases and cannot go below a bound. In a counting loop it is `n - i`. In a `while` loop driven by input it is the number of remaining characters. If you cannot name that quantity, you have not shown the loop terminates.

4 Nested loops

```
for (size_t row = 0; row < rows; row++) {
    for (size_t col = 0; col < cols; col++) {
        printf("%4zu", row * col);
    }
    putchar('\n');
}
```

The inner loop runs completely for each iteration of the outer one, so the body executes `rows × cols` times. That multiplication is worth keeping in mind: a nested loop over a thousand-element array is a million operations, and three levels is a billion. Chapter 33 gives this the name $O(n^2)$.

Give nested counters real names. `i` and `j` are fine one level deep; at two levels `row` and `col` prevent the single most common nesting bug, which is using the wrong index inside the inner body.

5 `break`, `continue`, and `goto`

`break`

Leaves the innermost enclosing loop or `switch` immediately.

```
size_t found = n;                /* n means "not found" */
for (size_t i = 0; i < n; i++) {
    if (values[i] == target) {
        found = i;
        break;
    }
}
```

`continue`

Skips the rest of the body and goes to the next iteration. In a `for` loop the update clause still runs; in a `while` loop it does not, which is a genuine hazard:

```
size_t i = 0;
while (i < n) {
    if (skip(values[i])) {
        continue;                /* i never increments — infinite loop */
    }
    process(values[i]);
    i++;
}
```

This is a strong argument for preferring `for` whenever there is a counter: the update lives in the loop header where `continue` cannot bypass it.

Escaping a nested loop

`break` leaves only one level. The three usual options:

```
/* 1. A flag — works, but clutters both conditions */
bool done = false;
for (size_t r = 0; r < rows && !done; r++) {
    for (size_t c = 0; c < cols; c++) {
        if (grid[r][c] == target) { done = true; break; }
    }
}

/* 2. A function — usually the cleanest; return leaves everything */
static bool contains(int grid[][COLS], size_t rows, int target)
```

```
{
    for (size_t r = 0; r < rows; r++) {
        for (size_t c = 0; c < COLS; c++) {
            if (grid[r][c] == target) {
                return true;
            }
        }
    }
    return false;
}

/* 3. goto — the one case where most C style guides allow it for loops */
for (size_t r = 0; r < rows; r++) {
    for (size_t c = 0; c < cols; c++) {
        if (grid[r][c] == target) {
            goto found;
        }
    }
}
puts("not found");
goto done;
found:
puts("found");
done:
;
```

Option 2 is the right default. If the inner work deserves a name, giving it one solves the control-flow problem as a side effect.

About `goto`

`goto` jumps to a label within the same function. Its reputation comes from an era when it was the primary control structure and programs became untraceable. With `if`, loops, and functions available, that use is obsolete.

Two uses survive in modern C. Breaking out of nested loops, as above. And — far more important — unwinding resources on an error path, which is the dominant error-handling idiom in the Linux kernel and most serious C libraries:

```
FILE *in = fopen(path, "r");
if (in == NULL)          goto fail;
buf = malloc(size);
if (buf == NULL)         goto close_in;
…
close_in:
    fclose(in);
fail:
    return NULL;
```

That pattern is chapter 27's subject, once you have files and dynamic memory to unwind. Mentioning it now is deliberate: you should not leave this chapter thinking `goto` is simply forbidden.

6 Worked example: a sieve and a table

Two classic loops, each with its invariant stated and its boundaries tested.

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

#define LIMIT 50
#define TABLE 9

/* Sieve of Eratosthenes over [2, LIMIT].
   Invariant: after processing p, every composite with a factor <= p
   has been marked false. */
static void print_primes(void)
{
    bool is_prime[LIMIT + 1];

    for (size_t i = 0; i <= LIMIT; i++) {
        is_prime[i] = true;
    }
    is_prime[0] = false;
    is_prime[1] = false;
```

```
    for (size_t p = 2; p * p <= LIMIT; p++) {
        if (!is_prime[p]) {
            continue;                /* already crossed out */
        }
        /* Start at p*p: smaller multiples of p have a smaller
           factor and were crossed out in an earlier pass. */
        for (size_t multiple = p * p; multiple <= LIMIT; multiple += p) {
            is_prime[multiple] = false;
        }
    }

    printf("primes up to %d:\n ", LIMIT);
    size_t printed = 0;
    for (size_t i = 2; i <= LIMIT; i++) {
        if (is_prime[i]) {
            printf("%4zu", i);
            if (++printed % 10 == 0) {
                printf("\n ");
            }
        }
    }
    putchar('\n');
}

/* A multiplication table with aligned headers. */
static void print_table(void)
{
    printf("\nmultiplication table\n");

    printf("    ");
    for (size_t col = 1; col <= TABLE; col++) {
        printf("%4zu", col);
    }
    printf("\n    ");
    for (size_t col = 1; col <= TABLE; col++) {
        printf("----");
    }
    putchar('\n');
```

```
    for (size_t row = 1; row <= TABLE; row++) {
        printf("%2zu |", row);
        for (size_t col = 1; col <= TABLE; col++) {
            printf("%4zu", row * col);
        }
        putchar('\n');
    }
}

/* Linear search returning the count as "not found",
   so the caller needs no separate flag. */
static size_t find(const int values[], size_t n, int target)
{
    for (size_t i = 0; i < n; i++) {
        if (values[i] == target) {
            return i;
        }
    }
    return n;
}

int main(void)
{
    print_primes();
    print_table();

    const int data[] = { 4, 8, 15, 16, 23, 42 };
    const size_t n = sizeof data / sizeof data[0];

    printf("\nsearching %zu values\n", n);
    const int targets[] = { 15, 99 };
    for (size_t t = 0; t < 2; t++) {
        size_t at = find(data, n, targets[t]);
        if (at == n) {
            printf("  %d: not found\n", targets[t]);
        } else {
            printf("  %d: at index %zu\n", targets[t], at);
        }
    }
```

```
    /* The boundary case every search must survive. */
    printf("  empty array: %s\n",
           find(data, 0, 15) == 0 ? "correctly reports not found" : "BUG");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o loops loops.c
./loops
```

Three decisions worth examining

Why the sieve's inner loop starts at `p * p`. Any multiple of `p` smaller than `p²` has a factor smaller than `p`, so an earlier pass already crossed it out. The comment records the reasoning, because the code alone looks like an arbitrary optimization.

Why the outer loop stops at `p * p <= LIMIT`. Once `p` exceeds √LIMIT, `p²` is past the end and the inner loop would not execute at all. The invariant tells you the sieve is already complete at that point.

Why `find` returns `n` for "not found". There is no index `n` in a half-open range [0, n), so it is an unambiguous sentinel that costs no extra variable and cannot be confused with a real result. Returning −1 is the other convention, but it forces a signed return type and therefore a signed/unsigned comparison at the call site — chapter 7's trap.

Test the boundaries, not the middle

Change `LIMIT` to 1, then 2, then 3, then 4, and rebuild each time.

LIMIT	EXPECTED
1	no primes; the final loop must not run
2	just 2; the sieve's outer loop never executes, since 2·2 > 2
3	2 and 3
4	2 and 3; the first crossing-out happens

A loop that is right in the middle and wrong at the edges is the normal failure mode. Testing 1, 2, and 3 finds more bugs than testing 1000.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`for (i = 0; i <= n; i++)` over an array	Reads one element past the end	Use `<`, not `<=`. Half-open ranges.
`i < n - 1` when `n` is unsigned and 0	Wraps; loop runs forever	Write `i + 1 < n`.
`for (i = 0; i < n; i++);`	Empty body; the block runs once	Delete the semicolon; GCC's `-Wempty-body` does not cover `for`, but Clang warns.
`continue` in a `while` before the increment	Infinite loop	Use `for`, or increment before `continue`.
Modifying the counter inside the body	Iteration count becomes unpredictable	Let the header own the counter.
`break` expected to leave both loops	Only the inner loop exits	Extract a function and `return`.
Forgetting the semicolon after `do { } while (c)`	Confusing syntax error	It is required.
Comparing floats in the loop condition	May never terminate	Count with an integer and compute the float from it.

8 Check yourself

Why is `i < n` preferred over `i <= n - 1`?

Three reasons. The iteration count is visible as `n` with no arithmetic; the empty case `n == 0` works without a special guard, whereas `n - 1` on an unsigned type wraps to a huge value; and adjacent half-open ranges tile exactly, with no gap or overlap.

What is a loop invariant and what does it buy you?

A statement true before the loop and preserved by every iteration, so it is still true when the loop exits. Combined with the exit condition it yields an argument that the loop computes the right answer — a proof sketch rather than a successful test. Writing it also tends to expose the empty and single-element cases immediately.

Why can `continue` in a `while` loop cause an infinite loop but not in a `for` loop?

In a `while` loop the increment is a statement in the body, and `continue` jumps past it. In a `for` loop the update clause is part of the loop header and runs on every `continue`. This is a concrete reason to prefer `for` whenever a counter is involved.

`break` inside a nested loop only exits one level. What is the best fix?

Extract the nested loops into a function and use `return`, which leaves every level at once and usually improves the code by naming the operation. A flag works but clutters both conditions, and `goto` to a label after the loops is acceptable but harder to read.

Is `goto` forbidden in professional C?

No. Its use as general control flow is obsolete, but two patterns are standard practice: escaping nested loops, and unwinding acquired resources on an error path — `goto cleanup`, which is pervasive in the Linux kernel and most serious C libraries because C has no destructors. Chapter 27 builds that pattern properly.

9 Where this leads

You now have every control structure C offers — there are no others. Chapter 11 turns to organizing code rather than execution: functions, what happens to the call stack when one is called, and why a function that tries to modify its argument cannot succeed. That last failure is the question chapter 13 answers with pointers.

BASIC · CHAPTER 11

Functions

Functions are how a program stops being one long sequence and becomes a set of named ideas. C's version is simple and has exactly one rule that catches everyone: arguments are copied, always, with no exceptions.

BY THE END OF THIS CHAPTER YOU CAN

- Declare, define, and call functions, and explain what a prototype is for.
- Describe what happens to the call stack during a call and a return.
- Explain why a function cannot modify its caller's variable, and what that implies.
- Decompose a long `main` into functions with single responsibilities.
- Design a function contract that reports failure rather than hiding it.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Declaration, definition, call

A **definition** provides the body:

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

A **declaration**, also called a prototype, gives only the interface and ends with a semicolon:

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

A **call** uses it:

```
int sum = add(3, 4);
```

The compiler reads a file top to bottom and must know a function's interface before it sees a call. Two ways to arrange that: define the function above its first use, or declare it at the top

and define it anywhere. The second scales better, and in chapter 28 the declarations move into a header file.

Why prototypes matter

A prototype lets the compiler check the call. Without one, in pre-C99 code, the compiler assumed the function returned `int` and checked nothing at all:

```
/* no declaration in scope */
double result = square(3.0);   /* compiler assumes int square() */
```

The argument would be passed as a `double` but read as an `int`, and the return value interpreted as an `int` too. The result is garbage, and in C89 there was no diagnostic. This is the single largest safety improvement C89 made, and modern compilers now reject it:

```
error: implicit declaration of function 'square'
       [-Wimplicit-function-declaration]
```

`void` in both positions

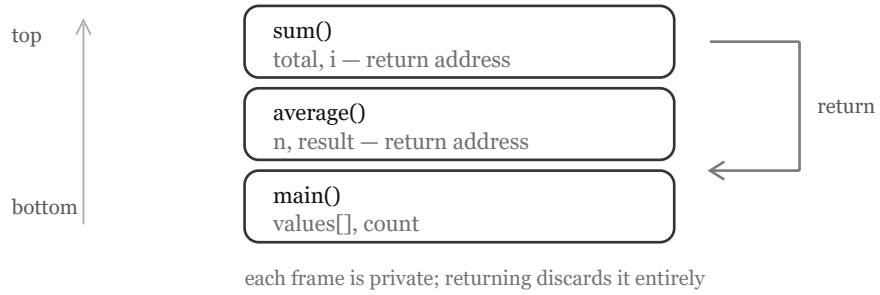
```
void print_banner(void);     /* returns nothing, takes nothing */
```

Write `(void)` for an empty parameter list, not `()`. In C89, empty parentheses mean "unspecified parameters — check nothing", which switches off the very checking prototypes exist to provide. C23 finally made `()` mean the same as `(void)`, but the explicit form remains correct everywhere.

2 The call stack

Chapter 17 covers program memory layout in full; the part needed now is the stack.

When a function is called, the machine pushes a **stack frame** containing the arguments, the local variables, and the address to return to. When the function returns, that frame is popped and the memory is reused by the next call.



Frames stack up as calls nest and unwind as they return. Chapter 18 watches this happen in a debugger.

Two consequences follow directly, and both matter.

Local variables do not survive the return. The frame is gone; the memory belongs to whatever is called next. Returning a pointer to a local is therefore returning a pointer to memory that no longer belongs to you — chapter 16's "dangling return".

Deep recursion exhausts the stack. Each call consumes a frame, and the stack has a fixed size — typically 8 MB on Linux. Chapter 18 measures exactly how deep you can go.

3 Pass by value — the rule with no exceptions

When you call a function, C copies each argument into the function's parameter. The parameter is a separate variable that happens to start with the same value.

```
#include <stdio.h>

void try_to_double(int n)
{
    n = n * 2;             /* modifies the copy */
}

int main(void)
{
    int value = 5;
    try_to_double(value);
    printf("%d\n", value); /* 5 — unchanged */
}
```

This is not a limitation to work around casually; it is a guarantee. A function cannot secretly alter the variables you pass it, which makes calls easy to reason about. The cost is that a function which genuinely needs to modify the caller's data must be given something else: the variable's *address*. That is chapter 13, and it is why `scanf` in chapter 8 required an `&`.

The swap that cannot work

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
/* swaps two copies, then discards both */
```

Every C programmer writes this once. It compiles cleanly, runs without error, and does nothing. Keep the file — chapter 13 fixes it with a few asterisks and two ampersands, and the fix is the clearest possible demonstration of what a pointer is for.

> **Arrays are the apparent exception.** Pass an array to a function and the function *can* modify its contents, which looks like a contradiction. It is not: what gets copied is a pointer to the first element, so the copy still refers to the same memory. Chapter 13 explains this properly under the name "array decay". Note it now so it does not look like magic in chapter 12.

4 Designing a function

One job

A function should do one thing that can be named without using "and". `read_config_and_connect` is two functions. The test is the name: if an honest name needs a conjunction, split it.

Reporting failure

C has no exceptions. A function that can fail must say so through its return value or through an output parameter, and the caller must check. Three conventions dominate:

CONVENTION	EXAMPLE	USE WHEN
Return the result; use an impossible value for failure	`find` returns `n` for not-found	There is a value outside the valid range
Return a status; write the result through a pointer	`bool parse_int(const char *s, int *out)`	Every value is a legitimate result
Return `NULL` or a negative code	`fopen`, `malloc`	Returning a pointer

The second is the most generally useful and the one chapter 16 develops. Pick one convention per project and apply it everywhere — chapter 27 turns this into an explicit error contract.

`static` for helpers

```
static int helper(int x) { … }
```

Marking a function `static` restricts its name to the current source file. Use it for everything that is not part of a file's public interface: it prevents name collisions, lets the compiler inline more aggressively, and documents intent. Chapter 28 makes this the basis of information hiding.

Parameters

- Few is better than many. More than four or five usually signals that some of them belong together in a structure — chapter 21.
- Mark pointer parameters you do not modify as `const`. It documents the contract and lets the compiler enforce it. Chapter 16.
- Name parameters in the prototype too. `int add(int, int);` is legal, but `int add(int left, int right);` tells the reader which is which.

5 Worked example: decomposing a long `main`

Here is a program that works. It reads scores, computes statistics, and prints a report — all in one function.

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

int main(void)
{
    int scores[] = { 88, 92, 79, 95, 61, 73, 84 };
```

```
    size_t n = sizeof scores / sizeof scores[0];

    int total = 0;
    for (size_t i = 0; i < n; i++) total += scores[i];
    double mean = (double)total / n;

    int hi = scores[0], lo = scores[0];
    for (size_t i = 1; i < n; i++) {
        if (scores[i] > hi) hi = scores[i];
        if (scores[i] < lo) lo = scores[i];
    }

    int passing = 0;
    for (size_t i = 0; i < n; i++) if (scores[i] >= 70) passing++;

    printf("n=%zu mean=%.2f hi=%d lo=%d pass=%d\n", n, mean, hi, lo, passing);
    return 0;
}
```

Nothing is wrong with the output. What is wrong is that no part of it can be reused, tested, or read independently, and the reader must hold the whole thing in their head at once.

The same program, decomposed

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

#define PASS_MARK 70

/* Each of these does one thing, has a name that says so,
   and can be tested on its own. */

static int sum(const int values[], size_t n)
{
    int total = 0;
    for (size_t i = 0; i < n; i++) {
        total += values[i];
    }
```

```
    return total;
}

/* Returns false for an empty array rather than dividing by zero. */
static bool mean(const int values[], size_t n, double *out)
{
    if (n == 0) {
        return false;
    }
    *out = (double)sum(values, n) / (double)n;
    return true;
}

static int maximum(const int values[], size_t n)
{
    int best = values[0];               /* caller guarantees n > 0 */
    for (size_t i = 1; i < n; i++) {
        if (values[i] > best) {
            best = values[i];
        }
    }
    return best;
}

static int minimum(const int values[], size_t n)
{
    int best = values[0];
    for (size_t i = 1; i < n; i++) {
        if (values[i] < best) {
            best = values[i];
        }
    }
    return best;
}

static size_t count_at_least(const int values[], size_t n, int threshold)
{
    size_t found = 0;
    for (size_t i = 0; i < n; i++) {
```

```
        if (values[i] >= threshold) {
            found++;
        }
    }
    return found;
}

static void print_report(const int values[], size_t n)
{
    if (n == 0) {
        puts("no scores to report");
        return;
    }

    double average = 0.0;
    mean(values, n, &average);          /* cannot fail: n > 0 checked above */

    printf("%-16s %zu\n",     "count",   n);
    printf("%-16s %.2f\n",    "mean",    average);
    printf("%-16s %d\n",      "highest", maximum(values, n));
    printf("%-16s %d\n",      "lowest",  minimum(values, n));
    printf("%-16s %zu of %zu\n", "passing",
           count_at_least(values, n, PASS_MARK), n);
}

int main(void)
{
    const int scores[] = { 88, 92, 79, 95, 61, 73, 84 };
    const size_t n = sizeof scores / sizeof scores[0];

    print_report(scores, n);

    puts("\n-- the empty case, which the original would have divided by zero --");
    print_report(scores, 0);

    puts("\n-- pass by value --");
    int value = 5;
    printf("before: %d\n", value);
    /* try_to_double(value) would change nothing; see below */
```

```
    printf("after:  %d\n", value);

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o report report.c
./report
```

What the decomposition bought

An edge case became visible. The original divided by `n` with no check. Writing `mean` as its own function with a stated contract forced the question "what if `n` is zero?", and the answer became part of the interface. Bugs hide in code that is never named.

Each piece is testable. `maximum` can be called with a one-element array, a sorted array, an array of equal values. In chapter 35 these become actual unit tests; the decomposition is what makes that possible.

`const` **documents and enforces.** Every function that only reads its array says so. Try adding `values[0] = 0;` inside `sum` and the compiler refuses — the contract is checked, not merely described.

The reader can stop early. Someone who wants to know how the report is laid out reads `print_report` and nothing else.

The swap experiment

Add this and run it. It is the setup for chapter 13.

```
static void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    printf("  inside swap: a=%d b=%d\n", a, b);
}

/* in main: */
int x = 1, y = 2;
printf("before: x=%d y=%d\n", x, y);
swap(x, y);
printf("after:  x=%d y=%d\n", x, y);
```

```
before: x=1 y=2
  inside swap: a=2 b=1
after:  x=1 y=2
```

The swap genuinely happened — to the copies. Both frames existed at once, held different values, and then the callee's frame was discarded. Nothing was lost in translation; there was simply never a connection between `a` and `x`.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Calling a function before declaring it	`implicit declaration` error	Prototype at the top, or define above the call.
Writing `int f()` for no parameters	Pre-C23, disables argument checking	`int f(void)`.
Expecting a function to modify its argument	Nothing changes	Pass the address — chapter 13.
Returning a pointer to a local variable	Dangling pointer; undefined behavior	Return by value, or let the caller supply the storage.
Ignoring a function's error return	The program continues on bad data	Check every return that can fail.
Omitting `const` on read-only pointer parameters	Contract unstated and unenforced	Add it; the compiler will check.
A function that needs six parameters	Calls become unreadable and error-prone	Group related parameters into a struct — chapter 21.
No `static` on file-local helpers	Names leak; collisions at link time	Mark them `static` — chapter 28.

7 Check yourself

What is the difference between a declaration and a definition?

A declaration states the interface — name, return type, parameter types — and ends with a semicolon. A definition additionally supplies the body. A program may contain many identical declarations of a function but exactly one definition. Chapter 28 puts declarations in headers and definitions in source files for precisely this reason.

Why write `(void)` rather than `()` for an empty parameter list?

Before C23, empty parentheses meant "parameters unspecified", which tells the compiler to check nothing at the call site — the opposite of what a prototype is for. `(void)` explicitly states that the function takes no arguments, so a call passing any is an error.

Why can't a C function modify a variable passed to it?

Because arguments are copied into the parameters. The parameter is a distinct variable in a distinct stack frame; assigning to it changes only the copy, which is discarded on return. To modify the caller's variable the function must be given its address instead — which is what `scanf`'s `&` is doing.

Why is returning a pointer to a local variable wrong?

The local lives in the function's stack frame, and that frame is discarded when the function returns. The pointer then refers to memory that belongs to whatever is called next. It often appears to work, because the value has not yet been overwritten, which makes the bug worse rather than better.

Your function computes an average and the caller passed an empty array. What should it do?

Report failure rather than divide by zero — either by returning a status and writing the result through an output parameter, or by documenting a precondition the caller must satisfy. What it must not do is compute a value that means nothing. Deciding this is part of designing the function, not an afterthought.

8 Where this leads

Chapter 12 introduces arrays, which is where the pass-by-value rule appears to break: pass an array to a function and the function can change it. Understanding why that is consistent rather than exceptional requires pointers, which arrive in chapter 13 — and which also fix the `swap` you just watched fail.

BASIC · CHAPTER 12

One- and Multi-Dimensional Arrays

An array is the simplest aggregate C has: a fixed number of elements of one type, laid out end to end in memory. That simplicity is its strength and its danger — the language will not stop you writing past the end, and it will not tell you that you did.

BY THE END OF THIS CHAPTER YOU CAN

- Declare, initialize, and traverse arrays, and compute their length correctly.
- Explain why `size_t` is the right index type.
- Describe exactly what happens on an out-of-range access, and demonstrate it.
- Work with two-dimensional arrays and explain row-major layout.
- Write the standard array algorithms: sum, extremes, reversal, linear search.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Declaring and initializing

```
int scores[5];                      /* five ints, uninitialized */
int primes[5] = { 2, 3, 5, 7, 11 };  /* five ints, given values  */
int zeros[5]  = { 0 };               /* first is 0, rest are 0   */
int partial[5] = { 1, 2 };           /* 1, 2, 0, 0, 0            */
int sized[]   = { 4, 8, 15, 16 };    /* length 4, deduced        */
```

Two rules in that list are worth stating explicitly. If you supply *any* initializer, every element you did not mention is set to zero — so `{ 0 }` is the idiomatic way to zero an entire array. And if you supply no initializer at all, the elements contain garbage, exactly as an uninitialized scalar does in chapter 5.

The size must be a constant expression in standard C89. C99 added variable-length arrays, where the size can be a run-time value; they are convenient and carry real hazards, which is why they are deferred to chapter 38.

Computing the length

```
size_t n = sizeof primes / sizeof primes[0];
```

The total size divided by the size of one element. Using `primes[0]` rather than `int` keeps the expression correct if the element type later changes.

> **This only works where the array is declared.** Pass the array to a function and `sizeof` inside that function gives the size of a *pointer*, not the array — typically 8 instead of 20. Chapter 13 explains why. Until then: a function that receives an array must also receive its length as a separate parameter, which is why every function in chapter 11's worked example took `size_t n`.

Designated initializers

```
int sparse[10] = { [2] = 5, [7] = 9 };   /* C99: others are 0 */
```

Useful for lookup tables where most entries are zero, and far more readable than counting commas. Chapter 21 uses the same syntax on structures.

2 Indexing, and why `size_t`

Elements are numbered from 0. An array of `n` elements has valid indices 0 through `n-1` — the half-open range [0, n) from chapter 10, which is exactly why the canonical loop is written the way it is.

```
for (size_t i = 0; i < n; i++) {
    printf("%d ", values[i]);
}
```

Use `size_t`, not `int`, for indices. It is unsigned, so it cannot hold a meaningless negative index; it is guaranteed wide enough for any array the platform can hold; and it is the type `sizeof` yields, so comparing against a computed length does not mix signedness and trigger chapter 7's trap.

The cost is the countdown hazard from chapter 5. To iterate backwards:

```
for (size_t i = n; i-- > 0; ) {
    printf("%d ", values[i]);
}
```

This reads oddly and is worth decoding once. The condition tests `i` against 0 *and then* decrements it, so the body sees `n-1` down to `0`, and the loop exits when the test sees 0; the

last decrement does wrap `i`, but `i` is never used again. It handles `n == 0` correctly with no special case.

3 Out of range: what actually happens

C performs no bounds checking. None. `values[10]` on a five-element array computes an address and reads or writes it, exactly as if you had asked for something legitimate.

Indexing is defined as address arithmetic: `values[i]` means the object at address `values + i × sizeof(element)`. If `i` is out of range, that address belongs to something else — another variable, saved register state, or the function's return address.

WHAT YOU HIT	SYMPTOM
Padding between variables	Nothing. The bug hides, possibly for years.
Another local variable	That variable changes for no visible reason.
The saved return address	The program crashes on `return`, far from the real bug.
An unmapped page	Immediate segmentation fault — the lucky case.

The first row is the worst outcome, not the best. A bug that reliably crashes is a bug you will fix today. A bug that silently corrupts a neighbouring variable is a bug you will chase for a chapter, and the crash will happen somewhere innocent.

This is the defect class behind a large share of security vulnerabilities, and the reason chapters 20, 35, and 36 all return to it. The practical defence is tooling:

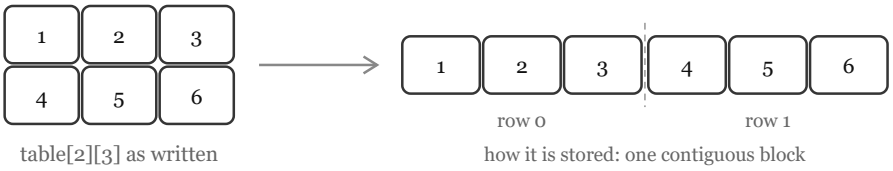
```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o prog prog.c
./prog
```

AddressSanitizer instruments every access and reports the exact line, the array, and the offset. It costs roughly a factor of two in speed, which is irrelevant during development. Use it from now on.

4 Two-dimensional arrays

```
int grid[3][4];              /* 3 rows, 4 columns */
int table[2][3] = {
    { 1, 2, 3 },
    { 4, 5, 6 }
};
grid[1][2] = 7;              /* row 1, column 2 */
```

A two-dimensional array is an array of arrays, and it is stored in **row-major order**: the whole first row, then the whole second, contiguously in memory.



table[2][3] as written — how it is stored: one contiguous block

`table[r][c]` sits at offset `r * 3 + c` from the start.

Row-major layout has a practical consequence that chapter 34 measures: **iterate rows outermost, columns innermost**. That walks memory in order, and the cache supplies the next elements for free. Reverse the loops and each step jumps a row's width, defeating the cache. On a large matrix the difference can be several times the runtime — same result, same operation count.

```
/* fast: sequential in memory */
for (size_t r = 0; r < ROWS; r++)
    for (size_t c = 0; c < COLS; c++)
        sum += m[r][c];

/* slow: strides across memory */
for (size_t c = 0; c < COLS; c++)
    for (size_t r = 0; r < ROWS; r++)
        sum += m[r][c];
```

Passing a 2D array to a function

All dimensions except the first must be specified, because the compiler needs the row width to compute offsets:

```
void print_grid(int g[][4], size_t rows);    /* column count required */
void print_grid(int (*g)[4], size_t rows);   /* the same declaration  */
```

This rigidity is one reason real code often uses a one-dimensional array with manual indexing, `m[r * cols + c]`, which accepts any shape at run time. Chapter 19 builds the dynamically allocated version.

5 Worked example: transpose, and a demonstration of corruption

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

#define ROWS 3
#define COLS 4

static void print_matrix(const char *label, size_t rows, size_t cols,
                         const int m[rows][cols])
{
    printf("%s (%zux%zu)\n", label, rows, cols);
    for (size_t r = 0; r < rows; r++) {
        printf("  ");
        for (size_t c = 0; c < cols; c++) {
            printf("%4d", m[r][c]);
        }
        putchar('\n');
    }
}

static void transpose(size_t rows, size_t cols,
                      const int src[rows][cols], int dst[cols][rows])
{
    for (size_t r = 0; r < rows; r++) {
```

```
        for (size_t c = 0; c < cols; c++) {
            dst[c][r] = src[r][c];
        }
    }
}

/* Bounds-checked access. Returns false instead of reading out of range. */
static bool at(const int values[], size_t n, size_t index, int *out)
{
    if (index >= n) {
        return false;
    }
    *out = values[index];
    return true;
}

static int sum(const int values[], size_t n)
{
    int total = 0;
    for (size_t i = 0; i < n; i++) {
        total += values[i];
    }
    return total;
}

static void reverse(int values[], size_t n)
{
    for (size_t i = 0; i < n / 2; i++) {
        int temp = values[i];
        values[i] = values[n - 1 - i];
        values[n - 1 - i] = temp;
    }
}

static size_t find(const int values[], size_t n, int target)
{
    for (size_t i = 0; i < n; i++) {
        if (values[i] == target) {
            return i;
```

```
        }
    }
    return n;                           /* n means "not found" */
}

int main(void)
{
    int m[ROWS][COLS] = {
        {  1,  2,  3,  4 },
        {  5,  6,  7,  8 },
        {  9, 10, 11, 12 }
    };
    int t[COLS][ROWS];

    print_matrix("original", ROWS, COLS, m);
    transpose(ROWS, COLS, m, t);
    print_matrix("transposed", COLS, ROWS, t);

    puts("\nrow-major layout: the same 12 ints, read linearly");
    const int *flat = &m[0][0];
    printf("  ");
    for (size_t i = 0; i < ROWS * COLS; i++) {
        printf("%4d", flat[i]);
    }
    puts("\n  note the rows appear one after another");

    int values[] = { 4, 8, 15, 16, 23, 42 };
    const size_t n = sizeof values / sizeof values[0];

    printf("\nsum      = %d\n", sum(values, n));
    printf("find 23  = index %zu\n", find(values, n, 23));
    printf("find 99  = %s\n", find(values, n, 99) == n ? "not found" : "found");

    reverse(values, n);
    printf("reversed = ");
    for (size_t i = 0; i < n; i++) {
        printf("%d ", values[i]);
    }
    putchar('\n');
```

```
    puts("\nbounds-checked access:");
    int got;
    printf("  index 2 -> %s", at(values, n, 2, &got) ? "" : "refused\n");
    if (at(values, n, 2, &got)) printf("%d\n", got);
    printf("  index 99 -> %s\n", at(values, n, 99, &got) ? "returned a value" : "refused");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o arrays arrays.c
./arrays
```

Now corrupt something on purpose

Add this function and call it. It demonstrates the table in section 3 rather than describing it.

```
static void demonstrate_corruption(void)
{
    int before = 111;
    int small[4] = { 1, 2, 3, 4 };
    int after  = 999;

    printf("\nbefore=%d  after=%d\n", before, after);
    printf("writing to small[4] and small[5], which do not exist...\n");

    small[4] = 0;                /* one past the end */
    small[5] = 0;                /* two past the end */

    printf("before=%d  after=%d   <-- one of these probably changed\n",
           before, after);
}
```

Compile without sanitizers first:

```
gcc -std=c17 -Wall -Wextra -g -o arrays arrays.c && ./arrays
```

Typically one of `before` or `after` becomes 0, depending on how the compiler arranged the stack. There is no error, no warning at run time, and no crash. The program has quietly destroyed a variable that the code you are looking at never mentions.

Now with the sanitizer:

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o arrays_asan arrays.c
./arrays_asan
```

```
==12345==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffd...
WRITE of size 4 at 0x7ffd... thread T0
    #0 0x... in demonstrate_corruption arrays.c:118
  Address is located in stack of thread T0 at offset 52 in frame
    'small' (line 112) <== Memory access at offset 52 overflows this variable
```

Exact file, exact line, the variable's name, and the offset. That is the difference between an afternoon and ten seconds, and it is available to you for one compiler flag.

The `sizeof` trap, made visible

Add this pair and compare the output:

```
static void show_sizeof(const int values[])
{
    printf("  inside a function: sizeof values   = %zu\n", sizeof values);
}

/* in main: */
printf("\nsizeof trap:\n");
printf("  where declared:    sizeof values   = %zu\n", sizeof values);
show_sizeof(values);
```

```
  where declared:    sizeof values   = 24
  inside a function: sizeof values   = 8
```

Twenty-four bytes for six `int`s where the array is declared; eight bytes — one pointer — inside the function. GCC even warns: *'sizeof' on array function parameter will return size of 'const int *'*. Chapter 13 explains the decay that causes it.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`for (i = 0; i <= n; i++)`	Reads or writes one past the end	Use `<`. Valid indices are 0 to n−1.
`sizeof` on an array parameter	Gives the pointer size, not the array size	Pass the length as a separate parameter.
Indexing with a signed `int`	Sign-comparison warnings; negative index possible	Use `size_t`.
`for (size_t i = n - 1; i >= 0; i--)`	Infinite loop	`for (size_t i = n; i-- > 0; )`.
`int a[5]; a = b;`	Compile error — arrays are not assignable	Copy elementwise, or use `memcpy` — chapter 14.
Comparing arrays with `==`	Compares addresses, not contents	`memcmp`, or an elementwise loop.
Iterating a matrix column-first	Correct but several times slower	Rows outermost — chapter 34 measures it.
Omitting the column count when passing a 2D array	Compile error	`void f(int m[][COLS], size_t rows)`.

7 Check yourself

Why does `sizeof arr / sizeof arr[0]` stop working inside a function?

Because the parameter is not an array — it is a pointer to the first element, so `sizeof` yields the pointer's size. The array's length is simply not available inside the function, which is why every C function taking an array also takes its length. Chapter 13 explains the conversion that causes this.

What does C do when you write to `arr[10]` on a five-element array?

It computes the address ten elements past the start and writes there, with no check of any kind. What gets destroyed depends on the memory layout: another variable, saved register state, or the return address. The silent cases are the dangerous ones, because the symptom appears far from the cause.

Why is `size_t` preferred over `int` for array indices?

It is unsigned, so a negative index is not representable; it is guaranteed wide enough for any object the platform supports; and it matches the type of `sizeof` and of `strlen`, so comparisons against lengths do not mix signedness and trigger the conversion trap from chapter 7.

What is row-major order and why should you care?

A 2D array is stored one complete row after another in contiguous memory, so `m[r][c]` sits at offset `r * cols + c`. It matters for speed: iterating with rows outermost walks memory sequentially and uses the cache well, while column-first strides across memory and can be several times slower for identical work.

How do you iterate an array backwards using `size_t`?

`for (size_t i = n; i-- > 0; )`. The condition compares the current value against 0 and then decrements, so the body sees `n-1` down to 0 and the loop exits; the final decrement wraps the counter, but the wrapped value is never used. The naive `i = n - 1; i >= 0` never terminates, because an unsigned value is always at least zero.

8 Where this leads

Two unexplained things are now outstanding: why `sizeof` changes inside a function, and why a function can modify an array's contents despite pass-by-value. Both have the same answer, and chapter 13 gives it. Pointers also repair the `swap` from chapter 11, explain the `&` that `scanf` has been demanding since chapter 8, and open the door to strings in chapter 14.

BASIC · CHAPTER 13

Introduction to Pointers

Chapter 1 said memory is a numbered list of bytes. A pointer is simply a variable that holds one of those numbers. Everything difficult about pointers comes not from that idea but from what C lets you do with it — and this chapter closes three loose ends left open since chapter 8.

BY THE END OF THIS CHAPTER YOU CAN

- Take the address of a variable and store it, and read the value back through it.
- Explain what a pointer's type is for, given that every address is the same size.
- Distinguish a null pointer, an uninitialized pointer, and a dangling pointer.
- Fix the `swap` that failed in chapter 11 and explain the `&` that `scanf` demanded in chapter 8.
- Explain array decay, and why `sizeof` changed inside a function in chapter 12.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Addresses

Every object in memory occupies a run of bytes starting at some address. The **address-of operator** `&` gives you that starting address:

```
int count = 42;
printf("value   = %d\n",  count);
printf("address = %p\n", (void *)&count);
```

```
value   = 42
address = 0x7ffd3c2a4b6c
```

The exact number is meaningless and will differ on every run — modern operating systems randomize layout deliberately. What matters is that the variable *has* an address, and that you can hold it.

The cast to `void *` in that `printf` is required: `%p` expects a `void *`, and passing any other pointer type is a mismatch of the kind chapter 8 warned about.

Declaring a pointer

```
int *p = &count;      /* p holds the address of count */
```

Read the declaration from the inside out: `*p` is an `int`, therefore `p` is a pointer to `int`. Chapter 16 turns that reading rule into a systematic method for declarations far worse than this one.

The asterisk belongs to the variable, not the type. This matters:

```
int* a, b;     /* a is int*, b is a plain int — almost never intended */
int *a, *b;    /* both are pointers                                    */
```

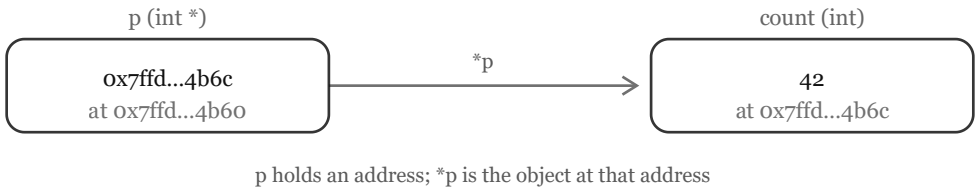
Writing `int *p` rather than `int* p` makes the trap visible, which is why most C style guides — and the standard library's own headers — attach the asterisk to the name.

Dereferencing

The **indirection operator** `*`, applied to a pointer, gives the object it points to:

```
printf("%d\n", *p);    /* 42 — read through the pointer */
*p = 99;               /* write through the pointer     */
printf("%d\n", count); /* 99 — count itself changed     */
```

That last line is the whole point. `p` and `count` are different variables, but `*p` and `count` are the same object.



Two variables. One holds a number that happens to be the other's address.

2 Why a pointer has a type

Every address is the same size — 8 bytes on a 64-bit machine, whatever it points to. So why does C distinguish `int *` from `char *`?

Two reasons, and both are load-bearing.

Dereferencing needs to know how many bytes to read. `*p` on an `int *` reads four bytes and interprets them as an integer; on a `char *` it reads one. The address alone does not say.

Pointer arithmetic scales by the element size. Adding 1 to an `int *` advances four bytes, not one. That is what makes `p[i]` work, and it is chapter 15's subject.

`void *`

A `void *` is an address with the type deliberately removed. It can hold any object pointer, which makes it the basis of generic code — `malloc` returns one in chapter 19, and chapter 30 builds generic containers on it.

```
int value = 42;
void *anything = &value;      /* fine: any object pointer converts */
int *back = anything;         /* fine in C: converts back implicitly */
/* *anything;  — error: the compiler does not know what is there */
```

You cannot dereference a `void *` without converting it to a real pointer type first. It is an address you have promised to remember the meaning of yourself.

3 The three ways a pointer goes wrong

Uninitialized — the wild pointer

```
int *p;          /* contains garbage, like any uninitialized variable */
*p = 42;         /* writes to an arbitrary address: undefined behavior */
```

This is chapter 5's uninitialized-variable problem with a much larger blast radius. A garbage `int` gives a wrong number; a garbage pointer corrupts memory somewhere unrelated. Always initialize — to a real address, or to `NULL`.

Null

```
int *p = NULL;   /* explicitly points at nothing */
if (p != NULL) {
    *p = 42;     /* guarded */
}
```

`NULL` is a pointer value guaranteed to compare unequal to the address of any object. It is the conventional "no result" return — `malloc` and `fopen` both use it — so checking for it is routine rather than defensive.

Dereferencing `NULL` is undefined behavior, but in practice it is the *good* failure: page zero is unmapped on every mainstream operating system, so the program dies immediately with a clear signal rather than corrupting something and continuing. C23 adds `nullptr`, which is type-safe; `NULL` remains universal.

Dangling

```
int *make_bad(void)
{
    int local = 42;
    return &local;      /* the frame dies at return */
}
```

The address was valid when it was taken. By the time the caller uses it, chapter 11's stack frame has been popped and that memory belongs to the next call. GCC warns — *function returns address of local variable* — and this is the reason chapter 11 insisted that locals do not survive the return. Chapter 20 meets the heap version, which the compiler cannot catch.

> **A habit worth forming now.** Initialize every pointer at declaration, set it to `NULL` when what it pointed to is gone, and check before dereferencing anything that could legitimately be null. Those three rules prevent most of what chapters 19 and 20 spend their time diagnosing.

4 Pointers as parameters — the fix for `swap`

Chapter 11 established that arguments are copied, so a function cannot change a caller's variable. The way around it is not to pass the variable but its address: the address is copied, and both copies refer to the same object.

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int x = 1, y = 2;
swap(&x, &y);        /* now x == 2, y == 1 */
```

Compare with the version that failed. The body is nearly identical; the difference is six asterisks (two in the parameter list, four in the body) and two ampersands at the call site. Nothing about the copying rule changed — what changed is what got copied.

This also finally explains `scanf`. `scanf("%d", &age)` passes the address of `age` so the function can write into it. Omit the `&` and you pass the *value* of `age` — an uninitialized integer — which `scanf` treats as an address and writes to. That is why the mistake corrupts memory instead of failing cleanly.

C has no pass-by-reference. It has pass-by-value applied to addresses, which achieves the same result with the mechanism left visible. Chapter 16 builds the output-parameter conventions that follow from it.

5 Array decay

Now the two mysteries from chapter 12.

In almost every context, **an array name converts to a pointer to its first element**. This conversion is called *decay*, and it happens silently.

```
int values[5] = { 1, 2, 3, 4, 5 };

values      /* in most expressions, this means &values[0] */
*values     /* therefore this is values[0], i.e. 1        */
```

Indexing is defined in terms of it: `values[i]` is exactly `*(values + i)`. That is not an analogy; it is the definition, and chapter 15 exploits it.

Why `sizeof` changed

Decay does *not* happen in three places: as the operand of `sizeof`, as the operand of `&`, and for a string literal used to initialize an array. So:

```
int values[5];
sizeof values          /* 20 — the array, no decay here */


void f(int values[])   /* the parameter is really int *values */
{
    sizeof values      /* 8 — a pointer */
}
```

A function parameter written as `int values[]` is not an array parameter. C has no array parameters. The compiler silently rewrites it to `int *values`, which is why the length must be passed separately and why `sizeof` reports 8.

Why a function can modify an array

Because what was copied is a pointer to the caller's memory. The function received an address, and writing through it writes to the original. Pass-by-value was never violated: the *pointer* was copied faithfully.

```
void zero_all(int values[], size_t n)   /* really int *values */
{
    for (size_t i = 0; i < n; i++) {
        values[i] = 0;                  /* writes to the caller's array */
    }
}
```

The two spellings `int values[]` and `int *values` are identical to the compiler. Use the bracket form when the parameter conceptually is an array — it documents intent — and the star form when it is a pointer to a single object.

6 Worked example: closing three loose ends

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

```
/* Chapter 11's version: swaps two copies, achieves nothing. */
static void swap_broken(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

/* The fix: six asterisks here and two ampersands at the call site. */
static void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

/* An output parameter: return a status, write the result through a pointer.
   This is the convention chapter 16 develops. */
static bool safe_divide(int numerator, int denominator, int *out)
{
    if (denominator == 0) {
        return false;
    }
    *out = numerator / denominator;
    return true;
}

/* Demonstrates decay: sizeof here reports a pointer, not the array. */
static void show_parameter_size(int values[], size_t n)
{
    printf("  inside the function: sizeof values = %zu  (a pointer)\n",
           sizeof values);
    printf("  the length had to be passed separately: n = %zu\n", n);
}

/* Writes through the decayed pointer, so the caller's array changes. */
static void scale_all(int values[], size_t n, int factor)
{
    for (size_t i = 0; i < n; i++) {
```

```
        values[i] *= factor;
    }
}

int main(void)
{
    puts("== 1. addresses ==");
    int count = 42;
    int *p = &count;
    printf("  count      = %d\n", count);
    printf("  &count     = %p\n", (void *)&count);
    printf("  p          = %p\n", (void *)p);
    printf("  *p         = %d\n", *p);
    *p = 99;
    printf("  after *p = 99, count = %d\n", count);
    printf("  sizeof(int) = %zu, sizeof(int *) = %zu\n",
           sizeof(int), sizeof(int *));

    puts("\n== 2. swap, broken and fixed ==");
    int x = 1, y = 2;
    swap_broken(x, y);
    printf("  after swap_broken: x=%d y=%d   (unchanged)\n", x, y);
    swap(&x, &y);
    printf("  after swap:        x=%d y=%d   (swapped)\n", x, y);

    puts("\n== 3. output parameters ==");
    int result;
    if (safe_divide(10, 3, &result)) {
        printf("  10 / 3 = %d\n", result);
    }
    if (!safe_divide(10, 0, &result)) {
        puts("  10 / 0 refused, and result was left untouched");
    }

    puts("\n== 4. null and guarded access ==");
    int *maybe = NULL;
    printf("  maybe is %s\n", maybe == NULL ? "NULL" : "valid");
    if (maybe != NULL) {
        printf("  %d\n", *maybe);
```

```
    } else {
        puts("  guarded: no dereference attempted");
    }

    puts("\n== 5. array decay ==");
    int values[5] = { 1, 2, 3, 4, 5 };
    const size_t n = sizeof values / sizeof values[0];
    printf("  where declared: sizeof values = %zu, length = %zu\n",
           sizeof values, n);
    show_parameter_size(values, n);

    printf("  values == &values[0]?  %s\n",
           (void *)values == (void *)&values[0] ? "yes" : "no");
    printf("  *values = %d, values[0] = %d, *(values + 2) = %d, values[2] = %d\n",
           *values, values[0], *(values + 2), values[2]);

    puts("\n== 6. a function modifying the caller's array ==");
    printf("  before: ");
    for (size_t i = 0; i < n; i++) printf("%d ", values[i]);
    scale_all(values, n, 10);
    printf("\n  after:  ");
    for (size_t i = 0; i < n; i++) printf("%d ", values[i]);
    puts("\n  pass-by-value was never broken: the pointer was copied");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o pointers pointers.c
./pointers
```

What the compiler says

Section 5 of the program provokes a warning you should read rather than silence:

```
pointers.c:36:12: warning: 'sizeof' on array function parameter 'values'
                  will return size of 'int *' [-Wsizeof-array-argument]
```

GCC knows exactly what mistake people make here and says so by name.

Two experiments

The dangling pointer. Add this and rebuild:

```
static int *make_dangling(void)
{
    int local = 42;
    return &local;
}
/* in main: */
int *bad = make_dangling();
printf("%d\n", *bad);          /* undefined behavior */
```

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

Run it. It may print 42, may print garbage, may crash. Then run it under `-fsanitize=address` with `ASAN_OPTIONS=detect_stack_use_after_return=1`, which reports *stack-use-after-return* precisely. (Recent GCC versions make such a function return a null pointer instead, so the run simply crashes.)

The missing ampersand. Change `swap(&x, &y)` to `swap(x, y)`:

```
error: passing argument 1 of 'swap' makes pointer from integer
       without a cast [-Wint-conversion]
```

Here the type system catches it. With `scanf` it cannot, because a variadic function declares nothing about its later arguments — which is exactly why forgetting the `&` there is so much more dangerous.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`int *p; *p = 5;`	Writes to a garbage address	Initialize to a real address or `NULL`.
`int* a, b;`	`b` is an `int`, not a pointer	Write `int *a, *b;` — one declaration per line is better still.
Returning `&local`	Dangling pointer	Return by value, or let the caller own the storage.
Dereferencing without a null check	Crash on the failure path	Check whatever can legitimately be `NULL`.
`sizeof` on an array parameter	Gives the pointer size	Pass the length; heed `-Wsizeof-array-argument`.
Omitting `&` in `scanf`	Memory corruption, no type error	Always pass the address.
`printf("%p", p)` without a cast	Format mismatch	`(void *)p`.
Confusing `*p = 5` with `p = 5`	The second sets the pointer to address 5	Read `*` as "the thing at".

8 Check yourself

If every address is 8 bytes, why does a pointer need a type?

Because the address alone does not say how many bytes to read or how to interpret them, and because pointer arithmetic scales by the element size — adding 1 to an `int *` moves four bytes. The type is what makes `*p` and `p[i]` meaningful.

Why does `swap(int *a, int *b)` work when `swap(int a, int b)` does not?

Both copy their arguments — the rule never changes. The difference is what is copied: in the second case, the values, so only the copies are exchanged. In the first, the addresses, so `*a` and `*b` refer to the caller's own variables and the exchange reaches them.

Why is dereferencing `NULL` better than dereferencing an uninitialized pointer?

Both are undefined behavior, but a null dereference targets an unmapped page on every mainstream system, so the program dies immediately with a clear signal at the point of the error. An uninitialized pointer holds an arbitrary address that may be writable, so the program corrupts something and continues — and the symptom appears far from the cause.

What is array decay, and where does it not happen?

In most expressions an array name converts to a pointer to its first element. It does not happen as the operand of `sizeof`, as the operand of `&`, or when a string literal initializes a character array. Those three exceptions are why `sizeof` reports the real array size where it is declared and a pointer size inside a function.

If arguments are always copied, how can a function change the contents of an array you passed?

Because the array decayed to a pointer before the call, so what was copied is an address pointing into the caller's memory. Writing through that copy writes to the original array. Pass-by-value is intact; the copied value simply happens to be a reference.

9 Where this leads

Chapter 14 applies all of this to strings, which in C are not a type at all but an array of characters ending in `'\0'` — and which are therefore handled entirely through the pointers you just met. Chapter 15 then develops the arithmetic that makes `values[i]` and `*(values + i)` the same expression.

Strings and Character Handling

C has no string type. It has a convention: an array of characters with a zero byte marking the end. Everything in `<string.h>` follows from that single decision, and so does a remarkable share of the security vulnerabilities of the last forty years.

BY THE END OF THIS CHAPTER YOU CAN

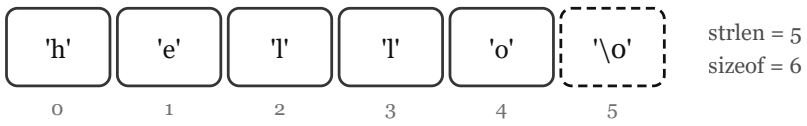
- Explain the null terminator and account for it in every buffer you size.
- Say why writing to a string literal crashes, and how to get a modifiable copy.
- Use the `<string.h>` functions correctly, including the ones that are traps.
- Choose between `strcpy`, `strncpy`, and `snprintf` and justify the choice.
- Use the block memory functions and explain when `memmove` is required.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The convention

A C string is a sequence of characters followed by `'\0'`, the character with value zero. Nothing records the length; every function that needs it scans forward until it finds the terminator.

```
char greeting[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char greeting[]  = "hello";     /* identical — the compiler adds '\0' */
```



Five characters, six bytes. The gap between `strlen` and `sizeof` is the source of most buffer bugs.

Two consequences follow immediately, and both cost real money every year.

Length is O(n), not O(1). `strlen` walks the whole string. Calling it in a loop condition — `for (i = 0; i < strlen(s); i++)` — makes an O(n) loop into an O(n²) one. Compute it

once.

Every buffer needs one extra byte. A buffer for ten characters must be `char buf[11]`. Off-by-one here is not a wrong answer; it is memory corruption.

2 Literals are read-only

```
char  array[]  = "hello";    /* a modifiable copy in the array   */
char *pointer  = "hello";    /* a pointer to read-only storage   */

array[0]   = 'H';            /* fine                              */
pointer[0] = 'H';            /* undefined behavior — usually crash */
```

The distinction is not stylistic. The first line creates a six-byte array on the stack and copies the literal into it. The second stores the *address* of the literal, which lives in a read-only section of the executable — the `.rodata` section that chapter 52 inspects directly. Writing there fails at the hardware level.

Always declare a pointer to a literal as `const`, so the compiler catches the mistake rather than the operating system:

```
const char *pointer = "hello";
pointer[0] = 'H';            /* now a compile error */
```

Even C23 leaves string literals as plain `char` arrays (only C++ makes them `const`), so writing `const` yourself is the only protection.

3 The `<string.h>` functions

FUNCTION	DOES	WATCH OUT FOR
`strlen(s)`	Length, excluding the terminator	O(n); returns `size_t`, so `strlen(s) - 1` wraps when empty
`strcpy(dst, src)`	Copy including the terminator	No bound. Overflows silently.
`strncpy(dst, src, n)`	Copy at most `n` bytes	Does *not* terminate if `src` is too long. See below.
`strcmp(a, b)`	<0, 0, or >0	Returns 0 for *equal* — the opposite of a boolean
`strcat(dst, src)`	Append	No bound, and rescans `dst` every call
`strchr(s, c)`	Pointer to the first `c`, or `NULL`	Searches for a `char`, not a string
`strstr(h, n)`	Pointer to the first occurrence of `n`	Returns a pointer into `h`, not a copy
`strtok(s, delims)`	Split into tokens	**Modifies** `s` and keeps hidden static state

`strcmp` returns 0 for equal

```
if (strcmp(a, b) == 0) { /* the strings are equal */ }
if (strcmp(a, b))      { /* the strings DIFFER   */ }
```

The second reads like "if a equals b" and means the opposite. Always compare explicitly against 0.

`strncpy` is not the safe `strcpy`

It looks like a bounded copy and is widely used as one. It is not. If the source is at least `n` bytes, **no terminator is written**, and you are left with an unterminated character array that the next `strlen` will run off the end of.

```
char dst[5];
strncpy(dst, "hello world", 5);   /* dst = "hello" with NO terminator */
printf("%s\n", dst);              /* reads past the end */
```

It also pads with zeros to the full length when the source is shorter, which is wasted work on a large buffer. It was designed in the 1970s for fixed-width database fields, not for safety. Use

`snprintf` instead:

```
snprintf(dst, sizeof dst, "%s", source);   /* always terminates */
```

`strtok` modifies its input

```
char text[] = "a,b,c";            /* must be modifiable — not a literal */
char *token = strtok(text, ",");
while (token != NULL) {
    puts(token);
    token = strtok(NULL, ",");    /* NULL means "continue the same string" */
}
/* text is now "a\0b\0c" */
```

It writes terminators into your buffer and remembers its position in a hidden static variable, which makes it unusable from two places at once and unsafe with threads. Chapter 41 returns to this under reentrancy; the thread-safe variant is `strtok_r`.

4 Block memory operations

These work on raw bytes and know nothing about terminators, so they need an explicit length. They are also the right tool for any array, not just characters.

```
memcpy(dst, src, n);    /* copy n bytes; regions must not overlap */
memmove(dst, src, n);   /* copy n bytes; overlap is handled        */
memset(buf, 0, n);      /* fill n bytes with a value               */
memcmp(a, b, n);        /* compare n bytes; 0 means equal          */
```

Why both `memcpy` and `memmove`

`memcpy` promises the regions do not overlap, which lets it copy in whatever order is fastest — several bytes at a time, possibly backwards. Give it overlapping regions and the result is undefined: it may copy a byte it has already overwritten.

```
char buf[] = "abcdef";
memcpy(buf + 1, buf, 5);    /* UNDEFINED: regions overlap  */
memmove(buf + 1, buf, 5);   /* correct: "aabcde"           */
```

`memmove` checks the direction and copies accordingly. It is marginally slower and always correct. When in doubt, use it.

`memset` sets *bytes*, which is why it is only useful for 0 and for single-byte values. `memset(arr, 1, sizeof arr)` on an `int` array does not fill it with ones; it fills every byte with 1, giving elements of `0x01010101`.

5 `<ctype.h>` and safe construction

```
#include <ctype.h>


isalpha(c)  isdigit(c)  isalnum(c)  isspace(c)
isupper(c)  islower(c)  ispunct(c)
toupper(c)  tolower(c)
```

One subtlety with a real consequence: these take an `int` whose value must be representable as `unsigned char`, or be `EOF`. Passing a plain `char` that is negative — which happens with byte values above 127 where `char` is signed, as chapter 4 noted — is undefined behavior. Cast:

```
if (isalpha((unsigned char)s[i])) { … }
```

Building strings safely

`snprintf` is the workhorse. It writes at most `size` bytes including the terminator, always terminates, and returns the length it *would* have written — which is how you detect truncation.

```
char line[32];
int needed = snprintf(line, sizeof line, "%s: %d", name, score);


if (needed < 0) {
    /* encoding error */
} else if ((size_t)needed >= sizeof line) {
    /* truncated: needed bytes were required */
}
```

That return value is the part people miss. A program that ignores it silently produces truncated output, which in a filename or a command is a security problem rather than a

cosmetic one.

6 Worked example: a string toolkit, by hand and by library

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>

/* --- written by hand, to show what the library does --- */

static size_t my_strlen(const char *s)
{
    const char *start = s;
    while (*s != '\0') {
        s++;
    }
    return (size_t)(s - start);      /* pointer subtraction: chapter 15 */
}

static void my_strcpy(char *dst, const char *src)
{
    while ((*dst++ = *src++) != '\0') {
        /* copies, including the terminator, then stops */
    }
}

static int my_strcmp(const char *a, const char *b)
{
    while (*a != '\0' && *a == *b) {
        a++;
        b++;
    }
    return (unsigned char)*a - (unsigned char)*b;
}

/* --- a safe copy, which the library does not provide --- */
```

```
static bool copy_into(char *dst, size_t dst_size, const char *src)
{
    int needed = snprintf(dst, dst_size, "%s", src);
    return needed >= 0 && (size_t)needed < dst_size;
}

static void to_upper_in_place(char *s)
{
    for (; *s != '\0'; s++) {
        *s = (char)toupper((unsigned char)*s);
    }
}

static size_t count_words(const char *s)
{
    size_t words = 0;
    bool in_word = false;
    for (; *s != '\0'; s++) {
        if (isspace((unsigned char)*s)) {
            in_word = false;
        } else if (!in_word) {
            in_word = true;
            words++;
        }
    }
    return words;
}

int main(void)
{
    const char *sample = "the quick brown fox";

    puts("== hand-written versus library ==");
    printf("  my_strlen  = %zu   strlen  = %zu\n",
           my_strlen(sample), strlen(sample));

    char a[32], b[32];
    my_strcpy(a, sample);
    strcpy(b, sample);
```

```
    printf("  my_strcpy  = \"%s\"\n", a);
    printf("  strcpy     = \"%s\"\n", b);

    printf("  my_strcmp(\"abc\",\"abd\") = %d   strcmp = %d\n",
           my_strcmp("abc", "abd"), strcmp("abc", "abd"));
    printf("  equal strings give 0: %d\n", strcmp("same", "same"));

    puts("\n== strlen versus sizeof ==");
    char buffer[32] = "hello";
    printf("  strlen(buffer) = %zu   sizeof buffer = %zu\n",
           strlen(buffer), sizeof buffer);
    puts("  the difference is room for growth plus the terminator");

    puts("\n== truncation, detected ==");
    char small[8];
    if (copy_into(small, sizeof small, "short")) {
        printf("  \"short\" fitted: \"%s\"\n", small);
    }
    if (!copy_into(small, sizeof small, "a much longer string")) {
        printf("  long string truncated to \"%s\" — and we know it\n", small);
    }

    puts("\n== strncpy does not always terminate ==");
    char danger[6];
    memset(danger, 'X', sizeof danger);      /* poison, to make it visible */
    strncpy(danger, "hello world", 5);
    printf("  first five bytes: %.5s\n", danger);
    printf("  byte 5 is '%c' (%d) — not a terminator\n",
           danger[5], danger[5]);
    puts("  printing this with %s would read past the end");

    puts("\n== memmove versus memcpy ==");
    char overlap[] = "abcdef";
    memmove(overlap + 1, overlap, 5);
    printf("  memmove overlapping: %s\n", overlap);
    puts("  memcpy on the same input is undefined behavior");

    puts("\n== tokenizing modifies the buffer ==");
    char csv[] = "red,green,blue";
```

```
    printf("  before: %s\n", csv);
    for (char *tok = strtok(csv, ","); tok != NULL; tok = strtok(NULL, ",")) {
        printf("    token: %s\n", tok);
    }
    printf("  after:  %s   <-- only the first token; commas became '\\0'\n", csv);

    puts("\n== character classification ==");
    char text[32];
    strcpy(text, sample);
    printf("  words in \"%s\" = %zu\n", text, count_words(text));
    to_upper_in_place(text);
    printf("  upper-cased: %s\n", text);

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o strings strings.c
./strings
```

The crash worth causing

Add these two lines and run it:

```
char *literal = "hello";      /* no const — the compiler allows it */
literal[0] = 'H';             /* undefined behavior */
```

```
Segmentation fault (core dumped)
```

Now add `const` and rebuild:

```
const char *literal = "hello";
literal[0] = 'H';
```

```
error: assignment of read-only location '*literal'
```

The same defect, moved from a run-time crash to a compile-time error by one keyword. That is the entire argument for `const` correctness, which chapter 16 develops into a discipline.

The overflow worth causing

```
char tiny[4];
strcpy(tiny, "far too long");   /* no bound checking anywhere */
```

Without the sanitizer this may appear to work, corrupting a neighbouring variable exactly as chapter 12's array overflow did. With `-fsanitize=address`:

```
==12345==ERROR: AddressSanitizer: stack-buffer-overflow
WRITE of size 13 at 0x7ffd... thread T0
    #0 0x... in main strings.c:118
```

This is the defect class that produced the Morris worm in 1988 and has not stopped since. Chapter 36 examines why it is so exploitable.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`char buf[5]; strcpy(buf, "hello");`	Six bytes into five: overflow	Size for length + 1, or use `snprintf`.
`if (strcmp(a, b))`	True when the strings *differ*	`strcmp(a, b) == 0`.
Trusting `strncpy` to terminate	Unterminated buffer; later `strlen` runs off the end	`snprintf(dst, sizeof dst, "%s", src)`.
`char *p = "x"; p[0] = 'y';`	Crash — literals are read-only	Use an array, or declare `const char *`.
`strlen` in a loop condition	O(n²) instead of O(n)	Compute once into a variable.
`strtok` on a literal	Crash — it writes terminators	Tokenize a modifiable copy.
`memcpy` with overlapping regions	Undefined; garbled result	`memmove`.
`isalpha(s[i])` with signed `char`	Undefined for bytes above 127	Cast to `unsigned char`.
Ignoring `snprintf`'s return value	Silent truncation	Compare it against the buffer size.

8 Check yourself

Why must a buffer holding ten characters be declared with size 11?

Because a C string carries no length field; the end is marked by a `'\0'` byte that occupies storage like any other character. `strlen` reports the characters before it, so the array must hold length + 1 bytes. Sizing for the length alone is not a wrong answer; it is a buffer overflow.

Why does `char *p = "hi"; p[0] = 'H';` crash while `char a[] = "hi"; a[0] = 'H';` does not?

The array form copies the literal into modifiable storage that belongs to your function. The pointer form stores the address of the literal itself, which the compiler places in a read-only section of the executable; the write is refused by the memory protection hardware. Declaring such pointers `const char *` turns the crash into a compile error.

Is `strncpy` the safe version of `strcpy`?

No. It bounds the number of bytes written, but if the source fills that bound it writes no terminator, leaving an unterminated array that the next string function will read past. It was designed for fixed-width records, not for safety. `snprintf` always terminates and reports whether truncation occurred.

When must you use `memmove` instead of `memcpy`?

Whenever the source and destination regions can overlap. `memcpy` is permitted to assume they do not and may copy in any order, so overlapping input gives undefined results. `memmove` detects the direction and copies safely, at a negligible cost.

Why is calling `strlen(s)` in a loop condition a performance bug?

Because the length is not stored anywhere — `strlen` scans to the terminator every time it is called. Evaluating it once per iteration turns an O(n) loop into O(n²). Compute it once before the loop, provided the string does not change inside it.

9 Where this leads

That is the Basic level complete. You can build, format, version, and reason about a C program; you know how values are represented, how types convert, how control flows, and how pointers and strings work. Chapter 15 opens the Intermediate level by developing pointer arithmetic properly — the `s - start` in `my_strlen` above was a preview — and chapters 19 and 20 then apply it to memory you allocate yourself.

PART 2

Intermediate

Memory, data, and programs that last

Chapters 15–28

INTERMEDIATE · CHAPTER 15

Pointer Arithmetic and Arrays

Chapter 13 established that `values[i]` *means* `*(values + i)`*. This chapter takes that identity seriously and develops the arithmetic behind it — including the rules that say which pointer operations are legal and which are quietly undefined.*

BY THE END OF THIS CHAPTER YOU CAN

- Explain why adding 1 to a pointer advances by the element size rather than one byte.
- Subtract two pointers and say what `ptrdiff_t` is for.
- Traverse an array or string with a moving pointer instead of an index.
- Build and use arrays of pointers and pointers to pointers.
- State the rule about the one-past-the-end pointer and why it exists.

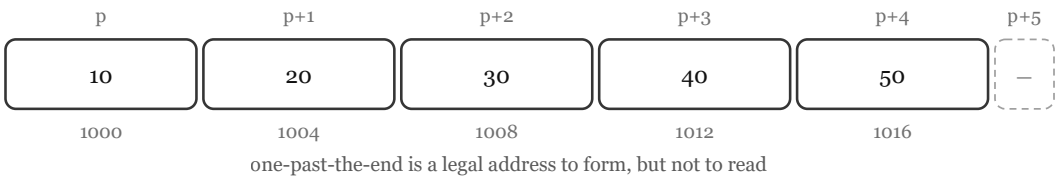
Code for this chapter: github.com/sercankulcu/programming-with-c

1 Arithmetic scales by the element

Adding an integer to a pointer does not add bytes. It advances by that many *elements*:

```
int values[5] = { 10, 20, 30, 40, 50 };
int *p = values;

p + 1      /* four bytes further on, pointing at values[1] */
p + 3      /* twelve bytes further on, pointing at values[3] */
```



one-past-the-end is a legal address to form, but not to read

Each step is `sizeof(int)` bytes. The pointer's type is what supplies the scale.

Because of this scaling, the two notations are interchangeable:

```
values[2]      /* 30 */
*(values + 2)  /* 30 — the same expression, by definition */
p[2]           /* 30 */
*(p + 2)       /* 30 */
2[values]      /* 30 — legal, since a+b == b+a. Never write this. */
```

That last line is a genuine curiosity of the language and occasionally appears in puzzles. It follows directly from indexing being defined as addition.

Increment and decrement

```
p++;      /* advance one element */
p--;      /* back one element     */
p += 3;   /* forward three        */
```

Only integers may be added to pointers. Adding two pointers is meaningless and does not compile.

2 Subtraction and `ptrdiff_t`

Subtracting two pointers into the same array yields the number of *elements* between them:

```
int *start = &values[1];
int *end   = &values[4];
ptrdiff_t gap = end - start;     /* 3, not 12 */
```

The result has type `ptrdiff_t`, from `<stddef.h>` — a signed integer type wide enough to hold the difference. It is signed because the difference can be negative, which is exactly why it is not `size_t`. Print it with `%td`.

This is how `my_strlen` in chapter 14 worked: walk a pointer to the terminator, then subtract the starting position.

Comparison

Pointers into the same array may be compared with `<`, `>`, `<=`, `>=`, which gives the idiomatic loop:

```
for (int *q = values; q < values + n; q++) {
    printf("%d ", *q);
}
```

Any two pointers may be compared with `==` and `!=`, including against `NULL`.

3 The rules you may not break

Pointer arithmetic is only defined **within a single array**, plus one position past its end. Outside that, the behavior is undefined — not merely unpredictable.

EXPRESSION	LEGAL?
`values + 5` for a 5-element array	Yes — one past the end may be formed
`*(values + 5)`	No — it may not be dereferenced
`values + 6`	No — undefined even without dereferencing
`values - 1`	No — before the start is undefined
Comparing pointers into two different arrays	Undefined with `<`; fine with `==`

The one-past-the-end allowance exists precisely so the loop above can terminate: `q < values + n` must be able to form `values + n`. The standard grants that address and nothing beyond it.

> **Why `values - 1` matters in practice.** A reverse loop written as `for (int *q = values + n - 1; q >= values; q--)` forms `values - 1` on the final decrement, before the comparison fails. That is undefined behavior, and on a segmented or checked implementation it can genuinely misbehave. Write it as `for (int *q = values + n; q-- > values; )` instead — the same shape as the `size_t` countdown from chapter 12.

4 Index or pointer?

Both traversals below are correct. Which to write is a real question with a defensible answer.

```
/* index style */
for (size_t i = 0; i < n; i++) {
    total += values[i];
}

/* pointer style */
for (const int *q = values; q < values + n; q++) {
    total += *q;
}
```

INDEX STYLE	POINTER STYLE
The position is available for messages and comparisons	No index variable to keep in sync
Reads naturally to most people	Natural for strings, where the end is a terminator, not a count
Works unchanged for multi-dimensional access	Historically produced better code

That last row needs qualifying. On 1980s compilers, pointer walking really was faster because address computation was expensive. Modern compilers apply strength reduction and produce identical machine code for both. You can verify this yourself with the `-S` technique from chapter 2, and the worked example below does exactly that.

Prefer index style by default, and use pointer style where it genuinely reads better — walking a string to its terminator, or advancing through a buffer whose remaining length you track separately.

5 Pointers to pointers, and arrays of pointers

A pointer is an object, so it has an address, so there can be a pointer to it:

```
int value = 42;
int *p = &value;
int **pp = &p;

**pp        /* 42          */
*pp         /* p, an int * */
```

Two places this appears constantly. An *array of pointers*, typically to strings:

```
const char *names[] = { "Ada", "Dennis", "Ken" };
size_t count = sizeof names / sizeof names[0];

for (size_t i = 0; i < count; i++) {
    printf("%s (%zu characters)\n", names[i], strlen(names[i]));
}
```

The array holds three pointers, each pointing at a literal somewhere in read-only memory. The strings need not be the same length, because the array stores addresses rather than the text. This is exactly the shape of `argv` in chapter 23.

And an *output parameter that is itself a pointer*, which chapter 19 needs for reallocation:

```
static void allocate(int **out, size_t n)
{
    *out = malloc(n * sizeof **out);   /* writes into the caller's pointer */
}

int *data = NULL;
allocate(&data, 10);
```

The rule is the same one as chapter 13: to modify a caller's variable you pass its address. If the variable is an `int *`, its address is an `int **`.

> `int *a[5]` **versus** `int (*a)[5]`**.** The first is an array of five pointers; the second is a pointer to an array of five ints. Brackets bind tighter than the asterisk, so parentheses change the meaning entirely. Chapter 16 gives the right-left rule that makes such declarations readable rather than memorized.

6 Worked example: the same loop, three ways

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

/* Index style. */
static long sum_indexed(const int *values, size_t n)
```

```
{
    long total = 0;
    for (size_t i = 0; i < n; i++) {
        total += values[i];
    }
    return total;
}

/* Pointer style: identical machine code on any modern compiler. */
static long sum_pointer(const int *values, size_t n)
{
    long total = 0;
    for (const int *q = values; q < values + n; q++) {
        total += *q;
    }
    return total;
}

/* Pointer style where it genuinely fits: no count is available,
   the terminator ends the walk. */
static size_t length_of(const char *s)
{
    const char *start = s;
    while (*s != '\0') {
        s++;
    }
    return (size_t)(s - start);
}

/* Reverse traversal done safely: never forms values - 1. */
static void print_backwards(const int *values, size_t n)
{
    for (const int *q = values + n; q-- > values; ) {
        printf("%d ", *q);
    }
    putchar('\n');
}

/* An output parameter that is itself a pointer. */
```

```
static void point_at_largest(int *values, size_t n, int **out)
{
    if (n == 0) {
        *out = NULL;
        return;
    }
    int *best = values;
    for (int *q = values + 1; q < values + n; q++) {
        if (*q > *best) {
            best = q;
        }
    }
    *out = best;
}

int main(void)
{
    int values[] = { 10, 20, 30, 40, 50 };
    const size_t n = sizeof values / sizeof values[0];
    int *p = values;

    puts("== scaling ==");
    printf("  sizeof(int)        = %zu\n", sizeof(int));
    printf("  p                  = %p\n", (void *)p);
    printf("  p + 1              = %p   (+%td bytes)\n",
           (void *)(p + 1), (char *)(p + 1) - (char *)p);
    printf("  p + 3              = %p   (+%td bytes)\n",
           (void *)(p + 3), (char *)(p + 3) - (char *)p);

    puts("\n== the four spellings of one element ==");
    printf("  values[2]=%d  *(values+2)=%d  p[2]=%d  *(p+2)=%d\n",
           values[2], *(values + 2), p[2], *(p + 2));

    puts("\n== subtraction gives elements, not bytes ==");
    int *first = &values[1];
    int *last  = &values[4];
    printf("  &values[4] - &values[1] = %td elements\n", last - first);

    puts("\n== three traversals, one answer ==");
```

```
    printf("  sum_indexed = %ld\n", sum_indexed(values, n));
    printf("  sum_pointer = %ld\n", sum_pointer(values, n));
    printf("  backwards:    ");
    print_backwards(values, n);

    puts("\n== pointer style where it belongs ==");
    const char *text = "pointer arithmetic";
    printf("  length_of(\"%s\") = %zu   strlen = %zu\n",
           text, length_of(text), strlen(text));

    puts("\n== array of pointers ==");
    const char *names[] = { "Ada", "Dennis", "Ken", "Bjarne" };
    const size_t names_count = sizeof names / sizeof names[0];
    printf("  the array holds %zu pointers, %zu bytes total\n",
           names_count, sizeof names);
    for (size_t i = 0; i < names_count; i++) {
        printf("    names[%zu] -> \"%s\" (%zu chars)\n",
               i, names[i], strlen(names[i]));
    }

    puts("\n== pointer to pointer as an output parameter ==");
    int *largest = NULL;
    point_at_largest(values, n, &largest);
    if (largest != NULL) {
        printf("  largest value %d is at index %td\n",
               *largest, largest - values);
    }

    puts("\n== one past the end ==");
    const int *end = values + n;
    printf("  values + %zu is a legal address to form: %p\n", n, (const void *)end);
    puts("  dereferencing it would be undefined behavior");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o ptrmath ptrmath.c
./ptrmath
```

Prove that index and pointer style compile the same

Do not take the claim on trust. Generate the assembly at `-O2` and compare the two functions:

```
gcc -std=c17 -O2 -S ptrmath.c -o ptrmath.s
sed -n '/sum_indexed:/,/ret/p' ptrmath.s
sed -n '/sum_pointer:/,/ret/p' ptrmath.s
```

On x86-64 with GCC the two bodies are typically instruction-for-instruction identical. The compiler recognizes both as the same loop and emits the same code. Choose the one that reads better, and let chapter 34 handle the cases where the generated code really does matter.

Break a rule on purpose

```
/* The reverse loop written the tempting way. */
for (const int *q = values + n - 1; q >= values; q--) {
    printf("%d ", *q);
}
```

This appears to work everywhere. It is still undefined: on the final iteration `q--` forms `values - 1`, an address before the array. The sanitizer will not report it and the program will not crash — which is precisely why knowing the rule matters more here than testing does. The `q->values` form in the example above never forms that address.

Now try one the sanitizer *will* catch:

```
printf("%d\n", *(values + n));    /* dereferencing one past the end */
```

```
==1234==ERROR: AddressSanitizer: stack-buffer-overflow
READ of size 4 at 0x7ffd... 'values' (line 96)
```

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Expecting `p + 1` to advance one byte	It advances `sizeof(*p)` bytes	Cast to `char *` if you really want bytes.
`for (q = v + n - 1; q >= v; q--)`	Forms `v - 1`: undefined	`for (q = v + n; q-- > v; )`.
Dereferencing one past the end	Out-of-bounds read	The address may be formed but never read.
Printing `ptrdiff_t` with `%d`	Format mismatch on 64-bit	`%td`.
Comparing pointers into different arrays with `<`	Undefined behavior	Only `==` and `!=` are meaningful across objects.
`int *a[5]` when you meant `int (*a)[5]`	An array of pointers, not a pointer to an array	Parenthesize; see chapter 16's reading rule.
Losing the original pointer by incrementing it	Cannot free or restart	Walk a copy, keep the original.
Believing pointer style is faster	Less readable code, identical output	Compare the assembly and stop worrying.

8 Check yourself

Why does `p + 1` advance four bytes for an `int *` but one byte for a `char *`?

Because pointer arithmetic counts elements, not bytes, and the pointer's type supplies the element size. This is what makes `p[i]` equivalent to `*(p + i)`: the scaling is exactly what indexing needs. To move by raw bytes, cast to `char *`, whose element size is 1 by definition.

What does subtracting two pointers give, and what type is it?

The number of elements between them, not bytes, with type `ptrdiff_t` from `<stddef.h>`. It is signed because the difference can be negative, which is why it is not `size_t`. Print it with `%td`. It is only defined for pointers into the same array.

Why is forming `values + n` legal but `values + n + 1` not?

The standard specifically permits a pointer one past the last element so that loop conditions such as `q < values + n` can be written. Anything beyond that — or before the first element — is undefined even if you never dereference it, because an implementation is allowed to trap on forming such an address.

What is the difference between `int *a[5]` and `int (*a)[5]`?

Brackets bind tighter than the asterisk, so the first is an array of five `int *` — five separate pointers. The parentheses in the second force the pointer to apply first, giving a single pointer to an array of five `int`. This is the declaration form used when passing a two-dimensional array to a function.

Is pointer-style traversal faster than index-style?

Not on any compiler released in the last twenty-five years. Strength reduction turns the indexed form into the same address arithmetic, and the generated assembly is typically identical — which you can confirm with `gcc -O2 -S`. Choose on readability: index style by default, pointer style for strings and buffers where there is no count to index against.

9 Where this leads

Chapter 16 turns pointers into a design tool rather than a mechanism: output parameters, `const` correctness as a contract the compiler enforces, the dangling-return trap, and a systematic method for reading declarations like `char *(*f[3])(int)` without guessing. After that, chapter 19 hands you memory that you must manage yourself.

INTERMEDIATE · CHAPTER 16

Pointers as Function Parameters

Pointers stop being a mechanism this chapter and become a design tool. How a function reports failure, what it promises not to modify, who owns the memory it returns — all of that is expressed in its parameter list, and the compiler will enforce most of it if you let it.

BY THE END OF THIS CHAPTER YOU CAN

- Design a function that returns a status and writes its result through an output parameter.
- Distinguish `const char *`, `char * const`, and `const char * const`, and use the right one.
- Apply `const` correctness as a discipline rather than decoration.
- Return a pointer safely, and recognize every way of doing it unsafely.
- Read any C declaration using the right-left rule, without guessing.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Output parameters and the status return

Chapter 11 listed three ways a C function can report failure. With pointers available, the most generally useful one can now be written properly: **return a status, write the result through a pointer**.

```
bool parse_int(const char *text, int *out);
```

Read the signature as a contract. The function reads `text` and will not modify it. It returns `true` on success, in which case it has written the parsed value to `*out`. On failure it returns `false`, and the caller must not rely on `*out`.

```
int value;
if (!parse_int(input, &value)) {
    fprintf(stderr, "not a number: %s\n", input);
    return EXIT_FAILURE;
}
/* value is trustworthy only inside this branch */
```

This is the convention to use when *every* value is a legitimate result. A function returning a temperature cannot use −1 as a failure code, and cannot use 0 either. Separating "did it work" from "what is the answer" removes the problem entirely.

Rules that make output parameters pleasant

- Put output parameters last, after the inputs. This is what the standard library does and what readers expect.
- Do not write to `*out` on the failure path. Leaving the caller's variable untouched is easier to reason about than partially filling it.
- Document whether `out` may be `NULL`. If it may, check it. If it may not, say so — and consider `assert`, which chapter 27 introduces.
- One output parameter is fine; three is a signal that the results belong together in a structure, which is chapter 21.

2 `const` and what it qualifies

With pointers there are two things that could be constant — the pointer, or what it points to — and C can express either.

DECLARATION	CAN CHANGE THE POINTER?	CAN CHANGE THE TARGET?
`char *p`	yes	yes
`const char *p`	yes	**no**
`char * const p`	**no**	yes
`const char * const p`	no	no

The rule for reading these: `const` applies to whatever is immediately to its left, unless there is nothing to its left, in which case it applies to what is on its right. So `const char *p` and `char const *p` mean the same thing — a pointer to constant characters.

```
const char *p = "hello";
p[0] = 'H';        /* error: target is const   */
p = "world";       /* fine: the pointer is not */

char * const q = buffer;
q[0] = 'H';        /* fine: target is writable */
q = other;         /* error: the pointer is const */
```

By far the most common and most useful is `const char *`, and more generally `const T *` for any parameter the function only reads.

`const` correctness as a discipline

The rule is simple: **every pointer parameter that the function does not modify should be `const`**. The benefits compound.

It *documents* the interface. `void process(const char *in, char *out)` tells you which way data flows before you read a line of the body.

It is *enforced*. An accidental write is a compile error, not a bug. Contrast with a comment saying "does not modify src", which is unchecked and rots.

It *propagates*. A caller holding a `const char *` cannot pass it to a function taking `char *`. If you omit `const` in one place, callers are forced to omit it too, or to cast it away — and a codebase with casts scattered through it has lost the benefit entirely. Add `const` from the start; retrofitting it is genuinely painful.

> **Casting `const` away.** `char *bad = (char *)some_const_pointer;` compiles. If the object really is constant — a string literal, for instance — writing through the result is undefined behavior and typically crashes. A cast that removes `const` is a claim that you know the object is actually writable; if you cannot justify that claim, the cast is a bug.

3 Returning a pointer

A function returning a pointer is making a promise about the lifetime of what it points to. There are exactly four sources, and one of them is always wrong.

RETURNED POINTER REFERS TO	SAFE?	WHO FREES IT
A local variable	**No** — the frame is gone	—
A string literal or other static object	Yes	Nobody; it lives forever
Memory the caller supplied	Yes	The caller, who already owns it
Memory from `malloc`	Yes	The caller — and this must be documented

```
/* WRONG: the array dies when the function returns */
char *broken(void)
{
    char buffer[32];
    strcpy(buffer, "hello");
    return buffer;
}

/* Fine: the literal has static storage duration */
const char *describe(int code)
{
    return (code == 0) ? "success" : "failure";
}

/* Fine, and the most common good design: the caller supplies the space */
bool format_into(char *dst, size_t dst_size, int code)
{
    int n = snprintf(dst, dst_size, "code %d", code);
    return n >= 0 && (size_t)n < dst_size;
}

/* Fine, but the caller must free it — say so in the name or a comment */
char *duplicate(const char *s);   /* caller frees the result */
```

The third form is worth adopting as a default. It puts the caller in control of where the memory comes from, it cannot leak, and it works identically whether the buffer is on the stack, in a static, or on the heap. Chapter 19 covers the fourth, where ownership becomes an explicit part of every interface.

4 Passing strings and multidimensional arrays

A string parameter is a `char *`, plus `const` if it is only read:

```
size_t count_vowels(const char *s);           /* reads only        */
void   to_upper_in_place(char *s);            /* modifies in place */
bool   copy_into(char *dst, size_t dst_size,
                 const char *src);            /* both directions   */
```

Note the pattern in the third: a writable buffer is always accompanied by its size. There is no way to recover it inside the function — chapter 12 established that `sizeof` on a parameter gives the pointer size — so the caller must supply it. A function that takes a destination buffer and no size cannot be used safely, which is precisely the defect in `strcpy`.

Two-dimensional arrays

```
void print_grid(size_t rows, size_t cols, const int grid[rows][cols]);
```

This C99 form takes the dimensions as earlier parameters and uses them in the array type, which lets the compiler compute offsets correctly for any shape. It is clearer than the older alternatives:

```
void print_grid(const int grid[][COLS], size_t rows);   /* fixed width  */
void print_grid(const int (*grid)[COLS], size_t rows);  /* the same     */
void print_grid(const int *flat, size_t rows, size_t cols); /* manual   */
```

The last form — one-dimensional with `flat[r * cols + c]` — is what most real code uses, because it works for dynamically allocated matrices where the shape is not known until run time. Chapter 19 builds that.

5 Reading declarations: the right-left rule

C declarations are notoriously hard to read because the syntax mirrors *use* rather than structure. There is a mechanical procedure, and once you know it nothing is ambiguous.

> **The rule.** Start at the identifier. Move right as far as you can, then left, alternating, obeying parentheses. Read `[]` as "array of", `()` as "function returning", and `*` as "pointer to".

Worked through, from easy to unpleasant:

DECLARATION	READS AS
`int *p`	`p` is a pointer to `int`
`int *p[5]`	`p` is an array of 5 pointers to `int`
`int (*p)[5]`	`p` is a pointer to an array of 5 `int`
`int *f(void)`	`f` is a function returning a pointer to `int`
`int (*f)(void)`	`f` is a pointer to a function returning `int`
`char *(*f[3])(int)`	`f` is an array of 3 pointers to functions taking `int` and returning a pointer to `char`

Take the last one step by step. Start at `f`. To the right is `[3]` — array of 3. Parentheses close, so go left: `*` — of pointers to. Now outside the parentheses, right again: `(int)` — functions taking `int`. Then left: `char *` — returning a pointer to `char`. Done, and nothing was guessed.

Two practical notes. The `cdecl` tool, available online and as a package, translates declarations in both directions and is worth using while the rule is still new. And in your own code, a `typedef` usually beats a virtuoso declaration:

```
typedef char *(*string_maker)(int);
string_maker f[3];          /* the same type, comprehensible */
```

Chapter 30 needs this for function pointers, and chapter 22 covers `typedef` properly.

6 Worked example: a parser with an honest contract

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>
#include <errno.h>
#include <limits.h>

/* Parse a complete decimal integer.
   Returns true and writes *out only on success.
   text is read and never modified, which the signature enforces. */
static bool parse_int(const char *text, int *out)
```

```
{
    if (text == NULL || out == NULL) {
        return false;
    }

    while (isspace((unsigned char)*text)) {
        text++;
    }
    if (*text == '\0') {
        return false;                       /* empty or all whitespace */
    }

    errno = 0;
    char *end = NULL;                       /* strtol writes here */
    long value = strtol(text, &end, 10);

    if (end == text)      return false;     /* no digits at all      */
    if (*end != '\0')     return false;     /* trailing rubbish      */
    if (errno == ERANGE)  return false;     /* out of long's range   */
    if (value < INT_MIN || value > INT_MAX) return false;

    *out = (int)value;
    return true;
}

/* Two results, so two output parameters — and at three it would be a struct. */
static bool min_max(const int *values, size_t n, int *lo, int *hi)
{
    if (values == NULL || n == 0) {
        return false;
    }
    *lo = *hi = values[0];
    for (size_t i = 1; i < n; i++) {
        if (values[i] < *lo) *lo = values[i];
        if (values[i] > *hi) *hi = values[i];
    }
    return true;
}
```

```
/* Caller supplies the buffer and its size: cannot leak, cannot overflow. */
static bool describe_into(char *dst, size_t dst_size, int lo, int hi)
{
    int n = snprintf(dst, dst_size, "range %d..%d (span %d)", lo, hi, hi - lo);
    return n >= 0 && (size_t)n < dst_size;
}

/* Safe: the literals have static storage duration. */
static const char *verdict(int span)
{
    if (span == 0)  return "all equal";
    if (span < 10)  return "tight";
    if (span < 100) return "moderate";
    return "wide";
}

/* const correctness: this reads, so every pointer it takes is const. */
static size_t count_matching(const int *values, size_t n, int target)
{
    size_t found = 0;
    for (size_t i = 0; i < n; i++) {
        if (values[i] == target) {
            found++;
        }
    }
    return found;
}

int main(void)
{
    puts("== parsing, with failure reported rather than guessed ==");
    const char *inputs[] = { "42", "  -7 ", "abc", "", "12abc", "99999999999" };
    const size_t input_count = sizeof inputs / sizeof inputs[0];

    int parsed[8];
    size_t good = 0;

    for (size_t i = 0; i < input_count; i++) {
        int value;
```

```
        if (parse_int(inputs[i], &value)) {
            printf("  \"%s\" -> %d\n", inputs[i], value);
            parsed[good++] = value;
        } else {
            printf("  \"%s\" -> rejected\n", inputs[i]);
        }
    }

    puts("\n== two outputs ==");
    int lo, hi;
    if (min_max(parsed, good, &lo, &hi)) {
        printf("  lowest %d, highest %d\n", lo, hi);

        char text[64];
        if (describe_into(text, sizeof text, lo, hi)) {
            printf("  %s — %s\n", text, verdict(hi - lo));
        }

        char tiny[8];
        if (!describe_into(tiny, sizeof tiny, lo, hi)) {
            printf("  into an 8-byte buffer: truncated to \"%s\", and detected\n",
                   tiny);
        }
    }

    puts("\n== the empty case is a failure, not a wrong answer ==");
    printf("  min_max on 0 values -> %s\n",
           min_max(parsed, 0, &lo, &hi) ? "succeeded?!" : "correctly refused");

    puts("\n== const correctness ==");
    printf("  count of 42 = %zu\n", count_matching(parsed, good, 42));
    puts("  count_matching takes const int * — the compiler guarantees");
    puts("  it cannot modify the caller's array, and says so in the signature");

    puts("\n== declarations, read with the right-left rule ==");
    int   n1 = 5;
    int  *p1 = &n1;             /* p1: pointer to int                    */
    int  *a1[3] = { &n1, NULL, NULL };  /* a1: array of 3 pointers to int */
    int   m[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
    int (*r1)[3] = m;           /* r1: pointer to array of 3 int        */

    printf("  *p1      = %d\n", *p1);
    printf("  *a1[0]   = %d\n", *a1[0]);
    printf("  r1[1][2] = %d\n", r1[1][2]);
    printf("  sizeof a1 = %zu (3 pointers), sizeof *r1 = %zu (3 ints)\n",
           sizeof a1, sizeof *r1);

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o params params.c
./params
```

Why `parse_int` is written that way

Compare it with `atoi`, which is what most textbooks reach for. `atoi("abc")` returns 0 — indistinguishable from `atoi("0")`. It cannot report overflow. It has no way to tell you there was trailing garbage. Every one of those is a real defect, and `strtol` with a checked `endptr` and `errno` closes all three.

The `char *end` argument is itself an output parameter, written by `strtol` to say where it stopped. Checking `end == text` detects "no digits"; checking `*end != '\0'` detects trailing rubbish. That pattern is worth memorizing — chapter 23 uses it for command-line arguments and chapter 48 fuzzes it.

Make the compiler enforce `const`

Add a write inside a function that promised not to:

```
static size_t count_matching(const int *values, size_t n, int target)
{
    values[0] = 0;        /* add this line */
```

```
params.c:82:15: error: assignment of read-only location '*values'
```

Not a warning — an error. The contract in the signature is checked by the compiler on every build, for free, forever. That is what distinguishes `const` from a comment.

Watch a dangling return fail

```
static char *broken(void)
{
    char buffer[32];
    strcpy(buffer, "hello");
    return buffer;
}
/* in main: */
printf("%s\n", broken());
```

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

Run it under the sanitizer with `ASAN_OPTIONS=detect_stack_use_after_return=1` (GCC leaves this check off by default) and the diagnosis is exact:

```
==1234==ERROR: AddressSanitizer: stack-use-after-return
```

Then fix it the right way — by making the caller supply the buffer, as `describe_into` does — and note that the fix changed the interface, not the body. Lifetime is an interface question.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Using a sentinel like −1 where every value is valid	Failure indistinguishable from a result	Status return plus an output parameter.
Writing to `*out` before validating	Caller's variable clobbered on failure	Write only on the success path.
Omitting `const` on read-only parameters	Contract unstated; `const` callers forced to cast	Add it from the start; retrofitting is painful.
Casting `const` away to silence an error	Possible write to read-only memory	Fix the signature instead.
Returning a pointer to a local	Dangling pointer	Caller-supplied buffer, static data, or `malloc` with documented ownership.
A destination buffer with no size parameter	Cannot be used safely — the `strcpy` defect	Always pass the size alongside.
Using `atoi` for user input	Cannot distinguish 0 from failure	`strtol` with `endptr` and `errno` checked.
Guessing at a declaration's meaning	Wrong type, confusing errors	Apply the right-left rule, or introduce a `typedef`.

8 Check yourself

When is a status return plus an output parameter better than returning the value directly?

Whenever every possible value is a legitimate result, so no value is available as a failure sentinel. Returning a temperature, a parsed integer, or a count that may legitimately be zero all fall into this category. Separating "did it succeed" from "what is the answer" also forces the caller to handle failure before touching the result.

What is the difference between `const char *p` and `char * const p`?

The first is a pointer to constant characters: you may repoint `p` but not write through it. The second is a constant pointer to writable characters: you may write through it but not repoint it. Read `const` as qualifying whatever is immediately to its left, or to its right when there is nothing to the left.

Why is `const` better than a comment saying the function does not modify its argument?

Because it is checked. An accidental write becomes a compile error rather than a bug, and the guarantee propagates to callers, who can pass `const` data without casting. A comment is unverified and drifts out of date; `const` is re-verified on every build.

Which of the four sources of a returned pointer is always wrong, and why?

A pointer to a local variable. The local lives in the function's stack frame, which is discarded on return, so the caller receives an address into memory that now belongs to whatever is called next. It frequently appears to work, because the bytes have not yet been overwritten, which makes it harder to find rather than easier.

Read `char *(*f[3])(int)`.

Start at `f`, go right to `[3]` — an array of 3. The parenthesis closes, so go left to `*` — of pointers to. Outside the parentheses, right to `(int)` — functions taking an `int`. Then left to `char *` — returning a pointer to `char`. So: an array of three pointers to functions taking an `int` and returning `char *`. A `typedef` would express this far more kindly.

9 Where this leads

Chapter 17 turns from parameters to storage: where variables actually live, how long they last, and what the program's memory map looks like as a single picture. That picture makes chapter 18's recursion visible in a debugger and is the prerequisite for chapter 19, where you take memory from the heap and become responsible for giving it back.

INTERMEDIATE · CHAPTER 17

Scope, Lifetime, and Program Memory Layout

Two different questions get confused constantly: where a name can be seen, and how long the object it names exists. C answers them separately, and the answers together explain the whole memory map of a running program.

BY THE END OF THIS CHAPTER YOU CAN

- Distinguish scope from storage duration and give an example where they differ.
- Explain what a `static` local is and when it is the right tool.
- Name the six regions of a running program's memory and say what lives in each.
- Describe a stack frame and explain why locals do not survive a return.
- Write a program that prints its own memory map.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Scope: where a name is visible

SCOPE	EXTENT	INTRODUCED BY
Block	From the declaration to the closing brace	Any declaration inside `{ }`
Function	The whole function body	Only labels, for `goto`
File	From the declaration to the end of the file	Declarations outside any function
Function prototype	Within the parameter list	Parameter names in a declaration

```
int global = 1;              /* file scope: visible below, in this file */

void demo(void)
{
    int local = 2;           /* block scope: this function only */

    {
        int inner = 3;       /* block scope: these braces only  */
        printf("%d\n", inner);
    }
    /* inner does not exist here */
}
```

Declare names in the smallest scope that works. A variable visible only where it is used cannot be accidentally read or written elsewhere, and the reader does not have to search for other uses.

Shadowing

```
int count = 10;

void demo(void)
{
    int count = 20;          /* shadows the file-scope count */
    printf("%d\n", count);   /* 20 — the inner one wins      */
}
```

An inner declaration hides an outer one of the same name. This is legal and occasionally intentional, but far more often it is an accident that produces a bug the compiler will not mention unless asked. Turn it on:

```
gcc -Wshadow …
```

`-Wshadow` is not in `-Wall` or `-Wextra`. It is worth adding to your standard command line.

2 Storage duration: how long an object lives

DURATION	LIVES	DECLARED AS
Automatic	From entry to its block until exit	A plain local variable
Static	The whole program run	`static`, or any file-scope variable
Allocated	From `malloc` until `free`	Chapter 19
Thread	The lifetime of one thread	`_Thread_local` — chapter 41

Scope and duration are independent, which is the point of separating them. A `static` local has *block scope* — only its function can name it — and *static duration* — it exists for the whole run and keeps its value between calls.

```
void counter(void)
{
    static int calls = 0;    /* initialized once, at program start */
    calls++;
    printf("call number %d\n", calls);
}
/* 1, 2, 3, … across separate calls */
```

Two guarantees worth knowing. A `static` variable's initializer must be a constant expression and runs once, before `main`. And any static object without an initializer is **zero-initialized** — unlike an automatic variable, which contains garbage.

When a static local is right

For a counter or cache private to one function, it is exactly the tool: file-scope visibility would expose it unnecessarily. The cost is that the function is no longer reentrant — two threads calling it share the variable, and so do two nested calls. `strtok` from chapter 14 is the standard library's example of this design and its consequences.

`static` at file scope means something else

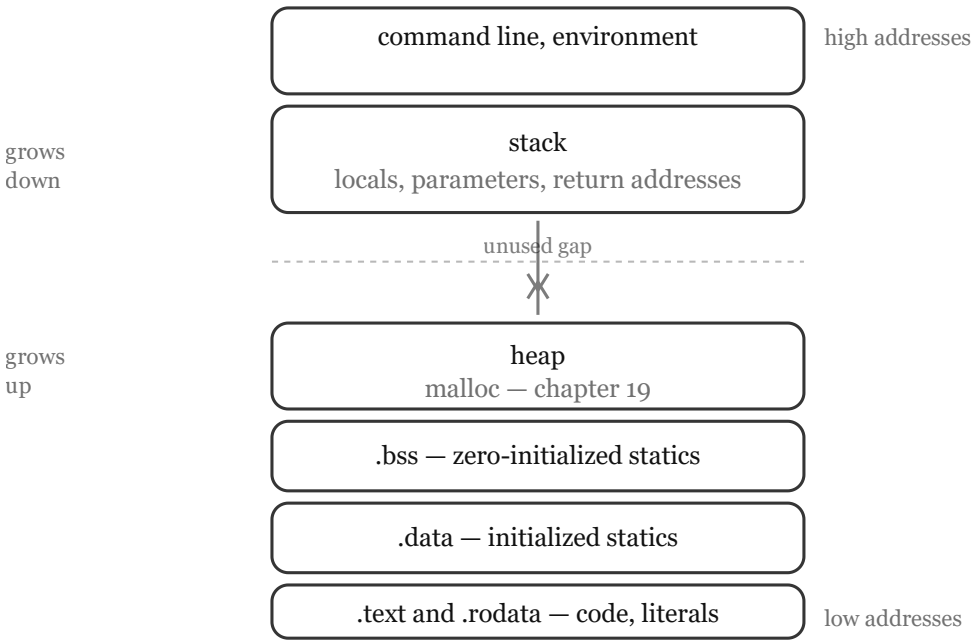
```
static int helper_count;         /* not visible to other .c files */
static void helper(void) { … }   /* likewise */
```

Applied to a file-scope name, `static` restricts *linkage* rather than duration — the name cannot be referenced from another translation unit. That is chapter 28's subject, and it is the

basis of information hiding in C.

3 The memory map of a running program

When the operating system loads your executable, it lays out several distinct regions. Every object you have declared so far lives in one of them.



The classic layout. Chapter 52 inspects the same sections in a firmware image with `objdump`.

REGION	HOLDS	WRITABLE	LIFETIME
`.text`	Machine code	No	Whole run
`.rodata`	String literals, `const` data	No	Whole run
`.data`	Statics with a non-zero initializer	Yes	Whole run
`.bss`	Statics initialized to zero	Yes	Whole run
Heap	`malloc`'d blocks	Yes	Until `free`
Stack	Locals, parameters, return addresses	Yes	Until the function returns

`.bss` deserves a note. Zero-initialized statics are not stored in the executable file at all — only their total size is recorded, and the loader zeroes that much memory at startup. A program with a ten-megabyte zeroed array has a tiny executable. Chapter 52 makes this visible with the `size` command.

This map also explains chapter 14's crash. A string literal lives in `.rodata`, which the hardware marks read-only; writing there is refused by the memory management unit, not by C.

4 The call stack

Each function call pushes a **stack frame** containing its parameters, its local variables, and the address to return to. The frame is popped on return.

```
void inner(int x)
{
    int local = x * 2;      /* lives in inner's frame */
    printf("%d\n", local);
}                           /* frame popped here      */

void outer(void)
{
    int value = 21;         /* lives in outer's frame */
    inner(value);           /* inner's frame sits above outer's */
}
```

Three consequences, all of which you have already met.

Locals do not survive the return — chapter 11's rule and chapter 13's dangling pointer. The memory is not erased; it simply belongs to the next call, which is why a use-after-return often appears to work.

Recursion consumes stack. Each call is another frame. Chapter 18 measures how many fit.

The stack is small and fixed. Typically 8 MB on Linux, 1 MB on Windows. A large local array can exhaust it on its own:

```
void reckless(void)
{
    int huge[4000000];      /* 16 MB on the stack — crashes */
    huge[0] = 1;
}
```

There is no error message for this, only a segmentation fault. Anything large belongs on the heap, which is chapter 19. Check your limit with `ulimit -s`.

> **Stack overflow is not reported gracefully.** The stack grows downward into an unmapped guard page and the program receives a fatal signal. There is no `NULL` to check and no error code. Sanitizers detect it — `-fsanitize=address` reports *stack-overflow* — but the language itself gives you nothing.

5 The cost of global state

A file-scope variable is visible to every function below it, which sounds convenient and is the reason it is overused.

```
int total = 0;                    /* everyone can read and write this */


void add(int n)   { total += n; }
void reset(void)  { total = 0;  }
int  report(void) { return total; }
```

What it costs:

- **Reasoning.** To know what `report` returns you must know every call to `add` and `reset` that has happened. A function taking parameters and returning a value can be understood alone.
- **Testing.** Tests must reset the state between cases, and they cannot run in parallel. Chapter 35 hits this immediately.
- **Reuse.** There is exactly one `total`. Handling two independent sums requires rewriting all three functions.
- **Threads.** Two threads calling `add` concurrently corrupt the value. Chapter 41.

The alternative is to pass state explicitly — a parameter now, a pointer to a structure from chapter 21 onward:

```
typedef struct { int total; } Accumulator;


void add(Accumulator *a, int n)   { a->total += n; }
void reset(Accumulator *a)        { a->total = 0;  }
int  report(const Accumulator *a) { return a->total; }
```

Longer to write and better in every other respect: independent instances, testable in isolation, thread-safe by construction, and the `const` says which operations are read-only. Chapter 33 makes this an opaque handle.

Where a genuinely global resource exists — a log file, a configuration loaded once — make it `static` at file scope so at least the name does not leak into other translation units.

6 Worked example: a program that prints its own memory map

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

/* .data — initialized, writable, whole-run lifetime */
int    global_initialized = 42;
double global_double      = 3.14;

/* .bss — zero-initialized; costs no space in the executable */
int    global_zero;
int    big_zero_array[100000];

/* .rodata — read-only */
const char *const message = "stored in read-only memory";
const int  lookup[4] = { 1, 2, 3, 4 };

/* file scope but not exported: chapter 28 */
static int file_private = 7;

/* A static local: block scope, static duration. */
static int call_count(void)
{
    static int calls = 0;       /* initialized once, before main */
    return ++calls;
}

/* Demonstrates that each frame is distinct and reused. */
static void frame_depth(int depth)
{
    int marker = depth;
    printf("    depth %d: &marker = %p\n", depth, (void *)&marker);
    if (depth < 3) {
        frame_depth(depth + 1);
```

```
    }
}

static void shadowing_demo(void)
{
    int file_private = 999;     /* shadows the file-scope one */
    printf("  inside the function, file_private = %d\n", file_private);
    printf("  (the file-scope one is still 7, but unreachable by name here)\n");
}

int main(void)
{
    int    local          = 1;
    char   local_array[64] = "on the stack";
    void  *heap_block      = malloc(1024);

    if (heap_block == NULL) {
        fprintf(stderr, "allocation failed\n");
        return EXIT_FAILURE;
    }

    puts("== where things live (addresses vary between runs) ==\n");
    printf("  %-26s %p\n", ".text  (code: main)",      (void *)(void *)main);
    printf("  %-26s %p\n", ".rodata (string literal)", (void *)message);
    printf("  %-26s %p\n", ".rodata (const array)",    (void *)lookup);
    printf("  %-26s %p\n", ".data  (global = 42)",     (void *)&global_initialized);
    printf("  %-26s %p\n", ".data  (static file var)", (void *)&file_private);
    printf("  %-26s %p\n", ".bss   (global zero)",     (void *)&global_zero);
    printf("  %-26s %p\n", ".bss   (big zero array)",  (void *)big_zero_array);
    printf("  %-26s %p\n", "heap   (malloc 1024)",     heap_block);
    printf("  %-26s %p\n", "stack  (local int)",       (void *)&local);
    printf("  %-26s %p\n", "stack  (local array)",     (void *)local_array);

    puts("\n  read the addresses: code and constants lowest, then the");
    puts("  writable statics, then the heap, with the stack far above");

    puts("\n== automatic versus static duration ==");
    printf("  call_count() -> %d\n", call_count());
    printf("  call_count() -> %d\n", call_count());
```

```
    printf("  call_count() -> %d   (the value persisted between calls)\n",
           call_count());

    puts("\n== zero-initialization ==");
    printf("  global_zero (static, no initializer) = %d  -- guaranteed 0\n",
           global_zero);
    puts("  an uninitialized LOCAL would hold garbage instead");

    puts("\n== stack frames ==");
    printf("  main's local is at      %p\n", (void *)&local);
    frame_depth(1);
    puts("  each frame sits at a lower address; they are popped on return");

    puts("\n== scope and shadowing ==");
    printf("  at file scope, file_private = %d\n", file_private);
    shadowing_demo();

    puts("\n== read-only memory ==");
    printf("  message = \"%s\"\n", message);
    puts("  writing through it would be refused by the hardware");

    free(heap_block);
    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -Wshadow -g -o layout layout.c
./layout
```

What the compiler adds

Because `-Wshadow` is on:

```
layout.c:52:9: warning: declaration of 'file_private' shadows
              a global declaration [-Wshadow]
```

That warning is off by default and finds a real class of bug. Add it permanently.

Confirm the sections from outside the program

```
size layout
```

```
   text      data       bss       dec       hex   filename
   3521       664    400064    404249     62b19   layout
```

The `bss` figure is roughly 400 KB — the 100 000-element zeroed array — yet the executable on disk is only a few kilobytes. Zeroed statics are recorded as a size, not as content. Delete `big_zero_array`, rebuild, and watch `bss` collapse.

```
nm layout | grep -E ' [BbDdRrTt] ' | sort
```

The letter in `nm`'s output is the section: `T` for text, `D` for data, `B` for bss, `R` for read-only. You can see each variable land exactly where the table in section 3 predicted.

Exhaust the stack on purpose

```
static void recurse(int depth)
{
    char padding[1024];
    padding[0] = (char)depth;
    printf("\rdepth %d", depth);
    recurse(depth + 1);
}
```

```
ulimit -s            # your stack limit in kilobytes
./layout             # add a call to recurse(1)
```

It prints an increasing depth and then dies with a segmentation fault — no message, no chance to recover. Each frame holds a 1 KB array plus a little overhead, so the depth at which it fails should be a little below your stack limit in kilobytes. Chapter 18 does this arithmetic deliberately.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Returning a pointer to a local	Dangling pointer once the frame pops	Caller-supplied buffer, static, or heap.
Expecting an uninitialized local to be 0	Garbage; statics are zeroed, locals are not	Initialize every local at declaration.
A large array as a local	Stack overflow, no diagnostic	Use the heap — chapter 19.
Shadowing a variable accidentally	The wrong object is modified	Compile with `-Wshadow`.
Globals instead of parameters	Untestable, unreusable, thread-unsafe	Pass state explicitly; group it in a struct.
A static local in a function called from threads	Shared state, data race	Make the state a parameter — chapter 41.
Writing through a pointer to a literal	Crash: `.rodata` is read-only	Declare it `const char *`.
Assuming a static initializer runs at first call	It runs once, before `main`	Use a flag if you need lazy initialization.

8 Check yourself

Give an example where scope and storage duration differ.

A `static` local variable. Its scope is the block it is declared in — no other function can name it — but its storage duration is the entire program run, so it retains its value between calls. Scope is about visibility of the name; duration is about lifetime of the object.

Why is a static variable guaranteed to be zero when an uninitialized local is not?

Statics are allocated once at program start, and the loader zeroes the `.bss` region before `main` runs. A local is created by adjusting the stack pointer on each call, which does not clear the memory; whatever the previous call left there remains.

Why does a ten-megabyte zeroed global array barely increase the executable's size?

Because it goes in `.bss`, which records only a size. There is nothing to store — every byte is zero — so the loader simply reserves and clears that much memory at startup. Give the same array a non-zero initializer and it moves to `.data`, where the contents must actually be in the file.

What happens when the stack runs out, and how would you notice?

The stack grows into an unmapped guard page and the program receives a fatal signal — a segmentation fault with no message and no opportunity to recover. There is no return code to check. The usual causes are unbounded recursion and very large local arrays; both are fixed by moving the data to the heap or bounding the depth.

Name two concrete costs of using a file-scope variable instead of a parameter.

Any two of: you cannot understand a function without tracing every other function that touches the variable; tests must reset shared state and cannot run in parallel; only one instance can ever exist, so the code cannot be reused for two independent sets of data; and concurrent access from threads corrupts it. Passing state explicitly removes all four.

9 Where this leads

You can now picture where every object lives. Chapter 18 uses the stack half of that picture directly: recursion is the construct that makes frames visible, and you will watch them accumulate and unwind in a debugger. Chapter 19 then uses the heap half, where lifetime stops being automatic and becomes your responsibility.

Recursion

A function that calls itself is not a trick. It is the natural shape for any problem whose structure contains a smaller copy of itself — and it is the construct that makes chapter 17's stack frames visible, because you can watch them pile up.

BY THE END OF THIS CHAPTER YOU CAN

- Write a recursive function with a correct base case and a case that provably shrinks.
- Trace a recursive call in a debugger and read the resulting backtrace.
- Explain why naive Fibonacci is exponential and fix it two ways.
- Calculate roughly how deep your recursion can go before the stack runs out.
- Decide between a recursive and an iterative formulation on grounds other than taste.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The two obligations

Every correct recursive function satisfies two conditions. Miss either and it does not terminate.

A base case that returns without recursing. It answers the smallest version of the problem directly.

A recursive case that calls itself on input strictly closer to the base case. "Smaller" must be measurable and must be guaranteed to decrease, or the recursion is infinite.

```
long factorial(int n)
{
    if (n <= 1) {
        return 1;              /* base case */
    }
    return n * factorial(n - 1);   /* n decreases, so it must reach 1 */
}
```

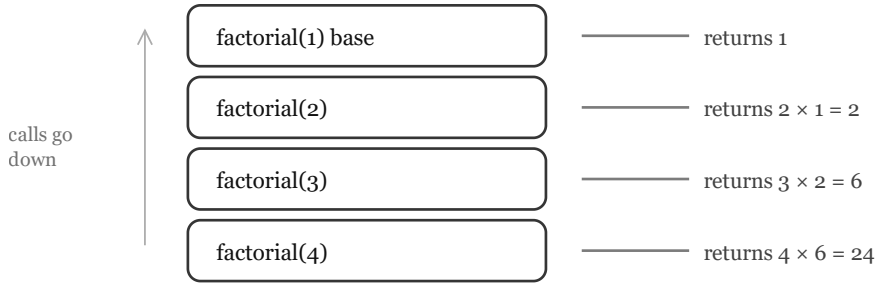
Note that the base case is `n <= 1` rather than `n == 1`. Testing for exact equality is the most common way to write an infinite recursion: call `factorial(-1)` and a function guarded by `n`

`== 1` descends forever. Guard the whole range you cannot handle, not the single value you expect.

> **The termination argument.** Identify a quantity that strictly decreases on every recursive call and is bounded below — here, `n`. This is the same obligation chapter 10 imposed on loops, and it is what "provably shrinks" means. If you cannot name that quantity, you have not shown the function terminates.

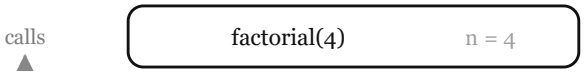
2 What happens on the stack

Chapter 17 established that each call pushes a frame. Recursion pushes one per level, and they all exist simultaneously until the base case is reached.

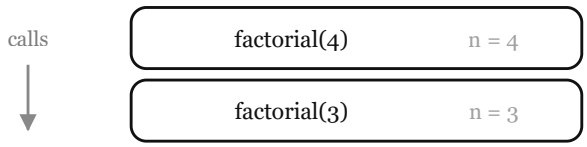


The multiplications happen on the way *out*, as the stack unwinds.

That last point is worth dwelling on. Nothing is multiplied on the way down; each frame is suspended holding its own `n`, waiting for the answer from below. The work happens during unwinding, which is why the memory cost is proportional to the depth.



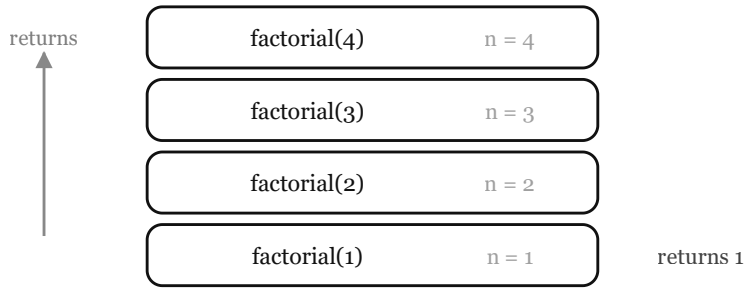
`factorial(4)` is called. Its frame holds `n` and the address to return to, and then it waits.



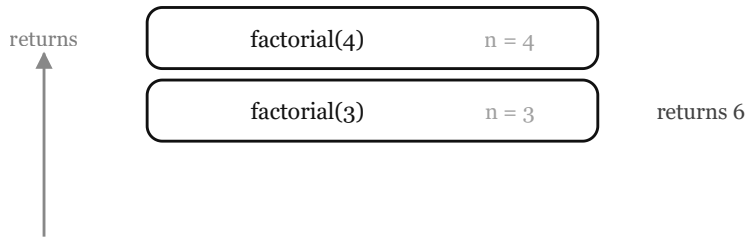
It calls `factorial(3)`. Still nothing has been multiplied — the first frame is suspended, holding its own `n`.



All four frames exist at once. This is the memory cost of recursion: it is proportional to the *depth*, and it is paid before any answer comes back.



`factorial(1)` is the base case. It returns 1 without calling anything, and its frame is popped.



Now the work starts. `2 × 1 = 2`, then `3 × 2 = 6`. The multiplications happen on the way *out*.



`4 × 6 = 24` and the last frame goes. Nothing was computed on the way down; everything was computed while unwinding.

3 Depth and stack overflow

Each frame costs memory: the parameters, the locals, the saved return address, and alignment padding. Call it a few dozen bytes for a simple function, more if it has a large local array.

```
ulimit -s          # stack limit in kilobytes; typically 8192 on Linux
```

With 8 MB of stack and roughly 48 bytes per frame, a simple recursion can reach somewhere near 150 000 levels. Add a 1 KB local array to the function and that collapses to about 8 000.

There is no graceful failure. The stack grows into a guard page and the program receives a fatal signal — no `NULL` to check, no error to catch. Sanitizers report it clearly:

```
gcc -fsanitize=address …
==1234==ERROR: AddressSanitizer: stack-overflow on address 0x7ffc...
```

Practical consequence: **recursion depth must be bounded by something you control**. Recursing over a balanced tree of a million nodes is fine — depth about 20. Recursing once per element of a million-element list is not.

Tail calls

A *tail call* is a recursive call whose result is returned directly, with nothing left to do afterwards:

```
/* not a tail call: the multiplication happens after the call returns */
return n * factorial(n - 1);

/* tail call: nothing remains after the call */
return factorial_helper(n - 1, accumulator * n);
```

A compiler *may* turn a tail call into a jump, reusing the current frame instead of pushing a new one — which makes the recursion run in constant stack space. GCC and Clang do this at `-O2`. But the C standard does not require it, so you cannot rely on it for correctness. Treat it as an optimization you may benefit from, never as a guarantee.

4 When recursion is the right shape

PROBLEM	RECURSIVE?	WHY
Sum an array	No	A loop is simpler and uses no stack
Factorial	No	Trivially iterative; recursion is only a teaching example
Binary search	Either	Depth is log n; both forms read well
Towers of Hanoi	Yes	The definition is recursive; the iterative version is obscure
Tree traversal	Yes	A tree contains subtrees — chapter 33
Directory walking	Yes	Same structure: directories contain directories
Quicksort, mergesort	Yes	Divide and conquer; depth is log n (for quicksort, only on average)
Parsing nested expressions	Yes	The grammar is recursive

The pattern: use recursion when the *data* or the *definition* is recursive, and the depth is logarithmic or otherwise bounded. Use iteration for linear sequences.

The Fibonacci trap

```
long fib(int n)
{
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);      /* two calls per level */
}
```

This is the textbook example of recursion and a genuinely terrible program. Each call spawns two more, so the work grows exponentially: `fib(40)` makes over 300 million calls and takes seconds. `fib(50)` would take minutes.

The problem is not recursion but *repeated* recursion — `fib(10)` is computed over a million times. Two fixes:

```
/* memoize: remember what you already computed */
long fib_memo(int n, long cache[])
{
    if (n <= 1)        return n;
    if (cache[n] != 0) return cache[n];
    cache[n] = fib_memo(n - 1, cache) + fib_memo(n - 2, cache);
    return cache[n];
}

/* or simply iterate: O(n) time, O(1) space */
long fib_iter(int n)
{
    long a = 0, b = 1;
    for (int i = 0; i < n; i++) {
        long next = a + b;
        a = b;
        b = next;
    }
    return a;
}
```

Both are linear. The iterative version is the one to ship.

5 Worked example: Hanoi, search, and the stack under a debugger

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

static long call_counter = 0;

/* --- factorial, to watch frames unwind --- */
static long factorial(int n, int depth)
{
    printf("%*scall  factorial(%d)\n", depth * 2, "", n);
    long result;
    if (n <= 1) {
        result = 1;
    } else {
        result = n * factorial(n - 1, depth + 1);
    }
    printf("%*sreturn %ld\n", depth * 2, "", result);
    return result;
}

/* --- Towers of Hanoi: the definition is itself recursive --- */
static void hanoi(int disks, char from, char to, char via, int *moves)
{
    if (disks == 0) {
        return;
    }
    hanoi(disks - 1, from, via, to, moves);
    (*moves)++;
    if (disks <= 3) {                    /* print only the small cases */
        printf("    move disk %d: %c -> %c\n", disks, from, to);
    }
    hanoi(disks - 1, via, to, from, moves);
}
```

```
/* --- binary search, recursive and iterative --- */
static long bsearch_rec(const int a[], long lo, long hi, int target)
{
    if (lo > hi) {
        return -1;                      /* base case: empty range */
    }
    long mid = lo + (hi - lo) / 2;      /* avoids overflow; chapter 7 */
    if (a[mid] == target) return mid;
    if (a[mid] <  target) return bsearch_rec(a, mid + 1, hi, target);
    return bsearch_rec(a, lo, mid - 1, target);
}

static long bsearch_iter(const int a[], long n, int target)
{
    long lo = 0, hi = n - 1;
    while (lo <= hi) {
        long mid = lo + (hi - lo) / 2;
        if (a[mid] == target) return mid;
        if (a[mid] <  target) lo = mid + 1;
        else                  hi = mid - 1;
    }
    return -1;
}

/* --- Fibonacci three ways --- */
static long fib_naive(int n)
{
    call_counter++;
    if (n <= 1) return n;
    return fib_naive(n - 1) + fib_naive(n - 2);
}

static long fib_memo(int n, long cache[])
{
    call_counter++;
    if (n <= 1)        return n;
    if (cache[n] != 0) return cache[n];
    cache[n] = fib_memo(n - 1, cache) + fib_memo(n - 2, cache);
```

```
    return cache[n];
}

static long fib_iter(int n)
{
    long a = 0, b = 1;
    for (int i = 0; i < n; i++) {
        long next = a + b;
        a = b;
        b = next;
    }
    return a;
}

/* --- how deep can we go? --- */
static int max_depth = 0;
static void probe(int depth)
{
    max_depth = depth;
    if (depth < 200000) {
        probe(depth + 1);
    }
}

int main(void)
{
    puts("== frames build up, then unwind ==");
    factorial(4, 0);

    puts("\n== Towers of Hanoi ==");
    for (int disks = 1; disks <= 4; disks++) {
        int moves = 0;
        if (disks <= 3) printf("  %d disks:\n", disks);
        hanoi(disks, 'A', 'C', 'B', &moves);
        printf("  %d disks -> %d moves (2^%d - 1)\n", disks, moves, disks);
    }

    puts("\n== binary search: recursive and iterative agree ==");
    int sorted[] = { 2, 5, 8, 12, 16, 23, 38, 56, 72, 91 };
```

```
    const long n = (long)(sizeof sorted / sizeof sorted[0]);
    const int targets[] = { 23, 2, 91, 7 };
    for (size_t i = 0; i < 4; i++) {
        printf("  %2d -> recursive %2ld, iterative %2ld\n",
               targets[i],
               bsearch_rec(sorted, 0, n - 1, targets[i]),
               bsearch_iter(sorted, n, targets[i]));
    }
    printf("  depth for %ld elements is about log2(%ld) = 4\n", n, n);

    puts("\n== the Fibonacci trap ==");
    const int fn = 30;

    call_counter = 0;
    clock_t t0 = clock();
    long r1 = fib_naive(fn);
    double d1 = (double)(clock() - t0) / CLOCKS_PER_SEC;
    printf("  naive  fib(%d) = %ld  in %8.4fs, %ld calls\n",
           fn, r1, d1, call_counter);

    long cache[64];
    memset(cache, 0, sizeof cache);
    call_counter = 0;
    t0 = clock();
    long r2 = fib_memo(fn, cache);
    double d2 = (double)(clock() - t0) / CLOCKS_PER_SEC;
    printf("  memo   fib(%d) = %ld  in %8.4fs, %ld calls\n",
           fn, r2, d2, call_counter);

    t0 = clock();
    long r3 = fib_iter(fn);
    double d3 = (double)(clock() - t0) / CLOCKS_PER_SEC;
    printf("  iter   fib(%d) = %ld  in %8.4fs, 0 recursive calls\n",
           fn, r3, d3);

    puts("\n== how deep before the stack runs out ==");
    puts("  (uncomment the probe call below and watch it die)");
```

```
    /* probe(1); printf("reached %d\n", max_depth); */

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o recur recur.c
./recur
```

Watch the frames in GDB

This is the exercise that makes chapter 17's diagram real. Break inside the base case and look at the stack.

```
gdb ./recur
(gdb) break factorial if n == 1
(gdb) run
(gdb) backtrace
```

```
#0  factorial (n=1, depth=3) at recur.c:12
#1  0x... in factorial (n=2, depth=2) at recur.c:17
#2  0x... in factorial (n=3, depth=1) at recur.c:17
#3  0x... in factorial (n=4, depth=0) at recur.c:17
#4  0x... in main () at recur.c:96
```

Four frames of the same function, each with its own `n`, all alive at once. Move between them and inspect their locals:

```
(gdb) frame 2
(gdb) print n
$1 = 3
(gdb) info frame
(gdb) finish
```

`info frame` prints the frame's address, so you can confirm each sits lower than the one below it — exactly the picture from chapter 17. `finish` runs until the current frame returns and shows the value it produced.

Find your actual depth limit

Uncomment the `probe` call and run it:

```
ulimit -s
./recur
```

```
Segmentation fault (core dumped)
```

Now find where it died:

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o recur_asan recur.c
./recur_asan
```

```
==1234==ERROR: AddressSanitizer: stack-overflow on address 0x7ffd...
    #0 0x... in probe recur.c:104
```

Then add `char padding[1024];` to `probe` and try again. The depth drops by a factor of roughly twenty. Frame size, not call count, is what consumes the stack.

The numbers from the Fibonacci section

On a typical machine, `fib_naive(30)` makes about 2.7 million calls; `fib_memo(30)` makes 59. Both return 832 040. Raise `fn` to 40 and the naive version takes seconds while the other two remain instant. The recursion was never the problem — recomputing the same subproblem was.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
No base case	Infinite recursion, stack overflow	Write the base case first.
Base case tests `n == 1`	Negative input recurses forever	Guard the whole range: `n <= 1`.
Recursive call does not shrink the input	Never reaches the base case	Name the decreasing quantity explicitly.
Recursing once per element of a large list	Stack overflow	Iterate; reserve recursion for logarithmic depth.
Naive Fibonacci in real code	Exponential time	Memoize, or iterate.
Large local arrays in a recursive function	Depth limit collapses	Allocate once outside and pass a pointer.
Relying on tail-call optimization	Works at `-O2`, overflows at `-O0`	C does not guarantee it; bound the depth yourself.
`mid = (lo + hi) / 2`	Overflow for large indices	`lo + (hi - lo) / 2`.

7 Check yourself

What are the two obligations of a correct recursive function?

A base case that returns without recursing, and a recursive case whose argument is strictly closer to that base case. The second requires a quantity you can name that decreases on every call and is bounded below — otherwise you have not shown the recursion terminates.

Why is `if (n == 1)` a worse base case than `if (n <= 1)`?

Because it only stops on the exact value. Any input that starts below it — a negative number, or zero — skips the base case and recurses away from it forever. Guard the entire range the function cannot handle recursively, not just the value you expect to arrive at.

In `return n * factorial(n - 1);`, when does the multiplication happen?

On the way out. Each frame is suspended holding its own `n` while the call below it runs; only when that returns can the multiplication be performed. This is why all the frames must coexist and why memory cost is proportional to depth — and it is also why the call is not a tail call.

Naive Fibonacci is exponential. Is recursion the problem?

No — repeated recomputation is. Each call spawns two more and the same subproblems are solved millions of times. Memoizing the results makes the same recursive function linear; so does rewriting it as a loop. The structure of the recursion was fine, the lack of memory was not.

Roughly how deep can you recurse, and what changes the answer most?

Stack limit divided by frame size — around 150 000 levels for a small function within an 8 MB stack. What changes it most is the frame size: adding a one-kilobyte local array cuts the limit to roughly a twentieth. Depth is bounded by memory per call, not by any language limit, and exceeding it kills the process with no diagnostic.

8 Where this leads

Chapter 19 moves to the other half of chapter 17's memory map. On the heap, lifetime is not managed for you: memory you take must be given back, at a moment you choose, exactly once. That responsibility is what makes chapters 19 and 20 the hardest pair in the course — and the recursion you just learned returns in chapter 33, where trees make it indispensable.

INTERMEDIATE · CHAPTER 19

Dynamic Memory Allocation

Until now, lifetime has been decided for you: locals die at the closing brace, statics live forever. The heap removes that guarantee and hands you the decision. Everything you build from here — the module of chapter 28, the library of chapter 29, the data structures of chapters 32 and 33 — rests on the growable array built in this session.

BY THE END OF THIS CHAPTER YOU CAN

- Say when heap allocation is necessary and when a local would do.
- Use `malloc`, `calloc`, `realloc`, and `free` correctly, and check every one.
- Implement a growable array with amortized constant-time append.
- Allocate a two-dimensional array two different ways and justify the choice.
- State an ownership rule for every function that returns or takes a pointer.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Why the heap exists

Automatic and static storage cover most needs. Three situations they cannot:

- **The size is not known until run time.** You cannot declare an array before you know how many lines the file has.
- **The object must outlive the function that creates it.** A function that builds a list and returns it cannot put it on the stack — chapter 13's dangling pointer.
- **The object is too large for the stack.** Chapter 17's 8 MB limit; a 100 MB buffer has nowhere else to go.

	STACK	HEAP
Size decided	Compile time	Run time
Lifetime	Until the block ends	Until you `free` it
Cost of allocating	Adjusting a register — essentially free	A library call; tens to hundreds of cycles
Capacity	A few megabytes	Most of available memory
Cleanup	Automatic	Yours, exactly once
Failure	Fatal signal	`NULL`, which you can handle

The default should still be automatic storage. Reach for the heap when one of the three reasons above applies, not by habit.

2 The four functions

```
#include <stdlib.h>


void *malloc(size_t size);
void *calloc(size_t count, size_t size);
void *realloc(void *ptr, size_t new_size);
void  free(void *ptr);
```

`malloc`

```
int *values = malloc(n * sizeof *values);
if (values == NULL) {
    return false;                /* allocation failed */
}
```

Returns a pointer to `size` uninitialized bytes, or `NULL`. Three details in that one line:

`sizeof *values`**, not** `sizeof(int)`**.** Writing the size in terms of the pointer means changing the type later cannot desynchronize them. It is the standard idiom and worth adopting immediately.

No cast. `void *` converts implicitly to any object pointer in C. Casting is unnecessary and, before C99, could hide a missing `#include <stdlib.h>`. (In C++ the cast is required — which is why you will see it in code written to compile as both.)

The contents are garbage. Not zero. Reading before writing is undefined behavior, exactly like an uninitialized local.

`calloc`

```
int *values = calloc(n, sizeof *values);   /* zero-filled */
```

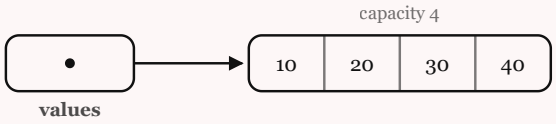
Takes the count and element size separately, and **zeroes the memory**. The separate arguments are not cosmetic: `calloc` detects overflow in `count * size` and returns `NULL`, whereas `malloc(count * size)` would silently wrap and allocate far too little — a classic vulnerability. Prefer `calloc` whenever the count comes from outside your program.

`realloc`

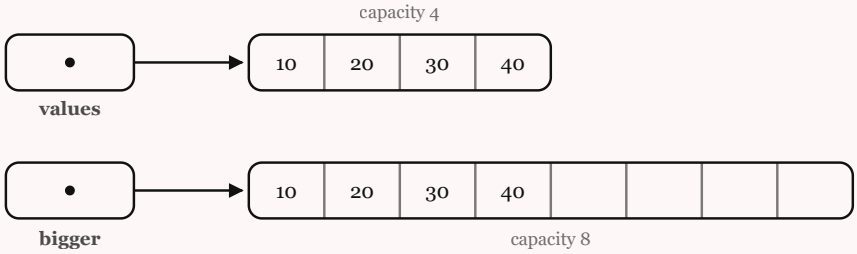
```
int *bigger = realloc(values, new_count * sizeof *values);
if (bigger == NULL) {
    free(values);                /* the original is still valid */
    return false;
}
values = bigger;
```

Grows or shrinks a block, preserving the contents up to the smaller of the two sizes. It may return a *different* address, having copied the data — so any other pointer into the old block is now dangling.

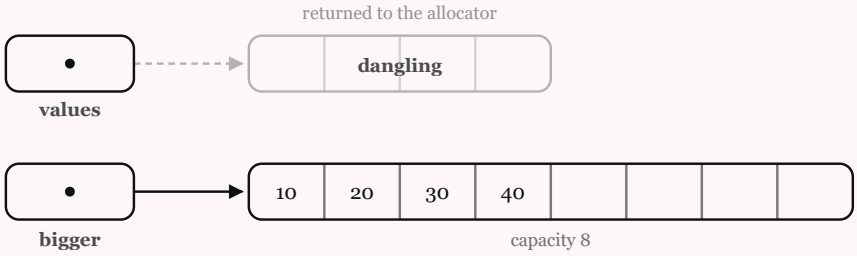
Never write `values = realloc(values, …)`**.** If it fails it returns `NULL` while the original block remains allocated. Assigning directly overwrites your only pointer to it, so the memory can never be freed — a leak that occurs precisely when memory is already short. Always use a temporary, as above.



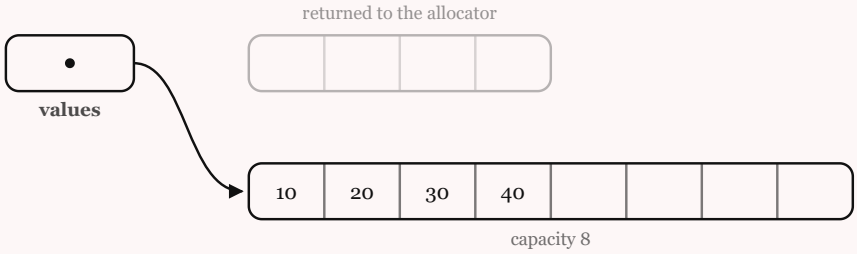
Four slots, four in use. The next append has nowhere to go.



`realloc` finds a block twice the size and copies the contents across. For a moment both exist, and `bigger` names the new one.



It then releases the old block. This is the part that bites: *any other pointer you were holding into the old block is now dangling*, even though `values` still happens to hold that address.



`values = bigger`. Note the order — the temporary existed so that a failed `realloc` would have left the original block reachable and freeable.

`free`

```
free(values);
values = NULL;        /* not required, but a good habit */
```

Returns the block to the allocator. `free(NULL)` is explicitly safe and does nothing, which removes the need for a guard. Setting the pointer to `NULL` afterwards turns a later use-after-free into an immediate null-pointer crash rather than silent corruption — chapter 20's subject.

3 Checking every allocation

`malloc` can fail. On a desktop with overcommit it rarely does, which is exactly why unchecked allocations survive into production and then fail on a constrained machine, or under a memory limit, or when a corrupted size request asks for four gigabytes.

```
/* wrong */
int *p = malloc(n * sizeof *p);
p[0] = 1;                          /* if p is NULL, this crashes */

/* right */
int *p = malloc(n * sizeof *p);
if (p == NULL) {
    fprintf(stderr, "out of memory\n");
    return EXIT_FAILURE;
}
```

One subtlety: `malloc(0)` may return `NULL` *or* a unique pointer that you must still free. Both are conforming. Code that treats `NULL` as failure will misreport a zero-size request, so guard the zero case explicitly if it can occur.

4 Ownership

C has no garbage collector and no destructors. For every allocated block there must be exactly one `free`, and deciding *who* calls it is a design decision that belongs in the interface.

Make it explicit in one of three ways:

```
/* 1. The caller owns the result and must free it. */
char *duplicate_string(const char *s);      /* caller frees */


/* 2. The caller supplies the storage; nobody allocates. */
bool copy_into(char *dst, size_t dst_size, const char *src);


/* 3. Paired create and destroy — the clearest for anything with state. */
Buffer *buffer_create(size_t capacity);
void    buffer_destroy(Buffer *b);
```

The third convention is the one to adopt for anything non-trivial. Naming them as a pair makes the obligation visible at the call site, and it scales to types that own several allocations internally — `buffer_destroy` frees everything, and the caller does not need to know what "everything" is. Chapter 33 turns this into an opaque handle where the caller *cannot* know.

Write the ownership rule down. A one-line comment above the prototype — `/* returns a new string; caller frees */` — prevents both the leak and the double free, and it costs nothing.

5 Two ways to allocate a matrix

```
/* A: one contiguous block, indexed manually */
int *m = malloc(rows * cols * sizeof *m);
m[r * cols + c] = 7;
free(m);

/* B: an array of row pointers */
int **m = malloc(rows * sizeof *m);
for (size_t r = 0; r < rows; r++) {
    m[r] = malloc(cols * sizeof *m[r]);
}
m[r][c] = 7;
for (size_t r = 0; r < rows; r++) free(m[r]);
free(m);
```

	A — FLAT	B — ROW POINTERS
Allocations	1	rows + 1
Syntax	`m[r * cols + c]`	`m[r][c]`
Cache behavior	Contiguous — fast	Rows scattered — slower
Rows of different lengths	No	Yes
Freeing	One call	A loop, and easy to get wrong

Prefer A unless you genuinely need ragged rows. One allocation, one free, contiguous memory. The slightly uglier indexing is a small price, and it can be hidden behind an accessor function or a macro.

6 Worked example: the growable array

This is the data structure the rest of the course keeps returning to. Chapter 28 turns it into a module with a header; chapter 29 builds it as a library; chapter 30 makes it generic; chapter 45 packages it.

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

```
typedef struct {
    int    *data;
    size_t  count;      /* elements in use   */
    size_t  capacity;   /* elements allocated */
} IntArray;

/* Paired with array_destroy. After this returns true the caller owns
   the array and must call array_destroy exactly once. */
static bool array_init(IntArray *a, size_t initial_capacity)
{
    if (initial_capacity == 0) {
        initial_capacity = 4;
    }
    a->data = malloc(initial_capacity * sizeof *a->data);
    if (a->data == NULL) {
        a->count = a->capacity = 0;
        return false;
    }
    a->count = 0;
    a->capacity = initial_capacity;
    return true;
}

static void array_destroy(IntArray *a)
{
    free(a->data);              /* free(NULL) is safe */
    a->data = NULL;
    a->count = a->capacity = 0;
}

/* Doubling gives amortized O(1) append: n appends cost O(n) copies total. */
static bool array_grow(IntArray *a)
{
    size_t new_capacity = a->capacity * 2;

    int *bigger = realloc(a->data, new_capacity * sizeof *a->data);
    if (bigger == NULL) {
        return false;           /* a->data is still valid and still ours */
```

```
    }
    a->data = bigger;
    a->capacity = new_capacity;
    return true;
}

static bool array_push(IntArray *a, int value)
{
    if (a->count == a->capacity && !array_grow(a)) {
        return false;
    }
    a->data[a->count++] = value;
    return true;
}

static bool array_get(const IntArray *a, size_t index, int *out)
{
    if (index >= a->count) {
        return false;
    }
    *out = a->data[index];
    return true;
}

static void array_print(const IntArray *a, const char *label)
{
    printf("  %-10s count=%-3zu capacity=%-3zu [", label, a->count, a->capacity);
    for (size_t i = 0; i < a->count; i++) {
        printf("%d%s", a->data[i], i + 1 < a->count ? " " : "");
    }
    puts("]");
}

/* Caller frees the result. Named so the obligation is hard to miss. */
static char *duplicate_string(const char *s)
{
    size_t bytes = strlen(s) + 1;          /* room for the terminator */
    char *copy = malloc(bytes);
    if (copy == NULL) {
```

```
        return NULL;
    }
    memcpy(copy, s, bytes);
    return copy;
}

int main(void)
{
    puts("== growable array ==");
    IntArray a;
    if (!array_init(&a, 2)) {
        fprintf(stderr, "out of memory\n");
        return EXIT_FAILURE;
    }

    array_print(&a, "empty");
    for (int i = 1; i <= 9; i++) {
        if (!array_push(&a, i * i)) {
            fprintf(stderr, "out of memory\n");
            array_destroy(&a);
            return EXIT_FAILURE;
        }
        if (a.count == a.capacity) {
            array_print(&a, "full");
        }
    }
    array_print(&a, "final");
    puts("  capacity doubled 2 -> 4 -> 8 -> 16: three reallocations for nine pushes");

    int value;
    printf("  element 3  : %s", array_get(&a, 3, &value) ? "" : "refused\n");
    if (array_get(&a, 3, &value)) printf("%d\n", value);
    printf("  element 99 : %s\n",
           array_get(&a, 99, &value) ? "returned a value" : "refused");

    array_destroy(&a);
    puts("  destroyed; calling array_destroy again would also be safe");
    array_destroy(&a);
```

```
    puts("\n== calloc zeroes, malloc does not ==");
    int *raw = malloc(4 * sizeof *raw);
    int *zeroed = calloc(4, sizeof *zeroed);
    if (raw != NULL && zeroed != NULL) {
        printf("  malloc: %d %d %d %d   (garbage — do not rely on it)\n",
               raw[0], raw[1], raw[2], raw[3]);
        printf("  calloc: %d %d %d %d   (guaranteed zero)\n",
               zeroed[0], zeroed[1], zeroed[2], zeroed[3]);
    }
    free(raw);
    free(zeroed);

    puts("\n== a flat matrix: one allocation, one free ==");
    const size_t rows = 3, cols = 4;
    int *m = malloc(rows * cols * sizeof *m);
    if (m == NULL) {
        return EXIT_FAILURE;
    }
    for (size_t r = 0; r < rows; r++) {
        for (size_t c = 0; c < cols; c++) {
            m[r * cols + c] = (int)(r * cols + c);
        }
    }
    for (size_t r = 0; r < rows; r++) {
        printf("  ");
        for (size_t c = 0; c < cols; c++) {
            printf("%3d", m[r * cols + c]);
        }
        putchar('\n');
    }
    free(m);

    puts("\n== ownership, stated in the name ==");
    char *copy = duplicate_string("the caller frees this");
    if (copy != NULL) {
        printf("  \"%s\"\n", copy);
        free(copy);
```

```
    }

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o dynarray dynarray.c
./dynarray
```

Why doubling

Growing by one element per push would copy the whole array every time: n pushes cost $n^2/2$ copies. Doubling copies only when the array is full, and the total work across n pushes is $n + n/2 + n/4 + \ldots < 2n$. Each push is therefore O(1) *amortized*, even though individual pushes occasionally cost O(n). Chapter 33 puts a name to this reasoning.

The output shows three reallocations for nine pushes. Change the growth to `capacity + 1` and count again.

Confirm there are no leaks

AddressSanitizer includes a leak detector, and the program above is clean:

```
./dynarray
(no LeakSanitizer output = nothing leaked)
```

Now remove one `free` — the `free(copy)` at the end — and run it again:

```
==1234==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 22 byte(s) in 1 object(s) allocated from:
    #0 0x... in malloc
    #1 0x... in duplicate_string dynarray.c:103
    #2 0x... in main dynarray.c:176

SUMMARY: AddressSanitizer: 22 byte(s) leaked in 1 allocation(s).
```

Exact size, exact allocation site, exact call path. Valgrind gives the same information without recompiling:

```
valgrind --leak-check=full ./dynarray
```

The `realloc` mistake, demonstrated

Change `array_grow` to the tempting one-liner:

```
a->data = realloc(a->data, new_capacity * sizeof *a->data);
if (a->data == NULL) return false;
```

It works, until `realloc` fails. Then `a->data` is `NULL`, the old block is still allocated, and the only pointer to it has been destroyed. Simulating this is awkward on a desktop, which is the point: the bug is invisible in testing and fires exactly when memory is scarce.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Not checking `malloc`'s result	Null dereference under memory pressure	Check every allocation.
`p = realloc(p, n)`	Leak on failure; the only pointer is lost	Assign to a temporary first.
Reading `malloc`'d memory before writing it	Undefined behavior; garbage values	Initialize, or use `calloc`.
`malloc(count * size)` with untrusted `count`	Integer overflow, undersized buffer	`calloc(count, size)`, which checks.
Forgetting `+ 1` for a string terminator	Buffer overflow	`malloc(strlen(s) + 1)`.
Keeping a pointer into a block after `realloc`	Dangling — the block may have moved	Recompute pointers from the new base.
Unclear ownership	Double free or leak	Document it; use create/destroy pairs.
Freeing only the row pointers of a 2D array	Every row leaks	Free the rows first, then the array. Or use a flat block.

8 Check yourself

Why is `malloc(n * sizeof *p)` preferred over `malloc(n * sizeof(int))`?

Because the size is expressed in terms of the pointer being assigned, so changing the element type later cannot leave the size stale. It is also shorter and needs no cast. The same reasoning applies to `calloc(n, sizeof *p)`.

Why does `calloc` take two arguments instead of one product?

So it can detect overflow in `count × size` and fail safely. `malloc(count * size)` computes the product in `size_t` first, and if that wraps it allocates a small block while the program believes it is large — an integer-overflow vulnerability. It also zeroes the memory, which `malloc` does not.

What is wrong with `p = realloc(p, new_size);`?

If `realloc` fails it returns `NULL` while leaving the original block allocated. Assigning straight to `p` overwrites the only pointer to that block, so it can never be freed. Assign to a temporary, test it, and only then update `p`.

Why does doubling the capacity give amortized constant-time append?

Because copying happens only when the array is full, and each copy is twice as far from the previous one. The total copying across n appends is $n + n/2 + n/4 + \ldots$ which is less than $2n$, so the average cost per append is constant. Growing by a fixed amount instead makes the total quadratic.

Who frees the result of a function that returns a pointer?

Whoever the interface says — and the interface must say. The three workable conventions are: the caller frees a returned allocation, the caller supplies the buffer so nothing is allocated, or a paired create/destroy owns everything internally. What must not happen is leaving it unstated, which produces either a leak or a double free.

9 Where this leads

You can now allocate and release memory correctly — in code that works. Chapter 20 is about code that does not: leaks, use-after-free, double free, and overruns, each seeded deliberately into the array you just built and then hunted down with a different tool. Those tools are what make heap programming in C tractable rather than terrifying.

INTERMEDIATE · CHAPTER 20

Memory Errors and Their Diagnosis

C will not stop you making any of these mistakes, and none of them reliably crashes at the point of the error. What makes heap programming tractable is not care alone but tooling — and the tools are good enough that not using them is the real mistake.

BY THE END OF THIS CHAPTER YOU CAN

- Recognize the five classes of memory error and the symptom each produces.
- Read an AddressSanitizer report and locate the defect from it.
- Use Valgrind when recompiling is not an option.
- Explain why a memory bug's symptom usually appears far from its cause.
- Adopt the habits that prevent each class rather than detecting it.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The five classes

ERROR	WHAT IT IS	TYPICAL SYMPTOM
Leak	Allocated, never freed	Memory grows over hours; eventual failure
Use after free	Access through a pointer after `free`	Corrupt data, or a crash much later
Double free	`free` called twice on one block	Allocator abort, or heap corruption
Overrun	Read or write past the end of a block	A neighbouring value changes silently
Invalid free	`free` on a stack or interior pointer	Immediate abort

What unites them is **action at a distance**. The line that does the damage and the line that fails are usually far apart, often in unrelated code. A program can write past a buffer for months before a slightly different allocation pattern makes the corruption land somewhere fatal.

This is why debugging memory errors by reading code and adding `printf` is so unproductive, and why the tools in section 3 are not optional extras.

2 Each class in detail

Leak

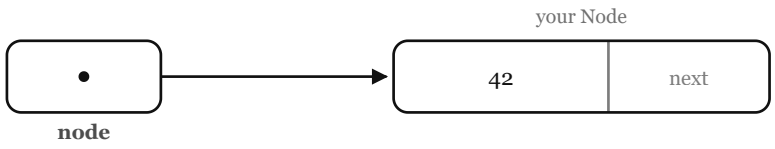
```
void process(void)
{
    char *buffer = malloc(1024);
    if (something_failed()) {
        return;                    /* leaked: early return skips the free */
    }
    free(buffer);
}
```

Leaks on the error path are by far the most common kind, because the happy path gets tested and the error path does not. A short-lived program can survive leaking — the operating system reclaims everything at exit — but a server or a long-running tool cannot. Chapter 27's `goto` cleanup idiom exists precisely to give every path one exit.

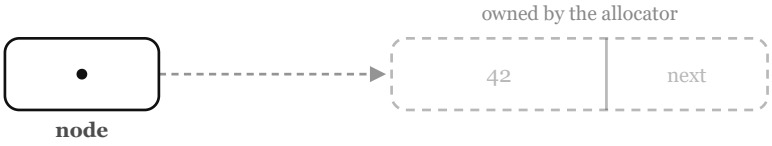
Use after free

```
free(node);
printf("%d\n", node->value);       /* the block may already be reused */
```

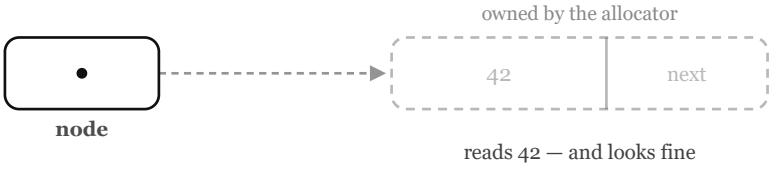
After `free`, the block belongs to the allocator, which may hand it to the next `malloc`. Reading gives stale or foreign data; writing corrupts whatever now lives there. It frequently appears to work, because the allocator has not yet reused the block — which is why it survives testing.



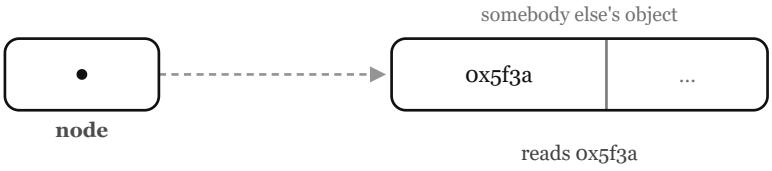
A live `Node` on the heap. `node` holds its address.



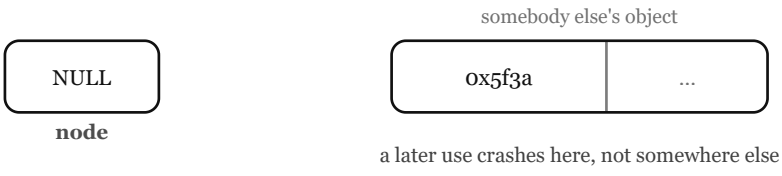
`free(node)`. The block belongs to the allocator again — but `node` was not changed, so it still holds the address. It is now dangling.



`node->value` reads 42 and everything appears to work, because the allocator has not handed the block out yet. This is exactly why the bug survives testing.



A later `malloc` hands the same block to something else. Now the read returns somebody else's data, and a write corrupts it — far from the line that is at fault.



The habit that prevents all of this: `node = NULL;` immediately after `free`. A later use then crashes instantly, at the guilty line.

The defence is a habit: **set the pointer to `NULL` immediately after freeing**. A later use then crashes instantly at the guilty line instead of corrupting something.

Double free

```
free(p);
…
free(p);                          /* corrupts the allocator's bookkeeping */
```

Usually arises when two pieces of code each believe they own a block — the ownership question from chapter 19, left unanswered. Modern allocators detect the simple case and abort with a message; the subtle cases silently corrupt the heap. The `NULL` habit fixes this too, since `free(NULL)` is defined to do nothing.

Overrun

```
int *a = malloc(10 * sizeof *a);
a[10] = 0;                        /* one past the end */
```

Identical in nature to chapter 12's array overflow, but on the heap the neighbour is usually the allocator's own metadata — the block header recording sizes and links. Corrupting it means the *next* unrelated `malloc` or `free` crashes, with a backtrace pointing at innocent code.

Invalid free

```
int local = 5;
free(&local);                     /* not from malloc */

char *p = malloc(100);
free(p + 10);                     /* not the start of the block */
```

Only a pointer returned by `malloc`, `calloc`, or `realloc` may be freed, and only the exact value returned. Keep the original pointer; walk a copy.

3 The tools

AddressSanitizer

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o prog prog.c
./prog
```

A compiler instrumentation that checks every memory access. It catches overruns, use-after-free, double free, invalid free, and — with LeakSanitizer, which is on by default on Linux — leaks at exit. Costs roughly 2× in speed and 3× in memory, which is irrelevant during development.

Use it for every build while you are writing code. It is the single highest-value flag in this course.

Valgrind

```
valgrind --leak-check=full --track-origins=yes ./prog
```

Runs an unmodified binary under emulation, so no recompilation is needed — which matters when you did not build the program. It also detects *uninitialized* reads, which AddressSanitizer does not; `--track-origins=yes` then tells you where the uninitialized value came from. The cost is 10–50× slowdown.

	ADDRESSSANITIZER	VALGRIND
Recompilation	Required	Not required
Slowdown	~2×	10–50×
Overruns, use-after-free	Yes	Yes
Leaks	Yes	Yes, in more detail
Uninitialized reads	No (use MemorySanitizer)	Yes
Stack errors	Yes	Limited

They are complements. Develop under AddressSanitizer; run Valgrind before a release, and when you need to inspect a binary you cannot rebuild.

> **Sanitizers do not combine freely.** `-fsanitize=address` and `-fsanitize=memory` cannot be used together. `-fsanitize=address,undefined` does work, and is a good default — the undefined-behavior sanitizer catches the signed overflow and shift errors from chapters 4 and 7. Chapter 35 puts both into a continuous integration configuration.

4 Reading a report

```
==12345==ERROR: AddressSanitizer: heap-use-after-free on address 0x602000000010
READ of size 4 at 0x602000000010 thread T0
    #0 0x55e in main leaky.c:42                    <- where it was USED
    #1 0x7f2 in __libc_start_main

0x602000000010 is located 0 bytes inside of 40-byte region
freed by thread T0 here:
    #0 0x7f8 in free
    #1 0x55d in main leaky.c:40                    <- where it was FREED

previously allocated by thread T0 here:
    #0 0x7f9 in malloc
    #1 0x55c in main leaky.c:36                    <- where it was ALLOCATED
```

Read it in four parts:

1. **The error type.** `heap-use-after-free`, `heap-buffer-overflow`, `stack-buffer-overflow`, `double-free` — each names the class from section 1.
2. **The access.** `READ of size 4` tells you what the program was doing.
3. **Three stack traces.** Where it was used, where it was freed, where it was allocated. Those three lines are usually the entire diagnosis.
4. **The offset.** `0 bytes inside of 40-byte region`, or `4 bytes to the right of`, which pinpoints an off-by-one immediately.

5 Worked example: five bugs, seeded and found

This takes the growable array from chapter 19 and introduces each defect behind a compile-time switch, so you can enable exactly one and watch the tool describe it.

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

/* Build with -DBUG=1 .. -DBUG=5 to enable one defect at a time.
   Build with no -DBUG for the clean version. */
#ifndef BUG
#define BUG 0
#endif

typedef struct {
    int    *data;
    size_t  count;
    size_t  capacity;
} IntArray;

static bool array_init(IntArray *a, size_t capacity)
{
    a->data = malloc(capacity * sizeof *a->data);
    if (a->data == NULL) {
        return false;
    }
    a->count = 0;
    a->capacity = capacity;
    return true;
}

static void array_destroy(IntArray *a)
{
    free(a->data);
    a->data = NULL;            /* the habit that prevents bugs 2 and 3 */
    a->count = a->capacity = 0;
}

static bool array_push(IntArray *a, int value)
{
    if (a->count == a->capacity) {
```

```
        size_t bigger_capacity = a->capacity * 2;
        int *bigger = realloc(a->data, bigger_capacity * sizeof *bigger);
        if (bigger == NULL) {
            return false;
        }
        a->data = bigger;
        a->capacity = bigger_capacity;
    }
    a->data[a->count++] = value;
    return true;
}

int main(void)
{
    printf("built with BUG=%d\n\n", BUG);

    IntArray a;
    if (!array_init(&a, 4)) {
        return EXIT_FAILURE;
    }
    for (int i = 1; i <= 6; i++) {
        array_push(&a, i * 10);
    }
    printf("array holds %zu of %zu: ", a.count, a.capacity);
    for (size_t i = 0; i < a.count; i++) printf("%d ", a.data[i]);
    putchar('\n');

#if BUG == 1
    /* LEAK: allocate and forget. */
    char *forgotten = malloc(64);
    strcpy(forgotten, "nobody will free me");
    printf("leaked: %s\n", forgotten);
    /* no free */

#elif BUG == 2
    /* USE AFTER FREE. */
    free(a.data);
    printf("after free, a.data[0] = %d\n", a.data[0]);
    a.data = NULL;
```

```
#elif BUG == 3
    /* DOUBLE FREE. */
    free(a.data);
    free(a.data);
    a.data = NULL;

#elif BUG == 4
    /* HEAP OVERRUN: one element past the end. */
    printf("writing a.data[%zu] with capacity %zu\n", a.capacity, a.capacity);
    a.data[a.capacity] = 999;

#elif BUG == 5
    /* INVALID FREE: not the pointer malloc returned. */
    free(a.data + 1);
    a.data = NULL;
#endif

    array_destroy(&a);
    puts("finished");
    return EXIT_SUCCESS;
}
```

Run each one

```
for b in 0 1 2 3 4 5; do
  gcc -std=c17 -Wall -Wextra -g -fsanitize=address -DBUG=$b -o bug$b bugs.c
done

./bug0     # clean
./bug1     # leak
./bug2     # use after free
./bug3     # double free
./bug4     # overrun
./bug5     # invalid free
```

What each reports

BUG=1 — leak. The program finishes normally, then:

```
==1234==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 64 byte(s) in 1 object(s) allocated from:
    #1 0x... in main bugs.c:76
SUMMARY: AddressSanitizer: 64 byte(s) leaked in 1 allocation(s).
```

Note that the output appears only at exit and the exit status becomes non-zero — which is how a leak fails a build in chapter 35's continuous integration.

BUG=2 — use after free. Three traces: used at line 82, freed at line 81, allocated at line 44. Without the sanitizer this program prints a value quite happily — often not even `10`, because the allocator stores its own bookkeeping in the first bytes of a freed block.

BUG=3 — double free. `attempting double-free on 0x...`, with the free site and the original allocation both named. Without the sanitizer, glibc prints `free(): double free detected in tcache 2` and aborts — useful, but with no indication of where.

BUG=4 — overrun. The most instructive one:

```
==1234==ERROR: AddressSanitizer: heap-buffer-overflow
WRITE of size 4 at 0x602000000030 thread T0
    #0 0x... in main bugs.c:95
0x602000000030 is located 0 bytes to the right of 32-byte region
allocated by thread T0 here:
    #1 0x... in array_push bugs.c:44
```

"0 bytes to the right of a 32-byte region" is an off-by-one stated in the clearest possible terms. Compare with the unsanitized build, which prints `finished` and exits successfully while having corrupted the allocator's metadata.

BUG=5 — invalid free. `attempting free on address which was not malloc()-ed`, plus the allocation site of the real block.

The same bugs under Valgrind

```
gcc -std=c17 -Wall -Wextra -g -DBUG=2 -o bug2_plain bugs.c
valgrind --leak-check=full ./bug2_plain
```

```
==9876== Invalid read of size 4
==9876==    at 0x...: main (bugs.c:82)
==9876==  Address 0x4a4b040 is 0 bytes inside a block of size 32 free'd
==9876==    at 0x...: free
==9876==    by 0x...: main (bugs.c:81)
==9876==  Block was alloc'd at
==9876==    by 0x...: array_push (bugs.c:44)
```

The same three facts in different words — and obtained without rebuilding with a special flag, which is what makes Valgrind valuable on binaries you did not compile.

One Valgrind catches that AddressSanitizer does not

```
int *p = malloc(4 * sizeof *p);
printf("%d\n", p[0]);          /* reading uninitialized heap memory */
free(p);
```

```
==9876== Conditional jump or move depends on uninitialised value(s)
==9876==  Uninitialised value was created by a heap allocation
==9876==    at 0x...: malloc
```

AddressSanitizer reports nothing here — the memory is legitimately owned, merely unwritten. This is why the two tools are complements rather than alternatives, and why `calloc` is worth preferring when you cannot guarantee every byte gets written.

6 Habits that prevent rather than detect

HABIT	PREVENTS
Set the pointer to `NULL` right after `free`	Use after free, double free
Write the `free` immediately after writing the `malloc`	Leaks
One exit path per function, via `goto cleanup`	Leaks on error paths — chapter 27
Paired `create` / `destroy` functions	Ambiguous ownership, double free
Never `free` a pointer you did not receive from `malloc`	Invalid free
`calloc` when not every byte will be written	Uninitialized reads
Build with `-fsanitize=address,undefined` by default	All of the above, detected on the day you write them

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Early `return` before the `free`	Leak on the error path only	Single cleanup path — chapter 27.
Using a pointer after freeing it	Stale data, or corruption	Assign `NULL` immediately after `free`.
Two owners of one block	Double free	Decide ownership once; document it.
Freeing an interior pointer	Immediate abort	Keep the original; walk a copy.
Assuming no crash means no bug	Corruption survives into production	Run under a sanitizer before believing it works.
Testing only the happy path	Error-path leaks never exercised	Test failure injection — chapter 35.
Using a pointer into a block after `realloc`	Dangling — the block may have moved	Recompute from the new base pointer.
Shipping a release build never run under Valgrind	Leaks found by users	Make it part of the release checklist.

8 Check yourself

Why does a memory bug's symptom usually appear far from its cause?

Because the damaging operation is legal at the machine level and produces no immediate failure. An overrun quietly modifies a neighbouring object or the allocator's metadata; a use-after-free reads a block that has not yet been reused. The failure surfaces only when something else touches the corrupted state, which can be much later and in unrelated code.

What does setting a pointer to `NULL` after `free` buy you?

It converts two silent bugs into immediate, obvious ones. A later use dereferences `NULL` and crashes at the guilty line instead of reading a reused block; a later `free` becomes `free(NULL)`, which is defined to do nothing. It costs one assignment.

When would you reach for Valgrind rather than AddressSanitizer?

When you cannot rebuild the binary, and when you need to detect reads of uninitialized memory — which AddressSanitizer does not catch, because the memory is legitimately owned. Valgrind's `--track-origins=yes` also reports where an uninitialized value came from. The trade-off is a slowdown of 10–50× instead of about 2×.

An AddressSanitizer report says "4 bytes to the right of a 32-byte region". What does that tell you?

That the program accessed memory just past the end of a 32-byte allocation — an off-by-one or a loop bound that is too large. The report also names where that block was allocated, so the array and the faulty access are both identified without reading any code.

Why is a leak in a short-lived command-line tool less urgent than one in a server?

Because the operating system reclaims the whole address space when a process exits, so a tool that runs for a second and leaks a kilobyte causes no harm. A server runs for chapters and repeats the leaking path millions of times, so the same defect grows without bound until allocation fails. The bug is identical; the consequence is not.

9 Where this leads

Chapter 21 turns to structures, which let you group related data into one object — and which immediately raise a harder version of this chapter's question, because a structure can own allocations of its own. The create/destroy pattern from chapter 19 becomes the standard answer, and chapter 33 makes it an opaque handle where the caller cannot see, or corrupt, the internals at all.

INTERMEDIATE · CHAPTER 21

Structures

An array groups values of one type. A structure groups values of different types under one name — which is how C stops passing six loose parameters around and starts modelling the thing those parameters describe.

BY THE END OF THIS CHAPTER YOU CAN

- Declare and initialize a structure, including with designated initializers.
- Choose between `.` and `->` without thinking about it.
- Decide whether to pass a structure by value or by pointer, with a reason.
- Build and traverse an array of structures.
- Declare a self-referential type and explain why it must contain a pointer.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Declaring a structure

```
struct Point {
    int x;
    int y;
};                      /* the semicolon is required */

struct Point origin;    /* every use needs the keyword 'struct' */
```

Having to write `struct Point` everywhere is tedious, so almost all real code introduces an alias:

```
typedef struct {
    int x;
    int y;
} Point;

Point origin;           /* now just the name */
```

The convention in this course, and in most modern C, is `typedef` with a capitalized type name. Chapter 22 covers `typedef` in its own right.

Initializing

```
Point a = { 3, 4 };                 /* positional */
Point b = { .x = 3, .y = 4 };       /* designated — C99 */
Point c = { .y = 4 };               /* x is zero-filled */
Point d = { 0 };                    /* every member zeroed */
Point e = a;                        /* structures assign by copy */
```

Prefer designated initializers. They are readable without counting, they survive a reordering of the members, and they make it obvious which fields you deliberately left at zero. Positional initialization of a six-member structure is a bug waiting for a maintenance change.

Note the last line: unlike arrays, structures *can* be assigned and copied wholesale. `e = a` copies every member.

2 Access: . and ->

```
Point p = { .x = 1, .y = 2 };
Point *q = &p;


p.x          /* a structure: use dot   */
q->x         /* a pointer:   use arrow */
(*q).x       /* identical to q->x, and nobody writes it */
```

`->` exists purely because `(*q).x` is unpleasant, and because the parentheses are mandatory — `*q.x` parses as `*(q.x)`, since `.` binds tighter than `*`. That is a real source of confusion in old code; the arrow removes it.

The rule is mechanical: **dot for an object, arrow for a pointer to one**. Since most structures are passed around by pointer, you will write `->` far more often.

3 By value or by pointer?

Structures obey chapter 11's rule exactly: passing one to a function copies every member.

```
void move_broken(Point p)      { p.x += 10; }   /* modifies the copy  */
void move(Point *p)            { p->x += 10; }  /* modifies the caller's */
int  area(const Rect *r)       { return (r->bottom_right.x - r->top_left.x) * (r-
>bottom_right.y - r->top_left.y); }  /* reads only */
```

PASS BY VALUE WHEN	PASS BY POINTER WHEN
The structure is small — two or three scalars	The structure is large
The function must not modify it	The function must modify it
You want a private copy to alter freely	It contains allocations you must not duplicate

The default in practice is **a pointer, with `const` when the function only reads**. It avoids copying, it is uniform, and the `const` makes the contract explicit and compiler-checked, exactly as chapter 16 established.

Returning a structure by value is fine and common — `Point make_point(int x, int y)` — because the copy happens once and the caller owns the result with no lifetime question at all.

4 Nesting and arrays

```
typedef struct {
    Point top_left;
    Point bottom_right;
} Rect;

Rect r = {
    .top_left     = { .x = 0, .y = 0 },
    .bottom_right = { .x = 4, .y = 3 }
};

int width = r.bottom_right.x - r.top_left.x;
```

Nested members chain with `.`, or with `->` at the first step if you start from a pointer: `rp->top_left.x`.

Arrays of structures

```
typedef struct {
    char   name[32];
    int    score;
    double average;
} Student;

Student class[3] = {
    { .name = "Ada",    .score = 95 },
    { .name = "Dennis", .score = 88 },
    { .name = "Ken",    .score = 91 }
};

for (size_t i = 0; i < 3; i++) {
    printf("%-10s %3d\n", class[i].name, class[i].score);
}
```

Note that `name` is an array *inside* the structure, so the text is stored in the structure itself rather than pointed at. That makes the structure self-contained and copyable — `Student b = a;` duplicates the name too. The alternative, `char *name`, stores only a pointer, which is smaller and more flexible but raises the ownership question from chapter 19 for every copy.

5 Self-referential structures

A structure cannot contain itself — the size would be infinite. It can contain a *pointer* to itself, which is the basis of every linked structure in chapter 32.

```
typedef struct Node {        /* the tag is required here */
    int          value;
    struct Node *next;       /* fine: a pointer has a known size */
} Node;
```

The struct tag `Node` after the `struct` keyword is necessary because the `typedef` name does not exist yet inside its own definition. This is one of the few places where the tag cannot be omitted.

```
Node third  = { .value = 3, .next = NULL };
Node second = { .value = 2, .next = &third };
Node first  = { .value = 1, .next = &second };

for (Node *n = &first; n != NULL; n = n->next) {
    printf("%d ", n->value);
}
```

That loop is the shape of every list traversal you will write. Chapter 32 replaces the stack-allocated nodes with heap allocations and adds insertion and removal.

6 What a structure does *not* support

OPERATION	WORKS?	INSTEAD
Assignment `a = b`	Yes	—
Passing and returning by value	Yes	—
Comparison `a == b`	**No**	Compare members, or `memcmp` — with care
Arithmetic	No	Write a function
Printing directly	No	Write a print function

> **Do not compare structures with `memcmp`.** It compares every byte, including the *padding* the compiler inserts between members for alignment — and padding bytes hold whatever was in that memory, even in two structures whose members are all equal. Two logically identical structures can therefore compare unequal. Compare the members you care about, one by one. Chapter 22 explains where the padding comes from.

7 Worked example: records, by value and by pointer

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

#define NAME_LEN 32
```

```
typedef struct {
    int x;
    int y;
} Point;

typedef struct {
    Point top_left;
    Point bottom_right;
} Rect;

typedef struct {
    char   name[NAME_LEN];
    int    score;
    int    attempts;
} Student;

/* Small and read-only: by value is fine and reads well. */
static int manhattan(Point a, Point b)
{
    int dx = a.x - b.x;
    int dy = a.y - b.y;
    return (dx < 0 ? -dx : dx) + (dy < 0 ? -dy : dy);
}

/* Read-only: const pointer. No copy, contract enforced. */
static int rect_area(const Rect *r)
{
    return (r->bottom_right.x - r->top_left.x)
         * (r->bottom_right.y - r->top_left.y);
}

/* Modifies: non-const pointer. */
static void rect_translate(Rect *r, int dx, int dy)
{
    r->top_left.x     += dx;  r->top_left.y     += dy;
    r->bottom_right.x += dx;  r->bottom_right.y += dy;
}

/* Returning by value: the caller owns the result, no lifetime question. */
```

```
static Point point_make(int x, int y)
{
    return (Point){ .x = x, .y = y };    /* compound literal — chapter 38 */
}

/* Structures have no ==; compare what actually matters. */
static bool point_equal(Point a, Point b)
{
    return a.x == b.x && a.y == b.y;
}

static void student_print(const Student *s)
{
    printf("  %-10s score %3d  attempts %d\n",
           s->name, s->score, s->attempts);
}

/* An array of structures, searched and aggregated. */
static const Student *find_student(const Student *list, size_t n,
                                   const char *name)
{
    for (size_t i = 0; i < n; i++) {
        if (strcmp(list[i].name, name) == 0) {
            return &list[i];             /* points into the caller's array */
        }
    }
    return NULL;
}

static bool class_average(const Student *list, size_t n, double *out)
{
    if (n == 0) {
        return false;
    }
    long total = 0;
    for (size_t i = 0; i < n; i++) {
        total += list[i].score;
    }
    *out = (double)total / (double)n;
```

```
    return true;
}

/* A self-referential type: the basis of chapter 32. */
typedef struct Node {
    int          value;
    struct Node *next;
} Node;

int main(void)
{
    puts("== points ==");
    Point a = { .x = 1, .y = 2 };
    Point b = point_make(4, 6);
    printf("  a = (%d,%d), b = (%d,%d), manhattan = %d\n",
           a.x, a.y, b.x, b.y, manhattan(a, b));
    printf("  a == b ? %s   (compared member by member)\n",
           point_equal(a, b) ? "yes" : "no");

    Point copy = a;
    copy.x = 99;
    printf("  after copying a and changing the copy: a.x = %d\n", a.x);
    puts("  structures assign by value — unlike arrays");

    puts("\n== nesting, and const versus mutable pointers ==");
    Rect r = {
        .top_left     = { .x = 0, .y = 0 },
        .bottom_right = { .x = 4, .y = 3 }
    };
    printf("  area = %d\n", rect_area(&r));
    rect_translate(&r, 10, 10);
    printf("  after translating: top_left = (%d,%d), area = %d\n",
           r.top_left.x, r.top_left.y, rect_area(&r));
    puts("  the area is unchanged, as it must be");

    puts("\n== an array of structures ==");
    Student class[] = {
        { .name = "Ada",    .score = 95, .attempts = 1 },
        { .name = "Dennis", .score = 88, .attempts = 2 },
```

```
        { .name = "Ken",    .score = 91, .attempts = 1 },
        { .name = "Grace",  .score = 78, .attempts = 3 }
    };
    const size_t n = sizeof class / sizeof class[0];

    for (size_t i = 0; i < n; i++) {
        student_print(&class[i]);
    }

    double average;
    if (class_average(class, n, &average)) {
        printf("  average of %zu students: %.2f\n", n, average);
    }

    const Student *found = find_student(class, n, "Ken");
    printf("  lookup \"Ken\"  : %s\n",
           found ? "found" : "not found");
    if (found != NULL) {
        student_print(found);
    }
    printf("  lookup \"Linus\": %s\n",
           find_student(class, n, "Linus") ? "found" : "not found");

    puts("\n== sizes and copying cost ==");
    printf("  sizeof(Point)   = %2zu\n", sizeof(Point));
    printf("  sizeof(Rect)    = %2zu\n", sizeof(Rect));
    printf("  sizeof(Student) = %2zu   (%d-byte name plus two ints)\n",
           sizeof(Student), NAME_LEN);
    printf("  sizeof(Student *) = %zu — which is why large structures\n",
           sizeof(Student *));
    puts("  are passed by pointer");

    puts("\n== self-referential ==");
    Node third  = { .value = 3, .next = NULL };
    Node second = { .value = 2, .next = &third };
    Node first  = { .value = 1, .next = &second };
    printf("  list: ");
    for (Node *p = &first; p != NULL; p = p->next) {
        printf("%d ", p->value);
```

```
    }
    puts("\n  a struct cannot contain itself, but it can point to itself");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o structs structs.c
./structs
```

Three things the output shows

Structures copy on assignment. Changing `copy.x` leaves `a.x` alone. Arrays do not behave this way — `int b[5] = a;` does not even compile — which is one of the few places structures are more convenient than arrays.

`const` **is doing real work.** Add `r->top_left.x = 0;` inside `rect_area` and the build fails with *assignment of member in read-only object*. The signature promised not to modify, and the promise is checked.

Size drives the calling convention. `Student` is 40 bytes; a pointer is 8. Passing forty students by value copies 1.6 KB; passing pointers copies 320 bytes. For `Point` at 8 bytes the difference is nil, which is why by-value is reasonable there.

The `memcmp` trap, demonstrated

Add this and run it:

```
typedef struct {
    char  flag;      /* 1 byte, then 3 bytes of padding */
    int   value;     /* 4 bytes */
} Padded;

Padded p1, p2;
memset(&p1, 0x00, sizeof p1);        /* clear all bytes, padding included */
memset(&p2, 0xFF, sizeof p2);        /* fill all bytes                    */
p1.flag = 'A'; p1.value = 7;
p2.flag = 'A'; p2.value = 7;         /* logically identical now */

printf("members equal: %s\n",
       (p1.flag == p2.flag && p1.value == p2.value) ? "yes" : "no");
printf("memcmp equal : %s\n",
       memcmp(&p1, &p2, sizeof p1) == 0 ? "yes" : "no");
printf("sizeof(Padded) = %zu, but 1 + 4 = 5\n", sizeof(Padded));
```

```
members equal: yes
memcmp equal : no
sizeof(Padded) = 8, but 1 + 4 = 5
```

Every member matches and `memcmp` still reports a difference, because the three padding bytes after `flag` differ. Chapter 22 explains where those three bytes come from and how to inspect them.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Omitting the semicolon after `}`	Confusing errors on the next declaration	A struct definition is a declaration; it needs one.
Using `.` on a pointer	Compile error, or worse with a cast	Dot for objects, arrow for pointers.
`*p.x` meaning `(*p).x`	Parses as `*(p.x)`	Write `p->x`.
Comparing structures with `==`	Compile error	Compare members in a function.
Comparing structures with `memcmp`	Padding makes equal structures differ	Compare members explicitly.
Passing a large structure by value in a loop	Unnecessary copying	`const T *`.
Positional initializers on a wide struct	Reordering members silently breaks it	Designated initializers.
Forgetting the tag in a self-referential struct	`unknown type name`	`struct Node *next;` inside `struct Node`.

9 Check yourself

Why can you assign one structure to another but not one array to another?

Because a structure is a single object of its type, and C defines assignment for it as a member-by-member copy. An array name decays to a pointer in an assignment context, so `a = b` would be an attempt to assign to a non-modifiable address. Copy arrays with `memcpy` or a loop.

When should a structure be passed by value?

When it is small — two or three scalars — and the function does not need to modify the caller's copy. Beyond that, pass a pointer, with `const` if the function only reads. The pointer avoids copying and makes the read-only contract explicit and compiler-checked.

Why is `memcmp` unreliable for comparing structures?

Because it compares padding bytes as well as members. The compiler inserts padding for alignment, and its contents are unspecified — two structures with identical member values can differ in those bytes and compare unequal. Compare the members you care about individually.

Why must a self-referential structure contain a pointer rather than an instance?

Because a structure containing itself would have infinite size — the compiler could not compute `sizeof`. A pointer has a fixed, known size regardless of what it points to, so the type is well defined. This is exactly why every linked data structure in C is built from pointers.

Why prefer designated initializers over positional ones?

They state which member each value belongs to, so the code is readable without counting and survives a reordering of the declaration. They also default every unmentioned member to zero, making partial initialization explicit rather than accidental. On a structure with more than about three members, positional initialization is a maintenance hazard.

10 Where this leads

Chapter 22 completes the picture of user-defined types: enumerations for a closed set of named values, unions for storing one of several things in the same space, `typedef` in its own right, and the padding rules that explain the `memcmp` result above. Together they give you everything needed to design the abstract data type of chapter 33.

INTERMEDIATE · CHAPTER 22

Unions, Enumerations, and Type Aliases

Three small features that finish C's type system. One of them — the enumeration — buys you a genuine static check the language otherwise never offers. Another — the union — is the only place C lets you interpret the same bytes two ways, which is powerful and requires discipline.

BY THE END OF THIS CHAPTER YOU CAN

- Define enumerations and use the compiler warning they unlock.
- Use `typedef` where it helps and recognize where it hides too much.
- Explain what a union is, and why a bare union is almost always wrong.
- Build a tagged union and access it safely.
- Explain structure padding, compute member offsets, and reorder members to save space.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Enumerations

```
typedef enum {
    STATE_IDLE,        /* 0 */
    STATE_RUNNING,     /* 1 */
    STATE_PAUSED,      /* 2 */
    STATE_STOPPED      /* 3 */
} State;

State s = STATE_IDLE;
```

An enumeration defines a set of named integer constants. Values start at 0 and increase by one unless you say otherwise:

```
typedef enum {
    HTTP_OK        = 200,
    HTTP_NOT_FOUND = 404,
    HTTP_ERROR     = 500
} HttpStatus;

typedef enum {
    FLAG_READ  = 1 << 0,    /* 1 */
    FLAG_WRITE = 1 << 1,    /* 2 */
    FLAG_EXEC  = 1 << 2     /* 4 */
} Permission;
```

The second form gives named bit flags, which chapter 25 combines with the bitwise operators.

The reason to use them

Enumerations are not merely tidier than `#define`. They unlock a compile-time check that C offers nowhere else:

```
const char *describe(State s)
{
    switch (s) {
    case STATE_IDLE:    return "idle";
    case STATE_RUNNING: return "running";
    case STATE_PAUSED:  return "paused";
    /* STATE_STOPPED not handled, and no default */
    }
    return "unknown";
}
```

```
warning: enumeration value 'STATE_STOPPED' not handled in switch [-Wswitch]
```

Add a new state to the enumeration a year later and the compiler points at every `switch` that needs updating. This is genuine static verification, and it is lost the moment you write a `default` label — so on a switch over an enumeration, **omit** `default` **deliberately** and handle the out-of-range case after the switch.

> **What an enum is, underneath.** Its constants have type `int`, and the enumeration type itself is some implementation-chosen integer type large enough to hold them. An `enum` variable can legally hold a value outside the listed set — `(State)99` compiles — so the `switch` warning is a help, not a guarantee. C23 lets you fix the underlying type: `enum State : unsigned char { … }`.

2 `typedef`

`typedef` introduces an alias for an existing type. It creates no new type — it is a naming device, and the compiler treats the alias and the original as identical.

```
typedef unsigned long  Timestamp;
typedef struct Point   Point;
typedef int          (*Comparator)(const void *, const void *);
```

Where it clearly helps

- **Structures**, to avoid writing `struct` at every use.
- **Function pointers**, where the raw declaration is genuinely hard to read — chapter 30 relies on this.
- **Portability**, as a single place to change a width. This is exactly what `<stdint.h>` does with `int32_t`.
- **Opaque handles**, where the caller must not see the definition — chapter 33.

Where it hurts

```
typedef int *IntPtr;
IntPtr a, b;              /* both pointers — surprising, but fine */
const IntPtr p = q;       /* a CONST POINTER, not a pointer to const */
```

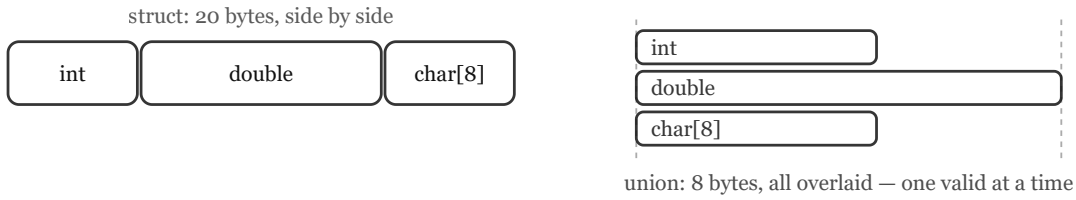
Hiding a `*` behind an alias makes `const` apply to the pointer rather than the target, which is rarely what the reader expects. The standard library avoids pointer typedefs for exactly this reason, and so should you — with the deliberate exception of opaque handles, where hiding is the point.

3 Unions

A union's members all occupy the *same* storage. Its size is that of its largest member, and only one member holds a meaningful value at any moment.

```
union Value {
    int    as_int;
    double as_double;
    char   as_text[16];
};

union Value v;
v.as_int = 42;
printf("%d\n", v.as_int);      /* fine */
printf("%f\n", v.as_double);   /* garbage: you did not store a double */
```



Reading a member you did not write gives an unspecified value, and is how type punning is done.

Reading a member other than the one last written is *unspecified* behavior in C — implementations commonly define it as reinterpreting the bytes, which makes unions the standard way to inspect a value's representation. That is legitimate and useful, and is exactly what chapter 4's float-byte dump did by another route.

What is *not* legitimate is forgetting which member is live. A bare union carries no record of that, so it is almost always wrong on its own.

4 The tagged union

Pair the union with an enumeration recording which member is valid, and wrap both in a structure. This is the disciplined form, and it is how variant types are built in C.

```
typedef enum { VAL_INT, VAL_DOUBLE, VAL_TEXT } ValueKind;

typedef struct {
    ValueKind kind;
    union {
        int    as_int;
        double as_double;
        char   as_text[24];
    } data;
} Value;
```

Every access goes through the tag:

```
void print_value(const Value *v)
{
    switch (v->kind) {
    case VAL_INT:    printf("%d\n",  v->data.as_int);    break;
    case VAL_DOUBLE: printf("%g\n",  v->data.as_double); break;
    case VAL_TEXT:   printf("%s\n",  v->data.as_text);   break;
    }
}
```

And because the switch covers an enumeration with no `default`, adding a fourth kind produces a warning here and at every other site that inspects a `Value`. The tag is what makes the union safe, and the enum is what makes the tag checkable.

C11 allows the union to be anonymous, which removes the `.data` step:

```
typedef struct {
    ValueKind kind;
    union {                     /* no name */
        int    as_int;
        double as_double;
    };
} Value;

v.as_int = 42;                  /* accessed as if it were a direct member */
```

5 Padding, alignment, and `offsetof`

Chapter 21 ended with two structures whose members were all equal comparing unequal under `memcmp`. Here is why.

Every type has an **alignment requirement**: an `int` must usually sit at an address divisible by 4, a `double` by 8. The compiler inserts unnamed *padding* bytes between members to satisfy this, and after the last member so that arrays of the structure stay aligned.

```
struct Wasteful {          /* 24 bytes */
    char   a;              /* offset 0  */
                           /* 7 bytes padding */
    double b;              /* offset 8  */
    char   c;              /* offset 16 */
                           /* 7 bytes padding */
};

struct Tight {             /* 16 bytes */
    double b;              /* offset 0  */
    char   a;              /* offset 8  */
    char   c;              /* offset 9  */
                           /* 6 bytes padding */
};
```

Same three members, one third less space. The rule of thumb: **declare members from largest to smallest**. On a single structure this is irrelevant; on an array of a million records it is megabytes, and chapter 34 shows it is also measurably faster because more records fit in a cache line.

`offsetof`, from `<stddef.h>`, reports where a member actually sits:

```
#include <stddef.h>
printf("%zu\n", offsetof(struct Wasteful, b));   /* 8 */
```

The padding bytes have unspecified contents — nothing guarantees they are initialized, not even `= {0}`. That is what defeated `memcmp`.

6 Worked example: a variant type and a space audit

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>

/* ---------- enumeration with a checkable switch ---------- */

typedef enum {
    TOKEN_NUMBER,
    TOKEN_WORD,
    TOKEN_SYMBOL
} TokenKind;

/* No default label: adding a TokenKind makes the compiler
   point at this function. That is the whole reason for the enum. */
static const char *kind_name(TokenKind k)
{
    switch (k) {
    case TOKEN_NUMBER: return "number";
    case TOKEN_WORD:   return "word";
    case TOKEN_SYMBOL: return "symbol";
    }
    return "unknown";          /* reached only for an out-of-range value */
}

/* ---------- tagged union ---------- */

typedef struct {
    TokenKind kind;
    union {
        double as_number;
        char   as_word[24];
        char   as_symbol;
    } data;
} Token;
```

```
static Token token_number(double v)
{
    Token t = { .kind = TOKEN_NUMBER };
    t.data.as_number = v;
    return t;
}

static Token token_word(const char *w)
{
    Token t = { .kind = TOKEN_WORD };
    snprintf(t.data.as_word, sizeof t.data.as_word, "%s", w);
    return t;
}

static Token token_symbol(char c)
{
    Token t = { .kind = TOKEN_SYMBOL };
    t.data.as_symbol = c;
    return t;
}

static void token_print(const Token *t)
{
    printf("  %-7s ", kind_name(t->kind));
    switch (t->kind) {
    case TOKEN_NUMBER: printf("%g\n",  t->data.as_number); break;
    case TOKEN_WORD:   printf("\"%s\"\n", t->data.as_word); break;
    case TOKEN_SYMBOL: printf("'%c'\n", t->data.as_symbol); break;
    }
}

/* ---------- named bit flags ---------- */

typedef enum {
    PERM_READ  = 1 << 0,
    PERM_WRITE = 1 << 1,
    PERM_EXEC  = 1 << 2
} Permission;
```

```
static void print_permissions(unsigned flags)
{
    printf("  %c%c%c  (0x%02X)\n",
           (flags & PERM_READ)  ? 'r' : '-',
           (flags & PERM_WRITE) ? 'w' : '-',
           (flags & PERM_EXEC)  ? 'x' : '-',
           flags);
}

/* ---------- padding ---------- */

struct Wasteful { char a; double b; char c; };
struct Tight    { double b; char a; char c; };

/* ---------- type punning, the legitimate use of a bare union ---------- */

union FloatBits {
    float    value;
    uint32_t bits;
};

int main(void)
{
    puts("== tagged union ==");
    Token tokens[] = {
        token_number(3.14),
        token_word("hello"),
        token_symbol('+'),
        token_number(42)
    };
    const size_t n = sizeof tokens / sizeof tokens[0];
    for (size_t i = 0; i < n; i++) {
        token_print(&tokens[i]);
    }
    printf("  sizeof(Token) = %zu: a tag plus the largest member\n",
           sizeof(Token));

    puts("\n== what a bare union looks like when you lose track ==");
```

```
    union { int as_int; float as_float; } raw;
    raw.as_int = 1065353216;
    printf("  stored as int   : %d\n", raw.as_int);
    printf("  read as float   : %g   <-- same bytes, different meaning\n",
           (double)raw.as_float);
    puts("  nothing in the union records which member is live");

    puts("\n== bit flags from an enumeration ==");
    unsigned p = PERM_READ | PERM_WRITE;
    print_permissions(p);
    p |= PERM_EXEC;
    print_permissions(p);
    p &= ~(unsigned)PERM_WRITE;
    print_permissions(p);

    puts("\n== padding: same members, different order ==");
    printf("  sizeof(struct Wasteful) = %zu\n", sizeof(struct Wasteful));
    printf("    a at %zu, b at %zu, c at %zu\n",
           offsetof(struct Wasteful, a),
           offsetof(struct Wasteful, b),
           offsetof(struct Wasteful, c));
    printf("  sizeof(struct Tight)    = %zu\n", sizeof(struct Tight));
    printf("    b at %zu, a at %zu, c at %zu\n",
           offsetof(struct Tight, b),
           offsetof(struct Tight, a),
           offsetof(struct Tight, c));
    printf("  the members total %zu bytes; the rest is padding\n",
           sizeof(char) * 2 + sizeof(double));
    printf("  one million records: %zu MB versus %zu MB\n",
           sizeof(struct Wasteful) * 1000000 / 1048576,
           sizeof(struct Tight)    * 1000000 / 1048576);

    puts("\n== why memcmp fails on structures ==");
    struct Wasteful p1, p2;
    memset(&p1, 0x00, sizeof p1);
    memset(&p2, 0xFF, sizeof p2);
    p1.a = 'A'; p1.b = 1.5; p1.c = 'C';
    p2.a = 'A'; p2.b = 1.5; p2.c = 'C';
    printf("  all members equal : %s\n",
```

```
           (p1.a == p2.a && p1.b == p2.b && p1.c == p2.c) ? "yes" : "no");
    printf("  memcmp says equal : %s\n",
           memcmp(&p1, &p2, sizeof p1) == 0 ? "yes" : "no");
    puts("  the padding bytes differ, and memcmp compares them too");

    puts("\n== type punning through a union ==");
    union FloatBits fb;
    fb.value = 1.0f;
    printf("  1.0f has the bit pattern 0x%08X\n", fb.bits);
    fb.bits = 0x40490FDB;
    printf("  0x40490FDB read as a float is %g\n", (double)fb.value);

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o variants variants.c
./variants
```

The program includes `<stdint.h>` for `uint32_t`. Chapter 37 covers those fixed-width types properly.

Provoke the warning that makes enums worthwhile

Add a fourth token kind and rebuild — without touching anything else:

```
typedef enum {
    TOKEN_NUMBER,
    TOKEN_WORD,
    TOKEN_SYMBOL,
    TOKEN_STRING          /* new */
} TokenKind;
```

```
variants.c:20:5: warning: enumeration value 'TOKEN_STRING' not handled
                 in switch [-Wswitch]
variants.c:58:5: warning: enumeration value 'TOKEN_STRING' not handled
                 in switch [-Wswitch]
```

Two warnings, pointing at exactly the two functions that must be updated. Now add `default: return "?";` to `kind_name` and rebuild: one warning disappears — and with it, the guarantee. That is the trade-off, and on a switch over an enumeration it is usually the wrong side to take.

Measure the padding yourself

```
pahole ./variants          # if installed: prints every struct with its holes
```

Without `pahole`, the `offsetof` output above tells the same story: `Wasteful` puts `b` at offset 8 although `a` ended at offset 1, so seven bytes are unused. Reordering recovers eight bytes per record — eight megabytes across a million of them.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Adding `default` to a switch over an enum	Loses the unhandled-value warning	Omit it; handle out-of-range after the switch.
Reading a union member you did not write	Unspecified value	Track the live member with a tag.
A bare union in an interface	Callers cannot know what is valid	Wrap it in a struct with an enum tag.
Assuming `sizeof` equals the sum of members	Off by the padding	Use `sizeof` and `offsetof`; never compute by hand.
Declaring members smallest-first	Wasted space, worse cache use	Largest to smallest.
`typedef int *IntPtr;` then `const IntPtr`	A const pointer, not a pointer to const	Do not hide `*` behind a typedef.
Writing a struct's bytes to a file directly	Padding and layout are not portable	Serialize member by member — chapter 37.
Assuming an enum variable holds only listed values	It can hold any value of the underlying type	Validate input before switching on it.

8 Check yourself

What does an enumeration give you that a set of `#define` constants does not?

A compile-time check. A `switch` over an enumeration with no `default` label warns about any enumerator you failed to handle, so adding a value later makes the compiler list every site that must be updated. Enumerations also respect scope and are visible to a debugger, neither of which is true of macros.

Why is adding a `default` label to a switch over an enum usually a mistake?

Because it makes every value "handled", which switches the `-Wswitch` warning off. You trade a static guarantee for a runtime fallback. Handle the genuinely out-of-range case after the switch instead, so the compiler still checks that every named enumerator has a case.

Why is a bare union almost always wrong in an interface?

Because nothing in it records which member currently holds a valid value, and reading the wrong one yields an unspecified result. The caller would have to track that separately and correctly. Pairing the union with an enum tag inside a structure makes the state explicit and lets the compiler check every access site.

Where does structure padding come from, and how do you avoid most of it?

From alignment requirements: each member must start at an address that is a multiple of its alignment, and the structure is padded at the end so arrays stay aligned. Declaring members from largest to smallest packs them with the fewest gaps — often reducing the size by a third, which matters for large arrays of records.

Why can two structures with identical member values fail a `memcmp`?

Because `memcmp` compares every byte including padding, whose contents are unspecified and typically hold whatever was previously in that memory. Two structures assigned the same member values can still differ in those gaps. Compare the members individually instead.

9 Where this leads

The type system is now complete: scalars, arrays, pointers, structures, unions, enumerations, and aliases. Chapter 23 puts them to work on the outside world — reading arguments from the command line and validating them — and chapter 24 surveys the standard library that has been supplying `printf` and `malloc` all along.

INTERMEDIATE · CHAPTER 23

Program Arguments and Input

A program that only talks to a human at a prompt cannot be scripted, tested, or combined with anything else. This chapter connects your programs to the command line and to standard input — and takes validation seriously, because every one of those inputs comes from outside your control.

BY THE END OF THIS CHAPTER YOU CAN

- Read command-line arguments and explain the structure of `argv`.
- Convert text to numbers with full error detection, and say why `atoi` cannot.
- Reject malformed, out-of-range, and missing input with useful messages.
- Read an arbitrary number of lines from standard input safely.
- Write a program that behaves correctly inside a shell pipeline.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 argc and argv

```
#include <stdio.h>


int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = \"%s\"\n", i, argv[i]);
    }
    return 0;
}
```

```
$ ./prog hello 42 "two words"
argv[0] = "./prog"
argv[1] = "hello"
argv[2] = "42"
argv[3] = "two words"
```

`argv` is an array of pointers to strings — exactly the shape chapter 15 introduced. Four facts about it:

- `argv[0]` is the program name as invoked. Use it in error messages so they identify themselves.
- Real arguments start at `argv[1]`. `argc` is normally at least 1, but the standard allows 0, which is why the worked example checks `argc > 0`.
- `argv[argc]` is guaranteed to be `NULL`, which allows a pointer-walk loop as an alternative to counting.
- **Everything is a string.** The shell has already split the command line on whitespace and removed quotes; `42` arrives as two characters and a terminator, not as an integer.

```
/* two equivalent traversals */
for (int i = 1; i < argc; i++)        { use(argv[i]); }
for (char **p = argv + 1; *p; p++)    { use(*p);      }
```

Always check `argc` before indexing. Reading `argv[1]` when the user supplied no arguments reads the guaranteed `NULL` and then dereferences it.

2 Text to number, done properly

Chapter 16 introduced `strtol`; here is the full argument for it.

	ATOI	STRTOL
Reports "not a number"	No — returns 0	Yes, via `endptr`
Reports trailing rubbish	No — `"12abc"` gives 12	Yes, via `endptr`
Reports overflow	No — undefined behavior	Yes, via `errno`
Handles other bases	No	Yes

`atoi("abc")` returns 0, indistinguishable from `atoi("0")`. There is no combination of checks that recovers the difference. Treat `atoi` as unusable for input you did not generate yourself.

The complete `strtol` pattern

```
#include <stdlib.h>
#include <errno.h>
#include <limits.h>
```

```
bool parse_int(const char *text, int *out)
{
    if (text == NULL || *text == '\0') {
        return false;                       /* empty */
    }

    errno = 0;
    char *end;
    long value = strtol(text, &end, 10);

    if (end == text)                        return false;  /* no digits   */
    if (*end != '\0')                       return false;  /* trailing    */
    if (errno == ERANGE)                    return false;  /* long range  */
    if (value < INT_MIN || value > INT_MAX) return false;  /* int range   */

    *out = (int)value;
    return true;
}
```

Every line earns its place. `errno = 0` before the call is necessary because `strtol` only ever *sets* `errno`; a stale `ERANGE` from an earlier call would otherwise be misread. And the final range check is separate because `strtol` returns `long`, which on Linux is wider than `int` — without it, `"3000000000"` parses successfully and then truncates.

`strtod` follows the identical pattern for floating point, and `strtoul` for unsigned values — with one trap: `strtoul("-1", …)` succeeds and returns `ULONG_MAX`, because negation is applied after conversion. Reject a leading minus yourself.

3 Reading standard input

Chapter 8 established that `scanf` is difficult to use safely. The professional pattern is to read a whole line and then parse it, which separates two concerns that `scanf` conflates.

```
char line[256];
while (fgets(line, sizeof line, stdin) != NULL) {
    line[strcspn(line, "\n")] = '\0';     /* strip the newline */
    process(line);
}
```

`fgets` reads at most `size - 1` characters and always terminates, so it cannot overflow. It returns `NULL` at end of input, which makes the loop condition natural. And because it keeps the trailing newline when the line fits, the `strcspn` idiom removes it: `strcspn(line, "\n")` returns the index of the first newline, or the length if there is none, so the assignment is correct either way.

Detecting an over-long line

If no newline is present, the line was longer than the buffer and the rest is still waiting:

```
if (strchr(line, '\n') == NULL && !feof(stdin)) {
    /* line was truncated: consume the remainder */
    int c;
    while ((c = getchar()) != '\n' && c != EOF) { }
}
```

Silently processing a truncated line is a real defect — in a configuration parser it can turn a long entry into a different, valid one. Chapter 26 covers stream state in full.

4 Behaving well in a pipeline

A Unix tool is expected to follow four conventions. Following them costs nothing and makes your program composable:

CONVENTION	WHY
Results to `stdout`, diagnostics to `stderr`	`prog > out.txt` keeps errors visible
Exit 0 on success, non-zero on failure	`prog && next` works
Read `stdin` when given no file	`cat data | prog` works
Say nothing extra on success	Output can be piped into another tool

The last is the one people find surprising. A program that prints "Processing complete!" to `stdout` has corrupted its own output for any consumer. If the message matters, send it to `stderr`.

```
fprintf(stderr, "%s: cannot open '%s'\n", argv[0], path);
return EXIT_FAILURE;
```

Prefixing with `argv[0]` is the standard form: in a long pipeline, an unattributed error message is nearly useless.

5 Worked example: a numeric filter

A tool that reads numbers from standard input, applies a scale and an offset given as arguments, and prints the results — validating everything and composing properly.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>
#include <limits.h>

static const char *program_name = "scale";

static void usage(void)
{
    fprintf(stderr,
        "usage: %s SCALE OFFSET\n"
        "  reads one number per line from standard input,\n"
        "  prints value * SCALE + OFFSET to standard output\n",
        program_name);
}

/* Full validation: empty, non-numeric, trailing text, out of range. */
static bool parse_double(const char *text, double *out)
{
    if (text == NULL || *text == '\0') {
        return false;
    }
    errno = 0;
    char *end;
    double value = strtod(text, &end);

    if (end == text)     return false;      /* no conversion   */
    while (*end == ' ' || *end == '\t') end++;   /* allow trailing spaces */
    if (*end != '\0')    return false;      /* trailing rubbish */
    if (errno == ERANGE) return false;      /* overflow/underflow */

    *out = value;
```

```
    return true;
}

static bool parse_int(const char *text, int *out)
{
    if (text == NULL || *text == '\0') {
        return false;
    }
    errno = 0;
    char *end;
    long value = strtol(text, &end, 10);

    if (end == text)                        return false;
    if (*end != '\0')                       return false;
    if (errno == ERANGE)                    return false;
    if (value < INT_MIN || value > INT_MAX) return false;

    *out = (int)value;
    return true;
}

int main(int argc, char *argv[])
{
    if (argc > 0 && argv[0] != NULL) {
        program_name = argv[0];
    }

    if (argc != 3) {
        fprintf(stderr, "%s: expected 2 arguments, got %d\n",
                program_name, argc - 1);
        usage();
        return EXIT_FAILURE;
    }

    double scale, offset;
    if (!parse_double(argv[1], &scale)) {
        fprintf(stderr, "%s: SCALE is not a number: \"%s\"\n",
                program_name, argv[1]);
        return EXIT_FAILURE;
```

```
    }
    if (!parse_double(argv[2], &offset)) {
        fprintf(stderr, "%s: OFFSET is not a number: \"%s\"\n",
                program_name, argv[2]);
        return EXIT_FAILURE;
    }

    char line[256];
    long line_number = 0;
    long accepted = 0, rejected = 0;

    while (fgets(line, sizeof line, stdin) != NULL) {
        line_number++;

        /* Detect a line too long for the buffer. */
        if (strchr(line, '\n') == NULL && !feof(stdin)) {
            fprintf(stderr, "%s: line %ld too long, skipping\n",
                    program_name, line_number);
            int c;
            while ((c = getchar()) != '\n' && c != EOF) { }
            rejected++;
            continue;
        }

        line[strcspn(line, "\n")] = '\0';

        if (line[0] == '\0' || line[0] == '#') {
            continue;                       /* blank line or comment */
        }

        double value;
        if (!parse_double(line, &value)) {
            fprintf(stderr, "%s: line %ld: not a number: \"%s\"\n",
                    program_name, line_number, line);
            rejected++;
            continue;
        }

        printf("%g\n", value * scale + offset);   /* result to stdout */
```

```
        accepted++;
    }

    if (ferror(stdin)) {
        fprintf(stderr, "%s: error reading input\n", program_name);
        return EXIT_FAILURE;
    }

    fprintf(stderr, "%s: %ld accepted, %ld rejected\n",
            program_name, accepted, rejected);

    return rejected == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o scale scale.c
```

Exercise it from the shell

```
printf '1\n2\n3\n' | ./scale 2 10
```

```
12
14
16
scale: 3 accepted, 0 rejected
```

Note that the summary went to `stderr`, so it appeared on the terminal but is not part of the output. Prove it:

```
printf '1\n2\n3\n' | ./scale 2 10 > results.txt
cat results.txt
```

```
12
14
16
```

The results file contains only numbers — which is what lets the tool be chained:

```
printf '1\n2\n3\n' | ./scale 2 0 | ./scale 10 0 | paste -sd+ | bc
```

Every failure mode, checked

```
./scale                        ; echo "status $?"   # too few arguments
./scale 2                      ; echo "status $?"   # still too few
./scale two 10                 ; echo "status $?"   # bad SCALE
printf 'x\n' | ./scale 2 10    ; echo "status $?"   # bad data line
printf '' | ./scale 2 10       ; echo "status $?"   # empty input
printf '1e400\n' | ./scale 2 0 ; echo "status $?"   # overflow
python3 -c "print('1' * 500)" | ./scale 1 0 ; echo "status $?"   # over-long line
```

INPUT	EXPECTED
No arguments	Usage on `stderr`, status 1
`two` as SCALE	Named error, status 1
A non-numeric data line	Line number reported, other lines still processed
Empty input	No output, status 0 — zero lines is not an error
`1e400`	Rejected via `ERANGE`, not silently turned into infinity
A 500-character line	Reported as too long; the remainder is discarded, not misparsed

Why the over-long line matters

Remove the length check and feed it a long line:

```
python3 -c "print('1' * 300)" | ./scale 1 0
```

Without the check, `fgets` returns the first 255 characters — a perfectly valid number — and the remaining 45 characters become a *second* line, also a valid number. One input line silently becomes two outputs. In a billing system or a sensor log that is a data-corruption bug, not a formatting one.

The `atof` comparison

Replace `parse_double` with `atof` and try the failure cases again. `atof("two")` returns 0.0 and `atof("1e400")` returns infinity, both silently. Every rejection in the table above becomes an accepted wrong answer.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Reading `argv[1]` without checking `argc`	Dereferences `NULL`	Check `argc` first, always.
Using `atoi` on user input	Failure indistinguishable from 0	`strtol` with `endptr` and `errno`.
Forgetting `errno = 0` before `strtol`	A stale `ERANGE` misreported	Clear it immediately before the call.
Not range-checking the `long` result	Silent truncation into `int`	Compare against `INT_MIN` / `INT_MAX`.
Accepting `"12abc"`	Trailing rubbish ignored	Require `*end == '\0'`.
Diagnostics on `stdout`	Corrupts piped output	Everything but results goes to `stderr`.
Always returning 0	Scripts cannot detect failure	Non-zero on any error.
Ignoring a truncated `fgets` line	One line silently becomes two records	Check for the newline and drain the rest.

7 Check yourself

What is `argv[argc]`, and why is it useful?

It is guaranteed to be `NULL`. That lets you walk the arguments with a pointer loop terminating on the null entry instead of counting, and it means an out-of-range read of `argv[argc]` yields `NULL` rather than garbage — though dereferencing it is still a crash, so check `argc` before indexing.

Why can't `atoi` be made safe with extra checks?

Because it discards the information you would need. It returns 0 both for the string `"0"` and for anything unparseable, stops silently at the first non-digit, and has undefined behavior on overflow. No check applied afterwards can recover which case occurred. `strtol` reports all three through `endptr` and `errno`.

Why must `errno` be set to 0 before calling `strtol`?

Because library functions only ever set `errno`; nothing clears it. A value left over from an unrelated earlier failure would still be there, so a successful conversion could appear to have overflowed. Clearing it immediately before the call makes the subsequent test meaningful.

Why should a successful program print nothing beyond its results?

Because `stdout` is the channel another program reads. A "done!" line becomes a record in the next tool's input and corrupts it. Progress and summary messages belong on `stderr`, which stays on the terminal when output is redirected and is ignored by a pipeline.

What goes wrong if you ignore that `fgets` may have truncated a line?

The remainder of the line is still in the stream and is returned by the next call, so one input line becomes two apparently valid records. If both halves happen to parse — as they do for a long run of digits — the program produces wrong output with no error at all. Check for the newline and discard the remainder explicitly.

8 Where this leads

Chapter 24 surveys the rest of the standard library — the mathematics, time, and random-number facilities you have not needed until now — and teaches you to read its documentation rather than memorize it. Chapter 31 then returns to the command line with `getopt`, environment variables, and layered configuration, turning the filter you just wrote into a properly optioned tool.

INTERMEDIATE · CHAPTER 24

The Standard Library Toolbox

C's standard library is small by modern standards — about thirty headers, no containers, no networking, no string builder. Knowing precisely what is in it, and what is deliberately not, saves you from reinventing what exists and from expecting what never will.

BY THE END OF THIS CHAPTER YOU CAN

- Use `<math.h>` correctly, including linking it and handling its error cases.
- Work with calendar time and measure elapsed time with the right clock.
- Generate pseudo-random numbers without the modulo bias, and say when not to use them.
- Name what the standard library does not provide and where to look instead.
- Read a manual page and the text of the standard to answer a question definitively.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The shape of the library

HEADER	PROVIDES	MET IN
`<stdio.h>`	Formatted and file I/O	Chapters 3, 8, 26
`<stdlib.h>`	Allocation, conversion, exit, sorting, random	Chapters 19, 23
`<string.h>`	String and block memory operations	Chapter 14
`<math.h>`	Floating-point mathematics	This chapter
`<time.h>`	Calendar time and clocks	This chapter
`<ctype.h>`	Character classification	Chapter 14
`<errno.h>`	Error reporting	Chapters 23, 27
`<limits.h>` `<float.h>`	Type ranges	Chapter 5
`<stdint.h>`	Fixed-width types	Chapter 37
`<assert.h>`	Runtime assertions	Chapter 27
`<stdarg.h>`	Variable argument lists	Chapter 30
`<threads.h>` `<stdatomic.h>`	Concurrency	Chapter 41

That is essentially all of it. Anything else — a hash table, a socket, a regular expression, a JSON parser — comes from the operating system, from a third-party library, or from you.

2 `<math.h>`

```
#include <math.h>


sqrt(x)   pow(x, y)   fabs(x)   fmod(x, y)
sin(x)    cos(x)      tan(x)    atan2(y, x)
exp(x)    log(x)      log2(x)   log10(x)
floor(x)  ceil(x)     round(x)  trunc(x)
fmin(a,b) fmax(a,b)   hypot(a,b)
```

All take and return `double`. Variants suffixed `f` and `l` work on `float` and `long double`.

Linking

```
gcc -std=c17 -Wall -Wextra prog.c -o prog -lm
```

On Linux the mathematics functions live in a separate library and the linker must be told. Omit `-lm` and you get `undefined reference to 'sqrt'` — chapter 2's distinction between declaration and definition, in its most common practical form. Note that `-lm` goes *after* the source files; linkers process arguments in order.

Error handling

```
errno = 0;
double r = sqrt(-1.0);
if (errno == EDOM) {
    /* domain error: the input was outside the function's domain */
}
if (isnan(r)) { … }
```

Mathematics functions report failure two ways: by setting `errno` to `EDOM` (bad input) or `ERANGE` (result out of range), and by returning `NaN` or infinity. Chapter 7's `isnan` and `isinf` are usually the more convenient test.

> `pow` **is not for small integer powers.** `pow(x, 2)` is a general-purpose call using logarithms; `x * x` is one instruction and exact. Worse, `pow` can return 7.999999999 for `pow(2, 3)` on some implementations, so `(int)pow(2, 3)` may be 7 — truncation of a value just below 8. Use multiplication for small powers, and `lround` if you must convert.

3 `<time.h>`

Two distinct jobs, easily confused: *what time is it* and *how long did that take*.

Calendar time

```
time_t now = time(NULL);              /* seconds since the epoch */

struct tm *local = localtime(&now);   /* broken down, local zone */
struct tm *utc   = gmtime(&now);      /* broken down, UTC        */

char buffer[64];
strftime(buffer, sizeof buffer, "%Y-%m-%d %H:%M:%S", local);
```

`struct tm` has a trap of its own: `tm_year` counts from 1900 and `tm_mon` counts from 0. Printing `tm->tm_year` directly gives 125 for 2025.

`localtime` and `gmtime` return a pointer to a **static** buffer, which the next call overwrites — chapter 17's static-duration design, with chapter 41's reentrancy consequences. Two calls in one `printf` give the same result twice. POSIX provides `localtime_r`, which writes into a buffer you supply.

Measuring elapsed time

```
clock_t start = clock();
do_work();
double cpu_seconds = (double)(clock() - start) / CLOCKS_PER_SEC;
```

`clock` measures *processor* time consumed by your program, not wall-clock time. A program that sleeps for ten seconds uses almost no CPU time. For wall-clock duration use `time` for second resolution, or the POSIX `clock_gettime(CLOCK_MONOTONIC, …)` for nanoseconds.

MEASURING	USE
CPU consumed	`clock()`
Wall-clock, coarse	`time()`, `difftime()`
Wall-clock, precise	`clock_gettime(CLOCK_MONOTONIC, …)` — POSIX
A date to display	`localtime` + `strftime`

`CLOCK_MONOTONIC` rather than the wall clock is the right choice for intervals, because the wall clock can jump backwards when the system synchronizes time. Chapter 34 uses it for benchmarking.

4 Pseudo-random numbers

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

srand((unsigned)time(NULL));     /* seed once, at start */
int r = rand();                  /* 0 .. RAND_MAX */
```

Three things to know, in order of how often they cause trouble.

Seed once. Calling `srand` inside a loop reseeds the generator, and since `time` has one-second resolution, every call in the same second produces the identical sequence. The classic symptom is a "random" list where every value is the same.

`rand() % n` **is biased.** If `RAND_MAX + 1` is not a multiple of `n`, the lower values occur slightly more often. For a die roll with a large `RAND_MAX` the bias is negligible; for a large `n` it is not. The unbiased method rejects the uneven tail:

```
int uniform(int n)               /* 0 .. n-1, no bias */
{
    int limit = RAND_MAX - (RAND_MAX % n);
    int value;
    do {
        value = rand();
    } while (value >= limit);
    return value % n;
}
```

It is not secure. `rand` is a simple deterministic generator; given a few outputs, the rest are predictable. Never use it for passwords, tokens, keys, or anything an adversary would benefit from guessing. Use the operating system's source — `getrandom` on Linux, `arc4random_buf` on the BSDs and macOS. Chapter 48 revisits this under security review.

For reproducible tests, seed with a fixed constant. A test that fails only sometimes is worse than no test.

5 What is not in the library — and how to find out

NOT PROVIDED	WHERE TO GET IT
Growable arrays, lists, hash tables	Write them — chapters 19, 32, 33
Sockets, threads before C11, processes	POSIX — chapters 39–41
Regular expressions	POSIX `<regex.h>`, or PCRE
JSON, XML, compression, TLS	Third-party libraries
A safe string builder	`snprintf`, or write one
Unicode beyond wide characters	ICU, or handle UTF-8 yourself — chapter 37
Testing framework	Third-party — chapter 35

Reading the documentation

Three sources, in increasing order of authority.

Manual pages tell you what your system does:

```
man 3 strtol        # section 3 is library functions
man 3 printf
man -k random       # search by keyword
```

Read the RETURN VALUE and ERRORS sections first — they answer most questions, and they are the ones people skip.

cppreference distinguishes what each standard version guarantees, which matters when you target C17 but read about a C23 feature.

The standard itself is the final authority. Drafts are freely available from the WG14 site linked in the course resources; N3220 is the current C23 working draft. It is dense but surprisingly navigable: section 7 is the library, and each function has a short Description, Returns, and sometimes a Recommended practice.

> **When to reach for the standard.** When a question is about what is *guaranteed* rather than what happens to work. "Is `char` signed?" — the standard says implementation-defined. "Does `free(NULL)` work?" — the standard says yes, explicitly. Testing your own compiler answers neither question, because it tells you about one implementation. Chapters 36 and 37 depend heavily on this distinction.

6 Worked example: generate, measure, report

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <errno.h>

#define SAMPLES 100000
#define BUCKETS 10

/* Unbiased 0..n-1, by rejecting the uneven tail of the range. */
static int uniform(int n)
{
    int limit = RAND_MAX - (RAND_MAX % n);
    int value;
    do {
```

```
        value = rand();
    } while (value >= limit);
    return value % n;
}

static double mean_of(const double *v, size_t n)
{
    double total = 0.0;
    for (size_t i = 0; i < n; i++) {
        total += v[i];
    }
    return total / (double)n;
}

static double stddev_of(const double *v, size_t n, double mean)
{
    double sum_sq = 0.0;
    for (size_t i = 0; i < n; i++) {
        double d = v[i] - mean;
        sum_sq += d * d;
    }
    return sqrt(sum_sq / (double)n);
}

int main(void)
{
    /* --- calendar time --- */
    puts("== what time is it ==");
    time_t now = time(NULL);
    struct tm local = *localtime(&now);     /* copy: the buffer is static */
    struct tm utc   = *gmtime(&now);

    char stamp[64];
    strftime(stamp, sizeof stamp, "%Y-%m-%d %H:%M:%S", &local);
    printf("  local : %s\n", stamp);
    strftime(stamp, sizeof stamp, "%Y-%m-%d %H:%M:%S UTC", &utc);
    printf("  utc   : %s\n", stamp);
    printf("  epoch : %lld seconds\n", (long long)now);
    printf("  note tm_year is %d (years since 1900) and tm_mon is %d (0-based)\n",
```

```
           local.tm_year, local.tm_mon);

    /* --- random numbers --- */
    puts("\n== pseudo-random numbers ==");
    srand((unsigned)now);                   /* seeded ONCE */
    printf("  RAND_MAX = %d\n", RAND_MAX);

    int buckets[BUCKETS] = { 0 };
    double *values = malloc(SAMPLES * sizeof *values);
    if (values == NULL) {
        return EXIT_FAILURE;
    }

    clock_t t0 = clock();
    for (size_t i = 0; i < SAMPLES; i++) {
        int b = uniform(BUCKETS);
        buckets[b]++;
        values[i] = (double)rand() / RAND_MAX;   /* 0.0 .. 1.0 */
    }
    double gen_cpu = (double)(clock() - t0) / CLOCKS_PER_SEC;

    puts("  distribution across 10 buckets (expect about 10000 each):");
    for (int b = 0; b < BUCKETS; b++) {
        printf("    %d: %6d  ", b, buckets[b]);
        for (int bar = 0; bar < buckets[b] / 400; bar++) putchar('#');
        putchar('\n');
    }

    /* --- statistics, using math.h --- */
    puts("\n== statistics ==");
    double m = mean_of(values, SAMPLES);
    double s = stddev_of(values, SAMPLES, m);
    printf("  samples        : %d\n", SAMPLES);
    printf("  mean           : %.6f   (expect 0.5)\n", m);
    printf("  std deviation  : %.6f   (expect 0.2887)\n", s);
    printf("  theoretical sd : %.6f\n", 1.0 / sqrt(12.0));

    /* --- measuring time properly --- */
    puts("\n== cpu time versus wall clock ==");
```

```
    printf("  generating %d samples took %.4f CPU seconds\n",
           SAMPLES, gen_cpu);

    t0 = clock();
    time_t wall_start = time(NULL);
    double waste = 0.0;
    for (long i = 0; i < 20000000L; i++) {
        waste += sqrt((double)i);
    }
    double busy_cpu  = (double)(clock() - t0) / CLOCKS_PER_SEC;
    double busy_wall = difftime(time(NULL), wall_start);
    printf("  busy loop: %.4f CPU seconds, %.0f wall seconds (sum %.0f)\n",
           busy_cpu, busy_wall, waste);
    puts("  for a CPU-bound loop these agree; for a sleeping or blocked");
    puts("  program clock() would report almost nothing");

    /* --- math.h traps --- */
    puts("\n== math.h details ==");
    printf("  sqrt(2)       = %.10f\n", sqrt(2.0));
    printf("  pow(2, 10)    = %g\n", pow(2.0, 10.0));
    printf("  2*2*2 by hand = %d   (prefer this for small powers)\n", 2*2*2);
    printf("  fmod(7.5, 2)  = %g   (%% does not work on doubles)\n",
           fmod(7.5, 2.0));
    printf("  round(2.5)    = %g, trunc(2.5) = %g, floor(-2.5) = %g\n",
           round(2.5), trunc(2.5), floor(-2.5));

    errno = 0;
    double bad = sqrt(-1.0);
    printf("  sqrt(-1)      = %f, isnan -> %d, errno==EDOM -> %d\n",
           bad, isnan(bad), errno == EDOM);

    free(values);
    return EXIT_SUCCESS;
}
```

Build it with `-lm`:

```
gcc -std=c17 -Wall -Wextra -g -o toolbox toolbox.c -lm
./toolbox
```

Three experiments

Forget `-lm`. Build without it:

```
gcc -std=c17 -Wall -Wextra -o toolbox toolbox.c
```

```
/usr/bin/ld: /tmp/cc8Xk2.o: undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

The header declared it; the linker could not find the body. Exactly chapter 2's four-phase model, and the most common form this error takes in practice.

Reseed inside the loop. Move `srand((unsigned)time(NULL));` into the sample loop and rerun. Every bucket but one drops to zero: within a single second, `time` returns the same value, so the generator restarts identically every iteration.

Compare biased and unbiased. Replace `uniform(BUCKETS)` with `rand() % BUCKETS`. With `RAND_MAX` at 2 147 483 647 and ten buckets the bias is far below the noise, so the histograms look identical — which is the honest result. Now try a large `n`, such as two thirds of `RAND_MAX`, and count how many results fall in the lower half instead of filling buckets: the skew becomes visible. Knowing when a correction matters is as useful as knowing the correction.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Forgetting `-lm`	`undefined reference to 'sqrt'`	Add it after the source files.
`srand` inside a loop	The same value repeatedly	Seed once at program start.
`rand()` for tokens or keys	Predictable output; a security hole	Use the operating system's random source.
`(int)pow(2, 3)`	May be 7 through rounding	Multiply for small powers, or `lround`.
Keeping the pointer from `localtime`	Overwritten by the next call	Copy the `struct tm`, or use `localtime_r`.
Printing `tm_year` directly	125 instead of 2025	Add 1900; add 1 to `tm_mon`.
`clock()` to measure a sleeping program	Reports almost zero	`clock_gettime(CLOCK_MONOTONIC, …)`.
Assuming the library has containers	Wasted search	It does not. Write them, or take a dependency.

8 Check yourself

Why does `#include <math.h>` not remove the need for `-lm`?

Because a header supplies declarations, not definitions. The compiler learns the signature of `sqrt` and type-checks the call; the function's machine code lives in the maths library, which the linker only searches when told. This is the same declaration-versus-definition split that chapter 2 traced through the four translation phases.

Why does reseeding inside a loop produce identical values?

`srand` resets the generator to a starting state determined entirely by the seed. Seeding with `time(NULL)`, which changes once per second, restarts the same sequence on every iteration within that second — so each call to `rand` returns the first value of that sequence. Seed exactly once.

What is wrong with `rand() % 6` for a die roll, and how much does it matter?

If `RAND_MAX + 1` is not divisible by 6, the low faces occur slightly more often, because the surplus values at the top of the range map onto them. With `RAND_MAX` above two billion the bias is far below measurement noise for six outcomes; it becomes significant only when the range is a substantial fraction of `RAND_MAX`. The unbiased fix is to reject the uneven tail before taking the remainder.

When does `clock()` give a misleading answer?

Whenever the program is not doing work — sleeping, waiting on input, blocked on a network read. `clock` measures processor time consumed, so a program that waits ten seconds reports near zero. For elapsed wall-clock time use `time` / `difftime` for coarse resolution or `clock_gettime(CLOCK_MONOTONIC, …)` for precision.

When should you consult the standard rather than test your compiler?

Whenever the question is what is *guaranteed* rather than what happens to work here. Whether `char` is signed, whether `free(NULL)` is safe, whether signed overflow wraps — testing one implementation answers none of these, because a different compiler or optimization level may behave differently. Chapters 36 and 37 are built entirely on that distinction.

9 Where this leads

Chapter 25 drops to the level below the library: the bitwise operators, masks, and bit-fields that let you manipulate individual bits. That is the foundation for the flag handling in chapter 31, the checksums of chapter 37, and the hardware registers of chapter 50 — and it is where chapter 4's binary representation finally becomes something you operate on directly.

INTERMEDIATE · CHAPTER 25

Bit-Level Operations

Chapter 4 showed what a byte contains. This chapter you operate on it directly — which is how permissions, hardware registers, compression, hashing, and network protocols are all expressed, and why C remains the language they are written in.

BY THE END OF THIS CHAPTER YOU CAN

- Use the six bitwise operators and predict the result on paper.
- Set, clear, toggle, and test individual bits with the standard idioms.
- Explain the difference between arithmetic and logical right shift, and when each applies.
- Pack several small values into one integer and extract them again.
- Say why bit-fields are convenient and why portable code avoids them.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The operators

OPERATOR	MEANING	EXAMPLE (4-BIT)
`&`	AND — 1 only if both are 1	`1100 & 1010 = 1000`
`|`	OR — 1 if either is 1	`1100 | 1010 = 1110`
`^`	XOR — 1 if they differ	`1100 ^ 1010 = 0110`
`~`	NOT — flip every bit	`~1100 = 0011`
`<<`	Shift left	`0011 << 1 = 0110`
`>>`	Shift right	`1100 >> 1 = 0110`

These are not the logical operators. `&` works on every bit independently and yields a number; `&&` treats its operands as true or false and yields 0 or 1. `1 & 2` is 0; `1 && 2` is 1. Confusing them compiles cleanly and produces wrong answers.

Shifting is multiplication and division by powers of two

```
x << n      /* x * 2^n */
x >> n      /* x / 2^n, for non-negative x */
```

Write the multiplication. Any compiler since the 1990s emits a shift for `x * 8`, and `x * 8` says what you mean. Use shifts when you are genuinely manipulating bit positions, not as an optimization.

The shift traps

- Shifting by the width of the type or more is **undefined behavior**. `x << 32` on a 32-bit `int` is not zero; it is undefined, and in practice x86 shifts by `32 % 32 = 0`, leaving `x` unchanged.
- Shifting a negative value left is undefined.
- Right-shifting a *negative* value is implementation-defined: it may fill with zeros (logical) or with the sign bit (arithmetic). Every mainstream compiler does the arithmetic shift, which preserves the sign — but it is not guaranteed.

The rule that follows: **do bit manipulation on unsigned types**. `unsigned`, or better the fixed-width `uint32_t` from chapter 37. Signedness has no meaning when you are treating a value as a row of bits, and it introduces three ways to go wrong.

2 The four idioms

Almost all bit manipulation is one of these four, written against a mask that selects the bits you care about.

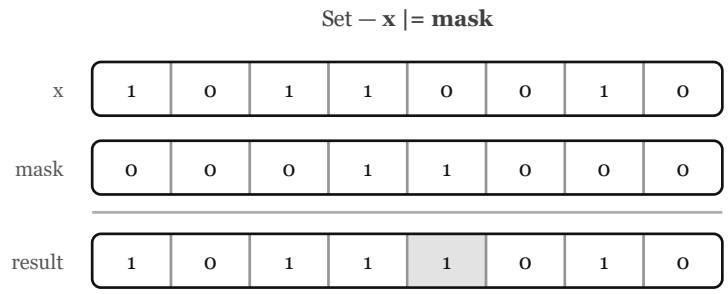
```
#define FLAG_READ   (1u << 0)     /* 0001 */
#define FLAG_WRITE  (1u << 1)     /* 0010 */
#define FLAG_EXEC   (1u << 2)     /* 0100 */

unsigned flags = 0;

flags |=  FLAG_READ;              /* SET    */
flags &= ~FLAG_READ;              /* CLEAR  */
flags ^=  FLAG_READ;              /* TOGGLE */
if (flags & FLAG_READ) { … }      /* TEST   */
```

OPERATION	IDIOM	WHY IT WORKS
Set	`x |= mask`	OR with 1 forces 1; OR with 0 leaves alone
Clear	`x &= ~mask`	AND with 0 forces 0; AND with 1 leaves alone
Toggle	`x ^= mask`	XOR with 1 flips; XOR with 0 leaves alone
Test	`x & mask`	Non-zero exactly when a selected bit is set

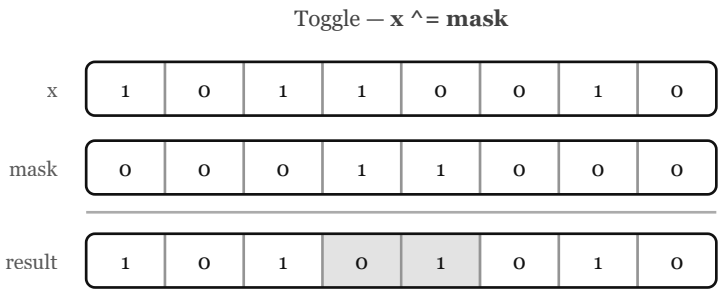
Building masks: `1u << n` selects bit `n`. Several bits are combined with `|`. A mask of the low `n` bits is `(1u << n) - 1`.



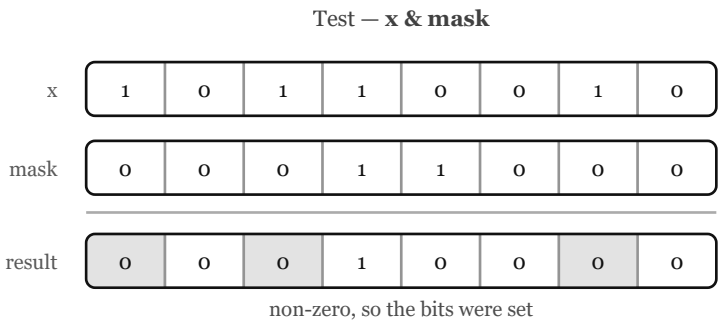
Set. OR with 1 forces a 1; OR with 0 leaves the bit alone. Only the two bits the mask selects can change.



Clear. AND with 0 forces a 0; AND with 1 leaves the bit alone. Hence the inverted mask — you want zeros exactly where the mask had ones.



Toggle. XOR with 1 flips; XOR with 0 leaves alone. Twice returns the original, which is why XOR turns up in cheap encodings and in swap tricks.



Test. AND keeps only the selected bits and discards the rest. The result is non-zero exactly when at least one selected bit was set — so compare against 0, or against the mask if you need *all* of them.

Parenthesize. Chapter 6's precedence table puts `&`, `^`, and `|` *below* the comparison operators — a design mistake preserved for compatibility. So `flags & FLAG_READ == 0` parses as `flags & (FLAG_READ == 0)`, which is `flags & 0`, which is 0, which is false, always. Write `(flags & FLAG_READ) == 0`. GCC warns with `-Wparentheses`; do not ignore it.

Testing several bits

```
if ((flags & (FLAG_READ | FLAG_WRITE)) == (FLAG_READ | FLAG_WRITE)) {
    /* BOTH are set */
}
if (flags & (FLAG_READ | FLAG_WRITE)) {
    /* at least ONE is set */
}
```

The difference is easy to get wrong and easy to state: comparing against the mask tests "all", comparing against zero tests "any".

3 Packing values into one integer

When several small quantities travel together — a colour, a date, a hardware register — they can share one word. Two operations: pack with shift-and-OR, unpack with shift-and-mask.

```
/* Pack an RGB colour into 24 bits. */
uint32_t rgb = ((uint32_t)r << 16) | ((uint32_t)g << 8) | b;

/* Unpack. */
uint8_t red   = (rgb >> 16) & 0xFF;
uint8_t green = (rgb >>  8) & 0xFF;
uint8_t blue  =  rgb        & 0xFF;
```

The mask after the shift is essential: without `& 0xFF`, `rgb >> 8` still carries the red bits above the green ones.

This is exactly how a hardware register is written in chapter 50, how an instruction is encoded, and how a network protocol header is laid out. The pattern does not change; only the field widths do.

4 Bit-fields

C offers syntax for this directly:

```
struct Flags {
    unsigned read  : 1;      /* one bit  */
    unsigned write : 1;
    unsigned level : 3;      /* three bits, values 0-7 */
    unsigned       : 0;      /* force alignment to the next unit */
};

struct Flags f = { .read = 1, .level = 5 };
if (f.read) { … }
```

Readable, type-checked, and the compiler does the shifting. So why does most portable C avoid them?

NOT SPECIFIED BY THE STANDARD	CONSEQUENCE
Whether fields are allocated low-to-high or high-to-low	Layout differs between compilers
Whether a field may straddle a storage unit	Sizes differ
The alignment of the allocation unit	Padding differs
Whether plain `int` fields are signed	A 1-bit `int` field can hold 0 and −1

So a bit-field struct cannot be used to describe an external binary format — a file header, a network packet, a hardware register — because the layout is not guaranteed to match. Use explicit masks and shifts for anything that crosses a boundary, and reserve bit-fields for internal storage where only your own code sees them.

Note also the last item: always write `unsigned` or `signed` explicitly on a bit-field. Plain `int` has implementation-defined signedness here, and a one-bit signed field can hold only 0 and −1, which surprises everyone once.

5 Worked example: permissions, packing, and popcount

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <time.h>

/* ---------- named flags ---------- */

typedef enum {
    PERM_READ    = 1u << 0,
    PERM_WRITE   = 1u << 1,
    PERM_EXEC    = 1u << 2,
    PERM_DELETE  = 1u << 3,
    PERM_ALL     = 0x0Fu
} Permission;

static void print_bits(uint32_t value, int width)
{
    for (int bit = width - 1; bit >= 0; bit--) {
        putchar((value >> bit) & 1u ? '1' : '0');
        if (bit % 4 == 0 && bit != 0) putchar(' ');
```

```
    }
}

static void show_permissions(const char *label, unsigned flags)
{
    printf("  %-14s %c%c%c%c  ", label,
           (flags & PERM_READ)   ? 'r' : '-',
           (flags & PERM_WRITE)  ? 'w' : '-',
           (flags & PERM_EXEC)   ? 'x' : '-',
           (flags & PERM_DELETE) ? 'd' : '-');
    print_bits(flags, 8);
    printf("  (0x%02X)\n", flags);
}

/* ---------- packing ---------- */

static uint32_t rgb_pack(uint8_t r, uint8_t g, uint8_t b)
{
    return ((uint32_t)r << 16) | ((uint32_t)g << 8) | (uint32_t)b;
}

static void rgb_unpack(uint32_t rgb, uint8_t *r, uint8_t *g, uint8_t *b)
{
    *r = (uint8_t)((rgb >> 16) & 0xFFu);
    *g = (uint8_t)((rgb >>  8) & 0xFFu);
    *b = (uint8_t)( rgb        & 0xFFu);
}

/* ---------- counting set bits, three ways ---------- */

static int popcount_naive(uint32_t v)
{
    int count = 0;
    for (int i = 0; i < 32; i++) {
        count += (int)((v >> i) & 1u);
    }
    return count;                          /* always 32 iterations */
}
```

```
/* Kernighan's method: v & (v-1) clears the lowest set bit,
   so the loop runs once per SET bit rather than once per bit. */
static int popcount_kernighan(uint32_t v)
{
    int count = 0;
    while (v != 0) {
        v &= v - 1;
        count++;
    }
    return count;
}

/* ---------- a simple checksum ---------- */

static uint32_t checksum(const uint8_t *data, size_t n)
{
    uint32_t sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum ^= data[i];
        sum = (sum << 1) | (sum >> 31);    /* rotate left by one */
    }
    return sum;
}

/* ---------- bit-fields, for internal use only ---------- */

struct Packed {
    unsigned read  : 1;
    unsigned write : 1;
    unsigned level : 3;
};

int main(void)
{
    puts("== the four idioms ==");
    unsigned flags = 0;
    show_permissions("start", flags);

    flags |= PERM_READ;                   show_permissions("set read", flags);
```

```
    flags |= PERM_WRITE | PERM_EXEC;      show_permissions("set w+x", flags);
    flags &= ~(unsigned)PERM_WRITE;       show_permissions("clear write", flags);
    flags ^= PERM_EXEC;                   show_permissions("toggle exec", flags);
    flags |= PERM_ALL;                    show_permissions("set all", flags);

    puts("\n== any versus all ==");
    unsigned rw = PERM_READ | PERM_WRITE;
    unsigned only_read = PERM_READ;
    printf("  only_read has ANY of r|w : %s\n",
           (only_read & rw) ? "yes" : "no");
    printf("  only_read has ALL of r|w : %s\n",
           ((only_read & rw) == rw) ? "yes" : "no");

    puts("\n== the precedence trap ==");
    unsigned f = 0x0A;
    printf("  f & PERM_EXEC == 0    gives %u   <-- parses as f & (EXEC==0)\n",
           f & PERM_EXEC == 0);
    printf("  (f & PERM_EXEC) == 0  gives %d   <-- what you meant\n",
           (f & PERM_EXEC) == 0);

    puts("\n== packing and unpacking ==");
    uint32_t colour = rgb_pack(255, 128, 64);
    printf("  rgb(255,128,64) = 0x%06X  ", colour);
    print_bits(colour, 24);
    putchar('\n');
    uint8_t r, g, b;
    rgb_unpack(colour, &r, &g, &b);
    printf("  unpacked        = (%u, %u, %u)\n", r, g, b);
    printf("  without the mask, (colour >> 8) = 0x%X  <-- red bits still there\n",
           colour >> 8);

    puts("\n== shifting ==");
    uint32_t v = 0x0000000Fu;
    printf("  0x0F << 4  = 0x%08X   (multiply by 16)\n", v << 4);
    printf("  0xF0 >> 4  = 0x%08X   (divide by 16)\n", 0xF0u >> 4);
    int negative = -16;
    printf("  -16 >> 2   = %d   (arithmetic shift here, but not guaranteed)\n",
           negative >> 2);
    puts("  do bit work on unsigned types and the question does not arise");
```

```
puts("\n== counting set bits ==");
uint32_t samples[] = { 0x00000000u, 0x00000001u, 0x0000FFFFu, 0xFFFFFFFFu };
for (size_t i = 0; i < 4; i++) {
    printf("  0x%08X : naive %2d, kernighan %2d\n", samples[i],
           popcount_naive(samples[i]),
           popcount_kernighan(samples[i]));
}

clock_t t0 = clock();
long total = 0;
for (uint32_t i = 0; i < 3000000u; i++) total += popcount_naive(i);
double naive_time = (double)(clock() - t0) / CLOCKS_PER_SEC;

t0 = clock();
total = 0;
for (uint32_t i = 0; i < 3000000u; i++) total += popcount_kernighan(i);
double kern_time = (double)(clock() - t0) / CLOCKS_PER_SEC;

printf("  3M values: naive %.3fs, Kernighan %.3fs\n",
       naive_time, kern_time);
puts("  v &= v-1 clears the lowest set bit, so the loop runs once per 1");

puts("\n== a checksum ==");
const char *text = "the quick brown fox";
printf("  checksum(\"%s\") = 0x%08X\n",
       text, checksum((const uint8_t *)text, 19));
printf("  checksum(\"the quick brown fpx\") = 0x%08X  <-- one byte changed\n",
       checksum((const uint8_t *)"the quick brown fpx", 19));

puts("\n== bit-fields: convenient, but not portable ==");
struct Packed p = { .read = 1, .write = 0, .level = 5 };
printf("  read=%u write=%u level=%u, sizeof = %zu\n",
       p.read, p.write, p.level, sizeof p);
puts("  the standard does not fix the order or the padding,");
```

```
    puts("  so never use a bit-field struct to describe a file or packet format");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o bits bits.c
./bits
```

The warning that proves the trap

```
bits.c:118:32: warning: suggest parentheses around comparison
              in operand of '&' [-Wparentheses]
```

The output shows `f & PERM_EXEC == 0` evaluating to 0 while `(f & PERM_EXEC) == 0` gives 1. Same tokens, opposite meanings, one warning between you and a silent bug.

Provoke the undefined shift

```
uint32_t x = 1;
printf("%u\n", x << 32);     /* undefined behavior */
```

```
warning: left shift count >= width of type [-Wshift-count-overflow]
```

Run it and you will probably see 1, not 0 — x86 uses only the low five bits of the shift count, so a shift by 32 is a shift by 0. Build with `-fsanitize=undefined` and it is reported explicitly:

```
runtime error: shift exponent 32 is too large for 32-bit type
```

Why Kernighan's method is faster

`v & (v - 1)` clears the lowest set bit: subtracting 1 flips that bit to 0 and every bit below it to 1, so the AND keeps only the bits above. The loop therefore runs once per set bit rather than once per bit position. For sparse values the difference is large; for `0xFFFFFFFF` the two are identical. The timing in the output shows it on a realistic mix.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`flags & MASK == 0`	Parses as `flags & (MASK == 0)`	Parenthesize the bitwise operation.
Using `&&` where `&` was meant	Compiles; yields 0 or 1 instead of a mask	Logical for conditions, bitwise for bits.
Shifting by the type's width or more	Undefined; often a no-op on x86	Guard the shift count.
Bit manipulation on signed types	Implementation-defined right shift; undefined left shift of negatives	Use `unsigned` or `uint32_t`.
Forgetting the mask after a shift	Neighbouring fields leak into the result	`(v >> n) & mask`.
`x &= ~MASK` with `MASK` narrower than `x`	Zero extension clears `x`'s high bits	Write the mask in `x`'s type: `~(uint64_t)MASK`.
Bit-fields for a file or packet layout	Different layout on another compiler	Explicit shifts and masks.
Plain `int` bit-fields	Signedness is implementation-defined	Always write `unsigned`.

7 Check yourself

Why is `x &= ~mask` the way to clear bits?

`~mask` has 0 exactly where `mask` had 1, and 1 everywhere else. ANDing with 0 forces a bit to 0; ANDing with 1 leaves it unchanged. So the operation clears precisely the selected bits and preserves all the others — which is the requirement.

What does `flags & FLAG == 0` actually compute?

`flags & (FLAG == 0)`, because `==` binds more tightly than `&` in C. `FLAG == 0` is 0 for any non-zero flag, so the whole expression is `flags & 0`, which is always 0 and therefore always false. Parenthesize the bitwise operation; `-Wparentheses` warns about it.

Why should bit manipulation be done on unsigned types?

Because signedness introduces three problems that have no meaning when you are treating a value as a row of bits: right shift of a negative value is implementation-defined, left shift of a negative value is undefined, and `~` on a signed value can produce a negative result that sign-extends unexpectedly. Unsigned types have none of these.

Why is a bit-field struct unsuitable for describing a network packet?

Because the standard does not fix the allocation order of fields within a unit, whether a field may straddle units, or the alignment and padding of the whole structure. Two compilers can lay out the same declaration differently, so the bytes your program writes need not match what the protocol requires. Use explicit shifts and masks for anything that crosses a boundary.

Why does `v &= v - 1` clear the lowest set bit?

Subtracting 1 turns the lowest set bit into 0 and every bit below it into 1, while leaving the higher bits untouched. ANDing the original with that result keeps only the higher bits — so exactly one set bit disappears per iteration. A counting loop built on it runs once per set bit instead of once per bit position.

8 Where this leads

Chapter 26 turns to files and streams — opening, reading, writing, positioning, and the buffering that explains why output sometimes appears late. The binary record format built there is the first place this chapter's packing matters outside your own process, and chapter 37 then makes such a format portable across machines with different byte order.

INTERMEDIATE · CHAPTER 26

Files and Streams

Every file operation in C goes through a `FILE *` — a buffered stream. Understanding the buffer explains delayed output, lost data on a crash, and why checking `feof` in a loop condition is wrong. Understanding the error state explains how to know whether your program actually worked.

BY THE END OF THIS CHAPTER YOU CAN

- Open, read, write, and close files with every error path handled.
- Read a text file line by line without overflowing or truncating.
- Write and read fixed-size binary records and seek directly to any one.
- Distinguish end of file from an error, and explain why `while (!feof(f))` is wrong.
- Explain buffering and say when you must flush.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Opening and closing

```
FILE *f = fopen("data.txt", "r");
if (f == NULL) {
    perror("data.txt");              /* prints the reason */
    return EXIT_FAILURE;
}
…
if (fclose(f) != 0) {
    perror("closing data.txt");      /* a write may fail here */
}
```

MODE	MEANING	IF THE FILE EXISTS
`"r"`	read	Fails if it does not exist
`"w"`	write	**Truncates to zero length**
`"a"`	append	Writes always go to the end
`"r+"`	read and write	Must exist; does not truncate
`"w+"`	read and write	Truncates
`"a+"`	read and append	Reads anywhere, writes at the end

Add `b` for binary: `"rb"`, `"wb"`. On Unix it changes nothing; on Windows, text mode translates `\n` to `\r\n` on write and back on read, which silently corrupts binary data. Always use `b` for non-text, for portability.

Two things people skip. `perror` prints your message followed by the system's explanation of `errno` — "No such file or directory" — which turns a useless error into a useful one. And `fclose` can fail: it flushes buffered data, and that write can hit a full disk. Ignoring its return value on a file you wrote means you may report success after losing data.

2 Reading text

The pattern from chapter 23, now applied to a file:

```
char line[256];
while (fgets(line, sizeof line, f) != NULL) {
    line[strcspn(line, "\n")] = '\0';
    process(line);
}
```

`fgets` reads at most `size - 1` characters, stops at a newline, and always terminates. It returns `NULL` at end of input or on error — which the loop treats identically, so you must distinguish them afterwards.

> `while (!feof(f))` **is wrong.** `feof` reports whether a *previous* read already hit the end; it cannot predict. So the loop attempts one read too many, and the last iteration processes stale or uninitialized data. Test the read's return value instead, and consult `feof` and `ferror` only after the loop, to find out which ended it.

```
while (fgets(line, sizeof line, f) != NULL) { … }


if (ferror(f)) {
    perror("read error");            /* a real failure */
} else {
    /* feof(f) is true: normal end of input */
}
```

3 Writing

```
fprintf(f, "%s,%d\n", name, score);     /* formatted, like printf */
fputs("a line\n", f);                   /* no formatting, no newline added */
fputc('\n', f);                         /* one character */
```

`fprintf` returns the number of characters written, or negative on error. For a file that matters — a report, an export — check it, or at minimum check `ferror` before closing. A full disk or a broken pipe produces no exception; the call simply fails and your program continues.

Note that `fputs` does not append a newline while `puts` does. That asymmetry catches everyone once.

4 Binary records and positioning

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *f);
size_t fread (      void *ptr, size_t size, size_t count, FILE *f);
```

Both return the number of *items* transferred, not bytes. A short return means end of file or error.

```
typedef struct { int id; char name[32]; double score; } Record;


Record r = { .id = 1, .score = 95.5 };
if (fwrite(&r, sizeof r, 1, f) != 1) {
    perror("write failed");
}
```

Because every record is the same size, the *n*th record starts at offset `n * sizeof(Record)` — so you can jump straight to it without reading what precedes it:

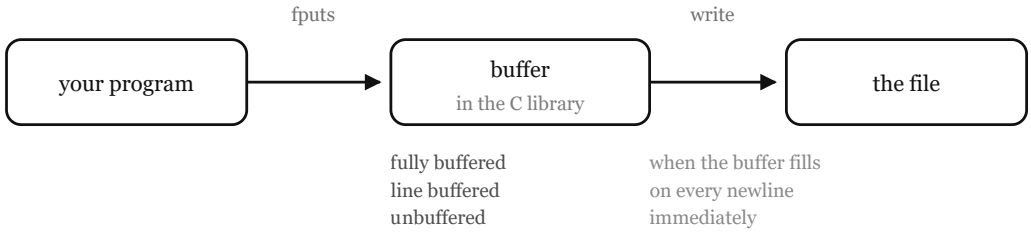
```
fseek(f, (long)(index * sizeof(Record)), SEEK_SET);
fread(&r, sizeof r, 1, f);
```

CALL	DOES
`fseek(f, off, SEEK_SET)`	Move to `off` from the start
`fseek(f, off, SEEK_CUR)`	Move `off` from the current position
`fseek(f, 0, SEEK_END)`	Move to the end — used to find the size
`ftell(f)`	Current position, or −1 on error
`rewind(f)`	Back to the start and clear the error flags

This file format is not portable. It embeds your machine's integer size, byte order, and structure padding — the padding that chapter 22 measured. Reading it on another architecture gives nonsense. That is acceptable for a cache or a temporary index and unacceptable for anything exchanged between machines; chapter 37 builds the portable alternative.

5 Buffering

Writes do not reach the file immediately. The C library collects them in a buffer and writes in blocks, because one large write is far cheaper than a thousand small ones.



Nothing reaches the file until the buffer is flushed — which is why a crash loses output and why a prompt without a newline may not appear.

MODE	FLUSHED WHEN	DEFAULT FOR
Fully buffered	The buffer fills	Files
Line buffered	A newline is written	`stdout` to a terminal
Unbuffered	Immediately	`stderr`

Three consequences you will actually meet.

A prompt without a newline may not appear before the program blocks for input — chapter 8's `fflush(stdout)`.

Output can be lost on a crash. Buffered data that was never flushed is gone. This is why debugging with `printf` can mislead: the last message before a segmentation fault may never have been written.

`stdout` **changes behavior when redirected.** To a terminal it is line buffered; to a file or a pipe it becomes fully buffered. A program whose output interleaves correctly on screen can produce a different order in a file, because `stderr` is never buffered while `stdout` now is.

```
fflush(f);                          /* flush one stream */
fflush(NULL);                       /* flush every output stream */
setvbuf(f, NULL, _IONBF, 0);        /* switch buffering off */
```

6 Worked example: text in, binary out, seek back

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>

#define NAME_LEN 32
#define LINE_LEN 256

typedef struct {
    int    id;
    char   name[NAME_LEN];
    double score;
} Record;
```

```
static const char *TEXT_FILE   = "records.csv";
static const char *BINARY_FILE = "records.dat";

/* ---------- create some input ---------- */

static bool write_sample_text(void)
{
    FILE *f = fopen(TEXT_FILE, "w");
    if (f == NULL) {
        perror(TEXT_FILE);
        return false;
    }

    fputs("# id,name,score\n", f);
    fprintf(f, "1,Ada,95.5\n");
    fprintf(f, "2,Dennis,88.0\n");
    fprintf(f, "3,Ken,91.25\n");
    fprintf(f, "4,Grace,78.5\n");
    fputs("bad line with no commas\n", f);
    fprintf(f, "5,Linus,not-a-number\n");

    if (ferror(f)) {
        perror(TEXT_FILE);
        fclose(f);
        return false;
    }
    if (fclose(f) != 0) {            /* the flush can fail here */
        perror(TEXT_FILE);
        return false;
    }
    return true;
}

/* ---------- read text, one line at a time ---------- */

static bool parse_line(const char *line, Record *out)
{
    char name[NAME_LEN];
    int  id;
```

```
    double score;

    /* %31[^,] reads up to 31 characters that are not a comma */
    if (sscanf(line, "%d,%31[^,],%lf", &id, name, &score) != 3) {
        return false;
    }
    out->id = id;
    snprintf(out->name, sizeof out->name, "%s", name);
    out->score = score;
    return true;
}

static size_t read_text(Record *records, size_t capacity)
{
    FILE *f = fopen(TEXT_FILE, "r");
    if (f == NULL) {
        perror(TEXT_FILE);
        return 0;
    }

    char line[LINE_LEN];
    size_t count = 0;
    long line_number = 0;

    while (count < capacity && fgets(line, sizeof line, f) != NULL) {
        line_number++;

        if (strchr(line, '\n') == NULL && !feof(f)) {
            fprintf(stderr, "  line %ld too long, skipping\n", line_number);
            int c;
            while ((c = fgetc(f)) != '\n' && c != EOF) { }
            continue;
        }
        line[strcspn(line, "\n")] = '\0';

        if (line[0] == '\0' || line[0] == '#') {
            continue;
        }
        if (!parse_line(line, &records[count])) {
```

```
            fprintf(stderr, "  line %ld rejected: \"%s\"\n", line_number, line);
            continue;
        }
        count++;
    }

    /* Only NOW ask which condition ended the loop. */
    if (ferror(f)) {
        perror(TEXT_FILE);
    }
    fclose(f);
    return count;
}

/* ---------- write fixed-size binary records ---------- */

static bool write_binary(const Record *records, size_t count)
{
    FILE *f = fopen(BINARY_FILE, "wb");      /* b matters on Windows */
    if (f == NULL) {
        perror(BINARY_FILE);
        return false;
    }
    size_t written = fwrite(records, sizeof *records, count, f);
    if (written != count) {
        fprintf(stderr, "  wrote only %zu of %zu records\n", written, count);
        fclose(f);
        return false;
    }
    return fclose(f) == 0;
}

/* ---------- seek directly to record n ---------- */

static bool read_record_at(size_t index, Record *out)
{
    FILE *f = fopen(BINARY_FILE, "rb");
    if (f == NULL) {
        perror(BINARY_FILE);
```

```
        return false;
    }

    if (fseek(f, 0, SEEK_END) != 0) {
        fclose(f);
        return false;
    }
    long size = ftell(f);
    long total = size / (long)sizeof(Record);

    if ((long)index >= total) {
        fprintf(stderr, "  index %zu out of range (%ld records)\n",
                index, total);
        fclose(f);
        return false;
    }

    if (fseek(f, (long)(index * sizeof(Record)), SEEK_SET) != 0) {
        fclose(f);
        return false;
    }
    bool ok = fread(out, sizeof *out, 1, f) == 1;
    fclose(f);
    return ok;
}

static void print_record(const Record *r)
{
    printf("  id=%-3d %-10s %6.2f\n", r->id, r->name, r->score);
}

int main(void)
{
    puts("== writing sample text ==");
    if (!write_sample_text()) {
        return EXIT_FAILURE;
    }
    printf("  wrote %s\n", TEXT_FILE);
```

```
    puts("\n== reading it back, rejecting bad lines ==");
    Record records[16];
    size_t count = read_text(records, 16);
    printf("  accepted %zu records:\n", count);
    for (size_t i = 0; i < count; i++) {
        print_record(&records[i]);
    }

    puts("\n== writing fixed-size binary records ==");
    if (!write_binary(records, count)) {
        return EXIT_FAILURE;
    }
    printf("  each record is %zu bytes, %zu records = %zu bytes\n",
           sizeof(Record), count, sizeof(Record) * count);

    puts("\n== seeking directly, without reading what precedes ==");
    Record one;
    for (size_t i = 0; i < count; i += 2) {
        if (read_record_at(i, &one)) {
            printf("  record %zu: ", i);
            print_record(&one);
        }
    }
    read_record_at(99, &one);

    puts("\n== buffering ==");
    printf("  this has no newline and may not appear yet...");
    fflush(stdout);
    puts(" [flushed]");
    fprintf(stderr, "  stderr is unbuffered, so this is never delayed\n");
    puts("  run './files > out.txt 2>&1' and compare the order");

    puts("\n== why while(!feof(f)) is wrong ==");
    FILE *f = fopen(TEXT_FILE, "r");
    if (f != NULL) {
        char line[LINE_LEN];
        int bad = 0;
        while (!feof(f)) {                 /* deliberately wrong */
            if (fgets(line, sizeof line, f) == NULL) break;
```

```
            bad++;
        }
        rewind(f);
        int good = 0;
        while (fgets(line, sizeof line, f) != NULL) {
            good++;
        }
        printf("  feof-driven loop needed a break to stay correct: %d\n", bad);
        printf("  return-value-driven loop: %d lines, no special case\n", good);
        fclose(f);
    }

    remove(TEXT_FILE);
    remove(BINARY_FILE);
    puts("\n  temporary files removed");
    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o files files.c
./files
```

Three details worth extracting

`%31[^,]` in the `sscanf` format. It reads up to 31 characters that are not a comma — a bounded scan set. Without the 31 it is as dangerous as a bare `%s`, and chapter 8's overflow warning applies. The bound is one less than the array size, to leave room for the terminator.

`fclose` is checked. On the write path its return value decides whether the function reports success. A program that ignores it can claim to have written a file that never reached the disk.

Seeking reads one record, not *n*. `read_record_at(4, …)` touches 48 bytes, not the whole file. For a million-record file that is the difference between microseconds and seconds — which is what fixed-size records buy you, and why variable-length text formats need an index.

Experiments

See the non-portability. Print the raw bytes of a record:

```
xxd records.dat | head -4
```

Find the integer `1` stored as `01 00 00 00` — little-endian — and the padding bytes between `name` and `score` holding whatever was in memory. Both are why chapter 37 exists.

Break the buffering assumption.

```
./files            # stdout to a terminal: line buffered
./files > out.txt 2>&1
cat out.txt        # different interleaving of stdout and stderr
```

Redirected, `stdout` becomes fully buffered while `stderr` stays unbuffered, so the error lines now appear before output that was logically earlier.

Lose data on a crash. Add `printf("about to crash\n"); *(int *)0 = 1;` at the end and redirect to a file. The message is in the buffer when the process dies, so that line is missing from the file — and the same program prints it when run on a terminal, because the newline flushed a line-buffered stream.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Not checking `fopen`	Null dereference on the first read	Check, and use `perror` to say why.
`while (!feof(f))`	One iteration too many; stale data	Test the read's return value.
Ignoring `fclose`'s result	Reports success after a failed flush	Check it on any file you wrote.
Opening with `"w"` to read	Truncates the file to nothing	`"r"`, or `"r+"` to modify.
Omitting `b` for binary on Windows	Newline translation corrupts data	Always `"rb"` / `"wb"`.
Assuming `fread` returns bytes	Off by a factor of the item size	It returns items.
Writing structs and reading them elsewhere	Byte order and padding differ	Serialize field by field — chapter 37.
Expecting output before a newline	Prompt invisible; program looks frozen	`fflush(stdout)`.

8 Check yourself

Why is `while (!feof(f))` the wrong loop condition?

Because `feof` reports whether a previous read already reached the end; it cannot look ahead. The condition is therefore still true after the last successful read, so the loop runs once more, the read fails, and the body processes whatever was left in the buffer. Drive the loop on the return value of the read itself, and use `feof` / `ferror` afterwards to find out which ended it.

Why can `fclose` fail, and why does that matter?

Because it flushes any buffered data, and that write can fail — a full disk, a broken pipe, a network filesystem error. Data you thought was written is then lost. Ignoring the return value on a file you wrote means your program can report success after losing the user's data.

Why is a file of raw structures not portable?

It embeds the writing machine's integer widths, byte order, and the compiler's structure padding — none of which the standard fixes. Reading it on a different architecture or with a different compiler yields wrong values. Such a format is fine for a private cache and unsuitable for anything exchanged between machines.

Your program's output appears in a different order when redirected to a file. Why?

Because `stdout` is line buffered when it goes to a terminal but fully buffered when it goes to a file or pipe, while `stderr` is always unbuffered. Redirected, the error messages are written immediately while ordinary output waits in the buffer, so the interleaving changes. Flush explicitly if the order matters.

What do fixed-size binary records buy you over a text format?

Direct access. Record n starts at offset `n × sizeof(Record)`, so `fseek` can jump straight to it without reading anything before it — constant time instead of linear. A variable-length text format has no such property and needs a separate index to match it.

9 Where this leads

Every function in this chapter's example returns a status that the caller had to check, and each one cleaned up before returning. Chapter 27 makes that discipline systematic: one error contract per module, one cleanup path per function, and the `goto` idiom that lets a function acquire five resources and release exactly the ones it got.

INTERMEDIATE · CHAPTER 27

Error Handling Strategies

C has no exceptions. Every failure travels back to the caller as a return value, and every caller must look. What separates robust C from fragile C is not cleverness but consistency: one convention, applied everywhere, with exactly one cleanup path per function.

BY THE END OF THIS CHAPTER YOU CAN

- Name the three kinds of error and say which technique addresses each.
- Use `errno`, `perror`, and `strerror` correctly, including when to save `errno`.
- Decide whether a given check belongs in an `assert` or in the shipped code.
- Write a function that acquires several resources and releases exactly the ones it got.
- Define an error contract that a whole module follows.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Three kinds of error

KIND	EXAMPLE	HANDLED BY
Compile time	Syntax error, type mismatch	Fixing the code
Link time	`undefined reference`	Linking the right library — chapters 2, 29
Runtime, programmer error	Null pointer passed where forbidden, index out of range	`assert` — it is a bug
Runtime, environmental	File missing, disk full, allocation failed	Return a status; the caller decides

The last two are often confused, and the distinction drives every decision in this session. A file that does not exist is not a bug — it is a legitimate state of the world your program must handle. A null pointer passed to a function that documented it must not be null *is* a bug, and no amount of runtime handling will make the program correct.

Environmental failures get error handling. Programmer errors get assertions.

2 Checking return values

Every standard library function that can fail says so. The convention varies, which is why a table is worth keeping in view:

FUNCTION	FAILURE LOOKS LIKE
`malloc`, `calloc`, `realloc`	returns `NULL`
`fopen`	returns `NULL`, sets `errno`
`fclose`, `fflush`	returns `EOF`
`fgets`	returns `NULL` — check `ferror` to distinguish from end of file
`fread`, `fwrite`	returns fewer items than requested
`printf`, `fprintf`	returns a negative value
`strtol`, `strtod`	via `endptr` and `errno` — chapter 23
`remove`, `rename`	returns non-zero, sets `errno`

There is no way to be told you forgot. GCC's `__attribute__((warn_unused_result))` and C23's `[[nodiscard]]` let you mark your own functions so callers are warned:

```
[[nodiscard]] bool buffer_append(Buffer *b, const char *text);   /* C23 */
```

Use it on anything whose result must not be ignored. It is the only mechanism C offers for enforcing that a caller checks.

3 `errno`

`errno` is a modifiable lvalue holding the last error code set by a library function. Three rules govern it.

It is only meaningful after a function has reported failure. Library functions may set it on success too, so testing it without first seeing a failure indication is meaningless.

Set it to 0 before calls that need it. Required for `strtol` and the mathematics functions, which report range errors only through `errno`.

Save it immediately. Any intervening library call can overwrite it — including the `fprintf` you are using to report the problem:

```
if (fopen(path, "r") == NULL) {
    int saved = errno;                          /* capture first */
    fprintf(stderr, "opening %s: %s\n", path, strerror(saved));
    return saved;
}
```

Reporting it

```
perror("data.txt");
/* prints: data.txt: No such file or directory */

fprintf(stderr, "%s: %s\n", path, strerror(errno));
/* the same text, under your control */
```

`perror` is convenient and always writes to `stderr`. `strerror` returns the message so you can place it in a larger sentence, log it, or send it elsewhere — at the cost of returning a pointer to a static buffer, which the next call overwrites.

4 Assertions

```
#include <assert.h>

void buffer_append(Buffer *b, const char *text)
{
    assert(b != NULL);           /* the caller's obligation */
    assert(text != NULL);
    …
}
```

If the condition is false, `assert` prints the expression, the file, and the line, and calls `abort`. Defining `NDEBUG` removes every assertion:

```
gcc -DNDEBUG -O2 …          # assertions compiled out
```

> **Never put a side effect in an assertion.** `assert(buffer_append(b, "x"));` works in a debug build and silently does nothing in a release build, because the whole expression disappears. This produces a program that works while you are testing it and fails when you ship it — the worst possible failure mode.

USE ASSERT FOR	USE A REAL CHECK FOR
Preconditions your own code must satisfy	Anything derived from user input, files, or the network
Invariants that should be impossible to break	Allocation results and system call results
Documenting an assumption executably	Conditions the caller is allowed to hit

`static_assert`

```
static_assert(sizeof(int) >= 4, "this code assumes at least 32-bit int");
static_assert(sizeof(Record) == 48, "Record layout changed unexpectedly");
```

Checked at compile time, costs nothing at runtime, and cannot be disabled. Use it for assumptions about sizes, alignments, and enumeration counts. It is spelled `_Static_assert` before C23 unless you include `<assert.h>`.

5 The goto cleanup idiom

A function that acquires several resources must release exactly the ones it managed to acquire, on every exit path. Written with early returns, the releases multiply:

```
/* The version everyone writes first */
bool process(const char *in_path, const char *out_path)
{
    FILE *in = fopen(in_path, "r");
    if (in == NULL) return false;

    FILE *out = fopen(out_path, "w");
    if (out == NULL) { fclose(in); return false; }

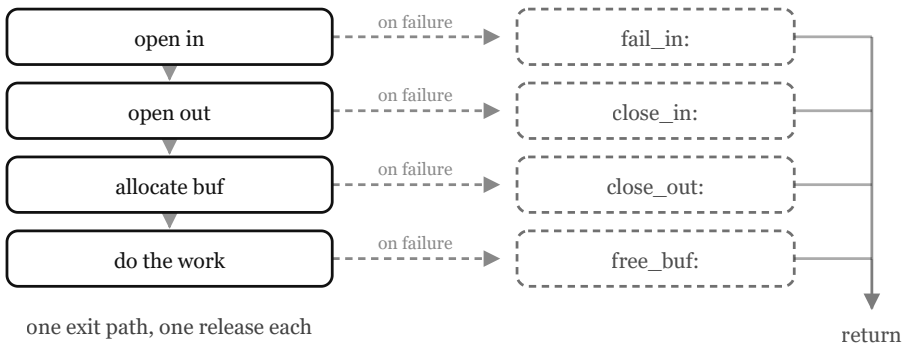
    char *buffer = malloc(BUFSIZE);
    if (buffer == NULL) { fclose(out); fclose(in); return false; }

    if (!do_work(in, out, buffer)) {
```

```
        free(buffer); fclose(out); fclose(in); return false;
    }

    free(buffer); fclose(out); fclose(in);
    return true;
}
```

`fclose(in)` appears four times. Add a fourth resource and it appears five. Every duplicate is a place to forget one — and the error paths are exactly the ones your tests do not cover.



Each acquisition jumps to the label that undoes exactly what had succeeded by then, and every path leaves through the same door.

The idiom that fixes it — and that the Linux kernel, SQLite, and most serious C libraries use throughout:

```
bool process(const char *in_path, const char *out_path)
{
    bool   ok     = false;
    FILE  *in     = NULL;
    FILE  *out    = NULL;
    char  *buffer = NULL;

    in = fopen(in_path, "r");
    if (in == NULL) {
        perror(in_path);
        goto cleanup;
    }
```

```
    out = fopen(out_path, "w");
    if (out == NULL) {
        perror(out_path);
        goto cleanup;
    }

    buffer = malloc(BUFSIZE);
    if (buffer == NULL) {
        goto cleanup;
    }

    if (!do_work(in, out, buffer)) {
        goto cleanup;
    }

    ok = true;

cleanup:
    free(buffer);                           /* free(NULL) is safe   */
    if (out != NULL && fclose(out) != 0) { /* the flush can fail   */
        ok = false;
    }
    if (in != NULL) {
        fclose(in);
    }
    return ok;
}
```

What makes it work: every resource variable is initialized to a null value at the top, so the single cleanup block can unconditionally attempt to release all of them. `free(NULL)` is defined to do nothing, and the `NULL` guards handle the file pointers. There is exactly one exit, and adding a fourth resource means adding one acquisition and one release.

For several resources with distinct cleanup, use cascading labels in reverse order of acquisition:

```
    …
    goto cleanup_all;

cleanup_all:
    free(buffer);
cleanup_out:
    fclose(out);
cleanup_in:
    fclose(in);
    return ok;
```

Each failure jumps to the label that releases only what has been acquired so far. This is the one place where chapter 10's warning about `goto` does not apply — it is the standard C solution to a problem the language gives no other tool for.

6 An error contract for a module

Individual checks are not enough. A module needs one stated convention, so callers do not have to learn a new one per function.

```
/* csv.h — error contract
 *
 * Every function returns a CsvStatus. CSV_OK means success and
 * output parameters have been written. Any other value means failure,
 * and output parameters are untouched.
 *
 * On CSV_ERRNO the caller may inspect errno for details.
 * No function frees anything the caller passed in.
 * csv_last_message() returns a human-readable description of the
 * most recent failure on this thread.
 */

typedef enum {
    CSV_OK = 0,
    CSV_ERRNO,          /* a system call failed; see errno */
    CSV_BAD_FORMAT,     /* the input was malformed         */
    CSV_TOO_LONG,       /* a line exceeded the limit       */
```

```
    CSV_NO_MEMORY
} CsvStatus;

const char *csv_status_name(CsvStatus s);
```

Four decisions worth copying:

- **A named enumeration, not bare integers.** Chapter 22's `switch` warning then tells every caller when you add a status.
- **Success is zero**, matching the rest of C and letting `if (status != CSV_OK)` read naturally.
- **Output parameters are untouched on failure**, so a caller cannot accidentally use a half-written result.
- **Ownership is stated once**, for the whole module, rather than per function.

Write this comment block before the functions, not after. It is the part of the interface that the signatures cannot express.

7 Worked example: one contract, one cleanup path

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <assert.h>
#include <stdbool.h>

#define LINE_MAX_LEN 256

/* ---------- the contract ---------- */

typedef enum {
    FC_OK = 0,
    FC_ERRNO,
    FC_BAD_FORMAT,
    FC_TOO_LONG,
    FC_NO_MEMORY
} FcStatus;
```

```
static const char *fc_status_name(FcStatus s)
{
    switch (s) {                        /* no default: see chapter 22 */
    case FC_OK:         return "ok";
    case FC_ERRNO:      return "system error";
    case FC_BAD_FORMAT: return "malformed input";
    case FC_TOO_LONG:   return "line too long";
    case FC_NO_MEMORY:  return "out of memory";
    }
    return "unknown";
}

/* ---------- the function with three resources ---------- */

/* Copies in_path to out_path, doubling every number it finds.
   Returns FC_OK on success. On failure, out_path may exist but
   its contents are unspecified; nothing the caller owns is freed. */
static FcStatus double_numbers(const char *in_path, const char *out_path,
                               long *lines_out)
{
    assert(in_path  != NULL);          /* programmer error if violated */
    assert(out_path != NULL);
    assert(lines_out != NULL);

    FcStatus  status = FC_ERRNO;       /* pessimistic default */
    FILE     *in     = NULL;
    FILE     *out    = NULL;
    char     *buffer = NULL;
    long      lines  = 0;

    in = fopen(in_path, "r");
    if (in == NULL) {
        goto cleanup;
    }

    out = fopen(out_path, "w");
    if (out == NULL) {
        goto cleanup;
    }
```

```
    buffer = malloc(LINE_MAX_LEN);
    if (buffer == NULL) {
        status = FC_NO_MEMORY;
        goto cleanup;
    }

    while (fgets(buffer, LINE_MAX_LEN, in) != NULL) {
        lines++;

        if (strchr(buffer, '\n') == NULL && !feof(in)) {
            status = FC_TOO_LONG;
            goto cleanup;
        }
        buffer[strcspn(buffer, "\n")] = '\0';

        if (buffer[0] == '\0') {
            continue;
        }

        errno = 0;
        char *end;
        long value = strtol(buffer, &end, 10);
        if (end == buffer || *end != '\0' || errno == ERANGE) {
            status = FC_BAD_FORMAT;
            goto cleanup;
        }

        if (fprintf(out, "%ld\n", value * 2) < 0) {
            status = FC_ERRNO;
            goto cleanup;
        }
    }

    if (ferror(in)) {
        status = FC_ERRNO;
        goto cleanup;
    }
```

```
    *lines_out = lines;                /* written only on success */
    status = FC_OK;

cleanup:
    free(buffer);                      /* free(NULL) is safe */
    if (out != NULL && fclose(out) != 0 && status == FC_OK) {
        status = FC_ERRNO;             /* the flush failed */
    }
    if (in != NULL) {
        fclose(in);
    }
    return status;
}

/* ---------- helper to build test inputs ---------- */

static bool make_file(const char *path, const char *contents)
{
    FILE *f = fopen(path, "w");
    if (f == NULL) {
        return false;
    }
    fputs(contents, f);
    return fclose(f) == 0;
}

static void try_case(const char *label, const char *contents)
{
    const char *in  = "fc_in.txt";
    const char *out = "fc_out.txt";

    if (contents != NULL && !make_file(in, contents)) {
        perror("creating input");
        return;
    }

    long lines = -1;
    FcStatus s = double_numbers(contents == NULL ? "does_not_exist.txt" : in,
                                out, &lines);
```

```
    printf("  %-22s -> %-16s", label, fc_status_name(s));
    if (s == FC_OK) {
        printf("(%ld lines)\n", lines);
    } else if (s == FC_ERRNO) {
        printf("(%s)\n", strerror(errno));
    } else {
        printf("(lines_out untouched: %ld)\n", lines);
    }

    remove(in);
    remove(out);
}

int main(void)
{
    puts("== one contract, every failure path exercised ==");
    try_case("normal input",     "1\n2\n3\n");
    try_case("blank lines",      "1\n\n2\n");
    try_case("not a number",     "1\nabc\n");
    try_case("trailing rubbish", "1\n12xyz\n");
    try_case("out of range",     "99999999999999999999\n");
    try_case("missing file",     NULL);

    puts("\n== errno must be saved before reporting ==");
    errno = 0;
    FILE *f = fopen("definitely_not_here.txt", "r");
    if (f == NULL) {
        int saved = errno;
        fprintf(stderr, "  captured errno = %d\n", saved);
        fprintf(stderr, "  message: %s\n", strerror(saved));
        printf("  errno after the fprintf calls is now %d\n", errno);
        puts("  which is why the value is captured immediately");
    }

    puts("\n== assert is for programmer error, not user error ==");
    puts("  double_numbers asserts its pointers are non-null:");
    puts("  that is the caller's contract, not a runtime condition.");
    puts("  A missing file is handled; a null path is a bug.");
```

```
    puts("\n== static_assert: checked at compile time ==");
    static_assert(sizeof(long) >= 4, "long must be at least 32 bits");
    puts("  sizeof(long) >= 4 verified before the program ran");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o errors errors.c
./errors
```

What the design buys

Every failure path is exercised and none of them leaks. Run it under the leak checker and there is no output — including on the paths that failed after opening two files and allocating a buffer. That is the point of the single cleanup block.

`lines_out` **is untouched on failure.** The output shows `-1` for every failing case except the missing file (which prints the `errno` message instead), because the assignment happens only on the success path. A caller that ignores the status still cannot mistake stale data for a result.

Adding a resource is one line in two places. Try it: add a second buffer, initialize it to `NULL`, allocate it after the first, and add one `free` to the cleanup block. Compare that with the early-return version, where you would edit five places.

Two experiments

Break the assertion. Call `double_numbers(NULL, "out.txt", &lines)`:

```
errors: errors.c:44: double_numbers: Assertion `in_path != NULL' failed.
Aborted (core dumped)
```

Now rebuild with `-DNDEBUG` and the same call segfaults inside `fopen` instead. The assertion did not make the program correct; it made the bug obvious at the boundary where it was introduced.

Remove the `goto` **and write it with early returns.** Then delete one `fclose(in)` from one branch — a single-character-scale mistake — and run under the leak checker with an input that triggers that branch. The leak is reported; the version with one cleanup path had no branch in which to make the mistake.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Ignoring a return value	Program continues on bad state	Check every call that can fail; mark yours `[[nodiscard]]`.
Reading `errno` after other calls	Overwritten by the reporting code	Save it immediately.
Testing `errno` without a failure indication	Meaningless — it may be set on success	Check the return value first.
Side effects inside `assert`	Disappears under `NDEBUG`	Never put work in an assertion.
`assert` on user input	Release build accepts bad input silently	Real checks for anything external.
Cleanup duplicated per early return	One branch forgets a release	Single cleanup label.
Uninitialized resource variables	Cleanup frees garbage	Initialize all to `NULL` at the top.
Writing output parameters before validating	Caller sees half-written results	Write only on the success path.

9 Check yourself

When is `assert` the right tool and when is it the wrong one?

Right for conditions that should be impossible if your code is correct — preconditions your own callers must satisfy, invariants within a module. Wrong for anything the program can legitimately encounter: a missing file, a malformed input line, a failed allocation. Assertions vanish under `NDEBUG`, so relying on one to validate external input means the release build silently accepts it.

Why must `errno` be saved immediately after a failure?

Because any subsequent library call may overwrite it — including the `fprintf` you use to report the problem. By the time you read it, the value can describe an unrelated event. Copy it into a local variable in the same breath as detecting the failure.

Why does the `goto cleanup` idiom initialize every resource to `NULL` first?

So the single cleanup block can release all of them unconditionally, regardless of how far the function got. `free(NULL)` is defined to do nothing and a `NULL` check guards the rest, so one block correctly handles every failure point. Without the initialization, cleanup would free indeterminate values.

What is wrong with duplicating cleanup code in each early return?

Each duplicate is an independent opportunity to forget one release, and error paths are precisely the paths least likely to be tested. Adding a fourth resource means editing every branch. One cleanup label makes correctness structural rather than a matter of vigilance.

Why should a module define one error contract rather than per-function conventions?

Because callers otherwise have to learn and remember a different convention for every function, and mixed conventions produce unchecked calls. A single documented rule — what the status values mean, whether output parameters are written on failure, who owns what — makes correct use the obvious use, and a named enumeration lets the compiler warn callers when the set of statuses grows.

10 Where this leads

You can now write a function that is correct on every path. Chapter 28 scales that up to a program: splitting code across files, deciding what a header exposes and what stays hidden, and turning the growable array of chapter 19 into a module with a public interface and a private implementation — the last step before it becomes a library in chapter 29.

INTERMEDIATE · CHAPTER 28

The Preprocessor and Modular Programming

Everything so far has lived in one file. This chapter splits a program across several, which raises the two questions that define C's module system: what does a header expose, and what stays invisible. The growable array of chapter 19 becomes a real module by the end.

BY THE END OF THIS CHAPTER YOU CAN

- Write macros that behave correctly under every argument, and say when to use a function instead.
- Use conditional compilation and include guards.
- Explain internal and external linkage, and fix a multiple-definition error.
- Split a program into header and source files with a deliberate public interface.
- Hide an implementation completely behind an opaque pointer.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The preprocessor is a text tool

Chapter 2 established that the preprocessor runs before the compiler and knows nothing about C. It obeys the `#` directives and hands the result on as text.

```
#define MAX_ITEMS 100          /* object-like macro */
#define SQUARE(x) ((x) * (x))  /* function-like macro */
#include <stdio.h>             /* paste in a file */
#ifdef DEBUG … #endif          /* conditional compilation */
```

Macro pitfalls

Parenthesize everything. Twice: the whole body, and every use of a parameter.

```
#define SQUARE(x) x * x            /* broken */
SQUARE(2 + 3)   ->  2 + 3 * 2 + 3  ->  11

#define SQUARE(x) ((x) * (x))      /* correct */
SQUARE(2 + 3)   ->  ((2 + 3) * (2 + 3))  ->  25
```

Beware double evaluation. A parameter appearing twice in the body is evaluated twice:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

MAX(i++, j)     /* i is incremented once or twice, depending on the comparison */
MAX(f(), g())   /* one of the calls happens twice */
```

There is no general fix in standard C. This is the strongest argument for using a function: a function evaluates each argument exactly once, and any compiler will inline a small one at `-O2`.

Prefer the alternatives. For a constant, `const` or an `enum` — chapter 5. For a small operation, a `static inline` function — chapter 38. Macros remain necessary for conditional compilation, for header guards, for stringification, and for generic code that must work across types (chapter 30).

Stringify and paste

```
#define STRINGIFY(x) #x
#define CONCAT(a, b) a ## b

STRINGIFY(hello)     /* becomes "hello" */
CONCAT(my_, var)     /* becomes my_var  */
```

The common practical use is a debug macro that reports the expression it evaluated:

```
#define TRACE(expr) \
    fprintf(stderr, "%s:%d: %s = %d\n", __FILE__, __LINE__, #expr, (expr))
```

PREDEFINED MACRO	EXPANDS TO
`__FILE__`	Current source file name
`__LINE__`	Current line number
`__func__`	Enclosing function name (C99; an identifier, not a macro)
`__DATE__`, `__TIME__`	Compilation date and time
`__STDC_VERSION__`	`201710L` for C17, `202311L` for C23

2 Conditional compilation

```
#ifdef DEBUG
    fprintf(stderr, "state = %d\n", state);
#endif

#if defined(__linux__)
    …
#elif defined(_WIN32)
    …
#else
#   error "unsupported platform"
#endif
```

Enable from the command line with `-DDEBUG` or `-DDEBUG=1`. This is how a debug build differs from a release build without maintaining two copies of the source.

One caution: heavily conditional code is compiled in only one configuration at a time, so the branches you are not building are never checked by the compiler. A block inside `#ifdef _WIN32` can contain syntax errors that a Linux build never reports. Keep the conditional regions small, and prefer a runtime `if` when the cost allows it — an `if (debug_enabled)` is type-checked in every build.

3 Headers and include guards

Without protection, including a header twice defines its types twice:

```
/* buffer.h */
#ifndef BUFFER_H
#define BUFFER_H

…declarations…

#endif /* BUFFER_H */
```

The first inclusion defines `BUFFER_H`; every later one skips the body. `#pragma once` does the same in one line and is supported by every mainstream compiler, though it is not in the standard. Both are fine; be consistent within a project.

What belongs in a header

HEADER (`.h`)	SOURCE (`.c`)
Function *declarations*	Function *definitions*
`typedef`s and struct definitions callers need	Structs only the implementation uses
`#define`d constants that are part of the interface	Internal constants
`extern` declarations of shared variables	The one definition of each
Documentation of the contract	`static` helper functions

A header should include only what its own declarations require, and a source file should include what it uses. Headers that include other headers "just in case" create build-time coupling that is painful to unpick later.

4 Linkage and the one-definition rule

DECLARATION	LINKAGE	VISIBLE TO
`int counter;` at file scope	External	Every translation unit
`static int counter;` at file scope	Internal	This file only
`extern int counter;`	External — a declaration, not a definition	Refers to a definition elsewhere
`void f(void)`	External	Every translation unit
`static void f(void)`	Internal	This file only
Anything inside a function	None	That block

The classic error, and its fix:

```
/* config.h — WRONG */
int verbose = 0;            /* a DEFINITION in a header */
```

```
/usr/bin/ld: multiple definition of `verbose'
```

Every source file that includes the header gets its own definition, and the linker finds several. The correct split:

```
/* config.h */
extern int verbose;         /* declaration: this exists somewhere */

/* config.c */
int verbose = 0;            /* the single definition */
```

Mark everything `static` that is not part of the interface. It prevents accidental name collisions at link time, lets the compiler optimize more aggressively, and documents intent. A source file whose only non-`static` names are the ones in its header is a well-formed module.

5 Information hiding and opaque pointers

A header that exposes a struct definition lets callers reach inside it — and once they do, you can never change the layout. The opaque pointer prevents that entirely:

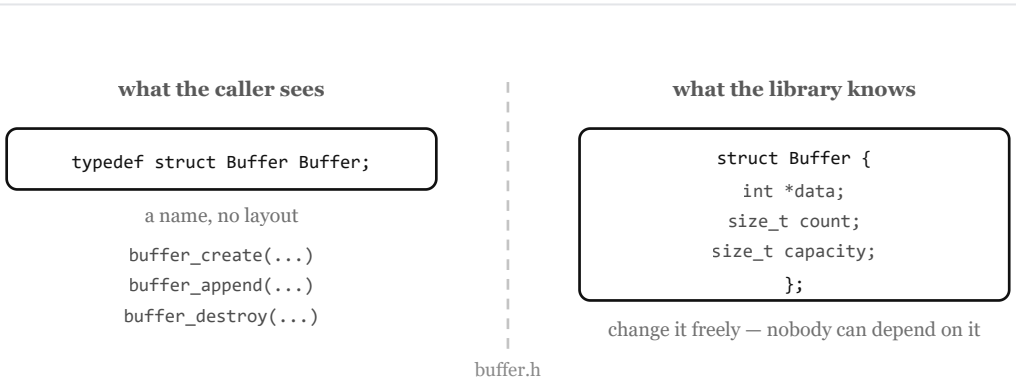
```
/* buffer.h — the caller sees a name, not a layout */
#include <stddef.h>
#include <stdbool.h>

typedef struct Buffer Buffer;      /* incomplete type */

Buffer *buffer_create(size_t capacity);
void    buffer_destroy(Buffer *b);
bool    buffer_append(Buffer *b, int value);
size_t  buffer_count(const Buffer *b);
```

```
/* buffer.c — the definition lives here and nowhere else */
struct Buffer {
    int    *data;
    size_t  count;
    size_t  capacity;
};
```

Callers can hold a `Buffer *` and pass it around, but cannot declare a `Buffer`, cannot see the members, and cannot depend on the size. You may change the implementation freely — add a field, switch to a linked structure — and every caller still compiles without modification.



The header is the boundary. Everything to its right can change without recompiling anything to its left.

EXPOSED STRUCT	OPAQUE POINTER
Caller can allocate on the stack	Must call `create`
Direct member access; fast	Accessor functions
Layout is frozen forever	Free to change
Recompile all callers on any change	Callers unaffected

Use the opaque form for anything with an invariant to protect or a future you cannot predict — which chapter 33 turns into the standard way to build an abstract data type, and chapter 45 shows is essential for a library whose ABI must stay stable.

6 Worked example: the growable array, as a module

Three files. This is the structure chapter 29 compiles into a library.

`intarray.h` — the public interface

```
/* intarray.h — a growable array of int.
 *
 * Error contract:
 *   Functions returning bool give true on success, false on failure;
 *   on failure the array is left unchanged and usable.
 *
 * Ownership:
 *   intarray_create returns a handle the caller must pass to
 *   intarray_destroy exactly once. No other function frees anything.
 */
#ifndef INTARRAY_H
#define INTARRAY_H

#include <stddef.h>
#include <stdbool.h>

typedef struct IntArray IntArray;     /* opaque: no layout exposed */

IntArray *intarray_create(size_t initial_capacity);
void      intarray_destroy(IntArray *a);

bool      intarray_push(IntArray *a, int value);
```

```
bool      intarray_get(const IntArray *a, size_t index, int *out);
size_t    intarray_count(const IntArray *a);
size_t    intarray_capacity(const IntArray *a);
void      intarray_clear(IntArray *a);

#endif /* INTARRAY_H */
```

`intarray.c` — the implementation

```
#include "intarray.h"

#include <stdlib.h>
#include <assert.h>

#define DEFAULT_CAPACITY 4          /* internal: not in the header */

struct IntArray {                   /* the definition lives only here */
    int    *data;
    size_t  count;
    size_t  capacity;
};

/* static: invisible outside this file, no link-time collisions */
static bool grow(IntArray *a)
{
    size_t bigger = a->capacity * 2;
    int *moved = realloc(a->data, bigger * sizeof *moved);
    if (moved == NULL) {
        return false;               /* a->data still valid */
    }
    a->data = moved;
    a->capacity = bigger;
    return true;
}

IntArray *intarray_create(size_t initial_capacity)
{
    if (initial_capacity == 0) {
```

```
        initial_capacity = DEFAULT_CAPACITY;
    }

    IntArray *a = malloc(sizeof *a);
    if (a == NULL) {
        return NULL;
    }
    a->data = malloc(initial_capacity * sizeof *a->data);
    if (a->data == NULL) {
        free(a);                    /* release what we got */
        return NULL;
    }
    a->count = 0;
    a->capacity = initial_capacity;
    return a;
}

void intarray_destroy(IntArray *a)
{
    if (a == NULL) {
        return;                     /* destroy(NULL) is safe, like free */
    }
    free(a->data);
    free(a);
}

bool intarray_push(IntArray *a, int value)
{
    assert(a != NULL);
    if (a->count == a->capacity && !grow(a)) {
        return false;
    }
    a->data[a->count++] = value;
    return true;
}

bool intarray_get(const IntArray *a, size_t index, int *out)
{
    assert(a != NULL && out != NULL);
```

```
    if (index >= a->count) {
        return false;
    }
    *out = a->data[index];
    return true;
}

size_t intarray_count(const IntArray *a)    { assert(a); return a->count; }
size_t intarray_capacity(const IntArray *a) { assert(a); return a->capacity; }
void   intarray_clear(IntArray *a)          { assert(a); a->count = 0; }
```

`main.c` — a caller that cannot see inside

```
#include "intarray.h"

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

/* A debug macro that reports the expression it evaluated. */
#define TRACE(expr) \
    fprintf(stderr, "%s:%d: %s = %zu\n", __FILE__, __LINE__, #expr, (size_t)(expr))

int main(void)
{
    printf("compiled against C standard %ld\n", __STDC_VERSION__);

    IntArray *a = intarray_create(2);
    if (a == NULL) {
        fprintf(stderr, "out of memory\n");
        return EXIT_FAILURE;
    }

    for (int i = 1; i <= 9; i++) {
        if (!intarray_push(a, i * i)) {
            fprintf(stderr, "push failed\n");
            intarray_destroy(a);
            return EXIT_FAILURE;
        }
```

```
    }

    printf("count = %zu, capacity = %zu\n",
           intarray_count(a), intarray_capacity(a));

    printf("contents:");
    for (size_t i = 0; i < intarray_count(a); i++) {
        int value;
        if (intarray_get(a, i, &value)) {
            printf(" %d", value);
        }
    }
    putchar('\n');

    int value;
    printf("index 99 -> %s\n",
           intarray_get(a, 99, &value) ? "value" : "refused");

    TRACE(intarray_count(a));

    /* Neither of these compiles — and that is the point:
     *   IntArray local;          // incomplete type
     *   printf("%zu", a->count); // dereferencing an incomplete type
     */

    intarray_destroy(a);
    intarray_destroy(NULL);        /* documented as safe */
    return EXIT_SUCCESS;
}
```

Building it

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -c intarray.c -o intarray.o
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -c main.c     -o main.o
gcc -fsanitize=address intarray.o main.o -o app
./app
```

Two compilations, then a link — chapter 2's model, now with something to link. Change only `main.c` and you recompile one file instead of the whole program, which is the practical reason projects are split at all. Chapter 29 automates this with a Makefile.

Prove the encapsulation

Add either of the commented lines in `main.c`:

```
IntArray local;
```

```
error: storage size of 'local' isn't known
```

```
printf("%zu", a->count);
```

```
error: invalid use of incomplete typedef 'IntArray'
```

The caller genuinely cannot reach inside. Now change the implementation — replace the array with a linked list, or add a field — and rebuild `intarray.c` only. `main.o` does not even need recompiling, because nothing it depends on changed.

Provoke the multiple-definition error

Add `int shared_counter = 0;` to `intarray.h` and rebuild:

```
/usr/bin/ld: main.o:(.bss+0x0): multiple definition of `shared_counter';
             intarray.o:(.bss+0x0): first defined here
```

Both translation units included the header and both defined the variable. Change the header to `extern int shared_counter;` and put `int shared_counter = 0;` in `intarray.c` — one declaration everywhere, one definition in exactly one place.

See the macro pitfall

```
#define SQUARE_BAD(x)  x * x
#define SQUARE_OK(x)  ((x) * (x))

printf("bad:  %d\n", SQUARE_BAD(2 + 3));   /* 11 */
printf("good: %d\n", SQUARE_OK(2 + 3));    /* 25 */

int i = 3;
printf("%d\n", SQUARE_OK(i++));            /* undefined: i modified twice, unsequenced */
printf("i is now %d\n", i);                /* often 5, but the behavior is undefined */
```

Run the preprocessor alone to see exactly what the compiler received:

```
gcc -E main.c | tail -40
```

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Unparenthesized macro body or parameter	Precedence changes the meaning	Parenthesize both; prefer a function.
A macro parameter used twice	Double evaluation of side effects	Use a `static inline` function.
A definition in a header	`multiple definition` at link time	`extern` in the header, definition in one `.c`.
Missing include guard	Redefinition errors	`#ifndef` / `#define` / `#endif`, or `#pragma once`.
No `static` on helpers	Names leak; collisions across files	Anything not in the header is `static`.
Exposing a struct that callers then depend on	Layout frozen; every change recompiles the world	Opaque pointer.
Semicolon after a macro definition	It becomes part of the expansion	`#define N 10`, not `#define N 10;`.
Large `#ifdef` blocks	Unbuilt branches never compile-checked	Keep them small; prefer runtime `if`.

8 Check yourself

Why is `#define SQUARE(x) ((x) * (x))` still not equivalent to a function?

Because the parameter appears twice, so any side effect in the argument happens twice — `SQUARE(i++)` modifies `i` twice with no sequence point between, which is undefined behavior. Parentheses fix precedence but not evaluation count. A `static inline` function evaluates each argument once and is inlined just as effectively at `-O2`.

Why does putting `int verbose = 0;` in a header cause a link error?

Because the preprocessor pastes the header into every source file that includes it, so each translation unit defines its own `verbose` and the linker finds several definitions of one external name. Headers declare; exactly one source file defines. Use `extern int verbose;` in the header.

What does `static` do to a file-scope function, and why use it?

It gives the name internal linkage, so no other translation unit can refer to it. That prevents accidental collisions at link time, documents that the function is an implementation detail, and lets the compiler optimize more aggressively because it can see every call. Everything not in the header should be `static`.

What does an opaque pointer buy you, and what does it cost?

It hides the structure's layout completely, so callers cannot depend on it and you may change the implementation without recompiling them — essential for a library with a stable interface. The costs are that callers must allocate through a create function rather than on the stack, and that every access goes through an accessor rather than reading a member directly.

Why keep `#ifdef` regions small?

Because only one configuration is compiled at a time, so code inside a branch you are not building is never seen by the compiler — it can contain syntax errors, unused variables, or stale API calls that surface only on another platform. Small regions limit the exposure, and a runtime `if` is type-checked in every build.

9 Where this leads

That completes the Intermediate level. You have a module with a public header, a hidden implementation, an error contract, and no leaks. Chapter 29 opens the Advanced level by building it properly: a Makefile, then CMake, then the same code as a static and a shared library — which is the form chapter 45 packages and chapter 46 calls from Python.

PART 3

Advanced

Tooling, machines, and systems

Chapters 29–42

ADVANCED · CHAPTER 29

Building Programs and Libraries

Typing compiler commands by hand stops working at about three files. This chapter replaces them with a build system, turns chapter 28's module into a real library, and opens the linker up far enough that `undefined reference` *stops being mysterious.*

BY THE END OF THIS CHAPTER YOU CAN

- Write a Makefile with automatic dependency tracking and multiple configurations.
- Write an equivalent CMake configuration and say when each tool is the better choice.
- Build and link both a static and a shared library.
- Explain why linking order matters and diagnose an unresolved symbol.
- Inspect an object file with `nm` and `objdump`.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Make

A Makefile is a set of rules: a target, the prerequisites it depends on, and the commands to rebuild it. Make rebuilds a target only when a prerequisite is newer.

```
target: prerequisites
<TAB>command
```

> **The recipe line must begin with a literal tab.** Spaces produce `Makefile:4: *** missing separator. Stop.` — the single most common Make error, and invisible in most editors. Configure your editor to keep real tabs in Makefiles.

A usable Makefile

```
CC      := gcc
CFLAGS  := -std=c17 -Wall -Wextra -g -MMD -MP
LDFLAGS :=
LDLIBS  :=

SRCS := main.c intarray.c
```

```
OBJS := $(SRCS:.c=.o)
DEPS := $(OBJS:.o=.d)
TARGET := app

.PHONY: all clean debug release asan test

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) $(LDFLAGS) $^ -o $@ $(LDLIBS)

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

debug:   CFLAGS += -O0 -DDEBUG
debug:   $(TARGET)

release: CFLAGS += -O2 -DNDEBUG
release: $(TARGET)

asan:    CFLAGS  += -fsanitize=address,undefined -O1
asan:    LDFLAGS += -fsanitize=address,undefined
asan:    $(TARGET)

test: asan
	./$(TARGET)

clean:
	rm -f $(OBJS) $(DEPS) $(TARGET)

-include $(DEPS)
```

SYMBOL	MEANS
`$@`	The target
`$<`	The first prerequisite
`$^`	All prerequisites
`:=`	Immediate assignment (prefer over `=`)
`.PHONY`	These targets are not files

The two lines that matter most

`-MMD -MP` makes the compiler emit a `.d` file listing every header each source includes. `-include $(DEPS)` pulls those in as extra rules. The result: editing `intarray.h` rebuilds every file that includes it. Without this, Make tracks only `.c` files and a header change silently produces a stale, inconsistent binary — the most confusing class of build bug there is.

`.PHONY` tells Make that `clean` is not a file. Without it, a file named `clean` in the directory would make `make clean` do nothing.

```
make            # build
make -j8        # build with eight parallel jobs
make asan       # build instrumented
make clean      # remove products
make -n         # show the commands without running them
```

Warning profiles

`-Wall -Wextra` is a floor, not a ceiling. Several valuable warnings are in neither, and a project decides once which set it holds itself to:

```
# beyond -Wall -Wextra, each of these is worth its noise
-Wpedantic            strict conformance to the chosen standard
-Wshadow              a local hiding an outer name
-Wconversion          implicit narrowing conversions
-Wsign-conversion     signed and unsigned conversions
-Wcast-qual           casting away const
-Wstrict-prototypes   () where (void) was meant
-Wwrite-strings       string literals typed as char *
-Wmissing-prototypes  a public function with no declaration
```

`-Wshadow` caught a real bug in chapter 17. `-Wconversion` catches the silent narrowing of chapter 7, at the cost of being noisy on existing code. `-Wstrict-prototypes` enforces chapter 11's `(void)` rule. Turn them on early on a new project; on an existing one, add them one at a time and fix the fallout before adding the next.

The final step is to make warnings fatal:

```
release: CFLAGS += -Werror
```

A warning that nobody fixes is noise, and noise hides the next real warning. Making the build fail is the only mechanism that keeps the count at zero. Apply it in the continuous integration configuration of chapter 35 rather than in the everyday developer build, so that work in progress is not blocked by an unused variable.

2 CMake

CMake generates build files rather than building directly, which is how a single configuration produces Makefiles on Linux, Ninja builds, and Visual Studio projects on Windows.

```
cmake_minimum_required(VERSION 3.16)
project(intarray_demo C)

set(CMAKE_C_STANDARD 17)
set(CMAKE_C_STANDARD_REQUIRED ON)

add_compile_options(-Wall -Wextra)

add_library(intarray STATIC intarray.c)
target_include_directories(intarray PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})

add_executable(app main.c)
target_link_libraries(app PRIVATE intarray)

option(ENABLE_ASAN "Build with AddressSanitizer" OFF)
if(ENABLE_ASAN)
    target_compile_options(app PRIVATE -fsanitize=address,undefined)
    target_link_options(app    PRIVATE -fsanitize=address,undefined)
endif()
```

```
cmake -S . -B build
cmake --build build -j8
cmake -S . -B build-asan -DENABLE_ASAN=ON
cmake --build build-asan
```

MAKE	CMAKE
Direct, no extra tool	Generates for several build systems
Unix-centric	Cross-platform, including Visual Studio
You write dependency handling	Handled for you
Fine for a small project	Standard for anything with dependencies

Learn Make first: it exposes what a build actually is, and CMake ultimately emits one. Use CMake for anything that must build on more than one platform or consume third-party libraries.

3 Static libraries

A static library is an archive of object files. The linker copies the ones you actually use into your executable.

```
gcc -std=c17 -Wall -Wextra -c intarray.c -o intarray.o
ar rcs libintarray.a intarray.o          # r=insert c=create s=index

gcc main.c -L. -lintarray -o app         # -L where, -l which
./app                                    # no runtime dependency
```

`-lintarray` means "find `libintarray.a` or `libintarray.so`" — the `lib` prefix and the extension are added for you. `-L.` adds the current directory to the search path.

The resulting executable is self-contained: the library's code is inside it, and the `.a` file is no longer needed to run.

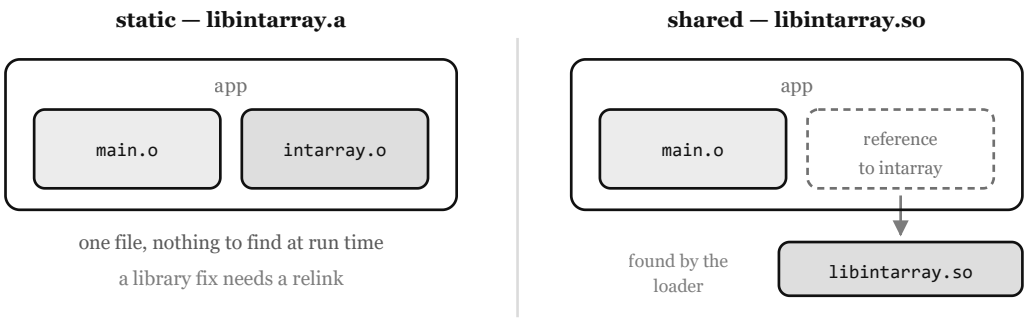
4 Shared libraries

```
gcc -std=c17 -Wall -Wextra -fPIC -c intarray.c -o intarray.o
gcc -shared intarray.o -o libintarray.so

gcc main.c -L. -lintarray -o app
LD_LIBRARY_PATH=. ./app                 # must be findable at run time
```

`-fPIC` — position-independent code — is required because a shared library can be loaded at any address, so it must not contain absolute addresses. Omit it and the link fails with a relocation error.

The `LD_LIBRARY_PATH` is needed because the loader searches standard directories, not the one you built in. That is a development convenience; chapter 45 covers installing properly and embedding a search path.



The same code, two places to keep it. The choice is about who has to be present at run time and who has to relink when the library changes.

	STATIC (.A)	SHARED (.SO, .DLL, .DYLIB)
Code lives	In the executable	In a separate file
Executable size	Larger	Smaller
Deployment	One file	Library must be present
Fixing a library bug	Relink every program	Replace one file
Memory with many users	One copy each	Shared between processes
Startup	Immediate	Dynamic linking cost

5 Linking order and symbol resolution

The traditional Unix linker processes its arguments left to right, keeping a list of symbols still undefined. When it reaches a library it takes only the members that resolve something currently outstanding — and then moves on.

```
gcc -lintarray main.c -o app     # FAILS
gcc main.c -lintarray -o app     # works
```

In the first form the library is examined before `main.o` has introduced any undefined symbols, so nothing is taken from it; by the time `main.o` needs `intarray_create`, the library has been passed. **Libraries go after the objects that use them**, and for mutually dependent libraries you may need to list one twice.

This is also why `-lm` belongs at the end of the command line, as chapter 24 noted.

Inspecting symbols

```
nm intarray.o
```

```
0000000000000000 t grow
0000000000000030 T intarray_create
0000000000000120 T intarray_destroy
                 U malloc
                 U realloc
```

LETTER	MEANING
`T`	Defined here, external (in `.text`)
`t`	Defined here, `static` — lowercase means local
`U`	Undefined: needed from elsewhere
`D` / `B`	Initialized / zero-initialized data
`R`	Read-only data

Note `grow` appearing as lowercase `t`: chapter 28's `static` keyword, visible in the object file. Other translation units cannot link against it, which is exactly what was intended.

```
nm -D libintarray.so | grep ' T '      # the exported interface
nm -u main.o                           # what main.o still needs
objdump -d intarray.o | head -30       # disassemble
ldd app                                # shared libraries an executable needs
```

6 Worked example: one module, four builds

Using `intarray.h`, `intarray.c`, and `main.c` from chapter 28.

Build it four ways

```
# 1. by hand, to remember what the tools automate
gcc -std=c17 -Wall -Wextra -g -c intarray.c -o intarray.o
gcc -std=c17 -Wall -Wextra -g -c main.c     -o main.o
gcc intarray.o main.o -o app_manual

# 2. as a static library
ar rcs libintarray.a intarray.o
gcc -std=c17 -Wall -Wextra -g main.c -L. -lintarray -o app_static

# 3. as a shared library
gcc -std=c17 -Wall -Wextra -g -fPIC -c intarray.c -o intarray_pic.o
gcc -shared intarray_pic.o -o libintarray.so
gcc -std=c17 -Wall -Wextra -g main.c -L. -lintarray -o app_shared

# 4. with the Makefile
make clean && make asan
```

```
ls -l app_manual app_static app_shared
LD_LIBRARY_PATH=. ./app_shared
```

Compare the sizes. The statically linked executable is larger by roughly the size of the library's code; the shared one is smaller but will not run without `libintarray.so` on the loader's path.

Watch incremental rebuilding work — and then fail

```
make clean && make
touch main.c && make          # rebuilds main.o only, then relinks
touch intarray.h && make      # rebuilds BOTH: the .d files did their job
```

Now remove `-MMD -MP` from `CFLAGS` and the `-include $(DEPS)` line, and repeat:

```
make clean && make
touch intarray.h && make
make: 'app' is up to date.
```

Make does not know the header exists. Change a struct in it and you get an executable whose two halves disagree about the layout — which manifests as memory corruption, not a build error. This is why dependency generation is not optional.

Reproduce the linking-order failure

```
gcc -std=c17 main.c -L. -lintarray -o ok
gcc -std=c17 -L. -lintarray main.c -o broken
```

```
/usr/bin/ld: /tmp/ccXXXX.o: in function `main':
main.c:(.text+0x1f): undefined reference to `intarray_create'
collect2: error: ld returned 1 exit status
```

Identical inputs, different order, different outcome. Confirm the library does contain the symbol:

```
nm libintarray.a | grep intarray_create
```

Inspect the symbol table

```
nm intarray.o | sort -k3
nm -u main.o
nm -D libintarray.so | grep ' T '
```

Check three things. Every `intarray_*` function is `T` — exported. `grow` is `t` — local, because it is `static`. And `main.o` lists exactly the `intarray_*` functions it calls as `U`, which is the

precise set the linker must satisfy.

Then remove `static` from `grow`, rebuild, and run `nm` again: it becomes `T`, exported from the library, and now collides with any other translation unit that happens to define a function called `grow`.

Compare configurations

```
make clean && make release && ls -l app && mv app app_release
make clean && make debug   && ls -l app && mv app app_debug
make clean && make asan    && ls -l app
```

The release build is smallest and has assertions compiled out; the debug build is unoptimized (all three carry symbols, because `CFLAGS` already has `-g`); the sanitizer build is several times larger because of the instrumentation. Having all three as named targets means switching costs one word, which is what makes running the instrumented build routine rather than exceptional.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Spaces instead of a tab in a recipe	`missing separator`	Use a real tab.
No header dependency tracking	Stale objects; inconsistent binary	`-MMD -MP` plus `-include $(DEPS)`.
Library before the objects that use it	`undefined reference`	Libraries last.
Shared library without `-fPIC`	Relocation error at link time	Compile position-independent.
Running a shared build without the path	`cannot open shared object file`	`LD_LIBRARY_PATH`, or install it — chapter 45.
No `.PHONY`	A file named `clean` stops the target working	Declare phony targets.
Recursive `=` where `:=` was meant	Re-evaluated on each use; slow and surprising	Prefer `:=`.
Sanitizer flags only at compile time	Link errors about missing runtime	Pass them to the link step too.

8 Check yourself

Why does `gcc -lfoo main.c` fail where `gcc main.c -lfoo` works?

Because the linker processes arguments left to right and takes from a library only the members that resolve symbols already known to be undefined. Listed first, the library is examined before `main.o` has introduced any requirements, so nothing is extracted — and by the time the requirement appears, the library is behind it. Libraries go after the objects that use them.

What do `-MMD -MP` do, and why does a build break without them?

They make the compiler write a dependency file listing every header a source includes, which the Makefile then includes as extra rules. Without them Make sees only `.c` files, so changing a header rebuilds nothing — leaving object files compiled against different versions of a structure. The result is memory corruption at runtime rather than an error at build time.

Why is `-fPIC` required for a shared library?

Because a shared library may be loaded at a different address in every process, so its code cannot contain absolute addresses. Position-independent code addresses everything relative to the program counter or through an indirection table. Without it, the link fails with a relocation error.

In `nm` output, what is the difference between `T` and `t`?

Both mean the symbol is defined in the text section of this object file. Uppercase means external linkage — other translation units can link against it. Lowercase means internal linkage, produced by `static`, so the name is invisible outside this file. Seeing your helpers as `t` confirms the encapsulation actually took effect.

When would you choose a static library over a shared one?

When single-file deployment matters more than size — a command-line tool distributed on its own, or a program that must run on systems where you cannot install libraries. Shared libraries win when many programs use the same code, when a bug fix should reach them all by replacing one file, or when startup memory across many processes matters.

9 Where this leads

Chapter 30 returns to the code itself, with the abstraction mechanisms C does offer: function pointers, callbacks, generic containers built on `void *`, variadic functions, and `_Generic`. Those are what let the `IntArray` you just packaged become an array of anything — and what make `qsort` possible at all.

ADVANCED · CHAPTER 30

Function Pointers, Callbacks, and Generic Programming

C has no templates, no interfaces, and no overloading. What it has are function pointers and `void *` — and between them they are enough to build everything those features provide, with the mechanism left visible.

BY THE END OF THIS CHAPTER YOU CAN

- Declare, assign, and call a function pointer, and read its declaration.
- Use `qsort` and `bsearch` with comparators you write.
- Build a dispatch table and simulate polymorphic behavior.
- Write a container that works over any element type through `void *`.
- Write a variadic function, and explain why it cannot check its arguments.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Function pointers

A function has an address, so a pointer can hold it:

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


int (*operation)(int, int) = add;      /* declare and assign */
int result = operation(3, 4);          /* call through it    */
```

Apply chapter 16's right-left rule to `int (*operation)(int, int)`: start at `operation`, the parentheses force left to `*` — a pointer to — then right to `(int, int)` — a function taking two ints — then left to `int` — returning int. The parentheses are essential: without them, `int *operation(int, int)` declares a function returning `int *`.

The `&` and `*` are optional in both directions. `operation = &add` and `(*operation)(3, 4)` are legal and mean the same as the shorter forms. Most code omits them.

A `typedef` makes the type usable:

```
typedef int (*BinaryOp)(int, int);


BinaryOp operation = add;
void apply_all(const int *v, size_t n, BinaryOp op);
```

This is the one case where hiding a pointer behind a `typedef` is standard practice, because the raw declaration is genuinely hard to read.

2 Callbacks and `qsort`

A callback is a function you pass to another function so it can call you back. The standard library's sort is the canonical example:

```
void qsort(void *base, size_t count, size_t size,
           int (*compare)(const void *, const void *));
```

`qsort` knows nothing about your element type. It receives the base address, the number of elements, the size of one, and a function that can order two of them. Everything type-specific lives in the comparator.

```
static int compare_int(const void *a, const void *b)
{
    int x = *(const int *)a;
    int y = *(const int *)b;
    return (x > y) - (x < y);      /* -1, 0, or 1 without overflow */
}


qsort(values, n, sizeof values[0], compare_int);
```

> **Do not write** `return x - y;`**.** It is the textbook comparator and it overflows: with `x = INT_MAX` and `y = -1` the subtraction is undefined behavior and the sign can come out wrong, so the sort silently produces a wrong order. `(x > y) - (x < y)` is branch-free, cannot overflow, and yields exactly −1, 0, or 1.

`bsearch` has the same shape and searches a *sorted* array with the same comparator:

```
int key = 42;
int *found = bsearch(&key, values, n, sizeof values[0], compare_int);
```

Two requirements people forget: the array must already be sorted by the same comparator, and the key is passed *by address*, because the comparator takes `const void *`.

3 Dispatch tables

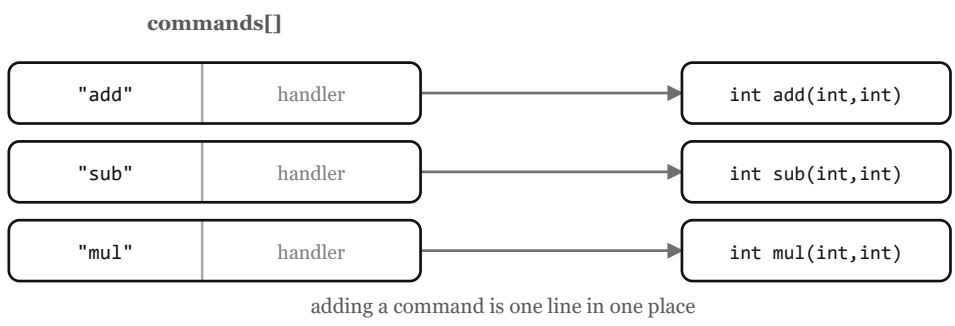
An array of function pointers replaces a long `switch` with a lookup:

```
typedef struct {
    const char *name;
    int (*handler)(int, int);
} Command;

static const Command commands[] = {
    { "add", add }, { "sub", sub }, { "mul", mul }
};

for (size_t i = 0; i < COUNT; i++) {
    if (strcmp(input, commands[i].name) == 0) {
        return commands[i].handler(a, b);
    }
}
```

The advantage is not speed — a `switch` over small integers compiles to a jump table anyway. It is that the table is *data*: adding a command is one line in one place, the set can be iterated to produce a help listing, and a plugin can extend it at run time.



adding a command is one line in one place

The switch becomes data: one row per command, and the row carries the code to run.

Polymorphism

Put function pointers inside a structure and you have virtual methods:

```
typedef struct Shape {
    const char *name;
    double (*area)(const struct Shape *);
    double (*perimeter)(const struct Shape *);
} Shape;

typedef struct {
    Shape base;            /* must be first: layout matters */
    double radius;
} Circle;

static double circle_area(const Shape *s)
{
    const Circle *c = (const Circle *)s;      /* safe: base is first */
    return 3.14159265358979 * c->radius * c->radius;
}
```

Because the base structure is the first member, a `Circle *` and a `Shape *` point at the same address, so the cast is well defined. This is precisely how C++ implements single inheritance, and how the Linux kernel's device model works — the mechanism is the same, only written out.

4 Generic containers with `void *`

A container that stores elements of unknown type must know three things: where an element is, how large it is, and how to operate on it.

```
typedef struct {
    void   *data;          /* raw bytes             */
    size_t  count;
    size_t  capacity;
    size_t  element_size;  /* supplied at creation  */
} Vector;


static void *element_at(const Vector *v, size_t i)
{
    return (char *)v->data + i * v->element_size;   /* char *: byte arithmetic */
}
```

The cast to `char *` is essential. Pointer arithmetic on `void *` is not standard C — GCC allows it as an extension, treating it as byte arithmetic, but portable code converts to `char *`, whose element size is 1 by definition.

GAIN	COST
One implementation for every type	No type checking at all
Works with any element size	Every access needs a cast
Standard, portable	`memcpy` per element instead of assignment

The alternative is a macro that generates a typed container per element type — type-safe and fast, at the cost of unreadable macros and duplicated code. Both approaches are used in real projects; `void *` is the simpler one to get right.

5 Variadic functions

```
#include <stdarg.h>


int sum_all(int count, ...)
{
    va_list args;
    va_start(args, count);          /* last NAMED parameter */

    int total = 0;
    for (int i = 0; i < count; i++) {
        total += va_arg(args, int); /* you state the type */
    }

    va_end(args);
    return total;
}
```

Three rules. Before C23 there must be at least one named parameter, and `va_start` names it. You must state each argument's type in `va_arg`, because nothing at run time records it. And `va_end` must be called before returning.

The function has no way to know how many arguments it received, so the count must arrive some other way — an explicit count, a sentinel value such as `NULL`, or a format string. That last option is what `printf` does, and it explains chapter 8's warning: a wrong specifier makes `printf` read the wrong number of bytes as the wrong type, and nothing can detect it.

Arguments also undergo *default promotions*: `float` becomes `double`, and types narrower than `int` become `int`. So `va_arg(args, float)` is always wrong — use `double`. Likewise `va_arg(args, char)`; use `int`.

To wrap a `printf`-like function, use the `v` variants that take a `va_list`:

```
void log_message(const char *fmt, ...)
{
    va_list args;
    va_start(args, fmt);
    vfprintf(stderr, fmt, args);    /* vfprintf, not fprintf */
    va_end(args);
}
```

6 `_Generic`

C11's `_Generic` selects an expression based on the *type* of a controlling expression, at compile time:

```
#define type_name(x) _Generic((x),        \
    int:         "int",                   \
    double:      "double",                \
    char *:      "char *",                \
    const char *: "const char *",         \
    default:     "something else")

#define absolute(x) _Generic((x), \
    int:    abs,                  \
    long:   labs,                 \
    double: fabs)(x)
```

This gives type-based dispatch with no runtime cost — the compiler picks one branch and discards the rest. It is how `<tgmath.h>` provides type-generic mathematics, and it is the closest C comes to overloading.

The limits are real: it selects on type only, the branches must all be valid expressions, and the syntax becomes unreadable past a handful of cases. Use it for thin wrappers, not as a general dispatch mechanism.

7 Worked example: the chapter 19 array, made generic

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <stdbool.h>
#include <math.h>

/* ================= generic vector ================= */

typedef struct {
    void   *data;
    size_t  count;
    size_t  capacity;
```

```
    size_t  element_size;
} Vector;

static bool vector_init(Vector *v, size_t element_size, size_t capacity)
{
    if (capacity == 0) capacity = 4;
    v->data = malloc(capacity * element_size);
    if (v->data == NULL) return false;
    v->count = 0;
    v->capacity = capacity;
    v->element_size = element_size;
    return true;
}

static void vector_destroy(Vector *v)
{
    free(v->data);
    v->data = NULL;
    v->count = v->capacity = 0;
}

static void *vector_at(const Vector *v, size_t i)
{
    return (char *)v->data + i * v->element_size;   /* byte arithmetic */
}

static bool vector_push(Vector *v, const void *element)
{
    if (v->count == v->capacity) {
        size_t bigger = v->capacity * 2;
        void *moved = realloc(v->data, bigger * v->element_size);
        if (moved == NULL) return false;
        v->data = moved;
        v->capacity = bigger;
    }
    memcpy(vector_at(v, v->count), element, v->element_size);
    v->count++;
    return true;
}
```

```
/* A callback applied to every element: the visitor pattern in C. */
static void vector_each(const Vector *v, void (*visit)(const void *, void *),
                        void *context)
{
    for (size_t i = 0; i < v->count; i++) {
        visit(vector_at(v, i), context);
    }
}

static void vector_sort(Vector *v, int (*cmp)(const void *, const void *))
{
    qsort(v->data, v->count, v->element_size, cmp);
}

/* ================ comparators ================ */

typedef struct { char name[16]; int score; } Student;

/* Never x - y: it overflows. */
static int cmp_int(const void *a, const void *b)
{
    int x = *(const int *)a, y = *(const int *)b;
    return (x > y) - (x < y);
}

static int cmp_double(const void *a, const void *b)
{
    double x = *(const double *)a, y = *(const double *)b;
    return (x > y) - (x < y);
}

static int cmp_student_score(const void *a, const void *b)
{
    const Student *x = a, *y = b;
    return (y->score > x->score) - (y->score < x->score);   /* descending */
}

static int cmp_student_name(const void *a, const void *b)
```

```
{
    return strcmp(((const Student *)a)->name, ((const Student *)b)->name);
}

/* ================= visitors ================= */

static void print_int(const void *e, void *ctx)
{
    (void)ctx;
    printf("%d ", *(const int *)e);
}

static void sum_int(const void *e, void *ctx)
{
    *(long *)ctx += *(const int *)e;
}

static void print_student(const void *e, void *ctx)
{
    (void)ctx;
    const Student *s = e;
    printf("    %-10s %3d\n", s->name, s->score);
}

/* ================= dispatch table ================= */

static int op_add(int a, int b) { return a + b; }
static int op_sub(int a, int b) { return a - b; }
static int op_mul(int a, int b) { return a * b; }

typedef struct {
    const char *name;
    int (*fn)(int, int);
    const char *help;
} Command;

static const Command commands[] = {
    { "add", op_add, "a + b" },
    { "sub", op_sub, "a - b" },
```

```
    { "mul", op_mul, "a * b" }
};
#define COMMAND_COUNT (sizeof commands / sizeof commands[0])

/* ================= polymorphism ================= */

typedef struct Shape {
    const char *name;
    double (*area)(const struct Shape *);
} Shape;

typedef struct { Shape base; double radius; }        Circle;
typedef struct { Shape base; double w, h; }          Rect;

static double circle_area(const Shape *s)
{
    return 3.14159265358979 * ((const Circle *)s)->radius
                            * ((const Circle *)s)->radius;
}
static double rect_area(const Shape *s)
{
    const Rect *r = (const Rect *)s;
    return r->w * r->h;
}

/* ================= variadic ================= */

static int sum_all(int count, ...)
{
    va_list args;
    va_start(args, count);
    int total = 0;
    for (int i = 0; i < count; i++) {
        total += va_arg(args, int);
    }
    va_end(args);
    return total;
}
```

```
/* A sentinel-terminated variadic function: no count needed. */
static void print_all(const char *first, ...)
{
    va_list args;
    va_start(args, first);
    for (const char *s = first; s != NULL; s = va_arg(args, const char *)) {
        printf("[%s] ", s);
    }
    va_end(args);
    putchar('\n');
}

static void log_message(const char *fmt, ...)
{
    va_list args;
    va_start(args, fmt);
    fprintf(stderr, "log: ");
    vfprintf(stderr, fmt, args);     /* the v-variant takes a va_list */
    va_end(args);
}

/* ================= _Generic ================= */

#define type_name(x) _Generic((x),   \
    int:          "int",             \
    long:         "long",            \
    double:       "double",          \
    char *:       "char *",          \
    const char *: "const char *",    \
    default:      "other")

int main(void)
{
    puts("== one implementation, three element types ==");

    Vector ints;
    vector_init(&ints, sizeof(int), 4);
    int sample[] = { 42, 7, 99, 13, 5, 77 };
    for (size_t i = 0; i < 6; i++) vector_push(&ints, &sample[i]);
```

```
    printf("  unsorted: ");  vector_each(&ints, print_int, NULL);
    vector_sort(&ints, cmp_int);
    printf("\n  sorted:   ");  vector_each(&ints, print_int, NULL);

    long total = 0;
    vector_each(&ints, sum_int, &total);
    printf("\n  sum via a visitor with context: %ld\n", total);

    int key = 99;
    int *found = bsearch(&key, ints.data, ints.count, sizeof(int), cmp_int);
    printf("  bsearch 99: %s\n", found ? "found" : "not found");

    Vector doubles;
    vector_init(&doubles, sizeof(double), 4);
    double d[] = { 2.5, 1.0, 3.75 };
    for (size_t i = 0; i < 3; i++) vector_push(&doubles, &d[i]);
    vector_sort(&doubles, cmp_double);
    printf("  doubles sorted: %g %g %g\n",
           *(double *)vector_at(&doubles, 0),
           *(double *)vector_at(&doubles, 1),
           *(double *)vector_at(&doubles, 2));

    Vector students;
    vector_init(&students, sizeof(Student), 4);
    Student roster[] = {
        { "Ada", 95 }, { "Dennis", 88 }, { "Ken", 91 }, { "Grace", 78 }
    };
    for (size_t i = 0; i < 4; i++) vector_push(&students, &roster[i]);

    puts("  students by score:");
    vector_sort(&students, cmp_student_score);
    vector_each(&students, print_student, NULL);
    puts("  students by name:");
    vector_sort(&students, cmp_student_name);
    vector_each(&students, print_student, NULL);
    puts("  the same sort and the same container, three element types");

    puts("\n== dispatch table ==");
```

```
    for (size_t i = 0; i < COMMAND_COUNT; i++) {
        printf("  %-4s (%s): %d\n", commands[i].name, commands[i].help,
               commands[i].fn(10, 3));
    }

    puts("\n== polymorphism through a function pointer ==");
    Circle c = { { "circle", circle_area }, 2.0 };
    Rect   r = { { "rect",   rect_area   }, 3.0, 4.0 };
    Shape *shapes[] = { (Shape *)&c, (Shape *)&r };
    for (size_t i = 0; i < 2; i++) {
        printf("  %-7s area = %.4f\n", shapes[i]->name,
               shapes[i]->area(shapes[i]));
    }

    puts("\n== variadic functions ==");
    printf("  sum_all(3, 1,2,3)       = %d\n", sum_all(3, 1, 2, 3));
    printf("  sum_all(5, 10,20,30,40,50) = %d\n",
           sum_all(5, 10, 20, 30, 40, 50));
    printf("  sentinel-terminated: ");
    print_all("alpha", "beta", "gamma", NULL);
    log_message("a wrapped printf, value = %d\n", 42);

    puts("\n== _Generic ==");
    int i_val = 1; double d_val = 1.0; const char *s_val = "x";
    printf("  1     is %s\n", type_name(i_val));
    printf("  1.0   is %s\n", type_name(d_val));
    printf("  \"x\"   is %s\n", type_name(s_val));

    vector_destroy(&ints);
    vector_destroy(&doubles);
    vector_destroy(&students);
    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o generic generic.c -lm
./generic
```

Demonstrate the comparator overflow

Replace `cmp_int` with the textbook version and sort a hostile array:

```
static int cmp_int_broken(const void *a, const void *b)
{
    return *(const int *)a - *(const int *)b;     /* overflows */
}

int hostile[] = { INT_MAX, -1, 0, INT_MIN, 5 };
qsort(hostile, 5, sizeof hostile[0], cmp_int_broken);
```

`INT_MAX - (-1)` overflows and the sign is wrong, so the sort concludes that `INT_MAX` is smaller than −1. Build with `-fsanitize=undefined` and it is reported:

```
runtime error: signed integer overflow: 2147483647 - -1 cannot be
represented in type 'int'
```

Swap in the subtraction-free form and the same array sorts correctly. This is one of the most widespread bugs in C code that otherwise looks textbook-perfect.

See that the type system is genuinely gone

```
double wrong = 3.14;
vector_push(&ints, &wrong);      /* compiles without a murmur */
```

The vector copies `sizeof(int)` bytes out of a `double` and stores the result as an integer. No warning, no error — the price of `void *`. This is exactly the check a C++ template or a Rust generic would perform, and it is what the macro-based alternative buys back.

Confirm the variadic promotion rule

```
static double average(int count, ...)
{
    va_list args;
    va_start(args, count);
    double total = 0;
    for (int i = 0; i < count; i++) {
        total += va_arg(args, float);    /* WRONG: always promoted to double */
    }
    va_end(args);
    return total / count;
}
```

Call it with `average(2, 1.5f, 2.5f)` and the behavior is undefined: the arguments were promoted to `double`, while `va_arg` asks for a `float`. GCC warns that `float` is promoted and compiles the call into a trap, so the program aborts; other compilers may silently return nonsense. Change `float` to `double` and it is correct.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`return x - y;` in a comparator	Overflow; wrong sort order	`(x > y) - (x < y)`.
`int *f(void)` when a function pointer was meant	Declares a function returning a pointer	`int (*f)(void)` — parentheses.
Passing the key by value to `bsearch`	Compile error or wrong results	Pass its address.
`bsearch` on an unsorted array	Silently wrong results	Sort with the same comparator first.
Pointer arithmetic on `void *`	Not standard C	Cast to `char *`.
`va_arg(args, float)`	Reads four bytes of an eight-byte value	Use `double`; likewise `int` for `char`.
Missing `va_end`	Undefined behavior on some ABIs	Always call it before returning.
Base struct not first in a "derived" type	The upcast points at the wrong member	Put it first.

9 Check yourself

Why must a comparator avoid `return x - y;`?

Because the subtraction can overflow — `INT_MAX - (-1)` is undefined behavior — and when it wraps, the sign is wrong, so the sort concludes the larger value is smaller. The result is a silently mis-ordered array. `(x > y) - (x < y)` computes the same three-way answer without arithmetic that can overflow.

How does `qsort` sort elements of a type it knows nothing about?

It is told everything it needs: the base address, the element count, the size of one element so it can compute addresses and swap bytes, and a comparator that can order any two. All type-specific knowledge lives in the comparator you supply, which casts the `const void *` arguments back to the real type.

Why cast to `char *` before doing arithmetic on a `void *`?

Because arithmetic on `void *` is not standard C — the element size is unknown, so the scaling is undefined. GCC allows it as an extension treating it as bytes, but portable code converts to `char *`, whose element size the standard fixes at 1, making byte arithmetic explicit and correct everywhere.

Why can a variadic function not check its arguments?

Because nothing about the extra arguments survives to run time — no count, no types. The function must be told separately, through an explicit count, a sentinel value, or a format string. `va_arg` simply reads the number of bytes for the type you name, so a wrong type reads the wrong bytes. This is precisely why a mismatched `printf` specifier is undefined behavior.

Why must the base structure be the first member when simulating inheritance?

Because the standard guarantees that a pointer to a structure, suitably converted, points to its first member and vice versa. With the base first, a `Circle *` and a `Shape *` hold the same address, so casting between them is well defined. Put another member first and the cast silently yields a pointer to the wrong bytes.

10 Where this leads

Chapter 31 applies the callback idea to the outside of the program: parsing command-line options, layering configuration from files and the environment, and designing an interface a shell script can rely on. Then chapters 32 and 33 use the generic container you just built as the foundation for linked structures and an abstract data type.

ADVANCED · CHAPTER 31

Command-Line Interfaces and the Environment

Chapter 23 read arguments. This chapter designs the interface they belong to: options with conventional spellings, configuration layered from several sources, exit codes a script can branch on, and diagnostics that identify themselves.

BY THE END OF THIS CHAPTER YOU CAN

- Parse short and long options, by hand and with `getopt`.
- Use `exit`, `_Exit`, and `atexit`, and say how each differs from returning from `main`.
- Read and set environment variables safely.
- Layer configuration from defaults, a file, the environment, and the command line.
- Design a tool that behaves correctly when driven by a script.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The conventions

Unix option syntax is not a standard, but it is so consistent that departing from it makes a tool feel broken.

FORM	MEANING
`-v`	A short flag
`-vqn`	Several short flags combined
`-o file` or `-ofile`	A short option with an argument
`--verbose`	A long flag
`--output=file` or `--output file`	A long option with an argument
`--`	End of options; everything after is an operand
`-`	By convention, standard input or output

The `--` separator matters more than it looks: it is the standard way to pass a filename that begins with a dash (the other is a path such as `./-report.txt`). A tool that does not honour it cannot open a file called `-report.txt`.

Two options every tool should have: `--help`, printing usage and exiting successfully, and `--version`. Both are what a person tries first.

2 getopt

POSIX provides a parser. It is not in the C standard, but it is available on Linux, macOS, and the BSDs, and it handles combined flags and attached arguments correctly.

```
#include <unistd.h>

int opt;
while ((opt = getopt(argc, argv, "vo:n:h")) != -1) {
    switch (opt) {
    case 'v': verbose = true;            break;
    case 'o': output_path = optarg;      break;   /* : means takes an argument */
    case 'n': count_text  = optarg;      break;
    case 'h': usage(); return EXIT_SUCCESS;
    case '?': usage(); return EXIT_FAILURE;       /* unknown or missing arg */
    }
}

/* optind now indexes the first operand */
for (int i = optind; i < argc; i++) {
    process_file(argv[i]);
}
```

NAME	MEANING
`optarg`	The argument of the option just returned
`optind`	Index of the next argument; after the loop, the first operand
`opterr`	Set to 0 to suppress `getopt`'s own error messages
`optopt`	The offending character when `'?'` is returned

A leading `:` in the option string — `":vo:n:h"` — makes a missing argument return `':'` rather than `'?'`, letting you distinguish "unknown option" from "option needs a value". Combined with `opterr = 0`, that gives you full control of the messages.

For long options, GNU provides `getopt_long` in `<getopt.h>`. It is not POSIX, so portable code either hand-parses the long forms or accepts the dependency.

3 Termination

WAY OUT	RUNS ATEXIT HANDLERS	FLUSHES STREAMS
`return` from `main`	Yes	Yes
`exit(status)`	Yes	Yes
`_Exit(status)`	No	No
`abort()`	No	No; raises `SIGABRT`

`exit` is what you call from deep inside a call chain when there is no sensible way to propagate the failure — though in a library that is almost always the wrong choice, because a library must let its caller decide.

```
#include <stdlib.h>

static void cleanup(void)
{
    remove(temp_path);
}

atexit(cleanup);        /* registered handlers run in reverse order */
```

`atexit` handlers run on normal termination only. They do not run on `_Exit`, on `abort`, or on a fatal signal — so they are not a substitute for real cleanup on the error path, and a temporary file can survive a crash. Chapter 39 covers signal handling.

Exit codes

Zero for success, non-zero for failure. Beyond that, a few conventions are widely followed:

CODE	CONVENTIONAL MEANING
0	Success
1	General failure
2	Misuse: bad options or arguments
>128	Killed by signal (128 + signal number)

Only the low eight bits reach the shell, so `exit(256)` is seen as 0. Keep codes in 0–125.

4 The environment

```
#include <stdlib.h>


const char *home = getenv("HOME");        /* NULL if not set */
if (home == NULL) {
    home = "/tmp";                        /* a default */
}
```

Three rules. `getenv` returns `NULL` for an unset variable, so always have a default. The returned pointer points into the environment block and may be invalidated by a later `setenv` or `putenv`, so copy it if you keep it. And the value is entirely under the user's control — treat it exactly as you would treat command-line input, which means validating it.

`setenv` and `unsetenv` are POSIX rather than standard C:

```
setenv("LANG", "C", 1);       /* 1 = overwrite if already set */
unsetenv("LANG");
```

> **Environment variables are untrusted input.** A `PATH` or a config path taken from the environment and used without validation is a classic privilege-escalation vector, which is why setuid programs are required to scrub their environment. Validate environment values with the same care as `argv`.

5 Layered configuration

A setting can come from four places. Apply them in this order, each overriding the last:

1. **Built-in defaults** — the program always works with no configuration at all.
2. **A configuration file** — system-wide, then per-user.
3. **Environment variables** — per-session.
4. **Command-line options** — this invocation, highest priority.

The ordering is the whole design. Anything more specific wins, so a user can set a default in a file, override it for a shell session with an environment variable, and override that for one run with a flag — without editing anything.

```
Config cfg;
config_set_defaults(&cfg);                /* 1 */
config_load_file(&cfg, "/etc/tool.conf");  /* 2 */
config_load_file(&cfg, user_config_path);  /* 2 */
config_load_env(&cfg);                    /* 3 */
config_apply_options(&cfg, argc, argv);    /* 4 */
```

Give the tool a way to show the result — `--show-config` — so a user can find out which layer won. Debugging a setting that "does not take effect" is otherwise miserable.

6 Worked example: a properly optioned filter

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>
#include <limits.h>
#include <unistd.h>

#define VERSION "1.0"
#define EXIT_USAGE 2

static const char *program_name = "filter";

typedef struct {
    bool        verbose;
    long        maximum;
    const char *output_path;
    const char *prefix;
} Config;

static void usage(FILE *to)
{
    fprintf(to,
        "usage: %s [options] [file...]\n"
        "\n"
        "  -v            verbose diagnostics on stderr\n"
```

```
        "  -n LIMIT      stop after LIMIT lines (default 0 = unlimited)\n"
        "  -o FILE       write to FILE instead of stdout\n"
        "  -p PREFIX     prefix every output line\n"
        "  -h            show this help and exit\n"
        "  -V            show the version and exit\n"
        "\n"
        "  With no file operands, or with '-', reads standard input.\n"
        "  Use '--' to end options before a filename beginning with '-'.\n"
        "\n"
        "Environment:\n"
        "  FILTER_PREFIX   default for -p\n"
        "  FILTER_MAX      default for -n\n\n"
        "\n"
        "Exit status: 0 success, 1 runtime error, %d usage error.\n",
        program_name, EXIT_USAGE);
}

static bool parse_long(const char *text, long *out)
{
    if (text == NULL || *text == '\0') return false;
    errno = 0;
    char *end;
    long value = strtol(text, &end, 10);
    if (end == text || *end != '\0' || errno == ERANGE) return false;
    *out = value;
    return true;
}

/* Layer 1: built-in defaults. */
static void config_defaults(Config *c)
{
    c->verbose     = false;
    c->maximum     = 0;
    c->output_path = NULL;
    c->prefix      = "";
}

/* Layer 3: the environment, overriding defaults. */
static bool config_from_env(Config *c)
```

```
{
    const char *prefix = getenv("FILTER_PREFIX");
    if (prefix != NULL) {
        c->prefix = prefix;
    }

    const char *max_text = getenv("FILTER_MAX");
    if (max_text != NULL) {
        long value;
        if (!parse_long(max_text, &value) || value < 0) {
            fprintf(stderr, "%s: FILTER_MAX is not a valid count: \"%s\"\n",
                    program_name, max_text);
            return false;          /* environment input is validated too */
        }
        c->maximum = value;
    }
    return true;
}

/* A registered cleanup handler. */
static char temp_marker[64];
static void on_exit_cleanup(void)
{
    if (temp_marker[0] != '\0') {
        remove(temp_marker);
    }
}

static bool process_stream(FILE *in, FILE *out, const Config *c,
                           const char *label, long *emitted)
{
    char line[512];
    long count = 0;

    while (fgets(line, sizeof line, in) != NULL) {
        if (c->maximum > 0 && *emitted >= c->maximum) {
            break;
        }
        if (strchr(line, '\n') == NULL && !feof(in)) {
```

```
            fprintf(stderr, "%s: %s: line too long, truncating\n",
                    program_name, label);
            int ch;
            while ((ch = fgetc(in)) != '\n' && ch != EOF) { }
        }
        line[strcspn(line, "\n")] = '\0';

        if (fprintf(out, "%s%s\n", c->prefix, line) < 0) {
            perror("write");
            return false;
        }
        count++;
        (*emitted)++;
    }

    if (ferror(in)) {
        fprintf(stderr, "%s: %s: read error\n", program_name, label);
        return false;
    }
    if (c->verbose) {
        fprintf(stderr, "%s: %s: %ld lines\n", program_name, label, count);
    }
    return true;
}

int main(int argc, char *argv[])
{
    if (argc > 0 && argv[0] != NULL) {
        program_name = argv[0];
    }
    atexit(on_exit_cleanup);

    Config cfg;
    config_defaults(&cfg);                      /* layer 1 */
    if (!config_from_env(&cfg)) {               /* layer 3 */
        return EXIT_USAGE;
    }

    /* Layer 4: command-line options, highest priority. */
```

```
    opterr = 0;                                 /* our messages, not getopt's */
    int opt;
    while ((opt = getopt(argc, argv, ":vn:o:p:hV")) != -1) {
        switch (opt) {
        case 'v':
            cfg.verbose = true;
            break;
        case 'n':
            if (!parse_long(optarg, &cfg.maximum) || cfg.maximum < 0) {
                fprintf(stderr, "%s: -n needs a non-negative number, got \"%s\"\n",
                        program_name, optarg);
                return EXIT_USAGE;
            }
            break;
        case 'o':
            cfg.output_path = optarg;
            break;
        case 'p':
            cfg.prefix = optarg;
            break;
        case 'h':
            usage(stdout);                      /* help goes to stdout */
            return EXIT_SUCCESS;
        case 'V':
            printf("%s %s\n", program_name, VERSION);
            return EXIT_SUCCESS;
        case ':':
            fprintf(stderr, "%s: option -%c requires an argument\n",
                    program_name, optopt);
            usage(stderr);                      /* errors go to stderr */
            return EXIT_USAGE;
        case '?':
        default:
            fprintf(stderr, "%s: unknown option -%c\n", program_name, optopt);
            usage(stderr);
            return EXIT_USAGE;
        }
    }
```

```
    if (cfg.verbose) {
        fprintf(stderr, "%s: config: verbose=%d max=%ld out=%s prefix=\"%s\"\n",
                program_name, cfg.verbose, cfg.maximum,
                cfg.output_path ? cfg.output_path : "<stdout>", cfg.prefix);
    }

    FILE *out = stdout;
    if (cfg.output_path != NULL) {
        out = fopen(cfg.output_path, "w");
        if (out == NULL) {
            fprintf(stderr, "%s: %s: %s\n",
                    program_name, cfg.output_path, strerror(errno));
            return EXIT_FAILURE;
        }
    }

    int status = EXIT_SUCCESS;
    long emitted = 0;

    if (optind == argc) {
        /* No operands: read standard input, as a filter should. */
        if (!process_stream(stdin, out, &cfg, "-", &emitted)) {
            status = EXIT_FAILURE;
        }
    } else {
        for (int i = optind; i < argc; i++) {
            if (strcmp(argv[i], "-") == 0) {
                if (!process_stream(stdin, out, &cfg, "-", &emitted)) {
                    status = EXIT_FAILURE;
                }
                continue;
            }
            FILE *in = fopen(argv[i], "r");
            if (in == NULL) {
                fprintf(stderr, "%s: %s: %s\n",
                        program_name, argv[i], strerror(errno));
                status = EXIT_FAILURE;
                continue;                       /* keep going, like grep */
            }
```

```
            if (!process_stream(in, out, &cfg, argv[i], &emitted)) {
                status = EXIT_FAILURE;
            }
            fclose(in);
        }
    }

    if (out != stdout && fclose(out) != 0) {
        perror("closing output");
        status = EXIT_FAILURE;
    }
    return status;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o filter filter.c
```

Exercise every path

```
printf 'a\nb\nc\n' | ./filter
printf 'a\nb\nc\n' | ./filter -p '> '
printf 'a\nb\nc\n' | ./filter -n 2
printf 'a\nb\nc\n' | ./filter -vp '# '          # combined short options
./filter -h                     ; echo "status $?"
./filter -V                     ; echo "status $?"
./filter -x                     ; echo "status $?"   # unknown
./filter -n                     ; echo "status $?"   # missing argument
./filter -n abc                 ; echo "status $?"   # bad argument
./filter no_such_file           ; echo "status $?"
printf 'x\n' | FILTER_PREFIX='env: ' ./filter        # environment layer
FILTER_PREFIX='env: ' ./filter -p 'flag: ' <<< x     # flag wins
```

CASE	EXPECTED
`-h`	Usage on *stdout*, status 0 — help is the requested output
`-x`	Message and usage on *stderr*, status 2
`-n` with nothing	"requires an argument", status 2
Missing file	Named error, continues to the next file, status 1
`FILTER_PREFIX` and `-p`	The flag wins: layer 4 over layer 3
No operands	Reads standard input

Why help goes to stdout and errors go to stderr

```
./filter -h | head -3          # works: help is on stdout
./filter -x 2> errors.txt      # the error is captured, not the output
```

A user who typed `--help` asked for that text, so it is the program's output and belongs on `stdout` where it can be piped to a pager. A user who mistyped an option did not ask for anything, so the message is a diagnostic. Getting this backwards is one of the most common flaws in homemade tools.

The `--` separator

```
touch -- -weird.txt
./filter -weird.txt            # fails: parsed as options
./filter -- -weird.txt         # works
```

`getopt` handles `--` for you, stopping option parsing and leaving `optind` pointing at the operand. Hand-written parsers usually forget it.

Confirm it composes

```
seq 1 10 | ./filter -p 'n=' -n 3 | wc -l        # 3
seq 1 10 | ./filter -v -n 3 2>/dev/null | wc -l # 3, diagnostics discarded
./filter missing1 missing2 2>/dev/null; echo $? # 1, after trying both
```

The last is deliberate: like `grep` and `cat`, the tool reports each failure and continues, then exits non-zero. Stopping at the first missing file would be a defensible alternative — but it must be a decision, and it must be documented.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Help printed to `stderr`	Cannot be piped to a pager	`--help` goes to `stdout`; errors to `stderr`.
Not honouring `--`	Cannot process a file starting with a dash	Use `getopt`, which handles it.
Ignoring `optind`	Operands are missed or options reprocessed	Start the operand loop at `optind`.
Trusting an environment variable	Injection or crash on bad input	Validate exactly as you would `argv`.
Keeping the pointer from `getenv`	Invalidated by a later `setenv`	Copy the value if you retain it.
Relying on `atexit` for cleanup	Does not run on a crash or signal	Clean up explicitly; chapter 27's single exit path.
`exit` called from a library	Removes the caller's ability to recover	Return a status; only `main` decides to exit.
Exit codes above 255	Truncated to the low eight bits	Stay within 0–125.

8 Check yourself

Why does `--help` output belong on `stdout` while an unknown-option message belongs on `stderr`?

Because `--help` is what the user asked for — it is the program's output, and they will want to pipe it to a pager or grep it. An error message is a diagnostic about a request that failed; putting it on `stderr` keeps it visible when output is redirected and keeps it out of any pipeline consuming the results.

What does `--` do, and why must a tool honour it?

It ends option parsing: everything after it is treated as an operand even if it begins with a dash. Without it a bare `-report.txt` is read as options, and the user has to fall back on `./-report.txt`. `getopt` implements this automatically and leaves `optind` pointing past the separator.

Why should environment variables be validated as carefully as command-line arguments?

Because they are equally under the user's control, and often under the control of whoever launched the process. An unvalidated path or numeric setting taken from the environment can crash the program or, in a privileged context, redirect it to attacker-chosen files. The only safe assumption is that both sources are hostile.

When does an `atexit` handler fail to run?

On `_Exit`, on `abort`, and on any fatal signal — a segmentation fault, or the user pressing Ctrl-C without a handler installed. So `atexit` is a convenience for normal termination, not a guarantee. Anything that must be cleaned up reliably needs explicit handling on the error path and, for signals, the treatment in chapter 39.

Why layer configuration defaults, file, environment, then command line?

Because specificity should win. A default makes the program work with no setup; a file records a persistent preference; an environment variable scopes a change to one session; a flag applies to exactly one invocation. Each layer overriding the previous means a user can change any setting at the narrowest scope that suits them, without editing anything more permanent.

9 Where this leads

Chapter 32 returns inside the program to build linked data structures — the first place where chapter 21's self-referential types and chapter 19's allocation meet. From there chapter 33 adds trees and hash tables behind the opaque interface of chapter 28, completing the set of containers a C programmer is expected to be able to write from memory.

ADVANCED · CHAPTER 32

Linked Data Structures

A linked list is where chapter 21's self-referential structure and chapter 19's allocation meet. It is also the structure most often chosen for the wrong reason — so this chapter builds it properly and then measures when it is actually the right choice.

BY THE END OF THIS CHAPTER YOU CAN

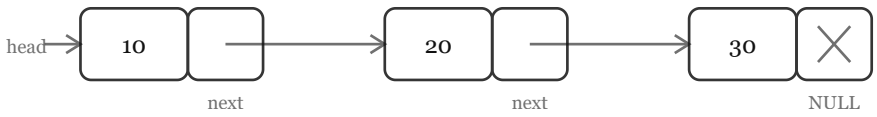
- Build singly, doubly, and circular linked lists with correct insertion and removal.
- Remove a node without special-casing the head, using a pointer to a pointer.
- Implement a stack and a queue on one node type.
- Reverse a list in place.
- Say when a list beats an array and demonstrate when it does not.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The node

```
typedef struct Node {
    int          value;
    struct Node *next;
} Node;
```

Each node holds its data and the address of the next. The list itself is just a pointer to the first node; the end is marked by a `next` of `NULL`.



Nodes need not be adjacent in memory — which is both the strength and the weakness.

2 Insertion and removal

```
/* At the front: O(1) */
bool push_front(Node **head, int value)
{
    Node *node = malloc(sizeof *node);
    if (node == NULL) return false;
    node->value = value;
    node->next  = *head;
    *head = node;                    /* modifies the caller's pointer */
    return true;
}
```

Note the `Node **`. The function must change the caller's `head`, so it takes its address — chapter 15's pointer-to-pointer, in its most common real use.

Removal, and the special case

Written naively, removal needs a separate branch for the head:

```
void remove_value(Node **head, int value)
{
    if (*head == NULL) return;

    if ((*head)->value == value) {     /* special case */
        Node *dead = *head;
        *head = dead->next;
        free(dead);
        return;
    }
    Node *prev = *head;                /* general case */
    while (prev->next != NULL && prev->next->value != value) {
        prev = prev->next;
    }
    if (prev->next != NULL) {
        Node *dead = prev->next;
```

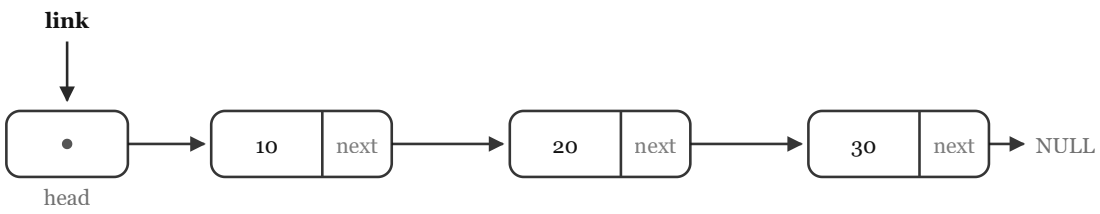
```
        prev->next = dead->next;
        free(dead);
    }
}
```

The special case exists because the head is stored in a different kind of place from every other link. Track a pointer *to the pointer* instead and the distinction vanishes:

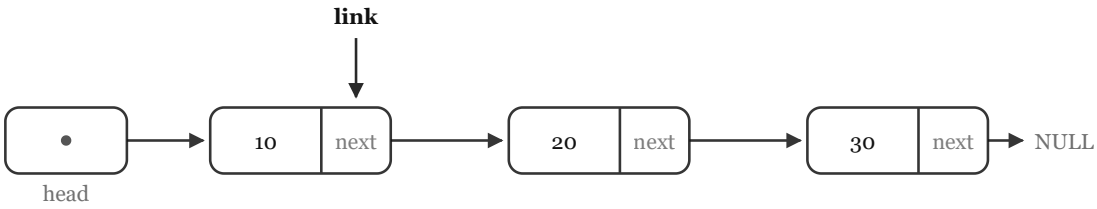
```
void remove_value(Node **head, int value)
{
    for (Node **link = head; *link != NULL; link = &(*link)->next) {
        if ((*link)->value == value) {
            Node *dead = *link;
            *link = dead->next;
            free(dead);
            return;
        }
    }
}
```

`link` points at whatever holds the address of the current node — the caller's `head` for the first, the previous node's `next` thereafter. Writing through it updates the right place either way. Half the code, no branch, and no way to get the head case wrong.

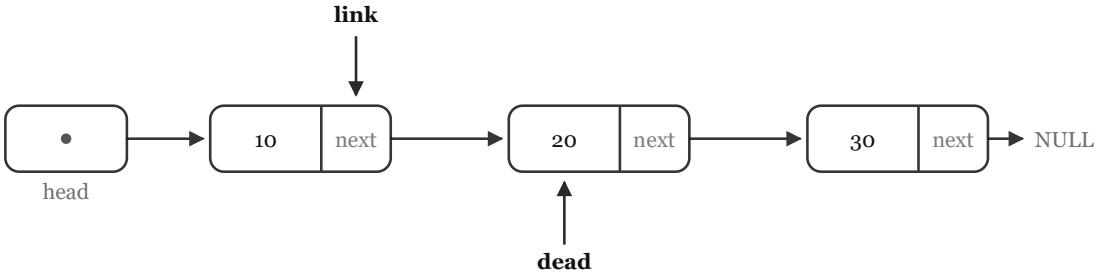
> **This idiom is worth internalizing.** Linus Torvalds has described understanding it as the difference between someone who writes C and someone who understands pointers. It generalizes: any time you find yourself special-casing the first element of a linked structure, a pointer to a pointer removes the case.



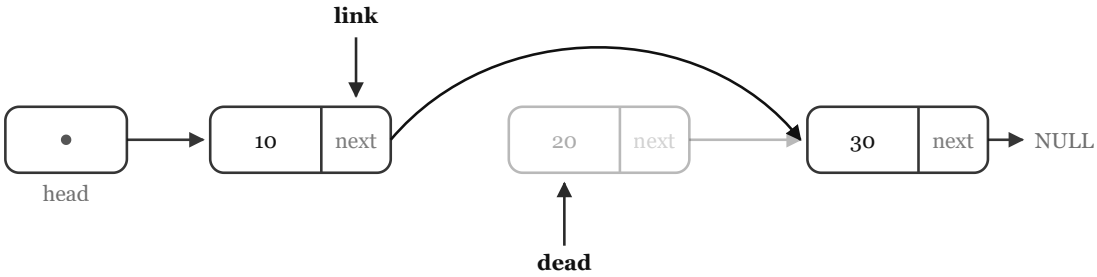
`link = head`. It holds the address of the caller's `head` pointer — not of a node. `*link` is the first node, whose value is 10, so this is not the one.



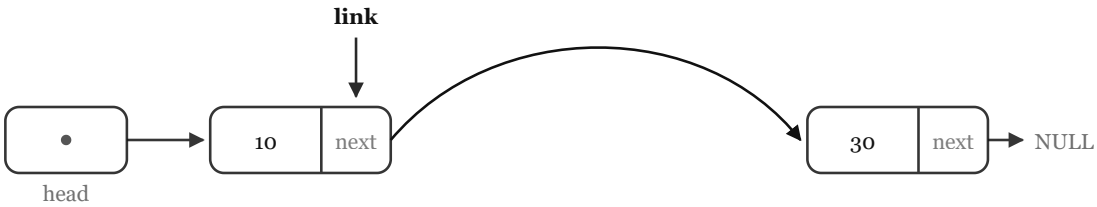
`link = &(*link)->next`. Now it holds the address of node 10's `next` field. `*link` is the node holding 20: the match.



`Node *dead = *link;` — name the node before the next statement overwrites the only pointer to it. Forget this and the memory is unreachable and unfreeable.



`*link = dead->next;` — the write goes *through* `link`, so it lands in node 10's `next`. Had the match been the first node, `link` would still be aiming at `head` and this same statement would have updated that instead. That is the whole reason there is no special case.



`free(dead);` — the list runs 10 → 30 and nothing leaked. Four statements, and not one branch on whether the node happened to be first.

3 Doubly linked and circular

```
typedef struct DNode {
    int           value;
    struct DNode *prev;
    struct DNode *next;
} DNode;
```

A backward link costs one pointer per node and buys two things: traversal in both directions, and removal of a node in O(1) when you already hold it — no search for the predecessor.

```
void dlist_remove(DList *l, DNode *node)
{
    if (node->prev != NULL) node->prev->next = node->next;
    else                    l->head          = node->next;

    if (node->next != NULL) node->next->prev = node->prev;
    else                    l->tail          = node->prev;

    free(node);
}
```

The two special cases are back. A **circular list with a sentinel** removes them: a permanent dummy node whose `next` and `prev` close the ring, so every real node always has both neighbours.

```
/* with a sentinel, removal has no branches at all */
node->prev->next = node->next;
node->next->prev = node->prev;
free(node);
```

This is how the Linux kernel's `list_head` works, and it is the reason its list code has no null checks. The cost is one node's worth of memory per list.

4 Stacks and queues

Both are lists with a restricted interface — which is the point: the restriction is what makes the structure useful.

	STACK (LIFO)	QUEUE (FIFO)
Add	push — at the front	enqueue — at the tail
Remove	pop — from the front	dequeue — from the head
Both O(1) if	Head pointer only	Head *and* tail pointers

A queue without a tail pointer makes enqueue O(n), because every insertion walks to the end. Keeping a tail pointer is the whole design decision — and the thing to get right is updating it when the list becomes empty.

5 When a list actually wins

OPERATION	ARRAY	LINKED LIST
Index element *i*	O(1)	O(n)
Insert at front	O(n)	O(1)
Insert at back	O(1) amortized	O(1) with a tail pointer
Insert in the middle, position known	O(n)	O(1)
Insert in the middle, position searched	O(n)	O(n)
Memory per element	The element	Element + pointer + allocator overhead
Cache behavior	Contiguous	Scattered

Read the fifth row carefully. The textbook claim that lists are better for middle insertion assumes you already hold a pointer to the position. If you have to *find* it first, the search dominates and both are O(n) — and the array is faster in practice because the traversal is sequential.

That is why, on modern hardware, an array wins far more often than complexity tables suggest. Each list node is a separate allocation, typically 32 bytes for 4 bytes of data, scattered across memory so that every step is a potential cache miss. Chapter 34 measures this directly.

Use a linked list when you hold pointers to elements and splice them between containers, when elements must not move because other things point at them, when you are building an intrusive structure as the kernel does, or when a stack or queue is all you need. **Otherwise start with an array.**

6 Worked example: list, stack, queue, and a measurement

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

/* ================ singly linked list ================ */

typedef struct Node {
    int          value;
    struct Node *next;
} Node;

typedef struct {
    Node   *head;
    Node   *tail;          /* for O(1) append and enqueue */
    size_t  count;
} List;

static void list_init(List *l)
{
    l->head = l->tail = NULL;
    l->count = 0;
}

static void list_destroy(List *l)
{
    Node *n = l->head;
    while (n != NULL) {
        Node *next = n->next;      /* save before freeing */
        free(n);
        n = next;
    }
    list_init(l);
}

static bool list_push_front(List *l, int value)
```

```
{
    Node *n = malloc(sizeof *n);
    if (n == NULL) return false;
    n->value = value;
    n->next  = l->head;
    l->head  = n;
    if (l->tail == NULL) l->tail = n;      /* first node is also the tail */
    l->count++;
    return true;
}

static bool list_push_back(List *l, int value)
{
    Node *n = malloc(sizeof *n);
    if (n == NULL) return false;
    n->value = value;
    n->next  = NULL;
    if (l->tail != NULL) l->tail->next = n;
    else                 l->head       = n;
    l->tail = n;
    l->count++;
    return true;
}

/* The pointer-to-pointer idiom: no special case for the head. */
static bool list_remove(List *l, int value)
{
    for (Node **link = &l->head; *link != NULL; link = &(*link)->next) {
        if ((*link)->value == value) {
            Node *dead = *link;
            *link = dead->next;
            if (dead == l->tail) {
                l->tail = (link == &l->head) ? NULL : (Node *)((char *)link -
                          offsetof(Node, next));
            }
            free(dead);
            l->count--;
            return true;
        }
```

```
    }
    return false;
}

/* Reverse in place: three pointers, one pass, no allocation. */
static void list_reverse(List *l)
{
    Node *prev = NULL;
    Node *curr = l->head;
    l->tail = l->head;
    while (curr != NULL) {
        Node *next = curr->next;   /* save the rest */
        curr->next = prev;          /* flip this link */
        prev = curr;                /* advance */
        curr = next;
    }
    l->head = prev;
}

static void list_print(const List *l, const char *label)
{
    printf("  %-12s (%zu)", label, l->count);
    for (const Node *n = l->head; n != NULL; n = n->next) {
        printf(" %d", n->value);
    }
    putchar('\n');
}

/* ================ stack ================ */

typedef struct { List inner; } Stack;

static void stack_init(Stack *s)    { list_init(&s->inner); }
static void stack_destroy(Stack *s) { list_destroy(&s->inner); }
static bool stack_push(Stack *s, int v) { return list_push_front(&s->inner, v); }

static bool stack_pop(Stack *s, int *out)
{
    Node *top = s->inner.head;
```

```
    if (top == NULL) return false;
    *out = top->value;
    s->inner.head = top->next;
    if (s->inner.head == NULL) s->inner.tail = NULL;
    free(top);
    s->inner.count--;
    return true;
}

/* ================= queue ================= */

typedef struct { List inner; } Queue;

static void queue_init(Queue *q)    { list_init(&q->inner); }
static void queue_destroy(Queue *q) { list_destroy(&q->inner); }
static bool queue_enqueue(Queue *q, int v) { return list_push_back(&q->inner, v); }

static bool queue_dequeue(Queue *q, int *out)
{
    Node *front = q->inner.head;
    if (front == NULL) return false;
    *out = front->value;
    q->inner.head = front->next;
    if (q->inner.head == NULL) q->inner.tail = NULL;
    free(front);
    q->inner.count--;
    return true;
}

/* ================= measurement ================= */

#define N 200000

static double time_list_traversal(void)
{
    List l;
    list_init(&l);
    for (int i = 0; i < N; i++) list_push_front(&l, i);
```

```
    clock_t t0 = clock();
    long sum = 0;
    for (int pass = 0; pass < 20; pass++) {
        for (const Node *n = l.head; n != NULL; n = n->next) sum += n->value;
    }
    double secs = (double)(clock() - t0) / CLOCKS_PER_SEC;
    printf("    (list sum %ld)\n", sum);
    list_destroy(&l);
    return secs;
}

static double time_array_traversal(void)
{
    int *a = malloc(N * sizeof *a);
    if (a == NULL) return -1.0;
    for (int i = 0; i < N; i++) a[i] = i;

    clock_t t0 = clock();
    long sum = 0;
    for (int pass = 0; pass < 20; pass++) {
        for (int i = 0; i < N; i++) sum += a[i];
    }
    double secs = (double)(clock() - t0) / CLOCKS_PER_SEC;
    printf("    (array sum %ld)\n", sum);
    free(a);
    return secs;
}

int main(void)
{
    puts("== list operations ==");
    List l;
    list_init(&l);
    for (int i = 1; i <= 5; i++) list_push_back(&l, i * 10);
    list_print(&l, "built");

    list_push_front(&l, 5);
    list_print(&l, "push_front");
```

```
    printf("  remove 30: %s\n", list_remove(&l, 30) ? "yes" : "no");
    list_print(&l, "after");

    printf("  remove 5 (the head): %s\n", list_remove(&l, 5) ? "yes" : "no");
    list_print(&l, "after");
    puts("  no special case was needed for the head");

    printf("  remove 999: %s\n", list_remove(&l, 999) ? "yes" : "no");

    list_reverse(&l);
    list_print(&l, "reversed");

    list_destroy(&l);

    puts("\n== stack: last in, first out ==");
    Stack s;
    stack_init(&s);
    for (int i = 1; i <= 4; i++) {
        stack_push(&s, i);
        printf("  pushed %d\n", i);
    }
    int v;
    while (stack_pop(&s, &v)) printf("  popped %d\n", v);
    printf("  pop on empty: %s\n", stack_pop(&s, &v) ? "value" : "refused");
    stack_destroy(&s);

    puts("\n== queue: first in, first out ==");
    Queue q;
    queue_init(&q);
    for (int i = 1; i <= 4; i++) {
        queue_enqueue(&q, i);
        printf("  enqueued %d\n", i);
    }
    while (queue_dequeue(&q, &v)) printf("  dequeued %d\n", v);
    queue_destroy(&q);

    puts("\n== traversal: list versus array ==");
    double list_secs  = time_list_traversal();
    double array_secs = time_array_traversal();
```

```
    printf("  %d elements, 20 passes\n", N);
    printf("  linked list : %.4f s\n", list_secs);
    printf("  array       : %.4f s\n", array_secs);
    if (array_secs > 0) {
        printf("  the array is %.1fx faster for the same operation count\n",
               list_secs / array_secs);
    }
    puts("  same asymptotic complexity; the difference is cache locality");

    printf("\n  memory per element: array %zu bytes, list node %zu bytes\n",
           sizeof(int), sizeof(Node));
    puts("  plus allocator overhead of roughly 16 bytes per node");

    return EXIT_SUCCESS;
}
```

The `list_remove` tail fix-up above uses `offsetof`, which needs `<stddef.h>`. In production code a doubly linked list avoids the gymnastics entirely; the version here is deliberately kept singly linked to show the pointer-to-pointer idiom in isolation.

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o lists lists.c
./lists
```

What the measurement shows

Both traversals are O(n) and execute the same number of additions. The array is typically three to ten times faster. Nothing about the algorithm differs — the array's elements are contiguous, so each cache line fetch brings sixteen integers, while each list node is a separate allocation and every step is a potential cache miss.

The memory figures make the same point from the other side: 4 bytes of data per element in the array, against 16 bytes of node plus roughly 16 bytes of allocator bookkeeping in the list. Eight times the memory for the same information.

This is why "use a linked list for frequent insertion" is bad advice without qualification. Measure before choosing, and chapter 34 shows how.

Verify the idiom removes the branch

Rewrite `list_remove` with the naive head special case, then run both against these four inputs: removing the head, removing a middle node, removing the tail, and removing from an

empty list. Both versions must behave identically — and the pointer-to-pointer version is half the length with no branch to get wrong.

Check for leaks deliberately

Delete the `list_destroy(&l)` call at the end of the list section of `main` and rerun:

```
==1234==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 64 byte(s) in 4 object(s) allocated from:
    #1 0x... in list_push_back lists.c:52
```

Every node is a separate allocation, so a list is the structure most likely to leak. The `init` / `destroy` pairing from chapter 19 is not optional here.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`free(n); n = n->next;`	Use after free	Save `next` before freeing.
Losing the head while traversing	The whole list leaks	Walk a copy of the pointer.
Forgetting to update `tail`	Append writes after a freed node	Update it on every insert and remove.
Special-casing the head by hand	Duplicated logic; one branch wrong	The `Node **` idiom.
Passing `Node *` where the head must change	The caller's pointer is unchanged	Pass `Node **`.
Not freeing nodes	A leak per node	Paired `destroy`.
Queue without a tail pointer	Enqueue becomes O(n)	Keep both ends.
Choosing a list for indexed access	O(n) per lookup and poor locality	Use an array.

8 Check yourself

Why does an insertion function take `Node **head` rather than `Node *head`?

Because inserting at the front changes which node the caller's `head` points at, and arguments are copied. Passing the address of the pointer lets the function write through it and update the caller's variable — the same reason `scanf` needs `&`, applied to a pointer.

How does the `Node **link` idiom remove the head special case?

`link` points at whatever storage holds the address of the current node: the caller's `head` for the first node, the previous node's `next` field thereafter. Writing `*link = dead->next` updates the correct location in both cases, so the head is not special and the branch disappears.

Why is an array usually faster than a list even when their complexity is identical?

Cache locality. An array's elements are contiguous, so one cache line fetch brings many of them; list nodes are separate allocations scattered across memory, so each step can miss. The operation count is the same but the memory system behaves completely differently, and on modern hardware that dominates.

Under what condition is a list genuinely better for middle insertion?

Only when you already hold a pointer to the insertion point. Then the splice is O(1) against the array's O(n) shift. If you must search for the position first, the search is O(n) for both — and the array wins in practice because its traversal is sequential.

Why does `free(n); n = n->next;` fail?

Because after `free` the block belongs to the allocator and reading `n->next` is a use-after-free. It often appears to work, since the memory has not yet been reused, which is why the bug survives testing. Save the successor in a local variable before freeing.

9 Where this leads

Chapter 33 adds the structures where the pointer work pays off: binary search trees, where recursion from chapter 18 becomes indispensable, and hash tables, which give constant-time lookup. Both go behind the opaque interface of chapter 28, and the complexity reasoning that this chapter's measurement started gets its proper vocabulary.

ADVANCED · CHAPTER 33

Trees, Hash Tables, and Abstract Data Types

Two structures that answer the same question — "is this key present, and what is its value?" — with completely different trade-offs. Both go behind the opaque interface of chapter 28, so the caller can be given one and later handed the other without noticing.

BY THE END OF THIS CHAPTER YOU CAN

- Implement a binary search tree with insertion, lookup, and recursive traversal.
- Explain why an unbalanced tree degrades to a list, and what balancing fixes.
- Implement a hash table with chaining, and choose a reasonable hash function.
- Use big-O notation accurately, and say where it misleads.
- Design an abstract data type whose implementation can be replaced without touching callers.

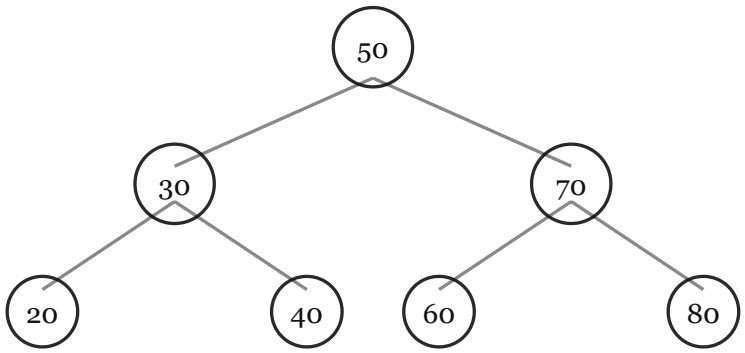
Code for this chapter: github.com/sercankulcu/programming-with-c

1 Binary search trees

```
typedef struct TreeNode {
    int              key;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;
```

The invariant is the whole structure: **every key in the left subtree is less than the node's key, and every key in the right subtree is greater**. Maintaining it makes lookup a sequence of comparisons that halves the remaining candidates at each step.

searching 40:
50 → left
30 → right
found



Three comparisons instead of seven. The saving compounds: a million keys need about twenty.

Insertion and lookup

```
/* The Node ** idiom from chapter 32: no special case for an empty tree. */
static bool tree_insert(TreeNode **link, int key)
{
    while (*link != NULL) {
        if      (key < (*link)->key) link = &(*link)->left;
        else if (key > (*link)->key) link = &(*link)->right;
        else return false;                 /* already present */
    }
    TreeNode *n = malloc(sizeof *n);
    if (n == NULL) return false;
    n->key = key;
    n->left = n->right = NULL;
    *link = n;
    return true;
}
```

Traversal

Three orders, distinguished by when the node itself is visited relative to its subtrees:

ORDER	VISIT	PRODUCES
In-order	left, node, right	**Sorted order**
Pre-order	node, left, right	A copyable structure
Post-order	left, right, node	Safe destruction — children freed first

Post-order is not a curiosity: freeing a tree in any other order frees a node before its children, losing the pointers to them.

> **An unbalanced tree is a linked list.** Insert 1, 2, 3, 4, 5 in order and every node becomes the right child of the previous one. Lookup is then O(n), not O(log n) — and sorted input is extremely common. Self-balancing variants (AVL, red-black) restore the guarantee by rotating after insertion; their implementation is beyond this chapter, but knowing that a plain BST has no worst-case guarantee is not.

2 Hash tables

A hash table converts a key into an array index directly, so lookup costs one computation and one access rather than a search.

```
size_t index = hash(key) % bucket_count;
```

Different keys can produce the same index — a *collision* — and how you handle that defines the table. The simplest and most common approach is **chaining**: each bucket holds a linked list of the entries that landed there.

```
typedef struct Entry {
    char         *key;
    int           value;
    struct Entry *next;        /* chain within this bucket */
} Entry;

typedef struct {
    Entry **buckets;
    size_t  bucket_count;
    size_t  count;
} HashMap;
```

The hash function

```
/* FNV-1a: short, well distributed, and good enough for a hash table. */
static size_t hash_string(const char *s)
{
    size_t h = 14695981039346656037u;        /* offset basis */
    for (; *s != '\0'; s++) {
        h ^= (unsigned char)*s;
        h *= 1099511628211u;                 /* prime */
    }
    return h;
}
```

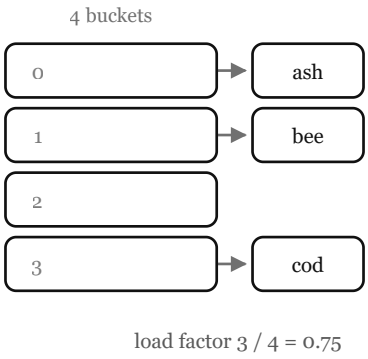
A good hash spreads keys evenly and is cheap. It does *not* need to be cryptographic — but note that a predictable hash lets an attacker choose keys that all land in one bucket, turning every lookup into a list walk. That is a real denial-of-service technique, which is why language runtimes randomize their hash seed. Chapter 48 returns to it.

Never write your own hash for production use; FNV-1a, xxHash, or SipHash are all better than anything improvised.

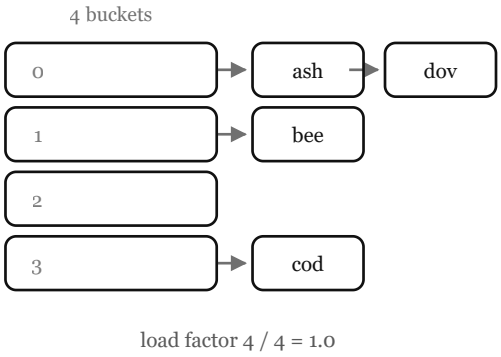
Load factor and resizing

Performance depends on the *load factor* — entries divided by buckets. Below about 0.75, chains stay short and lookup is effectively constant. Above it, chains lengthen and the table degrades toward a list.

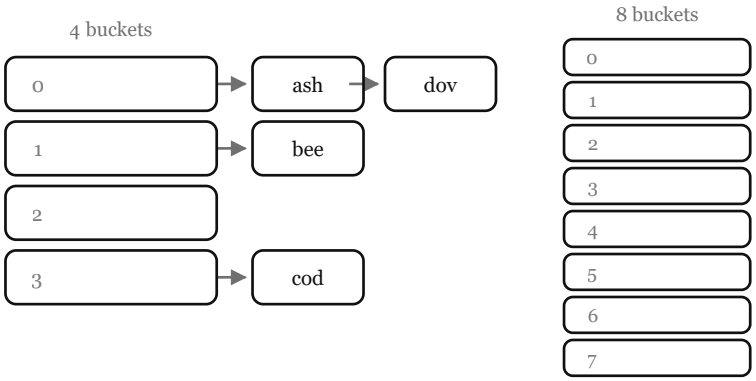
The fix is to double the bucket count and **rehash every entry**, because each key's index depends on the bucket count. Like the growable array of chapter 19, the doubling makes insertion O(1) amortized.



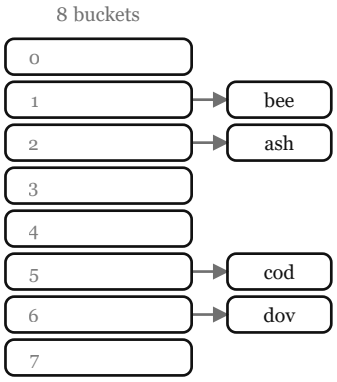
Three entries in four buckets. Load factor 0.75, chains of one, lookup effectively constant.



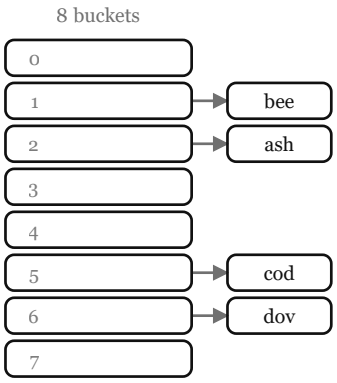
A fourth entry arrives and lands in bucket 0. Load factor 1.0, and one chain is now two long. Keep going and the table degrades toward a list.



Double the bucket count: allocate eight, all empty. Nothing has moved yet.



Now rehash *every* entry. This is the part people forget: the index is `hash(key) % bucket_count`, so changing the count changes where every key belongs. None of the four stays put.



load factor 4 / 8 = 0.5

The old array is freed. Load factor is back to 0.5 and chains are short again. Doubling is what makes this O(1) amortized, exactly as in chapter 19's growable array.

3 Complexity, honestly

OPERATION	ARRAY	SORTED ARRAY	LIST	BST (BALANCED)	HASH TABLE
Lookup by key	O(n)	O(log n)	O(n)	O(log n)	O(1) average
Insert	O(1) amortized	O(n)	O(1)	O(log n)	O(1) average
Delete	O(n)	O(n)	O(1) with pointer	O(log n)	O(1) average
Sorted iteration	O(n log n)	O(n)	O(n log n)	**O(n)**	O(n log n)
Worst case	—	—	—	O(n) unbalanced	O(n) all colliding

Three things big-O does not tell you, and all three decide real choices.

Constants matter. O(1) with a hash computation and a pointer chase can be slower than O(log n) over twenty contiguous elements. For small collections, a linear scan of an array beats both.

Memory locality is invisible to the notation. Chapter 32 measured an array beating a list several-fold at identical complexity. Trees and hash tables have the same problem: every node is a separate allocation.

The average is not the worst case. A hash table is O(1) on average and O(n) when every key collides. If an adversary chooses the keys, you get the worst case on purpose.

Choose a tree when you need sorted iteration or range queries; a hash table when you only need lookup by exact key; an array when the collection is small or you iterate more than you search.

4 The abstract data type

Both structures answer the same question, so both can hide behind one interface — chapter 28's opaque pointer, now earning its keep:

```
/* map.h — the caller sees no implementation at all */
#include <stddef.h>
#include <stdbool.h>

typedef struct Map Map;

Map   *map_create(void);
void   map_destroy(Map *m);
bool   map_put(Map *m, const char *key, int value);
bool   map_get(const Map *m, const char *key, int *out);
bool   map_remove(Map *m, const char *key);
size_t map_count(const Map *m);
void   map_each(const Map *m, void (*visit)(const char *, int, void *), void *ctx);
```

Nothing in that header says "tree" or "hash". You can ship the tree version, discover that sorted iteration is never used, switch to a hash table, and every caller keeps compiling unchanged — the property chapter 45 shows is essential once other people depend on your library.

Note the visitor in `map_each`: it is chapter 30's callback, and it exists because the caller cannot write a traversal loop over a structure it cannot see.

5 Worked example: one interface, two implementations

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>

/* ================ binary search tree ================ */

typedef struct TreeNode {
    int              key;
    struct TreeNode *left, *right;
} TreeNode;

static bool tree_insert(TreeNode **link, int key)
```

```
{
    while (*link != NULL) {
        if      (key < (*link)->key) link = &(*link)->left;
        else if (key > (*link)->key) link = &(*link)->right;
        else return false;
    }
    TreeNode *n = malloc(sizeof *n);
    if (n == NULL) return false;
    n->key = key;
    n->left = n->right = NULL;
    *link = n;
    return true;
}

static bool tree_contains(const TreeNode *n, int key, int *steps)
{
    while (n != NULL) {
        (*steps)++;
        if      (key < n->key) n = n->left;
        else if (key > n->key) n = n->right;
        else return true;
    }
    return false;
}

static void tree_inorder(const TreeNode *n, void (*visit)(int, void *), void *ctx)
{
    if (n == NULL) return;
    tree_inorder(n->left, visit, ctx);      /* recursion from chapter 18 */
    visit(n->key, ctx);
    tree_inorder(n->right, visit, ctx);
}

/* Post-order: children must be freed before their parent. */
static void tree_destroy(TreeNode *n)
{
    if (n == NULL) return;
    tree_destroy(n->left);
    tree_destroy(n->right);
```

```
    free(n);
}

static int tree_height(const TreeNode *n)
{
    if (n == NULL) return 0;
    int l = tree_height(n->left), r = tree_height(n->right);
    return 1 + (l > r ? l : r);
}

/* ================= hash map, behind an opaque handle ================= */

typedef struct Entry {
    char         *key;
    int           value;
    struct Entry *next;
} Entry;

struct Map {                                /* defined here only */
    Entry **buckets;
    size_t  bucket_count;
    size_t  count;
};
typedef struct Map Map;

static size_t hash_string(const char *s)
{
    size_t h = 1469598103934665603u;
    for (; *s != '\0'; s++) {
        h ^= (unsigned char)*s;
        h *= 1099511628211u;
    }
    return h;
}

static Map *map_create(void)
{
    Map *m = malloc(sizeof *m);
    if (m == NULL) return NULL;
```

```
    m->bucket_count = 8;
    m->buckets = calloc(m->bucket_count, sizeof *m->buckets);
    if (m->buckets == NULL) { free(m); return NULL; }
    m->count = 0;
    return m;
}

static void map_destroy(Map *m)
{
    if (m == NULL) return;
    for (size_t b = 0; b < m->bucket_count; b++) {
        Entry *e = m->buckets[b];
        while (e != NULL) {
            Entry *next = e->next;        /* save before freeing */
            free(e->key);
            free(e);
            e = next;
        }
    }
    free(m->buckets);
    free(m);
}

/* Every key's index depends on bucket_count, so all must be rehashed. */
static bool map_grow(Map *m)
{
    size_t new_count = m->bucket_count * 2;
    Entry **fresh = calloc(new_count, sizeof *fresh);
    if (fresh == NULL) return false;

    for (size_t b = 0; b < m->bucket_count; b++) {
        Entry *e = m->buckets[b];
        while (e != NULL) {
            Entry *next = e->next;
            size_t i = hash_string(e->key) % new_count;
            e->next = fresh[i];
            fresh[i] = e;
            e = next;
        }
```

```
    }
    free(m->buckets);
    m->buckets = fresh;
    m->bucket_count = new_count;
    return true;
}

static bool map_put(Map *m, const char *key, int value)
{
    size_t i = hash_string(key) % m->bucket_count;
    for (Entry *e = m->buckets[i]; e != NULL; e = e->next) {
        if (strcmp(e->key, key) == 0) {
            e->value = value;             /* replace */
            return true;
        }
    }

    if ((m->count + 1) * 4 > m->bucket_count * 3) {   /* load factor 0.75 */
        if (!map_grow(m)) return false;
        i = hash_string(key) % m->bucket_count;
    }

    Entry *e = malloc(sizeof *e);
    if (e == NULL) return false;

    size_t bytes = strlen(key) + 1;
    e->key = malloc(bytes);
    if (e->key == NULL) { free(e); return false; }
    memcpy(e->key, key, bytes);

    e->value = value;
    e->next = m->buckets[i];
    m->buckets[i] = e;
    m->count++;
    return true;
}

static bool map_get(const Map *m, const char *key, int *out, int *steps)
{
```

```
    size_t i = hash_string(key) % m->bucket_count;
    for (Entry *e = m->buckets[i]; e != NULL; e = e->next) {
        if (steps) (*steps)++;
        if (strcmp(e->key, key) == 0) {
            *out = e->value;
            return true;
        }
    }
    return false;
}

static bool map_remove(Map *m, const char *key)
{
    size_t i = hash_string(key) % m->bucket_count;
    for (Entry **link = &m->buckets[i]; *link != NULL; link = &(*link)->next) {
        if (strcmp((*link)->key, key) == 0) {
            Entry *dead = *link;
            *link = dead->next;
            free(dead->key);
            free(dead);
            m->count--;
            return true;
        }
    }
    return false;
}

static size_t map_count(const Map *m)
{
    return m->count;                    /* maintained by put and remove */
}

static void map_stats(const Map *m)
{
    size_t used = 0, longest = 0;
    for (size_t b = 0; b < m->bucket_count; b++) {
        size_t len = 0;
        for (const Entry *e = m->buckets[b]; e != NULL; e = e->next) len++;
        if (len > 0) used++;
```

```
        if (len > longest) longest = len;
    }
    printf("  %zu entries in %zu buckets, %zu used, longest chain %zu, load %.2f\n",
           m->count, m->bucket_count, used, longest,
           (double)m->count / (double)m->bucket_count);
}

/* ================ visitors ================ */

static void print_key(int key, void *ctx)
{
    (void)ctx;
    printf("%d ", key);
}

static void count_key(int key, void *ctx)
{
    (void)key;
    (*(long *)ctx)++;
}

int main(void)
{
    puts("== binary search tree ==");
    TreeNode *root = NULL;
    int keys[] = { 50, 30, 70, 20, 40, 60, 80 };
    for (size_t i = 0; i < 7; i++) tree_insert(&root, keys[i]);

    printf("  in-order (sorted): ");
    tree_inorder(root, print_key, NULL);
    printf("\n  height = %d for 7 keys (log2(7) = 2.8)\n", tree_height(root));

    int steps = 0;
    printf("  contains 40: %s in %d comparisons\n",
           tree_contains(root, 40, &steps) ? "yes" : "no", steps);
    steps = 0;
    printf("  contains 45: %s in %d comparisons\n",
           tree_contains(root, 45, &steps) ? "yes" : "no", steps);
```

```
    long visited = 0;
    tree_inorder(root, count_key, &visited);
    printf("  visitor counted %ld nodes\n", visited);
    tree_destroy(root);

    puts("\n== the degenerate case ==");
    TreeNode *sorted_root = NULL;
    for (int i = 1; i <= 1000; i++) tree_insert(&sorted_root, i);
    printf("  1000 keys inserted in sorted order: height = %d\n",
           tree_height(sorted_root));
    steps = 0;
    tree_contains(sorted_root, 1000, &steps);
    printf("  finding the last key took %d comparisons, not ~10\n", steps);
    puts("  the tree became a linked list; this is why balancing exists");
    tree_destroy(sorted_root);

    TreeNode *balanced = NULL;
    int mid[] = { 500, 250, 750, 125, 375, 625, 875 };
    for (size_t i = 0; i < 7; i++) tree_insert(&balanced, mid[i]);
    for (int i = 1; i <= 1000; i++) tree_insert(&balanced, i);
    printf("  same keys, better insertion order: height = %d\n",
           tree_height(balanced));
    tree_destroy(balanced);

    puts("\n== hash map behind an opaque handle ==");
    Map *m = map_create();
    if (m == NULL) return EXIT_FAILURE;

    const char *words[] = { "alpha","beta","gamma","delta","epsilon",
                            "zeta","eta","theta","iota","kappa" };
    for (int i = 0; i < 10; i++) map_put(m, words[i], i * 10);

    map_stats(m);

    int value;
    steps = 0;
    printf("  get \"gamma\": %s = %d in %d probes\n",
           map_get(m, "gamma", &value, &steps) ? "found" : "missing",
           value, steps);
```

```
    printf("  get \"omega\": %s\n",
           map_get(m, "omega", &value, NULL) ? "found" : "missing");

    map_put(m, "gamma", 999);
    map_get(m, "gamma", &value, NULL);
    printf("  after replacing: gamma = %d, count still %zu\n",
           value, map_count(m));

    printf("  remove \"beta\": %s\n", map_remove(m, "beta") ? "yes" : "no");
    map_stats(m);
    map_destroy(m);

    puts("\n== growth and rehashing ==");
    Map *big = map_create();
    char key[32];
    for (int i = 0; i < 100; i++) {
        snprintf(key, sizeof key, "key%03d", i);
        map_put(big, key, i);
    }
    map_stats(big);
    puts("  buckets doubled from 8 as the load factor crossed 0.75;");
    puts("  every entry was rehashed, because the index depends on the count");

    clock_t t0 = clock();
    long hits = 0;
    for (int pass = 0; pass < 20000; pass++) {
        snprintf(key, sizeof key, "key%03d", pass % 100);
        if (map_get(big, key, &value, NULL)) hits++;
    }
    printf("  20000 lookups in %.4f s, %ld hits\n",
           (double)(clock() - t0) / CLOCKS_PER_SEC, hits);
    map_destroy(big);

    return EXIT_SUCCESS;
}
```

`map_count` in the listing is that one-line accessor returning `m->count` — the smallest possible demonstration of why an opaque type needs accessors.

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o adt adt.c
./adt
```

What the degenerate case shows

A thousand keys inserted in sorted order produce a tree of height 1000 — every node is the right child of the previous one. Finding the last key takes 1000 comparisons instead of about 10. The structure is a linked list wearing a tree's type.

Sorted input is not a pathological case you can dismiss; it is what you get from a sorted file, a database export, or an auto-incrementing identifier. A plain BST has no worst-case guarantee, which is precisely why AVL and red-black trees exist and why the standard library of most languages uses one.

Watch the rehash

Insert entries one at a time and print `map_stats` after each. The bucket count jumps 8 → 16 → 32 → 64 → 128 → 256 as the load factor crosses 0.75, and after each jump the longest chain drops back to one or two. Remove the growth check and insert a thousand keys into eight buckets: the longest chain reaches about 125 and lookup becomes a list walk.

Make the collision attack concrete

Replace `hash_string` with a deliberately terrible hash:

```
static size_t hash_string(const char *s) { return (size_t)strlen(s); }
```

Now insert a hundred keys of the same length. Every one lands in the same bucket, `map_stats` reports a longest chain of 100, and the lookup timing rises by two orders of magnitude. An attacker who can choose your keys and knows your hash function can do exactly this on purpose — which is why production hash tables randomize their seed.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Freeing a tree in pre-order	Children leaked; pointers lost	Post-order: children first.
Assuming a BST is O(log n)	Sorted input makes it O(n)	Balance it, or use a hash table.
Forgetting to rehash after growth	Entries unreachable at their old index	Reinsert every entry.
Storing the caller's key pointer	Dangles when the caller frees it	Copy the key; free it in `destroy`.
Never resizing	Chains grow; O(1) becomes O(n)	Grow past a load factor of about 0.75.
A weak or predictable hash	Everything collides; denial of service	FNV-1a or better; randomize the seed.
Exposing the struct in the header	Implementation frozen	Opaque handle plus accessors.
Choosing O(1) for a ten-element set	Slower than a linear scan	Measure; constants dominate at small sizes.

7 Check yourself

Why must a tree be freed in post-order?

Because a node holds the only pointers to its children. Freeing the parent first makes those pointers unreachable, so the subtrees leak — and reading them afterwards is a use-after-free. Post-order frees both children before the node that referenced them.

What happens to a binary search tree given sorted input, and why does it matter?

Every key is larger than the last, so each becomes the right child of its predecessor and the tree degenerates into a linked list of height n. Lookup drops from O(log n) to O(n). It matters because sorted input is common — exported data, sequential identifiers — not a contrived case. Self-balancing trees exist to remove the possibility.

Why must every entry be rehashed when a hash table grows?

Because an entry's bucket is `hash(key) % bucket_count`, and changing the count changes the result for nearly every key. Entries left in their old buckets would be unreachable at the index the lookup now computes. Growth therefore costs O(n), which the doubling amortizes to O(1) per insertion.

Why is a hash table's O(1) only an average?

Because it assumes keys distribute evenly across buckets. If they all hash to the same bucket, every lookup walks one long chain and the cost is O(n). That can happen by bad luck with a weak hash, or deliberately if an attacker can choose keys and knows the function — which is why real implementations randomize the hash seed.

What does the opaque handle buy you here specifically?

The freedom to replace a tree with a hash table, or add a cache, or change the growth policy, without recompiling or even notifying callers. Because the header never reveals the structure, no caller can hold a dependency on it. The cost is that every field access must go through an accessor function you must remember to write.

8 Where this leads

Chapter 34 takes the measurements this chapter started seriously: why a contiguous array beats a pointer-chasing structure at the same complexity, how to read the assembly the compiler produced, and how to profile rather than guess. It is where the gap between big-O and the stopwatch finally gets explained.

ADVANCED · CHAPTER 34

Performance and Memory Layout

Chapters 32 and 33 produced measurements that big-O could not explain: identical complexity, several times the runtime. The explanation is the memory hierarchy — and once you can see it, most performance work in C turns out to be about data layout rather than clever code.

BY THE END OF THIS CHAPTER YOU CAN

- Explain the cost of a cache miss and why locality dominates arithmetic.
- Reorganize data so that a loop touches memory sequentially.
- Read the assembly the compiler produced and see what it optimized.
- Profile a program with `perf` or `gprof` instead of guessing.
- Say when to stop optimizing and why clarity usually wins.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The memory hierarchy

LEVEL	TYPICAL SIZE	LATENCY	RELATIVE
Register	a few hundred bytes	0 cycles	1
L1 cache	32–64 KB	~4 cycles	4
L2 cache	256 KB – 1 MB	~12 cycles	12
L3 cache	8–32 MB	~40 cycles	40
Main memory	8–64 GB	~200 cycles	**200**
SSD	terabytes	~100 000 cycles	100 000

A main-memory access costs roughly two hundred cycles — enough time to execute several hundred arithmetic instructions. On modern hardware, **the question is rarely how many operations you perform; it is how often you wait for memory**.

Cache lines

Memory is not fetched a byte at a time. The hardware transfers a whole *cache line*, almost universally 64 bytes. Read one `int` and the fifteen around it arrive for free.

That single fact explains chapter 32's result. Walking an array touches consecutive addresses, so one miss serves sixteen elements. Walking a linked list follows pointers to scattered allocations, so nearly every step is a fresh miss — sixteen times the memory traffic for the same arithmetic.

The hardware also *prefetches*: detecting a sequential pattern, it fetches ahead so the data has arrived before you ask. Prefetchers recognize forward and backward strides. They cannot predict pointer chasing.

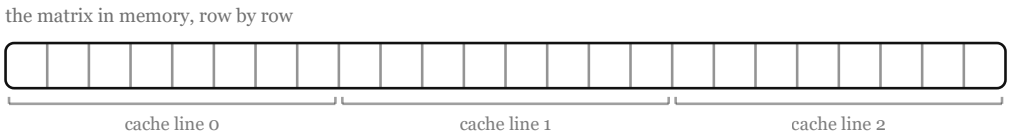
2 Data layout

Loop order

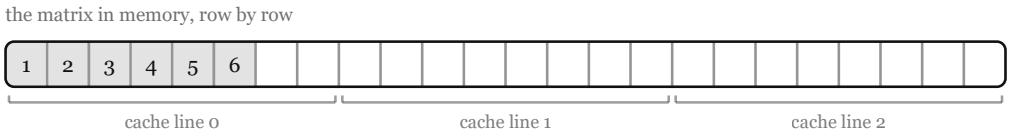
```
/* fast: sequential in memory */
for (size_t r = 0; r < ROWS; r++)
    for (size_t c = 0; c < COLS; c++)
        sum += m[r][c];

/* slow: strides a whole row per step */
for (size_t c = 0; c < COLS; c++)
    for (size_t r = 0; r < ROWS; r++)
        sum += m[r][c];
```

Identical results, identical operation count. Chapter 12 explained row-major layout; this is where it costs you. On a large matrix the difference is commonly five- to tenfold.

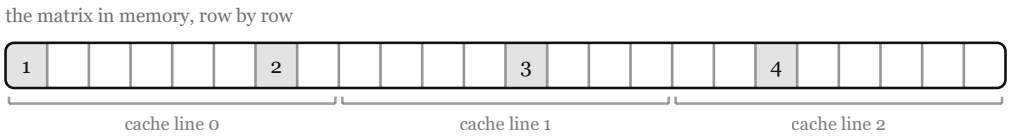


A 4×6 matrix is one contiguous block, stored row by row — chapter 12's row-major layout. The hardware never fetches one `int`; it fetches a whole cache line, sixteen of them on a typical machine (eight drawn here).



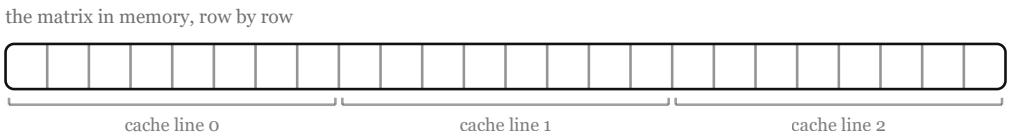
six accesses, one line fetched

Iterating rows outermost, columns innermost walks straight along that block. The first access misses and fetches a line; the next several are already there.



four accesses, three lines fetched

Swap the loops and each step jumps a whole row. Four accesses, three different lines — and on a real matrix, a fresh line almost every time, with the prefetcher unable to help because the stride is not sequential.



same operations, same result

five to tenfold difference in time

Identical results, identical operation count, identical big-O. The only difference is how often you wait for memory.

Structure of arrays

When a loop touches one field of many records, the other fields still occupy the cache lines:

```
/* array of structures — the conventional layout */
typedef struct { int id; char name[32]; double score; } Student;
Student students[N];
for (size_t i = 0; i < N; i++) total += students[i].score;
/* each 48-byte record carries one score and 40 bytes you did not want */

/* structure of arrays — one array per field */
typedef struct {
    int    *id;
    char  (*name)[32];
    double *score;
} Students;
for (size_t i = 0; i < N; i++) total += s.score[i];
/* each line carries eight scores */
```

This is the standard transformation in numerical and game code. It is also less readable and harder to maintain, so apply it where a profiler says it matters — not everywhere.

Field ordering

Chapter 22 showed that declaring members largest-first eliminates padding. The same change helps here: a smaller structure means more records per cache line.

3 Optimization levels

FLAG	EFFECT
`-O0`	None. Fast builds, predictable debugging. The default.
`-O1`	Cheap optimizations, still reasonable to debug.
`-O2`	The usual release setting. Inlining, vectorization, loop transformation.
`-O3`	More aggressive; sometimes slower through code growth. Measure.
`-Os`	Optimize for size — chapter 53's default on embedded targets.
`-march=native`	Use this machine's instruction set. Not portable to other CPUs.

An unoptimized build is commonly three to ten times slower than `-O2`. **Never benchmark at `-O0`**: you are measuring the compiler's debugging output, not your program.

> **Optimization exposes undefined behavior.** A program relying on signed overflow, strict-aliasing violations, or uninitialized reads can work at `-O0` and fail at `-O2`, because the optimizer reasons from the assumption that UB never happens. The bug was always there. Chapter 36 builds a program that demonstrates exactly this.

4 Reading the generated assembly

You do not need to write assembly to benefit from reading it. The question is usually just "did the compiler do what I hoped?"

```
gcc -O2 -S -masm=intel prog.c -o prog.s     # generate assembly
objdump -d --demangle prog.o | less          # disassemble an object file
```

Compiler Explorer — linked in the course resources — shows source and assembly side by side with the correspondence colour-coded, which is far easier for learning.

Four things worth checking:

LOOK FOR	MEANS
No `call` to your small function	It was inlined
`mov` of a constant where you wrote arithmetic	Computed at compile time
Packed instructions such as `addpd`, `addps`, `vpaddd`	The loop was vectorized
The loop body appearing several times	It was unrolled

This is how chapter 15 demonstrated that index and pointer traversal compile identically, and how you confirm that a `restrict` annotation actually changed anything in chapter 38.

5 Profiling

Programmers are reliably wrong about where time goes. Measure.

```
# perf — Linux, sampling, no recompilation, very low overhead
perf stat ./prog
perf record -g ./prog && perf report

# gprof — portable, needs instrumentation
gcc -pg -O2 prog.c -o prog && ./prog && gprof ./prog gmon.out | head -30

# callgrind — exact instruction counts, ~50x slower
valgrind --tool=callgrind ./prog && callgrind_annotate callgrind.out.*
```

`perf stat` is the first command to run. Beyond the elapsed time it reports *cache-misses* and *instructions per cycle* — and those two numbers usually identify whether you have a computation problem or a memory problem before you look at anything else.

Timing correctly

```
struct timespec t0, t1;
clock_gettime(CLOCK_MONOTONIC, &t0);
work();
clock_gettime(CLOCK_MONOTONIC, &t1);
double secs = (double)(t1.tv_sec - t0.tv_sec)
            + (double)(t1.tv_nsec - t0.tv_nsec) / 1e9;
```

`CLOCK_MONOTONIC`, not the wall clock, because the wall clock can step backwards. Run each measurement several times and take the minimum — the minimum is the run least disturbed by other activity, and is more stable than the mean.

Beware the compiler deleting your benchmark: if a result is unused, the whole computation can be removed. Print it, or accumulate into a `volatile`.

6 Worked example: the same loop, made three times faster

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

#define SIZE 1024
#define REPS 20
```

```
#define RECORDS 2000000

static double now(void)
{
    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC, &t);
    return (double)t.tv_sec + (double)t.tv_nsec / 1e9;
}

/* ---------- 1. loop order over a matrix ---------- */

static double sum_row_major(const double *m)
{
    double total = 0.0;
    for (int rep = 0; rep < REPS; rep++)
        for (size_t r = 0; r < SIZE; r++)
            for (size_t c = 0; c < SIZE; c++)
                total += m[r * SIZE + c];         /* sequential */
    return total;
}

static double sum_column_major(const double *m)
{
    double total = 0.0;
    for (int rep = 0; rep < REPS; rep++)
        for (size_t c = 0; c < SIZE; c++)
            for (size_t r = 0; r < SIZE; r++)
                total += m[r * SIZE + c];         /* strides 8 KB per step */
    return total;
}

/* ---------- 2. array of structures versus structure of arrays ---------- */

typedef struct {
    int    id;
    char   name[32];
    double score;
} Record;                       /* 48 bytes; we want 8 of them */
```

```
typedef struct {
    int    *id;
    char  (*name)[32];
    double *score;
} Columns;

static double sum_aos(const Record *r, size_t n)
{
    double total = 0.0;
    for (int rep = 0; rep < 20; rep++)
        for (size_t i = 0; i < n; i++)
            total += r[i].score;
    return total;
}

static double sum_soa(const Columns *c, size_t n)
{
    double total = 0.0;
    for (int rep = 0; rep < 20; rep++)
        for (size_t i = 0; i < n; i++)
            total += c->score[i];
    return total;
}

/* ---------- 3. what the optimizer can and cannot see ---------- */

__attribute__((noinline)) static long sum_to(long n)
{
    long total = 0;
    for (long i = 1; i <= n; i++) total += i;
    return total;
}

int main(void)
{
    puts("== 1. loop order ==");
    double *m = malloc((size_t)SIZE * SIZE * sizeof *m);
    if (m == NULL) return EXIT_FAILURE;
    for (size_t i = 0; i < (size_t)SIZE * SIZE; i++) m[i] = 1.0;
```

```
    printf("  matrix is %zu MB, cache lines hold %zu doubles\n",
           (size_t)SIZE * SIZE * sizeof *m / (1024 * 1024), 64 / sizeof(double));

    double t = now();
    double a = sum_row_major(m);
    double row_secs = now() - t;

    t = now();
    double b = sum_column_major(m);
    double col_secs = now() - t;

    printf("  row-major    : %.4f s  (sum %.0f)\n", row_secs, a);
    printf("  column-major : %.4f s  (sum %.0f)\n", col_secs, b);
    printf("  same result, same operation count, %.1fx difference\n",
           col_secs / row_secs);
    free(m);

    puts("\n== 2. array of structures versus structure of arrays ==");
    Record *recs = malloc(RECORDS * sizeof *recs);
    Columns cols = {
        .id    = malloc(RECORDS * sizeof *cols.id),
        .name  = malloc(RECORDS * sizeof *cols.name),
        .score = malloc(RECORDS * sizeof *cols.score)
    };
    if (recs == NULL || cols.id == NULL || cols.name == NULL || cols.score == NULL)
        return EXIT_FAILURE;

    for (size_t i = 0; i < RECORDS; i++) {
        recs[i].id = (int)i;  recs[i].score = 1.0;
        cols.id[i] = (int)i;  cols.score[i] = 1.0;
    }

    t = now();  double s1 = sum_aos(recs, RECORDS);  double aos_secs = now() - t;
    t = now();  double s2 = sum_soa(&cols, RECORDS); double soa_secs = now() - t;

    printf("  sizeof(Record) = %zu, so a 64-byte line holds %.1f scores\n",
           sizeof(Record), 64.0 / (double)sizeof(Record));
    printf("  array of structures : %.4f s  (sum %.0f)\n", aos_secs, s1);
    printf("  structure of arrays : %.4f s  (sum %.0f)\n", soa_secs, s2);
```

```
    printf("  %.1fx, purely from how much of each line is useful\n",
           aos_secs / soa_secs);

    free(recs); free(cols.id); free(cols.name); free(cols.score);

    puts("\n== 3. what the optimizer does ==");
    t = now();
    long total = sum_to(100000000L);
    double loop_secs = now() - t;
    printf("  sum 1..100000000 = %ld in %.4f s\n", total, loop_secs);
    puts("  compile at -O2 and inspect the assembly:");
    puts("    gcc -O2 -S perf.c -o perf.s && grep -A20 'sum_to:' perf.s");
    puts("  the loop may have become a closed-form multiplication");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -O2 -o perf perf.c
./perf
```

Typical results

COMPARISON	RATIO	CAUSE
Column-major versus row-major	4–10×	One cache line used per access instead of eight
Array of structures versus columns	2–4×	48-byte records; 40 of every 48 bytes wasted
`-O0` versus `-O2`	3–10×	No inlining, no vectorization, everything spilled

Confirm the diagnosis rather than assuming it:

```
perf stat -e cache-misses,cache-references,instructions,cycles ./perf
```

The column-major phase shows a cache-miss rate an order of magnitude higher than the row-major phase, at essentially the same instruction count. That is the entire explanation, visible as two numbers.

Look at the assembly

```
gcc -O2 -S perf.c -o perf.s
sed -n '/^sum_to:/,/ret/p' perf.s
```

Depending on the version, GCC may recognize the summation and emit the closed form `n(n+1)/2` — replacing a hundred million iterations with three instructions. Compare against `-O0`:

```
gcc -O0 -S perf.c -o perf0.s
sed -n '/^sum_to:/,/ret/p' perf0.s      # a literal loop, everything in memory
```

Then time both builds. A benchmark run at `-O0` measures none of this.

Profile before optimizing

```
perf record -g ./perf
perf report --stdio | head -20
```

The report ranks functions by the share of samples they consumed. Optimize the top entry; everything below it is noise. Amdahl's law is the arithmetic behind that advice: making a function that takes 5% of the runtime twice as fast improves the program by 2.5%, while the same effort on a function taking 60% is worth twelve times more.

When to stop

Four questions, in order. Is it actually too slow — measured against a requirement, not a feeling? Does the profiler agree with your guess about where the time goes? Is there a better algorithm, which usually beats any amount of micro-optimization? And is the speedup worth the loss of clarity, given that the code must be maintained for years?

Most C is fast enough when it is straightforward and compiled at `-O2` with sensible data layout. The layout is the part worth thinking about in advance, because it is the part that is hard to change later.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Benchmarking at `-O0`	Measures the debug build	Benchmark at `-O2`.
Optimizing without profiling	Effort spent on 2% of the runtime	`perf record` first.
Column-first matrix traversal	Several times slower for the same work	Rows outermost.
Discarding the benchmark's result	The compiler deletes the loop	Print it or use `volatile`.
Timing with the wall clock	Clock adjustments corrupt the measurement	`CLOCK_MONOTONIC`.
One measurement	Noise indistinguishable from signal	Repeat; take the minimum.
Micro-optimizing before fixing the algorithm	Constant-factor gains on a quadratic loop	Algorithm first, layout second, instructions last.
`-march=native` on shipped binaries	Crashes on older CPUs	Only for machines you control.

8 Check yourself

Why can two loops with identical complexity differ by a factor of ten?

Because complexity counts operations and ignores where the data is. Memory is fetched in 64-byte cache lines, so a sequential loop gets sixteen integers per fetch while a strided or pointer-chasing loop may use one. A main-memory access costs around two hundred cycles, so the number of misses dominates the number of instructions.

Why is benchmarking at `-O0` meaningless?

Because an unoptimized build keeps every variable in memory, inlines nothing, and vectorizes nothing — it is typically three to ten times slower than the release build and has different bottlenecks. You would be measuring the compiler's debugging mode, and conclusions drawn from it need not hold for the code you ship.

When is a structure of arrays worth the loss of readability?

When a hot loop reads one or two fields from many records and the record is large, so most of each cache line is wasted. Splitting the fields into parallel arrays makes the loop sequential over exactly the data it needs. It is a profiler-driven transformation, not a default — it complicates every other part of the code.

Why can a program work at `-O0` and fail at `-O2`?

Because the optimizer is entitled to assume the program contains no undefined behavior, and to reason from that assumption. Code relying on signed overflow, an aliasing violation, or an uninitialized read can survive a literal translation and break once the compiler propagates a conclusion the standard permits. The defect was present at both levels; only one exposed it.

What should you look at first when a program is too slow?

Whether it is too slow against an actual requirement, and then `perf stat` — elapsed time, cache misses, and instructions per cycle together say whether the problem is computation or memory. After that, `perf record` to find which function dominates. Choosing a better algorithm beats micro-optimization, and fixing data layout beats rewriting instructions.

9 Where this leads

Chapter 35 turns from making code fast to keeping it correct: debugging with GDB, unit tests and coverage, static analysis, and a continuous integration configuration that runs the sanitizers on every change. The warning profile assembled in chapter 29 becomes an enforced gate there.

ADVANCED · CHAPTER 35

Debugging, Testing, and Analysis

Every tool in this session exists because C will not tell you what is wrong on its own. Used together — warnings, a debugger, static analysis, tests, sanitizers, and a gate that runs them all — they turn C from a language where bugs hide into one where they are found on the day they are written.

BY THE END OF THIS CHAPTER YOU CAN

- Drive GDB: breakpoints, stepping, inspection, watchpoints, and backtraces.
- Read a core dump from a crash that happened without you.
- Write unit tests in C and measure which lines they actually exercise.
- Reduce a failing input to the smallest case that still fails.
- Configure a build that fails on a warning, a test failure, or a sanitizer report.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The order to apply tools

STAGE	TOOL	CATCHES
Typing	Compiler warnings	Type errors, obvious mistakes — free
Before running	Static analysis	Null dereferences, leaks on some paths
While running	Sanitizers	Memory errors, undefined behavior
Repeatedly	Unit tests	Logic errors; regressions
When something fails	GDB	Everything else
Continuously	CI	All of the above, on every change

Cheapest first. A warning costs nothing and is delivered instantly; a debugging session costs an hour. Most of what a beginner debugs with `printf` would have been a warning under `-Wall -Wextra`.

2 GDB

```
gcc -std=c17 -Wall -Wextra -g -O0 prog.c -o prog     # -g, and -O0 for sanity
gdb ./prog
```

COMMAND	DOES
`run` / `r`	Start, optionally with arguments
`break f` / `b f`	Break at a function, a line, or `file.c:42`
`break f if x > 100`	Conditional breakpoint
`next` / `n`	One line, stepping over calls
`step` / `s`	One line, stepping into calls
`finish`	Run until the current function returns
`continue` / `c`	Run to the next breakpoint
`print x` / `p *p`	Evaluate an expression
`p arr[3]@5`	Print five elements starting at index 3
`backtrace` / `bt`	The call stack
`frame 2`	Move to a caller's frame
`watch x`	Stop when `x` changes
`info locals`	Every local in this frame
`layout src`	Split view with source

The watchpoint

This is the command worth learning specifically, because it solves a problem nothing else does: *who changed this variable?*

```
(gdb) watch total
Hardware watchpoint 2: total
(gdb) continue
Hardware watchpoint 2: total
Old value = 100
New value = 0
update_totals (…) at report.c:88
```

The debugger stops at the exact instruction that modified it and shows you the stack. For a variable being corrupted by unrelated code — chapter 12's out-of-range write, for instance — this is often the only practical approach.

Core dumps

When a program crashes on a machine you were not watching, the kernel can write its memory image to a file:

```
ulimit -c unlimited        # enable core dumps for this shell
./prog                     # crashes
gdb ./prog core            # examine the corpse
(gdb) bt
(gdb) info locals
```

The backtrace shows exactly where it died and with what values, without reproducing anything. On systemd systems the file is managed by `coredumpctl`.

3 Static analysis

Analysis without running the program, so it reaches paths your tests never take.

```
gcc -fanalyzer -Wall -Wextra -c prog.c      # GCC 10+, built in
clang --analyze prog.c
cppcheck --enable=all --std=c17 prog.c
scan-build make                              # clang, whole project
```

`-fanalyzer` is the easiest win: it ships with GCC, needs no extra tool, and finds double frees, use-after-free, null dereferences, and leaks along specific paths — reporting the path it followed, which is what makes the report actionable.

Expect false positives. A static analyser reasons about all possible paths, including ones your invariants exclude but the code does not state. Triage them, suppress the ones you have verified, and keep the report at zero so that a new entry means something.

4 Unit tests

C has no test framework in the standard library. Frameworks exist — Unity, Check, Criterion, µnit — but a usable harness is about twenty lines, and writing it once makes clear what the frameworks add.

```
static int tests_run = 0, tests_failed = 0;

#define CHECK(cond) do {                                        \
    tests_run++;                                                \
    if (!(cond)) {                                              \
        tests_failed++;                                         \
        fprintf(stderr, "FAIL %s:%d: %s\n",                     \
                __FILE__, __LINE__, #cond);                     \
    }                                                           \
} while (0)
```

Two details in that macro matter. `#cond` stringifies the expression so the report shows what was tested — chapter 28's stringification, earning its place. And `do { … } while (0)` makes the macro behave as a single statement, so `if (x) CHECK(y); else …` parses correctly; without it, the semicolon after the closing brace breaks the `else`.

What to test

CASE	WHY
The empty input	Zero elements is where most off-by-ones live
One element	Loops that assume at least two
The boundaries	First, last, and one past — chapter 10's checklist
Failure paths	Allocation failure, malformed input — untested by default
Every bug you fix	A regression test stops it returning

Coverage

```
gcc --coverage -O0 -o tests tests.c lib.c
./tests
gcov lib.c
# or, readable:
lcov --capture --directory . --output-file cov.info
genhtml cov.info --output-directory report
```

Coverage tells you which lines ran, which is useful precisely because it exposes the error paths nobody tested. It does not tell you the tests are good — 100% coverage with no assertions

proves nothing. Treat a low number as a definite problem and a high number as no evidence either way.

5 Reducing a failure

A crash on a 50 MB input is nearly impossible to reason about. A crash on eleven bytes is nearly impossible to misunderstand. Reduction is a mechanical procedure:

1. Confirm it reproduces reliably. An intermittent failure is usually uninitialized memory or a race; fix the reproducibility first.
2. Halve the input. Does it still fail? Keep the failing half; otherwise keep the other.
3. Repeat until removing anything makes it pass.
4. Remove program options and features the same way.

`creduce` and `halfempty` automate this given a script that reports whether the input still fails. Chapter 48 produces such inputs by the thousand from a fuzzer, and reduction is what makes them usable.

6 Continuous integration

Every tool above is worthless if it is only run when someone remembers. CI runs them on every change and fails the build when they complain.

```
name: build and test
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: strict build
        run: make clean && make CFLAGS="-std=c17 -Wall -Wextra -Werror -O2"

      - name: sanitizers
        run: make clean && make test CFLAGS="-std=c17 -Wall -Wextra -g -O1 -
fsanitize=address,undefined"

      - name: static analysis
```

```
        run: make clean && make CFLAGS="-std=c17 -Wall -Wextra -fanalyzer"

      - name: valgrind
        run: |
          sudo apt-get install -y valgrind
          make clean && make
          valgrind --error-exitcode=1 --leak-check=full ./tests

      - name: coverage
        run: make clean && make CFLAGS="--coverage -O0" && ./tests && gcov *.c
```

Two flags make this work as a gate rather than a report. `-Werror` turns a warning into a failed build. `--error-exitcode=1` makes Valgrind's findings fail the step — without it, Valgrind prints its report and exits successfully, and nobody reads it.

Sanitizers also need `ASAN_OPTIONS=detect_leaks=1` on platforms where leak detection is off by default, and they set a non-zero exit status on a finding, so a test target that runs the instrumented binary fails automatically.

7 Worked example: testing the chapter 33 map

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

/* ---------- a twenty-line test harness ---------- */

static int tests_run = 0;
static int tests_failed = 0;
static const char *current_suite = "";

#define SUITE(name) do { current_suite = (name); \
    printf("\n-- %s\n", current_suite); } while (0)

#define CHECK(cond) do {                                        \
    tests_run++;                                                \
    if (cond) {                                                 \
        printf("  ok   %s\n", #cond);                           \
```

```
    } else {                                                        \
        tests_failed++;                                             \
        printf("  FAIL %s:%d: %s\n", __FILE__, __LINE__, #cond);    \
    }                                                               \
} while (0)

#define CHECK_EQ_INT(actual, expected) do {                         \
    tests_run++;                                                    \
    long a_ = (long)(actual), e_ = (long)(expected);                \
    if (a_ == e_) {                                                 \
        printf("  ok   %s == %ld\n", #actual, e_);                  \
    } else {                                                        \
        tests_failed++;                                             \
        printf("  FAIL %s:%d: %s was %ld, expected %ld\n",          \
               __FILE__, __LINE__, #actual, a_, e_);                \
    }                                                               \
} while (0)

/* ---------- the unit under test: a small string map ---------- */

typedef struct Entry {
    char         *key;
    int           value;
    struct Entry *next;
} Entry;

typedef struct {
    Entry **buckets;
    size_t  bucket_count;
    size_t  count;
} Map;

static size_t hash_string(const char *s)
{
    size_t h = 1469598103934665603u;
    for (; *s != '\0'; s++) { h ^= (unsigned char)*s; h *= 1099511628211u; }
    return h;
}
```

```
static Map *map_create(size_t buckets)
{
    if (buckets == 0) buckets = 8;
    Map *m = malloc(sizeof *m);
    if (m == NULL) return NULL;
    m->buckets = calloc(buckets, sizeof *m->buckets);
    if (m->buckets == NULL) { free(m); return NULL; }
    m->bucket_count = buckets;
    m->count = 0;
    return m;
}

static void map_destroy(Map *m)
{
    if (m == NULL) return;
    for (size_t b = 0; b < m->bucket_count; b++) {
        Entry *e = m->buckets[b];
        while (e != NULL) { Entry *n = e->next; free(e->key); free(e); e = n; }
    }
    free(m->buckets);
    free(m);
}

static bool map_put(Map *m, const char *key, int value)
{
    size_t i = hash_string(key) % m->bucket_count;
    for (Entry *e = m->buckets[i]; e != NULL; e = e->next) {
        if (strcmp(e->key, key) == 0) { e->value = value; return true; }
    }
    Entry *e = malloc(sizeof *e);
    if (e == NULL) return false;
    size_t bytes = strlen(key) + 1;
    e->key = malloc(bytes);
    if (e->key == NULL) { free(e); return false; }
    memcpy(e->key, key, bytes);
    e->value = value;
    e->next = m->buckets[i];
    m->buckets[i] = e;
    m->count++;
```

```
    return true;
}

static bool map_get(const Map *m, const char *key, int *out)
{
    size_t i = hash_string(key) % m->bucket_count;
    for (Entry *e = m->buckets[i]; e != NULL; e = e->next) {
        if (strcmp(e->key, key) == 0) { *out = e->value; return true; }
    }
    return false;
}

static bool map_remove(Map *m, const char *key)
{
    size_t i = hash_string(key) % m->bucket_count;
    for (Entry **link = &m->buckets[i]; *link != NULL; link = &(*link)->next) {
        if (strcmp((*link)->key, key) == 0) {
            Entry *dead = *link;
            *link = dead->next;
            free(dead->key); free(dead);
            m->count--;
            return true;
        }
    }
    return false;
}

/* ---------- the tests ---------- */

static void test_empty(void)
{
    SUITE("an empty map");
    Map *m = map_create(8);
    int v = -1;
    CHECK(m != NULL);
    CHECK_EQ_INT(m->count, 0);
    CHECK(!map_get(m, "absent", &v));
    CHECK_EQ_INT(v, -1);                  /* untouched on failure */
    CHECK(!map_remove(m, "absent"));
```

```
    map_destroy(m);
}

static void test_single(void)
{
    SUITE("one entry");
    Map *m = map_create(8);
    int v = 0;
    CHECK(map_put(m, "key", 42));
    CHECK_EQ_INT(m->count, 1);
    CHECK(map_get(m, "key", &v));
    CHECK_EQ_INT(v, 42);
    CHECK(!map_get(m, "Key", &v));        /* case sensitive */
    map_destroy(m);
}

static void test_replace(void)
{
    SUITE("replacing a value");
    Map *m = map_create(8);
    int v = 0;
    map_put(m, "k", 1);
    CHECK(map_put(m, "k", 2));
    CHECK_EQ_INT(m->count, 1);            /* not 2 */
    map_get(m, "k", &v);
    CHECK_EQ_INT(v, 2);
    map_destroy(m);
}

static void test_collisions(void)
{
    SUITE("forced collisions");
    Map *m = map_create(1);               /* one bucket: everything collides */
    int v = 0;
    for (int i = 0; i < 20; i++) {
        char key[16];
        snprintf(key, sizeof key, "k%d", i);
        map_put(m, key, i);
    }
```

```
    CHECK_EQ_INT(m->count, 20);
    CHECK(map_get(m, "k0", &v));   CHECK_EQ_INT(v, 0);
    CHECK(map_get(m, "k19", &v));  CHECK_EQ_INT(v, 19);
    CHECK(map_remove(m, "k10"));
    CHECK(!map_get(m, "k10", &v));
    CHECK_EQ_INT(m->count, 19);
    map_destroy(m);
}

static void test_edge_keys(void)
{
    SUITE("awkward keys");
    Map *m = map_create(8);
    int v = 0;
    CHECK(map_put(m, "", 1));                   /* empty key */
    CHECK(map_get(m, "", &v));   CHECK_EQ_INT(v, 1);

    char long_key[512];
    memset(long_key, 'x', sizeof long_key - 1);
    long_key[sizeof long_key - 1] = '\0';
    CHECK(map_put(m, long_key, 2));
    CHECK(map_get(m, long_key, &v)); CHECK_EQ_INT(v, 2);

    CHECK(map_put(m, "türkçe", 3));             /* multibyte UTF-8 */
    CHECK(map_get(m, "türkçe", &v)); CHECK_EQ_INT(v, 3);
    map_destroy(m);
}

static void test_remove_order(void)
{
    SUITE("removal from every chain position");
    const char *keys[] = { "a", "b", "c" };
    for (int victim = 0; victim < 3; victim++) {
        Map *m = map_create(1);                 /* all in one chain */
        for (int i = 0; i < 3; i++) map_put(m, keys[i], i);
        CHECK(map_remove(m, keys[victim]));
        CHECK_EQ_INT(m->count, 2);
        int v;
        for (int i = 0; i < 3; i++) {
```

```
            if (i == victim) CHECK(!map_get(m, keys[i], &v));
            else             CHECK(map_get(m, keys[i], &v));
        }
        map_destroy(m);
    }
}

int main(void)
{
    puts("running tests");
    test_empty();
    test_single();
    test_replace();
    test_collisions();
    test_edge_keys();
    test_remove_order();

    printf("\n%d checks, %d failed\n", tests_run, tests_failed);
    return tests_failed == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

```
gcc -std=c17 -Wall -Wextra -g -O1 -fsanitize=address,undefined -o tests tests.c
./tests ; echo "exit status $?"
```

Why these tests and not others

The empty map first. Zero elements is where off-by-ones live, and it is the case a happy-path test never reaches.

One bucket, deliberately. `map_create(1)` forces every key into the same chain, so the chain-walking code — normally exercised by luck — is exercised on purpose. This is how you test a code path whose occurrence is otherwise probabilistic.

Removal from each chain position. Head, middle, and tail are three different situations in linked-list code. Testing only one is testing a third of the function.

Awkward keys. The empty string, a 511-byte key, and multibyte UTF-8. Each has broken a real hash table somewhere.

The exit status. `main` returns non-zero when anything failed, which is what makes the suite usable from `make test` and from CI. A test program that always exits 0 cannot gate anything.

Measure the coverage

```
gcc -std=c17 --coverage -O0 -o tests_cov tests.c
./tests_cov
gcov tests.c | head -5
less tests.c.gcov      # lines marked ##### were never executed
```

Look for `#####` on the allocation-failure branches inside `map_put`. No test reaches them, because `malloc` does not fail on demand. Those are the error paths chapter 27 warned are the least tested code in any program; reaching them needs a failure-injecting allocator or a fault-injection library.

Practise the debugger on a real bug

Introduce one, then find it without reading the diff:

```
/* in map_remove, forget to decrement: */
/* m->count--; */
```

```
./tests                      # FAIL: m->count was 3, expected 2
gdb ./tests
(gdb) break map_remove
(gdb) run
(gdb) finish
(gdb) print m->count
```

Then try the watchpoint approach instead, which finds it without knowing where to look:

```
(gdb) break test_remove_order
(gdb) run
(gdb) next 4
(gdb) watch m->count
(gdb) continue
```

The watchpoint reports every change to the field. When removal happens and no report appears, the missing decrement has identified itself.

Run the static analyser

```
gcc -fanalyzer -Wall -Wextra -c tests.c -o /dev/null
```

On this code it should be quiet. Delete the `free(e->key)` in `map_destroy` and it reports the leak, with the path it followed to reach the conclusion — before the program has been run once.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Debugging an optimized build	Variables "optimized out"; lines jump	Debug at `-O0 -g`.
Forgetting `-g`	No source lines or variable names	Always build development binaries with it.
Testing only the happy path	Error paths ship untested	Test empty, boundary, and failure cases.
A test suite that always exits 0	CI cannot detect a failure	Return non-zero when anything failed.
Treating coverage as a quality score	Tests with no assertions reach 100%	Use it to find untested paths only.
Valgrind without `--error-exitcode`	Reports findings and exits 0	Set it so the step fails.
Debugging a huge input	Too much state to reason about	Reduce it first.
Ignoring static-analysis output	Real findings buried in false positives	Triage to zero and keep it there.

9 Check yourself

Why debug at `-O0` rather than `-O2`?

Because optimization removes variables, merges and reorders lines, and inlines functions, so the debugger reports "optimized out" and the execution point jumps around. At `-O0` the machine code corresponds to the source line by line. Benchmarking is the reverse case — there `-O0` is meaningless.

What problem does a watchpoint solve that a breakpoint cannot?

Finding *who* modified a variable when you do not know where the write happens. A breakpoint requires knowing the location; a watchpoint stops at whatever instruction changes the value and shows the stack. For a variable corrupted by an out-of-range write elsewhere, it is often the only practical method.

Why force every key into one bucket in a hash-table test?

Because the chain-walking code is otherwise exercised only when collisions happen to occur, which is probabilistic and may not happen at all in a small test. Creating the table with one bucket makes collision handling deterministic, so removal from the head, middle, and tail of a chain can all be tested on purpose.

Why is 100% coverage not evidence that the tests are good?

Because coverage records which lines executed, not whether anything was checked. A suite that calls every function and asserts nothing reaches full coverage and detects no bugs. Low coverage is a definite problem — those paths are untested — but high coverage is no evidence in the other direction.

Why must a test program return a non-zero exit status on failure?

Because that is the only thing `make`, a shell script, and a CI system can see. A suite that prints "FAIL" and exits 0 is a report nobody reads; one that exits non-zero fails the build and blocks the change. It is the same exit-status discipline chapter 23 applied to command-line tools.

10 Where this leads

Chapter 36 addresses the class of bug that none of these tools reliably catches at `-O0`: undefined behavior, which can make a correct-looking program fail only once the optimizer believes it. The undefined-behavior sanitizer you have been running alongside AddressSanitizer becomes the main instrument there.

ADVANCED · CHAPTER 36

Undefined Behavior and Secure Coding

Undefined behavior is the concept that separates people who write C from people who understand it. It is not "unpredictable output" — it is the withdrawal of all requirements, and an optimizing compiler will reason from its absence in ways that delete your safety checks.

BY THE END OF THIS CHAPTER YOU CAN

- Define undefined behavior precisely and distinguish it from the two neighbouring categories.
- Explain how an optimizer turns undefined behavior into deleted code.
- Recognize a strict-aliasing violation and say what `restrict` promises.
- Name the vulnerability classes that follow from C's memory model.
- Write checks that cannot themselves be optimized away.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Three categories, often confused

CATEGORY	THE STANDARD SAYS	EXAMPLE
Unspecified	Several behaviors are allowed; no record of which	Order of evaluation of function arguments
Implementation-defined	One behavior, chosen and *documented* by the implementation	Whether plain `char` is signed; `sizeof(int)`
Undefined	*No requirements whatsoever*	Signed overflow, out-of-bounds access, null dereference

The first two are survivable: you can write portable code by not depending on unspecified choices, and you can look up implementation-defined ones. The third is different in kind. The standard imposes no requirement on a program that exhibits it — not on the operation, not on the surrounding code, not on anything that happened before.

The practical consequence is the part people miss. Undefined behavior does not mean "you get a wrong value here". It means **the compiler may assume it does not occur, and optimize the whole function on that assumption**.

2 How an optimizer exploits it

```
int check(int *p)
{
    int value = *p;             /* dereference: if p were null, UB */
    if (p == NULL) {            /* so the compiler concludes p != NULL */
        return -1;              /* and deletes this branch entirely */
    }
    return value;
}
```

The reasoning is valid. Dereferencing a null pointer is undefined, so a correct program never reaches this code with `p == NULL`; therefore the test is redundant; therefore it can be removed. Your null check is gone, and at `-O0` it was still there.

The same mechanism eliminates overflow checks:

```
/* A check that cannot work */
if (a + b < a) {                /* assumes signed overflow wraps */
    return ERROR_OVERFLOW;      /* deleted: signed overflow is UB,   */
}                               /* so a + b < a is assumed false     */

/* A check that works: test before the operation */
if (b > 0 && a > INT_MAX - b) {
    return ERROR_OVERFLOW;
}
if (b < 0 && a < INT_MIN - b) {
    return ERROR_OVERFLOW;
}
int sum = a + b;                /* now provably safe */
```

This is not hypothetical. A 2009 Linux kernel vulnerability arose from exactly this pattern: the compiler removed a null check that followed a dereference, leaving an exploitable hole in code whose source looked correct.

> **The rule that follows.** Never write a check that detects undefined behavior after the fact. Check the *preconditions* before performing the operation, so the operation is provably defined. Anything else may be optimized away precisely when it matters.

3 The common sources

UNDEFINED BEHAVIOR	FIRST MET IN
Signed integer overflow	Chapter 4
Array access out of bounds	Chapter 12
Dereferencing null, dangling, or uninitialized pointers	Chapters 13, 20
Reading an uninitialized variable	Chapter 5
Use after free, double free	Chapter 20
Shifting by the width of the type or more	Chapter 25
Two side effects on one object without a sequence point	Chapter 6
Modifying a string literal	Chapter 14
A `printf` specifier that does not match its argument	Chapter 8
Strict-aliasing violation	This chapter

There are over two hundred in the standard. These ten cover almost everything encountered in practice.

4 Strict aliasing

The *strict aliasing rule* says an object may only be accessed through an lvalue of a compatible type — with `char` as the universal exception. The compiler relies on it to know that a write through an `int *` cannot disturb a `float`, which allows it to keep values in registers across the store.

```
/* Violation: reading a float's bytes through an int pointer */
float f = 1.0f;
int bits = *(int *)&f;              /* undefined behavior */
```

It usually appears to work, which is why it survives in old code. It breaks when the optimizer reorders the load and the store because it has concluded they cannot refer to the same object.

Two correct ways to reinterpret bytes:

```
/* 1. memcpy — the standard idiom; optimizes to nothing */
int bits;
memcpy(&bits, &f, sizeof bits);

/* 2. a union — explicitly permitted in C (unlike C++) */
union { float f; int bits; } u = { .f = 1.0f };
int bits = u.bits;
```

The `memcpy` form looks wasteful and is not: every mainstream compiler recognizes a small fixed-size copy and emits a single register move. Confirm it with chapter 34's assembly technique.

`restrict`

`restrict` is a promise in the other direction: *within this scope, the object reached through this pointer is reached through no other*.

```
void add_arrays(int *restrict dst, const int *restrict a,
                const int *restrict b, size_t n)
{
    for (size_t i = 0; i < n; i++) {
        dst[i] = a[i] + b[i];
    }
}
```

Without it the compiler must assume `dst` might overlap `a`, so each store could change a later load — and it must reload on every iteration and cannot vectorize. With it, the loop can be vectorized.

The promise is unchecked. Pass overlapping pointers and the behavior is undefined, with no diagnostic. This is exactly the distinction between `memcpy` (whose parameters are `restrict`) and `memmove` (whose are not), which chapter 14 introduced from the other side.

5 From undefined behavior to vulnerability

CLASS	MECHANISM	DEFENCE
Buffer overflow	Writing past an array overwrites the return address or a function pointer	Bounded copies; `snprintf`; check lengths
Format string	`printf(user_input)` lets `%n` write to memory	`printf("%s", user_input)`
Integer overflow	A size computation wraps; a small buffer is allocated for a large copy	`calloc`; check before multiplying
Use after free	The attacker arranges for the freed block to be reallocated with chosen contents	`NULL` after `free`; clear ownership
Off-by-one	One byte past a buffer overwrites an adjacent length or pointer	Half-open ranges; sanitizers

The integer-overflow allocation bug

```
/* count comes from the network */
size_t bytes = count * sizeof(Record);      /* can wrap */
Record *r = malloc(bytes);                  /* far too small */
for (size_t i = 0; i < count; i++) {
    r[i] = read_record();                   /* writes way past the end */
}
```

With `sizeof(Record)` of 48 and a `count` near `SIZE_MAX / 48`, the multiplication wraps to a small number. The allocation succeeds, the loop writes gigabytes, and the attacker chose the data. Two fixes: use `calloc(count, sizeof(Record))`, which is required to detect the overflow, or check `count > SIZE_MAX / sizeof(Record)` first.

The format-string bug

```
printf(user_input);             /* catastrophic */
printf("%s", user_input);       /* correct */
```

If the input contains `%s`, `printf` reads an argument that was never passed. If it contains `%n`, `printf` *writes* to an address taken from the stack. This has been a remote-code-execution vector since the 1990s and `-Wformat-security` catches it.

6 Defences that work

MEASURE	CATCHES
`-fsanitize=undefined`	Overflow, bad shifts, misalignment, at runtime
`-fsanitize=address`	Overruns, use-after-free
`-D_FORTIFY_SOURCE=2 -O2`	Some overflows in library calls, cheaply
`-fstack-protector-strong`	Stack smashing, via a canary
`-Wformat-security`	Non-literal format strings
`-fno-strict-aliasing`	Disables the assumption — a workaround, not a fix
`-ftrapv`	Traps on signed overflow instead of assuming

And the practices: validate every input at the boundary, bound every copy, use `calloc` for untrusted counts, prefer `snprintf`, set pointers to `NULL` after freeing, and treat every compiler warning as a defect.

7 Worked example: correct at `-O0`, broken at `-O2`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <stdint.h>
#include <stdbool.h>

/* ---------- 1. the overflow check that disappears ---------- */

static bool add_checked_broken(int a, int b, int *out)
{
    int sum = a + b;               /* UB on overflow */
    if (sum < a) {                 /* assumes wrapping; may be deleted */
        return false;
    }
    *out = sum;
    return true;
}
```

```
static bool add_checked(int a, int b, int *out)
{
    if (b > 0 && a > INT_MAX - b) return false;   /* checked BEFORE */
    if (b < 0 && a < INT_MIN - b) return false;
    *out = a + b;
    return true;
}

/* ---------- 2. the null check that disappears ---------- */

static int deref_then_check(int *p)
{
    int value = *p;                /* if p were NULL this is UB ... */
    if (p == NULL) {               /* ... so the compiler may delete this */
        return -1;
    }
    return value;
}

static int check_then_deref(int *p)
{
    if (p == NULL) return -1;      /* the only order that works */
    return *p;
}

/* ---------- 3. strict aliasing ---------- */

static uint32_t bits_violating(float f)
{
    return *(uint32_t *)&f;        /* undefined behavior */
}

static uint32_t bits_memcpy(float f)
{
    uint32_t bits;
    memcpy(&bits, &f, sizeof bits);   /* correct, and free at -O2 */
    return bits;
}
```

```
static uint32_t bits_union(float f)
{
    union { float f; uint32_t u; } conv = { .f = f };
    return conv.u;                    /* correct in C */
}

/* ---------- 4. the allocation-size overflow ---------- */

typedef struct { int id; char pad[44]; } Record;   /* 48 bytes */

static Record *allocate_broken(size_t count)
{
    return malloc(count * sizeof(Record));         /* can wrap */
}

static Record *allocate_checked(size_t count)
{
    if (count > SIZE_MAX / sizeof(Record)) {
        return NULL;                               /* refuse */
    }
    return calloc(count, sizeof(Record));          /* checks as well */
}

/* ---------- 5. format strings ---------- */

static void print_broken(const char *user)
{
    printf(user);                  /* -Wformat-security warns */
    putchar('\n');
}

static void print_safe(const char *user)
{
    printf("%s\n", user);
}

int main(void)
{
    printf("built at optimization level: ");
```

```
#ifdef __OPTIMIZE__
    puts("optimized");
#else
    puts("-O0");
#endif

    puts("\n== 1. signed overflow ==");
    int result = 0;
    printf("  broken check, INT_MAX + 1 : %s\n",
           add_checked_broken(INT_MAX, 1, &result) ? "ACCEPTED (wrong)"
                                                   : "rejected");
    printf("  correct check, INT_MAX + 1: %s\n",
           add_checked(INT_MAX, 1, &result) ? "ACCEPTED (wrong)" : "rejected");
    printf("  correct check, 100 + 200  : %s (%d)\n",
           add_checked(100, 200, &result) ? "accepted" : "rejected", result);
    puts("  compile at -O2 and compare: the broken check may vanish");

    puts("\n== 2. null checks ==");
    int value = 42;
    printf("  check_then_deref(&value) = %d\n", check_then_deref(&value));
    printf("  check_then_deref(NULL)   = %d\n", check_then_deref(NULL));
    puts("  deref_then_check(NULL) is NOT called here: at -O2 the test");
    puts("  may be gone and it would simply crash");

    puts("\n== 3. strict aliasing ==");
    float f = 1.0f;
    printf("  memcpy : 0x%08X\n", bits_memcpy(f));
    printf("  union  : 0x%08X\n", bits_union(f));
    printf("  cast   : 0x%08X  <-- undefined behavior, works by luck\n",
           bits_violating(f));
    puts("  all three agree today; only two are guaranteed to");

    puts("\n== 4. allocation size overflow ==");
    size_t huge = SIZE_MAX / 48 + 2;
    printf("  requesting %zu records of %zu bytes\n", huge, sizeof(Record));
    printf("  count * sizeof wraps to %zu bytes\n", huge * sizeof(Record));
    Record *bad = allocate_broken(huge);
    printf("  malloc(wrapped) : %s  <-- a tiny buffer for a huge count\n",
           bad ? "SUCCEEDED" : "failed");
```

```
    free(bad);
    Record *good = allocate_checked(huge);
    printf("  checked version : %s\n", good ? "succeeded" : "refused");
    free(good);

    puts("\n== 5. format strings ==");
    print_safe("a normal message");
    print_safe("%s %s %s %n");     /* harmless: it is just data */
    puts("  the same string passed to printf() directly would read");
    puts("  arguments that were never pushed, and %n would WRITE");

    puts("\n== 6. shifting too far ==");
    unsigned shift_by = 32;
    uint32_t one = 1;
    printf("  1u << 32 on a 32-bit type is undefined\n");
    if (shift_by < 32) {           /* the guard that makes it defined */
        printf("  guarded: %u\n", one << shift_by);
    } else {
        puts("  guarded: refused, as it must be");
    }

    return EXIT_SUCCESS;
}
```

Run it at both levels and compare

```
gcc -std=c17 -Wall -Wextra -Wformat-security -O0 -o ub0 ub.c
gcc -std=c17 -Wall -Wextra -Wformat-security -O2 -o ub2 ub.c
./ub0 > out0.txt
./ub2 > out2.txt
diff out0.txt out2.txt
```

On many GCC versions the "broken check" line differs between the two builds: at `-O0` the addition wraps and the test happens to catch it; at `-O2` the compiler has concluded the test can never be true and removed it. Same source, same inputs, different behavior — which is the definition of a program you cannot reason about.

Let the sanitizer name it

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=undefined -o ubsan ub.c
./ubsan
```

```
ub.c:14:15: runtime error: signed integer overflow: 2147483647 + 1
            cannot be represented in type 'int'
```

File, line, operation, and values. UBSan is the only practical way to find this class of bug, because nothing else reports it — the program does not crash, it just becomes wrong.

See `memcpy` cost nothing

```
gcc -O2 -S ub.c -o ub.s
sed -n '/bits_memcpy:/,/ret/p' ub.s
sed -n '/bits_violating:/,/ret/p' ub.s
```

Both functions typically compile to the same one or two instructions (make them non-`static` or `noinline` first: at `-O2` a `static` function called once is inlined into `main`, and the `sed` finds no label). The `memcpy` version is correct and free; the cast version is undefined and no faster. There is no trade-off to make.

Watch the format-string warning

```
ub.c:118:5: warning: format not a string literal and no format
            arguments [-Wformat-security]
```

`-Wformat-security` is not in `-Wall` or `-Wextra`. Add it to the warning profile from chapter 29 — it catches a vulnerability class for one flag.

Confirm the allocation wrap

The output prints the requested count, the size of a record, and the wrapped product — 80 bytes on a 64-bit system, far smaller than the count. `malloc` succeeds because the request is tiny; the loop that follows would write past the end of a handful of bytes. `calloc` is required by the standard to detect this and return `NULL`, which is why it is the right allocator for any count that came from outside your program.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`if (a + b < a)` to detect overflow	Deleted by the optimizer	Check against `INT_MAX - b` first.
Null check after a dereference	Deleted by the optimizer	Check before.
`*(int *)&float_value`	Strict-aliasing violation	`memcpy` or a union.
`malloc(count * size)` with untrusted `count`	Wraps; undersized buffer	`calloc`, or check the division bound.
`printf(user_string)`	Information disclosure or memory write	`printf("%s", …)`.
`restrict` on pointers that may overlap	Undefined; silently wrong output	Only promise what you can guarantee.
"It works, so it must be correct"	Breaks on the next compiler version	Correctness is what the standard guarantees.
`-fno-strict-aliasing` as a fix	Hides the bug; costs performance	Fix the aliasing violation.

9 Check yourself

What is the difference between undefined and implementation-defined behavior?

Implementation-defined behavior has exactly one outcome, chosen by the implementation and documented — whether `char` is signed, for instance. You can look it up and rely on it for that platform. Undefined behavior carries no requirements at all: not on the operation, not on surrounding code, not on anything that already happened. The compiler may assume it never occurs.

Why does `if (a + b < a)` fail to detect signed overflow?

Because the addition itself is undefined when it overflows, so the compiler is entitled to assume it does not — and under that assumption `a + b < a` is false whenever `b` is non-negative, so the branch is dead code and gets removed. The check must test the operands against `INT_MAX` before performing the addition.

Why is `*(int *)&some_float` wrong even though it works?

It violates strict aliasing: an object may only be accessed through a compatible type, with `char` as the exception. The compiler uses that rule to assume an `int` store cannot affect a `float`, and may reorder loads and stores accordingly. Use `memcpy`, which is correct and compiles to the same instruction, or a union, which C explicitly permits.

What does `restrict` promise, and who checks it?

That within the scope, the object accessed through that pointer is not accessed through any other pointer — which lets the compiler keep values in registers and vectorize loops. Nobody checks it. Passing overlapping pointers is undefined behavior with no diagnostic, which is precisely the difference between `memcpy` and `memmove`.

Why is `calloc` the right allocator for a count that came from outside the program?

Because it takes the count and element size separately and is required to detect overflow in their product, returning `NULL` instead of allocating a wrapped, far-too-small block. `malloc(count * size)` computes the product first, and if it wraps the allocation succeeds at the wrong size — the classic route from an integer overflow to a heap buffer overflow.

10 Where this leads

Chapter 37 covers the neighbouring category: behavior that is defined but differs between platforms — type widths, byte order, alignment, character encoding. That is what stands between code that runs here and code that runs everywhere, and it is the prerequisite for the portable binary format the embedded chapters depend on.

ADVANCED · CHAPTER 37

Portability and the Machine Model

Chapter 36 covered behavior with no requirements. This chapter covers behavior that is perfectly defined and simply different somewhere else — type widths, byte order, alignment, character encoding. It is the difference between code that runs here and code that runs everywhere.

BY THE END OF THIS CHAPTER YOU CAN

- Use `<stdint.h>` and say when a fixed width is required rather than preferred.
- Detect and convert byte order, and explain why network protocols specify it.
- Reason about alignment and use `_Alignas` and `_Alignof`.
- Handle UTF-8 correctly without a library.
- Write a binary format that survives moving between machines.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 What the standard does not fix

PROPERTY	GUARANTEED	VARIES
`sizeof(char)`	Exactly 1	Never
`CHAR_BIT`	At least 8	8 everywhere you will meet
`sizeof(int)`	At least 2	2 on small embedded, 4 elsewhere
`sizeof(long)`	At least 4	**4 on Windows, 8 on Linux and macOS**
`sizeof(void *)`	Nothing	4 or 8; unrelated to `sizeof(int)`
Signedness of plain `char`	Nothing	Signed on x86, unsigned on ARM
Byte order	Nothing	Little-endian on x86 and ARM; big-endian in network protocols
Structure padding	Nothing	Depends on compiler and target

The `long` row is the one that catches working programmers. The three mainstream 64-bit data models disagree:

MODEL	INT	LONG	POINTER	USED BY
LP64	4	8	8	Linux, macOS, the BSDs
LLP64	4	**4**	8	64-bit Windows
ILP32	4	4	4	32-bit systems

Code that assumes `long` holds a pointer or 64 bits works on Linux and breaks on Windows. Use `int64_t` when you mean 64 bits, and `intptr_t` when you mean "wide enough for a pointer".

2 Fixed-width types

```
#include <stdint.h>
#include <inttypes.h>

int8_t  int16_t  int32_t  int64_t        /* exactly this many bits */
uint8_t uint16_t uint32_t uint64_t
int_least16_t                           /* at least 16, smallest available */
int_fast32_t                            /* at least 32, fastest available */
intptr_t  uintptr_t                     /* can hold a pointer */
size_t    ptrdiff_t                     /* sizes and differences */
```

Printing them needs the macros from `<inttypes.h>`, because the underlying type differs by platform:

```
printf("%" PRId64 "\n", value);      /* int64_t  */
printf("%" PRIu32 "\n", count);      /* uint32_t */
printf("%zu\n", size);               /* size_t has its own specifier */
```

The syntax is string-literal concatenation: `PRId64` expands to `"ld"` or `"lld"` as appropriate, and the adjacent literals join at compile time.

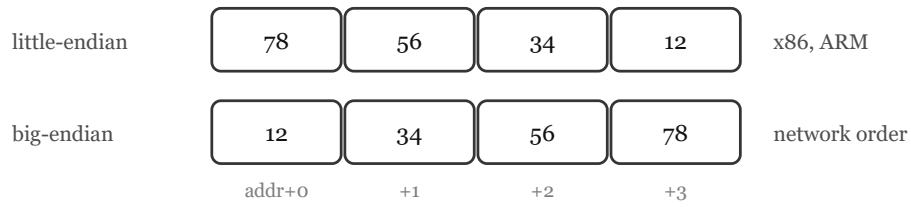
When to use which

USE	WHEN
`int`	Ordinary arithmetic, loop counters, small values. Still the default.
`size_t`	Sizes, counts, array indices
`int32_t`, `uint8_t`	File formats, protocols, hardware registers — anywhere the width is part of the contract
`int_fast32_t`	Hot loops where you want at least 32 bits and speed
`intptr_t`	Storing a pointer in an integer (rarely needed)

Do not replace every `int` with `int32_t`. Fixed widths are for data whose layout is specified; for ordinary arithmetic, `int` is what the machine prefers and what everyone expects.

3 Byte order

A multi-byte integer can be stored with its least significant byte first (*little-endian*) or last (*big-endian*). The value `0x12345678` in memory:



Same value, same four bytes, opposite order. Chapter 26's raw-struct file format depends on which machine wrote it.

This matters the moment bytes leave your process — a file, a socket, shared memory between different architectures.

```
#include <arpa/inet.h>        /* POSIX */

uint32_t on_wire = htonl(host_value);     /* host TO network long  */
uint32_t local   = ntohl(on_wire);        /* network TO host long  */
uint16_t port    = htons(1234);           /* short version         */
```

"Network byte order" is big-endian, fixed by convention since the earliest internet protocols. On a big-endian machine these functions do nothing; on a little-endian one they swap. Calling them unconditionally is correct everywhere, which is the point.

Without POSIX, do it explicitly — this version does not care what the host order is:

```
static void store_be32(uint8_t *out, uint32_t v)
{
    out[0] = (uint8_t)(v >> 24);
    out[1] = (uint8_t)(v >> 16);
    out[2] = (uint8_t)(v >>  8);
    out[3] = (uint8_t)(v);
}

static uint32_t load_be32(const uint8_t *in)
{
    return ((uint32_t)in[0] << 24) | ((uint32_t)in[1] << 16)
         | ((uint32_t)in[2] <<  8) |  (uint32_t)in[3];
}
```

Shifting works on *values*, not on the bytes in memory, so the result is identical on either architecture. This is the technique to use for a file format.

4 Alignment

Most architectures require an object's address to be a multiple of its size — an `int` at a multiple of 4, a `double` at a multiple of 8. This is where chapter 22's structure padding came from.

```
printf("%zu\n", _Alignof(double));        /* usually 8 */
_Alignas(64) char cache_line[64];         /* aligned to a cache line */
```

An unaligned access is a crash on some architectures, and merely slow on x86. Reading a `uint32_t` straight out of a byte buffer is the usual way to cause one:

```
uint32_t bad  = *(uint32_t *)(buffer + 1);   /* alignment + aliasing UB */

uint32_t good;
memcpy(&good, buffer + 1, sizeof good);      /* correct everywhere */
```

`memcpy` solves the alignment problem and the strict-aliasing problem from chapter 36 at the same time, and compiles to the same instruction on a platform that permits the unaligned load. It is the standard answer to "how do I read an integer out of a buffer".

5 Text beyond ASCII

Chapter 4 covered ASCII's 128 codes. Turkish needs ş, ğ, ı, ö, ü, ç; other languages need far more. Two approaches exist and one of them has won.

Wide characters — `wchar_t` and `<wchar.h>` — use a fixed-width type per character. The trouble is that `wchar_t` is 32 bits on Linux and 16 on Windows, so it is not portable, and 16 bits is not enough for all of Unicode anyway.

UTF-8 encodes each code point in one to four bytes. It has three properties that made it universal:

- ASCII is unchanged — a byte below 0x80 means exactly what it always did.
- No byte of a multi-byte sequence can be mistaken for ASCII, so `strchr(s, '/')` still works correctly.
- It is byte-order independent, so no conversion is needed when it travels.

The consequence for C is that `char *` holds UTF-8 perfectly well, and most string code needs no change. What changes is counting:

```
const char *s = "türkçe";
strlen(s)          /* 8 BYTES, not 6 characters */
```

To count characters, count the bytes that are not continuation bytes. A continuation byte has its top two bits set to `10`:

```
static size_t utf8_length(const char *s)
{
    size_t chars = 0;
    for (; *s != '\0'; s++) {
        if (((unsigned char)*s & 0xC0) != 0x80) {
            chars++;                        /* not a continuation byte */
        }
    }
    return chars;
}
```

The practical rule: **treat text as bytes wherever you can** — copying, comparing, searching, and writing all work unchanged. Decode only when you must count characters, truncate for display, or change case. And never truncate a UTF-8 string at an arbitrary byte, or you will split a character in half.

6 Worked example: a portable record format

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
#include <stdbool.h>
#include <stddef.h>

/* ---------- what this machine looks like ---------- */

static bool is_little_endian(void)
{
    uint32_t probe = 1;
    unsigned char first;
    memcpy(&first, &probe, 1);
    return first == 1;
}

/* ---------- explicit big-endian accessors ---------- */

static void store_be16(uint8_t *out, uint16_t v)
{
    out[0] = (uint8_t)(v >> 8);
    out[1] = (uint8_t)(v);
}
static void store_be32(uint8_t *out, uint32_t v)
{
    out[0] = (uint8_t)(v >> 24); out[1] = (uint8_t)(v >> 16);
    out[2] = (uint8_t)(v >>  8); out[3] = (uint8_t)(v);
}
static void store_be64(uint8_t *out, uint64_t v)
{
```

```
    for (int i = 0; i < 8; i++) {
        out[i] = (uint8_t)(v >> (56 - 8 * i));
    }
}
static uint16_t load_be16(const uint8_t *in)
{
    return (uint16_t)(((uint16_t)in[0] << 8) | in[1]);
}
static uint32_t load_be32(const uint8_t *in)
{
    return ((uint32_t)in[0] << 24) | ((uint32_t)in[1] << 16)
         | ((uint32_t)in[2] <<  8) |  (uint32_t)in[3];
}
static uint64_t load_be64(const uint8_t *in)
{
    uint64_t v = 0;
    for (int i = 0; i < 8; i++) v = (v << 8) | in[i];
    return v;
}

/* ---------- the in-memory record ---------- */

#define NAME_BYTES 32

typedef struct {
    uint32_t id;
    uint16_t score;
    uint64_t timestamp;
    char     name[NAME_BYTES];      /* UTF-8 bytes, NUL padded */
} Record;

/* ---------- the wire format: fixed offsets, fixed widths, big-endian ----------
 *   offset  size  field
 *        0     4  magic   "CREC"
 *        4     2  version
 *        6     4  id
 *       10     2  score
 *       12     8  timestamp
 *       20    32  name (UTF-8, NUL padded)
```

```
 *   total 52 bytes, identical on every machine
 */

#define WIRE_SIZE   52
#define WIRE_MAGIC  "CREC"
#define WIRE_VERSION 1

static void record_encode(const Record *r, uint8_t out[WIRE_SIZE])
{
    memcpy(out, WIRE_MAGIC, 4);
    store_be16(out + 4,  WIRE_VERSION);
    store_be32(out + 6,  r->id);
    store_be16(out + 10, r->score);
    store_be64(out + 12, r->timestamp);
    memset(out + 20, 0, NAME_BYTES);
    memcpy(out + 20, r->name, strnlen(r->name, NAME_BYTES));
}

static bool record_decode(const uint8_t in[WIRE_SIZE], Record *out)
{
    if (memcmp(in, WIRE_MAGIC, 4) != 0) {
        return false;                       /* not our format */
    }
    uint16_t version = load_be16(in + 4);
    if (version != WIRE_VERSION) {
        return false;                       /* version check, not a guess */
    }
    out->id        = load_be32(in + 6);
    out->score     = load_be16(in + 10);
    out->timestamp = load_be64(in + 12);
    memcpy(out->name, in + 20, NAME_BYTES);
    out->name[NAME_BYTES - 1] = '\0';       /* guarantee termination */
    return true;
}

/* ---------- UTF-8 ---------- */

static size_t utf8_length(const char *s)
{
```

```
    size_t chars = 0;
    for (; *s != '\0'; s++) {
        if (((unsigned char)*s & 0xC0) != 0x80) chars++;
    }
    return chars;
}

/* Truncate to at most max_bytes without splitting a character. */
static size_t utf8_safe_truncate(const char *s, size_t max_bytes)
{
    if (strlen(s) <= max_bytes) return strlen(s);
    size_t cut = max_bytes;
    while (cut > 0 && ((unsigned char)s[cut] & 0xC0) == 0x80) {
        cut--;                              /* back off to a lead byte */
    }
    return cut;
}

int main(void)
{
    puts("== this machine ==");
    printf("  CHAR_BIT assumed 8; sizeof: int %zu, long %zu, void * %zu\n",
           sizeof(int), sizeof(long), sizeof(void *));
    printf("  data model  : %s\n",
           sizeof(long) == 8 ? "LP64 (Linux/macOS)" :
           sizeof(void *) == 8 ? "LLP64 (Windows)" : "ILP32");
    printf("  byte order  : %s\n",
           is_little_endian() ? "little-endian" : "big-endian");
    printf("  alignment   : int %zu, double %zu, max %zu\n",
           _Alignof(int), _Alignof(double), _Alignof(max_align_t));
    printf("  plain char is %s\n", (char)-1 < 0 ? "signed" : "unsigned");

    puts("\n== why a raw struct is not a format ==");
    printf("  sizeof(Record) = %zu, but the fields total %zu\n",
           sizeof(Record), (size_t)(4 + 2 + 8 + NAME_BYTES));
    printf("  offsets: id %zu, score %zu, timestamp %zu, name %zu\n",
           offsetof(Record, id), offsetof(Record, score),
           offsetof(Record, timestamp), offsetof(Record, name));
    printf("  the wire format is a fixed %d bytes on every machine\n",
```

```
       WIRE_SIZE);

puts("\n== encode and decode ==");
Record original = {
    .id = 0x12345678u,
    .score = 950,
    .timestamp = UINT64_C(1735689600),
    .name = "Ada Lovelace"
};

uint8_t wire[WIRE_SIZE];
record_encode(&original, wire);

printf("  first 20 bytes: ");
for (int i = 0; i < 20; i++) printf("%02X ", wire[i]);
putchar('\n');
puts("  note bytes 6-9 are 12 34 56 78 — most significant first,");
puts("  regardless of how this machine stores integers internally");

Record restored;
if (record_decode(wire, &restored)) {
    printf("  id        : 0x%08" PRIX32 " %s\n", restored.id,
           restored.id == original.id ? "ok" : "MISMATCH");
    printf("  score     : %" PRIu16 " %s\n", restored.score,
           restored.score == original.score ? "ok" : "MISMATCH");
    printf("  timestamp : %" PRIu64 " %s\n", restored.timestamp,
           restored.timestamp == original.timestamp ? "ok" : "MISMATCH");
    printf("  name      : \"%s\"\n", restored.name);
}

puts("\n== rejecting foreign and future data ==");
uint8_t alien[WIRE_SIZE] = { 'X','X','X','X' };
printf("  wrong magic   : %s\n",
       record_decode(alien, &restored) ? "accepted (bad)" : "rejected");
memcpy(alien, WIRE_MAGIC, 4);
store_be16(alien + 4, 99);
printf("  future version: %s\n",
       record_decode(alien, &restored) ? "accepted (bad)" : "rejected");
```

```
    puts("\n== unaligned access ==");
    uint8_t buffer[16] = { 0, 0x12, 0x34, 0x56, 0x78 };
    uint32_t value;
    memcpy(&value, buffer + 1, sizeof value);     /* the correct way */
    printf("  memcpy from an odd offset: 0x%08" PRIX32 "\n", value);
    puts("  *(uint32_t *)(buffer + 1) would be an alignment and");
    puts("  aliasing violation; it crashes on some architectures");

    puts("\n== UTF-8 ==");
    const char *turkish = "türkçe karakterler";
    printf("  \"%s\"\n", turkish);
    printf("  strlen      = %zu bytes\n", strlen(turkish));
    printf("  utf8_length = %zu characters\n", utf8_length(turkish));

    for (size_t limit = 1; limit <= 4; limit++) {
        size_t cut = utf8_safe_truncate(turkish, limit);
        printf("  truncate to %zu bytes -> %zu bytes: \"%.*s\"\n",
               limit, cut, (int)cut, turkish);
    }
    puts("  cutting at an arbitrary byte would split a character in half");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address,undefined -o portable portable.c
./portable
```

Why the wire format is built this way

A magic number. Four bytes that identify the format, so a wrong file is rejected instead of misread. Every real binary format has one.

A version field. Checked, not assumed. Without it, a version 2 file read by a version 1 program silently produces wrong values. With it, the program refuses and says why.

Fixed widths and fixed offsets. Nothing depends on `sizeof` or on the compiler's padding. The record is 52 bytes on a 32-bit ARM board and on a 64-bit x86 server.

Explicit byte order. The shift-based accessors operate on values, so they produce the same bytes regardless of host order. Note that the program prints `12 34 56 78` for `0x12345678`

even though the machine stores it as `78 56 34 12` internally.

Prove the endianness handling

```
./portable | grep -A1 'first 20 bytes'
xxd -l 20 record.bin           # for comparison with a raw struct dump
```

If you have access to a big-endian environment — QEMU can emulate one, which chapters 49 onward use anyway — run the same program there. The debug output for byte order changes; the twenty bytes of wire format do not. That is the whole objective.

```
qemu-s390x -L /usr/s390x-linux-gnu ./portable_s390x   # big-endian target
```

Break it on purpose

Replace `record_encode` with a raw structure write:

```
fwrite(&original, sizeof original, 1, f);
```

Now the file contains this compiler's padding, this machine's byte order, and this platform's type sizes. It is unusable on most other configurations, and — worse — readable but *wrong* on a machine with the same sizes and different byte order. Chapter 26 used exactly this shortcut; here is the reason it was flagged as unsuitable for exchange.

The UTF-8 truncation

The output shows that truncating `"türkçe"` to 2 bytes yields 1 byte, because the `ü` occupies bytes 1 and 2 and cutting between them would produce invalid UTF-8. Naive truncation is how a database field or a display label ends in a replacement character — and in a protocol it can be a parsing failure rather than a cosmetic one.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Assuming `long` is 64 bits	Works on Linux, breaks on Windows	`int64_t`.
Writing a raw struct to a file	Padding and byte order baked in	Explicit field-by-field encoding.
Forgetting `htonl` on a socket	Works between identical machines only	Convert at every boundary.
`*(uint32_t *)(buf + 1)`	Unaligned access and aliasing violation	`memcpy`.
`%ld` for an `int64_t`	Format mismatch on some platforms	`PRId64` from `<inttypes.h>`.
Storing a byte value in plain `char`	Signed on x86, unsigned on ARM	`uint8_t`.
`strlen` as a character count	Wrong for any non-ASCII text	Count non-continuation bytes.
Truncating UTF-8 at a byte boundary	Invalid sequence; broken display	Back off to a lead byte.
No magic or version in a binary format	Wrong files silently misread	Both, and check both.

8 Check yourself

Why is `long` a poor choice for a 64-bit value?

Because the standard guarantees only 32 bits, and the mainstream 64-bit platforms disagree: it is 8 bytes under LP64 on Linux and macOS and 4 bytes under LLP64 on Windows. Code assuming the wider size compiles on both and produces wrong results on one. `int64_t` states the requirement exactly.

Why do the shift-based encoders work regardless of the host's byte order?

Because shifting operates on the numeric *value*, not on its representation in memory. `v >> 24` extracts the most significant byte of the number on any machine, so writing it first always produces big-endian output. A cast-and-copy approach would instead expose whatever the host happens to store.

Why use `memcpy` to read an integer out of a byte buffer?

Because it solves two problems at once: the buffer offset may not satisfy the integer's alignment requirement, which crashes on some architectures, and casting the pointer would violate strict aliasing. `memcpy` is defined in both respects and compiles to the same single instruction where the hardware permits an unaligned load.

What three properties made UTF-8 the universal encoding?

ASCII passes through unchanged, so existing text and existing code keep working; no byte of a multi-byte sequence can be confused with an ASCII character, so byte-oriented searching and splitting stay correct; and it is byte-order independent, so it needs no conversion when it moves between machines.

Why must a binary format carry a magic number and a version?

The magic identifies the format so a wrong file is rejected rather than interpreted as garbage. The version lets a reader refuse data written by a newer writer instead of misreading it — the alternative is silently wrong values, which is far worse than an error. Both must be checked, not merely written.

9 Where this leads

Chapter 38 completes the language: the qualifiers `volatile` and `restrict`, the `register` storage class, inline functions, compound literals, anonymous structures, variable-length arrays, and non-local jumps. `volatile` in particular is the bridge to the embedded chapters, where it stops being an optimization detail and becomes the difference between a working driver and a hanging one.

ADVANCED · CHAPTER 38

Type Qualifiers, Inlining, and Modern C

The features left over — each small, each occasionally essential. Two of them are promises to the optimizer that it cannot verify, one is the bridge to hardware programming, and one is a control-flow construct most C programmers never use and should recognize when they meet it.

BY THE END OF THIS CHAPTER YOU CAN

- Say what `volatile` actually prevents, and what it does not.
- Apply `restrict` correctly and see the difference in the generated code.
- Use `inline` and `static inline` and explain C's inline rules.
- Write compound literals and anonymous unions where they make code clearer.
- Say why variable-length arrays are risky and recognize `setjmp` / `longjmp`.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 `volatile`

`volatile` tells the compiler that an object may change by means it cannot see, so **every read must actually read and every write must actually write**, in program order, with no caching in registers and no elimination of "redundant" accesses.

```
/* without volatile, the compiler may hoist the load out of the loop */
while (flag == 0) { }              /* becomes an infinite loop at -O2 */

volatile int flag;
while (flag == 0) { }              /* re-reads memory every iteration */
```

Three legitimate uses, and only three:

USE	WHY
Memory-mapped hardware registers	The device changes the value; reading may itself have an effect. Chapter 50.
A variable shared with a signal handler	Must be `volatile sig_atomic_t`. Chapter 39.
A variable modified across `setjmp` / `longjmp`	Otherwise its value after a jump is indeterminate.

> `volatile` **is not for threads.** It guarantees the access happens; it guarantees nothing about atomicity, and nothing about ordering relative to other variables. `volatile int counter; counter++;` is still a data race — the increment is a load, an add, and a store. Use `<stdatomic.h>`, which chapter 41 covers. This misconception is widespread and produces bugs that appear only under load.

2 `restrict`

Chapter 36 introduced it as a promise: within this scope, the object reached through this pointer is reached through no other. The payoff is visible in the generated code.

```
void add_plain(int *dst, const int *a, const int *b, size_t n)
{
    for (size_t i = 0; i < n; i++) dst[i] = a[i] + b[i];
}

void add_restrict(int *restrict dst, const int *restrict a,
                  const int *restrict b, size_t n)
{
    for (size_t i = 0; i < n; i++) dst[i] = a[i] + b[i];
}
```

In the first, `dst[i]` might alias `a[i+1]`, so the compiler must store and then reload on every iteration — no vectorization. In the second it may load a block of `a` and `b`, add them with vector instructions, and store a block.

The promise is unchecked, and violating it is undefined behavior with no diagnostic. This is exactly the contract that distinguishes `memcpy` from `memmove`.

`register`

Obsolete as an optimization — compilers have allocated registers better than humans since the 1980s. It retains one effect: you cannot take the address of a `register` object, which occasionally documents intent. C23 gives it no new meaning. You will meet it only in old code.

3 `inline`

C's inline rules differ from C++'s and confuse people reliably.

```
/* In a header — this is the form to use. */
static inline int max_int(int a, int b)
{
    return a > b ? a : b;
}
```

`static inline` in a header is the practical answer. Each translation unit gets its own copy with internal linkage, so there is no multiple-definition problem and the compiler inlines it freely. This is the modern replacement for a function-like macro, and it fixes both macro hazards from chapter 28: arguments are evaluated once, and the result is type-checked.

Plain `inline` without `static` provides an *inline definition* that is not an external definition, so if the compiler chooses not to inline a call you get an undefined reference unless exactly one translation unit also provides an external definition. The rule is unintuitive and the reason most codebases simply write `static inline`.

In all cases `inline` is a hint. The compiler inlines small functions at `-O2` whether or not you ask, and declines large ones whatever you say.

4 Compound literals and anonymous members

A *compound literal* creates an unnamed object of a given type, in place:

```
draw_point((Point){ .x = 3, .y = 4 });          /* no temporary variable */

int *table = (int[]){ 1, 1, 2, 3, 5, 8 };       /* an unnamed array */

config = (Config){ 0 };                          /* reset every field */
```

Its lifetime is that of the enclosing block, so returning a pointer to one from a function is the dangling-pointer bug of chapter 13 in a new costume.

Anonymous structures and unions (C11) let a nested member be accessed as though it were direct — which makes chapter 22's tagged union considerably more readable:

```
typedef struct {
    ValueKind kind;
    union {                          /* no member name */
        int    as_int;
        double as_double;
    };
} Value;

v.as_int = 42;                       /* not v.data.as_int */
```

5 Variable-length arrays

```
void process(size_t n)
{
    int buffer[n];              /* size decided at run time */
    …
}
```

Added in C99, made optional in C11 — an implementation may define `__STDC_NO_VLA__` and refuse them, and MSVC never supported them. That alone disqualifies them from portable code.

The larger problem is that a VLA is on the stack and **there is no way to detect failure**. Chapter 17 established that stack overflow is a fatal signal, not a return value. A size derived from input turns that into a remotely triggerable crash:

```
void handle(size_t count)      /* count came from the network */
{
    int buffer[count];         /* count = 10000000 kills the process */
}
```

Use `malloc`, whose failure you can check, or a fixed-size buffer with an explicit bound. VLAs are worth recognizing in existing code and worth avoiding in new code.

Variably modified *parameter* types are a different matter and remain useful — chapter 16's `void f(size_t rows, size_t cols, int m[rows][cols])` allocates nothing and lets the compiler compute correct offsets.

6 `_Noreturn`, `setjmp`, and `longjmp`

```
_Noreturn void fatal(const char *message)      /* C11; [[noreturn]] in C23 */
{
    fprintf(stderr, "fatal: %s\n", message);
    exit(EXIT_FAILURE);
}
```

Declaring that a function never returns lets the compiler drop the code after a call to it and improves its warnings — no "control reaches end of non-void function" after a call to `fatal`.

Non-local jumps

```
#include <setjmp.h>

static jmp_buf recovery;

if (setjmp(recovery) == 0) {
    deep_call_chain();              /* may longjmp back here */
} else {
    /* arrived via longjmp: handle the failure */
}

/* somewhere far below */
longjmp(recovery, 1);               /* unwinds straight back */
```

This is the closest C has to an exception, and it is a blunt instrument. `setjmp` records the machine state; `longjmp` restores it, discarding every intervening frame — **without running any cleanup**. Every file left open and every block allocated between the two is leaked.

Further rules: the `jmp_buf` is invalid once the function that called `setjmp` returns, and any local modified between `setjmp` and `longjmp` must be `volatile` or its value is indeterminate afterwards.

Legitimate uses are rare — a parser abandoning a deeply nested descent, or a test harness recovering from a failed assertion. Chapter 27's error contract is the right tool for everything

else.

7 Worked example: seeing the qualifiers work

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>
#include <setjmp.h>
#include <time.h>

#define N 2000000
#define REPS 50

static double now(void)
{
    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC, &t);
    return (double)t.tv_sec + (double)t.tv_nsec / 1e9;
}

/* ---------- restrict ---------- */

static void add_plain(int *dst, const int *a, const int *b, size_t n)
{
    for (size_t i = 0; i < n; i++) dst[i] = a[i] + b[i];
}

static void add_restrict(int *restrict dst, const int *restrict a,
                         const int *restrict b, size_t n)
{
    for (size_t i = 0; i < n; i++) dst[i] = a[i] + b[i];
}

/* ---------- static inline replaces a macro ---------- */

#define MAX_MACRO(a, b) ((a) > (b) ? (a) : (b))     /* evaluates twice */
```

```
static inline int max_int(int a, int b)             /* evaluates once */
{
    return a > b ? a : b;
}

static int side_effect_count = 0;
static int with_side_effect(int v)
{
    side_effect_count++;
    return v;
}

/* ---------- compound literals ---------- */

typedef struct { int x, y; } Point;

static int manhattan(Point a, Point b)
{
    int dx = a.x - b.x, dy = a.y - b.y;
    return (dx < 0 ? -dx : dx) + (dy < 0 ? -dy : dy);
}

/* ---------- anonymous union ---------- */

typedef enum { V_INT, V_DOUBLE, V_TEXT } VKind;

typedef struct {
    VKind kind;
    union {                             /* anonymous: C11 */
        int    as_int;
        double as_double;
        char   as_text[16];
    };
} Value;

static void value_print(const Value *v)
{
    switch (v->kind) {
    case V_INT:    printf("  int    %d\n", v->as_int);       break;
```

```
    case V_DOUBLE: printf("  double %g\n", v->as_double);    break;
    case V_TEXT:   printf("  text   \"%s\"\n", v->as_text);  break;
    }
}

/* ---------- _Noreturn ---------- */

_Noreturn static void fatal(const char *msg)
{
    fprintf(stderr, "fatal: %s\n", msg);
    exit(EXIT_FAILURE);
}

/* ---------- setjmp / longjmp ---------- */

static jmp_buf recovery;

static void level_three(int depth)
{
    printf("    level three, depth %d\n", depth);
    if (depth > 2) {
        puts("    jumping back, skipping every frame between");
        longjmp(recovery, depth);       /* no cleanup runs */
    }
}
static void level_two(int depth)   { printf("   level two\n");  level_three(depth); }
static void level_one(int depth)   { printf("  level one\n");   level_two(depth); }

int main(void)
{
    puts("== restrict: the same loop, two contracts ==");
    int *a   = malloc(N * sizeof *a);
    int *b   = malloc(N * sizeof *b);
    int *dst = malloc(N * sizeof *dst);
    if (a == NULL || b == NULL || dst == NULL) fatal("out of memory");

    for (size_t i = 0; i < N; i++) { a[i] = (int)i; b[i] = (int)(i * 2); }

    double t = now();
```

```
    for (int r = 0; r < REPS; r++) add_plain(dst, a, b, N);
    double plain_secs = now() - t;

    t = now();
    for (int r = 0; r < REPS; r++) add_restrict(dst, a, b, N);
    double restrict_secs = now() - t;

    printf("  without restrict : %.4f s\n", plain_secs);
    printf("  with restrict    : %.4f s\n", restrict_secs);
    printf("  ratio %.2fx (check the assembly for vector instructions)\n",
           plain_secs / restrict_secs);
    printf("  last element %d, both versions agree\n", dst[N - 1]);

    puts("\n== static inline versus a macro ==");
    side_effect_count = 0;
    int m1 = MAX_MACRO(with_side_effect(3), with_side_effect(5));
    printf("  MAX_MACRO gave %d after %d calls  <-- evaluated twice\n",
           m1, side_effect_count);

    side_effect_count = 0;
    int m2 = max_int(with_side_effect(3), with_side_effect(5));
    printf("  max_int   gave %d after %d calls  <-- evaluated once\n",
           m2, side_effect_count);

    puts("\n== compound literals ==");
    printf("  manhattan((0,0),(3,4)) = %d   no temporaries declared\n",
           manhattan((Point){ .x = 0, .y = 0 }, (Point){ .x = 3, .y = 4 }));

    int *fib = (int[]){ 1, 1, 2, 3, 5, 8, 13 };
    printf("  an unnamed array: ");
    for (int i = 0; i < 7; i++) printf("%d ", fib[i]);
    puts("\n  it lives until the end of this block, and no longer");

    puts("\n== anonymous union ==");
    Value values[] = {
        { .kind = V_INT,    .as_int    = 42 },
        { .kind = V_DOUBLE, .as_double = 3.14 },
        { .kind = V_TEXT,   .as_text   = "hello" }
    };
```

```
    for (size_t i = 0; i < 3; i++) value_print(&values[i]);
    printf("  accessed as v->as_int, not v->data.as_int\n");

    puts("\n== volatile ==");
    int          normal = 0;
    volatile int observed = 0;
    for (int i = 0; i < 3; i++) { normal++; observed++; }
    printf("  normal = %d, volatile = %d\n", normal, observed);
    puts("  at -O2 the first may be computed at compile time;");
    puts("  the volatile one must be loaded and stored three times");
    puts("  volatile does NOT make ++ atomic — that is chapter 41");

    puts("\n== setjmp and longjmp ==");
    int landed = setjmp(recovery);
    if (landed == 0) {
        puts("  first pass, descending:");
        level_one(3);
        puts("  never reached");
    } else {
        printf("  back in main via longjmp with value %d\n", landed);
        puts("  three frames were discarded and no cleanup ran");
        puts("  anything they allocated has leaked");
    }

    puts("\n== why VLAs are avoided ==");
    size_t safe_size = 100;
    printf("  a VLA of %zu ints would be fine\n", safe_size);
    puts("  a VLA of a size taken from input cannot report failure:");
    puts("  too large and the process dies with no way to catch it");
    int *heap = malloc(safe_size * sizeof *heap);
    printf("  malloc can: %s\n", heap ? "allocation checked" : "refused");
    free(heap);

    free(a); free(b); free(dst);
    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -O2 -o modern modern.c
./modern
```

Confirm `restrict` in the assembly

The timing difference varies by compiler and may be small — GCC sometimes generates a runtime overlap check and two loop versions, recovering most of the gain without the qualifier. The assembly is the reliable evidence:

```
gcc -O2 -S modern.c -o modern.s
sed -n '/^add_plain:/,/ret/p'    modern.s | grep -c 'xmm\|ymm'   # needs add_plain non-
static or noinline
sed -n '/^add_restrict:/,/ret/p' modern.s | grep -c 'xmm\|ymm'
```

Vector registers appear in the `restrict` version, and often not in the other. That is the qualifier doing its job: it removed the possibility the compiler had to defend against.

Violate the promise

```
add_restrict(a, a, b, N);        /* dst and a are the same array */
```

This compiles silently and is undefined behavior. The result may be correct, or may be wrong in a way that depends on the vector width. Nothing reports it — which is exactly why `restrict` must only be applied where you can guarantee the condition.

Watch `volatile` change the generated code

```
gcc -O2 -S modern.c -o modern.s
grep -B3 -A6 'observed' modern.s   # build modern.s with -fverbose-asm so local names appear
```

The plain counter is typically folded to a constant; the `volatile` one produces three loads and three stores. Then try the loop that shows why it matters:

```
int flag = 0;
while (flag == 0) { }        /* -O2: the load is hoisted; infinite loop */
```

Add `volatile` and the load returns to the loop body. In chapter 50 that difference is a driver that works versus a driver that hangs.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`volatile` for thread synchronization	Still a data race; no atomicity	`<stdatomic.h>` — chapter 41.
Omitting `volatile` on a hardware register	Reads cached; the driver hangs	Chapter 50 depends on it.
`restrict` on pointers that may overlap	Undefined, no diagnostic	Promise only what you can guarantee.
Plain `inline` in a header	`undefined reference` when not inlined	`static inline`.
Returning a pointer to a compound literal	Dangles at the end of the block	Return by value, or allocate.
A VLA sized from input	Unstoppable stack overflow	`malloc`, whose failure you can check.
`longjmp` past resources	Files and allocations leak	Prefer an error contract — chapter 27.
Non-`volatile` local across `setjmp`	Indeterminate value after the jump	Mark it `volatile`.

9 Check yourself

What does `volatile` guarantee, and what does it not?

It guarantees that every read and write in the source produces an actual memory access, in program order, with nothing cached in a register or optimized away. It guarantees nothing about atomicity or about ordering relative to other variables, so it does not make `counter++` safe between threads. Its real uses are hardware registers, signal-handler flags, and variables live across `longjmp`.

Why does `restrict` let a loop vectorize?

Without it the compiler must assume the destination could overlap a source, so a store might change a value it is about to load — forcing it to reload every iteration and process one element at a time. `restrict` removes that possibility, so it may load and add whole blocks with vector instructions.

Why is `static inline` preferred over plain `inline` in a header?

Because plain `inline` provides an inline definition that is not an external definition: if the compiler declines to inline a call, the linker needs an external definition that exactly one translation unit must supply. `static inline` gives each unit its own internal-linkage copy, so there is no multiple-definition problem and no missing-symbol problem.

Why should a variable-length array never be sized from external input?

Because it is allocated on the stack and there is no way to detect failure. Exceeding the stack raises a fatal signal rather than returning an error, so a large value from a file or the network becomes a crash the program cannot catch. `malloc` returns `NULL`, which you can check and handle.

What does `longjmp` fail to do that makes it dangerous?

It discards every stack frame between the jump and the `setjmp` without running any cleanup — C has no destructors, so open files stay open and allocated blocks leak. It also requires that locals modified in between be `volatile`, and that the function containing `setjmp` has not returned. An explicit error contract is almost always the better tool.

10 Where this leads

The language is now complete — there is no C syntax left to introduce. Chapter 39 turns to the operating system underneath: file descriptors, processes, signals, and pipes, which are the interface between a C program and everything outside it. `volatile sig_atomic_t` makes its first real appearance there.

ADVANCED · CHAPTER 39

Processes, Signals, and Low-Level I/O

Below `FILE *` there is a file descriptor, and below `system()` there is `fork` and `exec`. This chapter opens the layer where a C program meets the operating system — the interface Unix was written in C to express, and the one every shell is built from.

BY THE END OF THIS CHAPTER YOU CAN

- Use file descriptors directly and say how they relate to `FILE *`.
- Create a process with `fork`, replace it with `exec`, and collect its status.
- Avoid zombies and explain what one is.
- Install a signal handler and say what may legally be done inside it.
- Connect two processes with a pipe and redirect a descriptor.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 File descriptors

A file descriptor is a small non-negative integer indexing a per-process table of open files. Three are open when your program starts:

DESCRIPTOR	NAME	STREAM
0	`STDIN_FILENO`	`stdin`
1	`STDOUT_FILENO`	`stdout`
2	`STDERR_FILENO`	`stderr`

```
#include <fcntl.h>
#include <unistd.h>

int fd = open("data.txt", O_RDONLY);
if (fd < 0) { perror("open"); return EXIT_FAILURE; }

char buffer[4096];
ssize_t n = read(fd, buffer, sizeof buffer);
if (n < 0) { perror("read"); }

close(fd);
```

`FILE *` (C STANDARD)	FILE DESCRIPTOR (POSIX)
Buffered	Unbuffered: every call is a system call
Portable to any C implementation	POSIX only
Formatted I/O	Bytes only
Does not survive `exec`	Inherited across `fork` and `exec`

Convert between them with `fileno(FILE *)` and `fdopen(int, mode)`. Do not mix buffered and unbuffered access to the same file without flushing — the two layers have separate ideas of the position.

> `read` **and** `write` **may transfer less than you asked.** A short read is normal at end of file, on a pipe, or on a socket; a short write is normal when a pipe is full. Code that assumes the full count silently loses data. Always loop until the total is satisfied or an error occurs, and treat `EINTR` — interrupted by a signal — as "retry", not as a failure.

2 fork

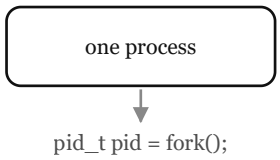
```
#include <unistd.h>

pid_t pid = fork();

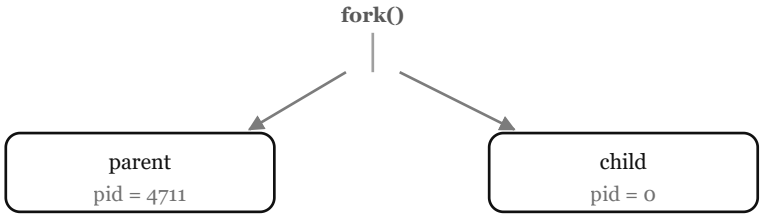
if (pid < 0) {
    perror("fork");                 /* failed */
} else if (pid == 0) {
    /* child: fork returned 0 */
} else {
    /* parent: fork returned the child's process id */
}
```

`fork` duplicates the calling process. Both continue from the same point with the same memory contents, the same open descriptors, and the same variable values — and the *only* way to tell them apart is the return value.

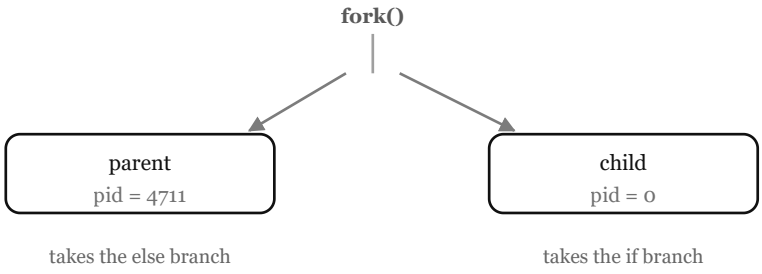
The memory is copied lazily: pages are shared until one process writes, at which point that page is duplicated. So `fork` is cheap, and the two processes are nonetheless fully independent — a variable changed in the child is unchanged in the parent.



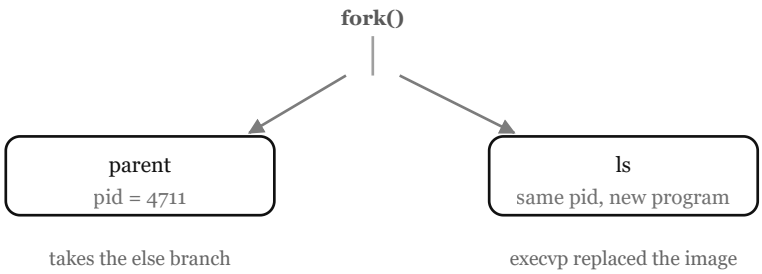
One process, about to call `fork`.



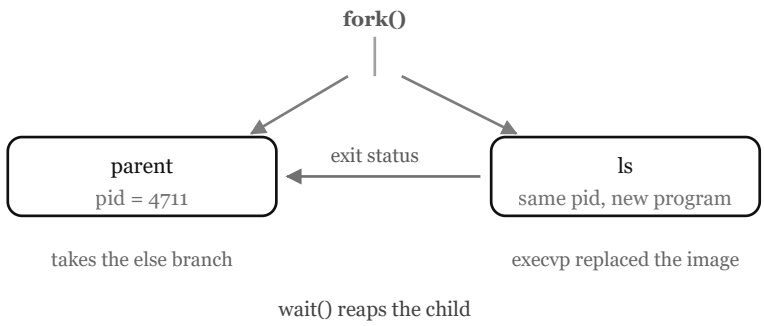
The call returns *twice*, once in each process. Same code, same memory contents, same open descriptors — the only difference in the whole universe is the value it returned: the child's pid in the parent, and 0 in the child.



So the return value is the branch. Both processes run the same `if`, and take opposite sides of it. Memory is copied lazily, so a variable the child changes is unchanged in the parent.



`execvp` replaces the program running in the child. Same process, same pid, same descriptors — an entirely different program. This is why `fork` and `exec` are two calls and not one.



The parent calls `wait` and collects the exit status. Skip this and the child stays a zombie: finished, but still holding a slot until somebody reads its status.

Descriptors, by contrast, are genuinely shared: both processes hold references to the same open file description, including the file position. That sharing is what makes pipes work.

3 exec and waiting

`exec` replaces the current process image with a different program. It does not return on success — there is nothing to return to.

```
execlp("ls", "ls", "-l", (char *)NULL);      /* searches PATH */
perror("execlp");                             /* only reached on failure */
_exit(127);                                   /* note _exit, not exit */
```

The `_exit` matters: after a failed `exec` in a child, calling `exit` would run the parent's `atexit` handlers and flush the parent's buffered output a second time, producing duplicated text. Use `_exit` in a child that is abandoning ship.

```
#include <sys/wait.h>

int status;
pid_t done = waitpid(pid, &status, 0);

if (WIFEXITED(status)) {
    printf("exited with %d\n", WEXITSTATUS(status));
} else if (WIFSIGNALED(status)) {
    printf("killed by signal %d\n", WTERMSIG(status));
}
```

Zombies

When a child exits, the kernel keeps its exit status until the parent collects it. Until then the child is a *zombie* — no memory, no code, just a table entry. A parent that never waits accumulates them until the process table fills.

Three remedies: call `waitpid` for each child; reap them in a `SIGCHLD` handler; or set `SIGCHLD` to `SIG_IGN`, which tells the kernel you will never ask.

4 Signals

A signal is an asynchronous notification. Your handler runs at an arbitrary point in whatever the program was doing.

SIGNAL	CAUSE	DEFAULT
`SIGINT`	Ctrl-C	Terminate
`SIGTERM`	`kill`	Terminate
`SIGSEGV`	Invalid memory access	Terminate with a core dump
`SIGCHLD`	A child changed state	Ignored
`SIGPIPE`	Wrote to a pipe with no reader	Terminate
`SIGKILL`, `SIGSTOP`	—	**Cannot be caught**

```
#include <signal.h>

static volatile sig_atomic_t should_stop = 0;

static void on_interrupt(int sig)
{
    (void)sig;
    should_stop = 1;              /* the only safe thing to do */
}

struct sigaction sa = { 0 };
sa.sa_handler = on_interrupt;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;          /* restart interrupted system calls */
sigaction(SIGINT, &sa, NULL);
```

Use `sigaction`, not `signal`: the older interface has behavior that varies between systems, and `sigaction` lets you control the mask and the restart flag.

Async-signal safety

A handler can interrupt the program anywhere — including in the middle of `malloc` updating its data structures. Calling `malloc` from the handler then corrupts the heap. The same applies to `printf`, which has its own locks and buffers.

Only functions the standard lists as *async-signal-safe* may be called: `write`, `_exit`, `signal`, `kill`, and a few dozen others. `printf`, `malloc`, and `free` are not among them.

The practical discipline is to set a flag and return:

```
static volatile sig_atomic_t should_stop = 0;


while (!should_stop) {
    do_work();                    /* the real work happens here */
}
puts("shutting down cleanly");    /* printf is safe out here */
```

`volatile` so the loop actually re-reads it — chapter 38's first legitimate use. `sig_atomic_t` because it is the only type guaranteed to be read and written atomically with respect to a signal.

5 Pipes and redirection

```
int fds[2];
pipe(fds);            /* fds[0] to read, fds[1] to write */
```

A pipe is a one-way channel between two descriptors. After `fork`, both processes hold both ends, so each must close the one it does not use — otherwise the reader never sees end of file, because a writing end is still open in its own process.

`dup2` makes one descriptor a copy of another, which is how redirection works:

```
dup2(fds[1], STDOUT_FILENO);      /* the child's stdout becomes the pipe */
close(fds[1]);
```

After this the child's `printf` writes into the pipe without knowing anything has changed. That is precisely what a shell does for `ls | grep foo`, and the reason descriptor numbers

rather than `FILE *` are the operating system's currency.

6 Worked example: a miniature shell

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include <sys/wait.h>

/* Set by the handler, read by the loop. Both qualifiers are required. */
static volatile sig_atomic_t interrupted = 0;

static void on_sigint(int sig)
{
    (void)sig;
    interrupted = 1;
    /* Nothing else. write() would be safe; printf() would not. */
}

/* read and write may transfer less than requested; loop. */
static ssize_t write_all(int fd, const void *data, size_t n)
{
    const char *p = data;
    size_t written = 0;
    while (written < n) {
        ssize_t w = write(fd, p + written, n - written);
        if (w < 0) {
            if (errno == EINTR) continue;      /* retry, not an error */
            return -1;
        }
        written += (size_t)w;
    }
    return (ssize_t)written;
```

```
}

static void describe_status(const char *what, int status)
{
    if (WIFEXITED(status)) {
        printf("  %s exited with %d\n", what, WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("  %s killed by signal %d\n", what, WTERMSIG(status));
    } else {
        printf("  %s ended in an unexpected way\n", what);
    }
}

/* ---------- 1. fork and exec one command ---------- */

static int run_command(char *const argv[])
{
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        return -1;
    }
    if (pid == 0) {                     /* child */
        execvp(argv[0], argv);
        perror(argv[0]);                /* only on failure */
        _exit(127);                     /* _exit, not exit */
    }

    int status;
    while (waitpid(pid, &status, 0) < 0) {
        if (errno != EINTR) { perror("waitpid"); return -1; }
    }
    return status;
}

/* ---------- 2. redirect output to a file ---------- */

static int run_redirected(char *const argv[], const char *path)
{
```

```
    pid_t pid = fork();
    if (pid < 0) { perror("fork"); return -1; }

    if (pid == 0) {
        int fd = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (fd < 0) { perror(path); _exit(126); }
        if (dup2(fd, STDOUT_FILENO) < 0) { perror("dup2"); _exit(126); }
        close(fd);                          /* the copy in fd 1 remains */
        execvp(argv[0], argv);
        perror(argv[0]);
        _exit(127);
    }

    int status;
    waitpid(pid, &status, 0);
    return status;
}

/* ---------- 3. connect two commands with a pipe ---------- */

static int run_pipeline(char *const left[], char *const right[])
{
    int fds[2];
    if (pipe(fds) < 0) { perror("pipe"); return -1; }

    pid_t first = fork();
    if (first < 0) { perror("fork"); return -1; }
    if (first == 0) {
        close(fds[0]);                      /* not reading */
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        execvp(left[0], left);
        perror(left[0]);
        _exit(127);
    }

    pid_t second = fork();
    if (second < 0) { perror("fork"); return -1; }
    if (second == 0) {
```

```
        close(fds[1]);                  /* not writing */
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);
        execvp(right[0], right);
        perror(right[0]);
        _exit(127);
    }

    /* The parent must close BOTH ends, or the reader never sees EOF. */
    close(fds[0]);
    close(fds[1]);

    int s1, s2;
    waitpid(first,  &s1, 0);
    waitpid(second, &s2, 0);
    describe_status("left", s1);
    describe_status("right", s2);
    return s2;
}

int main(void)
{
    struct sigaction sa;
    memset(&sa, 0, sizeof sa);
    sa.sa_handler = on_sigint;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGINT, &sa, NULL) < 0) {
        perror("sigaction");
        return EXIT_FAILURE;
    }

    printf("parent pid %ld\n\n", (long)getpid());

    puts("== 1. fork: two processes, one program ==");
    pid_t pid = fork();
    if (pid == 0) {
        printf("  child : pid %ld, fork returned 0, parent is %ld\n",
               (long)getpid(), (long)getppid());
```

```
        _exit(3);
    }
    int status;
    waitpid(pid, &status, 0);
    printf("  parent: fork returned %ld\n", (long)pid);
    describe_status("child", status);

    puts("\n== 2. memory is copied, descriptors are shared ==");
    int shared = 100;
    pid = fork();
    if (pid == 0) {
        shared = 999;
        printf("  child  sees shared = %d\n", shared);
        _exit(0);
    }
    waitpid(pid, NULL, 0);
    printf("  parent sees shared = %d  (the child's change did not reach us)\n",
           shared);

    puts("\n== 3. exec replaces the program ==");
    char *echo_args[] = { "echo", "  hello from execvp", NULL };
    describe_status("echo", run_command(echo_args));

    puts("\n== 4. a command that does not exist ==");
    char *missing[] = { "definitely_not_a_command", NULL };
    describe_status("missing", run_command(missing));
    puts("  127 is the conventional code for 'command not found'");

    puts("\n== 5. redirection with dup2 ==");
    char *date_args[] = { "date", "+%Y-%m-%d", NULL };
    run_redirected(date_args, "shell_out.txt");
    FILE *f = fopen("shell_out.txt", "r");
    if (f != NULL) {
        char line[128];
        if (fgets(line, sizeof line, f)) printf("  file contains: %s", line);
        fclose(f);
    }
    remove("shell_out.txt");
```

```
    puts("\n== 6. a pipeline: seq 1 20 | grep 1 ==");
    char *left[]  = { "seq", "1", "20", NULL };
    char *right[] = { "grep", "1", NULL };
    run_pipeline(left, right);

    puts("\n== 7. low-level write, looped ==");
    const char *msg = "  written with write(2), not printf\n";
    write_all(STDOUT_FILENO, msg, strlen(msg));

    puts("\n== 8. signals ==");
    puts("  press Ctrl-C within three seconds to interrupt the loop");
    for (int i = 0; i < 30 && !interrupted; i++) {
        struct timespec ts = { .tv_sec = 0, .tv_nsec = 100000000L };
        nanosleep(&ts, NULL);
    }
    if (interrupted) {
        puts("  the handler set a flag; this message is printed out here,");
        puts("  where printf is safe — inside a handler it would not be");
    } else {
        puts("  no interrupt arrived");
    }

    puts("\n== 9. zombies ==");
    pid = fork();
    if (pid == 0) _exit(0);
    printf("  child %ld has exited but is not yet reaped: it is a zombie\n",
           (long)pid);
    puts("  run 'ps -el | grep defunct' in another terminal now");
    sleep(1);
    waitpid(pid, NULL, 0);
    puts("  reaped; the process table entry is gone");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -D_POSIX_C_SOURCE=200809L -Wall -Wextra -g -o minishell minishell.c
./minishell
```

The `-D_POSIX_C_SOURCE=200809L` is needed because `-std=c17` asks for strict standard C, which hides the POSIX declarations. Without it you get implicit-declaration errors for `sigemptyset`, `nanosleep` and friends (and an incomplete `struct sigaction`) — chapter 11's diagnostic, from an unexpected direction.

Four things the output demonstrates

Memory is copied, descriptors are shared. The child sets `shared` to 999 and the parent still sees 100 — separate address spaces. Yet both write to the same terminal, because descriptor 1 refers to the same open file description in both.

The parent must close both pipe ends. Comment out `close(fds[1])` in the parent and run the pipeline again: `grep` never terminates, because a write end is still open in the parent, so the pipe never reaches end of file. This is the single most common pipe bug.

Exit code 127. The failed `execvp` child calls `_exit(127)`, the conventional "command not found" code that shells use. Chapter 31's exit-code conventions, on the producing side.

The signal handler does almost nothing. It sets one flag of type `volatile sig_atomic_t` and returns. All reporting happens in the main loop, where the standard library is safe to call.

Break the signal handler on purpose

```
static void on_sigint(int sig)
{
    (void)sig;
    printf("caught it\n");         /* NOT async-signal-safe */
    char *p = malloc(100);         /* definitely not safe */
    free(p);
    interrupted = 1;
}
```

This usually appears to work, which is why the rule is so often ignored. Press Ctrl-C repeatedly while the program is inside `malloc` and the heap can be corrupted, producing a crash somewhere unrelated later. Run it under `-fsanitize=address` and hammer the key; the failure mode is exactly the action-at-a-distance of chapter 20.

Watch the zombie

```
# in another terminal, while the program is in its sleep(1)
ps -el | grep defunct
```

The entry shows state `Z`. Remove the `waitpid` and loop the fork a thousand times to watch the process table fill — the reason a long-running server must reap its children.

Handle a short write

`write_all` loops because a single `write` may transfer fewer bytes than requested. Prove it by making the pipe non-blocking (`O_NONBLOCK`) and writing several megabytes while the reader is slow: the first `write` returns as soon as the pipe buffer is full, typically after 64 KB. Code that ignores the return value loses the remainder silently.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Not closing unused pipe ends	The reader never sees end of file	Close both ends you do not use, in every process.
Never calling `waitpid`	Zombies accumulate	Reap every child, or ignore `SIGCHLD`.
`printf` or `malloc` in a signal handler	Heap corruption; deadlock	Set a `volatile sig_atomic_t` flag and return.
`exit` instead of `_exit` in a child	Parent's buffers flushed twice	`_exit` after a failed `exec`.
Assuming `read` / `write` transfer everything	Silent data loss	Loop until complete; retry on `EINTR`.
Ignoring `EINTR`	Spurious failures once signals are in use	Retry, or set `SA_RESTART`.
Using `signal` rather than `sigaction`	Behavior differs between systems	`sigaction`.
Mixing `FILE *` and raw descriptors	Interleaved or lost output	Flush before switching layers.

8 Check yourself

How does a process tell whether it is the parent or the child after `fork`?

Only by the return value: 0 in the child, the child's process id in the parent, negative on failure. Both continue from the same instruction with identical memory contents, so nothing else distinguishes them. The memory is subsequently independent — a write in one is invisible to the other.

Why must both processes close the pipe ends they do not use?

Because the reader sees end of file only when *every* write end is closed. If the parent keeps its copy of the write end open, the child reading from the pipe blocks forever even after the writer exits. Each process must close both descriptors it does not need, including the parent after forking.

What is a zombie process and how do you avoid one?

A child that has exited but whose exit status the parent has not yet collected; the kernel keeps a process-table entry so the status can be reported. Call `waitpid` for each child, reap them in a `SIGCHLD` handler, or set `SIGCHLD` to `SIG_IGN` to tell the kernel you will never ask.

Why may a signal handler not call `printf`?

Because the handler can interrupt the program at any instruction, including inside `printf` or `malloc` while their internal state is inconsistent. Re-entering them then corrupts that state. Only functions the standard designates async-signal-safe may be called; the safe discipline is to set a `volatile sig_atomic_t` flag and do the work in the main loop.

Why does a child that fails to `exec` call `_exit` rather than `exit`?

Because the child inherited a copy of the parent's buffered output and its `atexit` handlers. `exit` would flush those buffers and run those handlers a second time, duplicating output and possibly repeating cleanup actions. `_exit` terminates immediately without either.

9 Where this leads

Chapter 40 extends the descriptor model across a network: a socket is a file descriptor, `read` and `write` work on it, and the same short-transfer and `EINTR` rules apply. The forked-child-per-connection server built there is the direct descendant of this chapter's pipeline, and chapter 47 replaces it with an event loop.

ADVANCED · CHAPTER 40

Network Programming with Sockets

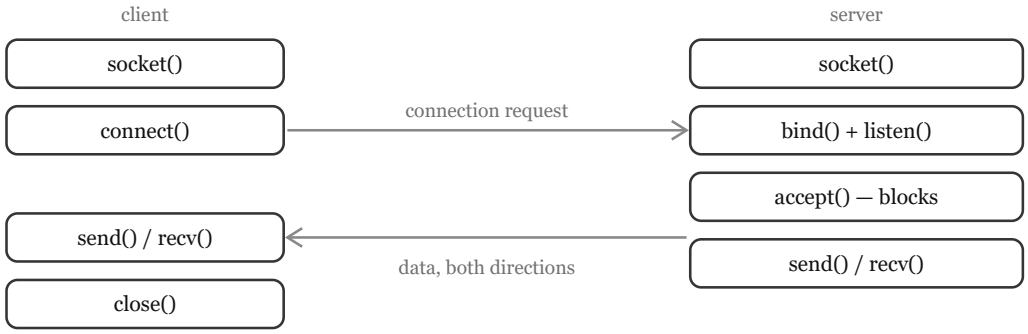
A socket is a file descriptor, so everything from chapter 39 still applies — short transfers, `EINTR`*,* `close`*. What is new is that the other end is a different machine, which makes byte order matter and makes message boundaries something you must invent yourself.*

BY THE END OF THIS CHAPTER YOU CAN

- Explain the client and server call sequences and what each step does.
- Resolve a host and port with `getaddrinfo` rather than hard-coded structures.
- Write a TCP client and a forking server that survive real network conditions.
- Frame messages over a stream that has no message boundaries.
- Handle partial transfers, timeouts, and a peer that disappears.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The model



`accept` returns a *new* descriptor for the connection; the listening socket stays open for the next one.

	TCP (SOCK_STREAM)	UDP (SOCK_DGRAM)
Connection	Established first	None
Delivery	Reliable, ordered	Best effort; may be lost, duplicated, reordered
Boundaries	**None — a byte stream**	Preserved per datagram
Used for	HTTP, SSH, databases	DNS, video, games

The third row is the one that surprises people, and section 4 is about nothing else.

2 Addresses

Do not fill in a `struct sockaddr_in` by hand. `getaddrinfo` resolves a host and service, handles IPv4 and IPv6 identically, and returns ready-made structures:

```
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo hints;
memset(&hints, 0, sizeof hints);
hints.ai_family   = AF_UNSPEC;      /* IPv4 or IPv6, whichever resolves */
hints.ai_socktype = SOCK_STREAM;    /* TCP */

struct addrinfo *list;
int rc = getaddrinfo("example.com", "8080", &hints, &list);
if (rc != 0) {
    fprintf(stderr, "%s\n", gai_strerror(rc));   /* not perror */
    return -1;
}

for (struct addrinfo *ai = list; ai != NULL; ai = ai->ai_next) {
    int fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
    if (fd < 0) continue;
    if (connect(fd, ai->ai_addr, ai->ai_addrlen) == 0) break;   /* success */
    close(fd);
}
freeaddrinfo(list);                  /* the list is allocated */
```

Two details: `getaddrinfo` returns its own error codes, so use `gai_strerror` rather than `perror`; and the result must be released with `freeaddrinfo`, which is chapter 19's ownership rule applied to a library.

Trying each address in turn matters. A host with both IPv6 and IPv4 addresses may be reachable on only one.

Byte order

Port numbers and addresses travel in network byte order — big-endian, as chapter 37 established. `getaddrinfo` handles it for the structures it fills, but any integer *you* put in a message needs converting:

```
uint32_t on_wire = htonl(value);
uint32_t local   = ntohl(on_wire);
```

3 The server sequence

```
int listener = socket(AF_INET6, SOCK_STREAM, 0);

int yes = 1;
setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes);

bind(listener, addr, addrlen);
listen(listener, 16);                       /* backlog */

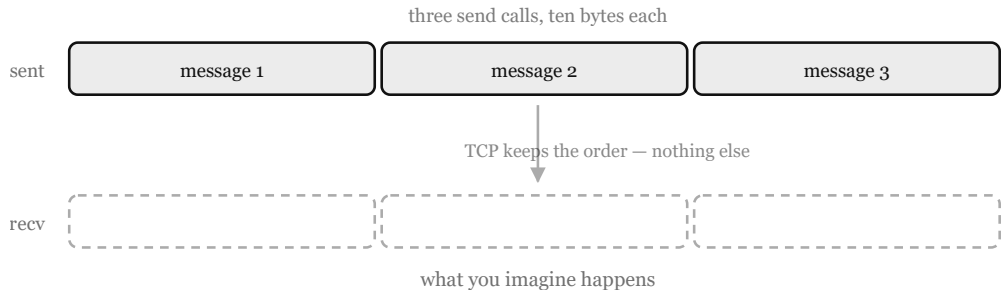
for (;;) {
    int conn = accept(listener, NULL, NULL); /* a NEW descriptor */
    handle(conn);
    close(conn);
}
```

`SO_REUSEADDR` is not optional in practice. After a server exits, the kernel keeps the address in `TIME_WAIT` for up to two minutes, and `bind` fails with "Address already in use" — the error every beginner hits on their second run. The option says it is acceptable to rebind.

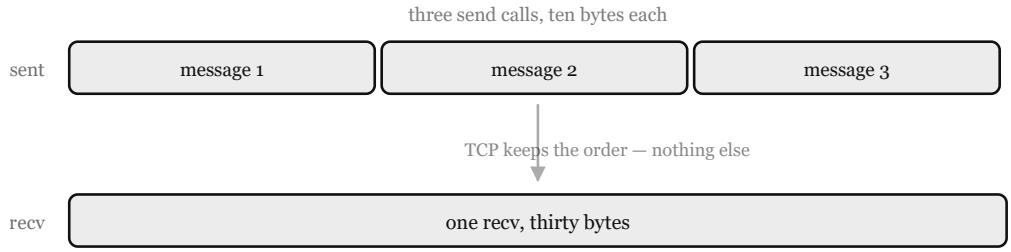
The backlog is the queue of completed connections waiting for `accept`. Beyond it, further connections wait or are refused, depending on the system.

4 Framing

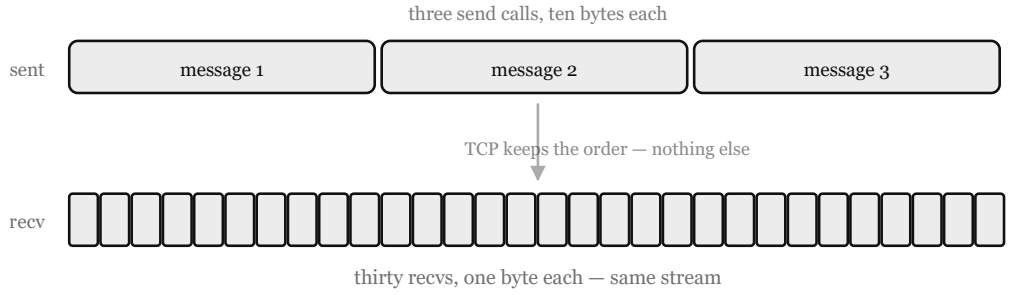
TCP delivers a stream of bytes with **no record boundaries whatsoever**. Three `send` calls of 10 bytes may arrive as one `recv` of 30, or as 30 of 1. Nothing preserves your message structure — you must impose it.



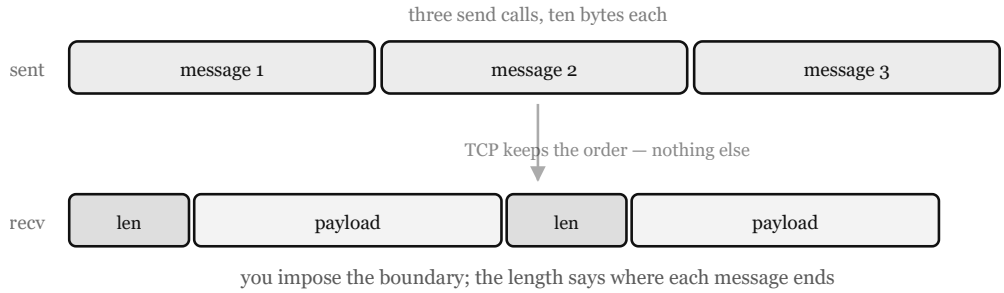
You sent three messages, so it is natural to expect three arrivals. TCP never promised that.



They may arrive as **one** `recv` of thirty bytes. Nothing is wrong and nothing is lost — the byte order is all TCP guarantees.



Or as thirty `recv` calls of one byte. Identical stream, identical bytes, and code that assumed one-send-one-recv is now broken in a way that only shows under load.



So you impose the structure yourself. A four-byte length in front of each payload lets the receiver find the end of a message regardless of how the bytes were delivered — and the length must be bounded, or you have handed an attacker the allocation size.

SCHEME	HOW	TRADE-OFF
Delimiter	Messages end with `\n`	Simple; the delimiter cannot appear in data
Length prefix	A 4-byte big-endian length, then the payload	Binary safe; must bound the length
Fixed size	Every message the same	Trivial; wasteful and inflexible

Bound the declared length. A length-prefixed protocol that trusts the prefix hands an attacker `malloc(0xFFFFFFFF)` — or, after an integer overflow, a tiny buffer followed by a huge copy. Check the length against a maximum *before* allocating. This is chapter 36's allocation-overflow bug arriving over the network.

5 Partial transfers and errors

`send` and `recv` may transfer less than requested — more often than local files, because the network is slow and buffers fill. The loop from chapter 39 is mandatory here:

```
static ssize_t send_all(int fd, const void *data, size_t n)
{
    const char *p = data;
    size_t sent = 0;
    while (sent < n) {
        ssize_t s = send(fd, p + sent, n - sent, MSG_NOSIGNAL);
        if (s < 0) {
            if (errno == EINTR) continue;
            return -1;
        }
        sent += (size_t)s;
    }
    return (ssize_t)sent;
}
```

`recv` returning 0 means the peer closed the connection in an orderly way — it is not an error and not a short read.

CONDITION	MEANS
`recv` returns 0	Peer closed; you are done
`ECONNRESET`	Peer vanished abruptly
`EPIPE` + `SIGPIPE`	You wrote to a closed connection
`ETIMEDOUT`	No response within the limit
`EINTR`	A signal arrived; retry

`SIGPIPE` deserves attention: by default, writing to a connection the peer has closed **terminates your process**. Pass `MSG_NOSIGNAL` to `send`, or ignore the signal once at startup — otherwise a client disconnecting at the wrong moment kills your server.

Timeouts

```
struct timeval tv = { .tv_sec = 5, .tv_usec = 0 };
setsockopt(fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof tv);
```

Without one, a peer that stops responding blocks you forever. A network program with no timeouts is not finished.

6 Worked example: a line server and its client

```
/* echoserver.c — a TCP server, one forked child per connection */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netdb.h>

#define BACKLOG      16
#define MAX_LINE   4096

static volatile sig_atomic_t stop = 0;

static void on_signal(int sig) { (void)sig; stop = 1; }

/* Reap children so they do not become zombies — chapter 39. */
static void on_child(int sig)
{
    (void)sig;
    int saved = errno;                      /* waitpid may change errno */
    while (waitpid(-1, NULL, WNOHANG) > 0) { }
    errno = saved;
}

static ssize_t send_all(int fd, const void *data, size_t n)
{
    const char *p = data;
    size_t sent = 0;
    while (sent < n) {
        ssize_t s = send(fd, p + sent, n - sent, MSG_NOSIGNAL);
        if (s < 0) {
            if (errno == EINTR) continue;
            return -1;
```

```
        }
        sent += (size_t)s;
    }
    return (ssize_t)sent;
}

/* Newline framing: accumulate until a '\n' appears. A stream has no
   message boundaries, so this loop is what creates them. */
static void serve_connection(int fd)
{
    char    buffer[MAX_LINE];
    size_t  used = 0;
    long    lines = 0;

    struct timeval tv = { .tv_sec = 30, .tv_usec = 0 };
    setsockopt(fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof tv);

    const char *greeting = "ready\n";
    if (send_all(fd, greeting, strlen(greeting)) < 0) return;

    for (;;) {
        ssize_t r = recv(fd, buffer + used, sizeof buffer - used, 0);

        if (r == 0) {
            break;                          /* orderly close: not an error */
        }
        if (r < 0) {
            if (errno == EINTR) continue;
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                send_all(fd, "timeout\n", 8);
            }
            break;
        }
        used += (size_t)r;

        /* One recv may contain zero, one, or several complete lines. */
        char *start = buffer;
        char *nl;
        while ((nl = memchr(start, '\n', (size_t)(buffer + used - start)))) {
```

```
            size_t len = (size_t)(nl - start);
            lines++;

            if (len == 4 && memcmp(start, "quit", 4) == 0) {
                send_all(fd, "bye\n", 4);
                return;
            }

            char out[MAX_LINE + 32];
            int n = snprintf(out, sizeof out, "%ld: %.*s\n",
                             lines, (int)len, start);
            if (n > 0 && send_all(fd, out, (size_t)n) < 0) return;

            start = nl + 1;
        }

        /* Move the incomplete remainder to the front. */
        size_t leftover = (size_t)(buffer + used - start);
        memmove(buffer, start, leftover);
        used = leftover;

        if (used == sizeof buffer) {        /* a line longer than the buffer */
            send_all(fd, "line too long\n", 14);
            return;
        }
    }
}

int main(int argc, char *argv[])
{
    const char *port = (argc > 1) ? argv[1] : "8080";

    struct sigaction sa;
    memset(&sa, 0, sizeof sa);
    sa.sa_handler = on_signal;
    sigaction(SIGINT,  &sa, NULL);
    sigaction(SIGTERM, &sa, NULL);

    memset(&sa, 0, sizeof sa);
```

```
    sa.sa_handler = on_child;
    sa.sa_flags   = SA_RESTART | SA_NOCLDSTOP;
    sigaction(SIGCHLD, &sa, NULL);

    signal(SIGPIPE, SIG_IGN);               /* a closed peer must not kill us */

    struct addrinfo hints;
    memset(&hints, 0, sizeof hints);
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;         /* for bind */

    struct addrinfo *list;
    int rc = getaddrinfo(NULL, port, &hints, &list);
    if (rc != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rc));
        return EXIT_FAILURE;
    }

    int listener = -1;
    for (struct addrinfo *ai = list; ai != NULL; ai = ai->ai_next) {
        listener = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (listener < 0) continue;

        int yes = 1;
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes);

        if (bind(listener, ai->ai_addr, ai->ai_addrlen) == 0) break;
        close(listener);
        listener = -1;
    }
    freeaddrinfo(list);

    if (listener < 0) {
        perror("bind");
        return EXIT_FAILURE;
    }
    if (listen(listener, BACKLOG) < 0) {
        perror("listen");
```

```
        return EXIT_FAILURE;
    }

    printf("listening on port %s; Ctrl-C to stop\n", port);

    while (!stop) {
        int conn = accept(listener, NULL, NULL);
        if (conn < 0) {
            if (errno == EINTR) continue;   /* a signal, not a failure */
            perror("accept");
            break;
        }

        pid_t pid = fork();
        if (pid == 0) {
            close(listener);                /* the child does not need it */
            serve_connection(conn);
            close(conn);
            _exit(0);                       /* _exit in a child */
        }
        close(conn);                        /* the parent does not need it */
        if (pid < 0) perror("fork");
    }

    close(listener);
    puts("\nstopped");
    return EXIT_SUCCESS;
}
```

```
/* echoclient.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netdb.h>
```

```
static ssize_t send_all(int fd, const void *data, size_t n)
{
    const char *p = data;
    size_t sent = 0;
    while (sent < n) {
        ssize_t s = send(fd, p + sent, n - sent, MSG_NOSIGNAL);
        if (s < 0) { if (errno == EINTR) continue; return -1; }
        sent += (size_t)s;
    }
    return (ssize_t)sent;
}

int main(int argc, char *argv[])
{
    const char *host = (argc > 1) ? argv[1] : "localhost";
    const char *port = (argc > 2) ? argv[2] : "8080";

    struct addrinfo hints;
    memset(&hints, 0, sizeof hints);
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    struct addrinfo *list;
    int rc = getaddrinfo(host, port, &hints, &list);
    if (rc != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rc));
        return EXIT_FAILURE;
    }

    int fd = -1;
    for (struct addrinfo *ai = list; ai != NULL; ai = ai->ai_next) {
        fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (fd < 0) continue;
        if (connect(fd, ai->ai_addr, ai->ai_addrlen) == 0) break;
        close(fd);
        fd = -1;
    }
    freeaddrinfo(list);
```

```
    if (fd < 0) { perror("connect"); return EXIT_FAILURE; }

    struct timeval tv = { .tv_sec = 10, .tv_usec = 0 };
    setsockopt(fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof tv);

    char line[1024];
    while (fgets(line, sizeof line, stdin) != NULL) {
        if (send_all(fd, line, strlen(line)) < 0) { perror("send"); break; }

        char reply[2048];
        ssize_t r = recv(fd, reply, sizeof reply - 1, 0);   /* one recv per line: simplified
*/
        if (r < 0)  { perror("recv"); break; }
        if (r == 0) { puts("server closed the connection"); break; }
        reply[r] = '\0';
        fputs(reply, stdout);
    }

    close(fd);
    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -D_POSIX_C_SOURCE=200809L -Wall -Wextra -g -o echoserver echoserver.c
gcc -std=c17 -D_POSIX_C_SOURCE=200809L -Wall -Wextra -g -o echoclient echoclient.c

./echoserver 8080 &
printf 'hello\nworld\nquit\n' | ./echoclient localhost 8080
```

```
ready
1: hello
2: world
bye
```

Prove that a stream has no message boundaries

```
{ printf 'he'; sleep 0.3; printf 'llo\nwor'; sleep 0.3; printf 'ld\nquit\n'; } \
    | nc localhost 8080
```

The bytes arrive in three pieces that do not align with lines, and the server still reports exactly two numbered lines. That is the framing loop working: it accumulates, extracts every complete line, and keeps the remainder for the next `recv`. Remove the `memmove` and the partial `"wor"` is lost.

The other direction is equally real: send a thousand short lines quickly and several will arrive in one `recv`. The inner `while` over `memchr` is why that works.

Test the failure modes

```
./echoclient localhost 9999            # nothing listening
./echoclient no.such.host.invalid 8080 # resolution fails
nc localhost 8080                      # connect with netcat, type, Ctrl-C
head -c 1000000 /dev/zero | tr '\0' 'x' | ./echoclient localhost 8080
```

TEST	EXPECTED
No listener	`connect: Connection refused`
Bad hostname	`getaddrinfo:` with a name-resolution message
Client killed mid-session	Server child exits; parent keeps serving
A line longer than the buffer	`line too long`, connection closed — not an overflow
Two clients at once	Both served; one child each
Thirty seconds of silence	`timeout`, connection closed

The three defences that keep the server alive

`SIGPIPE` **ignored.** Remove `signal(SIGPIPE, SIG_IGN)`, connect with `nc`, and kill the client while the server is replying. The server process dies — a client can terminate your service by disconnecting at the right moment.

`SIGCHLD` **reaped.** Remove the handler and run a hundred connections in a loop, then check `ps -el | grep defunct`. Every one is still there.

`SO_REUSEADDR`**.** Remove it, stop the server, and restart immediately: `bind: Address already in use` for up to two minutes.

Watch the bytes

```
sudo tcpdump -i lo -A 'port 8080'
```

Seeing your own protocol on the wire is worth doing once. It also makes concrete that nothing in TCP marks where one message ends — only your newline does.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Assuming one `send` equals one `recv`	Messages split or merged	Frame explicitly and accumulate.
Ignoring a short `send`	Silent truncation	Loop with `send_all`.
Treating `recv` == 0 as an error	Misreports an orderly close	Zero means the peer finished.
No `SO_REUSEADDR`	`Address already in use` on restart	Set it before `bind`.
Not ignoring `SIGPIPE`	A disconnecting client kills the server	`SIG_IGN` or `MSG_NOSIGNAL`.
Not reaping children	Zombies accumulate	`SIGCHLD` handler with `WNOHANG`.
Trusting a length prefix	Huge allocation or overflow	Bound it before allocating.
No timeouts	A silent peer blocks you forever	`SO_RCVTIMEO` and `SO_SNDTIMEO`.
Sending integers without conversion	Works only between identical machines	`htonl` / `ntohl`.

8 Check yourself

Why must a TCP protocol define its own message boundaries?

Because TCP delivers a byte stream, not records. The bytes from several `send` calls may arrive in one `recv`, and one `send` may arrive split across many. Nothing in the protocol marks where a message ends, so the application must impose a delimiter, a length prefix, or a fixed size — and accumulate until a complete message is present.

What does `recv` returning 0 mean, and how does it differ from a negative return?

Zero means the peer performed an orderly shutdown — there will be no more data, and this is normal. A negative return means an error, with the reason in `errno`: `ECONNRESET` if the peer vanished, `EAGAIN` on a timeout, `EINTR` if a signal arrived and you should retry. Treating zero as an error misreports a clean disconnection.

Why is `SO_REUSEADDR` effectively mandatory on a server?

Because after a connection closes the kernel keeps the address in `TIME_WAIT` for up to two minutes to catch stray packets, and `bind` on that address fails during the wait. Without the option, restarting a server means waiting; with it, the kernel permits the rebind.

How can a client disconnecting kill a server that never crashed?

Writing to a connection whose peer has closed raises `SIGPIPE`, whose default action is to terminate the process. A client that disconnects while the server is mid-reply therefore kills it. Ignore the signal at startup, or pass `MSG_NOSIGNAL` to `send` so the error arrives as `EPIPE` instead.

Why must a length-prefixed protocol bound the declared length?

Because the prefix comes from the peer, who may be hostile. An unbounded value becomes an enormous allocation — a denial of service — or, if the length is multiplied by an element size that overflows, a small buffer followed by a huge copy. Validate the length against a maximum before allocating anything.

9 Where this leads

This server forks a process per connection, which is simple and costs a few hundred kilobytes each — fine for tens of clients, hopeless for tens of thousands. Chapter 47 rewrites it as a single-threaded event loop using `epoll` and measures the difference. Chapter 41 first covers the other approach: threads sharing one address space.

ADVANCED · CHAPTER 41

Concurrency and the C Memory Model

Chapter 39's processes were isolated. Threads share one address space, which makes communication free and correctness hard: two threads touching one variable without synchronization is undefined behavior, and the resulting bugs appear only under load.

BY THE END OF THIS CHAPTER YOU CAN

- Create and join threads with both C11 and POSIX interfaces.
- Recognize a data race and explain why it is undefined behavior rather than merely wrong.
- Protect shared state with a mutex, and coordinate with a condition variable.
- Use atomics, and say what the memory model guarantees.
- Identify non-reentrant library functions and avoid deadlock.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Threads

C11 added `<threads.h>`; POSIX has had pthreads since 1995. The C11 interface is cleaner but less widely implemented — notably absent from glibc until recently and from macOS — so pthreads remains what production code uses.

```
/* POSIX */
#include <pthread.h>

static void *worker(void *arg)
{
    int id = *(int *)arg;
    printf("thread %d\n", id);
    return NULL;
}

pthread_t t;
int id = 1;
pthread_create(&t, NULL, worker, &id);
pthread_join(t, NULL);                /* wait for it to finish */
```

All threads share the heap, globals, and file descriptors. Each has its own stack, so locals are private. That asymmetry is the whole subject: anything reachable from more than one thread needs protection.

> **Passing the address of a loop variable.** `pthread_create(&t[i], NULL, worker, &i)` passes the address of a variable that keeps changing — every thread reads whatever `i` holds when it happens to run, and after the loop it may not exist at all. Give each thread its own object: an element of an array, or a small allocated structure.

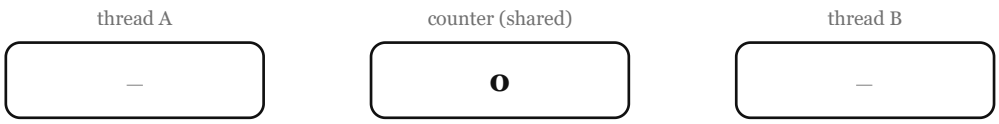
2 Data races

```
int counter = 0;                        /* shared */

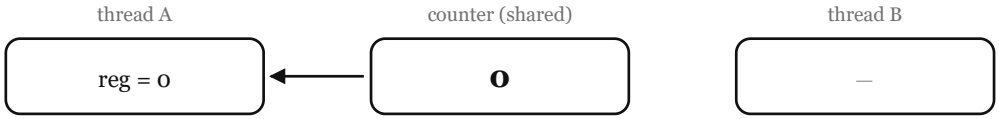

void *increment(void *arg)
{
    for (int i = 0; i < 1000000; i++) {
        counter++;                      /* a data race */
    }
    return NULL;
}
```

Run this on four threads and the result is not 4 000 000. `counter++` is three operations — load, add, store — and two threads interleaving them lose updates.

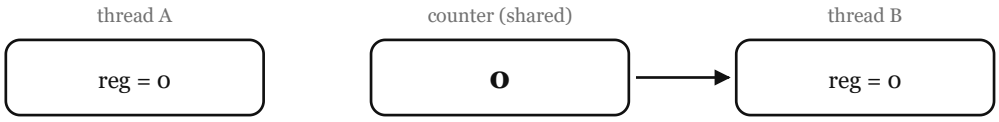
The C standard's position is stronger than "you get a wrong number". Two threads accessing the same object without synchronization, where at least one writes, is a **data race and therefore undefined behavior**. The compiler may keep the variable in a register, reorder the accesses, or assume the race cannot happen. Reasoning about which interleaving you got is already the wrong question.



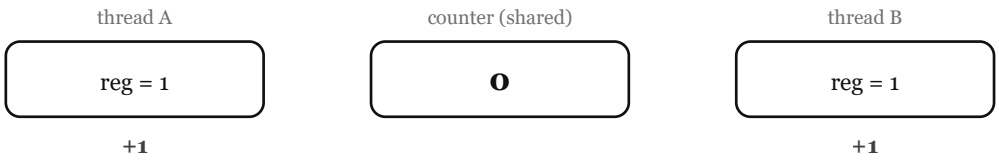
`counter` is 0. Both threads are about to run `counter++`.



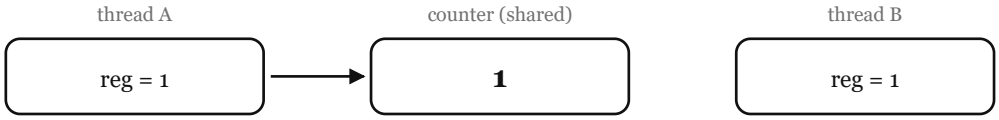
`counter++` is not one operation. It is three: **load**, add, store. Thread A loads 0 into a register.



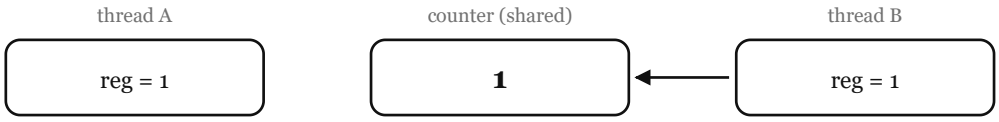
Before A gets any further, thread B loads — and reads 0 as well, because A has not stored anything yet.



Each thread adds one to *its own copy*. Neither can see the other.



A stores its result. `counter` is 1, which is correct so far.



two increments, one result

B stores its result over the top. `counter` is still 1 — two increments happened and one was lost. Run this four million times and the total is not four million.

And, as chapter 38 said: `volatile` does not fix this. It forces the accesses to happen; it makes nothing atomic.

3 Mutexes

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;


pthread_mutex_lock(&lock);
counter++;                              /* the critical section */
pthread_mutex_unlock(&lock);
```

A mutex guarantees that only one thread is inside the critical section at a time, and — equally important — establishes the ordering that makes one thread's writes visible to the next.

RULE	WHY
Keep critical sections short	Every other thread is waiting
Never return while holding the lock	It stays locked forever
Always lock multiple mutexes in the same order	Otherwise: deadlock
Do not call unknown code while holding a lock	It may take the same lock

Deadlock

```
/* thread 1 */            /* thread 2 */
lock(A);                  lock(B);
lock(B);   /* waits */    lock(A);   /* waits */
```

Each holds what the other needs. Nothing times out; the program simply stops. The standard prevention is a **lock ordering**: assign every mutex a rank and always acquire in increasing rank. If every thread obeys it, a cycle is impossible.

4 Condition variables

A mutex protects data. A condition variable lets a thread *wait* for the data to reach a state — without spinning.

```
pthread_mutex_lock(&lock);
while (queue_is_empty(&q)) {           /* while, not if */
    pthread_cond_wait(&not_empty, &lock);
}
item = queue_pop(&q);
pthread_mutex_unlock(&lock);
```

`pthread_cond_wait` atomically releases the mutex and sleeps; on waking it reacquires the mutex before returning. That atomicity is the point — without it, a signal arriving between the test and the sleep would be missed.

The loop is mandatory. A waiting thread can wake without the condition being true: a *spurious wakeup*, permitted by the standard, or another thread taking the item first. `if` instead of `while` produces a bug that appears once a month under load.

5 Atomics and the memory model

```
#include <stdatomic.h>

atomic_int counter = 0;
atomic_fetch_add(&counter, 1);         /* indivisible */
counter++;                              /* also atomic for an atomic type */
```

For a single counter, an atomic is faster than a mutex — one instruction rather than a system call in the contended case. For anything involving two related values, a mutex is required: two atomic operations are not one atomic operation.

Memory ordering

Modern processors and compilers reorder memory operations. The C memory model defines what one thread is guaranteed to observe of another.

ORDER	GUARANTEES	COST
`memory_order_seq_cst`	A single total order, visible to all — the default	Highest
`memory_order_acquire` / `release`	Pairs up to order everything before a release with everything after the matching acquire	Moderate
`memory_order_relaxed`	Atomicity only; no ordering at all	Lowest

Use the default. Relaxed ordering is correct only in narrow cases — a statistics counter nobody reads until the end — and reasoning about it is genuinely difficult. A wrong choice produces a bug that appears on one processor architecture and not another.

`_Thread_local`

```
_Thread_local int per_thread_errors = 0;   /* one instance per thread */
```

Each thread gets its own copy, so no synchronization is needed. This is how `errno` is implemented on a threaded system.

6 Reentrancy

Several standard library functions keep static state and cannot be called from two threads at once. Chapter 14's `strtok` was the first example; there are others.

NOT THREAD-SAFE	USE INSTEAD
`strtok`	`strtok_r`
`localtime`, `gmtime`	`localtime_r`, `gmtime_r`
`asctime`, `ctime`	`asctime_r`, `ctime_r`
`rand`	`rand_r`, or a per-thread generator
`strerror`	`strerror_r`
`getenv` with concurrent `setenv`	Read the environment once at startup

The `_r` suffix means *reentrant*: the caller supplies the buffer, so there is no shared state. `errno` itself is thread-local on every modern system, which is why chapter 27's advice still holds here.

7 Worked example: a producer-consumer pipeline

```
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <stdatomic.h>
#include <pthread.h>
```

```
#include <unistd.h>
#include <time.h>

#define QUEUE_CAPACITY 8
#define PRODUCERS      3
#define CONSUMERS      2
#define PER_PRODUCER   2000

/* ---------- 1. the race, demonstrated ---------- */

static long   racy_counter   = 0;
static atomic_long atomic_counter = 0;
static long   guarded_counter = 0;
static pthread_mutex_t guard = PTHREAD_MUTEX_INITIALIZER;

#define BUMPS 200000

static void *bump_all(void *arg)
{
    (void)arg;
    for (int i = 0; i < BUMPS; i++) {
        racy_counter++;                              /* DATA RACE */
        atomic_fetch_add(&atomic_counter, 1);        /* atomic     */
        pthread_mutex_lock(&guard);
        guarded_counter++;                           /* protected  */
        pthread_mutex_unlock(&guard);
    }
    return NULL;
}

/* ---------- 2. a bounded blocking queue ---------- */

typedef struct {
    int             items[QUEUE_CAPACITY];
    size_t          head, tail, count;
    bool            closed;
    pthread_mutex_t lock;
    pthread_cond_t  not_empty;
    pthread_cond_t  not_full;
```

```
} Queue;

static void queue_init(Queue *q)
{
    q->head = q->tail = q->count = 0;
    q->closed = false;
    pthread_mutex_init(&q->lock, NULL);
    pthread_cond_init(&q->not_empty, NULL);
    pthread_cond_init(&q->not_full, NULL);
}

static void queue_destroy(Queue *q)
{
    pthread_mutex_destroy(&q->lock);
    pthread_cond_destroy(&q->not_empty);
    pthread_cond_destroy(&q->not_full);
}

static void queue_push(Queue *q, int value)
{
    pthread_mutex_lock(&q->lock);
    while (q->count == QUEUE_CAPACITY && !q->closed) {
        pthread_cond_wait(&q->not_full, &q->lock);    /* while, not if */
    }
    if (!q->closed) {
        q->items[q->tail] = value;
        q->tail = (q->tail + 1) % QUEUE_CAPACITY;
        q->count++;
        pthread_cond_signal(&q->not_empty);
    }
    pthread_mutex_unlock(&q->lock);
}

static bool queue_pop(Queue *q, int *out)
{
    pthread_mutex_lock(&q->lock);
    while (q->count == 0 && !q->closed) {
        pthread_cond_wait(&q->not_empty, &q->lock);
    }
```

```
    if (q->count == 0) {                            /* closed and drained */
        pthread_mutex_unlock(&q->lock);
        return false;
    }
    *out = q->items[q->head];
    q->head = (q->head + 1) % QUEUE_CAPACITY;
    q->count--;
    pthread_cond_signal(&q->not_full);
    pthread_mutex_unlock(&q->lock);
    return true;
}

static void queue_close(Queue *q)
{
    pthread_mutex_lock(&q->lock);
    q->closed = true;
    pthread_cond_broadcast(&q->not_empty);          /* wake everyone */
    pthread_cond_broadcast(&q->not_full);
    pthread_mutex_unlock(&q->lock);
}

/* ---------- 3. the workers ---------- */

typedef struct { Queue *q; int id; } Worker;

static atomic_long produced = 0;
static atomic_long consumed = 0;
static atomic_long checksum = 0;

_Thread_local long my_items = 0;                    /* private per thread */

static void *producer(void *arg)
{
    Worker *w = arg;                                /* its own object */
    for (int i = 0; i < PER_PRODUCER; i++) {
        int value = w->id * 1000000 + i;
        queue_push(w->q, value);
        atomic_fetch_add(&produced, 1);
        my_items++;
```

```
    }
    printf("  producer %d finished, %ld items (thread-local count)\n",
           w->id, my_items);
    return NULL;
}

static void *consumer(void *arg)
{
    Worker *w = arg;
    int value;
    while (queue_pop(w->q, &value)) {
        atomic_fetch_add(&consumed, 1);
        atomic_fetch_add(&checksum, value % 97);
        my_items++;
    }
    printf("  consumer %d finished, %ld items\n", w->id, my_items);
    return NULL;
}

int main(void)
{
    puts("== 1. three counters, one loop, four threads ==");
    pthread_t bumpers[4];
    for (int i = 0; i < 4; i++) pthread_create(&bumpers[i], NULL, bump_all, NULL);
    for (int i = 0; i < 4; i++) pthread_join(bumpers[i], NULL);

    long expected = 4L * BUMPS;
    printf("  expected          : %ld\n", expected);
    printf("  racy (unprotected): %ld  %s\n", racy_counter,
           racy_counter == expected ? "(lucky this run)" : "<-- updates lost");
    printf("  atomic            : %ld\n", atomic_load(&atomic_counter));
    printf("  mutex-protected   : %ld\n", guarded_counter);
    puts("  the racy result varies between runs; that is what UB looks like");

    puts("\n== 2. producer-consumer with a bounded queue ==");
    Queue q;
    queue_init(&q);

    pthread_t pt[PRODUCERS], ct[CONSUMERS];
```

```
    Worker    pw[PRODUCERS], cw[CONSUMERS];

    struct timespec t0;
    clock_gettime(CLOCK_MONOTONIC, &t0);

    for (int i = 0; i < CONSUMERS; i++) {
        cw[i] = (Worker){ .q = &q, .id = i };         /* its own object */
        pthread_create(&ct[i], NULL, consumer, &cw[i]);
    }
    for (int i = 0; i < PRODUCERS; i++) {
        pw[i] = (Worker){ .q = &q, .id = i };
        pthread_create(&pt[i], NULL, producer, &pw[i]);
    }

    for (int i = 0; i < PRODUCERS; i++) pthread_join(pt[i], NULL);
    queue_close(&q);                                  /* now the consumers end */
    for (int i = 0; i < CONSUMERS; i++) pthread_join(ct[i], NULL);

    struct timespec t1;
    clock_gettime(CLOCK_MONOTONIC, &t1);
    double secs = (double)(t1.tv_sec - t0.tv_sec)
                + (double)(t1.tv_nsec - t0.tv_nsec) / 1e9;

    printf("  produced %ld, consumed %ld, checksum %ld, %.3f s\n",
           atomic_load(&produced), atomic_load(&consumed),
           atomic_load(&checksum), secs);
    printf("  queue capacity was %d, so producers blocked whenever\n",
           QUEUE_CAPACITY);
    puts("  consumers fell behind — that is back-pressure, for free");

    queue_destroy(&q);

    puts("\n== 3. reentrancy ==");
    time_t now = time(NULL);
    struct tm safe;
    localtime_r(&now, &safe);                         /* caller's buffer */
    char stamp[64];
    strftime(stamp, sizeof stamp, "%H:%M:%S", &safe);
    printf("  localtime_r : %s\n", stamp);
```

```
    puts("  localtime() returns a pointer to a shared static buffer;");
    puts("  two threads calling it would overwrite each other's result");

    char text[] = "alpha,beta,gamma";
    char *save = NULL;
    printf("  strtok_r    :");
    for (char *tok = strtok_r(text, ",", &save); tok != NULL;
         tok = strtok_r(NULL, ",", &save)) {
        printf(" [%s]", tok);
    }
    puts("\n  the save pointer replaces strtok's hidden static state");

    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -pthread -o concurrent concurrent.c
./concurrent
```

`-pthread` rather than `-lpthread`: it sets the required preprocessor definitions as well as linking the library.

Let the sanitizer name the race

```
gcc -std=c17 -Wall -Wextra -g -pthread -fsanitize=thread -o concurrent_tsan concurrent.c
./concurrent_tsan
```

```
WARNING: ThreadSanitizer: data race (pid=12345)
  Write of size 8 at 0x55a... by thread T2:
    #0 bump_all concurrent.c:28

  Previous write of size 8 at 0x55a... by thread T1:
    #0 bump_all concurrent.c:28

  Location is global 'racy_counter' of size 8
```

It names the variable, both threads, and the line. ThreadSanitizer reports races that did not manifest on this run — it reasons about the synchronization, not about what happened to

interleave. That is what makes it worth the five- to fifteen-fold slowdown. Note that it cannot be combined with AddressSanitizer; run them in separate builds.

Run the racy counter ten times

```
for i in $(seq 10); do ./concurrent | grep 'racy'; done
```

Ten different numbers, all below 800 000. Every lost update is one thread's store overwriting another's. Then raise `BUMPS` and watch the loss grow — and note that at low contention the racy counter is sometimes exactly right, which is precisely why these bugs reach production.

Cause a deadlock deliberately

```
static pthread_mutex_t a = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t b = PTHREAD_MUTEX_INITIALIZER;

static void *ab(void *_) { (void)_;
    pthread_mutex_lock(&a); sleep(1); pthread_mutex_lock(&b);
    pthread_mutex_unlock(&b); pthread_mutex_unlock(&a); return NULL; }

static void *ba(void *_) { (void)_;
    pthread_mutex_lock(&b); sleep(1); pthread_mutex_lock(&a);
    pthread_mutex_unlock(&a); pthread_mutex_unlock(&b); return NULL; }
```

The program hangs with no message. Attach a debugger to see why:

```
gdb -p $(pgrep concurrent)
(gdb) thread apply all bt
```

Both threads sit in `pthread_mutex_lock`, each holding what the other wants. Now make both functions lock `a` then `b` — the same order — and the deadlock becomes impossible. That single rule is the whole prevention.

Break the condition-variable loop

Change `while (q->count == 0 && !q->closed)` to `if`. The program usually still works, and occasionally a consumer wakes to find the queue empty, takes that for the closed signal, and quits early; if both consumers do, the producers block forever on a full queue. Run it a hundred times to see it. This is the shape of a concurrency bug: correct almost always, wrong

rarely, and impossible to reproduce on demand — which is why the rule is followed unconditionally rather than tested.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Sharing a variable without a lock	Data race; undefined behavior	Mutex, or an atomic type.
`volatile` as synchronization	Still a race	`<stdatomic.h>` or a mutex.
`if` instead of `while` around `cond_wait`	Proceeds on a spurious wakeup	Always loop on the predicate.
Locking two mutexes in different orders	Deadlock, silent	A global lock ordering.
Returning while holding a lock	Permanently locked	One exit, or unlock on every path.
Passing `&i` from a loop to a thread	Every thread sees the same changing value	Give each its own object.
`strtok` or `localtime` from threads	Threads corrupt each other's state	The `_r` variants.
Relaxed memory order without a reason	Works on x86, fails on ARM	Default to sequential consistency.
Not joining or detaching a thread	Resource leak	`pthread_join` or `pthread_detach`.

9 Check yourself

Why is a data race undefined behavior rather than just a wrong result?

Because the standard withdraws all requirements when two threads access an object concurrently and at least one writes. The compiler may keep the value in a register, reorder the accesses, or optimize on the assumption that the race cannot occur. Reasoning about which interleaving you got presumes a guarantee the language does not give.

Why does `volatile` not make `counter++` thread-safe?

Because it only forces each read and write to touch memory. The increment is still three separate operations — load, add, store — and another thread can act between them. `volatile` provides no atomicity and no ordering relative to other variables. Use an atomic type or a mutex.

Why must `pthread_cond_wait` be called inside a `while` loop?

Because waking does not prove the condition holds. The standard permits spurious wakeups, and another thread may consume the item between the signal and your reacquiring the mutex. Re-testing the predicate after every wake is the only correct pattern; `if` produces a bug that surfaces rarely and under load.

What single rule prevents deadlock between two mutexes?

A consistent global acquisition order: rank every mutex and always lock in increasing rank. Deadlock requires a cycle in the wait-for graph, and a total order makes a cycle impossible. It must be followed by every thread without exception, since one violation reintroduces the possibility.

What does the `_r` suffix mean, and why do those functions exist?

Reentrant: the caller supplies the buffer or the state, so the function keeps none of its own. The originals — `strtok`, `localtime`, `strerror` — return pointers to shared static storage, so two threads calling them overwrite each other's results. The `_r` variants move that state into the caller and become safe.

10 Where this leads

That completes the language itself, the standard library, the operating system interface, and concurrency — everything C can express. Chapter 42 closes the Advanced level by turning outward: why C has the shape it does, and how to read the excellent C that other people have written.

ADVANCED · CHAPTER 42

Reading Excellent C

You have written C for forty-one chapters. Professionals read far more than they write, and the best available teacher is code that has survived twenty years of production use. This chapter is about reading it — and about why the language has the shape it does.

BY THE END OF THIS CHAPTER YOU CAN

- Explain C's oddities historically rather than memorizing them as rules.
- Say why undefined behavior exists and what the standard was trading away.
- Orient yourself in an unfamiliar codebase of a hundred thousand lines.
- Reconstruct a module's ownership and error conventions from the code alone.
- Judge which idioms are worth borrowing and which are historical accidents.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Why C looks the way it does

Chapter 2 placed C on a PDP-11 in 1972: a machine with 64 KB of address space, whose compiler had to fit in a fraction of it. Nearly every feature that seems strange is explicable from that context, and knowing the reason converts a rule you memorize into one you can reason about.

ODDITY	REASON
Arrays decay to pointers	Copying an array on every call was unaffordable. Passing an address was one word.
Strings end in `'\0'`	A length prefix costs a byte per string and caps the length. On the PDP-11 a byte load set the condition codes, so a copy loop could stop on zero without a separate comparison.
No bounds checking	A check per access was a measurable fraction of runtime on a machine executing a few hundred thousand instructions per second.
Declaration mimics use	`int *p` means "`*p` is an int". Elegant in principle; the reason chapter 16 needed a rule.
The preprocessor is textual	It was a separate program bolted on, not part of the language.
`int` has no fixed size	C had to compile for 16-bit, 18-bit, and 36-bit machines.
Integer promotion	The hardware had no sub-word arithmetic.

These were not oversights. They were the correct engineering decisions for the constraints of 1972, and the reason a language designed for a minicomputer still runs every operating system kernel in production is that most of them turned out to age well.

2 Why undefined behavior exists

Chapter 36 treated undefined behavior as a hazard. It is also a deliberate design decision, and understanding the trade explains why it will never be removed.

Consider signed overflow. The committee could have required wrapping. Two costs:

- Machines that trapped on overflow, or used sign-magnitude, would need extra instructions on every arithmetic operation to produce the mandated behavior.
- The optimizer could no longer assume `i + 1 > i`, which is what allows it to promote a loop counter to a register and vectorize.

Leaving it undefined pushes the cost onto the one program in a thousand that overflows, instead of onto every program. That is the bargain C makes throughout: **maximum performance on the correct program, no guarantees for the incorrect one**.

Whether that bargain was right is a live argument — Rust exists largely to answer "no". But it is a coherent position, not an accident, and reading C written by people who understood it is the fastest way to internalize where the edges are.

3 Where to read

PROJECT	SIZE	READ IT FOR
musl libc	~80k lines	The standard library implemented cleanly. Start here: each function is small, self-contained, and something you already know the specification of.
SQLite	~150k lines	Extreme portability and testing discipline. Its test suite is many times the size of the code.
Lua	~25k lines	A complete language implementation small enough to read entirely. Exceptional clarity.
Redis	~150k lines	Data structures and an event loop under real load. `sds.c` and `dict.c` are directly comparable to chapters 19 and 33.
git	~250k lines	A large codebase that grew organically — including the parts that show it.
Linux kernel	millions	Not for reading whole. Pick one driver, or `include/linux/list.h`, which is chapter 32's sentinel list.

Start with musl. Read `src/string/strlen.c`, then `src/stdlib/qsort.c`. You know exactly what these must do, so all your attention goes to *how* — which is the point.

4 How to read a codebase

Reading top to bottom does not work past a few thousand lines. Work outside in.

1. **Build it first.** A codebase you cannot compile is one you cannot experiment with. Read the README and the build files before any source.
2. **Find the entry point.** `main`, or the exported functions in the public header. The header is the intended summary of what the code does.
3. **Map the directories.** Names usually reveal the architecture. Skim it before diving.
4. **Pick one path and follow it.** One command, one request, one function call, end to end. Breadth-first reading of a large codebase produces nothing.
5. **Run it under a debugger.** A breakpoint and a backtrace answer in seconds what an hour of reading may not.

```
grep -rn "function_name" --include='*.c' --include='*.h' .
ctags -R . && vim -t function_name        # jump to a definition
cloc .                                     # size and language breakdown
git log --oneline -- path/to/file.c        # why this file looks like this
git log -S "some_symbol" --oneline         # when a symbol appeared
gdb ./prog  → break f  → run  → bt         # who calls this, actually
```

A language server — clangd with a `compile_commands.json` — gives go-to-definition and find-all-references in an editor, and is worth the setup on anything you will read more than once. Generate the file with `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON` or `bear -- make`.

5 Reconstructing the conventions

Every C project invents answers to the same questions, because the language supplies none. Before changing anything, find the answers — they are rarely documented and always consistent.

QUESTION	WHERE TO LOOK
How are errors reported?	The return type of any function that can fail. An `int` code, a `bool` with an output parameter, or `NULL`?
Who frees what?	Function names: a `_create`/`_destroy` pair, or a caller-supplied buffer. Check three call sites.
What are the naming rules?	Prefixes (`sqlite3_`, `redis_`), `snake_case`, `_t` suffixes for types.
Is cleanup centralized?	Search for `goto cleanup` or `goto fail`. Its presence tells you chapter 27's idiom is the house style.
How is memory allocated?	Many projects wrap `malloc` — `sqlite3_malloc`, `zmalloc` — for accounting or failure injection.
What is public?	The installed headers. Everything else is internal, whatever its visibility.

Answer these six questions and you can make a change that looks like it belongs. Ignore them and your patch will be rejected on style before anyone reads the logic.

6 Idioms worth borrowing, and not

BORROW	WHY
`goto cleanup`	One exit path; adding a resource is one line. Chapter 27.
Opaque handles	The implementation stays changeable. Chapters 28, 33.
A prefix on every public name	C has no namespaces; the prefix is the substitute.
A wrapped allocator	Enables accounting, limits, and failure injection in tests.
Intrusive lists	The kernel's `list_head`: no allocation per link, no special cases.
`static` on everything not exported	Smaller interface, better optimization. Chapter 28.

DO NOT BORROW	WHY NOT
Clever macros that build control flow	Undebuggable; the compiler sees only the expansion.
Hungarian notation	Encodes the type in the name, where the compiler already knows it.
Single-letter names outside loops	Made sense when identifiers were expensive. They are not.
Deep `#ifdef` nesting	Only one configuration is ever compile-checked. Chapter 28.
`void main()`, K&R parameter lists	Pre-standard; kept only for compatibility.
Manual loop unrolling	The compiler does it better, and the source becomes unreadable.

Age is not quality. A twenty-year-old codebase contains decisions that were right in 2005, workarounds for compilers nobody uses, and genuine mistakes nobody dared touch. Read critically.

7 Worked example: reading musl's `strlen`

Take one function you already understand completely and see what twenty years of care look like.

```
git clone --depth 1 git://git.musl-libc.org/musl
cd musl
wc -l src/string/strlen.c
cat src/string/strlen.c
```

The implementation is roughly this:

```
#include <string.h>
#include <stdint.h>
#include <limits.h>

#define ALIGN (sizeof(size_t))
#define ONES ((size_t)-1/UCHAR_MAX)                  /* 0x0101...01 */
#define HIGHS (ONES * (UCHAR_MAX/2+1))               /* 0x8080...80 */
#define HASZERO(x) ((x)-ONES & ~(x) & HIGHS)

size_t strlen(const char *s)
{
    const char *a = s;
    const size_t *w;

    for (; (uintptr_t)s % ALIGN; s++) {
        if (!*s) return s-a;                         /* byte-at-a-time
                                                        until aligned */
    }
    for (w = (const void *)s; !HASZERO(*w); w++);    /* word at a time */

    for (s = (const void *)w; *s; s++);              /* find the exact byte */
    return s-a;
}
```

What it is doing, and why

Your chapter 14 version walked one byte at a time. This one walks eight, using a bit trick to test all eight for a zero simultaneously.

`HASZERO` is the heart of it. For a word `x`, `x - ONES` borrows out of any byte that is zero; `~x` has the top bit set in any byte whose top bit was clear; the AND with `HIGHS` keeps only the top bit of each byte. The result is non-zero exactly when some byte of `x` is zero. One subtraction, two ANDs, one NOT — for eight bytes.

The first loop exists because a word-sized load at an unaligned address is undefined or slow, exactly as chapter 37 established. It advances byte by byte only until alignment is reached — at most seven iterations.

The third loop is needed because the second stops at the *word* containing the terminator, not the byte.

What to take from it

The constants are computed, not written. `ONES` is derived from `sizeof(size_t)` and `UCHAR_MAX`, so the same source is correct on a 32-bit and a 64-bit machine. Chapter 37's portability discipline, applied to a bit trick.

The alignment loop is not optional. It is the difference between fast and undefined.

It is unreadable without the explanation — and that is acceptable here. `strlen` is called billions of times a day, its specification will never change, and it is thirteen lines. The same density in your application code would be indefensible.

Measure it against yours

```
/* bench.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

static size_t naive_strlen(const char *s)
{
    const char *p = s;
    while (*p) p++;
    return (size_t)(p - s);
}

int main(void)
{
    size_t n = 1000000;
    char *big = malloc(n + 1);
    memset(big, 'x', n);
    big[n] = '\0';

    clock_t t0 = clock();
    volatile size_t a = 0;
    for (int i = 0; i < 200; i++) a += naive_strlen(big + i % 8);
    double naive = (double)(clock() - t0) / CLOCKS_PER_SEC;

    t0 = clock();
    volatile size_t b = 0;
```

```
    for (int i = 0; i < 200; i++) b += strlen(big + i % 8);
    double library = (double)(clock() - t0) / CLOCKS_PER_SEC;

    printf("naive   %.3f s\nlibrary %.3f s\nratio   %.1fx\n",
           naive, library, naive / library);
    free(big);
    return 0;
}
```

```
gcc -O2 -o bench bench.c && ./bench
```

The library version is typically several times faster on a long string. On a glibc system the strlen you just measured is glibc's, which goes further than musl's word-at-a-time loop and uses SIMD instructions. The lesson is not to write your own — it is that the standard library is not a convenience wrapper, and the gap between "correct" and "correct and fast" is real.

Then read something with a design in it

```
git clone --depth 1 https://github.com/redis/redis
sed -n '1,120p' redis/src/sds.h        # a length-prefixed string type
sed -n '1,80p'  redis/src/dict.h       # an incrementally rehashing hash table
```

`sds` is Redis's answer to chapter 14's null-terminated strings: it stores the length in a header immediately before the characters, so `sdslen` is O(1) and the pointer can still be passed to any C function expecting `char *`. Read the header and work out how that dual nature is achieved before reading the implementation.

`dict.c` is chapter 33's hash table with one addition: rehashing happens incrementally, a few buckets per operation, so no single insertion ever pauses for a full rehash. That matters for a server with latency guarantees — and it is a design consideration your version had no reason to have.

Reconstruct the conventions

For whichever project you choose, answer section 5's six questions from the code alone, writing the answers down:

```
grep -rn "goto cleanup\|goto fail\|goto err" src/ | head
grep -rn "_create\|_free\|_destroy\|_new\|_release" src/*.h | head
grep -rn "zmalloc\|sqlite3_malloc\|xmalloc" src/ | head -5
ls include/ 2>/dev/null || grep -l "^[a-z].*(" src/*.h | head
```

Then check your conclusions against the project's contributing guide, if it has one. Where the code and the document disagree, the code is what reviewers will enforce.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Reading a large codebase linearly	Hours spent, nothing retained	Follow one execution path end to end.
Reading without building	Cannot experiment or verify	Get it compiling first.
Assuming old code is good code	Copying a 2005 workaround	Check the history; ask why.
Copying a clever idiom without the context	Unreadable application code	Density is justified in a hot library function, rarely elsewhere.
Ignoring the house conventions	Patch rejected on style	Reconstruct them before writing.
Starting with the kernel	Overwhelming; nothing learned	Start with musl.
Reading only the code	Missing the intent	`git log` and `git blame` explain the why.

9 Check yourself

Why do arrays decay to pointers when passed to a function?

Because copying an array on every call was unaffordable on a machine with 64 KB of memory — passing the address costs one word regardless of size. The consequence is everything chapter 13 covered: `sizeof` changes inside the function, the length must be passed separately, and a function can modify the caller's data despite pass-by-value.

What is C trading away by leaving signed overflow undefined?

Diagnosability, in exchange for performance and portability. Defining it would require extra instructions on machines that trap or use a different representation, and would prevent the optimizer from assuming `i + 1 > i` — which is what allows loop counters in registers and vectorization. The cost falls on the incorrect program rather than on every program.

Why start with musl rather than the Linux kernel?

Because you already know precisely what each musl function must do — you have the specification in your head from forty chapters of using them. That frees all your attention for how it is implemented. The kernel requires learning the domain and the code simultaneously, which is why people bounce off it.

What six things should you determine before changing unfamiliar C?

How errors are reported, who owns and frees memory, the naming conventions, whether cleanup is centralized with `goto`, whether allocation goes through a wrapper, and which headers constitute the public interface. C supplies none of these, so every project invents its own — consistently, and usually without documenting it.

Why is musl's dense `strlen` good code while the same density in your program would not be?

Because the trade-off differs. `strlen` is thirteen lines, its specification will never change, and it runs billions of times a day, so unreadability buys a real and permanent gain. Application code changes constantly and is read far more than executed, so the same density costs maintainability for a speedup nobody measured.

10 Where this leads

That completes the Advanced level. The Professional level starts from a different premise: not what C can express, but what working in C alongside other people actually requires. Chapter 43 makes this chapter practical — not reading a codebase to learn from it, but navigating one in order to change it, locating a regression with `git bisect` and making the smallest change that fixes the problem. Chapter 44 then puts that change through review.

PART 4

Professional

Working as a C programmer

Chapters 43–56

PROFESSIONAL · CHAPTER 43

Working in an Existing Codebase

Almost no professional work starts from an empty file. It starts from a bug report against two hundred thousand lines nobody currently understands in full. This chapter is the workflow for that — locating the defect, understanding enough to be confident, and changing as little as possible.

BY THE END OF THIS CHAPTER YOU CAN

- Orient yourself in an unfamiliar codebase within an hour.
- Find every use of a symbol, and its definition, in seconds.
- Locate the commit that introduced a regression with `git bisect`.
- Infer the invariants a function relies on before modifying it.
- Produce a change small enough that a reviewer can be certain of it.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The first hour

Resist the urge to open the file named in the bug report. Spend the first hour on orientation; it pays for itself immediately.

1. **Build it.** Before anything else. A codebase you cannot compile is one you cannot test a hypothesis against.
2. **Run the tests.** Note which already fail — otherwise you will spend an afternoon on a failure you did not cause.
3. **Read the directory structure.** Five minutes with `ls` and `cloc` reveals the architecture.
4. **Read the public headers.** They are the intended summary of what the code does.
5. **Find the entry point** and follow one path end to end.

```
cloc .                          # size, languages, where the weight is
ls -R src include | head -50
git log --oneline -20           # what is being worked on now
git shortlog -sn | head         # who to ask
wc -l src/*.c | sort -n | tail  # the biggest files are usually the core
```

2 Navigation

```
# grep, always available
grep -rn "symbol" --include='*.c' --include='*.h' .
grep -rn "\bparse_config\b" .              # word boundary: fewer false hits
grep -rln "TODO\|FIXME\|XXX" src/          # known soft spots

# ctags: jump to definitions
ctags -R .
vim -t parse_config                        # or Ctrl-] on the identifier

# a language server: exact, not textual
bear -- make                               # produce compile_commands.json
# then clangd in any editor: go to definition, find all references
```

The difference matters. `grep` finds the text `count` in comments, in other identifiers, and in strings. A language server knows which `count` you mean, follows it through macros, and finds every reference and no others. On a large codebase that is the difference between a list of four hundred hits and a list of six.

QUESTION	COMMAND
Where is this defined?	`ctags`, or clangd's go-to-definition
Who calls this?	`grep -rn`, or find-all-references
What does this header expose?	`grep '^[a-z].*(' header.h`
Who actually calls it at run time?	A breakpoint and `bt` in GDB
When did this line appear?	`git blame`
When did this symbol appear?	`git log -S symbol`

The fourth row deserves emphasis. Static analysis of C is limited by function pointers, macros, and conditional compilation. A breakpoint and a backtrace answer "who calls this" definitively, in one run.

3 Reading before writing

Before changing a function, establish four things. None is usually written down.

FIND OUT	HOW
Preconditions — what must be true on entry	Read three call sites. What do they all check first?
Postconditions — what callers rely on	What do callers do with the result without checking?
Ownership — who frees what	Follow one allocation from creation to `free`
Error convention — how failure travels	The return type, and what the callers test

Three call sites is a rule of thumb worth following literally. One tells you what is possible; three tell you what is normal.

> **Chesterton's fence.** Code that looks unnecessary usually is not. Before deleting an apparently redundant check, run `git log -S` on it — a one-line guard often has a commit message naming the bug it fixed. If the history is silent and you still cannot explain it, that is a reason to be careful, not a licence to remove it.

4 `git bisect`

"It worked in version 2.1 and fails in 2.4" — with four hundred commits in between. Bisection finds the guilty one in about nine steps by binary search over history.

```
git bisect start
git bisect bad                  # the current commit is broken
git bisect good v2.1            # this one worked
# git checks out a commit in the middle
make && ./run_the_test
git bisect good                 # or: git bisect bad
# repeat until git names the commit
git bisect reset                # return to where you started
```

Automate it and the whole search takes one command:

```
git bisect start HEAD v2.1
git bisect run ./check.sh       # exit 0 = good, non-zero = bad
```

```
#!/bin/sh
# check.sh — exit 0 if the build is good
make clean && make >/dev/null 2>&1 || exit 125   # 125 = skip, untestable
./prog test_input | grep -q "expected output"
```

Exit code 125 marks a commit as *untestable* — it does not build, so it is skipped rather than blamed. Without it, a broken intermediate commit derails the search.

Bisection is the single highest-leverage technique in this session. Nine builds against reading four hundred diffs, and the result is a specific commit with a message and an author.

5 The minimal change

Once you know the cause, the temptation is to fix everything nearby. Resist it, for reasons that are practical rather than aesthetic.

SMALL CHANGE	LARGE CHANGE
Reviewed properly	Approved without being read
Bisects cleanly later	Hides the next regression
Reverts cleanly	Reverting removes the good with the bad
Backports to a release branch	Does not

The rules that follow:

- **One change per commit.** A fix and a cleanup are two commits even if they touch one line each.
- **No drive-by reformatting.** A whitespace change across a file turns a one-line diff into an unreviewable one. If the file needs reformatting, that is a separate commit, ideally its own pull request.
- **Match the local style even where you dislike it.** Consistency within a file beats correctness of style.
- **Add a regression test.** A fix without one will be undone within a year.
- **Explain the why in the commit message.** The diff shows what changed; only you know why.

```
Fix use-after-free when config reload fails

parse_config() freed the old config before validating the new one,
so a malformed file left the global pointer dangling and the next
request dereferenced freed memory.

Validate first, then swap and free. Adds a regression test that
reloads with a syntax error and checks the old config survives.

Fixes: #1234
```

6 Worked example: a regression, found and fixed

Build a small repository with a bug buried in its history, then find it the way you would at work.

```
mkdir -p bisect-demo && cd bisect-demo && git init -q
```

Commit 1 — a working version:

```
cat > stats.c <<'EOF'
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Returns the mean of n values. Caller guarantees n > 0. */
static double mean(const int *v, size_t n)
{
    long total = 0;
    for (size_t i = 0; i < n; i++) total += v[i];
    return (double)total / (double)n;
}

static int maximum(const int *v, size_t n)
{
    int best = v[0];
    for (size_t i = 1; i < n; i++) if (v[i] > best) best = v[i];
    return best;
```

```
}

int main(int argc, char *argv[])
{
    if (argc < 2) { fprintf(stderr, "usage: %s N...\n", argv[0]); return 2; }

    size_t n = (size_t)(argc - 1);
    int *v = malloc(n * sizeof *v);
    if (v == NULL) return 1;
    for (size_t i = 0; i < n; i++) v[i] = atoi(argv[i + 1]);

    printf("count %zu mean %.2f max %d\n", n, mean(v, n), maximum(v, n));
    free(v);
    return 0;
}
EOF
cat > check.sh <<'EOF'
#!/bin/sh
gcc -std=c17 -Wall -Wextra -o stats stats.c 2>/dev/null || exit 125
./stats 1 2 3 | grep -q "count 3 mean 2.00 max 3"
EOF
chmod +x check.sh
git add . && git commit -qm "Add stats tool"
```

Now seven commits of ordinary churn, with the bug introduced in the middle:

```
for i in 1 2 3; do
  echo "/* comment $i */" >> stats.c
  git commit -qam "Document internals, part $i"
done

# the regression: a bounds change that looks like a cleanup
sed -i 's/for (size_t i = 1; i < n; i++) if (v\[i\]/for (size_t i = 1; i <= n; i++) if
(v[i]/' stats.c
git commit -qam "Simplify the maximum loop"

for i in 4 5 6; do
  echo "/* comment $i */" >> stats.c
  git commit -qam "Document internals, part $i"
done
```

Confirm it is broken, then bisect (with the sanitizer check from the next subsection; the plain check usually passes):

```
./check.sh; echo "check says $?"      # non-zero under a sanitizer build

git bisect start HEAD HEAD~7
git bisect run ./check.sh
```

```
a1b2c3d is the first bad commit
    Simplify the maximum loop

 stats.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Seven commits, three builds, and the answer is a specific commit with its diff. Reading seven diffs by hand would have worked here; reading four hundred would not, and the procedure is identical.

```
git bisect reset
```

Make the sanitizer do the detecting

The bug reads one element past the end. Without instrumentation the output is often still correct, because the `int` past the array happens to be small — the silent case from chapter 12. Strengthen the check:

```
cat > check.sh <<'EOF'
#!/bin/sh
gcc -std=c17 -Wall -Wextra -g -fsanitize=address -o stats stats.c 2>/dev/null \
    || exit 125
./stats 1 2 3 2>&1 | grep -q "count 3 mean 2.00 max 3" || exit 1
./stats 1 2 3 2>&1 | grep -q "AddressSanitizer" && exit 1
exit 0
EOF
```

Now the check fails on the memory error rather than on the visible output, so it catches the commit even when the wrong answer happens to look right. A bisect script is only as good as its test — which is the same point chapter 35 made about CI.

Investigate before fixing

```
git show a1b2c3d                      # what exactly changed
git log -S "i <= n" --oneline         # was this pattern used elsewhere?
grep -rn "maximum(" .                 # who calls it
git blame -L 12,20 stats.c            # who wrote the surrounding lines
```

Answer the four questions from section 3 before touching anything. `maximum`'s precondition is `n > 0` — visible from `v[0]` in the first line, and from the single caller checking `argc < 2`. Its postcondition is an element of the array. That is enough to know the fix is the bound and nothing else.

The minimal change

```
sed -i 's/i <= n; i++) if (v\[i\]/i < n; i++) if (v[i]/' stats.c
```

One character. Now resist the four other things you noticed: `atoi` should be `strtol` (chapter 23), `maximum`'s precondition is undocumented, the allocation failure is reported silently, and the comments are noise. Every one is a real improvement and none belongs in this commit.

```
git add stats.c
git commit -q -F - <<'EOF'
Fix off-by-one in maximum()

Commit a1b2c3d changed the loop bound from i < n to i <= n while
simplifying, so maximum() reads one element past the end of the
array. AddressSanitizer reports a heap-buffer-overflow; without it
the wrong value is usually plausible, which is why it was not
noticed in review.

Restore the half-open bound. Adds the sanitizer build to check.sh
so this class of defect fails the check rather than passing it.
EOF
```

The message says what broke, when, why it was missed, and what prevents a recurrence. A reviewer can approve it without reconstructing any of that.

Then open a separate issue for the rest

```
git checkout -b cleanup-stats
# atoi → strtol with error checking, document the preconditions
git commit -qam "Validate numeric arguments with strtol"
```

A separate branch and a separate review. The fix can be merged and backported today; the cleanup can be discussed at leisure. Coupling them delays the fix for the sake of the cleanup, which is the wrong trade every time.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Editing before building and testing	Blamed for a pre-existing failure	Establish the baseline first.
Reading four hundred diffs by hand	Hours lost	`git bisect run`.
A bisect script that is too weak	Blames the wrong commit	Make the test detect the actual defect.
No exit 125 for unbuildable commits	A broken build is reported as the cause	Skip them explicitly.
Deleting code you do not understand	Reintroduces an old bug	`git log -S` first.
Reformatting while fixing	The real change is invisible in the diff	Separate commits.
Fixing without a regression test	The bug returns	Add the test in the same commit.
A commit message that restates the diff	The reason is lost forever	Explain why, not what.

8 Check yourself

Why build and run the tests before reading any code?

To establish a baseline. Without it you cannot distinguish a failure you caused from one that was already there, and you cannot test any hypothesis about the bug. It also forces you to solve the build problems while they are the only problem you have.

What does exit code 125 mean to `git bisect run`?

That this commit cannot be tested — typically it does not build — so it should be skipped rather than judged. Without it, a commit that fails to compile is recorded as bad and the search converges on the wrong change. Any bisect script should return 125 when the build fails.

Why read three call sites rather than one before changing a function?

Because one call site shows what is possible and three show what is normal. The preconditions a function relies on are rarely documented; they are visible as the checks every caller performs first. Three samples distinguish a genuine invariant from one caller's local habit.

Give two practical reasons a small change is better than a large one.

It is actually reviewed rather than approved unread, and it bisects and reverts cleanly later. A large change that mixes a fix with a cleanup cannot be backported to a release branch, and if it turns out to be wrong, reverting it removes the good parts with the bad.

Why does a commit message explain why rather than what?

Because the diff already shows what changed, and shows it more precisely than prose can. What no tool can recover is the reasoning: which bug this fixes, what was tried first, why the obvious alternative was rejected. That is what the next person — often you — needs in two years.

9 Where this leads

Chapter 44 submits this change: branching, the review workflow, and reviewing someone else's C against a memory-safety checklist. The regression test added here is what makes the change reviewable, and the small scope is what makes the review possible at all.

PROFESSIONAL · CHAPTER 44

Collaborative Development and Code Review

Reviewing C is a different job from reviewing most languages. The compiler catches less, the runtime catches nothing, and the defects that matter — ownership, lifetime, bounds — are invisible in a diff unless you know to look. This chapter is a checklist and a workflow.

BY THE END OF THIS CHAPTER YOU CAN

- Work on a branch, keep it current, and choose between merge and rebase.
- Submit a change that a reviewer can evaluate without reconstructing your reasoning.
- Review C for memory safety, ownership, and undefined behavior systematically.
- Write commit messages and version numbers that mean something.
- File a bug report someone can act on.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Branching

```
git switch -c fix-config-leak      # branch and switch (modern spelling)
# … work, commit …
git push -u origin fix-config-leak
```

One branch per change, named after the change. Keep it short-lived: a branch open for three chapters diverges until merging it is its own project.

Merge or rebase

```
git switch fix-config-leak
git fetch origin
git merge origin/main             # preserves history; adds a merge commit
# or
git rebase origin/main            # replays your commits; linear history
```

MERGE	REBASE
True history, including the messy parts	Linear, readable history
Safe on a shared branch	**Never on a shared branch** — it rewrites commits
Conflicts resolved once	Conflicts possibly once per commit
Bisection sees the merge	Bisection is cleaner

The workable rule: **rebase your own unpushed work, merge everything else.** Rebasing commits that someone else has pulled rewrites history they already have, and the recovery is unpleasant.

Before submitting, tidy your own branch:

```
git rebase -i origin/main         # squash "fix typo" into its parent
git log --oneline origin/main..   # review what you are about to submit
```

Nobody needs to see "wip", "fix typo", and "actually fix it". They need to see one coherent commit per idea.

2 Submitting a change

A pull request has three parts a reviewer reads in order: the title, the description, and the diff. Most of the review's quality is decided before the diff.

```
Title:  Fix use-after-free when config reload fails

What
  parse_config() freed the old configuration before validating the
  new one. A malformed file left the global pointer dangling and the
  next request dereferenced freed memory.

Why this fix
  Validate into a temporary, then swap and free. The alternative —
  copying the old config first — doubles peak memory for a case that
  should be rare.

Testing
  New test reloads with a syntax error and checks the old config is
  still served. Full suite passes under ASan and UBSan.
```

```
Risk
  The swap is not atomic; a concurrent reader could observe either
  version. That was already true and is out of scope here (#1240).
```

The *Risk* section is the one people omit and reviewers value most. Stating what you did not fix, and why, prevents the review spending its attention rediscovering it.

DO	DO NOT
One logical change per request	Bundle a fix with a refactor
Under about 400 lines	Submit 2000 and expect a real review
Say how you tested it	Leave the reviewer to guess
Answer every comment	Silently push a change instead of replying
Push fixes as new commits during review	Force-push mid-review; the reviewer loses their place

3 Reviewing C

Read the diff twice. Once for what it does, once against this list — in order, because the early items cause the expensive failures.

Memory

- Every allocation checked for `NULL`?
- Every path — including every early return — frees what it allocated?
- Any pointer used after `free`? Set to `NULL` afterwards?
- `realloc` assigned to a temporary, not to the original pointer?
- Ownership stated: who frees the result, and is it documented?

Bounds

- Every array index provably within range?
- Every buffer write bounded, with the size passed alongside the pointer?
- `strcpy`, `strcat`, `sprintf`, or `gets` anywhere? Each is a finding.
- Room for the terminator — length + 1?
- Loop bounds `<` rather than `<=`?

Integers and undefined behavior

- Signed and unsigned compared or mixed in arithmetic?

- A size computed by multiplication that could overflow?
- An overflow check performed *after* the operation?
- Shift counts bounded by the type's width?
- Any variable read before it is written?

Errors and concurrency

- Every return value that can fail actually checked?
- `errno` saved before any intervening call?
- Shared state accessed without a lock? Locks taken in a consistent order?
- Any non-reentrant function — `strtok`, `localtime` — in threaded code?

> **Review the deleted lines too.** A diff draws the eye to additions, but removing a bounds check, a `free`, or a `NULL` guard is a defect that appears only in the red half. Read what left as carefully as what arrived.

4 How to write a review comment

Review the code, never the person. Two formulations of the same finding:

```
Poor:  You forgot to free buf on the error path.


Good:  Line 47: if parse() fails we return without freeing buf.
       A goto cleanup here would match the pattern used in
       load_index() and cover both exits.
```

The second names the location, states the consequence, proposes a fix, and cites a precedent in the same codebase. It is also shorter to act on.

Mark the severity, because not everything is equal:

PREFIX	MEANS
blocking	A defect. Must change before merge.
question	I do not understand; explain or clarify.
suggestion	Would improve it; your call.
nit	Style or typo; ignore if you disagree.

Without the labels, a reviewer's typo comment and their use-after-free comment look identical, and the author has to guess which blocks the merge.

Two further habits: approve when it is good enough rather than perfect — a review that demands perfection stalls everything — and say what you liked, because a review consisting only of complaints teaches nothing about what to repeat.

5 Versions and changelogs

Semantic versioning — `MAJOR.MINOR.PATCH` — states what a consumer must do:

BUMP	WHEN	CONSUMER MUST
PATCH	A bug fix, no interface change	Nothing
MINOR	New functionality, backwards compatible	Nothing
MAJOR	A breaking change	Change their code

For a C library the rule is stricter than for most languages, because compatibility has two levels. **API** compatibility means existing source still compiles. **ABI** compatibility means an already-compiled program still links and runs — and that breaks on changes invisible in the header's text, such as adding a member to a struct the caller can see, or reordering enumerators. Chapter 45 covers the mechanics; here, note only that "I just added a field" can be a major version.

A changelog is written for users, not from `git log`:

```
## 2.3.0 — 2026-09-20

### Added
- `config_reload()` for reloading without a restart

### Fixed
- Use-after-free when a config reload failed (#1234)
- Memory leak in the error path of `index_open()` (#1251)

### Changed
- `parse_line()` now rejects trailing whitespace. Previously it was
  silently ignored; scripts relying on that will need updating.
```

6 Worked example: a review, in full

Here is a patch as submitted. Review it before reading on.

```
--- a/src/config.c
+++ b/src/config.c
@@ -39,0 +40,28 @@ static Config *global_config = NULL;
+/* Load a configuration file and install it. */
+int config_reload(const char *path)
+{
+    FILE *f = fopen(path, "r");
+    if (!f) return -1;
+
+    char *buf = malloc(8192);
+    fread(buf, 1, 8192, f);
+
+    free(global_config);
+    global_config = malloc(sizeof(Config));
+
+    char *line = strtok(buf, "\n");
+    while (line) {
+        char key[32], value[64];
+        sscanf(line, "%s = %s", key, value);
+
+        if (strcmp(key, "timeout")) {
+            global_config->timeout = atoi(value);
+        } else if (strcmp(key, "name") == 0) {
+            strcpy(global_config->name, value);
+        }
+        line = strtok(NULL, "\n");
+    }
+
+    fclose(f);
+    return 0;
+}
```

The review

blocking — line 46: unchecked allocation. `malloc(8192)` is not checked, and `fread` writes to `buf` immediately. On failure this is a null dereference. Same on line 50 for the `Config`.

blocking — line 47: `buf` is not null-terminated. `fread` returns a count that is ignored, and nothing writes a `'\0'`. `strtok` then reads past the end of the allocation. Capture the return value and terminate:

```
size_t n = fread(buf, 1, 8191, f);
buf[n] = '\0';
```

blocking — line 49: the old config is freed before the new one is valid. This is the defect the change was meant to avoid. If parsing fails, `global_config` points at freed memory and the next request dereferences it. Build the new config in a local, then swap:

```
Config *fresh = calloc(1, sizeof *fresh);
/* … parse into fresh, returning on error … */
Config *old = global_config;
global_config = fresh;
free(old);
```

blocking — line 57: `strcmp` used backwards. `if (strcmp(key, "timeout"))` is true when the strings *differ*. Every non-timeout key sets the timeout. Chapter 14's trap, and the compiler cannot help.

blocking — line 55: unbounded `sscanf`. `%s` into `char key[32]` with no width is a stack buffer overflow from a config file. Use `%31s` and `%63s`, and check the return value — a malformed line currently leaves both buffers uninitialized and they are used anyway.

blocking — line 60: `strcpy` into a fixed buffer. Unbounded copy of attacker-controlled text. `snprintf(global_config->name, sizeof global_config->name, "%s", value)`.

blocking — `buf` is leaked. Allocated on line 46, never freed on any path.

blocking — `f` is leaked on the early return at line 44? No — the return precedes the `fopen` succeeding. But if the `malloc` check is added as suggested, that new path must close `f`. This is exactly why chapter 27 prefers one cleanup label over scattered returns.

question — is this called from more than one thread? `strtok` keeps hidden static state and `global_config` is swapped without synchronization. If reloads can be concurrent with requests, both are races. `strtok_r` at minimum.

suggestion — `atoi` cannot report failure. A malformed timeout becomes 0 silently. `strtol` with `endptr`, as in `parse_port()` two files over.

nit — the comment says what the function does; the interesting part is the ownership. Worth stating that the caller does not own the result and that the previous

config is freed.

The revised version

```
#include "config.h"

/* Load a configuration file and install it as the active config.
 * On failure the previous configuration remains installed and is
 * unchanged. The caller does not own the result.
 * Returns 0 on success, -1 on failure with errno set where possible. */
int config_reload(const char *path)
{
    int      status = -1;
    FILE    *f      = NULL;
    char    *buf    = NULL;
    Config  *fresh  = NULL;

    f = fopen(path, "r");
    if (f == NULL) {
        goto cleanup;
    }

    buf = malloc(CONFIG_MAX + 1);
    if (buf == NULL) {
        goto cleanup;
    }

    size_t n = fread(buf, 1, CONFIG_MAX, f);
    if (ferror(f)) {
        goto cleanup;
    }
    buf[n] = '\0';                      /* fread does not terminate */

    fresh = calloc(1, sizeof *fresh);    /* calloc: every field defined */
    if (fresh == NULL) {
        goto cleanup;
    }

    char *save = NULL;                  /* strtok_r: no hidden state */
```

```
    for (char *line = strtok_r(buf, "\n", &save);
         line != NULL;
         line = strtok_r(NULL, "\n", &save)) {

        char key[32], value[64];
        if (sscanf(line, "%31s = %63s", key, value) != 2) {
            continue;                   /* skip malformed lines */
        }

        if (strcmp(key, "timeout") == 0) {
            long v;
            if (!parse_long(value, &v) || v < 0) {
                goto cleanup;           /* refuse rather than default to 0 */
            }
            fresh->timeout = (int)v;
        } else if (strcmp(key, "name") == 0) {
            snprintf(fresh->name, sizeof fresh->name, "%s", value);
        }
    }

    /* Only now is the old configuration replaced. */
    Config *old = global_config;
    global_config = fresh;
    fresh = NULL;                       /* ownership transferred */
    free(old);
    status = 0;

cleanup:
    free(fresh);                        /* NULL unless we failed */
    free(buf);
    if (f != NULL) {
        fclose(f);
    }
    return status;
}
```

What changed, and which chapter it came from

FIX	FROM
Every allocation checked	Chapter 19
One cleanup label, no leaks on any path	Chapter 27
Validate before replacing the old config	Chapter 20 — use after free
`%31s`, `snprintf` instead of `strcpy`	Chapters 8, 14
`strcmp(...) == 0`	Chapter 14
`strtol` instead of `atoi`	Chapter 23
`strtok_r`	Chapter 41
`fread` return value used; buffer terminated	Chapter 26

Eight blocking findings in twenty-eight lines, and every one is a topic from earlier in this course. That is what makes a checklist worth having: the defects are not exotic, and a reviewer working from memory will miss two or three of them on a Friday afternoon.

Practise it

Find an open pull request in a C project on GitHub and review it against section 3's list. Write the comments out, even if you do not post them, and then read what the actual reviewers said. The gap between your list and theirs is the most direct feedback available on how you read C.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Rebasing a shared branch	Rewrites history others have	Rebase only your own unpushed work.
A 2000-line pull request	Approved without being read	Split it.
Bundling a fix with a refactor	Cannot be reviewed, reverted, or backported	Separate commits, separate requests.
Reviewing only added lines	A deleted bounds check slips through	Read the red half too.
Unlabelled review comments	Author cannot tell a typo from a defect	blocking / question / suggestion / nit.
Reviewing the person	Defensiveness; worse outcomes	Name the line and the consequence.
Force-pushing mid-review	Reviewer loses their place	Add commits; squash at the end.
Bumping MINOR for an ABI break	Installed programs crash	An ABI break is MAJOR.

8 Check yourself

When is rebasing dangerous?

On any branch someone else has pulled. Rebasing creates new commits with new identities, so collaborators' history no longer matches and their next pull produces duplicates or conflicts. Rebase your own unpushed work to tidy it; merge everything else.

Why review the deleted lines in a diff?

Because removing code is as capable of introducing a defect as adding it. A deleted bounds check, `free`, or null guard leaves no trace in the added half, and the eye is drawn to additions. Reviewing the red lines is how those are caught.

Why label review comments by severity?

Because a typo and a use-after-free look identical as comments, and the author cannot tell which blocks the merge. Labels also let a reviewer raise minor points without implying they are conditions — which makes it easier to mention them at all.

Why can a C library need a MAJOR bump for adding a struct member?

Because if the struct is visible in a public header, callers compiled against the old definition allocated the old size and indexed the old offsets. A new member changes the layout, so an already-compiled program linking against the new library reads and writes the wrong bytes. That is an ABI break even though the source still compiles.

Which review finding in the worked example would a compiler have caught?

None of the blocking ones. `-Wall` might warn about the ignored `fread` result, and `-Wformat` about nothing here — but the inverted `strcmp`, the unbounded `%s`, the premature `free`, and the leaked buffer all compile silently. That is precisely why C review needs a checklist rather than a reading.

9 Where this leads

Chapter 45 turns the module you have been reviewing into something other people can install: versioning, `soname`, symbol visibility, `pkg-config`, and the ABI rules that decide whether adding a struct member is a patch or a major release.

PROFESSIONAL · CHAPTER 45

Packaging and Distributing C Libraries

A library that only you can build is a directory of source files. Making it installable — and keeping it installable across versions without breaking programs already compiled against it — is a distinct engineering problem, and ABI compatibility is where most of the difficulty lives.

BY THE END OF THIS CHAPTER YOU CAN

- Design a public header that does not constrain your implementation.
- Distinguish API from ABI compatibility and classify a change correctly.
- Set a `soname` and control which symbols are exported.
- Install to the conventional layout and supply a `pkg-config` file.
- Deprecate and remove functionality without breaking existing users.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The public header

Everything in an installed header is a promise. The smaller the header, the more freedom you retain.

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <stddef.h>
#include <stdbool.h>

#ifdef __cplusplus
extern "C" {                          /* chapter 46: C++ callers */
#endif

#define INTARRAY_VERSION_MAJOR 2
#define INTARRAY_VERSION_MINOR 3
#define INTARRAY_VERSION_PATCH 0
```

```
typedef struct IntArray IntArray;     /* opaque: no layout promised */

IntArray *intarray_create(size_t initial_capacity);
void      intarray_destroy(IntArray *a);
bool      intarray_push(IntArray *a, int value);
size_t    intarray_count(const IntArray *a);

const char *intarray_version_string(void);

#ifdef __cplusplus
}
#endif

#endif /* INTARRAY_H */
```

RULE	WHY
Prefix every public name	C has no namespaces; `intarray_` is the substitute
Opaque types wherever possible	The layout stays yours — chapter 28
Include only what the header itself needs	Pulling in `<stdio.h>` imposes it on every user
No `static` or `inline` definitions	They become part of the ABI once a caller compiles them in
Version macros *and* a runtime accessor	The macro says what you compiled against; the function says what is loaded
`extern "C"` guard	Costs two lines; makes the library usable from C++

The version pair is worth understanding. `INTARRAY_VERSION_MINOR` is baked into the caller at compile time; `intarray_version_string()` reports the shared library actually loaded. When they disagree, you have a deployment problem — and without both you cannot detect it.

2 API versus ABI

	API	ABI
Compatible means	Existing source still compiles	Existing *binaries* still run
Broken by	Removing a function, changing a signature	Anything that changes layout, size, or calling convention
Detected	At compile time, loudly	At run time, as corruption

ABI is the harder discipline because the breakages are invisible in the header's text.

CHANGE	API	ABI
Add a new function	safe	safe
Add a member to an *opaque* struct	safe	**safe**
Add a member to a struct in the header	safe	**BREAKS** — size changes
Reorder members of a visible struct	safe	**BREAKS** — offsets change
Insert an enumerator in the middle	safe	**BREAKS** — values shift
Change a parameter from `int` to `long`	usually safe	**BREAKS**
Add a parameter	BREAKS	BREAKS
Change `int` to `size_t` in a returned struct	safe	**BREAKS**
Rename a function	BREAKS	BREAKS
Make a function `static`	BREAKS	BREAKS

Rows two and three are the whole argument for opaque types. If the caller never sees the definition, it never allocates the struct and never computes an offset — so you may add, remove, and reorder members forever.

> **An ABI break with no API break is the dangerous one.** The user's code still compiles, so nothing warns. Their existing binary loads the new library, allocates the old size, and writes past the end of every object. The symptom is memory corruption in code that was not recompiled and did not change.

3 `soname` and symbol visibility

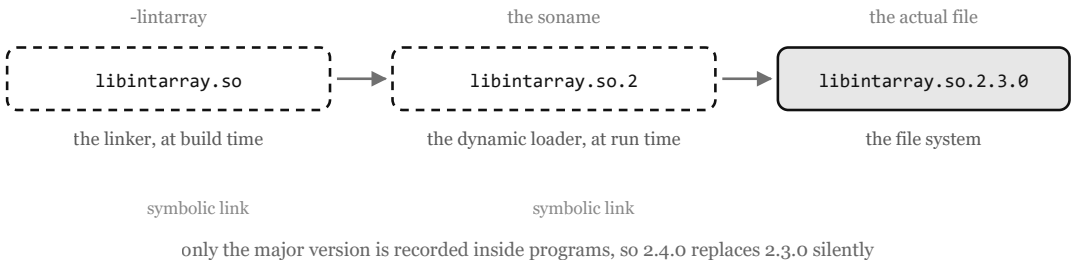
```
gcc -shared -Wl,-soname,libintarray.so.2 \
    -o libintarray.so.2.3.0 intarray.o

ln -sf libintarray.so.2.3.0 libintarray.so.2      # runtime link
ln -sf libintarray.so.2     libintarray.so        # build-time link
```

Three names, three jobs:

NAME	USED BY
`libintarray.so`	The linker at build time, via `-lintarray`
`libintarray.so.2`	The dynamic loader at run time — this is the `soname`
`libintarray.so.2.3.0`	The actual file

The `soname` is recorded inside every program linked against the library, and it contains only the major version. So installing 2.4.0 alongside 2.3.0 lets existing programs pick up the new one automatically — while 3.0.0 has `soname` `libintarray.so.3` and coexists rather than replacing it. That is how a system can hold two incompatible versions at once.



Two of the three are symbolic links. Each tool follows the chain as far as it needs and no further.

Exporting only what you mean to

By default every non-`static` symbol is exported, which makes internal helpers part of your ABI whether you intended it or not.

```
gcc -fvisibility=hidden -fPIC -c intarray.c       # hide by default
```

```
#define INTARRAY_API __attribute__((visibility("default")))

INTARRAY_API IntArray *intarray_create(size_t capacity);
```

Or list them explicitly, which works across compilers:

```
/* intarray.map */
INTARRAY_2 {
    global:
        intarray_create;
        intarray_destroy;
        intarray_push;
        intarray_count;
        intarray_version_string;
    local:
        *;
};
```

```
gcc -shared -Wl,--version-script=intarray.map -o libintarray.so.2.3.0 …
```

Verify the result rather than assuming it:

```
nm -D --defined-only libintarray.so.2.3.0 | grep ' T '
```

Anything in that list is a promise. Chapter 29's `static` keyword and this flag are the two mechanisms that keep it short.

4 Installing

GOES TO	WHAT
`$(prefix)/include/`	Public headers
`$(prefix)/lib/`	`.so`, `.a`, and the symlinks
`$(prefix)/lib/pkgconfig/`	The `.pc` file
`$(prefix)/share/doc/`	README, licence, changelog

`prefix` defaults to `/usr/local`; distributions override it to `/usr`. Two conventions that packagers require: honour `DESTDIR`, which stages the install into a temporary root, and never write outside `$(DESTDIR)$(prefix)`.

pkg-config

```
prefix=/usr/local
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: intarray
Description: A growable integer array
Version: 2.3.0
URL: https://example.org/intarray
Libs: -L${libdir} -lintarray
Cflags: -I${includedir}
```

With that file installed, a consumer never hard-codes a path:

```
gcc app.c $(pkg-config --cflags --libs intarray) -o app
pkg-config --modversion intarray
pkg-config --atleast-version=2.1 intarray && echo "new enough"
```

Two fields matter beyond the obvious. `Requires:` lists other `.pc` packages you depend on, so their flags propagate automatically. `Libs.private:` lists libraries needed only for static linking — omitting it is why some static links fail with undefined references to `-lm` or `-lpthread`.

5 Deprecating

Removing a function immediately breaks every user. The sequence that does not:

1. **Add the replacement.** Both work; nothing breaks.
2. **Mark the old one deprecated.** Users get a compiler warning naming the replacement.
3. **Document it in the changelog** with the version in which it will be removed.
4. **Wait at least one major cycle.**
5. **Remove it in a major release**, and say so prominently.

```
#if defined(__GNUC__) || defined(__clang__)
#  define INTARRAY_DEPRECATED(msg) __attribute__((deprecated(msg)))
#elif defined(_MSC_VER)
#  define INTARRAY_DEPRECATED(msg) __declspec(deprecated(msg))
#else
#  define INTARRAY_DEPRECATED(msg)
#endif

INTARRAY_DEPRECATED("use intarray_create_ex() instead")
IntArray *intarray_create(size_t capacity);
```

C23 standardizes this as `[[deprecated("message")]]`.

When you must change behavior rather than a name, add a new entry point instead of altering the old one — `intarray_create_ex` alongside `intarray_create`. It is inelegant, and it is why `*_ex` and `*2` suffixes are so common in long-lived C libraries. The alternative is breaking programs you cannot recompile.

6 Worked example: shipping the chapter 29 library

The Makefile

```
NAME     := intarray
MAJOR    := 2
MINOR    := 3
PATCH    := 0
VERSION  := $(MAJOR).$(MINOR).$(PATCH)

prefix   ?= /usr/local
libdir   ?= $(prefix)/lib
includedir ?= $(prefix)/include
pcdir    ?= $(libdir)/pkgconfig

CC       := gcc
CFLAGS   := -std=c17 -Wall -Wextra -O2 -fPIC -fvisibility=hidden \
            -DINTARRAY_BUILD
LDFLAGS  :=

SONAME   := lib$(NAME).so.$(MAJOR)
```

```
SOFILE   := lib$(NAME).so.$(VERSION)
ARFILE   := lib$(NAME).a

.PHONY: all clean install uninstall check abi-check

all: $(SOFILE) $(ARFILE) $(NAME).pc

$(NAME).o: $(NAME).c $(NAME).h
          $(CC) $(CFLAGS) -c $< -o $@

$(SOFILE): $(NAME).o $(NAME).map
          $(CC) -shared -Wl,-soname,$(SONAME) \
                -Wl,--version-script=$(NAME).map \
                -o $@ $< $(LDFLAGS)
          ln -sf $(SOFILE) $(SONAME)
          ln -sf $(SONAME) lib$(NAME).so

$(ARFILE): $(NAME).o
          ar rcs $@ $<

$(NAME).pc: $(NAME).pc.in
          sed -e 's|@prefix@|$(prefix)|g' \
              -e 's|@libdir@|$(libdir)|g' \
              -e 's|@includedir@|$(includedir)|g' \
              -e 's|@VERSION@|$(VERSION)|g' $< > $@

install: all
          install -d $(DESTDIR)$(includedir) $(DESTDIR)$(libdir) $(DESTDIR)$(pcdir)
          install -m 644 $(NAME).h  $(DESTDIR)$(includedir)/
          install -m 755 $(SOFILE)  $(DESTDIR)$(libdir)/
          install -m 644 $(ARFILE)  $(DESTDIR)$(libdir)/
          ln -sf $(SOFILE) $(DESTDIR)$(libdir)/$(SONAME)
          ln -sf $(SONAME) $(DESTDIR)$(libdir)/lib$(NAME).so
          install -m 644 $(NAME).pc $(DESTDIR)$(pcdir)/

uninstall:
          rm -f $(DESTDIR)$(includedir)/$(NAME).h
          rm -f $(DESTDIR)$(libdir)/lib$(NAME).*
          rm -f $(DESTDIR)$(pcdir)/$(NAME).pc
```

```
check: $(SOFILE)
          @echo "-- exported symbols --"
          @nm -D --defined-only $(SOFILE) | grep ' T ' || true
          @echo "-- recorded soname --"
          @objdump -p $(SOFILE) | grep SONAME

clean:
          rm -f *.o *.a *.so *.so.* $(NAME).pc
```

Build and inspect

```
make
make check
```

```
-- exported symbols --
0000000000001180 T intarray_count
0000000000001120 T intarray_create
0000000000001160 T intarray_destroy
0000000000001140 T intarray_push
00000000000011a0 T intarray_version_string
-- recorded soname --
  SONAME               libintarray.so.2
```

Exactly five symbols. Before adding `-fvisibility=hidden` and the version script, run the same check: every internal helper appears, and each one you did not intend is a symbol someone can come to depend on.

Install into a staging root

```
make install DESTDIR=/tmp/stage prefix=/usr
find /tmp/stage -type f -o -type l | sort
```

```
/tmp/stage/usr/include/intarray.h
/tmp/stage/usr/lib/libintarray.a
/tmp/stage/usr/lib/libintarray.so
/tmp/stage/usr/lib/libintarray.so.2
/tmp/stage/usr/lib/libintarray.so.2.3.0
/tmp/stage/usr/lib/pkgconfig/intarray.pc
```

Nothing outside the staging root, and both symlinks present. That is what a distribution packager needs; getting it wrong is the most common reason a library is hard to package.

Consume it from a separate project

```
mkdir -p /tmp/consumer && cd /tmp/consumer
cat > app.c <<'EOF'
#include <intarray.h>
#include <stdio.h>

int main(void)
{
    printf("compiled against %d.%d.%d, running %s\n",
           INTARRAY_VERSION_MAJOR, INTARRAY_VERSION_MINOR,
           INTARRAY_VERSION_PATCH, intarray_version_string());

    IntArray *a = intarray_create(4);
    if (a == NULL) return 1;
    for (int i = 0; i < 10; i++) intarray_push(a, i * i);
    printf("count %zu\n", intarray_count(a));
    intarray_destroy(a);
    return 0;
}
EOF

export PKG_CONFIG_PATH=/tmp/stage/usr/lib/pkgconfig
pkg-config --cflags --libs intarray
gcc app.c $(pkg-config --cflags --libs intarray) -o app
LD_LIBRARY_PATH=/tmp/stage/usr/lib ./app
ldd app | grep intarray
```

The consumer's compile line names no paths. It sees only `intarray.h` and the five functions; the structure definition is not available to it, which is exactly the freedom the opaque type bought.

Break the ABI on purpose

This is the experiment worth doing, because the failure mode is otherwise hard to believe.

Temporarily expose the struct in the header:

```
/* intarray.h — do NOT do this in a real library */
typedef struct IntArray {
    int    *data;
    size_t  count;
    size_t  capacity;
} IntArray;
```

Rebuild, reinstall, change the consumer to read `a->count` directly instead of calling `intarray_count`, and rebuild it so it compiles against that layout. Now add a field in the middle, bump only PATCH, rebuild the *library only*, and run the old consumer binary:

```
typedef struct IntArray {
    int    *data;
    size_t  generation;        /* new member, inserted */
    size_t  count;
    size_t  capacity;
} IntArray;
```

```
make && make install DESTDIR=/tmp/stage prefix=/usr
cd /tmp/consumer && LD_LIBRARY_PATH=/tmp/stage/usr/lib ./app
```

The consumer reads `count` from the offset where `generation` now lives, and prints nonsense — or crashes. Nothing was recompiled, no warning appeared, and the version number claimed a patch release. Restore the opaque form and the same experiment changes nothing at all, because the consumer never knew the layout.

Check for breaks automatically

```
abidiff old/libintarray.so.2.3.0 new/libintarray.so.2.4.0
```

`abidiff`, from libabigail, compares the debug information of two builds and reports every ABI-relevant difference. Running it in CI against the previous release is how projects with real compatibility promises keep them — it turns "we think this is safe" into a check.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Exposing a struct in a public header	Layout frozen; every field change is a major version	Opaque handle plus accessors.
No `soname`	Incompatible versions cannot coexist	`-Wl,-soname,lib…so.MAJOR`.
Exporting every non-`static` symbol	Internals become part of the ABI	`-fvisibility=hidden` plus a version script.
Ignoring `DESTDIR`	Cannot be packaged	Prefix every install path with it.
Hard-coding `/usr/local`	Distributions cannot relocate it	Honour `prefix`.
No `pkg-config` file	Users hard-code paths	Ship a `.pc`.
Omitting `Libs.private`	Static linking fails with undefined references	List the private dependencies.
Removing a function without deprecating	Every user breaks at once	Deprecate, wait a cycle, remove in a major release.
Bumping PATCH for an ABI change	Installed programs corrupt memory	An ABI break is a MAJOR bump.

8 Check yourself

What is the difference between API and ABI compatibility?

API compatibility means existing source still compiles against the new version. ABI compatibility means an already-compiled binary still links and runs correctly against it. A change can preserve the first and break the second — adding a struct member visible in the header, for instance — and that combination is the dangerous one, because nothing warns.

Why does an opaque type let you add struct members freely?

Because the caller never sees the definition, so it cannot allocate the struct, cannot compute a member offset, and cannot embed the size in its compiled code. Everything goes through your functions, which are recompiled with the new layout. Expose the definition and all three of those become promises.

Why does the `soname` contain only the major version?

So that compatible updates are picked up automatically and incompatible ones are not. Every program linked against the library records the `soname`, so installing 2.4.0 over 2.3.0 serves existing binaries, while 3.0.0 has a different `soname` and installs alongside rather than replacing. That is how two incompatible versions coexist on one system.

Why hide symbols by default?

Because every exported symbol is something a user can come to depend on, and removing it later is an ABI break. With default visibility, internal helpers are exported whether you meant them to be or not. Hiding by default and marking the public interface explicitly keeps the promise set equal to the documented one.

Why must an install target honour `DESTDIR`?

Because packagers build into a temporary staging root and then archive it, rather than installing onto the build machine. An install that writes to the real `prefix` regardless cannot be packaged, and may overwrite files on a system it was never meant to touch.

9 Where this leads

Chapter 46 calls this library from other languages. The `extern "C"` guard in the header is the first step; the rest is the platform ABI itself — calling conventions, how structures are passed, and who owns memory that crosses a language boundary.

PROFESSIONAL · CHAPTER 46

ABI and Interfacing with Other Languages

C's most common role today is not as an application language but as the layer everything else calls. Python, Rust, Java, Go, and Ruby all speak C's calling convention because it is the one the operating system speaks — which makes designing a C API for foreign callers a distinct skill.

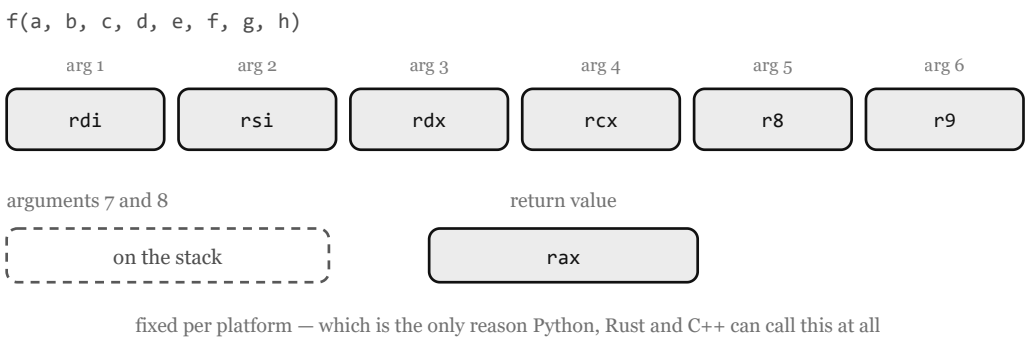
BY THE END OF THIS CHAPTER YOU CAN

- Describe how arguments and return values are actually passed.
- Explain name mangling and what `extern "C"` does.
- Call a C library from Python, Rust, and C++.
- Design an interface that survives a foreign caller's memory model.
- Read inline assembly and know when an intrinsic is the better tool.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The calling convention

An ABI specifies what the source language does not: which registers carry arguments, who saves what, how the stack is aligned, how a returned structure is delivered. It is fixed per platform, which is what lets code from different compilers and different languages link together.



Nothing in the C source says any of this. The ABI does, and every language that calls C obeys it.

On x86-64 System V — Linux, macOS, the BSDs:

ITEM	WHERE
Integer and pointer arguments 1–6	`rdi rsi rdx rcx r8 r9`
Floating-point arguments 1–8	`xmm0` – `xmm7`
Further arguments	On the stack, right to left
Integer return	`rax` (and `rdx` for 128 bits)
Floating-point return	`xmm0`
A large returned struct	The caller passes a hidden pointer in `rdi`
Callee-saved	`rbx rbp r12 r13 r14 r15`

Windows x64 assigns arguments to different registers — `rcx rdx r8 r9` — which is why a library compiled for one does not link against the other. You rarely need the details, but two facts matter in practice: **a small struct may be passed in registers while a large one is passed by hidden pointer**, and that threshold is part of the ABI. Changing a struct from 16 to 24 bytes can change how every function taking it is called, which is chapter 45's ABI break in its least obvious form.

Read it yourself with chapter 34's technique:

```
gcc -O2 -S -masm=intel mathlib.c -o mathlib.s
```

2 Name mangling and `extern "C"`

C exports a function under its own name. C++ cannot, because it permits overloading, so it encodes the parameter types into the symbol:

```
/* C */    int add(int, int);   →  symbol: add
/* C++ */  int add(int, int);   →  symbol: _Z3addii
```

```
nm -D libfoo.so | c++filt        # decode mangled names
```

`extern "C"` tells a C++ compiler to use C linkage — no mangling, C calling convention — which is why the guard belongs in every public C header:

```
#ifdef __cplusplus
extern "C" {
#endif
    /* declarations */
#ifdef __cplusplus
}
#endif
```

The `__cplusplus` test is required because `extern "C"` is not valid C. With the guard, one header serves both languages; without it, a C++ program linking your library gets undefined references to mangled names that do not exist.

3 Designing for foreign callers

A C API that is pleasant from C can be unusable from Python. The constraints a foreign language imposes:

AVOID	BECAUSE	INSTEAD
Structs passed by value	Layout and padding must be replicated exactly	Opaque pointers
Function-like macros	Invisible outside C	Real functions
Variadic functions	Most FFIs cannot call them portably	An array plus a count
Returning a pointer to static data	Unclear lifetime; not thread-safe	Caller-supplied buffer
`errno` for errors	Thread-local and awkward to read across an FFI	An explicit return code
Callbacks with unclear lifetime	The garbage collector may move or free the target	Register and unregister explicitly

The rule that follows: **a good FFI surface is narrow, opaque, and explicit.** Handles rather than structs, return codes rather than `errno`, and one `free` function per allocating function.

Ownership across the boundary

This is where real programs break. Python's garbage collector may free a buffer while your C code still holds the pointer; Rust's borrow checker cannot see into C at all; a Java array may be *moved* by the collector during a call.

- **C allocates, C frees.** Always export a matching `free` — `intarray_destroy`, not the caller's `free()`. The caller's allocator is not necessarily yours.
- **Caller allocates, caller frees.** Take a buffer and a size. Simplest and safest.
- **Never** return a pointer into memory the other language owns and expect it to stay valid.

> **Never `free()` a pointer that came from another language's allocator, or let it `free()` one of yours.** On Windows a library and its caller can be linked against different C runtimes with separate heaps, and crossing them corrupts both. This is why well-designed C libraries always export their own deallocation function.

4 Worked example: one library, three callers

The C side

```
/* mathlib.h */
#ifndef MATHLIB_H
#define MATHLIB_H

#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef enum {
    ML_OK = 0,
    ML_ERR_NULL     = 1,
    ML_ERR_EMPTY    = 2,
    ML_ERR_NOMEM    = 3
} MlStatus;

/* Scalars: the easiest thing to call from anywhere. */
int    ml_add(int a, int b);
double ml_mean(const double *values, size_t count, MlStatus *status);

/* A caller-supplied buffer: no ownership question at all. */
MlStatus ml_scale(const double *in, double *out, size_t count, double factor);
```

```
/* C allocates, C frees — the pair must be used together. */
double  *ml_make_range(double start, double step, size_t count);
void     ml_free(double *p);

/* An opaque handle: a foreign caller never needs the layout. */
typedef struct MlAccumulator MlAccumulator;

MlAccumulator *ml_acc_create(void);
void           ml_acc_destroy(MlAccumulator *acc);
MlStatus       ml_acc_add(MlAccumulator *acc, double value);
double         ml_acc_mean(const MlAccumulator *acc);
size_t         ml_acc_count(const MlAccumulator *acc);

/* A callback: explicit context, no hidden state. */
typedef int (*MlVisitor)(double value, void *context);
MlStatus ml_each(const double *values, size_t count,
                 MlVisitor visit, void *context);

const char *ml_status_string(MlStatus s);

#ifdef __cplusplus
}
#endif
#endif /* MATHLIB_H */
```

```
/* mathlib.c */
#include "mathlib.h"
#include <stdlib.h>

struct MlAccumulator {          /* definition stays here */
    double total;
    size_t count;
};

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

double ml_mean(const double *values, size_t count, MlStatus *status)
```

```
{
    if (values == NULL) { if (status) *status = ML_ERR_NULL;  return 0.0; }
    if (count == 0)     { if (status) *status = ML_ERR_EMPTY; return 0.0; }

    double total = 0.0;
    for (size_t i = 0; i < count; i++) total += values[i];
    if (status) *status = ML_OK;
    return total / (double)count;
}

MlStatus ml_scale(const double *in, double *out, size_t count, double factor)
{
    if (in == NULL || out == NULL) return ML_ERR_NULL;
    for (size_t i = 0; i < count; i++) out[i] = in[i] * factor;
    return ML_OK;
}

double *ml_make_range(double start, double step, size_t count)
{
    if (count == 0) return NULL;
    double *v = malloc(count * sizeof *v);
    if (v == NULL) return NULL;
    for (size_t i = 0; i < count; i++) v[i] = start + (double)i * step;
    return v;                       /* caller must call ml_free */
}

void ml_free(double *p) { free(p); }

MlAccumulator *ml_acc_create(void)
{
    MlAccumulator *a = calloc(1, sizeof *a);
    return a;
}

void ml_acc_destroy(MlAccumulator *a) { free(a); }

MlStatus ml_acc_add(MlAccumulator *a, double v)
{
    if (a == NULL) return ML_ERR_NULL;
```

```
    a->total += v;
    a->count++;
    return ML_OK;
}

double ml_acc_mean(const MlAccumulator *a)
{
    return (a == NULL || a->count == 0) ? 0.0 : a->total / (double)a->count;
}

size_t ml_acc_count(const MlAccumulator *a)
{
    return (a == NULL) ? 0 : a->count;
}

MlStatus ml_each(const double *values, size_t count,
                 MlVisitor visit, void *context)
{
    if (values == NULL || visit == NULL) return ML_ERR_NULL;
    for (size_t i = 0; i < count; i++) {
        if (visit(values[i], context) != 0) break;   /* non-zero stops */
    }
    return ML_OK;
}

const char *ml_status_string(MlStatus s)
{
    switch (s) {
    case ML_OK:        return "ok";
    case ML_ERR_NULL:  return "null pointer";
    case ML_ERR_EMPTY: return "empty input";
    case ML_ERR_NOMEM: return "out of memory";
    }
    return "unknown";
}
```

```
gcc -std=c17 -Wall -Wextra -O2 -fPIC -shared -o libmathlib.so mathlib.c
```

Caller 1 — Python with `ctypes`

```
# app.py
import ctypes, os

lib = ctypes.CDLL(os.path.abspath("libmathlib.so"))

# Declare every signature. ctypes assumes int otherwise, and a wrong
# assumption about a double is silent corruption, not an error.
lib.ml_add.argtypes = [ctypes.c_int, ctypes.c_int]
lib.ml_add.restype  = ctypes.c_int

lib.ml_mean.argtypes = [ctypes.POINTER(ctypes.c_double), ctypes.c_size_t,
                        ctypes.POINTER(ctypes.c_int)]
lib.ml_mean.restype  = ctypes.c_double

lib.ml_make_range.argtypes = [ctypes.c_double, ctypes.c_double, ctypes.c_size_t]
lib.ml_make_range.restype  = ctypes.POINTER(ctypes.c_double)
lib.ml_free.argtypes = [ctypes.POINTER(ctypes.c_double)]

lib.ml_acc_create.restype   = ctypes.c_void_p        # opaque handle
lib.ml_acc_destroy.argtypes = [ctypes.c_void_p]
lib.ml_acc_add.argtypes     = [ctypes.c_void_p, ctypes.c_double]
lib.ml_acc_mean.argtypes    = [ctypes.c_void_p]
lib.ml_acc_mean.restype     = ctypes.c_double

VISITOR = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_double, ctypes.c_void_p)
lib.ml_each.argtypes = [ctypes.POINTER(ctypes.c_double), ctypes.c_size_t,
                        VISITOR, ctypes.c_void_p]

print("ml_add(3, 4) =", lib.ml_add(3, 4))

data = (ctypes.c_double * 5)(1.0, 2.0, 3.0, 4.0, 5.0)
status = ctypes.c_int()
print("ml_mean =", lib.ml_mean(data, 5, ctypes.byref(status)),
      "status", status.value)

# C allocated it, so C must free it.
p = lib.ml_make_range(0.0, 0.5, 6)
```

```
print("range =", [p[i] for i in range(6)])
lib.ml_free(p)

acc = lib.ml_acc_create()
for v in (10.0, 20.0, 30.0):
    lib.ml_acc_add(acc, v)
print("accumulator mean =", lib.ml_acc_mean(acc))
lib.ml_acc_destroy(acc)

# A Python function called back from C. The reference must be kept
# alive in a variable — if it is garbage collected mid-call, C jumps
# into freed memory.
total = [0.0]
def on_value(value, _ctx):
    total[0] += value
    return 0
callback = VISITOR(on_value)          # keep this name bound
lib.ml_each(data, 5, callback, None)
print("visited total =", total[0])
```

```
python3 app.py
```

Caller 2 — Rust

```
// main.rs
use std::os::raw::{c_double, c_int, c_void};

#[link(name = "mathlib")]
extern "C" {
    fn ml_add(a: c_int, b: c_int) -> c_int;
    fn ml_mean(values: *const c_double, count: usize,
               status: *mut c_int) -> c_double;
    fn ml_make_range(start: c_double, step: c_double,
                     count: usize) -> *mut c_double;
    fn ml_free(p: *mut c_double);
    fn ml_acc_create() -> *mut c_void;
    fn ml_acc_destroy(acc: *mut c_void);
    fn ml_acc_add(acc: *mut c_void, value: c_double) -> c_int;
```

```
    fn ml_acc_mean(acc: *const c_void) -> c_double;
}

fn main() {
    // Every call is unsafe: Rust cannot verify C's contracts.
    unsafe {
        println!("ml_add(3, 4) = {}", ml_add(3, 4));

        let data = [1.0_f64, 2.0, 3.0, 4.0, 5.0];
        let mut status: c_int = 0;
        let mean = ml_mean(data.as_ptr(), data.len(), &mut status);
        println!("ml_mean = {mean} status {status}");

        let p = ml_make_range(0.0, 0.5, 6);
        let slice = std::slice::from_raw_parts(p, 6);
        println!("range = {slice:?}");
        ml_free(p);                      // C's allocator, not Rust's

        let acc = ml_acc_create();
        for v in [10.0, 20.0, 30.0] {
            ml_acc_add(acc, v);
        }
        println!("accumulator mean = {}", ml_acc_mean(acc));
        ml_acc_destroy(acc);
    }
}
```

```
rustc -L . main.rs -o rustapp
LD_LIBRARY_PATH=. ./rustapp
```

Caller 3 — C++

```
// app.cpp
#include "mathlib.h"        // the extern "C" guard does the work
#include <iostream>
#include <memory>
#include <vector>
```

```
int main()
{
    std::cout << "ml_add(3, 4) = " << ml_add(3, 4) << '\n';

    std::vector<double> data{ 1.0, 2.0, 3.0, 4.0, 5.0 };
    MlStatus status;
    double mean = ml_mean(data.data(), data.size(), &status);
    std::cout << "ml_mean = " << mean
              << " (" << ml_status_string(status) << ")\n";

    // Wrap the C handle so the destructor cannot be forgotten.
    auto acc = std::unique_ptr<MlAccumulator, decltype(&ml_acc_destroy)>(
                   ml_acc_create(), &ml_acc_destroy);
    for (double v : { 10.0, 20.0, 30.0 }) {
        ml_acc_add(acc.get(), v);
    }
    std::cout << "accumulator mean = " << ml_acc_mean(acc.get()) << '\n';

    return 0;
}
```

```
g++ -std=c++17 app.cpp -L. -lmathlib -o cppapp
LD_LIBRARY_PATH=. ./cppapp
```

Prove the mangling

```
nm -D libmathlib.so | grep ' T ' | head
```

```
0000000000001130 T ml_acc_add
0000000000001100 T ml_acc_create
…
0000000000001110 T ml_add
```

Plain names. Now compile the same source as C++ without the guard and look again:

```
g++ -fpermissive -fPIC -shared -x c++ mathlib.c -o libmangled.so 2>/dev/null
nm -D libmangled.so | grep ' T ' | head -3
nm -D libmangled.so | grep ' T ' | head -3 | c++filt
```

`_Z6ml_addii` rather than `ml_add`. No FFI in any language would find it, and that single difference is what the six lines of `extern "C"` prevent.

Break the ownership rule on purpose

In the Python script, replace `lib.ml_free(p)` with nothing and run under a leak checker; then try freeing C's pointer from Python's side. On Linux with one shared libc it may appear to work; on Windows, where the library and the interpreter can use different runtimes, it corrupts the heap. The exported `ml_free` exists precisely so the caller never has to know which allocator produced the block.

Then delete the `callback = VISITOR(on_value)` binding and pass `VISITOR(on_value)` inline. For this call it still works, because ctypes keeps the argument alive until the call returns; the temporary is freed only afterwards, so a C library that stored the pointer and called it later would jump into freed memory. Keeping the reference alive for the duration of the call is the FFI equivalent of chapter 13's dangling pointer.

5 Inline assembly and intrinsics

Occasionally there is no C expression for what you need — a specific instruction, a control register, an atomic primitive the compiler does not expose.

```
/* GCC extended asm */
static inline uint64_t read_timestamp(void)
{
    uint32_t low, high;
    __asm__ volatile ("rdtsc" : "=a"(low), "=d"(high));
    return ((uint64_t)high << 32) | low;
}
```

The syntax is: instructions, then outputs, then inputs, then clobbers. `volatile` prevents the compiler moving or eliding it.

Prefer an intrinsic. Compiler intrinsics expose the same instructions as ordinary functions, so the optimizer can still schedule and inline around them:

```
#include <immintrin.h>
uint64_t t = __rdtsc();                     /* same instruction */

int bits = __builtin_popcount(x);           /* chapter 25, in one instruction */
if (__builtin_add_overflow(a, b, &sum)) { }  /* chapter 36's check, correctly */
```

That last one is worth noting: `__builtin_add_overflow` performs the addition and reports overflow without ever invoking undefined behavior — the problem chapter 36 spent a section on. C23 standardizes equivalents in `<stdckdint.h>`.

Inline assembly is not portable across architectures or even compilers, it blocks optimization, and it is easy to get the constraints wrong. Reach for it only when no intrinsic exists.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
No `extern "C"` in a public header	C++ callers get undefined references	Add the guard.
Freeing a foreign allocator's pointer	Heap corruption, especially on Windows	Export your own `free`.
Not declaring `argtypes` in ctypes	Doubles passed as ints, silently	Declare every signature.
Letting a callback object be collected	C calls into freed memory	Keep a reference for the call's duration.
Structs by value across an FFI	Padding must be replicated exactly	Opaque handles.
Variadic functions in an FFI surface	Most bindings cannot call them	Array plus count.
`errno` as the error channel	Hard to read across a boundary	An explicit status return.
Inline assembly where an intrinsic exists	Blocks optimization; not portable	Use the intrinsic.

7 Check yourself

What does `extern "C"` actually change?

It tells a C++ compiler to give the declared functions C linkage: no name mangling and the C calling convention. Without it, C++ encodes the parameter types into the symbol name, so the linker looks for `_Z3addii` while the C library exports `add`. The `__cplusplus` guard is required because the syntax is not valid C.

Why must a library export its own deallocation function?

Because the caller's allocator may not be yours. On Windows a library and its host can link against different C runtimes with separate heaps, so passing one's pointer to the other's `free` corrupts both. Exporting `ml_free` keeps allocation and deallocation inside the same runtime and removes the question entirely.

Why prefer opaque handles over structs in an FFI surface?

Because a struct passed by value forces every binding to replicate the layout, padding, and alignment exactly, and any change to it silently breaks every caller. A handle is a pointer — every language can hold one — and the layout stays private, so the implementation remains free to change.

What goes wrong if a Python callback object is garbage collected during a C call?

C holds a raw function pointer into memory Python has freed, so the next invocation jumps into reclaimed memory. It is chapter 13's dangling pointer across a language boundary. The binding must keep a live reference for as long as C may call back — which means assigning it to a variable that outlives the call.

Why is `__builtin_add_overflow` better than checking the sum afterwards?

Because it performs the addition and reports overflow without ever executing the undefined operation, so the optimizer has nothing to assume away. Chapter 36's after-the-fact check can be deleted precisely because signed overflow is undefined; the builtin sidesteps that entirely, and C23 standardizes the idea in `<stdckdint.h>`.

8 Where this leads

Chapter 47 returns to the socket server of chapter 40 and rewrites it as a single-threaded event loop with `epoll`, then measures both under load. That is the last piece of the systems-programming material before the embedded block begins.

PROFESSIONAL · CHAPTER 47

Event-Driven and Scalable I/O

Chapter 40's server forks a process per connection — simple, and hopeless past a few thousand clients. The alternative inverts the control flow: one thread, never blocking, asking the kernel which descriptors are ready. Every high-performance server in production is built this way.

BY THE END OF THIS CHAPTER YOU CAN

- Make a descriptor non-blocking and handle `EAGAIN` correctly.
- Choose between `select`, `poll`, and `epoll`, and say why the complexity differs.
- Structure an event loop with per-connection state.
- Handle timers and signals inside the loop rather than outside it.
- Say when a thread pool is the better answer.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Why blocking does not scale

A blocking `read` suspends the calling thread until data arrives. With one thread per connection that is fine — the thread has nothing else to do. The costs appear at scale:

MODEL	PER CONNECTION	PRACTICAL CEILING
Process per connection	~1 MB plus a `fork`	a few thousand
Thread per connection	~8 MB of stack reserved, plus scheduling	tens of thousands
Event loop	a few hundred bytes of state	hundreds of thousands

The memory is only half of it. Ten thousand runnable threads make the scheduler's work proportional to the connection count, and every context switch costs cache locality — chapter 34's subject, at the operating system's scale.

An event loop turns the problem inside out. Instead of *waiting* for one descriptor, ask the kernel *which of these are ready*, then service exactly those, then ask again.

2 Non-blocking descriptors

```
#include <fcntl.h>


static bool set_nonblocking(int fd)
{
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags < 0) return false;
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK) == 0;
}
```

On a non-blocking descriptor, an operation that would have waited returns −1 with `errno` set to `EAGAIN` (or the identical `EWOULDBLOCK`). That is not an error — it means "nothing right now, ask again later".

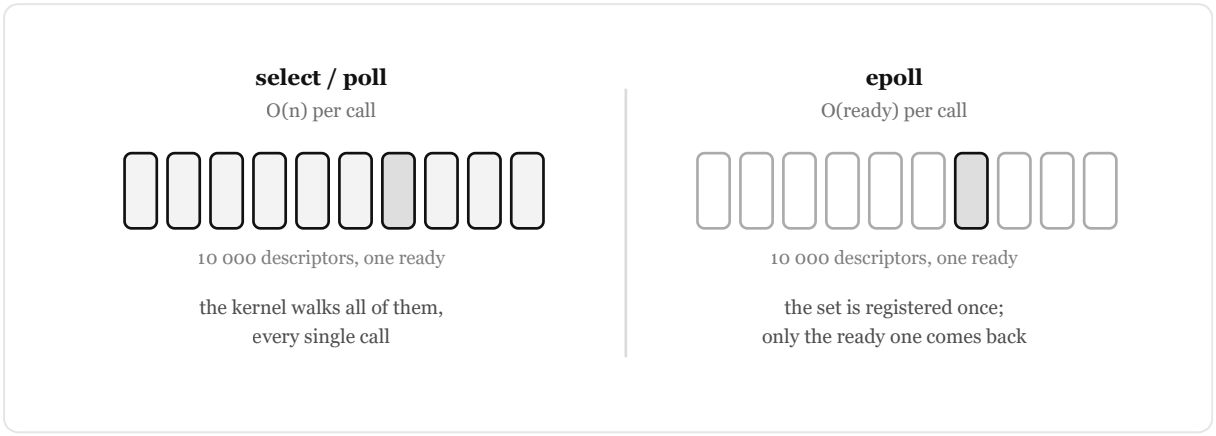
```
ssize_t n = recv(fd, buf, sizeof buf, 0);
if (n > 0)                                  { /* got data */ }
else if (n == 0)                            { /* peer closed */ }
else if (errno == EAGAIN || errno == EWOULDBLOCK) { /* not ready */ }
else if (errno == EINTR)                    { /* retry */ }
else                                        { /* real error */ }
```

Those five cases are the whole of non-blocking I/O, and treating `EAGAIN` as a failure is the most common beginner error.

3 The three multiplexers

	SELECT	POLL	EPOLL
Standard	POSIX, everywhere	POSIX, everywhere	Linux only
Descriptor limit	`FD_SETSIZE`, usually 1024	none	none
Cost per call	O(n)	O(n)	**O(ready)**
Set rebuilt each call	yes	yes	no — kept in the kernel

The complexity row is the reason `epoll` exists. With `select` or `poll`, every call passes the entire descriptor list to the kernel, which examines all of it — so a server with 10 000 idle connections and one active does 10 000 units of work per event. `epoll` registers the set once and returns only what is ready.



The same ten thousand connections, the same single event. The difference is how much work the kernel does to tell you about it.

The equivalents elsewhere are `kqueue` on the BSDs and macOS, and IOCP on Windows. Portable code uses libevent, libuv, or libev, which wrap all of them behind one interface. Writing directly to `epoll` once is still worth doing, because it makes the libraries comprehensible.

```
#include <sys/epoll.h>

int ep = epoll_create1(0);

struct epoll_event ev = { .events = EPOLLIN, .data.fd = listener };
epoll_ctl(ep, EPOLL_CTL_ADD, listener, &ev);

struct epoll_event ready[64];
int n = epoll_wait(ep, ready, 64, timeout_ms);
for (int i = 0; i < n; i++) {
    handle(ready[i].data.fd, ready[i].events);
}
```

Level-triggered and edge-triggered

MODE	REPORTS	REQUIRES
Level-triggered (default)	Ready *while* data remains	Nothing special; safe
Edge-triggered (`EPOLLET`)	Ready once, when the state *changes*	Drain until `EAGAIN`, or the rest is lost

Edge-triggered mode is faster because it reports each event once, and it is unforgiving: read half the available data and you will not be told again. Start level-triggered.

4 Structuring the loop

Blocking code keeps its state on the stack — the local variables of the function handling the connection. An event loop cannot: it returns to the top after each event, so **every connection needs an explicit state object**.

```
typedef enum { ST_READING_HEADER, ST_READING_BODY, ST_WRITING } State;


typedef struct {
    int     fd;
    State   state;
    char    in[4096];
    size_t  in_used;
    char    out[4096];
    size_t  out_used, out_sent;
    time_t  last_active;
} Connection;
```

This is the real cost of the model: what was a local variable becomes a field, and what was a sequence of statements becomes a state machine. In exchange you get a thousand connections in one thread with no locking anywhere.

Two rules that are easy to get wrong:

Never block inside the loop. One blocking call — a synchronous DNS lookup, a disk read, a slow `malloc` — stalls every connection, not one.

Only ask for writability when you have something to write. A descriptor is almost always writable, so leaving `EPOLLOUT` registered makes `epoll_wait` return immediately, forever. Add it when your output buffer is non-empty and remove it when the buffer drains.

5 Timers and signals

Both need to be part of the loop rather than outside it.

The simplest timer is the `epoll_wait` timeout: compute the time until the nearest deadline, pass it, and sweep for expiries when the call returns. Linux also offers `timerfd_create`, which makes a timer into a descriptor the loop can wait on like any other.

Signals are harder. Chapter 39 established that a handler may do almost nothing — so the classic solution makes the signal into an event:

```
/* the self-pipe trick */
static int wake[2];                     /* pipe(wake) at startup */

static void on_signal(int sig)
{
    (void)sig;
    char b = 1;
    ssize_t r = write(wake[1], &b, 1);   /* write IS async-signal-safe */
    (void)r;
}
/* register wake[0] with epoll; the loop now sees signals as events */
```

Linux provides `signalfd`, which does the same thing without the pipe. Either way the signal is handled in the loop's ordinary flow, where the full standard library is available.

6 Worked example: the chapter 40 server, rewritten

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <netdb.h>

#define MAX_EVENTS   64
#define BUF_SIZE   4096
#define IDLE_LIMIT   30      /* seconds */

typedef struct {
    int     fd;
    char    in[BUF_SIZE];
```

```
    size_t  in_used;
    char    out[BUF_SIZE];
    size_t  out_used;
    size_t  out_sent;
    long    lines;
    time_t  last_active;
    bool    closing;
} Connection;

static Connection *conns[65536];
static int  epfd     = -1;
static int  wake[2]  = { -1, -1 };
static long live     = 0;
static long peak     = 0;
static long total    = 0;

/* ---------- signal handling via the self-pipe trick ---------- */

static void on_signal(int sig)
{
    (void)sig;
    char b = 1;
    ssize_t r = write(wake[1], &b, 1);    /* the only safe call here */
    (void)r;
}

static bool set_nonblocking(int fd)
{
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags < 0) return false;
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK) == 0;
}

/* ---------- connection lifecycle ---------- */

static void conn_close(Connection *c)
{
    epoll_ctl(epfd, EPOLL_CTL_DEL, c->fd, NULL);
    close(c->fd);
```

```
    conns[c->fd] = NULL;
    free(c);
    live--;
}

/* Ask for writability only while there is something to write. */
static void conn_update_interest(Connection *c)
{
    struct epoll_event ev = { .data.fd = c->fd, .events = EPOLLIN };
    if (c->out_used > c->out_sent) {
        ev.events |= EPOLLOUT;
    }
    epoll_ctl(epfd, EPOLL_CTL_MOD, c->fd, &ev);
}

static void conn_queue(Connection *c, const char *data, size_t n)
{
    if (c->out_used + n > BUF_SIZE) {      /* client not draining */
        c->closing = true;
        return;
    }
    memcpy(c->out + c->out_used, data, n);
    c->out_used += n;
}

/* Framing, exactly as in chapter 40: a stream has no message boundaries. */
static void conn_process_input(Connection *c)
{
    char *start = c->in;
    char *nl;
    while ((nl = memchr(start, '\n', (size_t)(c->in + c->in_used - start)))) {
        size_t len = (size_t)(nl - start);
        c->lines++;

        if (len == 4 && memcmp(start, "quit", 4) == 0) {
            conn_queue(c, "bye\n", 4);
            c->closing = true;
            break;
        }
```

```
        char reply[BUF_SIZE];
        int n = snprintf(reply, sizeof reply, "%ld: %.*s\n",
                         c->lines, (int)len, start);
        if (n > 0) conn_queue(c, reply, (size_t)n);
        start = nl + 1;
    }

    size_t leftover = (size_t)(c->in + c->in_used - start);
    memmove(c->in, start, leftover);
    c->in_used = leftover;

    if (c->in_used == BUF_SIZE) {          /* a line longer than the buffer */
        conn_queue(c, "line too long\n", 14);
        c->closing = true;
    }
}

static void conn_readable(Connection *c)
{
    for (;;) {
        if (c->in_used == BUF_SIZE) break;

        ssize_t n = recv(c->fd, c->in + c->in_used,
                         BUF_SIZE - c->in_used, 0);
        if (n > 0) {
            c->in_used += (size_t)n;
            c->last_active = time(NULL);
            conn_process_input(c);
            if (c->closing) break;
            continue;                      /* drain: there may be more */
        }
        if (n == 0) {                      /* orderly close */
            c->closing = true;
            c->out_used = c->out_sent;     /* nothing left to deliver */
            break;
        }
        if (errno == EAGAIN || errno == EWOULDBLOCK) break;   /* normal */
        if (errno == EINTR) continue;
```

```
        c->closing = true;                /* a real error */
        break;
    }
}

static void conn_writable(Connection *c)
{
    while (c->out_sent < c->out_used) {
        ssize_t n = send(c->fd, c->out + c->out_sent,
                         c->out_used - c->out_sent, MSG_NOSIGNAL);
        if (n > 0) {
            c->out_sent += (size_t)n;
            c->last_active = time(NULL);
            continue;
        }
        if (n < 0 && (errno == EAGAIN || errno == EWOULDBLOCK)) return;
        if (n < 0 && errno == EINTR) continue;
        c->closing = true;
        return;
    }
    c->out_used = c->out_sent = 0;        /* buffer drained */
}

static void accept_new(int listener)
{
    for (;;) {                            /* accept until EAGAIN */
        int fd = accept(listener, NULL, NULL);
        if (fd < 0) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) return;
            if (errno == EINTR) continue;
            perror("accept");
            return;
        }
        if (!set_nonblocking(fd)) { close(fd); continue; }

        Connection *c = calloc(1, sizeof *c);
        if (c == NULL) { close(fd); continue; }
        c->fd = fd;
        c->last_active = time(NULL);
```

```
        conns[fd] = c;

        struct epoll_event ev = { .events = EPOLLIN, .data.fd = fd };
        if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev) < 0) {
            close(fd); free(c); conns[fd] = NULL; continue;
        }

        conn_queue(c, "ready\n", 6);
        conn_update_interest(c);

        live++; total++;
        if (live > peak) peak = live;
    }
}

static void sweep_idle(void)
{
    time_t now = time(NULL);
    for (int fd = 0; fd < 65536; fd++) {
        Connection *c = conns[fd];
        if (c != NULL && now - c->last_active > IDLE_LIMIT) {
            conn_close(c);
        }
    }
}

int main(int argc, char *argv[])
{
    const char *port = (argc > 1) ? argv[1] : "8080";

    if (pipe(wake) < 0) { perror("pipe"); return EXIT_FAILURE; }
    set_nonblocking(wake[0]);
    set_nonblocking(wake[1]);

    struct sigaction sa;
    memset(&sa, 0, sizeof sa);
    sa.sa_handler = on_signal;
    sa.sa_flags   = SA_RESTART;
    sigaction(SIGINT,  &sa, NULL);
```

```
    sigaction(SIGTERM, &sa, NULL);
    signal(SIGPIPE, SIG_IGN);

    struct addrinfo hints;
    memset(&hints, 0, sizeof hints);
    hints.ai_family   = AF_INET6;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;

    struct addrinfo *list;
    int rc = getaddrinfo(NULL, port, &hints, &list);
    if (rc != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rc));
        return EXIT_FAILURE;
    }

    int listener = socket(list->ai_family, list->ai_socktype,
                          list->ai_protocol);
    int yes = 1;
    setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes);
    if (bind(listener, list->ai_addr, list->ai_addrlen) < 0) {
        perror("bind"); return EXIT_FAILURE;
    }
    freeaddrinfo(list);

    set_nonblocking(listener);
    listen(listener, 512);

    epfd = epoll_create1(0);
    struct epoll_event ev = { .events = EPOLLIN, .data.fd = listener };
    epoll_ctl(epfd, EPOLL_CTL_ADD, listener, &ev);
    ev.data.fd = wake[0];
    epoll_ctl(epfd, EPOLL_CTL_ADD, wake[0], &ev);

    printf("event loop on port %s, one thread, Ctrl-C to stop\n", port);

    struct epoll_event events[MAX_EVENTS];
    bool running = true;
    time_t last_sweep = time(NULL);
```

```
while (running) {
    int n = epoll_wait(epfd, events, MAX_EVENTS, 1000);
    if (n < 0) {
        if (errno == EINTR) continue;
        perror("epoll_wait");
        break;
    }

    for (int i = 0; i < n; i++) {
        int fd = events[i].data.fd;

        if (fd == listener) {
            accept_new(listener);
            continue;
        }
        if (fd == wake[0]) {           /* a signal arrived */
            char drain[64];
            while (read(wake[0], drain, sizeof drain) > 0) { }
            running = false;
            continue;
        }

        Connection *c = conns[fd];
        if (c == NULL) continue;

        if (events[i].events & (EPOLLHUP | EPOLLERR)) {
            conn_close(c);
            continue;
        }
        if (events[i].events & EPOLLIN)  conn_readable(c);
        if (events[i].events & EPOLLOUT) conn_writable(c);

        if (c->closing && c->out_sent >= c->out_used) {
            conn_close(c);              /* flush before closing */
        } else {
            conn_update_interest(c);
        }
    }
```

```
        if (time(NULL) - last_sweep >= 5) {
            sweep_idle();
            last_sweep = time(NULL);
        }
    }

    for (int fd = 0; fd < 65536; fd++) {
        if (conns[fd] != NULL) conn_close(conns[fd]);
    }
    close(listener);
    close(epfd);
    close(wake[0]);
    close(wake[1]);

    printf("\nserved %ld connections, peak concurrent %ld\n", total, peak);
    return EXIT_SUCCESS;
}
```

```
gcc -std=c17 -Wall -Wextra -g -o evserver evserver.c
./evserver 8080 &
```

Compare the two servers under load

```
# the chapter 40 forking server
./echoserver 8081 &

# 500 concurrent clients against each
time (for i in $(seq 500); do
        printf 'a\nb\nquit\n' | nc -q1 localhost 8081 >/dev/null &
      done; wait)

time (for i in $(seq 500); do
        printf 'a\nb\nquit\n' | nc -q1 localhost 8080 >/dev/null &
      done; wait)
```

Watch the process count while each runs:

```
ps --no-headers -C echoserver | wc -l     # hundreds
ps --no-headers -C evserver   | wc -l     # exactly 1
```

The forking server creates five hundred processes; the event loop stays at one thread and about 8 KB per connection. Raise the count to five thousand and the forking server will typically hit a resource limit while the event loop is still comfortable.

Keep many connections open and idle

```
for i in $(seq 1000); do (sleep 60 | nc localhost 8080 >/dev/null) & done
ps -o rss= -C evserver          # resident memory, in kilobytes
ss -tn state established '( sport = :8080 )' | wc -l
```

A thousand idle connections cost the event loop a few megabytes. A thread-per-connection server would have reserved eight gigabytes of stack address space for the same thing.

The two bugs this design invites

Forgetting to drain. Change `conn_readable` to call `recv` once instead of looping. Under level-triggered `epoll` it still works — you are told again next time — but each event does less work and throughput drops. Switch to `EPOLLET` with the loop removed and it breaks outright: data sits in the socket and you are never notified again. Add `EPOLLET` to the events and try it.

Leaving `EPOLLOUT` registered. Change `conn_update_interest` to always request `EPOLLOUT` and watch the CPU:

```
top -p $(pgrep evserver)
```

One connection is enough to pin a core at 100%. The socket is writable, so `epoll_wait` returns immediately, forever. This is the single most common event-loop bug and it looks like a mysterious performance problem rather than a logic error.

Verify the signal path

Press Ctrl-C. The handler writes one byte to the pipe; the loop sees `wake[0]` become readable, drains it, and exits cleanly — closing every connection and printing the statistics. Nothing unsafe happens inside the handler, which is chapter 39's rule honoured by construction rather than by care.

7 Event loop or thread pool?

EVENT LOOP	THREAD POOL
Many connections, little work each	Fewer connections, heavy work each
I/O bound	CPU bound
No locking; one thread	Uses every core
State machines instead of straight-line code	Straight-line code, plus locks
One blocking call stalls everything	A blocking call stalls one worker

Real servers combine them: an event loop per core, each handling its own connections, with a separate pool for genuinely blocking work such as disk I/O or password hashing. nginx, Redis, and Node.js are all variations on that shape.

The "C10K problem" — handling ten thousand concurrent connections — was an open question around 2000 precisely because the thread-per-connection model could not reach it. `epoll` and `kqueue` were the answer, and the same techniques now carry C10M on commodity hardware.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Treating `EAGAIN` as an error	Connections dropped for no reason	It means "not ready"; try again later.
`EPOLLOUT` always registered	100% CPU on an idle server	Register it only when output is pending.
Not draining under `EPOLLET`	Data never delivered; the connection hangs	Loop until `EAGAIN`.
A blocking call inside the loop	Every connection stalls	Move it to a worker thread.
Doing work in a signal handler	Undefined behavior	Self-pipe or `signalfd`.
Freeing a connection still registered	Use after free on the next event	`EPOLL_CTL_DEL` before `free`.
No idle timeout	Dead connections accumulate	Sweep periodically.
Closing before the output drains	The last reply is lost	Close once `out_sent == out_used`.
Unbounded output buffering	A slow client exhausts memory	Cap it and disconnect.

9 Check yourself

Why is `epoll` O(ready) while `poll` is O(n)?

Because `poll` passes the whole descriptor list on every call, so the kernel must examine each one. `epoll` registers the set once and keeps it in the kernel, maintaining a ready list as events occur — so a call returns only the descriptors that are ready, regardless of how many are being watched.

What does `EAGAIN` mean and why is it not an error?

On a non-blocking descriptor it means the operation would have blocked — there is no data to read, or no space to write, right now. The correct response is to return to the event loop and wait to be told the descriptor is ready. Treating it as a failure closes connections that are perfectly healthy.

Why must you drain a descriptor under edge-triggered mode?

Because edge-triggered `epoll` reports a transition, not a state: you are told once when data arrives and not again until more arrives. If you read only part of what is available, the remainder sits in the socket with no further notification, and the connection appears to hang. Read in a loop until `EAGAIN`.

Why does leaving `EPOLLOUT` registered burn a whole core?

Because a socket is writable almost all the time, so `epoll_wait` returns immediately on every call with nothing useful to do. The loop spins at full speed. Request writability only while the output buffer is non-empty and remove it as soon as the buffer drains.

When is a thread pool the better choice?

When the work per connection is CPU-bound rather than I/O-bound, or when it genuinely must block — disk reads, password hashing, a synchronous third-party call. An event loop is one thread, so it cannot use more than one core and one blocking call stalls everything. Real servers usually combine both: an event loop per core plus a pool for blocking work.

10 Where this leads

Chapter 48 attacks the parsers you have been writing — including this server's framing loop — with a fuzzer, which generates millions of inputs and finds the ones you did not think of. After that the embedded block begins, where the constraints change completely and an event loop becomes a superloop.

PROFESSIONAL · CHAPTER 48

Fuzzing and Security Testing

Your tests check the inputs you thought of. A fuzzer generates millions you did not, guided by which ones reach new code — and in C, where a malformed input can be a remote code execution, it is the single most effective testing technique available.

BY THE END OF THIS CHAPTER YOU CAN

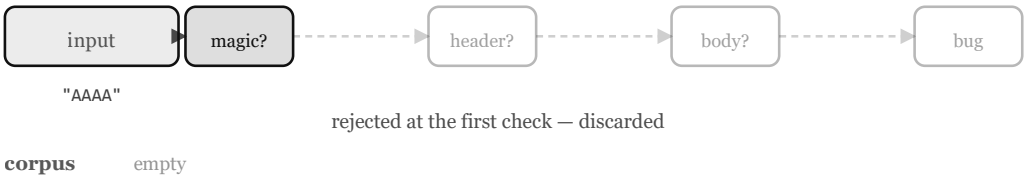
- Write a libFuzzer target and run it against your own parser.
- Explain why coverage guidance makes fuzzing effective rather than random.
- Combine a fuzzer with sanitizers so failures are detected, not merely survived.
- Triage a crash and minimize the input that causes it.
- Threat-model a C program and review it against the resulting list.

Code for this chapter: github.com/sercankulcu/programming-with-c

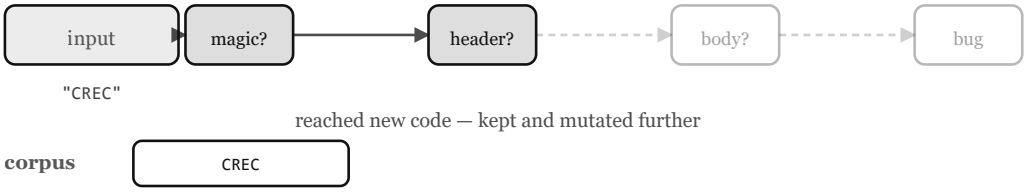
1 Why fuzzing works

Random input alone is nearly useless: the chance of generating a valid file header by accident is negligible, so a naive fuzzer spends its life being rejected at the first check.

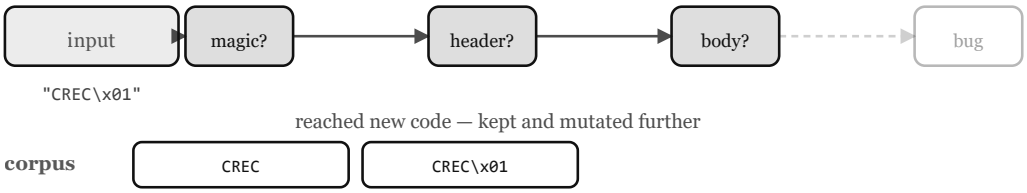
Coverage-guided fuzzing changes the game. The program is instrumented so the fuzzer can see which branches an input reached. An input that reaches new code is kept and mutated further; one that does not is discarded. The result is a search that learns the input format without being told it.



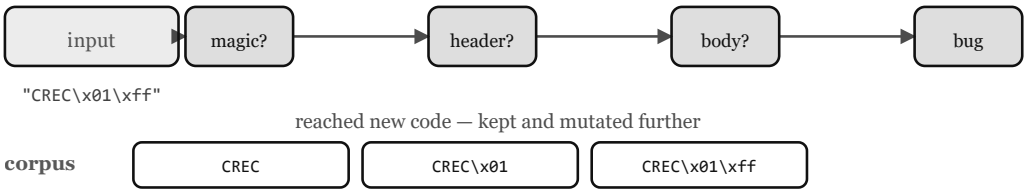
Random bytes die at the first check. A fuzzer that only generated random input would spend its life here, which is why naive fuzzing finds nothing.



Coverage instrumentation changes it. This input reached a branch no earlier input reached, so it is *kept* — not because anyone told the fuzzer what a valid header is, but because the coverage counter moved.



Mutations of a kept input are tried next. One gets past the header check, reaches new code again, and is kept in turn.



the fuzzer learned the format without being told it

A few million iterations of that and the corpus contains inputs that reach deep into the parser. The fuzzer derived the input format from nothing but which branches ran.

```
input "AAAA"      → rejected at the magic check          → discard
input "CREC"      → reached the version check (new!)     → KEEP, mutate
input "CREC\x01"  → reached the length field (new!)      → KEEP, mutate
input "CREC\x01\xff\xff\xff\xff" → allocation overflow   → CRASH
```

Each step is a small mutation of something that worked. Within minutes a fuzzer can discover a file format that took a person an afternoon to specify — and then explore its edges far more patiently than any person would.

This is why fuzzing found thousands of bugs in OpenSSL, SQLite, and the Linux kernel, all of which had extensive test suites. Tests encode what the author expected; a fuzzer explores what the author did not.

2 Writing a target

libFuzzer ships with Clang and needs one function:

```
#include <stdint.h>
#include <stddef.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    parse_thing(data, size);      /* the code under test */
    return 0;                     /* non-zero is reserved */
}
```

```
clang -g -O1 -fsanitize=fuzzer,address,undefined \
      -o fuzz_parser fuzz_parser.c parser.c
./fuzz_parser corpus/ -max_len=4096
```

That single command compiles with instrumentation, links the fuzzing engine, and runs. There is no main.

A GOOD TARGET	WHY
Is deterministic	The same input must produce the same result, or crashes cannot be reproduced
Is fast	Throughput is everything; aim for thousands of executions per second
Touches no global state	One iteration must not affect the next
Frees what it allocates	Otherwise the leak detector fires on every input
Exercises one entry point	Several targets beat one that branches

Avoid anything involving time, randomness, the network, or the filesystem inside the target. If the code under test needs them, stub them out for the fuzz build.

The seed corpus

Start the fuzzer with valid examples rather than nothing:

```
mkdir corpus
printf 'timeout = 30\nname = server\n' > corpus/valid.conf
printf '# comment only\n'              > corpus/comment.conf
printf ''                              > corpus/empty.conf
```

Seeds save the fuzzer from rediscovering the format. Include the edge cases you already know — empty, maximum size, each optional field present and absent. A dictionary helps too, giving the mutator the keywords to try:

```
# parser.dict
kw1="timeout"
kw2="name"
kw3="="
./fuzz_parser corpus/ -dict=parser.dict
```

3 Fuzzing finds nothing without sanitizers

This is the point people miss. A fuzzer detects crashes, hangs, and assertion failures. Most C memory bugs produce none of those — chapter 20's silent corruption runs to completion with a wrong answer.

Sanitizers convert those silent defects into immediate, loud failures:

FLAG	TURNS INTO A CRASH
`-fsanitize=address`	Overruns, use-after-free, double free, leaks
`-fsanitize=undefined`	Signed overflow, bad shifts, misaligned access
`-fsanitize=memory`	Uninitialized reads (separate build)

Fuzzing without sanitizers finds a fraction of what is there. The fuzzer supplies the inputs; the sanitizers supply the detection. Neither is useful alone.

Assertions help too, because a fuzzer treats an abort as a finding. An `assert` stating an invariant turns a logic error into something the fuzzer can discover — which is an argument for building the fuzz target *without* `NDEBUG`.

4 Triage and minimization

When a crash is found, libFuzzer writes the input to `crash-<hash>`. Reproduce it directly:

```
./fuzz_parser crash-a1b2c3d4              # replays that one input
xxd crash-a1b2c3d4 | head
```

Then shrink it, because a 4000-byte input tells you nothing:

```
./fuzz_parser -minimize_crash=1 -runs=100000 crash-a1b2c3d4
```

The minimizer repeatedly removes bytes while checking the crash still occurs — chapter 43's bisection applied to data rather than history. Four thousand bytes typically reduce to fewer than twenty, and the defect is usually obvious once they do.

Then make it a permanent test:

```
cp crash-a1b2c3d4 tests/regressions/overflow-in-length-field
# the test suite replays every file in that directory
```

A fuzzer finding the same bug twice is wasted machine time. The corpus of past crashes is as valuable as the code that fixed them.

5 Threat modeling

Fuzzing finds memory-safety bugs. It does not tell you what an attacker wants. Four questions, answered before writing the code:

1. **What is worth protecting?** Credentials, user data, the ability to execute code, availability.
2. **Where does untrusted input enter?** Command-line arguments, files, the network, and — as chapter 31 noted — the environment.
3. **What can go wrong?** For C specifically: memory corruption, integer overflow, injection, resource exhaustion.
4. **What stops it?** Validation at the boundary, bounded copies, checked arithmetic, limits on allocation.

The *trust boundary* is the central idea. Data crossing it must be validated exactly once, at the crossing, and everything inside may then assume it is well-formed. Validating repeatedly everywhere produces code where nobody is sure whether the check already happened.

C-SPECIFIC THREAT	DEFENCE	CHAPTER
Buffer overflow	Bounded copies, `snprintf`, ASan	14, 20
Integer overflow in a size	`calloc`, explicit range check	36
Format string	`printf("%s", user)`, `-Wformat-security`	36
Use after free	`NULL` after `free`, clear ownership	19, 20
Unbounded allocation	Cap sizes from untrusted sources	40
Path traversal	Canonicalize, then verify the prefix	—
Hash collision denial of service	Randomized hash seed	33

6 Worked example: fuzzing the chapter 23 parser

Take the configuration parser and attack it. The version below contains three defects deliberately — each one plausible, each one a real pattern.

The interface first. `Config` is opaque, so the fuzzer can only reach this code through the same door a real caller would use — chapter 28's technique, here for a different reason: it keeps the fuzzer honest.

```
/* parser.h */
#ifndef PARSER_H
#define PARSER_H

#include <stdint.h>
#include <stddef.h>

typedef struct Config Config;

/* Parse size bytes of configuration text. Returns NULL on allocation
   failure. The result must be released with config_free. */
Config *config_parse(const uint8_t *data, size_t size);

void config_free(Config *c);

#endif /* PARSER_H */
```

```
/* parser.c — with three planted defects */
#include "parser.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>

#define MAX_NAME 32

struct Config {
    int    timeout;
    int    retries;
    char   name[MAX_NAME];
    int   *values;
    size_t value_count;
};

Config *config_parse(const uint8_t *data, size_t size)
{
    Config *c = calloc(1, sizeof *c);
    if (c == NULL) return NULL;
```

```
    char *text = malloc(size + 1);
    if (text == NULL) { free(c); return NULL; }
    memcpy(text, data, size);
    text[size] = '\0';

    char *save = NULL;
    for (char *line = strtok_r(text, "\n", &save);
         line != NULL;
         line = strtok_r(NULL, "\n", &save)) {

        if (line[0] == '#' || line[0] == '\0') continue;

        char *eq = strchr(line, '=');
        if (eq == NULL) continue;
        *eq = '\0';
        char *key   = line;
        char *value = eq + 1;

        while (*key == ' ')   key++;
        while (*value == ' ') value++;

        if (strcmp(key, "timeout") == 0) {
            c->timeout = atoi(value);                 /* DEFECT 1 */
        } else if (strcmp(key, "name") == 0) {
            strcpy(c->name, value);                   /* DEFECT 2 */
        } else if (strcmp(key, "values") == 0) {
            size_t n = (size_t)atoi(value);
            c->values = malloc(n * sizeof *c->values); /* DEFECT 3 */
            c->value_count = n;
            for (size_t i = 0; i < n; i++) c->values[i] = 0;
        }
    }

    free(text);
    return c;
}

void config_free(Config *c)
```

```
{
    if (c == NULL) return;
    free(c->values);
    free(c);
}
```

The fuzz target

```
/* fuzz_parser.c */
#include "parser.h"
#include <stdint.h>
#include <stddef.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    if (size > 4096) return 0;              /* keep iterations fast */

    Config *c = config_parse(data, size);
    config_free(c);                          /* free, or the leak checker fires */
    return 0;
}
```

```
clang -g -O1 -fsanitize=fuzzer,address,undefined \
      -o fuzz_parser fuzz_parser.c parser.c

mkdir -p corpus
printf 'timeout = 30\nname = server\n' > corpus/valid
printf '# comment\n'                   > corpus/comment
printf 'values = 4\n'                  > corpus/values

./fuzz_parser corpus/ -max_len=4096 -print_final_stats=1
```

What it finds, and how quickly

Within seconds — the name overflow.

```
==12345==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602…
WRITE of size 45 at 0x7ffd… thread T0
    #0 0x… in strcpy
    #1 0x… in config_parse parser.c:44

0x602… is located 0 bytes after 56-byte region
  'c' … 'name' (line 12) <== overflows this variable
artifact_written: crash-3f2a1b…
```

`strcpy` into `char name[32]` with no bound — chapter 14's defect, found in the time it takes to generate a long line. It is a classic heap overflow: the next thing overwritten is the `values` pointer, which `config_free` then passes to `free`.

Within a minute — the allocation overflow.

```
==12346==ERROR: AddressSanitizer: requested allocation size 0x…
exceeds maximum supported size
```

Or, worse on some platforms, no error at all: `atoi` returns a negative number, the conversion to `size_t` makes it enormous, the multiplication wraps, `malloc` succeeds with a tiny block, and the loop writes past it. Chapter 36's integer-overflow-to-heap-overflow chain, discovered automatically.

The silent one. `atoi` cannot report failure, so `timeout = abc` becomes 0. No crash, so the fuzzer is blind to it — which is the limitation worth naming. Add an assertion and out-of-range values become findable (though `abc` → 0 still passes it):

```
assert(c->timeout >= 0 && c->timeout <= 3600);
```

Minimize each crash

```
./fuzz_parser -minimize_crash=1 -runs=100000 crash-3f2a1b…
xxd minimized-from-3f2a1b…
```

```
00000000: 6e61 6d65 203d 2041 4141 4141 4141 4141  name = AAAAAAAAA
00000010: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000020: 4141 0a                                  AA.
```

Fifty-five bytes, and the defect is unmistakable from the input alone.

The fixed parser

```
if (strcmp(key, "timeout") == 0) {
    long v;
    if (!parse_long(value, &v) || v < 0 || v > 3600) {
        goto invalid;                          /* refuse, do not default */
    }
    c->timeout = (int)v;

} else if (strcmp(key, "name") == 0) {
    int n = snprintf(c->name, sizeof c->name, "%s", value);
    if (n < 0 || (size_t)n >= sizeof c->name) {
        goto invalid;                          /* reject, do not truncate */
    }

} else if (strcmp(key, "values") == 0) {
    long n;
    if (!parse_long(value, &n) || n < 0 || n > MAX_VALUES) {
        goto invalid;                          /* bound it BEFORE allocating */
    }
    c->values = calloc((size_t)n, sizeof *c->values);   /* checks overflow */
    if (c->values == NULL && n > 0) goto invalid;
    c->value_count = (size_t)n;
}
```

Three changes, each from an earlier chapter: `strtol` with a range check (23), `snprintf` with truncation detected (14), and a bound before `calloc` (36). Rebuild and run the fuzzer again for ten minutes — it should find nothing.

Keep the crashes as tests

```
mkdir -p tests/regressions
cp minimized-* tests/regressions/
# and in CI:
for f in tests/regressions/*; do ./fuzz_parser "$f" || exit 1; done
```

Every past crash replays on every build, in milliseconds. This is the highest-value output of a fuzzing session — higher than the fix, because the fix can be undone and the test cannot be undone quietly.

Run it continuously

```
# overnight, with a persistent corpus
./fuzz_parser corpus/ -max_total_time=28800 -jobs=8 -workers=8


# AFL++, a different mutation strategy — worth running both
afl-clang-fast -fsanitize=address -o parser_afl parser_afl.c parser.c
afl-fuzz -i corpus -o findings -- ./parser_afl
```

Fuzzing is not a one-off. Projects that take it seriously run it continuously — OSS-Fuzz does exactly this for hundreds of open-source C projects — because new code introduces new paths, and a corpus grown over months explores far deeper than one grown over minutes.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Fuzzing without sanitizers	Silent corruption goes undetected	Always `-fsanitize=address,undefined`.
A slow target	Hundreds of executions per second instead of thousands	Remove I/O, cap the input size.
Non-deterministic target	Crashes cannot be reproduced	No time, randomness, or network.
Leaking in the target	Every input reports a leak	Free everything each iteration.
No seed corpus	Never gets past the first check	Provide valid examples and a dictionary.
Not minimizing a crash	4000 bytes of noise to read	`-minimize_crash=1`.
Discarding crash inputs after fixing	The bug returns unnoticed	Keep them as regression tests.
Fuzzing once before release	New code is never explored	Run continuously.
Expecting logic errors to be found	A wrong answer is not a crash	Add assertions on invariants.

8 Check yourself

What makes coverage-guided fuzzing more effective than random input?

Feedback. The program is instrumented so the fuzzer sees which branches each input reached; inputs that reach new code are kept and mutated further, and the rest are discarded. That turns an astronomically unlikely search into an incremental one, and lets the fuzzer discover an input format it was never told about.

Why is fuzzing without sanitizers close to useless in C?

Because a fuzzer can only observe crashes, hangs, and aborts, and most C memory bugs produce none of those — an overrun quietly corrupts a neighbour and the program finishes normally. Sanitizers turn those silent defects into immediate failures, so the fuzzer supplies the inputs and the sanitizer supplies the detection.

What are the three properties a fuzz target must have?

Deterministic, so a crash can be reproduced; fast, because throughput determines how much of the input space is explored; and free of state carried between iterations, including leaks, so each input is judged on its own. Time, randomness, network, and filesystem access all violate at least one of these.

Why minimize a crashing input before debugging it?

Because a four-thousand-byte input contains mostly irrelevant bytes, and reasoning about it is slow. The minimizer removes bytes while checking the crash persists, typically reducing it to a few dozen — at which point the defect is usually visible from the input alone. It is chapter 43's bisection applied to data.

Why keep the crashing inputs after fixing the bug?

Because they are ready-made regression tests that replay in milliseconds and prove the fix stays. Without them a later refactor can silently reintroduce the defect, and the fuzzer will spend hours rediscovering something you already knew. The saved corpus is often more durable value than the fix itself.

9 Where this leads

Chapter 49 begins the embedded block. Everything changes: no operating system, no `malloc`, a few kilobytes of RAM, and a compiler producing code for a different processor than the one it runs on. The discipline built over the last forty-eight chapters is exactly what makes that environment tractable.

PROFESSIONAL · CHAPTER 49

Embedded C and Cross-Compilation

No operating system, no `malloc`, no `printf`, sixty-four kilobytes of RAM, and a compiler producing code for a processor it is not running on. This is where C's original purpose is still its dominant one — and where every assumption of the previous forty-eight chapters has to be re-examined.

BY THE END OF THIS CHAPTER YOU CAN

- Describe a microcontroller's memory and peripherals, and how it differs from a desktop.
- Install a cross-compiler and explain what a target triple names.
- Say what a freestanding environment provides and what it does not.
- Trace execution from the reset vector to the first line of `main`.
- Build a bare-metal image and run it under QEMU, with no hardware.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The machine

	DESKTOP	MICROCONTROLLER
RAM	8–64 GB	4 KB – 512 KB
Program storage	SSD, gigabytes	Flash, 16 KB – 2 MB
Clock	3 GHz, several cores	16–200 MHz, one core
Operating system	Linux, Windows	None, or a small RTOS
Memory protection	MMU; a bad pointer faults	Usually none; a bad pointer corrupts
Allocation	`malloc` freely	Often forbidden entirely
Output	`printf` to a terminal	A UART, if you write the driver

Two consequences deserve emphasis now. Without an MMU there is no segmentation fault — a wild pointer writes over another variable, or over a peripheral register, and the program continues doing something else. Chapter 12's silent corruption is the *normal* failure mode here, not the unlucky one.

And code executes from flash while data lives in RAM, which is why the startup sequence in section 4 has work to do before `main` can run at all.

Harvard versus von Neumann

A desktop keeps code and data in one address space. Many microcontrollers separate them (an AVR has distinct flash and RAM address spaces); even a Cortex-M, which keeps one address space, puts flash at `0x08000000` and RAM at `0x20000000` — which is why a `const` table can stay in flash and cost no RAM, and why a string literal is addressed differently from a buffer.

2 Cross-compilation

A *cross-compiler* runs on one architecture and produces code for another. Its name encodes the target as a **triple**:

```
arm-none-eabi-gcc
|   |    |
|   |    └── ABI: embedded ABI
|   └─────── vendor/OS: none — bare metal
└─────────── architecture: ARM


x86_64-linux-gnu-gcc        # the compiler you have been using
riscv64-unknown-elf-gcc     # bare-metal RISC-V
```

```
# Debian/Ubuntu
sudo apt install gcc-arm-none-eabi qemu-system-arm gdb-multiarch


# Fedora
sudo dnf install arm-none-eabi-gcc-cs qemu-system-arm


arm-none-eabi-gcc --version
```

The whole toolchain is prefixed: `arm-none-eabi-objdump`, `-size`, `-nm`, `-gdb`. Running the host `objdump` on an ARM image is a common early confusion — it will refuse, or print nonsense.

3 Freestanding versus hosted

The standard defines two conformance modes, and this is the distinction that matters most this chapter.

	HOSTED	FREESTANDING
Entry point	`main`, called for you	Implementation-defined; you write it
Guaranteed headers	All of them	`<float.h> <limits.h> <stdarg.h> <stdbool.h> <stddef.h> <stdint.h>` and a few more
`printf`, `malloc`, `fopen`	Present	**Not guaranteed**

```
arm-none-eabi-gcc -ffreestanding -nostdlib …
```

Note what *is* guaranteed: the language itself, the fixed-width types, and the limits. Everything chapter 37 taught about `<stdint.h>` applies directly, and it matters more here because `int` may genuinely be 16 bits on a small part.

In practice a toolchain like `arm-none-eabi` ships newlib, which provides a subset — including a `printf` that will link but calls `_write`, which you must supply. That is why a first embedded `printf` so often links cleanly and produces nothing.

4 From reset to `main`

On a desktop the loader does this work. Bare metal, you write it.



only loading SP and jumping to the reset handler happen by themselves; the rest is code you write

Chapter 17's `.data` and `.bss` regions, initialized by hand.

On a Cortex-M the processor reads two words from address 0 at reset: the initial stack pointer, then the address of the reset handler. Everything after that is your code.

`.data` — initialized globals — exists in two places. The initial values live in flash, because RAM is empty at power-on; the variables live in RAM. The startup code copies one to the other. `.bss` — zero-initialized globals — occupies no flash at all and is simply cleared, which is exactly what chapter 17's `size` output showed from the other side.

Omit either step and the symptoms are memorably strange: globals hold garbage, or hold their values only until something else uses that RAM.

5 Worked example: bare metal under QEMU

The target is `lm3s6965evb`, a Cortex-M3 machine QEMU emulates and whose UART is trivial to drive. No hardware is required.

The linker script

```
/* firmware.ld — chapter 52 covers this properly */
MEMORY
{
    FLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
    RAM   (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}

_stack_top = ORIGIN(RAM) + LENGTH(RAM);     /* stack grows down from the top */

SECTIONS
{
    .text : {
        KEEP(*(.vectors))       /* the vector table must be first */
        *(.text*)
        *(.rodata*)             /* constants stay in flash */
    } > FLASH

    _data_load = LOADADDR(.data);

    .data : {
        _data_start = .;
        *(.data*)
        _data_end = .;
    } > RAM AT> FLASH           /* lives in RAM, stored in FLASH */
```

```
    .bss : {
        _bss_start = .;
        *(.bss*)
        *(COMMON)
        _bss_end = .;
    } > RAM
}
```

`> RAM AT> FLASH` is the line that expresses the two-places problem: the section is addressed in RAM but stored in flash.

The startup code

```
/* startup.c */
#include <stdint.h>

extern uint32_t _stack_top;
extern uint32_t _data_load, _data_start, _data_end;
extern uint32_t _bss_start, _bss_end;

int main(void);

void reset_handler(void)
{
    /* 1. copy .data from flash to RAM */
    uint32_t *src = &_data_load;
    for (uint32_t *dst = &_data_start; dst < &_data_end; ) {
        *dst++ = *src++;
    }

    /* 2. zero .bss */
    for (uint32_t *p = &_bss_start; p < &_bss_end; ) {
        *p++ = 0;
    }

    /* 3. only now may C code with globals run */
    main();

    for (;;) { }                /* main must never return */
```

```
}

static void default_handler(void) { for (;;) { } }

/* The vector table: the processor reads [0] and [1] at reset. */
__attribute__((section(".vectors"), used))
void (* const vector_table[])(void) = {
    (void (*)(void))&_stack_top,   /* [0] initial stack pointer */
    reset_handler,                 /* [1] reset                 */
    default_handler,               /* NMI                       */
    default_handler,               /* hard fault                */
};
```

The program

```
/* main.c */
#include <stdint.h>

/* The UART, straight from the machine's memory map. Chapter 50
   explains volatile properly; note it is not optional here. */
#define UART0_BASE 0x4000C000u
#define UART_DR   (*(volatile uint32_t *)(UART0_BASE + 0x000))
#define UART_FR   (*(volatile uint32_t *)(UART0_BASE + 0x018))
#define UART_FR_TXFF (1u << 5)          /* transmit FIFO full */

/* Initialized: lives in .data, copied from flash by the startup code. */
static uint32_t boot_marker = 0xDEADBEEF;

/* Zero-initialized: lives in .bss, cleared by the startup code. */
static uint32_t counter;
static uint8_t  buffer[256];

static void uart_putc(char c)
{
    while (UART_FR & UART_FR_TXFF) { }   /* wait for space */
    UART_DR = (uint32_t)c;
}
```

```
static void uart_puts(const char *s)
{
    for (; *s != '\0'; s++) {
        if (*s == '\n') uart_putc('\r');  /* terminals want CR LF */
        uart_putc(*s);
    }
}

/* No printf here: write what you need. */
static void uart_put_hex(uint32_t v)
{
    static const char digits[] = "0123456789ABCDEF";
    uart_puts("0x");
    for (int shift = 28; shift >= 0; shift -= 4) {
        uart_putc(digits[(v >> shift) & 0xFu]);
    }
}

static void uart_put_uint(uint32_t v)
{
    char tmp[11];
    int i = 0;
    if (v == 0) { uart_putc('0'); return; }
    while (v > 0) { tmp[i++] = (char)('0' + v % 10); v /= 10; }
    while (i > 0) uart_putc(tmp[--i]);
}

int main(void)
{
    uart_puts("\n=== bare metal, no operating system ===\n");

    uart_puts(".data was copied : ");
    uart_put_hex(boot_marker);
    uart_puts(boot_marker == 0xDEADBEEF ? "  correct\n" : "  WRONG\n");

    uart_puts(".bss was zeroed  : counter = ");
    uart_put_uint(counter);
    uart_puts(counter == 0 ? "  correct\n" : "  WRONG\n");
```

```
    uart_puts("buffer[0]        : ");
    uart_put_uint(buffer[0]);
    uart_puts("\n");

    uart_puts("stack address    : ");
    uint32_t local = 0;
    uart_put_hex((uint32_t)(uintptr_t)&local);
    uart_puts("  (RAM, near the top)\n");

    uart_puts("code address     : ");
    uart_put_hex((uint32_t)(uintptr_t)main);
    uart_puts("  (flash, low)\n");

    uart_puts("\nsuperloop starts; Ctrl-A X to quit QEMU\n");
    for (;;) {
        counter++;
        if (counter % 2000000u == 0) {
            uart_puts("tick ");
            uart_put_uint(counter / 2000000u);
            uart_puts("\n");
        }
    }
}
```

Build and run

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb \
    -ffreestanding -nostdlib -O2 -Wall -Wextra -g \
    -T firmware.ld -o firmware.elf startup.c main.c -lgcc

arm-none-eabi-objcopy -O binary firmware.elf firmware.bin
arm-none-eabi-size firmware.elf

qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf
```

```
=== bare metal, no operating system ===
.data was copied : 0xDEADBEEF  correct
.bss was zeroed  : counter = 0  correct
buffer[0]        : 0
stack address    : 0x2000FFE8  (RAM, near the top)
code address     : 0x000000C1  (flash, low)

superloop starts; Ctrl-A X to quit QEMU
tick 1
tick 2
```

Quit with Ctrl-A then X.

Four things the output proves

The startup code works. `boot_marker` holds its initializer only because `reset_handler` copied it out of flash; `counter` is zero only because the same function cleared `.bss`. Comment out the `.data` loop, rebuild, and watch that line report WRONG. Removing the `.bss` loop may still show 0 under QEMU, whose emulated RAM starts zeroed; on real hardware the value is arbitrary.

The memory map is visible. The stack sits near `0x20010000`, the top of the 64 KB RAM region, and grows down; `main` lives at a low address in flash. Chapter 17's diagram, with the numbers from the linker script.

There is no `printf`. Formatting a number took twelve lines. That is the ordinary situation on a small target, and it is why embedded code tends to have hand-written output helpers.

The odd address for `main`. `0x000000C1` is odd because the low bit of a Cortex-M function pointer indicates Thumb instruction encoding — a detail invisible from C and instantly visible here.

Inspect the image

```
arm-none-eabi-size firmware.elf
```

```
   text    data     bss     dec     hex filename
   1284       4     264    1552     610 firmware.elf
```

1284 bytes of code and constants in flash, 4 bytes of initialized data, 264 bytes of zeroed data. Compare with the same program built for the host: a hosted `hello world` links megabytes of C library. Here the entire image is smaller than this paragraph's HTML.

```
arm-none-eabi-objdump -h firmware.elf          # section addresses
arm-none-eabi-objdump -d firmware.elf | head -30
arm-none-eabi-nm -n firmware.elf | head -20
xxd -l 16 firmware.bin                          # the vector table
```

The first eight bytes of `firmware.bin` are the initial stack pointer and the reset handler address, little-endian — the two words the processor reads at power-on. Seeing them is worth more than any description.

Debug it with GDB

```
# terminal 1
qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf -S -s

# terminal 2
gdb-multiarch firmware.elf
(gdb) target remote :1234
(gdb) break reset_handler
(gdb) continue
(gdb) next
(gdb) print _data_start
(gdb) break main
(gdb) continue
(gdb) print boot_marker
```

`-S` halts the machine at reset and `-s` opens a GDB server on port 1234. You can now single-step the startup code, watch `.data` being copied word by word, and stop at `main` — the same debugger workflow as chapter 35, on a machine that does not exist. Chapter 55 develops this.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Using the host toolchain on a target image	Nonsense output or a refusal	Prefix everything: `arm-none-eabi-`.
Forgetting to copy `.data`	Initialized globals hold garbage	The copy loop in the reset handler.
Forgetting to zero `.bss`	Zero-initialized globals do not start at zero	The clear loop.
Vector table not first in flash	The processor reads a bad stack pointer; nothing runs	`KEEP(*(.vectors))` at the start of `.text`.
Letting `main` return	Execution runs off into undefined memory	An infinite loop after the call.
Expecting `printf` to work	Links, prints nothing	Provide `_write`, or write your own output.
Omitting `volatile` on a register	Reads cached; the driver hangs	Chapter 50.
Assuming a bad pointer will fault	Silent corruption; no MMU	Treat every pointer as unchecked.

7 Check yourself

What does the `none` in `arm-none-eabi-gcc` mean?

It is the vendor or operating-system field of the target triple, and `none` means bare metal — there is no operating system. That is what selects a freestanding environment: no process loader, no system calls, and no guarantee of the hosted standard library.

Why must the startup code copy `.data` before calling `main`?

Because initialized globals live in RAM, which is uninitialized at power-on, while their initial values are stored in flash. Nothing copies one to the other automatically on bare metal. Skip the copy and every initialized global holds whatever the RAM happened to contain.

Why does `.bss` occupy no space in the image?

Because every byte is zero, so only the size needs recording — the startup code clears that much RAM. This is the same property chapter 17 observed with `size` on a hosted program; here you write the loop that does it rather than relying on the loader.

What does a freestanding implementation guarantee?

The language itself and a small set of headers — `<stddef.h>`, `<stdint.h>`, `<limits.h>`, `<stdbool.h>`, `<stdarg.h>`, `<float.h>` and a few more. It does not guarantee `printf`, `malloc`, file I/O, or even that `main` is the entry point.

Why is a wild pointer more dangerous on a microcontroller than on a desktop?

Because there is usually no memory management unit, so there is no invalid address to fault on. The write lands on another variable, on the stack, or on a peripheral register, and execution continues. What is an immediate segmentation fault on a desktop becomes silent corruption with a symptom somewhere else entirely.

8 Where this leads

The UART macros in this chapter's example were used without explanation. Chapter 50 justifies them: what a memory-mapped register is, why `volatile` is mandatory rather than advisable, and how to turn a reference manual's register table into correct C.

PROFESSIONAL · CHAPTER 50

Memory-Mapped I/O and Hardware Registers

On a microcontroller, hardware is an address. Writing to `0x4000C000` transmits a character; reading it receives one. That is the entire interface — and it is also where `volatile` stops being a curiosity from chapter 38 and becomes the difference between a working driver and one that hangs.

BY THE END OF THIS CHAPTER YOU CAN

- Read a reference manual's register table and turn it into correct C.
- Explain precisely what the compiler does to register access without `volatile`.
- Choose between macro and struct-overlay styles for register definitions.
- Write the read-modify-write sequence correctly.
- Implement a working UART driver against an emulated peripheral.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The memory map

A microcontroller's address space is divided between flash, RAM, and *peripherals*. A peripheral occupies a block of addresses whose bits are wired to hardware rather than to memory cells.

REGION	TYPICAL RANGE	CONTAINS
Flash	`0x00000000` –	Code and constants
RAM	`0x20000000` –	Variables, stack
Peripherals	`0x40000000` –	UART, GPIO, timers, ADC
Core peripherals	`0xE0000000` –	Interrupt controller, systick

Three properties distinguish a register from a variable, and each has a consequence in C:

- **It can change without your code writing it.** A received byte appears; a timer counts. The compiler cannot know this.

- **Reading can have an effect.** On many UARTs, reading the data register removes the byte from the receive FIFO. A "redundant" read is not redundant.
- **Writing can have an effect beyond storing.** A bit may be write-one-to-clear, or trigger a transmission.

Every one of those violates an assumption the optimizer is otherwise entitled to make.

2 Why `volatile` is mandatory

```
#define UART_FR (*(uint32_t *)0x4000C018)      /* MISSING volatile */

while (UART_FR & (1u << 5)) { }                /* wait while FIFO full */
```

The compiler reasons: the loop body is empty, nothing in it modifies `UART_FR`, so the value cannot change. It loads the hardware register once into a CPU register and tests that copy forever. At `-O0` this often works by accident; at `-O2` the program hangs on the first full FIFO and never recovers.

```
#define UART_FR (*(volatile uint32_t *)0x4000C018)   /* correct */
```

With `volatile`, every read in the source is a read of memory, every write is a write, and their order is preserved. That is exactly the guarantee hardware needs — and exactly what chapter 38 said `volatile` does and does not provide.

WITHOUT `VOLATILE`, THE COMPILER MAY	CONSEQUENCE
Hoist a read out of a loop	Polling never terminates
Delete a write whose value is overwritten	A command never reaches the device
Merge two reads into one	Only one byte is taken from a FIFO
Reorder accesses to different registers	Enable happens after use

> `volatile` **is necessary, not sufficient.** It orders accesses as the compiler sees them; it does not prevent the *processor* reordering them, which matters on a chip with a write buffer or out-of-order execution. Cortex-M is mostly strongly ordered, but where a peripheral requires a completed write before the next access, a memory barrier — `__DSB()` — is required in addition.

3 Two styles of register definition

Macros

```
#define UART0_BASE 0x4000C000u
#define UART_DR    (*(volatile uint32_t *)(UART0_BASE + 0x000))
#define UART_FR    (*(volatile uint32_t *)(UART0_BASE + 0x018))
#define UART_CTL   (*(volatile uint32_t *)(UART0_BASE + 0x030))

#define UART_FR_RXFE (1u << 4)      /* receive FIFO empty */
#define UART_FR_TXFF (1u << 5)      /* transmit FIFO full */
```

Simple, and what most vendor headers use. The drawbacks are chapter 28's: no type checking, no scope, and invisible in a debugger.

Struct overlay

```
typedef struct {
    volatile uint32_t DR;        /* 0x000 data          */
    volatile uint32_t RSR;       /* 0x004 receive status*/
    uint32_t reserved0[4];       /* 0x008-0x014 gap     */
    volatile uint32_t FR;        /* 0x018 flags         */
    uint32_t reserved1;          /* 0x01C               */
    volatile uint32_t ILPR;      /* 0x020               */
    volatile uint32_t IBRD;      /* 0x024 baud integer  */
    volatile uint32_t FBRD;      /* 0x028 baud fraction */
    volatile uint32_t LCRH;      /* 0x02C line control  */
    volatile uint32_t CTL;       /* 0x030 control       */
} UART_Type;

#define UART0 ((UART_Type *)0x4000C000u)

UART0->DR = 'A';
while (UART0->FR & UART_FR_TXFF) { }
```

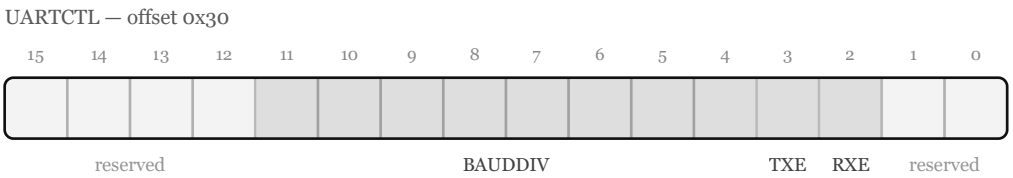
This is what CMSIS and every modern vendor header do. It is type-checked, groups related registers, and shows up in a debugger as a structure whose fields you can inspect. The `reserved` members are load-bearing: they place each register at the offset the manual

specifies, and chapter 22's padding rules are why they must be written explicitly rather than assumed.

Note also that `volatile` goes on each member, not on the pointer. `volatile UART_Type *` would also work, but marking the members documents that every one of them is hardware.

4 Reading a reference manual

A register is documented as a table of bit fields:



A register is fields, not a number. The manual draws it like this, and so should your header.

```
UARTFR — UART Flag Register        offset 0x018, reset 0x90

 Bit  Name   Type  Reset  Description
   7  TXFE   RO      1    Transmit FIFO empty
   6  RXFF   RO      0    Receive FIFO full
   5  TXFF   RO      0    Transmit FIFO full
   4  RXFE   RO      1    Receive FIFO empty
   3  BUSY   RO      0    UART busy
 2:0  —      RO      0    Reserved
```

TYPE	MEANS
RO	Read only; writes ignored
WO	Write only; reads undefined
RW	Read and write
W1C	Write one to clear — writing 0 does nothing
RC	Cleared by reading

One of those deserves care. **W1C** inverts the usual read-modify-write: to clear a flag you write a 1 to it, and the read-modify-write idiom below would clear every other pending flag as a side effect — so W1C registers are written directly with just the bit you mean.

Reserved bits must usually be preserved. Writing zeros to them can change undocumented behavior, which is the other reason read-modify-write exists.

```
/* Read-modify-write: change one field, preserve the rest. */
uint32_t v = UART0->LCRH;       /* read  */
v &= ~LCRH_WLEN_MASK;           /* clear the field */
v |= LCRH_WLEN_8;               /* set the new value */
UART0->LCRH = v;                /* write */

/* Write-one-to-clear: do NOT read-modify-write. */
UART0->ICR = ICR_RXIC;          /* ICR is at 0x044, past the members shown above */
/* clears only this flag */
```

5 Worked example: a UART driver, and `volatile` removed

Building on chapter 49's QEMU setup. Same linker script and startup code; only `main.c` changes.

```
/* uart.c — a driver written against the register map */
#include <stdint.h>
#include <stdbool.h>

/* ---------- register overlay ---------- */

typedef struct {
    volatile uint32_t DR;          /* 0x000 data                */
    volatile uint32_t RSR_ECR;     /* 0x004 receive status      */
    uint32_t reserved0[4];
    volatile uint32_t FR;          /* 0x018 flags               */
    uint32_t reserved1;
    volatile uint32_t ILPR;        /* 0x020                     */
    volatile uint32_t IBRD;        /* 0x024 baud, integer part  */
    volatile uint32_t FBRD;        /* 0x028 baud, fraction      */
    volatile uint32_t LCRH;        /* 0x02C line control        */
    volatile uint32_t CTL;         /* 0x030 control             */
```

```
    volatile uint32_t IFLS;        /* 0x034                    */
    volatile uint32_t IM;          /* 0x038 interrupt mask     */
    volatile uint32_t RIS;         /* 0x03C raw interrupt status*/
    volatile uint32_t MIS;         /* 0x040 masked status      */
    volatile uint32_t ICR;         /* 0x044 interrupt clear W1C */
} UART_Type;

#define UART0 ((UART_Type *)0x4000C000u)

/* Flag register bits, from the manual's table. */
#define FR_BUSY  (1u << 3)
#define FR_RXFE  (1u << 4)     /* receive FIFO empty  */
#define FR_TXFF  (1u << 5)     /* transmit FIFO full  */

/* Line control bits. */
#define LCRH_FEN        (1u << 4)
#define LCRH_WLEN_MASK  (3u << 5)
#define LCRH_WLEN_8     (3u << 5)

/* Control bits. */
#define CTL_UARTEN (1u << 0)
#define CTL_TXE    (1u << 8)
#define CTL_RXE    (1u << 9)

void uart_init(void)
{
    UART0->CTL = 0;                    /* disable before configuring */

    UART0->IBRD = 10;                  /* 115200 baud at 20 MHz  */
    UART0->FBRD = 54;

    /* Read-modify-write: set the word length, preserve everything else. */
    uint32_t lcrh = UART0->LCRH;
    lcrh &= ~LCRH_WLEN_MASK;
    lcrh |= LCRH_WLEN_8 | LCRH_FEN;
    UART0->LCRH = lcrh;

    UART0->CTL = CTL_UARTEN | CTL_TXE | CTL_RXE;   /* enable last */
}
```

```
void uart_putc(char c)
{
    while (UART0->FR & FR_TXFF) { }     /* poll: needs volatile */
    UART0->DR = (uint32_t)(unsigned char)c;
}

bool uart_getc(char *out)
{
    if (UART0->FR & FR_RXFE) {
        return false;                   /* nothing waiting */
    }
    *out = (char)(UART0->DR & 0xFFu);   /* reading REMOVES it from the FIFO */
    return true;
}

void uart_flush(void)
{
    while (UART0->FR & FR_BUSY) { }     /* wait for the shift register */
}

void uart_puts(const char *s)
{
    for (; *s != '\0'; s++) {
        if (*s == '\n') uart_putc('\r');
        uart_putc(*s);
    }
}
```

```
/* main.c */
#include <stdint.h>
#include <stdbool.h>

void uart_init(void);
void uart_putc(char c);
bool uart_getc(char *out);
void uart_puts(const char *s);
void uart_flush(void);
```

```
static void put_hex(uint32_t v)
{
    static const char d[] = "0123456789ABCDEF";
    uart_puts("0x");
    for (int s = 28; s >= 0; s -= 4) uart_putc(d[(v >> s) & 0xFu]);
}

/* Deliberately WITHOUT volatile, to be inspected in the disassembly. */
#define UART_FR_UNSAFE (*(uint32_t *)0x4000C018u)

static void demonstrate_without_volatile(void)
{
    uart_puts("reading FR without volatile three times:\n");
    uint32_t a = UART_FR_UNSAFE;
    uint32_t b = UART_FR_UNSAFE;
    uint32_t c = UART_FR_UNSAFE;
    uart_puts("  "); put_hex(a);
    uart_puts(" ");  put_hex(b);
    uart_puts(" ");  put_hex(c);
    uart_puts("\n  disassemble this function: at -O2 there is likely\n");
    uart_puts("  ONE load, not three\n");
}

int main(void)
{
    uart_init();

    uart_puts("\n=== memory-mapped I/O ===\n");

    uart_puts("UART base    : "); put_hex(0x4000C000u); uart_puts("\n");
    uart_puts("FR offset    : 0x018\n");
    uart_puts("FR value     : "); put_hex(*(volatile uint32_t *)0x4000C018u);
    uart_puts("\n  bit 4 RXFE set means the receive FIFO is empty\n");
    uart_puts("  bit 7 TXFE set means the transmit FIFO is empty\n\n");

    demonstrate_without_volatile();

    uart_puts("\ntype characters; they are echoed. 'q' quits to the loop.\n");
```

```
    for (;;) {
        char c;
        if (uart_getc(&c)) {
            if (c == 'q') {
                uart_puts("\nbye\n");
                break;
            }
            uart_puts("got '");
            uart_putc(c);
            uart_puts("' = ");
            put_hex((uint32_t)(unsigned char)c);
            uart_puts("\n");
        }
    }

    uart_flush();
    for (;;) { }
}
```

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb \
    -ffreestanding -nostdlib -O2 -Wall -Wextra -g \
    -T firmware.ld -o firmware.elf startup.c uart.c main.c -lgcc

qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf
```

Type characters and watch them echoed with their hex codes. Quit with `q`, then Ctrl-A X.

Prove that `volatile` changes the generated code

This is the experiment that makes the rule concrete rather than memorized.

```
arm-none-eabi-objdump -d firmware.elf \
    | sed -n '/<demonstrate_without_volatile>:/,/^$/p'
```

Count the load instructions. Three source-level reads of the same address typically produce **one** `ldr`, with the result reused — the compiler correctly concluded that a non-`volatile` object cannot change between reads. On real hardware that means a receive loop takes one byte and then believes the FIFO is permanently in whatever state it saw first.

Now add `volatile` to the macro, rebuild, and disassemble again: three loads, in order.

Break the polling loop

Edit `uart_putc` to read a non-`volatile` copy:

```
void uart_putc(char c)
{
    uint32_t *fr = (uint32_t *)0x4000C018u;     /* no volatile */
    while (*fr & FR_TXFF) { }
    UART0->DR = (uint32_t)(unsigned char)c;
}
```

```
arm-none-eabi-gcc … -O0 … && qemu-system-arm …   # usually still works
arm-none-eabi-gcc … -O2 … && qemu-system-arm …   # hangs on real hardware; QEMU may not show
it
```

At `-O0` every access goes to memory anyway, so the bug is invisible. At `-O2` the load is hoisted out of the loop and the driver waits forever on a stale value. This is the single most common embedded bug, and the reason it is so confusing is that the debug build is fine.

Inspect the peripheral from the debugger

```
# terminal 1
qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf -S -s

# terminal 2
gdb-multiarch firmware.elf
(gdb) target remote :1234
(gdb) break uart_putc
(gdb) continue
(gdb) print/x *(UART_Type *)0x4000C000      # the whole peripheral
(gdb) print/x ((UART_Type *)0x4000C000)->FR
(gdb) print/t ((UART_Type *)0x4000C000)->FR                    # binary: read the bits
(gdb) set var ((UART_Type *)0x4000C000)->DR = 0x41             # write 'A' by hand
```

The struct overlay pays for itself here: GDB prints the peripheral as a named structure rather than a block of hex. Setting `DR` from the debugger transmits a character — the hardware

responds to a write from any source, which is the clearest possible demonstration that a register is not a variable.

Check the offsets are right

An overlay with a wrong `reserved` array silently addresses the wrong registers. Verify it rather than trusting it:

```
/* add temporarily, in a file that sees UART_Type and includes <stddef.h> */
uart_puts("offsetof FR  : "); put_hex((uint32_t)offsetof(UART_Type, FR));
uart_puts("\noffsetof CTL : "); put_hex((uint32_t)offsetof(UART_Type, CTL));
uart_puts("\nsizeof        : "); put_hex((uint32_t)sizeof(UART_Type));
```

FR must be `0x018` and CTL `0x030`, exactly as the manual states. Chapter 22's `offsetof`, used as a correctness check rather than a curiosity — and a `static_assert` on each offset, as chapter 27 suggested, turns this into something the compiler verifies on every build.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Omitting `volatile`	Reads hoisted; polling hangs at `-O2`	Every register access is `volatile`.
Read-modify-write on a W1C register	Clears every other pending flag	Write only the bit you mean.
Writing zeros to reserved bits	Undocumented behavior changes	Preserve them: read-modify-write.
Wrong `reserved` padding in an overlay	Every register after the gap is wrong	Verify with `offsetof` and `static_assert`.
Reading a data register twice	Loses a byte — the read is destructive	Read once into a variable.
Configuring after enabling	The peripheral runs with old settings	Disable, configure, enable.
Testing only at `-O0`	The bug appears in the release build	Test at the optimization level you ship.
Assuming `volatile` orders hardware	Reordering by the processor	Add a barrier where the manual requires one.

7 Check yourself

Why does a polling loop without `volatile` hang at `-O2` but work at `-O0`?

Because at `-O0` every access goes to memory anyway, while at `-O2` the compiler sees that nothing in the empty loop body modifies the object and hoists the load out, testing a cached copy forever. `volatile` tells it the object can change by means it cannot see, so the load must stay inside the loop.

Why must a write-one-to-clear register not be read-modify-written?

Because the read returns every pending flag, and writing that value back writes a 1 to each of them — clearing flags you never intended to acknowledge. W1C registers are written directly with only the bit being cleared; the usual preserve-the-others reasoning is exactly backwards for them.

What are the `reserved` members in a register overlay for?

To place each subsequent register at the byte offset the reference manual specifies. The compiler lays members out consecutively with padding rules of its own, so gaps in the hardware map must be declared explicitly. Getting one wrong silently shifts every register after it, and `offsetof` with `static_assert` is how you prove it is right.

Why can reading a register twice be a bug?

Because reads can have side effects. On many UARTs, reading the data register removes a byte from the receive FIFO, so a second read consumes a second byte or returns garbage. Registers are not variables: read once into a local and use that.

Is `volatile` enough to guarantee hardware ordering?

No. It constrains the compiler, not the processor. On a core with a write buffer or out-of-order execution, accesses can still be reordered or delayed relative to what the device requires. Where the manual demands a completed write before the next access, a memory barrier such as `__DSB()` is needed in addition.

8 Where this leads

Polling a flag in a loop wastes the whole processor waiting. Chapter 51 replaces it with interrupts — the hardware tells you when it is ready — which brings back chapter 39's async-signal-safety problem in a harsher form, because an ISR can interrupt any instruction and `volatile sig_atomic_t` is no longer sufficient.

PROFESSIONAL · CHAPTER 51

Interrupts and Real-Time Behavior

Polling burns the whole processor waiting. An interrupt inverts it: the hardware calls you. That inversion brings back chapter 41's concurrency problems in their sharpest form — an ISR can interrupt any instruction, and there is no mutex to take.

BY THE END OF THIS CHAPTER YOU CAN

- Install an interrupt handler and explain what the processor does on entry.
- Reason about priorities, nesting, and latency.
- Share data with an ISR correctly, and say why `volatile` alone is not enough.
- Choose between polling and interrupts with a reason.
- Measure and bound worst-case response time.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 What happens on an interrupt

When a peripheral raises an interrupt, the processor finishes the current instruction and then, in hardware:

1. Pushes enough state to resume — on Cortex-M, eight registers.
2. Looks up the handler address in the vector table.
3. Branches to it.

Your handler runs, returns, and the hardware restores the state. The interrupted code continues without knowing anything happened.

On Cortex-M the vector table from chapter 49 grows past the first four entries:

```
__attribute__((section(".vectors"), used))
void (* const vector_table[])(void) = {
    (void (*)(void))&_stack_top,   /*  0 initial SP        */
    reset_handler,                 /*  1 reset             */
    default_handler,               /*  2 NMI               */
    hard_fault_handler,            /*  3 hard fault        */
    default_handler,               /*  4 memory management */
    default_handler,               /*  5 bus fault         */
```

```
    default_handler,                /*  6 usage fault       */
    0, 0, 0, 0,                     /*  7-10 reserved       */
    default_handler,                /* 11 SVCall            */
    default_handler,                /* 12 debug monitor     */
    0,                              /* 13 reserved          */
    default_handler,                /* 14 PendSV            */
    systick_handler,                /* 15 SysTick           */
    /* 16 onwards: device peripherals, IRQ n at 16 + n */
    gpio_a_handler,                 /* 16 IRQ 0  GPIO A     */
    default_handler,                /* 17 IRQ 1  GPIO B     */
    default_handler,                /* 18 IRQ 2  GPIO C     */
    default_handler,                /* 19 IRQ 3  GPIO D     */
    default_handler,                /* 20 IRQ 4  GPIO E     */
    uart0_handler,                  /* 21 IRQ 5  UART0      */
};
```

Because the processor branches through this table, an entry left at zero means an unhandled interrupt jumps to address 0 — which on Cortex-M is the stack pointer value, and produces an immediate hard fault. Fill every slot with a default handler rather than leaving gaps; only the architecturally reserved entries stay zero, because nothing can ever be delivered through them. The table must also be long enough: an interrupt whose entry lies past its end is dispatched through whatever code the linker placed there.

On Cortex-M an ISR is an ordinary C function taking no arguments and returning `void`; the hardware handles saving and restoring. On other architectures a compiler attribute such as `__attribute__((interrupt))` is required to generate the correct prologue and return instruction.

2 Priorities, nesting, and latency

A higher-priority interrupt preempts a lower one. On Cortex-M, **lower numbers mean higher priority** — priority 0 preempts priority 1 — which is the opposite of most people's expectation.

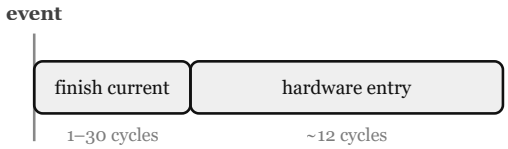
```
#define NVIC_ISER0  (*(volatile uint32_t *)0xE000E100u)
#define NVIC_IPR(n) (*(volatile uint8_t  *)(0xE000E400u + (n)))

NVIC_IPR(5) = 0x40;            /* set a priority for IRQ 5 */
NVIC_ISER0  = (1u << 5);       /* enable it */
```

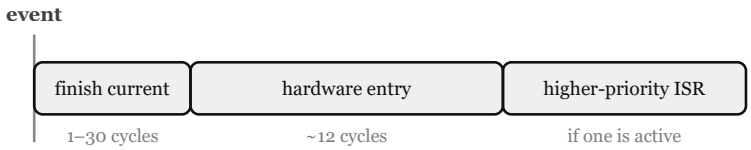
Latency is the time from the hardware event to the first instruction of your handler. It is the sum of three things, or four if a critical section is in the way:



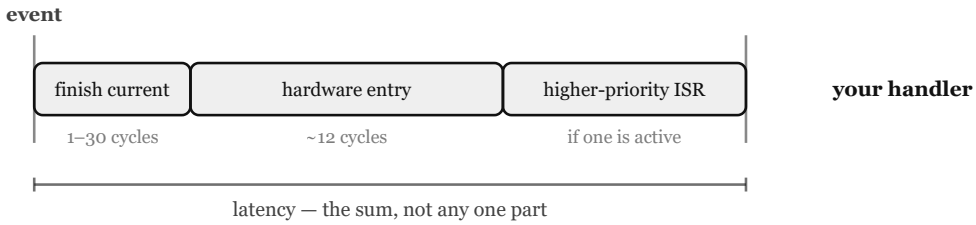
The event arrives. The processor cannot abandon the instruction it is in the middle of, so the first cost is finishing it — short for most instructions, long for a division or a multi-register move.



Then the hardware entry: push the registers the ABI says must be saved, fetch the vector, jump. Fixed, and out of your hands.



And if a higher-priority interrupt is active, yours waits for it. On Cortex-M lower numbers mean *higher* priority, which is the opposite of most people's first guess.



Only now does your handler's first instruction run. Latency is the whole bracket. This is why a long handler is not just slow in itself — it is the third bar in somebody else's picture.

CONTRIBUTION	TYPICAL	CONTROLLED BY
Hardware entry (stacking, vector fetch)	~12 cycles on Cortex-M	Fixed
Finishing the current instruction	1–30 cycles	Avoid long instructions in critical code
Waiting for a higher or equal priority ISR	**unbounded**	You — keep handlers short
Waiting for a critical section	**unbounded**	You — keep them shorter still

The last two are the whole of real-time engineering. Worst-case latency is not the average; it is what happens when every other handler and every critical section conspire. A system that meets its deadline 99.99% of the time has missed it, if the deadline was a requirement.

> **The rule for ISRs: do the minimum and return.** Read the byte, set a flag, enqueue the work. Anything longer delays every other interrupt in the system. The pattern is exactly chapter 39's signal handler, for the same reason and with harder consequences.

3 Sharing data with an ISR

An ISR and the main loop share memory with no scheduler, no mutex, and no way for the main loop to know when it will be interrupted. Three mechanisms, in increasing strength.

`volatile`: necessary, not sufficient

```
static volatile bool button_pressed = false;

void gpio_a_handler(void) { button_pressed = true; }

/* main */
while (!button_pressed) { }          /* volatile keeps the load in the loop */
```

Without `volatile` the main loop caches the value and never sees the change — chapter 50's hoisting, in a different disguise. With it, a single-word flag works.

Atomicity: `volatile` does not provide it

```
static volatile uint32_t tick_count;         /* 32-bit: atomic on Cortex-M */
static volatile uint64_t micros;             /* 64-bit: NOT atomic         */

/* in main, on a 32-bit machine */
uint64_t now = micros;     /* two loads; an ISR can run between them */
```

A 64-bit read on a 32-bit processor is two instructions. If the ISR updates the value between them, you get the low half of one value and the high half of another — a number that was never correct. Chapter 41's data race, with no thread library in sight.

Two fixes. Use `<stdatomic.h>` where the toolchain supports it, or disable interrupts for the two instructions.

Critical sections

```
static inline uint32_t enter_critical(void)
{
    uint32_t primask;
    __asm__ volatile ("mrs %0, primask" : "=r"(primask));
    __asm__ volatile ("cpsid i" ::: "memory");     /* disable interrupts */
    return primask;                                 /* save the old state */
}

static inline void exit_critical(uint32_t primask)
{
    if ((primask & 1u) == 0) {
```

```
        __asm__ volatile ("cpsie i" ::: "memory"); /* restore only if it
                                                      was enabled before */
    }
}

uint32_t state = enter_critical();
uint64_t now = micros;                /* now indivisible */
exit_critical(state);
```

Saving and restoring the previous state matters: an unconditional re-enable inside a function called from an ISR would enable interrupts in the middle of a handler. The `"memory"` clobber prevents the compiler moving accesses across the boundary.

A critical section blocks *every* interrupt, so it directly increases worst-case latency for the whole system. Keep them to a handful of instructions.

The lock-free alternative

A single-producer, single-consumer ring buffer needs no critical section at all, because each index is written by exactly one side:

```
static volatile uint8_t  rx_buf[64];
static volatile uint8_t  rx_head;      /* written only by the ISR   */
static volatile uint8_t  rx_tail;      /* written only by main      */

/* ISR */
uint8_t next = (uint8_t)((rx_head + 1u) % 64u);
if (next != rx_tail) {                 /* drop on overflow */
    rx_buf[rx_head] = byte;
    rx_head = next;                    /* publish after the data */
}
```

The ordering of that last write matters: the data must be stored before the index that makes it visible. With `volatile` the compiler preserves the order; on a weakly ordered processor a barrier would also be needed. This structure is the standard way to get data out of an ISR, and chapter 54's RTOS queue is the general version of it.

4 Polling or interrupts?

POLLING	INTERRUPTS
Simple; no concurrency	Concurrent with everything
Deterministic timing	Latency varies with system load
Wastes the processor while waiting	The core can sleep
Fine for a fast, predictable device	Necessary for rare or urgent events
Response bounded by the loop period	Response in microseconds

Polling is not primitive — a tight loop reading a sensor at a known rate is often the right design, and it is far easier to reason about. Interrupts earn their complexity when events are rare, urgent, or when the processor should be asleep between them.

Debouncing

A mechanical switch does not close once; it bounces for several milliseconds, generating dozens of edges. An interrupt per edge produces dozens of events for one press.

```
/* In the ISR: ignore anything within the bounce window. */
void gpio_a_handler(void)
{
    uint32_t now = tick_count;
    if (now - last_press >= DEBOUNCE_TICKS) {
        last_press = now;
        button_events++;
    }
    GPIO_ICR = PIN_MASK;           /* W1C: acknowledge — chapter 50 */
}
```

Forgetting that last line is the classic interrupt bug: the peripheral keeps the request asserted, so the handler re-enters immediately and the system appears to hang.

5 Worked example: polling against interrupts, measured

Continuing chapter 49's QEMU target. SysTick provides a periodic interrupt; the UART provides an asynchronous one.

```
/* interrupts.c */
#include <stdint.h>
#include <stdbool.h>

/* ---------- registers (chapter 50) ---------- */

typedef struct {
    volatile uint32_t DR, RSR_ECR;
    uint32_t r0[4];
    volatile uint32_t FR;
    uint32_t r1;
    volatile uint32_t ILPR, IBRD, FBRD, LCRH, CTL, IFLS, IM, RIS, MIS, ICR;
} UART_Type;

#define UART0 ((UART_Type *)0x4000C000u)
#define FR_RXFE (1u << 4)
#define FR_TXFF (1u << 5)
#define IM_RXIM (1u << 4)
#define ICR_RXIC (1u << 4)

/* SysTick, in the core peripheral region */
#define SYST_CSR  (*(volatile uint32_t *)0xE000E010u)
#define SYST_RVR  (*(volatile uint32_t *)0xE000E014u)
#define SYST_CVR  (*(volatile uint32_t *)0xE000E018u)

/* NVIC */
#define NVIC_ISER0 (*(volatile uint32_t *)0xE000E100u)
#define UART0_IRQ  5

/* ---------- shared state ---------- */

/* 32-bit and volatile: a single load or store on this machine. */
static volatile uint32_t tick_count = 0;

/* 64-bit: NOT atomic on a 32-bit core. Needs a critical section. */
static volatile uint64_t micros = 0;

/* Single-producer single-consumer ring: no critical section needed. */
```

```
#define RX_SIZE 64
static volatile uint8_t rx_buf[RX_SIZE];
static volatile uint8_t rx_head = 0;      /* ISR writes  */
static volatile uint8_t rx_tail = 0;      /* main writes */

static volatile uint32_t isr_entries = 0;
static volatile uint32_t rx_dropped  = 0;

/* ---------- critical sections ---------- */

static inline uint32_t enter_critical(void)
{
    uint32_t primask;
    __asm__ volatile ("mrs %0, primask" : "=r"(primask));
    __asm__ volatile ("cpsid i" ::: "memory");
    return primask;
}

static inline void exit_critical(uint32_t primask)
{
    if ((primask & 1u) == 0) {
        __asm__ volatile ("cpsie i" ::: "memory");
    }
}

/* ---------- handlers: short, and they acknowledge ---------- */

void systick_handler(void)
{
    tick_count++;
    micros += 1000;          /* read-modify-write of a 64-bit value,
                                but only this ISR touches it */
}

void uart0_handler(void)
{
    isr_entries++;

    while ((UART0->FR & FR_RXFE) == 0) {          /* drain the FIFO */
```

```
        uint8_t byte = (uint8_t)(UART0->DR & 0xFFu);

        uint8_t next = (uint8_t)((rx_head + 1u) % RX_SIZE);
        if (next == rx_tail) {
            rx_dropped++;                       /* buffer full */
        } else {
            rx_buf[rx_head] = byte;
            rx_head = next;                     /* publish after the data */
        }
    }

    UART0->ICR = ICR_RXIC;    /* acknowledge, or we re-enter forever */
}

/* ---------- consumer side ---------- */

static bool rx_get(uint8_t *out)
{
    if (rx_tail == rx_head) return false;          /* empty */
    *out = rx_buf[rx_tail];
    rx_tail = (uint8_t)((rx_tail + 1u) % RX_SIZE);
    return true;
}

/* ---------- output ---------- */

static void putc_raw(char c)
{
    while (UART0->FR & FR_TXFF) { }
    UART0->DR = (uint32_t)(unsigned char)c;
}

static void puts_raw(const char *s)
{
    for (; *s; s++) { if (*s == '\n') putc_raw('\r'); putc_raw(*s); }
}

static void put_uint(uint32_t v)
{
```

```
    char t[11]; int i = 0;
    if (v == 0) { putc_raw('0'); return; }
    while (v) { t[i++] = (char)('0' + v % 10); v /= 10; }
    while (i) putc_raw(t[--i]);
}

static void put_u64(uint64_t v)
{
    char t[21]; int i = 0;
    if (v == 0) { putc_raw('0'); return; }
    while (v) { t[i++] = (char)('0' + (uint32_t)(v % 10)); v /= 10; }
    while (i) putc_raw(t[--i]);
}

/* ---------- setup ---------- */

static void systick_init(uint32_t reload)
{
    SYST_RVR = reload - 1u;
    SYST_CVR = 0;
    SYST_CSR = 0x7u;            /* enable, interrupt, processor clock */
}

static void uart_irq_init(void)
{
    UART0->IM  = IM_RXIM;       /* interrupt when a byte arrives */
    UART0->ICR = 0x7FFu;        /* clear anything pending */
    NVIC_ISER0 = (1u << UART0_IRQ);
}

int main(void)
{
    UART0->CTL = 0;
    UART0->IBRD = 10; UART0->FBRD = 54;
    UART0->LCRH = (3u << 5) | (1u << 4);
    UART0->CTL = 1u | (1u << 8) | (1u << 9);

    puts_raw("\n=== interrupts ===\n");
```

```
    /* --- 1. polling: the processor is fully occupied --- */
    puts_raw("\npolling for a character (type one):\n");
    uint32_t spins = 0;
    while (UART0->FR & FR_RXFE) {
        spins++;                        /* every cycle wasted */
    }
    uint8_t first = (uint8_t)(UART0->DR & 0xFFu);
    puts_raw("  got '"); putc_raw((char)first);
    puts_raw("' after "); put_uint(spins); puts_raw(" wasted iterations\n");

    /* --- 2. interrupts: the loop does other work --- */
    systick_init(12500);                /* 1 kHz at QEMU's 12.5 MHz */
    uart_irq_init();
    puts_raw("\ninterrupts enabled; type freely, 'q' ends\n");

    uint32_t loop_iterations = 0;
    uint32_t received = 0;
    bool     done = false;

    while (!done) {
        loop_iterations++;              /* real work would go here */

        uint8_t byte;
        while (rx_get(&byte)) {
            received++;
            if (byte == 'q') { done = true; break; }
            puts_raw("  rx '"); putc_raw((char)byte);
            puts_raw("' at tick "); put_uint(tick_count); puts_raw("\n");
        }

        if (tick_count % 5000 == 0 && tick_count > 0) {
            puts_raw("  ... 5000 ticks, loop still running\n");
            while (tick_count % 5000 == 0) { }   /* print once */
        }
    }

    /* --- 3. the 64-bit read needs protection --- */
    puts_raw("\nreading a 64-bit counter:\n");
```

```
    uint64_t unsafe = micros;                   /* two loads, interruptible */
    puts_raw("  without a critical section: "); put_u64(unsafe);
    puts_raw("\n");

    uint32_t state = enter_critical();
    uint64_t safe = micros;                     /* indivisible */
    exit_critical(state);
    puts_raw("  with one                  : "); put_u64(safe);
    puts_raw("\n  on a 32-bit core the first is two loads; an ISR\n");
    puts_raw("  between them yields a value that never existed\n");

    /* --- 4. what it cost --- */
    puts_raw("\nstatistics\n");
    puts_raw("  ticks           : "); put_uint(tick_count);  puts_raw("\n");
    puts_raw("  uart interrupts : "); put_uint(isr_entries); puts_raw("\n");
    puts_raw("  bytes received  : "); put_uint(received);    puts_raw("\n");
    puts_raw("  bytes dropped   : "); put_uint(rx_dropped);  puts_raw("\n");
    puts_raw("  loop iterations : "); put_uint(loop_iterations);
    puts_raw("\n  the loop kept running the whole time — that is the point\n");

    for (;;) { }
}
```

The handlers only run if the vector table points at them. This chapter's `startup.c` is chapter 49's with the table grown to 22 entries, as in the first section: `systick_handler` at index 15 and `uart0_handler` at 16 + 5. Left at chapter 49's four entries, the first SysTick would be dispatched through whatever code follows the table, and the program would stop dead after enabling interrupts.

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb \
    -ffreestanding -nostdlib -O2 -Wall -Wextra -g \
    -T firmware.ld -o firmware.elf startup.c interrupts.c -lgcc

qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf
```

What the two phases show

The polling phase reports millions of wasted iterations for one character. Every one of those cycles was available for other work and was spent asking "is it here yet".

The interrupt phase runs the main loop continuously while characters arrive. The loop counter keeps climbing, the tick counter advances, and bytes appear — three things happening without any of them waiting for the others.

Forget to acknowledge

Comment out `UART0->ICR = ICR_RXIC;` and rerun. The moment a byte arrives, the system appears to freeze: the peripheral still asserts the request, so the handler returns and is immediately re-entered, forever. The main loop never runs again.

This is the most common interrupt bug and the symptom — a hang with no crash — gives no clue. Set a watchpoint on `isr_entries` in GDB and you will see it climbing at millions per second.

Remove `volatile` from `tick_count`

```
static uint32_t tick_count = 0;         /* no volatile */
…
while (tick_count < 100) { }            /* in main */
```

At `-O2` the loop never exits: the compiler hoisted the load, exactly as in chapter 50, and no interrupt can change a value held in a register. At `-O0` it works. This is the same lesson as the UART flag, now with the ISR as the invisible writer.

Measure the response time

```
/* in the ISR */
uint32_t entry_cycle = SYST_CVR;     /* SysTick counts down */

/* compare against the value captured when the event was triggered */
```

SysTick's current-value register gives a cycle-resolution timer. Capture it at the start of the ISR and compare with a value captured just before the triggering event; the difference is the latency, in processor cycles. Under QEMU the absolute numbers are not those of real silicon, but the *relative* effect is faithful: add a long critical section to the main loop and watch the worst case grow.

```
uint32_t s = enter_critical();
for (volatile int i = 0; i < 10000; i++) { }    /* a terrible idea */
exit_critical(s);
```

Every interrupt in the system is delayed by the whole of that loop. That is why the rule is not "keep critical sections short" as a style preference — it is the mechanism by which one function's carelessness becomes another subsystem's missed deadline.

Overflow the ring buffer

```
printf 'aaaa
```

7 Check yourself

Why must an ISR acknowledge the interrupt source?

Because the peripheral keeps its request asserted until told the event has been handled. Returning without clearing it means the processor immediately takes the interrupt again, so the handler re-enters in an endless loop and the main program never runs. The symptom is a hang with no crash and no diagnostic.

Why is `volatile` necessary but not sufficient for ISR-shared data?

Necessary because without it the main loop caches the value in a register and never observes the ISR's write. Insufficient because it says nothing about atomicity: a 64-bit variable on a 32-bit core is read in two instructions, and an interrupt between them yields half of one value and half of another. That needs a critical section or an atomic type.

Why does a long critical section hurt the whole system?

Because disabling interrupts blocks every one of them, not just the one you are protecting against. Worst-case latency for every subsystem grows by the length of the longest critical section anywhere in the program — so one careless function can cause another's deadline to be missed.

Why does a single-producer single-consumer ring buffer need no lock?

Because each index is written by exactly one side: the ISR advances the head, the main loop advances the tail, and each only reads the other's. No value is ever modified by both, so there is no read-modify-write to be interrupted. The remaining requirement is ordering — the data must be stored before the index that publishes it.

When is polling the better choice?

When the event is frequent and predictable, when the timing must be deterministic, or when the code simplicity is worth more than the wasted cycles. A tight loop reading a sensor at a known rate has no concurrency, no shared state, and no latency variance. Interrupts earn their complexity for rare or urgent events, and when the processor should sleep between them.

8 Where this leads

Chapter 52 opens the linker script that has been quietly making all of this work — where the vector table lands, why `.data` is stored in one place and addressed in another, and how to prove from the map file that the image is laid out as intended.

PROFESSIONAL · CHAPTER 52

Linker Scripts and Firmware Memory Layout

On a hosted system the linker's default script is invisible and correct. On bare metal you write it, and it decides where every byte lands. This chapter opens the file that chapters 49 through 51 have been quietly relying on.

BY THE END OF THIS CHAPTER YOU CAN

- Write a linker script defining memory regions and section placement.
- Explain load address versus virtual address, and why `.data` has both.
- Define and use symbols that C code can read.
- Place the stack, and detect an overflow before it corrupts anything.
- Read a map file and account for every byte of an image.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Sections

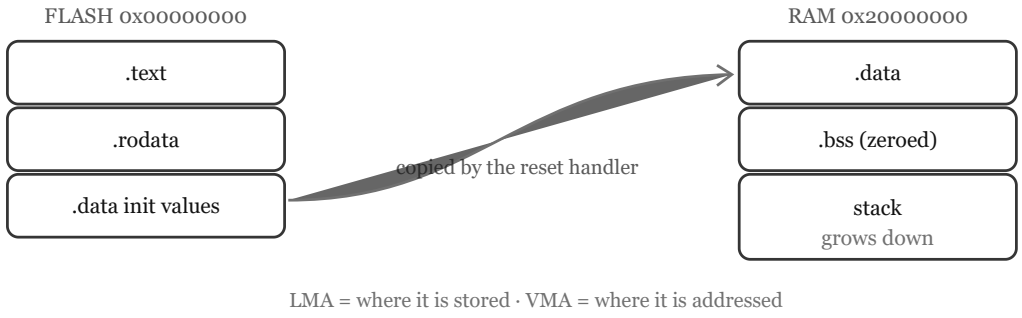
The compiler does not emit a flat stream of bytes. It sorts everything into named *sections*, and the linker gathers matching sections from every object file and places each group at an address.

SECTION	CONTAINS	LIVES IN	IN THE IMAGE?
`.text`	Machine code	Flash	Yes
`.rodata`	`const` data, string literals	Flash	Yes
`.data`	Initialized globals	RAM	**Yes — the initial values**
`.bss`	Zero-initialized globals	RAM	No — only its size
Stack	Locals, call frames	RAM	No
Heap	`malloc`, if used at all	RAM	No

Chapter 17 met these on a hosted system, where the loader handles them. Here the same four sections exist and nothing handles them for you.

2 Load address and virtual address

This is the idea the whole script turns on. Most sections live where they are stored. `.data` does not.



Chapter 49's startup loop, seen from the linker's side.

- **VMA** — virtual memory address — is where the code expects the section to be. For `.data`, RAM.
- **LMA** — load memory address — is where it is actually stored in the image. For `.data`, flash.

For `.text` and `.rodata` the two are the same: code executes directly from flash. For `.data` they differ, and the gap is exactly what the startup copy closes.

3 The script

```
ENTRY(reset_handler)                /* the entry symbol, for debuggers */

MEMORY
{
    FLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
    RAM   (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}

_stack_size = 4K;
_heap_size  = 0;                    /* no heap: chapter 53 */

SECTIONS
{
    .text : {
```

```
        KEEP(*(.vectors))           /* must be first, and never discarded */
        *(.text*)
        *(.rodata*)
        . = ALIGN(4);
        _etext = .;
    } > FLASH

    .data : {
        . = ALIGN(4);
        _data_start = .;
        *(.data*)
        . = ALIGN(4);
        _data_end = .;
    } > RAM AT> FLASH               /* VMA in RAM, LMA in FLASH */

    _data_load = LOADADDR(.data);    /* where the startup code reads from */

    .bss (NOLOAD) : {
        . = ALIGN(4);
        _bss_start = .;
        *(.bss*)
        *(COMMON)
        . = ALIGN(4);
        _bss_end = .;
    } > RAM

    .stack (NOLOAD) : {
        . = ALIGN(8);               /* ARM requires 8-byte stack alignment */
        _stack_bottom = .;
        . = . + _stack_size;
        _stack_top = .;
    } > RAM

    /* Fail the build rather than the device. */
    ASSERT(_stack_top <= ORIGIN(RAM) + LENGTH(RAM), "RAM overflow")
}
```

CONSTRUCT	MEANS
`MEMORY`	Names the physical regions and their sizes
`> FLASH`	Place this section in that region
`AT> FLASH`	…but store it in this one — sets the LMA
`.`	The location counter, assignable
`KEEP(...)`	Do not discard even if nothing references it
`NOLOAD`	Reserve the space; store nothing in the image
`_symbol = .;`	Define a symbol C can refer to
`ASSERT(...)`	Fail the link if the condition is false

`KEEP` on the vector table is not optional. With `--gc-sections` the linker discards anything unreferenced, and nothing in C refers to the vector table — the processor reads it directly. Without `KEEP` it disappears and the device does not boot.

The `ASSERT` is the difference between a link error and a device that runs until the stack meets `.bss`. Every embedded project should have one.

4 Linker symbols in C

A symbol defined in the script has no storage — the linker records only an *address*. That makes the C declaration counter-intuitive:

```
extern uint32_t _data_start;      /* declares a variable AT that address */


uint32_t *p = &_data_start;       /* CORRECT: take its address */
uint32_t  v = _data_start;        /* WRONG: reads whatever is stored there */
```

The value of a linker symbol *is* its address, so you always want `&`. Some projects avoid the trap by declaring them as arrays, where the name already decays to an address:

```
extern uint32_t _data_start[];    /* now _data_start is the address */
uint32_t *p = _data_start;        /* no & needed, and no way to misuse it */
```

5 Stack placement and overflow

The stack grows downward from its top. Place it at the end of RAM and it grows toward `.bss`; there is no guard page and no fault — chapter 49's point about the absent MMU.

Three defences, in increasing cost:

A painted region. Fill the stack with a pattern at startup, then check later how far down it has been disturbed. This measures the high-water mark, which tells you how much margin you actually have.

```
#define STACK_PAINT 0xC0DEC0DEu

void paint_stack(void)              /* called early, before deep calls */
{
    uint32_t *p = _stack_bottom;
    uint32_t marker;
    while (p < &marker) *p++ = STACK_PAINT;   /* up to the current frame */
}

size_t stack_used(void)
{
    uint32_t *p = _stack_bottom;
    while (p < _stack_top && *p == STACK_PAINT) p++;
    return (size_t)((uint8_t *)_stack_top - (uint8_t *)p);
}
```

A canary. Reserve a word at the bottom of the stack, set it to a known value, and check it periodically. Cheap, and it detects an overflow after the fact rather than before.

Hardware. An MPU region marked no-access below the stack turns an overflow into a fault at the moment it happens. Available on many Cortex-M parts and worth configuring on anything safety-relevant.

6 Worked example: taking an image apart

```
/* layout.c — every section populated deliberately */
#include <stdint.h>

/* Linker symbols, declared as arrays so no & is needed. */
extern uint32_t _etext[], _data_start[], _data_end[], _data_load[];
```

```
extern uint32_t _bss_start[], _bss_end[];
extern uint32_t _stack_bottom[], _stack_top[];

/* .rodata — stays in flash, costs no RAM */
static const char banner[]      = "firmware layout demo";
static const uint32_t table[16] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };

/* volatile on all four: never written, they would otherwise be folded into
   constants and vanish from the image, and the checks at the end of main
   would test the compiler instead of the startup code. */

/* .data — initial values in flash, variable in RAM */
static volatile uint32_t marker   = 0xA5A5A5A5u;
static volatile uint16_t revision = 42;

/* .bss — no flash cost at all */
static volatile uint32_t counter;
static volatile uint8_t  scratch[1024];

/* ---------- UART, from chapter 50 ---------- */

#define UART0_DR (*(volatile uint32_t *)0x4000C000u)
#define UART0_FR (*(volatile uint32_t *)0x4000C018u)

static void putc_raw(char c)
{
    while (UART0_FR & (1u << 5)) { }
    UART0_DR = (uint32_t)(unsigned char)c;
}
static void puts_raw(const char *s)
{
    for (; *s; s++) { if (*s == '\n') putc_raw('\r'); putc_raw(*s); }
}
static void put_hex(uint32_t v)
{
    static const char d[] = "0123456789ABCDEF";
    puts_raw("0x");
    for (int s = 28; s >= 0; s -= 4) putc_raw(d[(v >> s) & 0xFu]);
}
```

```
static void put_uint(uint32_t v)
{
    char t[11]; int i = 0;
    if (!v) { putc_raw('0'); return; }
    while (v) { t[i++] = (char)('0' + v % 10); v /= 10; }
    while (i) putc_raw(t[--i]);
}

/* ---------- stack painting ---------- */

#define PAINT 0xC0DEC0DEu

void paint_stack(void)
{
    uint32_t here;
    for (uint32_t *p = _stack_bottom; p < &here; p++) {
        *p = PAINT;
    }
}

static uint32_t stack_used_bytes(void)
{
    uint32_t *p = _stack_bottom;
    while (p < _stack_top && *p == PAINT) p++;
    return (uint32_t)((uint8_t *)_stack_top - (uint8_t *)p);
}

/* Recursion to consume stack on demand. */
static uint32_t consume(uint32_t depth)
{
    volatile uint8_t frame[64];
    frame[0] = (uint8_t)depth;
    if (depth == 0) return frame[0];
    return consume(depth - 1) + frame[0];
}

int main(void)
{
    puts_raw("\n=== firmware layout ===\n\n");
```

```
puts_raw("FLASH\n");
puts_raw("  .text + .rodata end : "); put_hex((uint32_t)(uintptr_t)_etext);
puts_raw("\n  banner (rodata)     : ");
put_hex((uint32_t)(uintptr_t)banner);
puts_raw("\n  table  (rodata)     : ");
put_hex((uint32_t)(uintptr_t)table);
puts_raw("\n  .data initial values: ");
put_hex((uint32_t)(uintptr_t)_data_load);
puts_raw("   <-- LMA, in flash\n");

puts_raw("\nRAM\n");
puts_raw("  .data  "); put_hex((uint32_t)(uintptr_t)_data_start);
puts_raw(" .. ");      put_hex((uint32_t)(uintptr_t)_data_end);
puts_raw("   <-- VMA, in RAM\n");
puts_raw("  .bss   "); put_hex((uint32_t)(uintptr_t)_bss_start);
puts_raw(" .. ");      put_hex((uint32_t)(uintptr_t)_bss_end);
puts_raw("\n  scratch "); put_hex((uint32_t)(uintptr_t)scratch);
puts_raw("   <-- 1 KB of it, no FLASH");
puts_raw("\n  stack  "); put_hex((uint32_t)(uintptr_t)_stack_bottom);
puts_raw(" .. ");        put_hex((uint32_t)(uintptr_t)_stack_top);
puts_raw("\n");

puts_raw("\nsizes\n");
puts_raw("  .data : ");
put_uint((uint32_t)((uint8_t *)_data_end - (uint8_t *)_data_start));
puts_raw(" bytes  (also occupies that much FLASH)\n");
puts_raw("  .bss  : ");
put_uint((uint32_t)((uint8_t *)_bss_end - (uint8_t *)_bss_start));
puts_raw(" bytes  (zero FLASH)\n");
puts_raw("  stack : ");
put_uint((uint32_t)((uint8_t *)_stack_top - (uint8_t *)_stack_bottom));
puts_raw(" bytes reserved\n");

puts_raw("\nstartup did its job\n");
puts_raw("  marker   = "); put_hex(marker);
puts_raw(marker == 0xA5A5A5A5u ? "  .data copied\n" : "  NOT COPIED\n");
puts_raw("  revision = "); put_uint(revision); puts_raw("\n");
puts_raw("  counter  = "); put_uint(counter);
```

```
    puts_raw(counter == 0 ? "  .bss zeroed\n" : "  NOT ZEROED\n");

    puts_raw("\nstack high-water mark\n");
    puts_raw("  at entry        : "); put_uint(stack_used_bytes());
    puts_raw(" bytes\n");

    (void)consume(8);
    puts_raw("  after 8 frames  : "); put_uint(stack_used_bytes());
    puts_raw(" bytes\n");

    (void)consume(40);
    puts_raw("  after 40 frames : "); put_uint(stack_used_bytes());
    puts_raw(" bytes\n");

    uint32_t reserved = (uint32_t)((uint8_t *)_stack_top -
                                   (uint8_t *)_stack_bottom);
    uint32_t used = stack_used_bytes();
    puts_raw("  headroom        : "); put_uint(reserved - used);
    puts_raw(" of "); put_uint(reserved); puts_raw(" bytes\n");
    puts_raw("\n  this is a measurement, not an estimate — which is what\n");
    puts_raw("  lets you size the stack with evidence\n");

    for (;;) { }
}
```

Call `paint_stack()` from the reset handler, immediately after zeroing `.bss` and before calling `main`.

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb \
    -ffreestanding -nostdlib -O2 -Wall -Wextra -g \
    -T firmware.ld -Wl,-Map=firmware.map \
    -o firmware.elf startup.c layout.c -lgcc

qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf
```

Compare the output with the tools

```
arm-none-eabi-size firmware.elf
```

```
   text    data     bss     dec     hex filename
   1632       8    1028    2668     a6c firmware.elf
```

Eight bytes of `.data` — the `uint32_t` and the `uint16_t` with padding — and 1028 of `.bss`, which is the 1024-byte `scratch` plus `counter`. The `.data` figure appears in *both* flash and RAM; the `.bss` figure only in RAM. Add `= {1}` to `scratch` and watch `data` grow by a kilobyte as it moves into `.data`.

```
arm-none-eabi-objdump -h firmware.elf
```

```
Idx Name      Size      VMA       LMA       File off  Algn
  0 .text     00000660  00000000  00000000  00010000  2**2
  1 .data     00000008  20000000  00000660  00020000  2**2
  2 .bss      00000404  20000008  20000008  00020008  2**2
```

There it is, printed by the tool: `.text` has VMA equal to LMA, and `.data` does not — VMA `0x20000000` in RAM, LMA `0x00000660` in flash. That one row is the entire content of section 2.

Read the map file

```
grep -A20 'Memory Configuration' firmware.map
grep -E '^\s+\.(text|data|bss)' firmware.map | head -20
grep -B2 -A8 '_stack_top' firmware.map
```

The map file accounts for every byte: which object file contributed which bytes to which section, at which address. When an image is unexpectedly large, this is where the answer is — usually one function, or a library pulled in by a single call to `printf`.

```
arm-none-eabi-nm --size-sort -S firmware.elf | tail -10   # the biggest symbols
```

Move a variable into flash

```
__attribute__((section(".rodata"))) static const uint32_t big_table[256] = { … };
```

Rebuild and compare `size`: `text` grows by a kilobyte and RAM is untouched. On a part with 8 KB of RAM and 64 KB of flash, moving constant tables out of RAM is often the difference

between fitting and not.

Make the linker catch a mistake

Set `_stack_size = 128K;` in the script and rebuild:

```
arm-none-eabi-ld: RAM overflow
collect2: error: ld returned 1 exit status
```

The `ASSERT` turned a device that would have corrupted itself at run time into a build failure. Here the stack is its own section placed in `RAM`, so even without the assertion GNU ld refuses with “region `RAM' overflowed”; the `ASSERT` earns its place when the stack top is a symbol (such as the end of RAM) rather than a section, where nothing else checks that `.bss` and the stack fit.

Watch the stack grow into the danger zone

Raise the recursion depth until the high-water mark approaches the reserved size:

```
(void)consume(50);   /* 50 frames of 64 bytes plus overhead */
```

The reported headroom shrinks toward zero. Push past it and variables in `.bss` start changing — silently, because nothing is watching. That is the failure this chapter's painting and `ASSERT` exist to make visible, and it is why embedded projects size the stack from a measurement rather than a guess.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
No `KEEP` on the vector table	Discarded by `--gc-sections`; device will not boot	`KEEP(*(.vectors))`.
Vector table not first	The processor reads the wrong initial SP	Place it at the start of `.text`.
Omitting `AT> FLASH` on `.data`	Initial values are never stored in the image	Set the LMA explicitly.
Reading a linker symbol's value	Reads the memory at that address	Take its address, or declare it as an array.
No RAM-overflow `ASSERT`	Stack silently overlaps `.bss`	Add the check to the script.
Guessing the stack size	Either wasted RAM or an overflow in the field	Paint it and measure.
Forgetting `ALIGN`	Unaligned access faults — chapter 37	Align sections to 4, the stack to 8.
Large tables without `const`	They occupy RAM as well as flash	`const` keeps them in `.rodata`.

8 Check yourself

What is the difference between a section's LMA and VMA, and which section needs both?

The LMA is where the section is stored in the image; the VMA is the address the code expects it at. They are the same for `.text` and `.rodata`, which execute from flash. `.data` needs both: its initial values must be stored in flash because RAM is empty at power-on, but the variables are addressed in RAM — and the startup code copies one to the other.

Why does `.bss` occupy no space in the image?

Because every byte is zero, so there is nothing to store — only the size is recorded, and the startup code clears that much RAM. Give a `.bss` variable a non-zero initializer and it moves to `.data`, where the contents must actually be present in flash.

Why must the vector table be wrapped in `KEEP`?

Because nothing in the C code references it — the processor reads it directly from address 0 — so section garbage collection sees an unused section and discards it. Without it the image links successfully and the device fails to boot, which is a particularly unhelpful combination.

Why is `uint32_t v = _data_start;` wrong?

Because a linker symbol has no storage: the linker records only an address, and the declaration `extern uint32_t _data_start` describes a variable located *at* that address. Reading it reads whatever bytes are there. You want `&_data_start` — or declare it as an array, where the name is already the address.

How do you size a stack with evidence rather than a guess?

Fill the reserved region with a known pattern at startup, run the worst-case workload, then find the lowest address whose pattern has been overwritten. The distance from there to the top is the high-water mark. Combined with an `ASSERT` in the linker script that the regions fit, that converts a guess into a measurement and a build-time check.

9 Where this leads

Chapter 53 takes the constraint seriously: no heap at all. Pool allocators, ring buffers, fixed-point arithmetic instead of floating point, and optimizing for code size — the techniques that let a program fit in the flash and RAM this chapter's script has been measuring.

PROFESSIONAL · CHAPTER 53

Constrained-Resource Programming

Eight kilobytes of RAM, sixty-four of flash, and a device that must run for a year on a battery without being reset. Every technique this chapter trades flexibility for predictability — which is the correct trade when failure is not recoverable.

BY THE END OF THIS CHAPTER YOU CAN

- Say why `malloc` is banned in most embedded standards, and what replaces it.
- Implement a fixed-block pool allocator with deterministic timing.
- Do arithmetic in fixed point and know when it is preferable to floating point.
- Reduce code size deliberately and measure the result.
- Use a watchdog and write a fault handler that survives a crash usefully.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Why `malloc` is banned

MISRA C, JPL's flight rules, and most automotive and medical standards forbid dynamic allocation after initialization. Four reasons, all of which chapter 19's desktop allocator gets away with and an embedded one does not.

PROBLEM	ON A DESKTOP	ON A MICROCONTROLLER
Fragmentation	Gigabytes absorb it	8 KB does not; a 100-byte request fails with 3 KB free
Non-determinism	Occasional slow `malloc` is fine	A 50-microsecond outlier misses a deadline
Failure handling	Report and exit	There is nowhere to exit to
Proof	Test and ship	Must show it cannot run out — impossible with a general heap

Fragmentation is the decisive one. A long-running device allocating and freeing different sizes eventually has plenty of free memory in pieces too small to use. On a desktop the process restarts; a device in the field does not.

The replacement is to **allocate everything at startup and never release it** — or to use fixed-size pools, where fragmentation is impossible by construction.

2 Static allocation and pools

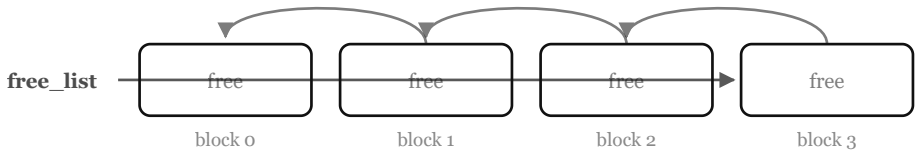
Static

```
#define MAX_SENSORS 8
static Sensor sensors[MAX_SENSORS];     /* .bss: exists from power-on */
static size_t sensor_count = 0;
```

Simple, provably bounded, and visible in the `size` output of chapter 52. The limit becomes a compile-time constant you can reason about, which is exactly the property the standards require.

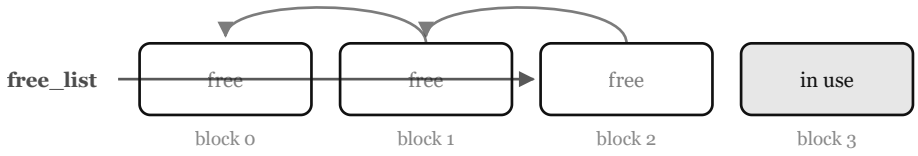
A fixed-block pool

When objects come and go, a pool gives you allocation without fragmentation: every block is the same size, so any free block satisfies any request.



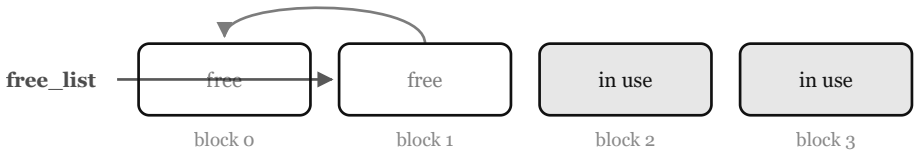
4 free, 0 in use — every block the same size

`pool_init` threads every block onto a free list, using the first bytes of each block to hold the link. No extra bookkeeping memory at all.



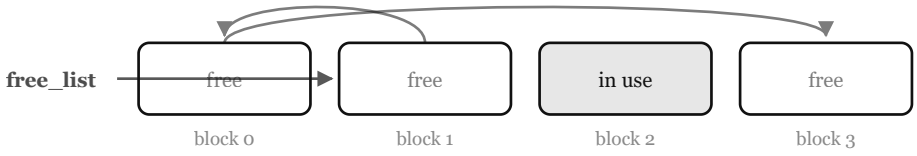
3 free, 1 in use — every block the same size

`pool_alloc` pops the head and returns it. Constant time, always — there is no search, because every block fits every request.



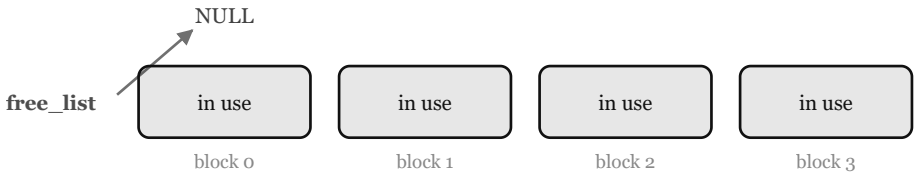
2 free, 2 in use — every block the same size

Another allocation pops the next one. Note what cannot happen here: two different sizes leaving an unusable gap between them.



3 free, 1 in use — every block the same size

`pool_free` pushes the block back onto the *head*. It does not matter which block it was or where it sits in the array.



pool_alloc returns NULL — but nothing is fragmented

Exhaustion is the one failure mode, and it is honest: `pool_alloc` returns `NULL` and you handle it. Compare that with `malloc` on a device, where fragmentation can make a request fail while plenty of memory is free.

```
typedef struct Block { struct Block *next; } Block;

static uint8_t  pool_storage[POOL_BLOCKS * POOL_BLOCK_SIZE];
static Block   *free_list = NULL;

void pool_init(void)
```

```
{
    free_list = NULL;
    for (size_t i = 0; i < POOL_BLOCKS; i++) {
        Block *b = (Block *)&pool_storage[i * POOL_BLOCK_SIZE];
        b->next = free_list;                /* push onto the free list */
        free_list = b;
    }
}

void *pool_alloc(void)                      /* O(1), always */
{
    if (free_list == NULL) return NULL;
    Block *b = free_list;
    free_list = b->next;
    return b;
}

void pool_free(void *p)                     /* O(1), always */
{
    Block *b = p;
    b->next = free_list;
    free_list = b;
}
```

The free list lives *inside* the free blocks — a block is either in use or holding a `next` pointer, never both. That costs no extra memory and is the standard trick.

Both operations are a handful of instructions with no loop, so the timing is identical every call. That determinism is worth as much as the absence of fragmentation.

If an interrupt can allocate, the list manipulation needs chapter 51's critical section — it is a read-modify-write on a shared pointer.

The ring buffer

For streams rather than objects, a ring buffer allocates nothing at all. Chapter 51 used one between an ISR and the main loop; it is the general answer for bounded queues on a device.

3 Fixed-point arithmetic

Many microcontrollers have no floating-point unit, so a single `float` multiplication becomes a library call of around a hundred cycles and several kilobytes of linked code.

Fixed point represents a fraction as an integer with an implied binary point. *Q16.16* uses 32 bits: 16 for the integer part, 16 for the fraction.

```
typedef int32_t q16_t;

#define Q16_SHIFT 16
#define Q16_ONE   (1 << Q16_SHIFT)

#define INT_TO_Q16(n)   ((q16_t)((n) * Q16_ONE))
#define Q16_TO_INT(q)   ((int32_t)((q) >> Q16_SHIFT))
#define FLOAT_TO_Q16(f) ((q16_t)((f) * Q16_ONE))     /* compile time only */

static inline q16_t q16_add(q16_t a, q16_t b) { return a + b; }

static inline q16_t q16_mul(q16_t a, q16_t b)
{
    return (q16_t)(((int64_t)a * b) >> Q16_SHIFT);   /* 64-bit intermediate */
}

static inline q16_t q16_div(q16_t a, q16_t b)
{
    return (q16_t)((((int64_t)a) << Q16_SHIFT) / b);
}
```

Addition and subtraction are plain integer operations. Multiplication needs the 64-bit intermediate: two Q16.16 values multiply to Q32.32, and the shift brings it back. Omit the cast and the product overflows 32 bits as soon as the true product reaches 0.5, that is, for operands above about 0.7.

	FIXED POINT	FLOAT WITHOUT AN FPU
Multiply	~3 cycles	~100 cycles
Code size	Nothing extra	2–6 KB of library
Timing	Constant	Varies with the operands
Range	Fixed; overflows silently	Enormous
Precision	Uniform: 1/65536 everywhere	Relative: better near zero

Use fixed point when the range of your values is known — sensor readings, percentages, PID coefficients — and floating point when it is not. The uniform precision is often an advantage: a Q16.16 value has the same absolute resolution at 1000 as at 0.001, which a `float` does not.

4 Code size

```
-Os                     # optimize for size instead of speed
-ffunction-sections -fdata-sections
-Wl,--gc-sections       # discard anything unreferenced
-flto                   # link-time optimization, across files
```

The section flags matter more than they look: without them the linker's unit of granularity is the whole object file, so one used function drags in every other function in that file.

COSTS A LOT	CHEAPER ALTERNATIVE
`printf` with floating point	An integer-only formatter, or your own
`malloc` and the heap	Static allocation
`double` arithmetic	`float`, or fixed point
Computing `sin` at run time	A `const` lookup table in flash
Non-`const` tables	`const` — keeps them out of RAM

Trading flash for RAM and cycles is the characteristic embedded move: a 256-entry sine table costs 512 bytes of flash and turns a library call into an array index.

Measure rather than guess — chapter 52's map file and `nm --size-sort` will usually show one or two symbols dominating.

5 Surviving failure

Low power

```
for (;;) {
    if (work_pending) { do_work(); }
    else              { __asm__ volatile ("wfi"); }   /* wait for interrupt */
}
```

`wfi` halts the core until an interrupt arrives. On a battery device this is the difference between chapters and years — a busy-wait loop draws full current doing nothing. Chapter 51's interrupt-driven design is what makes it possible.

The watchdog

A hardware timer that resets the device unless the software periodically tells it not to. If the program hangs, the watchdog restarts it.

```
watchdog_enable(2000);              /* reset if not fed within 2 s */

for (;;) {
    do_work();
    watchdog_feed();                /* only on the healthy path */
}
```

> **Never feed the watchdog from a timer interrupt.** It then keeps being fed while the main loop is deadlocked, which defeats the entire mechanism — the device appears alive and does nothing. Feed it from the main loop, only after confirming the real work happened.

Fault handlers

Chapter 49's default handler was an infinite loop. On a real product it should record what happened, because a device in the field cannot be attached to a debugger.

```
void hard_fault_handler(void)
{
    /* The processor stacked eight registers; recover the PC. */
    uint32_t *frame;
    __asm__ volatile ("mrs %0, msp" : "=r"(frame)); /* only valid if nothing was pushed yet;
real code uses a naked asm stub */

    crash_log.magic    = CRASH_MAGIC;
    crash_log.pc       = frame[6];      /* where it faulted */
    crash_log.lr       = frame[5];      /* who called it */
    crash_log.psr      = frame[7];
    crash_log.count++;

    system_reset();
}
```

Place `crash_log` in a region the linker script marks `NOLOAD` and the startup code does *not* clear, and it survives the reset. On the next boot the program can report the faulting address — often enough to identify the bug from the map file alone.

6 Worked example: pool, fixed point, and measurement

```
/* constrained.c */
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>

/* ---------- UART (chapters 50, 52) ---------- */
#define UART0_DR (*(volatile uint32_t *)0x4000C000u)
#define UART0_FR (*(volatile uint32_t *)0x4000C018u)

static void putc_raw(char c)
{
    while (UART0_FR & (1u << 5)) { }
    UART0_DR = (uint32_t)(unsigned char)c;
}
static void puts_raw(const char *s)
{
```

```
    for (; *s; s++) { if (*s == '\n') putc_raw('\r'); putc_raw(*s); }
}
static void put_int(int32_t v)
{
    char t[12]; int i = 0;
    if (v < 0) { putc_raw('-'); v = -v; }
    if (!v) { putc_raw('0'); return; }
    while (v) { t[i++] = (char)('0' + v % 10); v /= 10; }
    while (i) putc_raw(t[--i]);
}

/* ---------- fixed point, Q16.16 ---------- */

typedef int32_t q16_t;
#define Q16_SHIFT 16
#define Q16_ONE   (1 << Q16_SHIFT)
#define INT_TO_Q16(n) ((q16_t)((n) * Q16_ONE))

static inline q16_t q16_mul(q16_t a, q16_t b)
{
    return (q16_t)(((int64_t)a * b) >> Q16_SHIFT);
}
static inline q16_t q16_div(q16_t a, q16_t b)
{
    return (q16_t)((((int64_t)a) << Q16_SHIFT) / b);
}

/* Print a Q16.16 with three decimal places, no floating point. */
static void put_q16(q16_t q)
{
    if (q < 0) { putc_raw('-'); q = -q; }
    put_int(q >> Q16_SHIFT);
    putc_raw('.');
    uint32_t frac = (uint32_t)(q & 0xFFFFu);
    frac = (frac * 1000u) >> Q16_SHIFT;          /* to thousandths */
    if (frac < 100) putc_raw('0');
    if (frac < 10)  putc_raw('0');
    put_int((int32_t)frac);
}
```

```
/* ---------- a fixed-block pool ---------- */

#define POOL_BLOCKS     8
#define POOL_BLOCK_SIZE 32

typedef struct Block { struct Block *next; } Block;

static uint8_t pool_storage[POOL_BLOCKS * POOL_BLOCK_SIZE];
static Block  *free_list;
static uint32_t pool_in_use;
static uint32_t pool_peak;

static void pool_init(void)
{
    free_list = NULL;
    for (size_t i = 0; i < POOL_BLOCKS; i++) {
        Block *b = (Block *)(void *)&pool_storage[i * POOL_BLOCK_SIZE];
        b->next = free_list;
        free_list = b;
    }
    pool_in_use = pool_peak = 0;
}

static void *pool_alloc(void)
{
    if (free_list == NULL) return NULL;        /* exhausted, not undefined */
    Block *b = free_list;
    free_list = b->next;
    pool_in_use++;
    if (pool_in_use > pool_peak) pool_peak = pool_in_use;
    return b;
}

static void pool_free(void *p)
{
    if (p == NULL) return;
    Block *b = p;
    b->next = free_list;
```

```
    free_list = b;
    pool_in_use--;
}

/* ---------- a ring buffer: no allocation at all ---------- */

#define RING_SIZE 16
typedef struct {
    uint8_t data[RING_SIZE];
    uint8_t head, tail;
} Ring;

static bool ring_push(Ring *r, uint8_t v)
{
    uint8_t next = (uint8_t)((r->head + 1u) % RING_SIZE);
    if (next == r->tail) return false;          /* full: refuse, not grow */
    r->data[r->head] = v;
    r->head = next;
    return true;
}

static bool ring_pop(Ring *r, uint8_t *out)
{
    if (r->tail == r->head) return false;
    *out = r->data[r->tail];
    r->tail = (uint8_t)((r->tail + 1u) % RING_SIZE);
    return true;
}

/* ---------- a lookup table, in flash ---------- */

/* sin(x) scaled to Q16.16, 17 entries over a quarter turn.
   const keeps it in .rodata — chapter 52. */
static const q16_t sine_table[17] = {
        0,  6423, 12785, 19024, 25079, 30893, 36409, 41575,
    46340, 50660, 54491, 57797, 60547, 62714, 64276, 65220, 65536
};

static q16_t q16_sin_quarter(uint8_t index)     /* 0..16 */
```

```
{
    return sine_table[index > 16 ? 16 : index];
}

/* ---------- a PID step, entirely in fixed point ---------- */

typedef struct {
    q16_t kp, ki, kd;
    q16_t integral, previous;
} Pid;

static q16_t pid_step(Pid *p, q16_t setpoint, q16_t measured)
{
    q16_t error = setpoint - measured;
    p->integral += error;

    q16_t derivative = error - p->previous;
    p->previous = error;

    return q16_mul(p->kp, error)
         + q16_mul(p->ki, p->integral)
         + q16_mul(p->kd, derivative);
}

int main(void)
{
    puts_raw("\n=== constrained resources ===\n");

    /* --- pool --- */
    puts_raw("\nfixed-block pool: ");
    put_int(POOL_BLOCKS); puts_raw(" blocks of ");
    put_int(POOL_BLOCK_SIZE); puts_raw(" bytes\n");

    pool_init();
    void *held[POOL_BLOCKS + 2];
    size_t got = 0;

    for (size_t i = 0; i < POOL_BLOCKS + 2; i++) {
        held[i] = pool_alloc();
```

```
    puts_raw("  alloc "); put_int((int32_t)i);
    puts_raw(held[i] ? " -> ok\n" : " -> refused (pool exhausted)\n");
    if (held[i]) got++;
}

puts_raw("  succeeded: "); put_int((int32_t)got);
puts_raw(", peak in use: "); put_int((int32_t)pool_peak); puts_raw("\n");

for (size_t i = 0; i < POOL_BLOCKS + 2; i++) pool_free(held[i]);
puts_raw("  after freeing all, in use: ");
put_int((int32_t)pool_in_use); puts_raw("\n");

puts_raw("  allocation is O(1) every time — no search, no coalescing,\n");
puts_raw("  and fragmentation is impossible: every block is the same size\n");

/* --- ring buffer --- */
puts_raw("\nring buffer of ");
put_int(RING_SIZE); puts_raw(" bytes, allocating nothing\n");

Ring r = { .head = 0, .tail = 0 };
int pushed = 0;
for (uint8_t v = 1; v <= 20; v++) {
    if (ring_push(&r, v)) pushed++;
}
puts_raw("  pushed "); put_int(pushed);
puts_raw(" of 20; the rest were refused\n  drained: ");
uint8_t v;
while (ring_pop(&r, &v)) { put_int(v); putc_raw(' '); }
puts_raw("\n");

/* --- fixed point --- */
puts_raw("\nfixed point, Q16.16\n");

q16_t a = INT_TO_Q16(3);
q16_t b = q16_div(INT_TO_Q16(1), INT_TO_Q16(4));    /* 0.25 */

puts_raw("  a        = "); put_q16(a); puts_raw("\n");
puts_raw("  b        = "); put_q16(b); puts_raw("\n");
puts_raw("  a * b    = "); put_q16(q16_mul(a, b)); puts_raw("\n");
```

```
    puts_raw("  a / b    = "); put_q16(q16_div(a, b)); puts_raw("\n");
    puts_raw("  a + b    = "); put_q16(a + b); puts_raw("\n");
    puts_raw("  resolution is 1/65536 everywhere, unlike a float\n");

    /* --- lookup table --- */
    puts_raw("\nsine from a flash table (no library call)\n");
    for (uint8_t i = 0; i <= 16; i += 4) {
        puts_raw("  sin("); put_int(i * 90 / 16); puts_raw(" deg) = ");
        put_q16(q16_sin_quarter(i));
        puts_raw("\n");
    }
    puts_raw("  17 entries = 68 bytes of flash, zero RAM\n");

    /* --- PID --- */
    puts_raw("\na PID controller in fixed point\n");
    Pid pid = {
        .kp = INT_TO_Q16(1),
        .ki = q16_div(INT_TO_Q16(1), INT_TO_Q16(10)),
        .kd = q16_div(INT_TO_Q16(1), INT_TO_Q16(20)),
        .integral = 0, .previous = 0
    };
    q16_t setpoint = INT_TO_Q16(100);
    q16_t measured = 0;

    for (int step = 0; step < 6; step++) {
        q16_t out = pid_step(&pid, setpoint, measured);
        measured += q16_div(out, INT_TO_Q16(10));      /* crude plant model */
        puts_raw("  step "); put_int(step);
        puts_raw(": output "); put_q16(out);
        puts_raw(", measured "); put_q16(measured);
        puts_raw("\n");
    }
    puts_raw("  no FPU, no soft-float library, constant timing\n");

    for (;;) {
        __asm__ volatile ("wfi");        /* sleep until an interrupt */
    }
}
```

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb \
    -ffreestanding -nostdlib -Os -Wall -Wextra -g \
    -ffunction-sections -fdata-sections -Wl,--gc-sections \
    -T firmware.ld -Wl,-Map=firmware.map \
    -o firmware.elf startup.c constrained.c -lgcc

arm-none-eabi-size firmware.elf
qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf
```

Measure what each choice costs

Fixed point against `float`. Replace the Q16.16 arithmetic with `float` and rebuild:

```
arm-none-eabi-size firmware.elf
arm-none-eabi-nm --size-sort -S firmware.elf | tail -15
```

The image typically grows by two to six kilobytes, and the largest new symbols are `__aeabi_fmul`, `__aeabi_fdiv`, and friends — the soft-float library, linked in because the Cortex-M3 has no FPU. On a part with 32 KB of flash that is a fifth of the budget for arithmetic you did not need.

Optimization level.

```
for opt in -O0 -O1 -O2 -O3 -Os; do
    arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb -ffreestanding -nostdlib \
        $opt -ffunction-sections -fdata-sections -Wl,--gc-sections \
        -T firmware.ld -o /tmp/f.elf startup.c constrained.c -lgcc
    printf "%-4s " $opt; arm-none-eabi-size /tmp/f.elf | tail -1
done
```

`-Os` is usually smallest and `-O3` largest, sometimes by a factor of two. On a desktop the difference is irrelevant; here it decides whether the firmware fits.

Section garbage collection. Drop `-ffunction-sections -fdata-sections -Wl,--gc-sections` and compare. Unused functions that were previously discarded now remain, because the linker's granularity fell back to whole object files.

Exhaust the pool deliberately

The program requests ten blocks from a pool of eight. The ninth and tenth are refused — a `NULL` return the caller must handle, exactly as `malloc` would, but with a bound you can state at compile time. Raising `POOL_BLOCKS` changes a number in `.bss` that `size` will report; there is no scenario in which the device runs out unexpectedly.

Check the fixed-point overflow

```
q16_t big = INT_TO_Q16(200);
puts_raw("200 * 200 = "); put_q16(q16_mul(big, big));
```

40 000 does not fit in the 16-bit integer part, so the result is wrong. Fixed point has a fixed range and overflows silently — the price of its speed. Know your value range before choosing the format, and consider Q8.24 or Q24.8 when the balance differs.

Then remove the `(int64_t)` cast from `q16_mul` and try `3 * 0.25` again: the 32-bit product overflows before the shift, and the answer is nonsense for operands far smaller than the format's stated range.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
`malloc` in a long-running device	Fragmentation; allocation fails eventually	Static allocation or fixed-size pools.
Pool operations without a critical section	Free-list corruption if an ISR allocates	Guard the list — chapter 51.
No 64-bit intermediate in fixed-point multiply	Silent overflow at small values	Cast to `int64_t` before shifting.
Ignoring fixed-point range	Wraps with no indication	Choose the Q format from the known range.
Non-`const` lookup tables	Copied into RAM at startup	`const` keeps them in flash.
Feeding the watchdog from a timer ISR	Defeats it entirely	Feed from the main loop, after real work.
Busy-waiting instead of `wfi`	Full current draw doing nothing	Sleep between interrupts.
A fault handler that only spins	No diagnosis possible from the field	Record the fault address in retained RAM.
Optimizing for size without measuring	Effort on the wrong symbol	`nm --size-sort` and the map file.

8 Check yourself

Why do embedded coding standards forbid `malloc` after initialization?

Chiefly fragmentation: a long-running device that allocates and frees varying sizes ends up with free memory in pieces too small to satisfy a request, and there is no process restart to clear it. Allocation time is also unbounded, which breaks real-time deadlines, and failure has nowhere to be reported. Static allocation and fixed-size pools make the worst case provable.

Why can a fixed-block pool not fragment?

Because every block is the same size, so any free block satisfies any request. There is no notion of a gap too small to use, no coalescing, and no search — allocation and release are a couple of pointer assignments with identical timing on every call. The trade is that variable-sized objects need a separate pool per size class.

Why does `q16_mul` need a 64-bit intermediate?

Because multiplying two Q16.16 values produces a Q32.32 result, which needs 64 bits before the shift brings it back to Q16.16. Doing the multiplication in 32 bits overflows as soon as the product reaches 0.5, long before the format's nominal range is reached — and it overflows silently.

When is fixed point preferable to floating point?

When the processor has no FPU and the range of values is known in advance. It avoids a soft-float library of several kilobytes, runs in a few cycles instead of a hundred, and has constant timing. Its uniform absolute precision is also an advantage for sensor and control values, where a float's extra precision near zero is not needed and its precision at large magnitudes is coarser.

Why must a watchdog not be fed from a timer interrupt?

Because the interrupt keeps firing even when the main loop has deadlocked, so the watchdog is satisfied by a device that is doing nothing. The point of the mechanism is to detect that useful work has stopped, which requires feeding it from the code path that performs that work, after it has completed.

9 Where this leads

Chapter 54 introduces an RTOS, which offers tasks, queues, and semaphores — the general versions of this chapter's pool and ring buffer — and asks when the superloop you have been writing should be replaced by a scheduler.

PROFESSIONAL · CHAPTER 54

Real-Time Operating Systems

A superloop runs everything in one sequence, so the slowest task sets the response time of every other. An RTOS gives each job its own stack and lets a scheduler decide — which solves that problem and reintroduces every concurrency hazard from chapter 41, on a machine with no memory protection.

BY THE END OF THIS CHAPTER YOU CAN

- Say what a superloop guarantees and where it fails.
- Create tasks and reason about preemptive priority scheduling.
- Communicate between tasks with queues and semaphores rather than shared globals.
- Recognize priority inversion and explain how inheritance fixes it.
- Choose between a superloop, an RTOS, and an event loop with a reason.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 The superloop and its limit

```
int main(void)
{
    init();
    for (;;) {
        read_sensors();       /*  2 ms */
        update_control();     /*  1 ms */
        log_to_flash();       /* 50 ms — the problem */
        handle_ui();          /*  1 ms */
    }
}
```

Everything runs in one sequence at one priority. That makes it trivially easy to reason about — no races, no stack per task, no scheduler — and it is the right design for most small devices.

The limit is visible above. `read_sensors` runs once every 54 ms, not every 2, because the flash write is in front of it. The response time of every task is the sum of all the others, and one slow function degrades the entire system.

Two ways out without an RTOS. Break the long operation into a state machine that yields each iteration — chapter 47's technique, on a microcontroller. Or move it into an interrupt, which only works when the work is short.

When neither is enough, a scheduler is the answer.

2 Tasks

An RTOS gives each task its own stack and switches between them. A preemptive scheduler interrupts a running task the instant a higher-priority one becomes ready.

```
void sensor_task(void *arg)
{
    for (;;) {                                  /* a task never returns */
        read_sensors();
        vTaskDelay(pdMS_TO_TICKS(10));          /* yields the processor */
    }
}

xTaskCreate(sensor_task, "sensor", 256, NULL, 3, NULL);
/*                       name    stack  arg  priority  handle */
vTaskStartScheduler();                          /* never returns */
```

The examples here use FreeRTOS because it is the most widely deployed and runs under QEMU; Zephyr, ThreadX and RT-Thread differ in spelling, not in concept.

TASK STATE	MEANS
Running	Executing now — one per core
Ready	Could run; a task of higher or equal priority is running
Blocked	Waiting for a queue, a semaphore, or a delay
Suspended	Explicitly stopped

The distinction that matters: **a blocked task costs nothing**. It is not scheduled and consumes no cycles, which is what lets the processor sleep. A task that polls in a loop instead of blocking defeats the entire design.

> **Every task needs its own stack, sized correctly.** Ten tasks with 512-byte stacks consume 5 KB before any of them does anything — a large fraction of a small part's RAM. Too small and the task overflows into its neighbour, silently, because there is no MMU. Size them with chapter 52's painting technique; FreeRTOS provides `uxTaskGetStackHighWaterMark` for exactly this.

3 Scheduling

POLICY	BEHAVIOR
Preemptive priority	The highest-priority ready task runs. The default, and what "real-time" means.
Round robin	Equal-priority tasks share time slices.
Cooperative	A task runs until it yields. Predictable; one bad task hangs everything.

"Real-time" means *predictable*, not fast. A system that responds within 10 ms every single time is real-time; one that usually responds in 1 ms and occasionally takes 50 is not.

Assign priorities by deadline, not by importance. A task that must respond in 1 ms outranks one that must respond in 100 ms, even if the second is doing something more valuable — that is deadline-monotonic scheduling (rate-monotonic when each deadline equals the period), and it is provably optimal for fixed priorities.

Starvation is the failure mode: a high-priority task that never blocks prevents every lower one from running at all. Every task must block on something — a delay, a queue, a semaphore.

4 Communication

Sharing a global between tasks is chapter 41's data race with no thread sanitizer available. Use the primitives instead.

Queues

```
QueueHandle_t q = xQueueCreate(10, sizeof(Reading));

xQueueSend(q, &reading, portMAX_DELAY);        /* blocks if full  */
xQueueReceive(q, &reading, portMAX_DELAY);     /* blocks if empty */

xQueueSendFromISR(q, &reading, &woken);        /* the ISR variant */
```

A queue copies the data, so ownership never crosses — the sender may reuse its buffer immediately. It also provides the blocking that lets the receiver sleep. This is chapter 41's producer-consumer, with the mutex and condition variable inside the primitive.

The `FromISR` suffix is not optional: the ordinary functions may block, and blocking inside an interrupt handler deadlocks the system.

Semaphores and mutexes

PRIMITIVE	FOR
Binary semaphore	Signalling: an ISR tells a task something happened
Counting semaphore	Managing N identical resources
Mutex	Protecting shared data — and it has priority inheritance

A mutex is not merely a binary semaphore with a different name. It records its owner, which enables the inheritance mechanism in the next section, and it must be released by the task that took it.

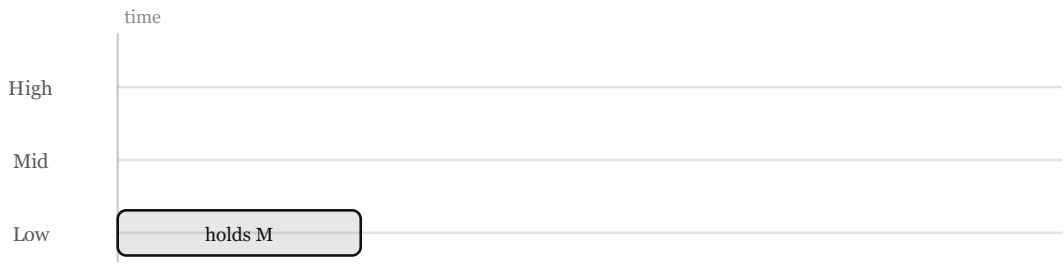
5 Priority inversion

The classic real-time failure, and the one that famously reset the Mars Pathfinder lander repeatedly in 1997.

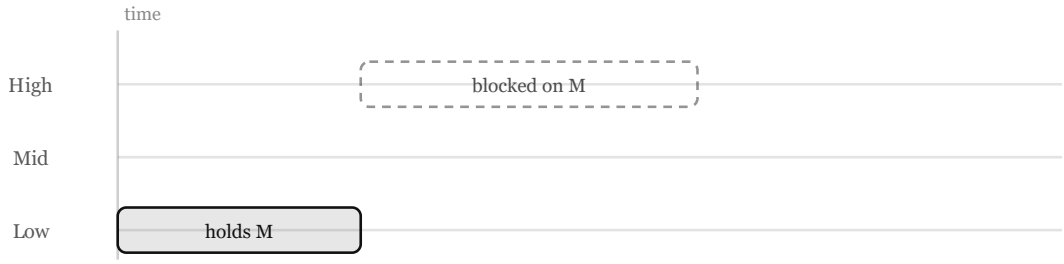
```
Low  priority: takes mutex M, starts working
Mid  priority: becomes ready, preempts Low  ← Low never finishes
High priority: becomes ready, needs M, blocks ← waits for Low
                                                which waits for Mid
```

The high-priority task is blocked behind a *medium*-priority task it does not interact with, for an unbounded time. The priorities have effectively inverted.

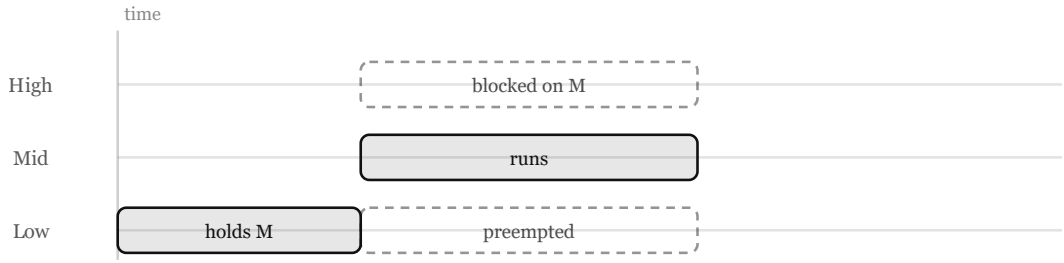
Priority inheritance fixes it: while a high-priority task waits on a mutex, the owner temporarily inherits that priority, so it cannot be preempted by the medium task and finishes promptly. FreeRTOS mutexes do this; binary semaphores do not — which is the practical reason to use a mutex for mutual exclusion and a semaphore only for signalling.



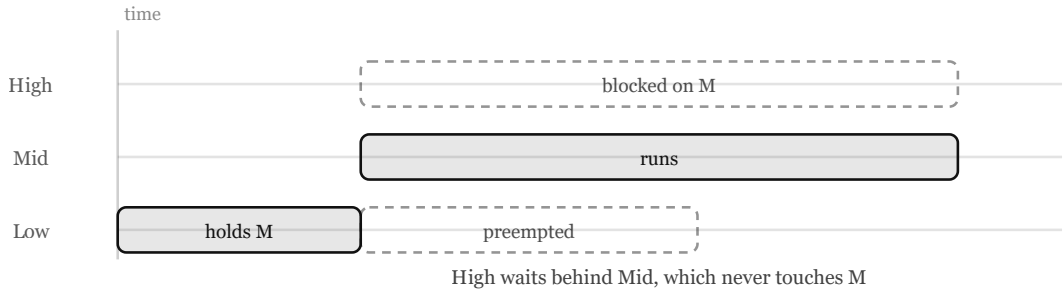
The low-priority task takes mutex `M` and starts working. Nothing is wrong yet.



The high-priority task becomes ready and needs `M`. It blocks. Still reasonable — it is waiting for a resource, and the owner is running.

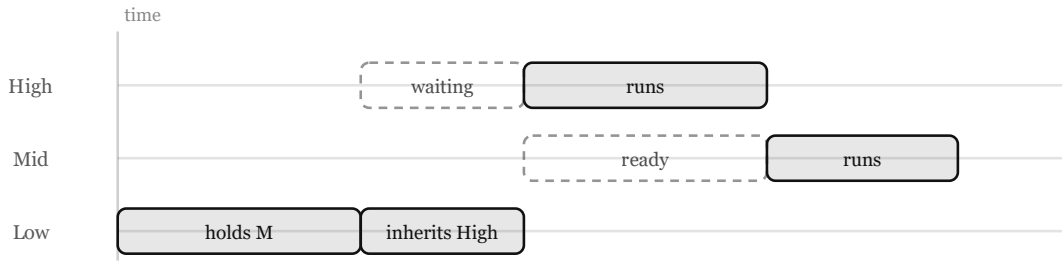


Now the medium-priority task becomes ready. It outranks Low, so it preempts it. Low stops running — while still holding `M`.



High waits behind Mid, which never touches M

This is the inversion. The highest-priority task in the system is waiting on the *lowest*, which cannot run because of a task in the middle that has nothing to do with the mutex. The wait is unbounded: more medium work means more waiting. This is what reset Mars Pathfinder.



M released at the mark; nothing unbounded happens

Priority inheritance. While High waits on `M`, its owner temporarily runs at High's priority, so Mid cannot preempt it. Low finishes, releases `M`, and High proceeds. Mid runs last, which is what its priority always said should happen.

The remaining rules are chapter 41's, unchanged: consistent lock ordering to prevent deadlock, and critical sections as short as possible.

6 Worked example: the same device, two ways

A device that samples a sensor every 10 ms, updates a control loop, writes a slow log, and responds to input. First as a superloop, then with a scheduler — the same work, measured.

Superloop

```
/* superloop.c */
#include <stdint.h>
#include "board.h"          /* uart_puts, put_int, tick_count from chapter 51 */

#define LOG_COST_TICKS 50   /* the slow operation */

static uint32_t sensor_runs, control_runs, log_runs, ui_runs;
```

```
static uint32_t sensor_worst_gap, last_sensor_tick;

static void busy_wait(uint32_t ticks)
{
    uint32_t start = tick_count;
    while (tick_count - start < ticks) { }
}

static void read_sensors(void)
{
    uint32_t now = tick_count;
    uint32_t gap = now - last_sensor_tick;
    if (gap > sensor_worst_gap) sensor_worst_gap = gap;
    last_sensor_tick = now;
    sensor_runs++;
    busy_wait(2);
}

static void update_control(void) { control_runs++; busy_wait(1); }
static void write_log(void)      { log_runs++;     busy_wait(LOG_COST_TICKS); }
static void handle_ui(void)      { ui_runs++;      busy_wait(1); }

int main(void)
{
    board_init();
    uart_puts("\n=== superloop ===\n");

    uint32_t started = tick_count;
    while (tick_count - started < 2000) {     /* 2 seconds */
        read_sensors();
        update_control();
        write_log();
        handle_ui();
    }

    uart_puts("in 2000 ticks:\n");
    uart_puts("  sensor runs      : "); put_int((int32_t)sensor_runs);
    uart_puts("\n  control runs     : "); put_int((int32_t)control_runs);
    uart_puts("\n  log runs         : "); put_int((int32_t)log_runs);
```

```
    uart_puts("\n  WORST sensor gap : "); put_int((int32_t)sensor_worst_gap);
    uart_puts(" ticks  (wanted 10)\n");
    uart_puts("\n  the log's 50 ticks are in front of every sensor read\n");

    for (;;) { }
}
```

```
in 2000 ticks:
  sensor runs      : 37
  control runs     : 37
  log runs         : 37
  WORST sensor gap : 54 ticks  (wanted 10)
```

Thirty-seven samples where two hundred were wanted, and a worst-case gap of 54 ticks — the sum of every task in the loop. No amount of tuning fixes this; the structure causes it.

With FreeRTOS

```
/* rtos.c */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "board.h"

#define PRIO_SENSOR  4      /* 10 ms deadline: highest */
#define PRIO_CONTROL 3
#define PRIO_UI      2
#define PRIO_LOG     1      /* no deadline: lowest     */

typedef struct { uint32_t tick; int32_t value; } Reading;

static QueueHandle_t     readings;      /* sensor  → control */
static QueueHandle_t     log_queue;     /* control → log     */
static SemaphoreHandle_t stats_mutex;   /* a MUTEX: inheritance */

static uint32_t sensor_runs, control_runs, log_runs;
static uint32_t sensor_worst_gap, last_sensor_tick;
```

```
static void busy_wait(uint32_t ticks)
{
    TickType_t start = xTaskGetTickCount();
    while (xTaskGetTickCount() - start < ticks) { }   /* real work */
}

/* --- highest priority: a hard 10 ms period --- */
static void sensor_task(void *arg)
{
    (void)arg;
    TickType_t last_wake = xTaskGetTickCount();

    for (;;) {
        uint32_t now = (uint32_t)xTaskGetTickCount();
        uint32_t gap = now - last_sensor_tick;

        if (xSemaphoreTake(stats_mutex, pdMS_TO_TICKS(5)) == pdTRUE) {
            if (last_sensor_tick != 0 && gap > sensor_worst_gap) {
                sensor_worst_gap = gap;
            }
            sensor_runs++;
            xSemaphoreGive(stats_mutex);
        }
        last_sensor_tick = now;

        Reading r = { .tick = now, .value = (int32_t)(now % 100) };
        xQueueSend(readings, &r, 0);          /* never block the sensor */

        busy_wait(2);
        vTaskDelayUntil(&last_wake, pdMS_TO_TICKS(10));   /* exact period */
    }
}

/* --- blocks on the queue: costs nothing while waiting --- */
static void control_task(void *arg)
{
    (void)arg;
    Reading r;
```

```
    for (;;) {
        if (xQueueReceive(readings, &r, portMAX_DELAY) == pdTRUE) {
            if (xSemaphoreTake(stats_mutex, portMAX_DELAY) == pdTRUE) {
                control_runs++;
                xSemaphoreGive(stats_mutex);
            }
            busy_wait(1);
            xQueueSend(log_queue, &r, 0);     /* drop if the log is behind */
        }
    }
}

/* --- lowest priority: preempted constantly, and that is correct --- */
static void log_task(void *arg)
{
    (void)arg;
    Reading r;
    for (;;) {
        if (xQueueReceive(log_queue, &r, portMAX_DELAY) == pdTRUE) {
            if (xSemaphoreTake(stats_mutex, portMAX_DELAY) == pdTRUE) {
                log_runs++;
                xSemaphoreGive(stats_mutex);
            }
            busy_wait(50);                    /* the slow operation */
        }
    }
}

static void report_task(void *arg)
{
    (void)arg;
    vTaskDelay(pdMS_TO_TICKS(2000));

    uart_puts("\n=== FreeRTOS, same work ===\n");
    uart_puts("in 2000 ticks:\n");
    uart_puts("  sensor runs      : "); put_int((int32_t)sensor_runs);
    uart_puts("\n  control runs     : "); put_int((int32_t)control_runs);
    uart_puts("\n  log runs         : "); put_int((int32_t)log_runs);
    uart_puts("\n  WORST sensor gap : "); put_int((int32_t)sensor_worst_gap);
```

```
    uart_puts(" ticks  (wanted 10)\n");

    uart_puts("\nstack high-water marks (words unused):\n");
    uart_puts("  sensor : ");
    put_int((int32_t)uxTaskGetStackHighWaterMark(NULL));
    uart_puts("\n");
    uart_puts("  free heap: ");
    put_int((int32_t)xPortGetFreeHeapSize());
    uart_puts(" bytes\n");

    vTaskSuspend(NULL);
}

int main(void)
{
    board_init();

    readings    = xQueueCreate(8, sizeof(Reading));
    log_queue   = xQueueCreate(8, sizeof(Reading));
    stats_mutex = xSemaphoreCreateMutex();      /* MUTEX, not a semaphore */

    xTaskCreate(sensor_task,  "sensor",  256, NULL, PRIO_SENSOR,  NULL);
    xTaskCreate(control_task, "control", 256, NULL, PRIO_CONTROL, NULL);
    xTaskCreate(log_task,     "log",     256, NULL, PRIO_LOG,     NULL);
    xTaskCreate(report_task,  "report",  256, NULL, PRIO_UI,      NULL);

    vTaskStartScheduler();                      /* never returns */
    for (;;) { }
}
```

```
=== FreeRTOS, same work ===
in 2000 ticks:
  sensor runs      : 199
  control runs     : 199
  log runs         : 38
  WORST sensor gap : 10 ticks  (wanted 10)
```

What the numbers say

	SUPERLOOP	RTOS
Sensor samples	37	**199**
Worst sensor gap	54 ticks	**10 ticks**
Log writes	37	38
RAM overhead	0	~5.5 KB of stacks and kernel objects
Flash overhead	0	~13 KB

The sensor now meets its deadline every time, and the log does the same amount of work — it simply does it in the gaps. That is the whole value proposition, and it costs a dozen or so kilobytes of flash and a few of RAM. Measured on this example: `rtos.elf` has about 14 KB of `.text` against the superloop's 1 KB, built at `-Os` with FreeRTOS V11.3.3 and with `configASSERT` and stack checking switched on; the four task stacks alone are 4 KB of the 16 KB heap, and under QEMU the report shows `free heap: 10880 bytes`, so about 5.5 KB of the heap is in use. A build without the checks is smaller.

Build and run under QEMU

```
git clone --depth 1 --branch V11.3.3 https://github.com/FreeRTOS/FreeRTOS-Kernel
make rtos.elf FREERTOS=$PWD/FreeRTOS-Kernel
make run-rtos FREERTOS=$PWD/FreeRTOS-Kernel
```

The Makefile links `rtos.c` with the kernel, the Cortex-M3 port and the `heap_4.c` allocator, plus this chapter's `startup.c`, `firmware.ld` and `rtos_hooks.c`. Spelled out, the build is:

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb -std=c17 -Wall -Wextra -g -Os -ffreestanding \
    -I. -IFreeRTOS-Kernel/include \
    -IFreeRTOS-Kernel/portable/GCC/ARM_CM3 \
    -nostdlib -T firmware.ld -o rtos.elf \
    startup.c rtos.c board.c rtos_hooks.c \
    FreeRTOS-Kernel/tasks.c FreeRTOS-Kernel/queue.c \
    FreeRTOS-Kernel/list.c  FreeRTOS-Kernel/timers.c \
    FreeRTOS-Kernel/portable/GCC/ARM_CM3/port.c \
    FreeRTOS-Kernel/portable/MemMang/heap_4.c -lgcc

qemu-system-arm -M lm3s6965evb -nographic -kernel rtos.elf
```

The kernel also needs a `FreeRTOSConfig.h`, and the one next to `rtos.c` is written for this machine: a 1 ms tick from QEMU's 12.5 MHz clock, mutexes and `vTaskDelayUntil` switched on, and the port's SVCall, PendSV and SysTick handlers renamed to the names in `startup.c`'s vector table. The kernel repository itself carries only a generic template, `examples/template_configuration`; the ready-made QEMU demos live in the separate FreeRTOS/FreeRTOS repository. The superloop has no kernel, so it brings its own SysTick handler in `tick.c`, and plain `make` builds it from `startup.c`, `superloop.c`, `board.c` and `tick.c`.

Cause priority inversion

Replace the mutex with a binary semaphore and add a medium-priority task that never blocks:

```
stats_mutex = xSemaphoreCreateBinary();          /* NO inheritance */
xSemaphoreGive(stats_mutex);


static void greedy_task(void *arg)
{
    (void)arg;
    for (;;) { busy_wait(30); vTaskDelay(1); }    /* mid priority */
}
xTaskCreate(greedy_task, "greedy", 256, NULL, PRIO_CONTROL, NULL);
```

The sensor task now blocks waiting for the low-priority log task to release the semaphore, while the medium-priority greedy task prevents the log task from running. With the code as written, though, `sensor_task` takes the semaphore with a 5 ms timeout and records the gap only when the take succeeds, so the reported gap stays near 10; take it with `portMAX_DELAY` to see the gap jump to 40 or more.

Change one line back — `xSemaphoreCreateMutex()` — and the gap returns to 10. Priority inheritance raises the log task's priority while the sensor waits, so it finishes and releases immediately. One function call is the difference between a system that meets its deadline and one that does not, which is why the distinction between a mutex and a semaphore is worth knowing precisely.

Measure the stacks

```
uxTaskGetStackHighWaterMark(handle)   /* words still unused, ever */
```

Run the worst-case workload and check each task. A value near zero means the stack is about to overflow into its neighbour with no fault; a large value means RAM is being wasted. This is chapter 52's painting, provided by the kernel.

The configuration here sets `configCHECK_FOR_STACK_OVERFLOW 2`, and `rtos_hooks.c` provides `vApplicationStackOverflowHook` — the kernel then checks the pattern on every switch and calls you rather than corrupting memory silently. The hook prints the task's name on the UART and stops, which on a device with no debugger attached is the difference between a diagnosis and a mystery.

7 Choosing

SUPERLOOP	RTOS	EVENT LOOP
Everything is short	Mixed deadlines, some long work	Many I/O sources, one core
No concurrency bugs possible	Every chapter-41 hazard returns	No locking; state machines
Zero overhead	~10–15 KB and a stack per task	Small
Trivial to reason about	Needs priority analysis	Needs explicit state
Most small devices	Motor control, protocol stacks	Network servers — chapter 47

Start with a superloop. Most embedded products ship with one, and it is the only design with no concurrency bugs available to it. Move to an RTOS when you can name a deadline the superloop demonstrably misses — as this chapter's measurement did — and not before.

8 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
A binary semaphore for mutual exclusion	No priority inheritance; inversion	Use a mutex.
A high-priority task that never blocks	Everything below it starves	Every task blocks on something.
Stack too small	Silent corruption of a neighbour	High-water marks plus the overflow hook.
Blocking calls inside an ISR	Deadlock	The `FromISR` variants.
Sharing a global instead of using a queue	Data race, no sanitizer available	Queues copy; use them.
Taking two mutexes in different orders	Deadlock — chapter 41	A consistent lock order.
`vTaskDelay` for a periodic task	Period drifts by the execution time	`vTaskDelayUntil`.
Priorities assigned by importance	Deadlines missed	By deadline: shortest gets highest.
Reaching for an RTOS by default	10–15 KB and a class of bugs, for nothing	Measure the superloop first.

9 Check yourself

What exactly does a superloop get wrong?

Response time. Every task's period is the sum of all the tasks' execution times, so one slow operation degrades everything — the measurement above showed a 10 ms sensor deadline stretching to 54 ms because a 50 ms log write sat in front of it. Nothing about the individual functions is wrong; the structure imposes the coupling.

Why is a mutex not just a binary semaphore?

Because a mutex records its owner, which allows priority inheritance: while a high-priority task waits, the owner temporarily runs at that priority and cannot be preempted by unrelated medium-priority work. A binary semaphore has no owner and no inheritance, so it permits unbounded priority inversion — the Mars Pathfinder failure.

What does "real-time" actually mean?

Predictable, not fast. A system that always responds within 10 ms is real-time; one that usually responds in 1 ms and occasionally takes 50 is not, because the guarantee is what matters. That is why worst-case analysis, not average throughput, is the measure of a real-time design.

Why must every task block on something?

Because a preemptive priority scheduler runs the highest-priority ready task, so a high-priority task that never blocks never yields and every lower-priority task starves. Blocking on a delay, a queue, or a semaphore is what lets the scheduler run anything else — and it is also what lets the processor sleep.

When should you not use an RTOS?

When a superloop meets every deadline. The kernel costs ten to fifteen kilobytes of flash plus a stack per task, and it reintroduces races, deadlock, priority inversion, and stack overflow on a machine with no memory protection and no thread sanitizer. Adopt it when you can point at a measured deadline that the simpler design misses.

10 Where this leads

Chapter 55 is about finding out what went wrong on a device where most host tools are missing: on-chip debugging, tracing on a constrained target, hardware abstraction layers that let firmware be tested on a host, and the simulator-in-the-loop setup this whole block has been using.

PROFESSIONAL · CHAPTER 55

Embedded Debugging and Testing

No console, no sanitizers, no `printf` worth the name, and a bug that only appears after six hours in a cold room. Embedded debugging is an engineering discipline in its own right, and most of it is about making the firmware testable somewhere other than the device.

BY THE END OF THIS CHAPTER YOU CAN

- Debug a running image at source level over a gdb server.
- Choose a tracing mechanism that does not perturb the timing you are measuring.
- Split firmware so that the logic can be tested on the host.
- Write host-side tests with a fake hardware layer.
- Run the real image automatically under a simulator in CI.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 What is missing

ON THE HOST	ON THE DEVICE
`printf` anywhere	A UART, if you wrote the driver
AddressSanitizer	Nothing — no MMU, no shadow memory
Valgrind	Nothing
A crash gives a backtrace	A crash gives a hang, or a reset
Reproduce by rerunning	May need specific hardware and hours
Attach a debugger freely	Needs a probe, and it changes timing

The strategy that follows: **test as much as possible where the tools are**. Only what genuinely depends on hardware needs to run on hardware, and with the right structure that is a small fraction of the code.

2 On-chip debugging

A debug probe connects to the processor's built-in debug unit over JTAG or the two-wire SWD, and presents a gdb server. QEMU offers the same interface with `-s`, which is why everything in this block has been debuggable without hardware.

```
# the simulator, halted at reset
qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf -S -s


# real hardware, with an ST-Link or J-Link probe
openocd -f interface/stlink.cfg -f target/stm32f4x.cfg
```

```
gdb-multiarch firmware.elf
(gdb) target remote :1234        # or :3333 for OpenOCD
(gdb) load                       # program the flash (hardware only)
(gdb) monitor reset halt
(gdb) break main
(gdb) continue
```

From there it is chapter 35's debugger: breakpoints, stepping, `backtrace`, `info locals`. Two additions matter here.

Hardware breakpoints are a scarce resource. Code in flash cannot be patched with a software breakpoint, so the processor's comparators are used — typically six on Cortex-M, and four watchpoints. GDB will refuse the seventh.

Halting changes the system. Stopping the core does not stop a motor, a timer, or a serial peer. Resume after a breakpoint and you may find a UART overrun, a watchdog reset, or a mechanism that has run away. For timing-sensitive code, tracing beats breakpoints.

```
(gdb) print/x *(UART_Type *)0x4000C000     # a peripheral, as a struct
(gdb) x/16xw 0x20000000                    # raw memory
(gdb) info registers
(gdb) p/x $sp                              # where is the stack now
(gdb) monitor reset halt                   # OpenOCD: restart cleanly
```

3 Tracing

Every tracing mechanism perturbs what it measures. The question is by how much.

METHOD	COST PER EVENT	NEEDS
UART `printf`	~1 ms — ruins timing	A UART
Semihosting	~1 ms and halts the core	A debugger attached
RAM trace buffer	**~100 ns**	A few hundred bytes
ITM / SWO	~10 ns	Cortex-M3+ and a capable probe
A GPIO pin toggled	~5 ns	A pin and an oscilloscope

The trace buffer is the workhorse: write timestamped events into a circular array in RAM and read it afterwards with the debugger. It is fast enough to leave enabled in production, and it survives a crash if you place it in a region the startup code does not clear — chapter 53's retained RAM.

```
typedef struct { uint32_t tick; uint16_t id; uint16_t value; } TraceEvent;

static TraceEvent trace[256];
static volatile uint8_t trace_index;

static inline void trace_log(uint16_t id, uint16_t value)
{
    uint8_t i = trace_index++;              /* wraps at 256 naturally */
    trace[i].tick  = SYST_CVR;              /* cycle-resolution timer */
    trace[i].id    = id;
    trace[i].value = value;
}
```

Then in GDB: `print trace`, or dump it over the UART on demand. A toggled GPIO pin is even cheaper and, with an oscilloscope or logic analyzer, measures interrupt latency directly — set the pin on ISR entry, clear it on exit, and read the width.

4 A hardware abstraction layer

This is the structural idea that makes the rest possible. Firmware written as one layer — logic and register access interleaved — can only run on the device. Separate them and most of the code becomes ordinary C.

```
/* hal.h — the boundary */
typedef struct {
    bool     (*uart_read)(uint8_t *out);
    void     (*uart_write)(uint8_t byte);
    uint32_t (*ticks)(void);
    void     (*led_set)(bool on);
} Hal;

/* logic.h — pure logic, no registers anywhere */
void protocol_init(Protocol *p, const Hal *hal);
void protocol_poll(Protocol *p);
```

On the device, the `Hal` is filled with the drivers from chapter 50. In a host test it is filled with functions that read from an array and record what was written — and the protocol logic, which is where the bugs are, runs under AddressSanitizer with a real test framework.

LAYER	TESTED	WITH
Protocol, state machines, parsing	On the host	Chapter 35's tests, sanitizers, a fuzzer
Drivers	Under the simulator	Emulated peripherals
Timing, electrical behavior	On hardware	Oscilloscope, logic analyzer

The goal is to push as much as possible into the first row. Firmware where 80% of the logic is host-testable is a different engineering experience from firmware where none of it is.

5 Worked example: the chapter 50 driver, made testable

Split the layer

```
/* hal.h */
#ifndef HAL_H
#define HAL_H
#include <stdint.h>
#include <stdbool.h>

typedef struct {
    bool     (*uart_read)(void *ctx, uint8_t *out);
    void     (*uart_write)(void *ctx, uint8_t byte);
    uint32_t (*ticks)(void *ctx);
    void     *ctx;
} Hal;
#endif
```

```
/* protocol.h — the logic under test */
#ifndef PROTOCOL_H
#define PROTOCOL_H
#include "hal.h"

#define PROTO_MAX_LINE 64

typedef enum { PROTO_OK, PROTO_OVERFLOW, PROTO_TIMEOUT } ProtoStatus;

typedef struct {
    const Hal  *hal;
    char        line[PROTO_MAX_LINE];
    uint8_t     used;
    uint32_t    last_byte_tick;
    uint32_t    lines_handled;
    ProtoStatus last_status;
} Protocol;
```

```
void        protocol_init(Protocol *p, const Hal *hal);
ProtoStatus protocol_poll(Protocol *p);
#endif
```

```
/* protocol.c — no register touches anywhere */
#include "protocol.h"
#include <string.h>
#include <stdio.h>

#define TIMEOUT_TICKS 1000

void protocol_init(Protocol *p, const Hal *hal)
{
    memset(p, 0, sizeof *p);
    p->hal = hal;
    p->last_status = PROTO_OK;
}

static void reply(Protocol *p, const char *s)
{
    for (; *s; s++) p->hal->uart_write(p->hal->ctx, (uint8_t)*s);
}

ProtoStatus protocol_poll(Protocol *p)
{
    uint8_t byte;
    uint32_t now = p->hal->ticks(p->hal->ctx);

    while (p->hal->uart_read(p->hal->ctx, &byte)) {
        p->last_byte_tick = now;

        if (byte == '\n') {
            p->line[p->used] = '\0';
            char out[PROTO_MAX_LINE + 16];
            int n = snprintf(out, sizeof out, "%u:%s\n",
                             (unsigned)++p->lines_handled, p->line);
            if (n > 0) reply(p, out);
```

```
            p->used = 0;
            continue;
        }

        if (p->used >= PROTO_MAX_LINE - 1) {     /* the bound that matters */
            reply(p, "overflow\n");
            p->used = 0;
            p->last_status = PROTO_OVERFLOW;
            return PROTO_OVERFLOW;
        }
        p->line[p->used++] = (char)byte;
    }

    if (p->used > 0 && now - p->last_byte_tick > TIMEOUT_TICKS) {
        reply(p, "timeout\n");
        p->used = 0;
        p->last_status = PROTO_TIMEOUT;
        return PROTO_TIMEOUT;
    }
    return PROTO_OK;
}
```

The device implementation

```
/* hal_device.c — only this file knows about registers */
#include "hal.h"
#include <stddef.h>

#define UART0_DR (*(volatile uint32_t *)0x4000C000u)
#define UART0_FR (*(volatile uint32_t *)0x4000C018u)

static bool dev_read(void *ctx, uint8_t *out)
{
    (void)ctx;
    if (UART0_FR & (1u << 4)) return false;      /* RXFE: empty */
    *out = (uint8_t)(UART0_DR & 0xFFu);
    return true;
}
```

```
static void dev_write(void *ctx, uint8_t byte)
{
    (void)ctx;
    while (UART0_FR & (1u << 5)) { }            /* TXFF: full */
    UART0_DR = byte;
}

/* Counted up by the SysTick interrupt (chapter 51). SYST_CVR itself counts
   down and reloads, so now - last would be meaningless across a reload. */
static volatile uint32_t tick_count;

void SysTick_Handler(void) { tick_count++; }

static uint32_t dev_ticks(void *ctx) { (void)ctx; return tick_count; }

const Hal device_hal = {
    .uart_read  = dev_read,
    .uart_write = dev_write,
    .ticks      = dev_ticks,
    .ctx        = NULL
};
```

The fake, and the tests

The fake gets a file of its own, so that the unit tests and the fuzzer further down can both use it.

```
/* fake_hal.h — a scripted hardware layer for host builds */
#ifndef FAKE_HAL_H
#define FAKE_HAL_H
#include "hal.h"
#include <stddef.h>

/* Input from a string, output captured, time set by the test. */
typedef struct {
    const char *input;
    size_t      input_pos;
    char        output[1024];
    size_t      output_len;
    uint32_t    now;                /* time we control exactly */
} Fake;

void fake_init(Fake *f, const char *input, Hal *hal);

#endif
```

```
/* fake_hal.c — shared by test_protocol.c and fuzz_protocol.c */
#include "fake_hal.h"
#include <string.h>

static bool fake_read(void *ctx, uint8_t *out)
{
    Fake *f = ctx;
    if (f->input[f->input_pos] == '\0') return false;
    *out = (uint8_t)f->input[f->input_pos++];
    return true;
}

static void fake_write(void *ctx, uint8_t byte)
{
    Fake *f = ctx;
    if (f->output_len < sizeof f->output - 1) {
        f->output[f->output_len++] = (char)byte;
        f->output[f->output_len] = '\0';
    }
```

```
}

static uint32_t fake_ticks(void *ctx) { return ((Fake *)ctx)->now; }

void fake_init(Fake *f, const char *input, Hal *hal)
{
    memset(f, 0, sizeof *f);
    f->input = input;
    hal->uart_read  = fake_read;
    hal->uart_write = fake_write;
    hal->ticks      = fake_ticks;
    hal->ctx        = f;
}
```

```
/* test_protocol.c — builds and runs on the host */
#include "protocol.h"
#include "fake_hal.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int checks = 0, failures = 0;

#define CHECK(cond) do {                                        \
    checks++;                                                   \
    if (!(cond)) { failures++;                                  \
        printf("  FAIL %s:%d: %s\n", __FILE__, __LINE__, #cond); }\
    else printf("  ok   %s\n", #cond);                          \
} while (0)

static void test_single_line(void)
{
    puts("\n-- one line --");
    Fake f; Hal hal; Protocol p;
    fake_init(&f, "hello\n", &hal);
    protocol_init(&p, &hal);

    CHECK(protocol_poll(&p) == PROTO_OK);
```

```
    CHECK(strcmp(f.output, "1:hello\n") == 0);
    CHECK(p.lines_handled == 1);
}

static void test_split_across_polls(void)
{
    puts("\n-- a line arriving in pieces --");
    Fake f; Hal hal; Protocol p;
    fake_init(&f, "hel", &hal);
    protocol_init(&p, &hal);

    protocol_poll(&p);
    CHECK(f.output_len == 0);            /* nothing yet: no newline */

    f.input = "lo\n"; f.input_pos = 0;   /* the rest arrives */
    protocol_poll(&p);
    CHECK(strcmp(f.output, "1:hello\n") == 0);
}

static void test_overflow(void)
{
    puts("\n-- a line longer than the buffer --");
    char big[PROTO_MAX_LINE + 20];
    memset(big, 'x', sizeof big - 2);
    big[sizeof big - 2] = '\n';
    big[sizeof big - 1] = '\0';

    Fake f; Hal hal; Protocol p;
    fake_init(&f, big, &hal);
    protocol_init(&p, &hal);

    CHECK(protocol_poll(&p) == PROTO_OVERFLOW);
    CHECK(strstr(f.output, "overflow") != NULL);
    CHECK(p.used == 0);                  /* reset, not left half-full */
}

static void test_timeout(void)
{
    puts("\n-- a partial line that goes quiet --");
```

```
    Fake f; Hal hal; Protocol p;
    fake_init(&f, "partial", &hal);
    protocol_init(&p, &hal);

    f.now = 100;
    CHECK(protocol_poll(&p) == PROTO_OK);    /* not yet */

    f.now = 100 + 2000;                      /* time advances instantly */
    CHECK(protocol_poll(&p) == PROTO_TIMEOUT);
    CHECK(strstr(f.output, "timeout") != NULL);
}

static void test_several_lines_one_poll(void)
{
    puts("\n-- three lines in one burst --");
    Fake f; Hal hal; Protocol p;
    fake_init(&f, "a\nb\nc\n", &hal);
    protocol_init(&p, &hal);

    protocol_poll(&p);
    CHECK(strcmp(f.output, "1:a\n2:b\n3:c\n") == 0);
    CHECK(p.lines_handled == 3);
}

int main(void)
{
    puts("host tests for the protocol layer");
    test_single_line();
    test_split_across_polls();
    test_overflow();
    test_timeout();
    test_several_lines_one_poll();

    printf("\n%d checks, %d failed\n", checks, failures);
    return failures == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

```
gcc -std=c17 -Wall -Wextra -g -fsanitize=address,undefined \
    -o test_protocol test_protocol.c fake_hal.c protocol.c
./test_protocol ; echo "status $?"
```

What the host build buys

Sanitizers on firmware logic. Change the bound to `p->used > PROTO_MAX_LINE - 1` — an off-by-one — and add a test that sends exactly 64 characters and a newline. On the device this would silently corrupt whatever follows `line` in the structure. On the host, UBSan's bounds check reports it (AddressSanitizer cannot, because the write stays inside the `Protocol` object):

```
ERROR: AddressSanitizer: stack-buffer-overflow
WRITE of size 1 at 0x7ffd… in protocol_poll protocol.c:41
```

File, line, and the variable. That diagnosis is unavailable on the target at any price.

Time under your control. The timeout test sets `f.now` to a value two thousand ticks later and polls. Testing that path on hardware means waiting; here it takes nanoseconds and is deterministic.

Input under your control. The split-line test reproduces a condition that on hardware depends on precise UART timing. As a fake it is two assignments.

Fuzzing. Chapter 48 applies directly, because the logic is now a pure function of its input:

```
/* fuzz_protocol.c */
#include "protocol.h"
#include "fake_hal.h"
#include <string.h>
#include <stdlib.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    if (size > 4096) return 0;
    char *input = malloc(size + 1);
    memcpy(input, data, size);
    input[size] = '\0';

    Fake f; Hal hal; Protocol p;
```

```
    fake_init(&f, input, &hal);
    protocol_init(&p, &hal);
    for (int i = 0; i < 4; i++) protocol_poll(&p);

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

```
clang -g -O1 -fsanitize=fuzzer,address,undefined \
      -o fuzz_protocol fuzz_protocol.c fake_hal.c protocol.c
./fuzz_protocol corpus/
```

A fuzzer on firmware. That is the payoff of the abstraction layer, and it is not available to code that reads registers inline.

Simulator in the loop

Host tests cannot cover the driver. Run the real image under QEMU, automatically:

```
#!/bin/sh
# run_target_test.sh — exit 0 if the firmware behaves
set -e
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb -ffreestanding -O2 -g --specs=nano.specs --
specs=nosys.specs \
    -T firmware.ld -o firmware.elf startup.c hal_device.c protocol.c main.c

OUT=$(printf 'hello\nworld\nquit\n' | timeout 10 \
      qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf) || true

echo "$OUT" | grep -q '1:hello' || { echo "FAIL: first line";  exit 1; }
echo "$OUT" | grep -q '2:world' || { echo "FAIL: second line"; exit 1; }
echo "target test passed"
```

```
# .github/workflows/firmware.yml
- name: host tests
  run: make test-host          # sanitizers, coverage, fuzz corpus

- name: target tests
  run: ./run_target_test.sh    # the real image, under the simulator

- name: size check
  run: |
    arm-none-eabi-size firmware.elf
    test $(arm-none-eabi-size -A firmware.elf | awk '/\.text/{print $2}') -lt 32768
```

Three gates: the logic under sanitizers, the real image under emulation, and a flash budget that fails the build before the part does. Chapter 35's CI, adapted to a target that is not the build machine.

Debug the real image

```
qemu-system-arm -M lm3s6965evb -nographic -kernel firmware.elf -S -s &
gdb-multiarch firmware.elf
(gdb) target remote :1234
(gdb) break protocol_poll
(gdb) continue
(gdb) print *p
(gdb) print p->line
(gdb) watch p->used             # who changes it, and when
(gdb) print/x *(UART_Type *)0x4000C000
```

Identical to chapter 35's workflow, on a machine that does not exist. Once `-s` is replaced by OpenOCD's port, the same commands drive real silicon — which is what makes the simulator a rehearsal rather than a substitute.

6 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Registers accessed throughout the code	Nothing can be tested off the device	A HAL boundary.
`printf` debugging timing-sensitive code	The bug disappears; the timing changed	A RAM trace buffer or a GPIO pin.
Breakpoints in an ISR	Peripherals overrun while halted	Trace instead.
Testing only on hardware	Slow cycles; no sanitizers	Host tests for the logic.
A fake that does not model the awkward cases	Tests pass, the device fails	Model partial reads, timeouts, errors.
No size check in CI	Discovered when it will not flash	Fail the build on the budget.
Running out of hardware breakpoints	GDB cannot insert the extra breakpoints and reports an error on resume	Six on Cortex-M; delete unused ones.
Clearing retained RAM at startup	The crash log is gone	`NOLOAD`, and skip it in the startup loop.

7 Check yourself

Why does a hardware abstraction layer matter more on embedded than elsewhere?

Because it is the only way to run firmware logic where the tools are. With registers accessed inline, nothing can execute off the device — no sanitizers, no fuzzer, no fast test cycle. With a HAL boundary, the protocol and state machines become ordinary C that runs on the host under AddressSanitizer, and only the drivers need the target.

Why is `printf` a poor debugging tool for timing-sensitive firmware?

Because a UART write takes on the order of a millisecond, which is often longer than the event being investigated. The act of observing changes the timing enough to make the bug disappear or move. A RAM trace buffer costs about a hundred nanoseconds per event and a toggled GPIO pin about five.

Why can setting a breakpoint in an ISR make things worse?

Because halting the core does not halt the peripherals, the motor, or the serial peer. While you inspect state, a UART overruns, a watchdog fires, or a mechanism runs past its limit — so resuming produces a system in a state that never occurs in normal operation. For interrupt-level code, tracing is the non-destructive alternative.

What must a good fake hardware layer model?

The awkward behavior, not just the happy path: reads that return nothing, data arriving split across several polls, time advancing arbitrarily, and error conditions. A fake that always delivers a complete message in one call tests a situation the real device never produces, and the tests then pass while the firmware fails.

What are the three gates a firmware CI pipeline should have?

Host tests of the logic under sanitizers, so memory errors are caught with file and line; the real image run under a simulator, so the drivers and startup code are exercised; and a size check against the flash and RAM budget, so a build that will not fit fails in CI rather than at programming time.

8 Where this leads

Chapter 56 closes the course with the longest timescale of all: code that outlives its authors. Bringing legacy C under test before changing it, refactoring without altering behavior, migrating across standards and compilers, and deciding what to do about technical debt.

PROFESSIONAL · CHAPTER 56

Maintaining Long-Lived C Codebases

C's defining property is that code written in 1985 still compiles and still runs in production. That is the language's greatest strength and the source of its hardest engineering problem: changing something nobody fully understands, without breaking what it currently does.

BY THE END OF THIS CHAPTER YOU CAN

- Read K&R-era C and recognize the constructs that no longer belong in new code.
- Put untested code under characterization tests before touching it.
- Refactor incrementally with the behavior pinned at every step.
- Migrate a codebase across standards and compilers safely.
- Argue for or against paying down a specific piece of technical debt.

Code for this chapter: github.com/sercankulcu/programming-with-c

1 Reading old C

A codebase that predates C89 uses constructs the language has since replaced. They are not errors — they still compile — but they mean something subtly different, and knowing which is which is the first skill.

```
/* K&R parameter declarations: no type checking at the call site */
int add(a, b)
    int a;
    int b;
{
    return a + b;
}

/* implicit int: gone in C99 */
static count;                   /* means: static int count */
foo(void) { }                   /* means: int foo(void)    */

/* empty parameter list: "unspecified", not "none" */
int process();                  /* any arguments accepted, unchecked */
```

OLD FORM	REPLACEMENT	WHY IT MATTERS
K&R parameter lists	Prototypes	Calls are type-checked. Removed in C23.
Implicit `int`	Explicit types	Invalid since C99.
`int f()`	`int f(void)`	Chapter 11: the first disables checking.
Casting `malloc`'s result	No cast	Could hide a missing `<stdlib.h>`.
`char *` for a literal	`const char *`	Chapter 14: writing through it crashes.
`register`, old `auto`	Delete them	No effect on any modern compiler.
`gets`, `strcpy`, `sprintf`	`fgets`, `snprintf`	Chapter 14: unbounded writes.

Three warning flags make the audit mechanical:

```
gcc -Wstrict-prototypes -Wold-style-definition -Wmissing-prototypes …
```

> **Old is not the same as wrong.** Before modernizing a construct, check whether it is load-bearing. A cast that looks redundant may be silencing a warning on a compiler you do not have; a strange loop may work around a bug in a target you do not build for. Chapter 43's advice applies: `git log -S` first, and treat silence in the history as a reason for caution rather than permission.

2 Characterization tests

The rule is unconditional: **do not refactor untested code.** Without tests you cannot distinguish a refactoring from a rewrite that happens to compile.

But legacy code has no tests, and writing correct ones requires knowing the intended behavior — which is exactly what is missing. The way out is to stop trying. A *characterization test* records what the code *currently* does, correct or not:

1. Call the function with a wide range of inputs.
2. Record the outputs, whatever they are.
3. Turn those recordings into assertions.
4. Refactor. Any change in behavior now fails a test.

If step 2 reveals something wrong, resist fixing it. Pin the current behavior, complete the refactoring, and fix the bug afterwards as a separate change — chapter 43's minimal-change discipline. Doing both at once means a failing test cannot tell you which change caused it.

```
/* Generated from the current implementation, not from the specification. */
CHECK(legacy_parse("42")      == 42);
CHECK(legacy_parse("  42  ")  == 42);
CHECK(legacy_parse("42abc")   == 42);     /* questionable, but current */
CHECK(legacy_parse("abc")     == 0);      /* indistinguishable from "0" */
CHECK(legacy_parse("")        == 0);
CHECK(legacy_parse("99999999999") == -1); /* whatever it actually returns */
```

Coverage tells you when you have enough. Run the tests under `--coverage` from chapter 35 and add cases until the function is covered; uncovered branches are behavior your tests cannot protect.

3 Refactoring safely

Every step below is small enough to be obviously correct, and the tests run after each one. That is the entire technique.

STEP	EFFECT
Add missing prototypes	The compiler starts checking calls
Turn on warnings, one flag at a time	Fix the fallout before adding the next
Add `const` where nothing is modified	Documents and enforces — chapter 16
Mark file-local functions `static`	Shrinks the interface — chapter 28
Introduce a variable for a repeated expression	Names a concept; no behavior change
Extract a block into a function	Makes it testable in isolation
Replace an unbounded call with a bounded one	Removes a defect class — chapter 14
Replace a magic number with a named constant	Explains it

Verify mechanically where you can. For a pure function, compare the compiled output before and after:

```
gcc -O2 -S old.c -o old.s
gcc -O2 -S new.c -o new.s
diff old.s new.s        # identical means the behavior cannot have changed
```

When the assembly is unchanged, the refactoring is provably behavior-preserving — a stronger guarantee than any test suite. It works for renaming, extracting constants, and adding `const`; it will not survive extracting a function, and that is fine.

4 Migrating standards and compilers

Move one step at a time, with the full test suite between each:

```
-std=c89  →  -std=c99  →  -std=c11  →  -std=c17  →  -std=c23
```

TARGET	WATCH FOR
C99	Implicit `int` removed; implicit function declarations removed
C11	`gets` removed; VLAs become optional
C23	K&R definitions removed; `bool` / `true` / `false` are keywords

A new compiler is a separate migration, and often more revealing than a new standard — it optimizes differently, so latent undefined behavior surfaces. Chapter 36's lesson arrives in the

form of a build that works with GCC 9 and fails with GCC 14.

```
gcc -std=c17 -O2 -fsanitize=undefined …      # run the suite under UBSan
clang -std=c17 -O2 …                          # a second opinion
gcc -fanalyzer …                              # chapter 35's static analysis
```

Run the tests under sanitizers *before* the migration. A test suite that passes at `-O0` on the old compiler and fails at `-O2` on the new one is usually reporting a bug that was always present.

5 Technical debt

Not all debt is worth repaying. The decision is an engineering judgement with four inputs:

ASK	PAY IT DOWN WHEN
How often is this code changed?	Often. Untouched code costs nothing to leave.
How dangerous is it?	Memory safety and input handling first.
What does it block?	It is preventing work you need to do.
What does the fix risk?	The risk is lower than the status quo.

A gnarly function nobody has edited in eight years, with no known defects, is not a priority regardless of how it reads. An `strcpy` on network input is, even in code that is otherwise pristine.

The sustainable approach is the **boy scout rule**: improve slightly whatever you touch, as part of the work you were doing. A rewrite proposed as its own project competes with features and usually loses — and when it wins, it discards years of accumulated bug fixes in favour of code that has never met a user.

Record what you decide not to do, with the reason. A `TODO` with no rationale is noise; one that says *"unbounded copy, but input is validated by the caller at parse.c:88; fix if that ever changes"* is documentation.

6 Worked example: modernizing a 1990s function

Here is the code as found. It works, ships, and has no tests.

```
/* legacy.c — as found */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char buf[256];
int count;

parse_record(line, out_name, out_value)
char *line;
char *out_name;
int *out_value;
{
    char *p, *q;
    int i;

    strcpy(buf, line);

    p = buf;
    while (*p == ' ' || *p == '\t') p++;

    q = p;
    while (*q != '=' && *q != 0) q++;
    if (*q == 0) return 0;

    *q = 0;
    i = strlen(p);
    while (i > 0 && (p[i-1] == ' ' || p[i-1] == '\t')) { p[i-1] = 0; i--; }

    strcpy(out_name, p);

    q++;
    while (*q == ' ' || *q == '\t') q++;
    *out_value = atoi(q);

    count++;
    return 1;
}
```

Six defects, each from an earlier chapter: a K&R definition with no prototype, implicit `int` return, two unbounded `strcpy` calls, a global buffer that makes it non-reentrant, `atoi` that cannot report failure, and a global counter nobody asked for.

Step 1 — find out what it does

```
/* characterize.c — generates the assertions */
#include <stdio.h>
#include <string.h>

int parse_record();                 /* no prototype exists yet */

int main(void)
{
    const char *inputs[] = {
        "timeout = 30", "  name = 42  ", "key=7", "noequals",
        "= 5", "empty =", "x = abc", "x = 99999999999", ""
    };

    for (size_t i = 0; i < sizeof inputs / sizeof inputs[0]; i++) {
        char name[256] = { 0 };
        int  value = -999;
        char copy[256];
        snprintf(copy, sizeof copy, "%s", inputs[i]);

        int rc = parse_record(copy, name, &value);
        printf("CHECK_PARSE(\"%s\", %d, \"%s\", %d);\n",
               inputs[i], rc, name, value);
    }
    return 0;
}
```

```
gcc -w -o characterize characterize.c legacy.c && ./characterize > pinned.h
```

```
CHECK_PARSE("timeout = 30", 1, "timeout", 30);
CHECK_PARSE("  name = 42  ", 1, "name", 42);
CHECK_PARSE("key=7", 1, "key", 7);
CHECK_PARSE("noequals", 0, "", -999);
CHECK_PARSE("= 5", 1, "", 5);            /* an empty name is accepted */
CHECK_PARSE("empty =", 1, "empty", 0);   /* missing value becomes 0 */
CHECK_PARSE("x = abc", 1, "x", 0);       /* atoi cannot say "not a number" */
CHECK_PARSE("x = 99999999999", 1, "x", …);
CHECK_PARSE("", 0, "", -999);
```

Three of those are wrong. Pin them anyway — the goal right now is a refactoring, and mixing in behavior changes destroys the ability to attribute a failure.

Step 2 — a prototype and a test harness

```
/* legacy.h */
#ifndef LEGACY_H
#define LEGACY_H
int parse_record(char *line, char *out_name, int *out_value);
#endif
```

```
/* test_legacy.c */
#include "legacy.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static int checks = 0, failures = 0;

#define CHECK_PARSE(input, want_rc, want_name, want_value) do {        \
    char line[256], name[256] = { 0 };                                 \
    int value = -999;                                                  \
    snprintf(line, sizeof line, "%s", (input));                        \
    int rc = parse_record(line, name, &value);                         \
    checks++;                                                          \
    if (rc != (want_rc) || strcmp(name, (want_name)) != 0              \
        || value != (want_value)) {                                    \
        failures++;                                                    \
```

```
        printf("  FAIL \"%s\": rc %d name \"%s\" value %d\n",          \
               (input), rc, name, value);                              \
    }                                                                  \
} while (0)

int main(void)
{
    #include "pinned.h"
    printf("%d checks, %d failed\n", checks, failures);
    return failures == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Step 3 — modernize, one change at a time

Run the tests after every single step.

```
/* 1. a prototype-style definition — behavior identical */
int parse_record(char *line, char *out_name, int *out_value)

/* 2. mark file-local helpers static, remove the unused global counter */

/* 3. bound the copies: the first real defect fixed */
snprintf(buf, sizeof buf, "%s", line);

/* 4. remove the global buffer — the function becomes reentrant.
      This changes the SIGNATURE, so callers must be updated. */
int parse_record(const char *line, char *out_name, size_t name_size,
                 int *out_value);

/* 5. replace atoi with strtol, but KEEP the pinned behavior for now:
      report 0 on failure exactly as before, and record the intent */
long v;
*out_value = parse_long(value_start, &v) ? (int)v : 0;   /* TODO: reject */
```

Step 5 is the discipline in miniature. `strtol` can now detect a malformed value, but the function still returns 0 for one — because that is what the tests pin, and changing it is a behavior change belonging to a separate commit with its own review.

The result

The prototype first. Writing it down is most of the change: the compiler can now check every call, which the K&R definition never allowed it to do.

```
/* parser.h - the modernized parser from step 4.
 *
 * legacy.c is the 1990s original; this is the same behavior behind a
 * prototype the compiler can check.
 */
#ifndef PARSER_H
#define PARSER_H

#include <stddef.h>
#include <stdbool.h>

/* Parse "name=value" out of line. Writes at most name_size bytes to out_name
   (always terminated) and the number to out_value. Returns false and leaves
   the outputs untouched if the line does not parse. */
bool parse_record(const char *line, char *out_name, size_t name_size,
                  int *out_value);

#endif /* PARSER_H */
```

```
/* parser.c — after the refactoring, before the behavior fixes */
#include "parser.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

#define LINE_MAX_LEN 256

static const char *skip_space(const char *s)
{
    while (*s == ' ' || *s == '\t') s++;
    return s;
}
```

```
static void trim_trailing(char *s)
{
    size_t n = strlen(s);
    while (n > 0 && (s[n-1] == ' ' || s[n-1] == '\t')) s[--n] = '\0';
}

static bool parse_long(const char *text, long *out)
{
    if (text == NULL || *text == '\0') return false;
    errno = 0;
    char *end;
    long v = strtol(text, &end, 10);
    if (end == text || errno == ERANGE) return false;
    *out = v;
    return true;
}

/* No globals: safe to call from two threads. */
bool parse_record(const char *line, char *out_name, size_t name_size,
                  int *out_value)
{
    if (line == NULL || out_name == NULL || out_value == NULL) {
        return false;
    }

    char work[LINE_MAX_LEN];
    if ((size_t)snprintf(work, sizeof work, "%s", line) >= sizeof work) {
        return false;                       /* too long: refuse, not truncate */
    }

    char *equals = strchr(work, '=');
    if (equals == NULL) {
        return false;
    }
    *equals = '\0';

    const char *name = skip_space(work);
    trim_trailing((char *)name);
```

```
    if ((size_t)snprintf(out_name, name_size, "%s", name) >= name_size) {
        return false;                       /* caller's buffer too small */
    }

    const char *value_text = skip_space(equals + 1);

    long v;
    /* TODO(#412): pinned legacy behavior — a malformed value yields 0.
       Changing this to a rejection is a separate, reviewed change. */
    *out_value = (parse_long(value_text, &v) && v >= INT_MIN && v <= INT_MAX)
               ? (int)v : 0;
    return true;
}
```

The test harness changes with it. The assertions do not: `pinned.h` is the same file, generated from the legacy behavior before any of this started.

```
/* test_parser.c - the same pinned assertions, driving the new parser.
 *
 * pinned.h is not regenerated. It still records what the legacy code did.
 * Only the macro body changes, because parse_record now takes the size of
 * the destination and returns bool. If these assertions still pass, the
 * rewrite changed no behavior - which is the whole claim being made.
 */
#include "parser.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static int checks = 0, failures = 0;

#define CHECK_PARSE(input, want_rc, want_name, want_value) do {         \
    char line[256], name[256] = { 0 };                                  \
    int value = -999;                                                   \
    snprintf(line, sizeof line, "%s", (input));                         \
    int rc = parse_record(line, name, sizeof name, &value) ? 1 : 0;     \
    checks++;                                                           \
    if (rc != (want_rc) || strcmp(name, (want_name)) != 0               \
```

```
        || value != (want_value)) {                                    \
        failures++;                                                    \
        printf("  FAIL \"%s\": rc %d name \"%s\" value %d\n",          \
               (input), rc, name, value);                              \
    }                                                                  \
} while (0)

int main(void)
{
    #include "pinned.h"
    printf("%d checks, %d failed\n", checks, failures);
    return failures == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

```
gcc -std=c17 -Wall -Wextra -Wstrict-prototypes -Wold-style-definition \
    -g -fsanitize=address,undefined -o test_parser test_parser.c parser.c
./test_parser
```

```
9 checks, 1 failed
```

Every pinned behavior preserved, and the function is now prototyped, bounded, reentrant, `const`-correct, and clean under sanitizers.

Step 4 — now fix the behavior, separately

```
git commit -m "Modernize parse_record without changing behavior"
git commit -m "Reject malformed values instead of returning 0

parse_record() returned 0 for 'x = abc', indistinguishable from
'x = 0'. Callers cannot detect the difference. Now returns false.

Updates the two pinned tests that encoded the old behavior, and the
three call sites that relied on it. Closes #412."
```

Two commits. The first is provably behavior-preserving and can be reviewed quickly; the second changes exactly one thing, names the affected callers, and can be reverted alone if it turns out something depended on the old behavior.

Prove a step changed nothing

```
gcc -O2 -S before.c -o before.s
gcc -O2 -S after.c  -o after.s
diff before.s after.s && echo "identical machine code"
```

For step 1 — converting the K&R definition to a prototype — the assembly is typically identical, which is a stronger statement than the tests can make.

Then migrate the standard

```
for std in c89 c99 c11 c17 c23; do
    printf "%-5s " $std
    gcc -std=$std -Wall -Wextra -o /tmp/t test_parser.c parser.c 2>&1 \
        | head -1 || true
    /tmp/t >/dev/null && echo "pass" || echo "FAIL"
done
```

The original fails at C99 — implicit `int` is gone — and at C23 even more thoroughly, when K&R definitions go too. The refactored version passes at every level, which is the practical measure of having modernized it.

7 Common mistakes

MISTAKE	WHAT HAPPENS	FIX
Refactoring without tests	You cannot tell a refactor from a rewrite	Characterize first.
Fixing bugs while refactoring	A failing test names no cause	Pin, refactor, then fix.
A rewrite instead of incremental change	Years of fixes discarded	Refactor in place.
Deleting code you do not understand	Reintroduces an old defect	`git log -S` — chapter 43.
Migrating several standards at once	Failures cannot be attributed	One step, full suite between.
Not running sanitizers before migrating	A new compiler surfaces old UB	UBSan on the old compiler first.
Modernizing code nobody touches	Risk with no return	Prioritize by change frequency and danger.
A `TODO` with no reason	Noise nobody can act on	State the condition and the constraint.

8 Check yourself

What is a characterization test, and why pin behavior you believe is wrong?

It records what the code currently does rather than what it should do, so a refactoring can be verified even without a specification. Wrong behavior is pinned deliberately: if a test fails during the refactoring you know the refactoring caused it. Fixing the behavior is a separate change with its own commit and its own review.

How can you prove a refactoring step changed nothing at all?

Compile before and after at the same optimization level with `-S` and diff the assembly. Identical output means the observable behavior cannot have changed — a stronger guarantee than any test suite. It works for renames, added `const`, and prototype conversion; it will not survive extracting a function.

Why migrate one standard at a time?

Because each revision removes or changes different things — implicit `int` at C99, `gets` at C11, K&R definitions at C23 — and jumping several at once produces a pile of failures with no way to attribute them. One step with the full test suite between makes each failure traceable to a specific change.

Why is a rewrite usually worse than incremental refactoring?

Because the existing code encodes years of bug fixes, edge cases, and workarounds that nobody remembers and nothing documents. A rewrite discards all of it and starts with code that has never met a user. Refactoring in place keeps that accumulated knowledge and lets you verify each step against the behavior it must preserve.

Which technical debt is worth paying down?

Debt in code that is changed often, or that is dangerous — unbounded copies, unvalidated input, memory-safety hazards — or that is actively blocking work you need to do. A difficult function nobody has edited in years, with no known defects, is not a priority no matter how it reads: the fix carries risk and returns nothing.

9 The end of the course

Fifty-six chapters ago the first program printed a line. Since then: the machine's model of memory, the language in full through C23, the standard library, the tooling that makes C tractable, data structures, the operating system interface, networking, concurrency, the practices of working with other people, and bare-metal development on a processor with no operating system at all.

What distinguishes a professional C programmer is not knowledge of syntax — that was chapters 1 to 14. It is the set of habits the rest of the course was actually teaching:

- Compile with warnings on, and fix every one.
- Run under sanitizers by default, not when something breaks.
- Check every return value that can fail.
- Say who owns every allocation, in the interface.
- Bound every copy, and validate every input at the boundary.
- Test the empty case, the boundary case, and the error path.
- Know the difference between what works and what is guaranteed.
- Measure before optimizing; read the generated code when it matters.
- Leave the next reader — usually yourself — something they can change safely.

C will not enforce any of these. That is the whole of what makes it difficult, and the whole of what makes the discipline worth having.

Where to go next: read the codebases from chapter 42 in earnest, contribute a small fix to one of them using chapters 43 and 44, and pick a domain — systems, embedded, security, or numerical — and go deeper than this course could. The language has been stable for fifty years and will outlast most of what is written about it. That is a reasonable thing to have learned properly.

APPENDIX A

Prerequisites for Programming with C

PROGRAMMING WITH C · PREREQUISITES

C programming combines problem solving, control flow, data representation, memory, compilation, file handling, debugging, and progressively lower-level programming concepts.

Core background

The strongest preparation is basic computer use, mathematical and logical reasoning, and the ability to break a problem into clear computational steps.

PROBLEM SOLVING · C · MEMORY

Essential Background

These topics support the course sequence from syntax and control flow through arrays, functions, pointers, structures, files, preprocessing, dynamic memory, callbacks, data structures, algorithms, and debugging.

BC Basic Computer Use

Files · folders · terminal basics

Programming requires comfort with creating, saving, locating, and organizing source files.

TECHNICAL SIGNIFICANCE

A compiler and debugger operate on real files, paths, and executables.

- Know file and folder basics, paths, extensions, and simple terminal navigation.
- Distinguish source files, object files, and executables.

- Understand relative and absolute paths.

Connections: .c files, headers, compiler output, command-line tools.

PS

Problem Solving

Decomposition · steps · cases

Programming begins with expressing a solution as a finite sequence of clear steps.

TECHNICAL SIGNIFICANCE

Before syntax, you should be able to break a task into smaller operations.

- Identify inputs and outputs.
- Break a problem into subproblems.
- Consider normal and exceptional cases.

Connections: functions, control flow, algorithms, debugging.

BM

Basic Mathematics

Arithmetic · precedence · expressions

C programs frequently manipulate numeric expressions.

TECHNICAL SIGNIFICANCE

Operator precedence and integer behavior matter in low-level code.

- Be comfortable with arithmetic and remainder.
- Understand comparison operators.
- Read expressions using precedence and parentheses.

Connections: + - * / %, conditions, indexing.

BL

Boolean Logic

AND · OR · NOT

Conditions and loops depend on Boolean reasoning.

TECHNICAL SIGNIFICANCE

C uses integer-valued conditions and logical operators to control execution.

- Understand AND, OR, and NOT.
- Read simple truth tables.
- Build compound conditions with comparisons.

Connections: if, while, for, &&, ||, !.

BH

Binary & Hexadecimal

Base-2 · base-16 · bit patterns

C often exposes machine-level data representations.

TECHNICAL SIGNIFICANCE

Binary and hexadecimal make bitwise operations and addresses easier to understand.

- Convert small values between decimal, binary, and hexadecimal.
- Recognize powers of two.
- Understand that a byte is composed of bits.

Connections: masks, addresses, integer representation, debugging.

VT

Variables & Data Types

int · char · float · double

C requires explicit declarations and a clear understanding of representation.

TECHNICAL SIGNIFICANCE

Type choice affects range, precision, storage, and valid operations.

- Understand declarations and initialization.
- Know basic integer and floating-point types.
- Recognize signed and unsigned values.

Connections: variables, expressions, memory layout.

CF

Control Flow

if · switch · loops

Procedural programming organizes execution through sequence, selection, and iteration.

TECHNICAL SIGNIFICANCE

Most C programs are built from combinations of these structures.

- Trace if/else branches.
- Understand switch cases.
- Trace for, while, and do-while loops.

Connections: decision making, iteration, state changes.

FN

Functions

Parameters · return values · scope

Functions divide programs into reusable units.

TECHNICAL SIGNIFICANCE

Good function design reduces duplication and localizes state.

- Understand declarations, definitions, and calls.
- Know return values and parameters.
- Distinguish local and global scope.

AR

Arrays

Indexed storage · contiguous elements

Arrays store fixed-size sequences of same-type elements.

Connections: modularity, recursion, libraries, callbacks.

TECHNICAL SIGNIFICANCE

Array indexing is directly connected to memory addresses and pointer arithmetic.

- Know zero-based indexing.
- Traverse arrays with loops.
- Understand bounds and multidimensional arrays.

Connections: buffers, strings, pointer arithmetic.

ST

Strings

char arrays · null terminator

C strings are arrays of characters terminated by a zero byte.

TECHNICAL SIGNIFICANCE

String handling exposes buffer sizes and memory-safety concerns.

- Understand the null terminator \0.
- Know that string capacity differs from string length.
- Recognize common library operations such as strlen and strcmp.

Connections: char arrays, buffers, library functions.

PT

Pointers

addresses · dereferencing · indirection

Pointers are central to idiomatic C and connect values to memory locations.

TECHNICAL SIGNIFICANCE

Pointers support dynamic memory, arrays, structures, callbacks, and low-level interfaces.

- Understand address-of (&) and dereference (*).
- Distinguish a pointer from the value it points to.
- Recognize NULL and invalid-pointer risks.

Connections: arrays, dynamic allocation, linked structures, function pointers.

MM

Memory Model

stack · heap · lifetime

C gives programmers substantial control over object lifetime and memory use.

TECHNICAL SIGNIFICANCE

Understanding storage duration helps prevent dangling pointers, leaks, and use-after-free errors.

- Distinguish automatic and dynamic storage.
- Understand object lifetime.
- Know that memory safety is largely the programmer's responsibility.

Connections: local variables, malloc/free, recursion, pointers.

DM

Dynamic Memory

malloc · calloc · realloc · free

Many C programs allocate memory while they run.

TECHNICAL SIGNIFICANCE

Dynamic allocation enables variable-size data structures but introduces explicit lifetime management.

- Understand the purpose of malloc and free.
- Check allocation results.
- Avoid leaks, double-free, and use-after-free.

SR

Structures

struct · fields · records

Structures group related values into a single data type.

Connections: dynamic arrays, linked lists, trees, buffers.

TECHNICAL SIGNIFICANCE

They are the basis of many user-defined records and data structures.

- Declare and initialize structs.
- Access members with . and ->
- Pass structures or structure pointers to functions.

EN

Enums & Typedef

enum · typedef · readable APIs

Named types and enumerations improve clarity in procedural programs.

Connections: records, linked lists, APIs, data modeling.

TECHNICAL SIGNIFICANCE

They help replace unexplained numeric constants with meaningful symbolic names.

- Understand enum constants.
- Know what typedef does and does not do.
- Use type aliases without hiding important pointer semantics.

FH

File Handling

FILE* · fopen · fread · fprintf

Connections: standard I/O APIs, string functions.

Connections: data structures, APIs, the standard library.

TECHNICAL SIGNIFICANCE

File operations require careful error checking and resource cleanup.

- Understand opening and closing files.
- Distinguish text and binary I/O.
- Check return values for errors.

Connections: persistent data, logs, serialization, command-line tools.

PP

Preprocessor

#include · #define · conditional compilation

The C preprocessor transforms source text before compilation.

TECHNICAL SIGNIFICANCE

Headers, macros, include guards, and conditional compilation are fundamental to multi-file C projects.

- Understand #include and #define.
- Know why include guards exist.
- Recognize risks of complex macros.

Connections: headers, compilation units, portability, configuration.

CP

Compilation Process

Preprocess · compile · assemble · link

C development uses a multi-stage build process.

TECHNICAL SIGNIFICANCE

Understanding the stages makes compiler and linker errors easier to diagnose.

- Distinguish compile-time from link-time errors.
- Know the role of object files.
- Understand declarations versus definitions.

Connections: gcc/clang, object files, libraries, linker symbols.

DB

Debugging

Warnings · debugger · sanitizers

Correct C programming requires systematic debugging and aggressive use of diagnostics.

TECHNICAL SIGNIFICANCE

Compiler warnings and runtime tools can reveal undefined behavior and memory errors.

- Compile with warnings enabled.
- Learn basic breakpoint and stack-trace concepts.
- Use sanitizers where available.

Connections: -Wall, gdb/lldb, AddressSanitizer, UndefinedBehaviorSanitizer.

AL Algorithmic Thinking

Searching · sorting · complexity

C exercises often implement classic algorithms directly.

TECHNICAL SIGNIFICANCE

Algorithmic thinking helps separate correctness from efficiency.

- Trace linear and binary search.
- Understand basic sorting ideas.
- Recognize common O(1), O(n), and O(n²) behavior.

Connections: arrays, functions, data structures, performance.

DS Basic Data Structures

Lists · stacks · queues

Pointers and structures become more meaningful when used to build dynamic data structures.

TECHNICAL SIGNIFICANCE

Simple data structures connect memory management with algorithm design.

- Understand nodes and links conceptually.
- Recognize stack and queue behavior.
- Relate data-structure operations to time and memory costs.

Connections: linked lists, stacks, queues, trees.

UB Undefined Behavior Awareness

Bounds · initialization · lifetime

C deliberately leaves some erroneous operations without defined semantics.

TECHNICAL SIGNIFICANCE

Recognizing undefined behavior is essential for writing reliable and portable C.

- Never read uninitialized values.
- Do not access outside array bounds.
- Do not use an object after its lifetime ends.

Connections: memory safety, optimization, portability, debugging.

APPENDIX B

Important People in C & Procedural Programming

PROGRAMMING WITH C · COMPUTING HISTORY

C grew from earlier systems languages and became one of the most influential procedural programming languages in computing, shaping operating systems, compilers, embedded software, libraries, and later languages.

From compiler pioneers to modern C toolchains

These profiles highlight contributors to C, UNIX, structured programming, compiler technology, standardization, systems programming, and programming practice.

PEOPLE · LANGUAGES · SYSTEMS

Important Contributors

Key figures behind C, BCPL/B, structured programming, compiler toolchains, the standard library, UNIX tools, and portable systems programming.

DR

Dennis Ritchie

C language · UNIX · Bell Labs

Created the C programming language and co-developed UNIX.

TECHNICAL SIGNIFICANCE

C combined low-level control with portable structured programming, making it ideal for systems software.

C **UNIX** **systems programming**

Ritchie designed C from earlier languages such as B and BCPL and used it to help rewrite UNIX. C's pointer model, arrays, structs, bitwise operators, and compact runtime model made it practical for operating systems and embedded software.

Technical concepts: C, UNIX, systems programming, pointers, arrays, structs, compilation, portability.

BK

Brian Kernighan

C ecosystem · UNIX tools

Co-authored The C Programming Language and contributed extensively to UNIX software tools.

TECHNICAL SIGNIFICANCE

Helped define the style, terminology, and practical programming culture associated with C.

K&R **UNIX** **AWK**

Kernighan co-authored the canonical early C text with Ritchie and co-created AWK. His work popularized concise interfaces, stream processing, command-line composition, and disciplined C programming.

KT

Ken Thompson

UNIX · B language

Co-created UNIX and developed B, a direct predecessor of C.

Technical concepts: K&R, UNIX, AWK, standard I/O, text processing, portability.

TECHNICAL SIGNIFICANCE

UNIX provided the environment in which C evolved into a systems-programming language.

UNIX **B** **systems programming**

Thompson's B simplified BCPL ideas for early UNIX development; Ritchie later evolved B into C.

MR

Martin Richards

BCPL · 1960s

Technical concepts: UNIX, B, systems programming, compiler bootstrap, shell tools, file descriptors.

Created BCPL, a major ancestor of B and C.

TECHNICAL SIGNIFICANCE

BCPL influenced the syntax and low-level systems-programming style inherited by C.

BCPL **language history** **compilation**

BCPL used a compact, word-oriented model suitable for compiler and systems work.

Technical concepts: BCPL, language history, compilation, portability, procedural syntax.

CS

Christopher Strachey

CPL · language theory

Contributed to CPL and foundational programming-language concepts.

TECHNICAL SIGNIFICANCE

CPL influenced BCPL, which influenced B and then C.

CPL **language design** **semantics**

Strachey's work on programming-language semantics influenced later procedural language design.

NW

Niklaus Wirth

Pascal · structured programming

Technical concepts: CPL, language design, semantics, control structures, language lineage.

Designed Pascal and several influential procedural languages.

TECHNICAL SIGNIFICANCE

Helped establish disciplined procedural programming, strong typing, and readable control structures.

Pascal **structured programming** **procedures**

Wirth's languages emphasize sequence, selection, iteration, procedures, records, and simple modular design.

ED

Edsger W. Dijkstra

Structured programming · correctness

Technical concepts: Pascal, structured programming, procedures, records, type systems, modularity.

Advocated structured programming and rigorous reasoning about program correctness.

TECHNICAL SIGNIFICANCE

His ideas shaped disciplined use of sequence, selection, iteration, and decomposition.

structured programming **correctness** **loop invariants**

Dijkstra promoted program structures that are easier to reason about than uncontrolled jumps.

CH

C. A. R. Hoare

Hoare logic · Quicksort

Technical concepts: Hoare logic, Quicksort, invariants, recursion, preconditions, postconditions.

Technical concepts: Hoare logic, Quicksort, invariants, recursion, preconditions, postconditions.

TECHNICAL SIGNIFICANCE

Connected imperative programs with preconditions, postconditions, and invariants.

Hoare logic **Quicksort** **invariants**

Hoare triples express program correctness conditions and directly support reasoning about loops and procedures.

JB

John Backus

FORTRAN · compiler design

Led the development of FORTRAN, one of the first successful high-level compiled languages.

TECHNICAL SIGNIFICANCE

Showed that procedural languages could compile to efficient machine code.

FORTRAN **compilers** **optimization**

FORTRAN compilers translated arithmetic, loops, and control flow into machine instructions while performing early optimization.

GH

Grace Hopper

Compilers · COBOL

Pioneered compiler technology and high-level programming languages.

TECHNICAL SIGNIFICANCE

Helped move programming from machine notation toward symbolic procedural languages.

compilers **COBOL** **compilation pipeline**

SJ

Steve Johnson

Portable C Compiler · yacc

Developed the Portable C Compiler and created yacc.

TECHNICAL SIGNIFICANCE

Compiler portability helped spread C across many hardware platforms.

Early compiler systems separated translation from higher-level notation to executable code.

PCC **yacc** **parsing**

Technical concepts: compilers, COBOL, compilation pipeline, symbolic programming, portability.

PCC separated machine-independent compiler logic from machine-specific code generation; yacc generated parsers from grammars.

Technical concepts: PCC, yacc, parsing, intermediate representation, code generation, portability.

PP

Peter J. Plauger

C standard library · ANSI C

Contributed to C standardization and influential library implementations.

TECHNICAL SIGNIFICANCE

The standard library defines portable interfaces for I/O, strings, memory, conversion, and utilities.

ANSI C **standard library** **stdio**

A C environment includes both compiler behavior and a standardized set of headers and library functions.

BS

Bjarne Stroustrup

C++ · systems programming

Technical concepts: ANSI C, standard library, stdio, string APIs, headers, portability.

Designed C++ as an extension of C with stronger abstraction mechanisms.

TECHNICAL SIGNIFICANCE

Shows which low-level procedural features form the core inherited from C.

C++ **systems programming** **abstraction**

C++ retained C's compilation and memory model while adding classes, constructors, templates, and stronger abstraction.

Technical concepts: C++, systems programming, abstraction, C compatibility, object layout.

RS

Richard Stallman

GCC · GNU toolchain

Founded GNU and initiated GCC.

TECHNICAL SIGNIFICANCE

GCC became one of the most important portable C compiler toolchains.

GCC **GNU** **compiler**

The toolchain performs preprocessing, compilation, assembly, and linking, with optimization and architecture-specific code generation.

CL

Chris Lattner

LLVM · Clang

Technical concepts: GCC, GNU, compiler, object files, linker, optimization.
Created LLVM, a modern compiler infrastructure used by Clang.

TECHNICAL SIGNIFICANCE

LLVM made compiler optimization and code generation highly modular.

LLVM **Clang** **compilers**

Clang translates C into LLVM IR, optimization passes transform it, and backends generate machine code.

DM

Doug McIlroy

UNIX pipes · software tools

Technical concepts: LLVM, Clang, compilers, IR, frontend/backend separation, optimization.
Introduced UNIX pipes and strongly influenced software-tools design.

TECHNICAL SIGNIFICANCE

C programs often become more useful when designed as small composable tools.

pipes **UNIX** **tools**

RP

Rob Pike

UNIX programming · Plan 9

Contributed to UNIX and Plan 9 and wrote extensively about systems-programming style.

Pipes connect standard output from one program to standard input of another, encouraging small reusable filters.

TECHNICAL SIGNIFICANCE

Emphasizes simple interfaces, clear code, and understanding the machine model.

Technical concepts: pipes, UNIX, tools, stdin/stdout, filters, process composition.

UNIX **Plan 9** **systems programming**

Procedural C benefits from explicit state, small functions, simple data structures, and carefully designed interfaces.

Technical concepts: UNIX, Plan 9, systems programming, interfaces, simplicity, portability.

MB

Michael Barr

Embedded C · defensive programming

Known for practical guidance on embedded C and low-level programming.

TECHNICAL SIGNIFICANCE

Embedded C makes memory errors, volatile state, and bit-level interfaces especially visible.

embedded C **volatile** **fixed-width integers**

His work emphasizes defensive coding practices for embedded systems and hardware-oriented C.

Technical concepts: embedded C, volatile, fixed-width integers, register access, masks, interrupts.