

LEARN TO CODE IN C

C Programming from Zero

Learn to Code in C Step by Step — From Your First Program to Pointers, Memory, and Real Projects

No experience needed · Compile, debug, and build one real command-line program from start to finish

POINTERS · STRUCTS · MEMORY · FILES · MAKE

THENEXBOOKS

C Programming from Zero

Learn to Code in C Step by Step — From Your First Program to Pointers, Memory, and Real Projects

First Edition

THENEXBOOKS

C Programming from Zero
Learn to Code in C Step by Step — From Your First Program to Pointers, Memory, and Real Projects

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. The code in this book may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Compilers, tools, and operating systems change over time; consult their official documentation for the versions you use.

This book is an independent publication. GCC, Clang, LLVM, Microsoft Visual Studio, Visual Studio Code, Windows, macOS, Linux, MSYS2, and other product names are trademarks of their respective owners; their use here is for identification only. The Fernbrook Tool Library, the town of Fernbrook, and all people in the examples are fictional.

Preface

C is one of the most influential programming languages ever created. It sits underneath much of the software you use every day: operating systems, databases, programming language runtimes, network tools, and the tiny computers inside appliances, cars, and medical devices. Learning C teaches you how a computer really works — how values live in memory, what a pointer is, why some bugs crash a program and others silently corrupt data — and that understanding makes you a better programmer in every other language too.

C also has a reputation for being hard. Part of that reputation is earned: C trusts you, and it will let you make mistakes that other languages catch for you. But most of the difficulty comes from the way C is usually taught — quickly, assuming you already program, and with the dangerous parts left as a surprise. This book takes the opposite approach. It starts from zero, explains every idea with a small complete program you can compile and run, and shows the safe habit right next to every sharp edge.

Instead of disconnected exercises, we build one real program across the whole book: **toolshed**, a command-line application for a community tool-lending library. It grows from a single `printf` into a multi-file project with structs, dynamic memory, CSV files, a waiting list, date calculations, late fees, tests, a Makefile, and a proper command-line interface. By the end you will have written a program you understand line by line, and you will be ready to read and write real C code.

Who This Book Is For

- **Complete beginners** who have never programmed before — no experience is assumed.
- **Students** taking a first course in C who want a patient, practical companion to their lectures.
- **Programmers from other languages** such as Python or JavaScript who want to understand memory, pointers, and how programs really run.
- **Makers and future embedded developers** who want a solid foundation before moving on to microcontrollers and systems programming.

What You Will Learn

- How to install a C compiler, use the terminal, and compile and run programs on Windows, macOS, or Linux.
- Variables, types, arithmetic, input and output, decisions, loops, and functions.
- Arrays, strings, and pointers — explained gently, with pictures in words and plenty of practice.
- Structs, dynamic memory with `malloc` and `free`, files, and linked lists.
- How to organize real programs into modules with header files, build them with Make, and test them.
- How to debug, handle errors, and avoid undefined behavior so your programs are safe and robust.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) sets up your tools and writes your first programs. **Part II — Control Flow and Functions** (Chapters 6–10) teaches decisions, loops, functions, scope, and debugging. **Part III — Arrays, Strings, and Pointers** (Chapters 11–15) covers the heart of C and ends with a complete Tool Inventory project. **Part IV — Building Data** (Chapters 16–19) introduces structs, dynamic memory, files, and linked lists. **Part V — Organizing Real Programs** (Chapters 20–24) splits the program into modules and adds Make, integer details, error handling, and tests. **Part VI — Projects and Next Steps** (Chapters 25–28) completes the loan tracker and the command-line tool, collects the rules for writing safe C, and points the way forward. The appendices provide a cheat sheet, a standard library reference, checklists, and a glossary.

Every chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box.

Welcome to the Fernbrook Tool Library

The Fernbrook Tool Library is a small (and fictional) community project in the town of Fernbrook. Instead of everyone buying a drill they use twice a year, members borrow tools for free for seven days. Rosa Delgado coordinates the library, Ben Okafor repairs tools and checks their condition, and Priya Nair runs the front desk during opening hours on Wednesday evenings and Saturday mornings. Right now they track everything in a paper notebook. Over the course of this book, you will write them a real program to do it.

How to Use This Book

Type the programs yourself instead of copying them. Compile often, read every warning, and run each program to see what it does. When the compiler complains — and it will, for everyone — read the message carefully, compare your code with the book, and remember that fixing errors is a normal part of programming. Chapter 10 is devoted to debugging, and Appendix C contains checklists for when you get stuck.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a shortcut that makes your work easier.

> **NOTE**
>
> Background information, platform differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause bugs or crashes.

Code appears in `monospace`. Program listings appear in dark boxes with the file name above them, such as `src/date.c`. Commands you type in the terminal and the output they produce appear in lighter boxes labeled **Terminal**; lines starting with `$` are commands you type. On Windows, run programs as `hello.exe` or `.\hello.exe` instead of `./hello`.

Contents

Preface

PART I — GETTING STARTED

1 What Is C, and Why Learn It?

2 Setting Up: Compiler, Editor, and Terminal

3 Your First Program: Anatomy of hello.c

4 Variables, Types, and Arithmetic

5 Input and Output with printf and scanf

PART II — CONTROL FLOW AND FUNCTIONS

6 Making Decisions: if, else, and switch

7 Loops: while, for, and do-while

8 Functions: Breaking Programs into Pieces

9 Scope, Lifetime, and the Preprocessor

10 Debugging Your Programs

PART III — ARRAYS, STRINGS, AND POINTERS

11 Arrays

12 Strings and Characters

13 Pointers Explained Gently

14 Pointers and Arrays, Together

15 Project: The Tool Inventory

PART IV — BUILDING DATA

16 Structs: Grouping Related Data

17 Dynamic Memory: malloc, free, and Friends

18 Working with Files

19 Linked Lists and Other Dynamic Structures

PART V — ORGANIZING REAL PROGRAMS

20 Multi-File Programs and Header Files

21 Building with Make

22 Bits, Bytes, and Integer Details

23 Error Handling and Defensive Programming

24 Testing C Code

PART VI — PROJECTS AND NEXT STEPS

25 Project: The Loan Tracker

26 Project: A Command-Line Tool with Arguments

27 Writing Safe and Robust C

28 Next Steps: Where C Can Take You

APPENDICES

A C Cheat Sheet

B Standard Library Quick Reference

C Checklists

D Glossary

PART I

Getting Started

Install a compiler, learn the terminal, and write your first C programs with variables, arithmetic, input, and output.

1 · What Is C, and Why Learn It?
2 · Setting Up: Compiler, Editor, and Terminal
3 · Your First Program: Anatomy of hello.c
4 · Variables, Types, and Arithmetic
5 · Input and Output with printf and scanf

CHAPTER 1 · PART I — GETTING STARTED

What Is C, and Why Learn It?

Before you write a single line of code, it helps to know what a program really is, how C turns your words into something a computer can run, and where this book is going. In this chapter we meet the language, the tools, and the small community tool library whose software you will build from scratch.

Welcome

If you have never written a program before, you are in exactly the right place. This book assumes nothing. We will not use a word like "variable", "function", or "pointer" without first explaining what it means and showing it in a tiny, complete program that you can type, compile, and run yourself.

If you already know a little Python or JavaScript, you are welcome too. Some ideas will feel familiar, and you will notice places where C asks you to be more explicit. We will point out those differences as we go, but we will never assume you know another language.

C has a reputation. People call it "low level", "hard", or "dangerous". There is some truth in all three words, and we will not hide it. C trusts the programmer. It lets you do almost anything the machine can do, including things that are mistakes. But that same trust is what makes C fast, small, and everywhere. Our approach is simple: we explain each sharp edge calmly, show you exactly what goes wrong, and then show you the safe habit that avoids it. By the end of the book those habits will feel natural.

Learning to program takes practice, not talent. Every programmer, no matter how experienced, sees error messages every single day. When the compiler complains, it is not judging you; it is helping you. One of the most valuable skills this book teaches is reading those messages calmly and fixing problems one at a time.

What Is a Program?

A **program** is a list of instructions that a computer follows, one after another. A recipe is a good comparison. A recipe says: preheat the oven, mix the flour and sugar, add two eggs, bake for thirty minutes. Each step is small and precise. The cook does not have to invent anything; they just follow the steps in order, sometimes repeating a step ("stir until smooth") or choosing between steps ("if the dough is sticky, add more flour").

Programs work the same way. They read some input, do some work, and produce some output. A calculator program reads two numbers and prints their sum. A music player reads a file and sends sound to your speakers. A web browser reads text from a server and draws a page on your screen. Under all the colors and buttons, each of these is a long, carefully organized list of small instructions.

The difference between a recipe and a program is the reader. A human cook can handle vague instructions like "a pinch of salt". A computer cannot. The processor inside your computer, called the **CPU** (central processing unit), understands only very simple, very precise commands: copy this number from here to there, add these two numbers, compare this number with zero, jump to a different instruction. These commands are encoded as numbers, and that numeric language is called **machine code**.

Nobody wants to write machine code by hand. It is tedious, it is easy to get wrong, and machine code for one kind of processor does not run on another kind. So people invented **programming languages**: languages that humans can read and write comfortably, plus tools that translate them into machine code. C is one of those languages, and it is one of the most influential ever created.

Source code and executables

The text you write in a programming language is called **source code**. A C source file is an ordinary text file whose name ends in `.c`, such as `hello.c`. You can open it in any text editor; there is nothing magic about it.

The machine-code file that the computer can actually run is called an **executable** (or a "binary"). On Windows, executables usually end in `.exe`. On macOS and Linux they typically have no extension at all; they are simply marked as runnable.

Getting from source code to an executable is the job of a **compiler**.

Compiled and Interpreted Languages

There are two broad ways to run a program written in a human-friendly language.

In a **compiled** language such as C, you hand your whole source file to a compiler. The compiler reads all of it, checks it for mistakes, and translates it into a machine-code executable. Then you run the executable. The translation happens once, ahead of time. After that, the executable runs directly on the processor with no translator in between, which is one reason compiled C programs are fast and start instantly.

In an **interpreted** language, the classic picture is different. You hand your source file to a program called an **interpreter**, which reads your instructions and carries them out as it goes. There is no separate executable to produce first. Python is usually described this way, and JavaScript in a web browser feels the same from the outside. In reality, modern language tools mix these approaches in clever ways, so treat "compiled" and "interpreted" as a useful simplification rather than a strict law.

Here is how the two workflows compare for a beginner:

Step	C (compiled)	Python (interpreted)
Write code	`hello.c` in an editor	`hello.py` in an editor
Translate	Run the compiler	No separate step
Run	Run the executable	Run the interpreter on the file
Errors found	Many before the program runs	Most while it runs
To share	Give someone the executable	They need Python installed

That "errors found" row matters a lot to you as a learner. Because the C compiler reads your whole program before anything runs, it catches many mistakes early: a missing semicolon, a misspelled name, a number used where text was expected. You will see a lot of compiler messages in the first few chapters. That is a feature, not a punishment.

The other side of that coin is that the compiler cannot catch everything. Some mistakes only show up when the program runs, and in C a few of them can make the program behave in strange, unpredictable ways. We will meet those carefully, starting in Chapter 4 and in detail in Chapter 27.

> **NOTE**
>
> In everyday conversation people say "the compiler" for the whole tool that turns `hello.c` into an executable. Strictly speaking that tool runs several stages: a preprocessor, the compiler proper, an assembler, and a linker. You do not need those details yet. Chapter 3 walks through each stage with a real example.

A Very Short History of C

C was created by Dennis Ritchie at Bell Labs in the early 1970s. It grew up alongside the Unix operating system: much of Unix was rewritten in C, which was unusual at the time, because operating systems were normally written in the machine-specific assembly language of each computer. Writing an operating system in a higher-level language meant it could be moved to new kinds of hardware far more easily. That idea, **portability**, is still one of C's great strengths.

As C spread, different compilers began to accept slightly different versions of the language. To fix that, C was standardized. The first official standard is usually called **ANSI C** or **C89**; the international version from 1990 is called **C90**. Later revisions followed: **C99**, **C11**, **C17**, and **C23**. Each one added features and fixed problems while keeping old programs working as much as possible.

This book uses **C17**. It is mature, it is supported by every mainstream compiler, and the code you write with it will build almost anywhere. C23 is the newest standard, supported by recent compilers; it adds conveniences such as `bool` and `true` as built-in keywords and a `nullptr` constant. Where C23 changes something you are learning, we will mention it in a short note, and Chapter 28 looks at it more closely. Every program in this book, though, is written and tested as C17.

Where C Is Used Today

C is not a museum piece. It quietly runs a remarkable amount of the world's software, often in places you never see.

- **Operating systems.** The core of Linux, called the kernel, is written mainly in C. Large parts of other operating systems use C as well. When software needs to talk directly to hardware, manage memory, and run on many kinds of processors, C is a natural fit.
- **Embedded systems.** Microwave ovens, washing machines, car engine controllers, thermostats, fitness trackers, and countless other devices contain small computers called **microcontrollers**. Many of these have very little memory and no operating system at all. C is one of the most common languages for programming them, because it produces small, efficient code and gives precise control over the hardware.
- **Databases and servers.** Several widely used database engines and web servers are written in C, because they need to be fast and to use memory carefully.
- **Programming languages themselves.** The standard Python interpreter is written in C. So are many other language tools. When you learn C, you are learning the language that a lot of other languages are built on.
- **Command-line tools and libraries.** Many of the small utilities that developers use every day, and many libraries for compression, graphics, networking, and cryptography, are written in C so that programs in other languages can call them.

Why Learn C?

You might wonder why you should start with C when friendlier languages exist. Here are honest reasons, without hype.

You learn how computers really work

In C, you can see where your data lives. You will learn that every piece of data occupies some bytes of memory, that those bytes have addresses, and that a program can work with those addresses directly. These ideas exist in every language, but most languages hide them. Once you understand them in C, many mysteries in other languages suddenly make sense: why some operations are slow, why some bugs happen, why memory "leaks".

C is small

The C language itself has a modest number of keywords and rules. You can realistically learn the whole core language, which is not true of every language. The challenge in C is not memorizing a huge list of features; it is using a small set of features carefully.

C is fast and lightweight

Because C compiles directly to machine code and adds very little behind your back, well-written C programs tend to run quickly and use little memory. That matters on tiny devices and in software that must handle a lot of work.

C opens doors

If you are interested in embedded systems, robotics, operating systems, game engines, or performance-sensitive software, C is a practical foundation. Even if you mostly work in other languages later, knowing C makes you a more confident programmer. Chapter 28 discusses some of those paths.

The honest downside

C will not hold your hand. If you ask it to store eleven characters in a space for ten, it may simply do so and overwrite whatever was next door in memory. If you forget to give a variable a starting value, it may contain leftover garbage. The language standard calls many of these situations **undefined behavior**: the rules say nothing about what happens, so anything can. Your program might crash, print nonsense, or seem to work perfectly until one day it does not.

That sounds scary, but it is manageable. Professional C programmers rely on habits and tools: compiler warnings turned all the way up, careful checks on sizes, testing, and special run-time checkers called sanitizers. You will learn all of these. From the very first compile in Chapter 2, we will use warning flags so the compiler can point out suspicious code.

> **TIP**
>
> Type the examples yourself instead of copying and pasting them. It feels slower, but your fingers and eyes learn the shape of C much faster that way, and the small typing mistakes you make will teach you how to read compiler messages. When something does not work, compare your version with the book's line by line.

Meet Fernbrook Tool Library

Learning a language from disconnected examples can feel like memorizing a phrasebook. To give the examples a purpose, this book follows one small, realistic organization from start to finish.

Fernbrook Tool Library is a community tool-lending library in the town of Fernbrook. It works like a book library, but instead of novels the shelves hold drills, saws, garden tools, and cleaning machines. Most people need a pressure washer or a tile cutter only once or twice a year, so it makes sense for a neighborhood to share them. Members borrow tools for free and bring them back for the next person.

The people

The library is run by volunteers, and you will meet them in examples throughout the book.

- **Rosa Delgado** is the coordinator. She keeps the inventory, sets the rules, and wants reliable reports: which tools are out, which are overdue, and how much is owed in late fees.
- **Ben Okafor** is the repair volunteer. He checks the condition of each tool when it comes back, sharpens blades, replaces batteries, and decides when something needs repair.
- **Priya Nair** works the front desk. She checks tools out to members and back in again, so she needs software that is quick and hard to misuse.

Opening hours

The library is open on **Wednesdays from 5 PM to 8 PM** and on **Saturdays from 9 AM to 1 PM**. Those are the only times tools can be borrowed or returned, which will matter later when we work with dates.

The rules

Rosa keeps the rules short so that everyone remembers them:

- The **loan period is 7 days**. A tool borrowed on Saturday is due back the following Saturday.
- A member may borrow **at most 3 tools at once**.
- Late returns cost **$1.00 per day late**, with a **maximum fee of $10.00** for any one loan. In the program we will store these amounts as whole numbers of cents: 100 cents per day, capped at 1000 cents. Chapter 4 explains why programmers avoid fractional numbers for money.

The tools

The library starts with ten tools. Each tool has an id number, a name, a category, and a condition rating from 1 to 5, where 1 means "needs repair", 2 "worn", 3 "good", 4 "very good", and 5 "like new". All ten are on the shelf and available at the start.

Id	Name	Category	Condition
1	Cordless drill	power	4
2	Circular saw	power	5
3	Socket set	hand	3
4	Hedge trimmer	garden	4
5	Pressure washer	cleaning	3
6	Stud finder	hand	5
7	Tile cutter	hand	2
8	Wheelbarrow	garden	4

9	Carpet cleaner	cleaning	4
10	Jigsaw	power	3

There are exactly four categories: `power`, `hand`, `garden`, and `cleaning`. Keeping categories in a fixed, lowercase form will make searching and comparing them much easier when we work with strings in Chapter 12.

The members

The library has many members, but five of them appear again and again in our examples: **Sam Whitaker** (member 101), **Lena Fischer** (102), **Marcus Bell** (103), **Aiko Sato** (104), and **Diego Ramos** (105). Giving members id numbers, like tools, lets the program refer to them precisely even if two people share a name.

The Program You Will Build: toolshed

Right now, Rosa tracks everything in a paper notebook. Priya writes down who took what, Ben scribbles condition notes in the margins, and at the end of the month Rosa spends an evening working out who is overdue. It works, mostly, but it is slow and easy to get wrong.

Over the course of this book you will write `toolshed`, a **console program** (also called a terminal or command-line program) that manages the library's tools, members, and loans. A console program has no windows or buttons; it reads what you type and prints text back. That may sound old-fashioned, but it is the best way to learn: there is nothing between you and the logic of your program, and console programs are exactly what C is used for in many real jobs.

The program grows step by step, and each step uses only what you have learned so far:

- In the early chapters, `toolshed` is just a few lines that print the library's name, compute a late fee, or describe one tool using plain variables.
- With decisions and loops (Chapters 6 and 7), it gains a simple text menu.
- With functions (Chapter 8), we write small, reusable pieces such as `late_fee_cents`, which works out the fee for a given number of days late.
- With arrays, strings, and pointers (Part III), it can hold the whole inventory, search tools by name, and check tools in and out. Chapter 15 brings these together into a first complete version with a menu offering list, search, check out, return, and report.
- With structs, dynamic memory, and files (Part IV), it stores tools, members, and loans properly, grows as the library grows, and saves everything to CSV files so nothing is lost when the program closes. It even gains a waiting list so members can reserve a popular tool.
- In Part V you split the program into several files, build it with Make, and add tests.

- In Part VI you finish the loan tracker with real dates and late fees, and turn `toolshed` into a proper command-line tool that understands commands such as `toolshed list` and `toolshed overdue --date 2026-03-14`.

By the end, you will have written a real, multi-file C program of a few thousand lines, and you will understand every one of them.

A First Look at C

You do not have a compiler yet (Chapter 2 fixes that), but it is worth seeing what C looks like. Here is a tiny program that prints a welcome message for the library:

`welcome.c`

```
#include <stdio.h>

int main(void)
{
    printf("Welcome to Fernbrook Tool Library!\n");
    printf("Open Wednesdays 5 PM-8 PM and Saturdays 9 AM-1 PM.\n");
    printf("Borrow up to 3 tools for 7 days.\n");
    return 0;
}
```

When this program is compiled and run, it prints three lines:

`Terminal`

```
$ gcc -std=c17 -Wall -Wextra -pedantic -g welcome.c -o welcome
$ ./welcome
Welcome to Fernbrook Tool Library!
Open Wednesdays 5 PM-8 PM and Saturdays 9 AM-1 PM.
Borrow up to 3 tools for 7 days.
```

Even without any explanation you can probably guess a lot. The lines with `printf` print text. The text to print is inside double quotes. The `\n` at the end of each message means "start a new line". The curly braces group the lines that belong together, and each instruction ends with a semicolon.

Some parts are less obvious: what is `#include <stdio.h>`? Why `int main(void)`? What does `return 0` return, and to whom? Chapter 3 answers every one of those questions line by line. For now, just notice that a complete C program can be very short, and that it reads almost like a list of instructions in English.

Here is the same idea in Python, for readers who have seen it:

`welcome.py (Python, for comparison only)`

```
print("Welcome to Fernbrook Tool Library!")
```

Python needs less ceremony. C asks you to say more: where `printf` comes from, where the program starts, and what it reports when it finishes. That explicitness is typical of C. It is a little more typing, and in exchange you always know exactly what your program is doing.

How to Read This Book

A few conventions will help you move quickly.

Code listings

Complete programs appear in dark boxes. The small label above each box tells you the file name, such as `welcome.c`. When you see a file name, you can type that program into a file with that exact name and compile it. Short fragments that are not complete programs are labeled clearly in the text.

Terminal sessions

Boxes labeled **Terminal** show commands you type and the text the computer prints back. Lines that start with `$` are commands; the `$` stands for your terminal's prompt and is not something you type. Everything else is output. On Windows your prompt may look like `PS C:\Users\you>` instead, and you run programs with `.\welcome.exe` rather than `./welcome`. Chapter 2 covers these differences.

The program output shown in this book comes from really running each program. When we show compiler error messages, they are typical of GCC and Clang; your compiler's exact wording, line formatting, and column numbers may differ slightly, so we say "similar to" in those places. Learn to recognize the important parts — the file name, the line number, and the main message — rather than memorizing the exact text.

Callout boxes

Three kinds of boxes appear throughout the book. A **Tip** suggests a better or faster way to do something. A **Note** adds background or detail you may want to come back to. A **Warning** marks a real trap: a common bug, a security problem, or behavior that differs between systems. Read the warnings even when you are in a hurry.

Exercises

Every chapter ends with a **Try It Yourself** section of small exercises, followed by **Key Takeaways**. The exercises are short on purpose. Do at least one or two before moving on; changing a working program and predicting what will happen is the fastest way to learn. If an exercise feels hard, reread the relevant section and try a smaller version first.

> **WARNING**
>
> Do not skip the setup in Chapter 2, even if you plan to use an online C compiler at first. Online tools are fine for quick experiments, but they often hide warnings, use different compiler settings, and cannot handle the multi-file projects and data files we build later. Having your own compiler working early saves frustration.

What You Need

You need surprisingly little to follow this book:

- A computer running Windows, macOS, or Linux. It does not need to be new or powerful; C compilers are light.
- A C compiler. We use GCC or Clang, both free. Chapter 2 shows how to install one on each operating system.
- A text editor. We suggest Visual Studio Code (usually called VS Code) with its C/C++ extension, but any plain-text editor works. A word processor does not: it adds formatting that the compiler cannot read.
- A terminal, the text window where you type commands. Every operating system has one built in, and Chapter 2 teaches the handful of commands you need.
- Patience with yourself. You will make mistakes. Every programmer does, every day.

You do not need to be good at mathematics. If you can work out how many days late a returned tool is, you can follow every example.

A Map of the Journey

The book is divided into six parts. Here is what each one gives you:

Part	Chapters	You will learn
I — Getting Started	1–5	Setup, first programs, types, input and output
II — Control Flow and Functions	6–10	Decisions, loops, functions, scope, debugging
III — Arrays, Strings, and Pointers	11–15	Collections, text, memory addresses, a first inventory
IV — Building Data	16–19	Structs, dynamic memory, files, linked lists
V — Organizing Real Programs	20–24	Multiple files, Make, bits, errors, testing
VI — Projects and Next Steps	25–28	Loan tracker, a command-line tool, safety, what next

Parts I and II build the foundation that every C programmer uses daily. Part III contains the ideas C is famous for — pointers and memory addresses — explained slowly, with plenty of pictures in words and small experiments. Part IV is where your programs start handling real amounts of data. Part V teaches

the habits that make larger programs manageable, and Part VI brings everything together.

It is fine to go at your own pace. Some readers move through Part I in a weekend; others spend a week on each chapter. What matters is that you type the programs, run them, break them on purpose, and fix them again.

Try It Yourself

1. In your own words, write two or three sentences explaining the difference between source code and an executable. Then explain what a compiler does to get from one to the other.
2. Look around your home and list three devices that probably contain a small computer (a microcontroller). For each one, guess what its program has to do: what input it reads and what output it produces.
3. Using the rules of Fernbrook Tool Library, work out by hand: if Lena Fischer borrows the hedge trimmer on Saturday 2026-03-07, when is it due? If she returns it 4 days late, what is the fee in dollars and in cents? What if she returns it 15 days late?
4. Look at `welcome.c` again. Without running anything, write down what you think would change in the output if you removed the `\n` from the end of the first message. You will be able to check your answer in Chapter 3.
5. Make a short list of the features you would want in a program for Rosa, Ben, and Priya. Keep it; at the end of the book, compare it with what `toolshed` can do.